

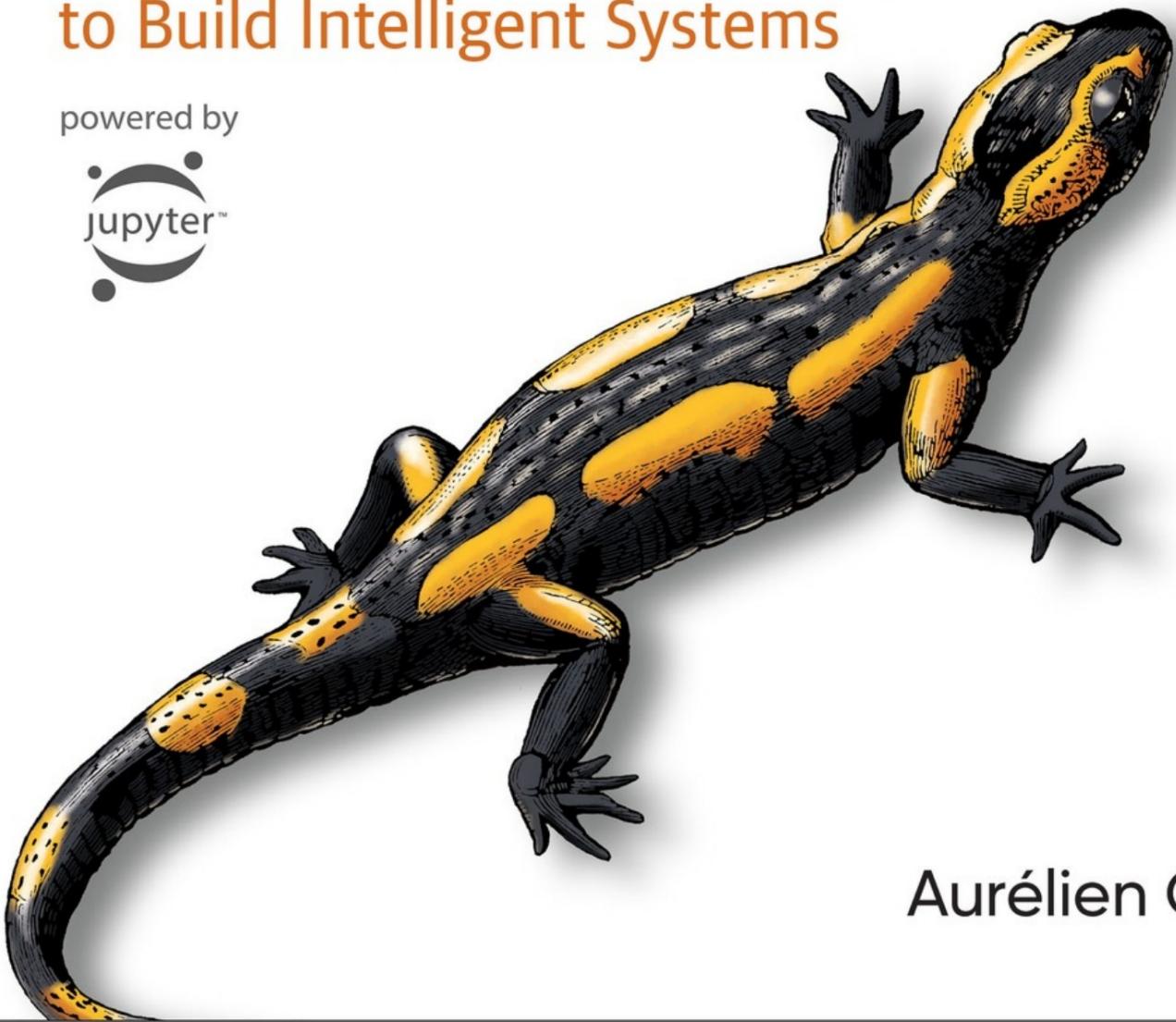
O'REILLY®

Third  
Edition

# Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow

Concepts, Tools, and Techniques  
to Build Intelligent Systems

powered by



Aurélien Géron

# Aprendizaje automático práctico con Scikit-Learn, Keras y TensorFlow

TERCERA EDICION

Conceptos, herramientas y técnicas para construir de forma inteligente  
Sistemas

Aurélien Géron



Beijing • Boston • Farnham • Sebastopol • Tokyo

Aprendizaje automático práctico con Scikit-Learn, Keras y  
TensorFlow

por Aurélien Geron

Copyright © 2023 Aurélien Géron. Reservados todos los derechos.

Impreso en los Estados Unidos de América.

Publicado por O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Los libros de O'Reilly se pueden comprar para uso educativo, comercial o promocional de ventas. También hay ediciones en línea disponibles para la mayoría de los títulos (<https://oreilly.com>). Para obtener más información, comuníquese con nuestro departamento de ventas corporativo/institucional: 800-998-9938 o [corporativo@oreilly.com](mailto:corporativo@oreilly.com).

- Editora de adquisiciones: Nicole Butterfield
- Editores de desarrollo: Nicole Taché y Michele Cronin
- Editor de producción: Beth Kelly
- Editor: Kim Cofer
- Correctora: Rachel Head
- Indexador: Potomac Indexing, LLC
- Diseñador de interiores: David Futato
- Diseñador de portada: Karen Montgomery
- Ilustradora: Kate Dullea
- Marzo de 2017: Primera edición

- Septiembre 2019: Segunda Edición
- Octubre de 2022: Tercera edición

#### Historial de revisiones de la tercera edición

- 2022-10-03: Primer lanzamiento

Ver <https://oreilly.com/catalog/errata.csp?isbn=9781492032649> para detalles de lanzamiento.

El logotipo de O'Reilly es una marca registrada de O'Reilly Media, Inc.

El aprendizaje automático práctico con Scikit-Learn, Keras y TensorFlow, la imagen de portada y la imagen comercial relacionada son marcas comerciales de O'Reilly Media, Inc.

Las opiniones expresadas en este trabajo son las del autor y no representan las opiniones del editor. Si bien el editor y el autor han realizado esfuerzos de buena fe para garantizar que la información y las instrucciones contenidas en este trabajo sean precisas, el editor y el autor renuncian a toda responsabilidad por errores u omisiones, incluida, entre otras, la responsabilidad por los daños resultantes del uso o confianza en este trabajo. El uso de la información y las instrucciones contenidas en este trabajo es bajo su propio riesgo. Si algún ejemplo de código u otra tecnología que este trabajo contiene o describe está sujeto a licencias de código abierto o derechos de propiedad intelectual de otros, es su responsabilidad asegurarse de que su uso cumpla con dichas licencias y/o derechos.

978-1-098-12597-4

[LSI]

# Prefacio

---

## El tsunami del aprendizaje automático

En 2006, Geoffrey Hinton et al. publicó [un artículo](#) mostrando <sup>1</sup> cómo entrenar una red neuronal profunda capaz de reconocer dígitos escritos a mano con una precisión de última generación (>98%). Calificaron esta técnica de "aprendizaje profundo". Una red neuronal profunda es un modelo (muy) simplificado de nuestra corteza cerebral, compuesto por una pila de capas de neuronas artificiales. Entrenar una red neuronal profunda se consideraba ampliamente imposible en ese momento, y la mayoría de los investigadores <sup>2</sup> abandonaron la idea a finales de los años 1990. Este artículo reavivó el interés de la comunidad científica y, en poco tiempo, muchos artículos nuevos demostraron que el aprendizaje profundo no solo era posible, sino que también era capaz de alcanzar logros alucinantes que ninguna otra técnica de aprendizaje automático (ML) podría aspirar a igualar (con la ayuda de tremenda potencia informática y grandes cantidades de datos). Este entusiasmo pronto se extendió a muchas otras áreas del aprendizaje automático.

Una década más tarde, el aprendizaje automático había conquistado la industria y hoy está en el centro de gran parte de la magia de los productos de alta tecnología: clasifica los resultados de búsqueda web, potencia el reconocimiento de voz de tu teléfono inteligente, recomienda videos y tal vez incluso conduce tu auto. .

## Aprendizaje automático en tus proyectos

Entonces, naturalmente, estás entusiasmado con el aprendizaje automático y te encantaría unirte a la fiesta.

¿Quizás le gustaría darle a su robot casero un cerebro propio? ¿Hacer que reconozca caras? ¿O aprender a caminar?

O tal vez su empresa tenga toneladas de datos (registros de usuarios, datos financieros, datos de producción, datos de sensores de máquinas, estadísticas de líneas directas, informes de recursos humanos, etc.), y lo más probable es que pueda descubrir algunas gemas ocultas si supiera dónde buscar. Con el aprendizaje automático, podría lograr lo siguiente **y mucho más**:

- Segmenta a los clientes y encuentra la mejor estrategia de marketing para cada grupo.
- Recomendar productos para cada cliente en función de lo que compraron clientes similares.
- Detecta qué transacciones son susceptibles de ser fraudulentas.
- Pronosticar los ingresos del próximo año.

Cualquiera sea el motivo, has decidido aprender el aprendizaje automático e implementarlo en tus proyectos. ¡Gran idea!

## Objetivo y enfoque

Este libro supone que usted no sabe casi nada sobre el aprendizaje automático. Su objetivo es brindarle los conceptos, herramientas e intuición que necesita para implementar programas capaces de aprender de los datos.

Cubriremos una gran cantidad de técnicas, desde las más simples y utilizadas (como la regresión lineal) hasta algunas de las técnicas de aprendizaje profundo que regularmente ganan competencias. Para esto, usaremos frameworks Python listos para producción:

- **Scikit-Aprende** es muy fácil de usar, pero implementa muchos algoritmos de aprendizaje automático de manera eficiente, por lo que es un excelente punto de entrada al aprendizaje automático. Fue creado por David Cournapeau en 2007 y ahora está dirigido por un equipo de investigadores del Instituto Francés de Investigación en Informática y Automatización (Inria).

- **TensorFlow** es una biblioteca más compleja para cálculo numérico distribuido. Permite entrenar y ejecutar redes neuronales muy grandes de manera eficiente mediante la distribución de los cálculos en potencialmente cientos de servidores multi-GPU (unidad de procesamiento de gráficos). TensorFlow (TF) se creó en Google y admite muchas de sus aplicaciones de aprendizaje automático a gran escala. Fue de código abierto en noviembre de 2015 y la versión 2.0 se lanzó en septiembre de 2019.
- **Keras** es una API de aprendizaje profundo de alto nivel que hace que sea muy sencillo entrenar y ejecutar redes neuronales. Keras viene incluido con TensorFlow y depende de TensorFlow para todos los cálculos intensivos.

El libro favorece un enfoque práctico, aumentando una comprensión intuitiva del aprendizaje automático a través de ejemplos prácticos concretos y solo un poco de teoría.

CONSEJO

Si bien puedes leer este libro sin levantar tu computadora portátil, te recomiendo que experimentes con los ejemplos de código.

## Ejemplos de código

Todos los ejemplos de código de este libro son de código abierto y están disponibles en línea en <https://github.com/ageron/handson-ml3>. como cuadernos Jupyter. Se trata de documentos interactivos que contienen texto, imágenes y fragmentos de código ejecutable (Python en nuestro caso). La forma más fácil y rápida de comenzar es ejecutar estos cuadernos usando Google Colab: este es un servicio gratuito que le permite ejecutar cualquier cuaderno Jupyter directamente en línea, sin tener que instalar nada en su máquina. Todo lo que necesitas es un navegador web y una cuenta de Google.

#### NOTA

En este libro, asumiré que estás usando Google Colab, pero también probé los cuadernos en otras plataformas en línea como Kaggle y Binder, por lo que puedes usarlos si lo prefieres. Alternativamente, puede instalar las bibliotecas y herramientas necesarias (o la imagen de Docker para este libro) y ejecutar los cuadernos directamente en su propia máquina. Consulte las instrucciones en <https://homl.info/install>.

Este libro está aquí para ayudarle a realizar su trabajo. Si desea utilizar contenido adicional más allá de los ejemplos de código, y ese uso queda fuera del alcance de las pautas de uso justo (como vender o distribuir contenido de los libros de O'Reilly o incorporar una cantidad significativa de material de este libro en el contenido de su producto), documentación), comuníquese con nosotros para obtener permiso, en [permisos@oreilly.com](mailto:permisos@oreilly.com).

Apreciamos la atribución, pero no la exigimos. Una atribución suele incluir el título, el autor, la editorial y el ISBN. Por ejemplo: “Aprendizaje automático práctico con Scikit-Learn, Keras y TensorFlow de Aurélien Géron. Copyright 2023 Aurélien Géron, 978-1-098-12597-4.”

## Requisitos previos

Este libro asume que tienes algo de experiencia en programación Python. Si aún no conoce Python, <https://learnpython.org> es un gran lugar para comenzar. El tutorial oficial en [Python.org](https://python.org) también es bastante bueno.

Este libro también supone que está familiarizado con las principales bibliotecas científicas de Python, en particular, **NumPy**, **pandas**, y **Matplotlib**. Si nunca has usado estas bibliotecas, no te preocupes; son fáciles de aprender y he creado un tutorial para cada uno de ellos. Puede acceder a ellos en línea en <https://homl.info/tutorials>.

Además, si desea comprender completamente cómo funcionan los algoritmos de aprendizaje automático (no solo cómo usarlos), debe tener al menos una comprensión básica de algunos conceptos matemáticos, especialmente de álgebra lineal. Específicamente, debes saber qué son los vectores y las matrices, y cómo realizar algunas operaciones simples como sumar vectores o transponer y multiplicar matrices. Si necesita una introducción rápida al álgebra lineal (¡realmente no es ciencia espacial!), le proporciono un tutorial en <https://homl.info/tutorials>. También encontrará un tutorial sobre cálculo diferencial, que puede resultar útil para comprender cómo se entrenan las redes neuronales, pero no es del todo esencial para comprender los conceptos importantes. Este libro también utiliza ocasionalmente otros conceptos matemáticos, como exponentiales y logaritmos, un poco de teoría de probabilidad y algunos conceptos estadísticos básicos, pero nada demasiado avanzado. Si necesita ayuda con alguno de estos, consulte <https://khanacademy.org>, que ofrece muchos cursos de matemáticas excelentes y gratuitos en línea.

## Mapa vial

Este libro está organizado en dos partes. La Parte I, "Los fundamentos del aprendizaje automático", cubre los siguientes temas:

- Qué es el aprendizaje automático, qué problemas intenta resolver y las principales categorías y conceptos fundamentales de sus sistemas
- Los pasos de un proyecto típico de aprendizaje automático
- Aprender ajustando un modelo a los datos
- Optimización de una función de costos
- Manejo, limpieza y preparación de datos.
- Selección y características de ingeniería.
- Seleccionar un modelo y ajustar hiperparámetros mediante validación cruzada

- Los desafíos del aprendizaje automático, en particular el desajuste y el sobreajuste (la compensación entre sesgo y varianza)
- Los algoritmos de aprendizaje más comunes: regresión lineal y polinómica, regresión logística, k vecinos más cercanos, máquinas de vectores de soporte, árboles de decisión, bosques aleatorios y métodos de conjunto.
- Reducir la dimensionalidad de los datos de entrenamiento para luchar contra la "maldición de la dimensionalidad"
- Otras técnicas de aprendizaje no supervisadas, incluida la agrupación, la estimación de densidad y la detección de anomalías

La Parte II, "Redes neuronales y aprendizaje profundo", cubre los siguientes temas:

- Qué son las redes neuronales y para qué sirven
- Construyendo y entrenando redes neuronales usando TensorFlow y Keras
- Las arquitecturas de redes neuronales más importantes: redes neuronales de avance para datos tabulares, redes convolucionales para visión por computadora, redes recurrentes y redes de memoria a corto plazo (LSTM) para procesamiento de secuencias, codificadores-decodificadores y transformadores para procesamiento de lenguaje natural (¡y más!) , codificadores automáticos, redes generativas adversarias (GAN) y modelos de difusión para el aprendizaje generativo
- Técnicas para entrenar redes neuronales profundas
- Cómo crear un agente (por ejemplo, un bot en un juego) que pueda aprender buenas estrategias mediante prueba y error, utilizando el aprendizaje por refuerzo.
- Cargar y preprocesar grandes cantidades de datos de manera eficiente
- Entrenamiento e implementación de modelos TensorFlow a escala

La primera parte se basa principalmente en Scikit-Learn, mientras que la segunda parte utiliza TensorFlow y Keras.

#### PRECAUCIÓN

No se lance a aguas profundas demasiado apresuradamente: si bien el aprendizaje profundo es sin duda una de las áreas más interesantes del aprendizaje automático, primero debe dominar los fundamentos. Además, la mayoría de los problemas se pueden resolver bastante bien utilizando técnicas más simples, como bosques aleatorios y métodos de conjuntos (que se analizan en la [Parte I](#)). El aprendizaje profundo es más adecuado para problemas complejos como el reconocimiento de imágenes, el reconocimiento de voz o el procesamiento del lenguaje natural, y requiere una gran cantidad de datos, potencia informática y paciencia (a menos que pueda aprovechar una red neuronal previamente entrenada, como verá).

## Cambios entre el primero y el segundo Edición

Si ya leyó la primera edición, estos son los principales cambios entre la primera y la segunda edición:

- Todo el código se migró de TensorFlow 1.x a TensorFlow 2.x y reemplacé la mayor parte del código de TensorFlow de bajo nivel (gráficos, sesiones, columnas de características, estimadores, etc.) con código Keras mucho más simple.
- La segunda edición presentó la API de datos para cargar y preprocesar grandes conjuntos de datos, la API de estrategias de distribución para entrenar e implementar modelos TF a escala, TF Serving y Google Cloud AI Platform para producir modelos y (brevemente) TF Transform, TFLite, TF Addons/ Seq2Seq, TensorFlow.js y agentes TF.
- También introdujo muchos temas adicionales de ML, incluido un nuevo capítulo sobre aprendizaje no supervisado y técnicas de visión por computadora.

para detección de objetos y segmentación semántica, manejo de secuencias utilizando redes neuronales convolucionales (CNN), procesamiento del lenguaje natural (NLP) utilizando redes neuronales recurrentes (RNN), CNN y transformadores, GAN y más.

Ver <https://homl.info/changes2> para más detalles.

## Cambios entre el segundo y el tercero Edición

Si lees la segunda edición, estos son los principales cambios entre la segunda y la tercera edición:

- Todo el código se actualizó a las últimas versiones de la biblioteca. En particular, esta tercera edición presenta muchas adiciones nuevas a Scikit-Learn (por ejemplo, seguimiento de nombres de características, aumento de gradiente basado en histogramas, propagación de etiquetas y más). También presenta la biblioteca Keras Tuner para el ajuste de hiperparámetros, la biblioteca Transformers de Hugging Face para el procesamiento del lenguaje natural y las nuevas capas de preprocessamiento y aumento de datos de Keras.
- Se agregaron varios modelos de visión (ResNeXt, DenseNet, MobileNet, CSPNet y EfficientNet), así como pautas para elegir el correcto.
- **El capítulo 15** ahora analiza los datos de pasajeros de autobuses y trenes de Chicago en lugar de las series de tiempo generadas, e introduce el modelo ARMA y sus variantes.
- **El capítulo 16** sobre procesamiento del lenguaje natural ahora construye un modelo de traducción del inglés al español, primero usando un codificador-decodificador RNN y luego usando un modelo transformador. El capítulo también cubre modelos de lenguaje como Switch Transformers, DistilBERT, T5 y PaLM (con indicaciones de cadena de pensamiento). Además, introduce transformadores de visión (ViT) y ofrece una

descripción general de algunos modelos visuales basados en transformadores, como transformadores de imágenes eficientes en datos (DeiT), Perceiver y DINO, así como una breve descripción general de algunos modelos multimodales grandes, incluidos CLIP, DALL·E, Flamingo y GATO.

- El [capítulo 17](#) sobre aprendizaje generativo presenta ahora los modelos de difusión y muestra cómo implementar un modelo probabilístico de difusión de eliminación de ruido (DDPM) desde cero.
- El [Capítulo 19](#) migró de Google Cloud AI Platform a Google Vertex AI y utiliza Keras Tuner distribuido para búsquedas de hiperparámetros a gran escala. El capítulo ahora incluye código TensorFlow.js con el que puede experimentar en línea. También presenta técnicas de capacitación distribuida adicionales, incluidas PipeDream y Pathways.
- Para permitir todo el contenido nuevo, algunas secciones se movieron en línea, incluidas instrucciones de instalación, análisis de componentes principales del kernel (PCA), detalles matemáticos de mezclas bayesianas gaussianas, agentes TF y apéndices anteriores A (soluciones de ejercicios), C (soporte matemáticas de máquinas vectoriales) y E (arquitecturas de redes neuronales adicionales).

Ver <https://homl.info/changes3> para más detalles.

## Otros recursos

Hay muchos recursos excelentes disponibles para aprender sobre el aprendizaje automático. Por ejemplo, el curso de ML de Andrew Ng [en Coursera](#) Es sorprendente, aunque requiere una importante inversión de tiempo.

También hay muchos sitios web interesantes sobre aprendizaje automático, incluida la excepcional [Guía del usuario de Scikit-Learn](#). También puedes disfrutar [de Dataquest](#), que proporciona tutoriales interactivos muy interesantes y blogs de ML como los que figuran en [Quora](#).

Hay muchos otros libros introductorios sobre el aprendizaje automático. En particular:

- **Ciencia de datos desde cero** de Joel Grus , 2<sup>a</sup> edición (O'Reilly), presenta los fundamentos del aprendizaje automático e implementa algunos de los principales algoritmos en Python puro (desde cero, como su nombre indica).
- Machine Learning: An Algorithmic Perspective, segunda edición de Stephen Marsland (Chapman & Hall), es una excelente introducción al aprendizaje automático, que cubre una amplia gama de temas en profundidad con ejemplos de código en Python (también desde cero, pero usando NumPy).
- Python Machine Learning de Sebastian Raschka , tercera edición (Packt Publishing), también es una excelente introducción al aprendizaje automático y aprovecha las bibliotecas de código abierto de Python (Pylearn 2 y Theano).
- Deep Learning with Python de François Chollet , segunda edición (Manning), es un libro muy práctico que cubre una amplia gama de temas de forma clara y concisa, como se podría esperar del autor de la excelente biblioteca Keras. Favorece los ejemplos de código sobre la teoría matemática.
- **El libro de cien páginas sobre aprendizaje automático** de Andriy Burkov (autoeditado) es muy breve pero cubre una impresionante variedad de temas, presentándolos en términos accesibles sin rehuir las ecuaciones matemáticas.
- Learning from Data (MLBook), de Yaser S. Abu-Mostafa, Malik Magdon-Ismail y Hsuan-Tien Lin, es un enfoque bastante teórico del aprendizaje automático que proporciona conocimientos profundos, en particular sobre el equilibrio entre sesgo y varianza (consulte [el Capítulo 4](#)) . .
- La inteligencia artificial de Stuart Russell y Peter Norvig : una Modern Approach, cuarta edición (Pearson), es un gran (y enorme)

Libro que cubre una increíble cantidad de temas, incluido el aprendizaje automático.  
Ayuda a poner el aprendizaje automático en perspectiva.

- Deep Learning for Coders with fastai y PyTorch (O'Reilly) de Jeremy Howard y Sylvain Gugger proporciona una introducción maravillosamente clara y práctica al aprendizaje profundo utilizando las bibliotecas fastai y PyTorch.

Finalmente, unirse a sitios web de competencias de ML como [Kaggle.com](#) le permitirá practicar sus habilidades en problemas del mundo real, con la ayuda y el conocimiento de algunos de los mejores profesionales de ML que existen.

## Las convenciones usadas en este libro

En este libro se utilizan las siguientes convenciones tipográficas:

### Itálico

Indica nuevos términos, URL, direcciones de correo electrónico, nombres de archivos y extensiones de archivos.

### Ancho constante

Se utiliza para listados de programas, así como dentro de párrafos para hacer referencia a elementos del programa, como nombres de funciones o variables, bases de datos, tipos de datos, variables de entorno, declaraciones y palabras clave.

### Negrita de ancho constante

Muestra comandos u otro texto que el usuario debe escribir literalmente.

### Cursiva de ancho constante

Muestra texto que debe reemplazarse con valores proporcionados por el usuario o por valores determinados por el contexto.

## Puntuación

Para evitar confusiones, la puntuación aparece fuera de las comillas a lo largo del libro. Mis disculpas a los puristas.

### CONSEJO

Este elemento significa un consejo o sugerencia.

### NOTA

Este elemento significa una nota general.

### ADVERTENCIA

Este elemento indica una advertencia o precaución.

## Aprendizaje en línea de O'Reilly

### NOTA

Durante más de 40 años, **O'Reilly Media** ha proporcionado tecnología y capacitación empresarial, conocimientos y perspectivas para ayudar a las empresas a tener éxito.

Nuestra red única de expertos e innovadores comparte su conocimiento y experiencia a través de libros, artículos y nuestra plataforma de aprendizaje en línea. La plataforma de aprendizaje en línea de O'Reilly le brinda acceso bajo demanda a cursos de capacitación en vivo, rutas de aprendizaje en profundidad, entornos de codificación interactivos y una amplia colección de texto

O'Reilly y más de 200 editoriales más. Para obtener más información, visite <https://oreilly.com>.

## Cómo contactarnos

Por favor dirija sus comentarios y preguntas sobre este libro al editor:

- O'Reilly Media, Inc.
- 1005 Gravenstein Carretera Norte
- Sebastopol, CA 95472
- 800-998-9938 (en Estados Unidos o Canadá)
- 707-829-0515 (internacional o local)
- 707-829-0104 (fax)

Tenemos una página web para este libro, donde enumeramos erratas, ejemplos y cualquier información adicional. Puede acceder a esta página en <https://homl.info/oreilly3>.

Envíe un correo electrónico a [bookquestions@oreilly.com](mailto:bookquestions@oreilly.com) para comentar o hacer preguntas técnicas sobre este libro.

Para noticias e información sobre nuestros libros y cursos, visite <https://oreilly.com>.

Encuéntrenos en LinkedIn: <https://linkedin.com/company/oreilly-media>

Síguenos en Twitter: <https://twitter.com/oreillymedia>

Míranos en YouTube: <https://youtube.com/oreillymedia>

## Expresiones de gratitud

Nunca en mis sueños más locos imaginé que la primera y segunda edición de este libro tendrían una audiencia tan grande. Recibí muchísimos mensajes de lectores, muchos de ellos haciendo preguntas, algunos amablemente señalando erratas y la mayoría enviándome palabras de aliento. No puedo expresar lo agradecido que estoy con todos estos lectores por su tremendo apoyo.

¡Muchas gracias a todos! No dude en [enviar problemas en GitHub](#). si encuentra errores en los ejemplos de código (o simplemente para hacer preguntas), o para enviar [erratas](#) si encuentra errores en el texto. Algunos lectores también compartieron cómo este libro les ayudó a conseguir su primer trabajo o cómo les ayudó a resolver un problema concreto en el que estaban trabajando. Estos comentarios me parecen increíblemente motivadores. Si encuentra útil este libro, me encantaría que pudiera compartir su historia conmigo, ya sea de forma privada (por ejemplo, a través de [LinkedIn](#)) . o públicamente (por ejemplo, envíame un tweet a [@aureliengeron](#) o escribe una [reseña en Amazon](#) ).

Muchas gracias también a todas las maravillosas personas que ofrecieron su tiempo y experiencia para revisar esta tercera edición, corrigiendo errores y haciendo innumerables sugerencias. Esta edición es mucho mejor gracias a ellos: Olzhas Akpambetov, George Bonner, François Chollet, Siddha Gangju, Sam Goodman, Matt Harrison, Sasha Sobran, Lewis Tunstall, Leandro von Werra y mi querido hermano Sylvain. ¡Sois todos increíbles!

También estoy muy agradecido a las muchas personas que me apoyaron a lo largo del camino, respondiendo mis preguntas, sugiriendo mejoras y contribuyendo al código en GitHub: en particular, Yannick Assogba, Ian Beauregard, Ulf Bissbort, Rick Chao, Peretz Cohen, Kyle Gallatin, Hannes Hapke, Victor Khaustov, Soonson Kwon, Eric Lebigot, Jason Mayes, Laurence Moroney, Sara Robinson, Joaquín Ruales y Yuefeng Zhou.

Este libro no existiría sin el fantástico personal de O'Reilly, en particular Nicole Taché, quien me brindó comentarios perspicaces y siempre fue alegre, alentadora y servicial: no podría soñar con un editor mejor. Muchas gracias también a Michele Cronin, que me animó.

A través de los capítulos finales y logré llevarme más allá de la línea de meta. Gracias a todo el equipo de producción, en particular a Elizabeth Kelly y Kristen Brown. Gracias también a Kim Cofer por la minuciosa edición y a Johnny O'Toole, quien manejó la relación con Amazon y respondió muchas de mis preguntas. Gracias a Kate Dullea por mejorar enormemente mis ilustraciones. Gracias a Marie Beaugureau, Ben Lorica, Mike Loukides y Laurel Ruma por creer en este proyecto y ayudarme a definir su alcance. Gracias a Matt Hacker y a todo el equipo de Atlas por responder todas mis preguntas técnicas sobre formato, AsciiDoc, MathML y LaTeX, y gracias a Nick Adams, Rebecca Demarest, Rachel Head, Judith McConville, Helen Monroe, Karen Montgomery, Rachel Roumeliotis, y a todos los demás en O'Reilly que contribuyeron a este libro.

Nunca olvidaré a todas las personas maravillosas que me ayudaron con la primera y segunda edición de este libro: amigos, colegas, expertos, incluidos muchos miembros del equipo de TensorFlow. La lista es larga: Olzhas Akpambetov, Karmel Allison, Martin Andrews, David Andrzejewski, Paige Bailey, Lukas Biewald, Eugene Brevdo, William Chargin, François Chollet, Clément Courbet, Robert Crowe, Mark Daoust, Daniel “Wolff” Dobson, Julien Dubois, Mathias Kende, Daniel Kitachewsky, Nick Felt, Bruce Fontaine, Justin Francis, Goldie Gadde, Irene Giannoumis, Ingrid von Glehn, Vincent Guilbeau, Sandeep Gupta, Priya Gupta, Kevin Haas, Eddy Hung, Konstantinos Katsiapis, Viacheslav Kovalevskyi, Jon Krohn, Allen Lavoie, Karim Matrah, Grégoire Mesnil, Clemens Mewald, Dan Moldovan, Dominic Monn, Sean Morgan, Tom O’Malley, James Pack, Alexander Pak, Haesun Park, Alexandre Passos, Ankur Patel, Josh Patterson, André Susano Pinto, Anthony Platanios, Anosh Raj, Oscar Ramirez, Anna Revinskaya, Saurabh Saxena, Salim Sémaoune, Ryan Sepassi, Vitor Sessak, Jiri Simska, Iain Smears, Xiaodan Song, Christina Sorokin, Michel Tessier, Wiktor Tomczak, Dustin Tran, Todd Wang, Pete Warden, Rich Washington

Wicke, Edd Wilder-James, Sam Witteveen, Jason Zaman, Yuefeng Zhou y mi hermano Sylvain.

Por último, pero no menos importante, estoy infinitamente agradecido a mi amada esposa, Emmanuelle, y a nuestros tres maravillosos hijos, Alexandre, Rémi y Gabrielle, por animarme a trabajar duro en este libro. Su insaciable curiosidad no tenía precio: explicar algunos de los conceptos más difíciles de este libro a mi esposa e hijos me ayudó a aclarar mis pensamientos y mejoró directamente muchas partes del mismo. Además me siguen trayendo galletas y café, ¿quién podría pedir más?

---

<sup>1</sup> Geoffrey E. Hinton et al., “Un algoritmo de aprendizaje rápido para redes de creencias profundas”, Computación neuronal 18 (2006): 1527-1554.

<sup>2</sup> A pesar de que las redes neuronales convolucionales profundas de Yann LeCun habían funcionado bien para el reconocimiento de imágenes desde la década de 1990, aunque no eran de propósito tan general.

# Parte I. Los fundamentos del aprendizaje automático

---

# Capítulo 1. El panorama del aprendizaje automático

---

No hace mucho, si hubieras cogido el teléfono y le hubieras preguntado cómo llegar a casa, te habría ignorado y la gente habría cuestionado tu cordura. Pero el aprendizaje automático ya no es ciencia ficción: miles de millones de personas lo utilizan cada día. Y la verdad es que existe desde hace décadas en algunas aplicaciones especializadas, como el reconocimiento óptico de caracteres (OCR). La primera aplicación de aprendizaje automático que realmente se generalizó y mejoró la vida de cientos de millones de personas, se apoderó del mundo en la década de 1990: el filtro de spam. No es exactamente un robot consciente de sí mismo, pero técnicamente califica como aprendizaje automático: en realidad ha aprendido tan bien que ya rara vez es necesario marcar un correo electrónico como spam. Le siguieron cientos de aplicaciones de aprendizaje automático que ahora impulsan silenciosamente cientos de productos y funciones que utiliza con regularidad: indicaciones de voz, traducción automática, búsqueda de imágenes, recomendaciones de productos y muchas más.

¿Dónde comienza y dónde termina el aprendizaje automático? ¿Qué significa exactamente que una máquina aprenda algo? Si descargo una copia de todos los artículos de Wikipedia, ¿realmente mi computadora ha aprendido algo? ¿Es de repente más inteligente? En este capítulo comenzaré aclarando qué es el aprendizaje automático y por qué es posible que desee utilizarlo.

Luego, antes de salir a explorar el continente del aprendizaje automático, echaremos un vistazo al mapa y conoceremos las principales regiones y los hitos más notables: aprendizaje supervisado versus no supervisado y sus variantes, aprendizaje en línea versus aprendizaje por lotes, aprendizaje basado en instancias versus aprendizaje por lotes, aprendizaje basado en modelos. Luego, analizaremos el flujo de trabajo de un proyecto de aprendizaje automático típico, analizaremos los principales desafíos que puede enfrentar y cubriremos cómo evaluar y ajustar un sistema de aprendizaje automático.

Este capítulo presenta muchos conceptos (y jerga) fundamentales que todo científico de datos debería saber de memoria. Será una descripción general de alto nivel (es el único capítulo sin mucho código), todo bastante simple, pero mi objetivo es asegurarme de que todo quede muy claro antes de continuar con el resto del libro. ¡Así que tómate un café y comencemos!

CONSEJO

Si ya está familiarizado con los conceptos básicos del aprendizaje automático, puede pasar directamente al [Capítulo 2](#). Si no está seguro, intente responder todas las preguntas enumeradas al final del capítulo antes de continuar.

## ¿Qué es el aprendizaje automático?

El aprendizaje automático es la ciencia (y el arte) de programar computadoras para que puedan aprender de los datos.

Aquí hay una definición un poco más general:

[El aprendizaje automático es el] campo de estudio que brinda a las computadoras la capacidad de aprender sin ser programadas explícitamente.

—Arturo Samuel, 1959

Y uno más orientado a la ingeniería:

Se dice que un programa de computadora aprende de la experiencia E con respecto a alguna tarea T y alguna medida de desempeño P, si su desempeño en T, medido por P, mejora con la experiencia E.

—Tom Mitchell, 1997

Su filtro de spam es un programa de aprendizaje automático que, dados ejemplos de correos electrónicos no deseados (marcados por los usuarios) y ejemplos de correos electrónicos normales (no spam, también llamados "ham"), puede aprender a marcar spam. Los ejemplos que utiliza el sistema para aprender se denominan conjunto de entrenamiento. Cada

El ejemplo de entrenamiento se llama instancia de entrenamiento (o muestra). La parte de un sistema de aprendizaje automático que aprende y hace predicciones se llama modelo. Las redes neuronales y los bosques aleatorios son ejemplos de modelos.

En este caso, la tarea T es marcar spam para nuevos correos electrónicos, la experiencia E son los datos de entrenamiento y es necesario definir la medida de rendimiento P ; por ejemplo, puedes utilizar la proporción de correos electrónicos clasificados correctamente. Esta medida de desempeño particular se llama precisión y a menudo se usa en tareas de clasificación.

Si simplemente descarga una copia de todos los artículos de Wikipedia, su computadora tendrá muchos más datos, pero de repente no mejorará en ninguna tarea. Esto no es aprendizaje automático.

## ¿Por qué utilizar el aprendizaje automático?

Considere cómo escribiría un filtro de spam utilizando técnicas de programación tradicionales ([Figura 1-1](#)):

1. En primer lugar, examinaría cómo se ve normalmente el spam. Tú Es posible que notes que algunas palabras o frases (como “4U”, “tarjeta de crédito”, “gratis” y “increíble”) tienden a aparecer con frecuencia en la línea de asunto. Quizás también notes algunos otros patrones en el nombre del remitente, el cuerpo del correo electrónico y otras partes del correo electrónico.
2. Escribirías un algoritmo de detección para cada uno de los patrones. que haya notado, y su programa marcaría los correos electrónicos como spam si se detectaran varios de estos patrones.
3. Probaría su programa y repetiría los pasos 1 y 2 hasta que fuera lo suficientemente bueno para iniciararlo.

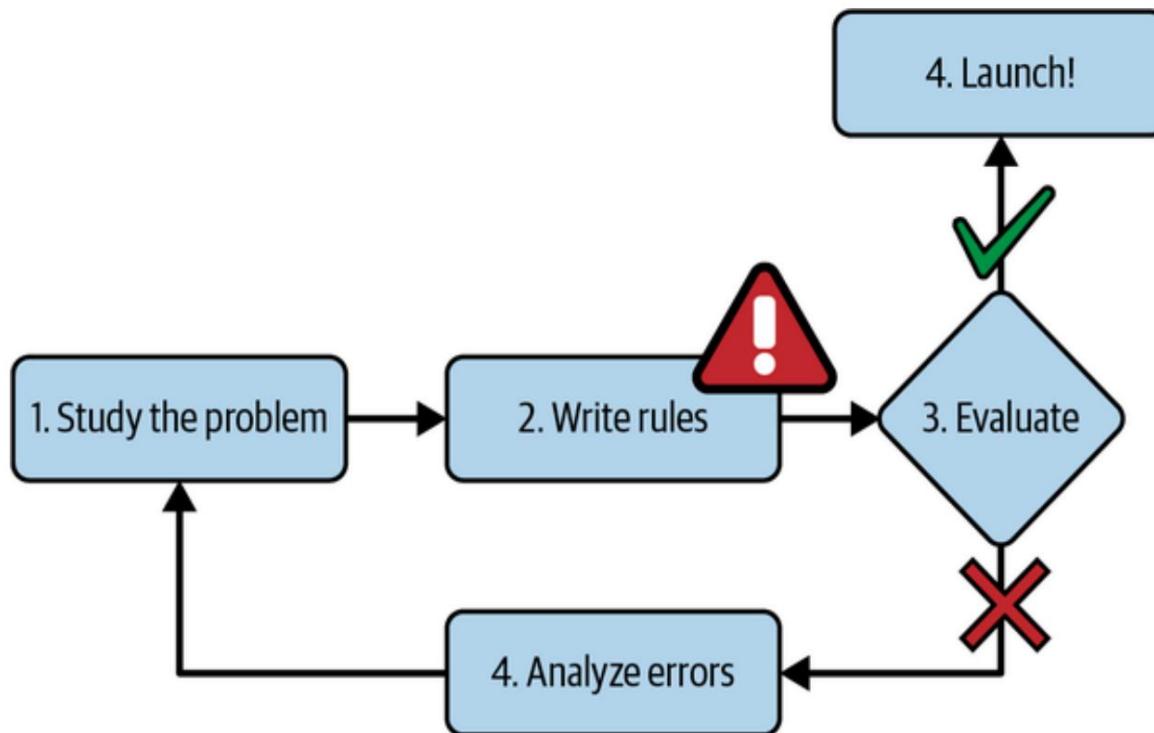


Figura 1-1. El enfoque tradicional

Dado que el problema es difícil, su programa probablemente se convertirá en una larga lista de reglas complejas, bastante difíciles de mantener.

Por el contrario, un filtro de spam basado en técnicas de aprendizaje automático aprende automáticamente qué palabras y frases son buenos predictores de spam al detectar patrones de palabras inusualmente frecuentes en los ejemplos de spam en comparación con los ejemplos de spam (Figura 1-2) . El programa es mucho más corto, más fácil de mantener y probablemente más preciso.

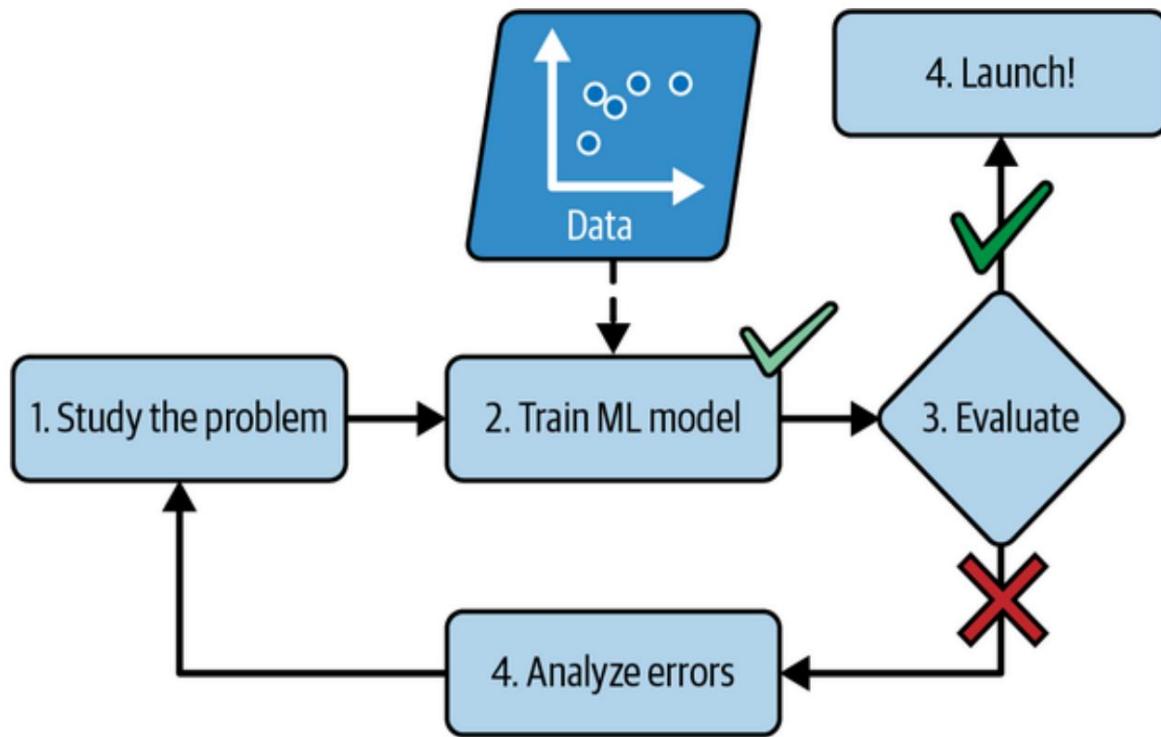


Figura 1-2. El enfoque del aprendizaje automático

¿Qué pasa si los spammers notan que todos sus correos electrónicos que contienen "4U" están bloqueados? En su lugar, podrían empezar a escribir "For U". Sería necesario actualizar un filtro de spam que utilice técnicas de programación tradicionales para marcar los correos electrónicos "Para U". Si los spammers siguen trabajando en torno a su filtro de spam, tendrá que seguir escribiendo nuevas reglas para siempre.

Por el contrario, un filtro de spam basado en técnicas de aprendizaje automático detecta automáticamente que "Para U" se ha vuelto inusualmente frecuente en el spam marcado por los usuarios y comienza a marcarlo sin su intervención (Figura 1-3 ).

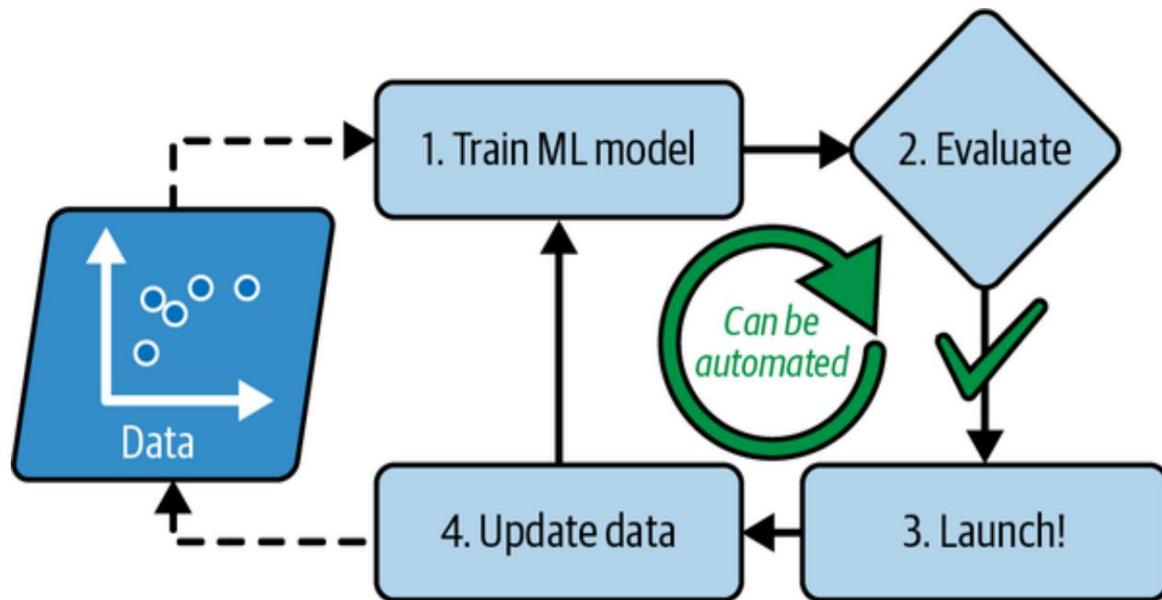


Figura 1-3. Adaptarse automáticamente al cambio

Otra área donde brilla el aprendizaje automático es en problemas que son demasiado complejos para los enfoques tradicionales o que no tienen un algoritmo conocido. Por ejemplo, considere el reconocimiento de voz. Supongamos que quiere empezar de forma sencilla y escribir un programa capaz de distinguir las palabras "uno" y "dos". Podrías notar que la palabra "dos" comienza con un sonido agudo ("T"), por lo que podrías codificar un algoritmo que mida la intensidad del sonido agudo y usarlo para distinguir unos y dos, pero obviamente esta técnica no escalar a miles de palabras pronunciadas por millones de personas muy diferentes en ambientes ruidosos y en docenas de idiomas. La mejor solución (al menos hoy en día) es escribir un algoritmo que aprenda por sí solo, teniendo en cuenta muchos ejemplos de grabaciones para cada palabra.

Finalmente, el aprendizaje automático puede ayudar a los humanos a aprender ([Figura 1-4](#)). Los modelos de ML se pueden inspeccionar para ver qué han aprendido (aunque para algunos modelos esto puede resultar complicado). Por ejemplo, una vez que un filtro de spam ha sido entrenado para detectar suficiente spam, se puede inspeccionar fácilmente para revelar la lista de palabras y combinaciones de palabras que cree que son los mejores predictores de spam. A veces esto revelará correlaciones insospechadas o nuevas tendencias y, por tanto, conducirá a una mejor comprensión del problema. Profundizar en grandes cantidades de

Descubrir patrones ocultos se llama minería de datos, y el aprendizaje automático sobresale en ello.

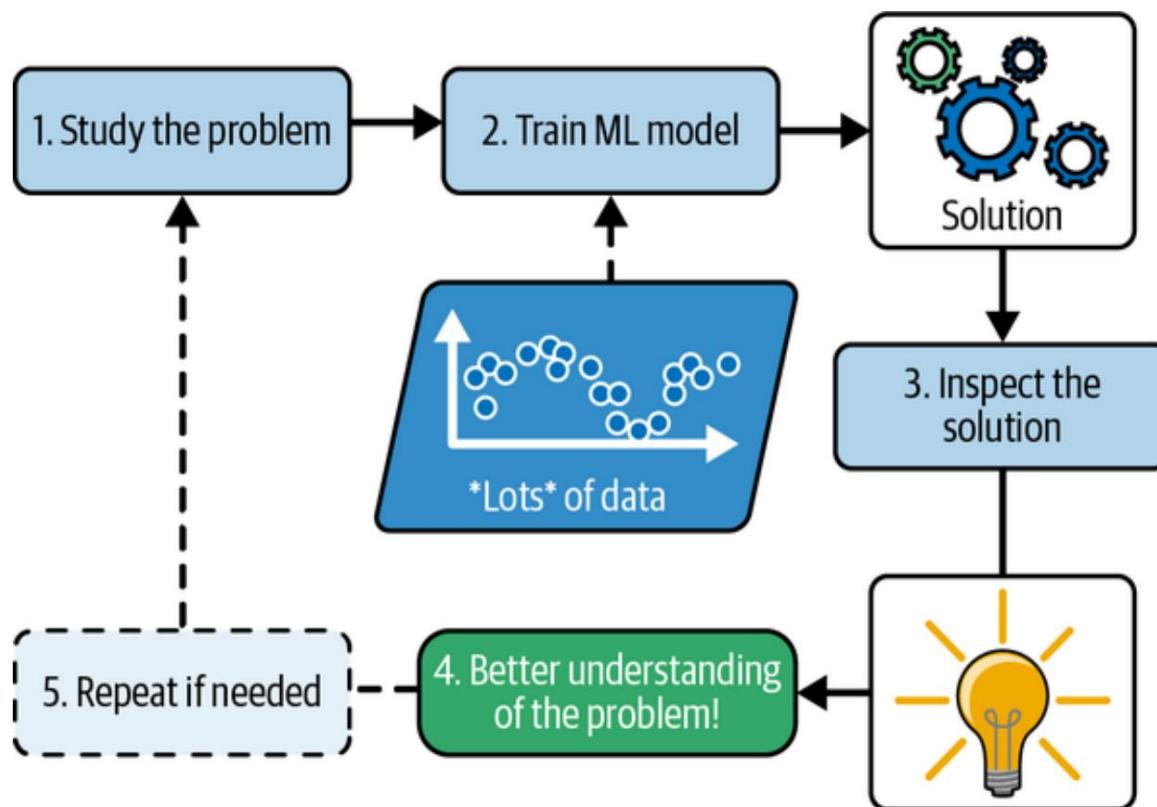


Figura 1-4. El aprendizaje automático puede ayudar a los humanos a aprender

En resumen, el aprendizaje automático es excelente para:

- Problemas para los cuales las soluciones existentes requieren muchos ajustes o largas listas de reglas (un modelo de aprendizaje automático a menudo puede simplificar el código y funcionar mejor que el enfoque tradicional)
- Problemas complejos para los cuales el uso de un enfoque tradicional no ofrece una buena solución (las mejores técnicas de aprendizaje automático tal vez puedan encontrar una solución)
- Entornos fluctuantes (un sistema de aprendizaje automático se puede volver a entrenar fácilmente con nuevos datos, manteniéndolos siempre actualizados)
- Obtener información sobre problemas complejos y grandes cantidades de datos

## Ejemplos de aplicaciones

Veamos algunos ejemplos concretos de tareas de aprendizaje automático, junto con las técnicas que pueden abordarlas:

Analizar imágenes de productos en una línea de producción para clasificarlos automáticamente

Esta es la clasificación de imágenes, que generalmente se realiza utilizando redes neuronales convolucionales (CNN; consulte [el Capítulo 14](#)) o, a veces, transformadores (consulte [el Capítulo 16](#)).

Detección de tumores en escáneres cerebrales

Se trata de segmentación semántica de imágenes, donde se clasifica cada píxel de la imagen (ya que queremos determinar la ubicación exacta y la forma de los tumores), normalmente utilizando CNN o transformadores.

Clasificación automática de artículos de noticias.

Se trata del procesamiento del lenguaje natural (NLP), y más específicamente de la clasificación de texto, que se puede abordar utilizando redes neuronales recurrentes (RNN) y CNN, pero los transformadores funcionan aún mejor (consulte el Capítulo 16 ).

Marcar automáticamente comentarios ofensivos en foros de discusión

Esto también es clasificación de texto, utilizando las mismas herramientas de PNL.

Resumir documentos largos automáticamente

Esta es una rama de la PNL llamada resumen de texto, que nuevamente utiliza las mismas herramientas.

Crear un chatbot o un asistente personal

Esto implica muchos componentes de PNL, incluida la comprensión del lenguaje natural (NLU) y los módulos de respuesta a preguntas.

Pronosticar los ingresos de su empresa el próximo año, basándose en muchas métricas de desempeño

Esta es una tarea de regresión (es decir, predecir valores) que puede abordarse utilizando cualquier modelo de regresión, como un modelo de regresión lineal o un modelo de regresión polinómica (consulte el Capítulo 4), una máquina de vectores de soporte de regresión (consulte el Capítulo 5), un **bosque aleatorio de regresión** (ver [Capítulo 7](#)), o una red neuronal artificial (ver [Capítulo 10](#)). Si desea tener en cuenta secuencias de métricas de rendimiento pasadas, es posible que desee utilizar RNN, CNN o transformadores (consulte los Capítulos [15](#) y [16](#)).

Hacer que tu aplicación reaccione a los comandos de voz

Se trata de reconocimiento de voz, que requiere procesar muestras de audio: dado que son secuencias largas y complejas, normalmente se procesan utilizando RNN, CNN o transformadores (consulte los Capítulos [15](#) y [16](#) ).

Detectar fraude con tarjetas de crédito

Se trata de detección de anomalías, que se puede abordar utilizando bosques de aislamiento, modelos de mezcla gaussiana (consulte el [Capítulo 9](#)) o codificadores automáticos (consulte [el Capítulo 17](#)).

Segmentar a los clientes en función de sus compras para poder diseñar una estrategia de marketing diferente para cada segmento.

Esto es agrupamiento, que se puede lograr usando k-means, DBSCAN y más (consulte [el Capítulo 9](#)).

Representar un conjunto de datos complejo y de alta dimensión en un diagrama claro y revelador.

Se trata de visualización de datos, que a menudo implica técnicas de reducción de dimensionalidad (ver [Capítulo 8](#)).

Recomendar un producto que pueda interesarle a un cliente, basándose en compras anteriores.

Este es un sistema de recomendación. Un enfoque consiste en alimentar compras pasadas (y otra información sobre el cliente) a una red neuronal artificial (consulte [el Capítulo 10](#)) y hacer que genere la próxima compra más probable. Esta red neuronal normalmente se entrenaría en secuencias pasadas de compras de todos los clientes.

### Construyendo un bot inteligente para un juego

Esto a menudo se aborda mediante el aprendizaje por refuerzo (RL; consulte [el Capítulo 18](#)), que es una rama del aprendizaje automático que entrena a los agentes (como los bots) para elegir las acciones que maximizarán sus recompensas con el tiempo (por ejemplo, un bot puede obtener una recompensa). cada vez que el jugador pierde algunos puntos de vida), dentro de un entorno determinado (como el juego). El famoso programa AlphaGo que derrotó al campeón mundial de Go se creó utilizando RL.

Esta lista podría seguir y seguir, pero es de esperar que le dé una idea de la increíble amplitud y complejidad de las tareas que el aprendizaje automático puede abordar y los tipos de técnicas que utilizaría para cada tarea.

## Tipos de sistemas de aprendizaje automático

Hay tantos tipos diferentes de sistemas de aprendizaje automático que resulta útil clasificarlos en categorías amplias, según los siguientes criterios:

- Cómo son supervisados durante el entrenamiento (supervisados, no supervisados, semisupervisados, autosupervisados y otros)
- Si pueden o no aprender de forma incremental sobre la marcha (aprendizaje en línea versus aprendizaje por lotes)

- Ya sea que funcionen simplemente comparando nuevos puntos de datos con puntos de datos conocidos, o detectando patrones en los datos de entrenamiento y construyendo un modelo predictivo, como lo hacen los científicos (aprendizaje basado en instancias versus aprendizaje basado en modelos).

Estos criterios no son excluyentes; puedes combinarlos como quieras. Por ejemplo, un filtro de spam de última generación puede aprender sobre la marcha utilizando un modelo de red neuronal profunda entrenado utilizando ejemplos de spam y jamón proporcionados por humanos; esto lo convierte en un sistema de aprendizaje supervisado, basado en modelos y en línea.

Veamos cada uno de estos criterios un poco más de cerca.

## Supervisión de formación

Los sistemas de ML se pueden clasificar según la cantidad y el tipo de supervisión que reciben durante la formación. Hay muchas categorías, pero discutiremos las principales: aprendizaje supervisado, aprendizaje no supervisado, aprendizaje autosupervisado, aprendizaje semisupervisado y aprendizaje por refuerzo.

Aprendizaje supervisado En

el aprendizaje supervisado, el conjunto de entrenamiento que alimenta al algoritmo incluye las soluciones deseadas, llamadas etiquetas ([Figura 1-5](#)).

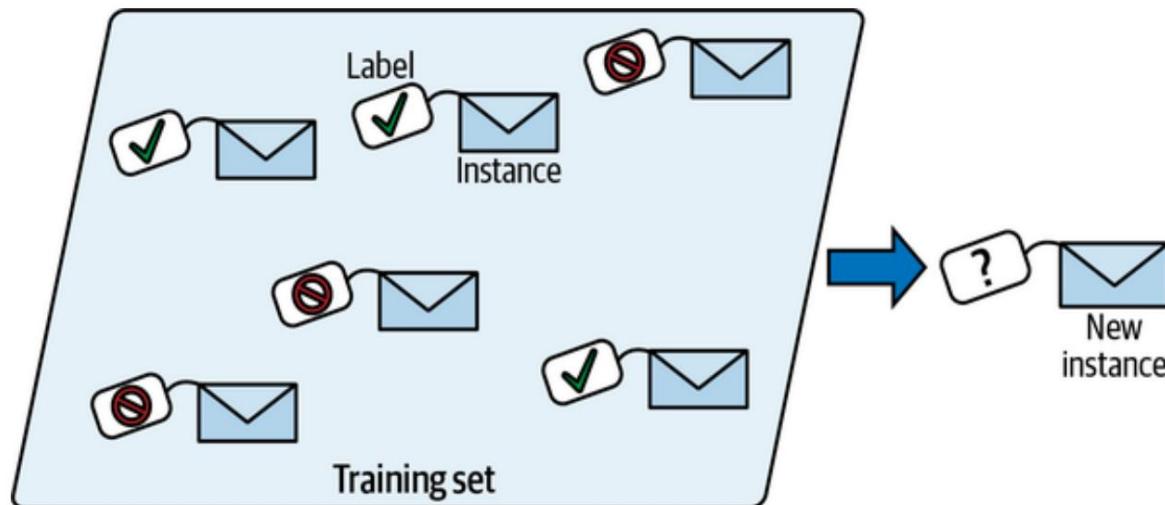


Figura 1-5. Un conjunto de entrenamiento etiquetado para la clasificación de spam (un ejemplo de aprendizaje supervisado)

Una tarea típica de aprendizaje supervisado es la clasificación. El filtro de spam es un buen ejemplo de esto: se entrena con muchos correos electrónicos de ejemplo junto con su clase (spam o ham), y debe aprender a clasificar los correos electrónicos nuevos.

Otra tarea típica es predecir un valor numérico objetivo , como el precio de un automóvil, dado un conjunto de características (kilometraje, antigüedad, marca, etc.). Este tipo de tarea se llama regresión ([Figura 1-6](#)). Para <sup>1</sup>entrenar el sistema, es necesario darle muchos ejemplos de automóviles, incluidas sus características y sus objetivos (es decir, sus precios).

Tenga en cuenta que algunos modelos de regresión también se pueden utilizar para la clasificación y viceversa. Por ejemplo, la regresión logística se usa comúnmente para la clasificación, ya que puede generar un valor que corresponde a la probabilidad de pertenecer a una clase determinada (por ejemplo, 20% de probabilidad de ser spam).

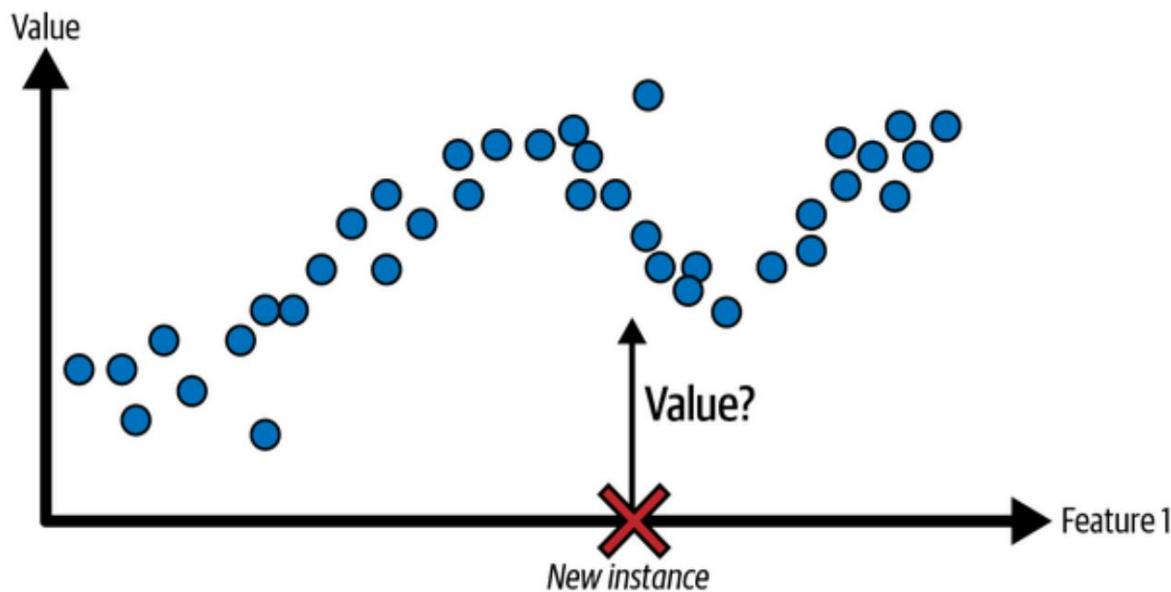


Figura 1-6. Un problema de regresión: predecir un valor, dada una característica de entrada (generalmente hay múltiples características de entrada y, a veces, múltiples valores de salida)

### NOTA

Las palabras objetivo y etiqueta generalmente se tratan como sinónimos en el aprendizaje supervisado, pero objetivo es más común en tareas de regresión y etiqueta es más común en tareas de clasificación. Además, las características a veces se denominan predictores o atributos. Estos términos pueden referirse a muestras individuales (por ejemplo, "la característica de kilometraje de este automóvil es igual a 15.000") o a todas las muestras (por ejemplo, "la característica de kilometraje está fuertemente correlacionada con el precio").

## Aprendizaje sin supervisión

En el aprendizaje no supervisado, como se puede imaginar, los datos de entrenamiento no están etiquetados ([Figura 1-7](#)). El sistema intenta aprender sin profesor.

Por ejemplo, supongamos que tiene muchos datos sobre los visitantes de su blog. Es posible que desee ejecutar un algoritmo de agrupación para intentar detectar grupos de visitantes similares ([Figura 1-8](#)). En ningún momento le dices al algoritmo a qué grupo pertenece un visitante: encuentra esas conexiones sin tu ayuda. Por ejemplo, podría notar que el 40% de sus visitantes son adolescentes a quienes les encantan los cómics y generalmente leen su blog después.

escuela, mientras que el 20% son adultos que disfrutan de la ciencia ficción y la visitan los fines de semana. Si utiliza un algoritmo de agrupamiento jerárquico , también puede subdividir cada grupo en grupos más pequeños. Esto puede ayudarte a orientar tus publicaciones para cada grupo.

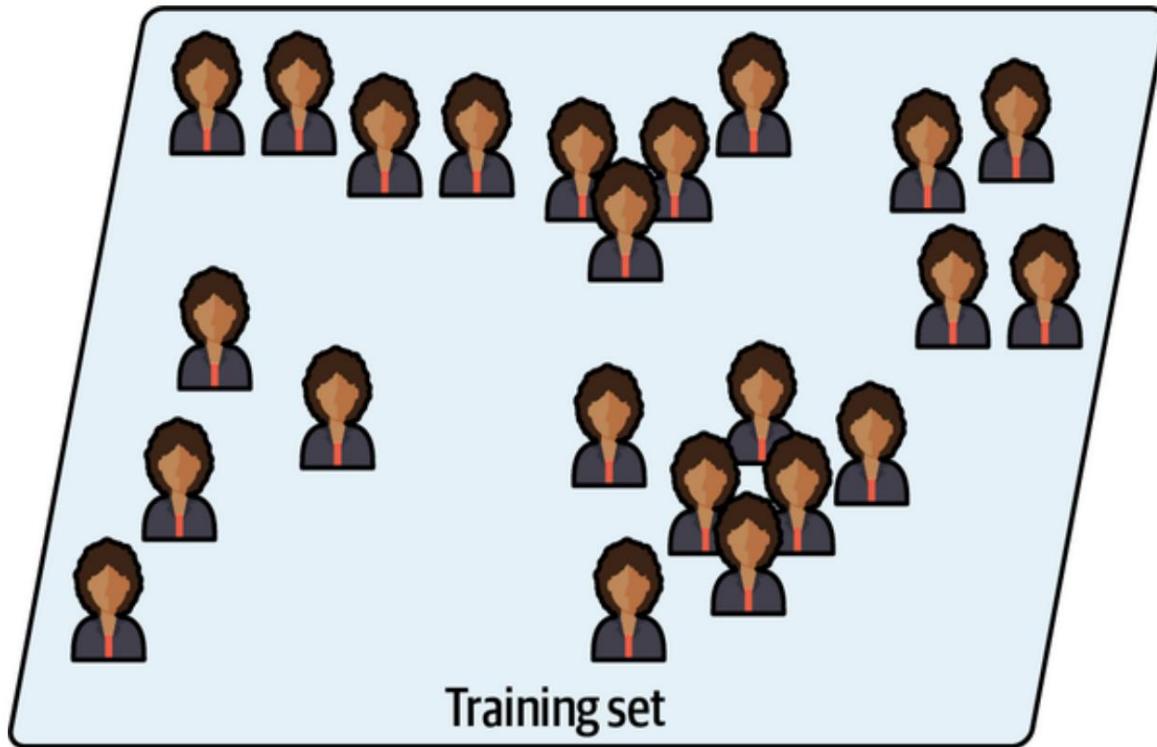


Figura 1-7. Un conjunto de entrenamiento sin etiquetar para el aprendizaje no supervisado

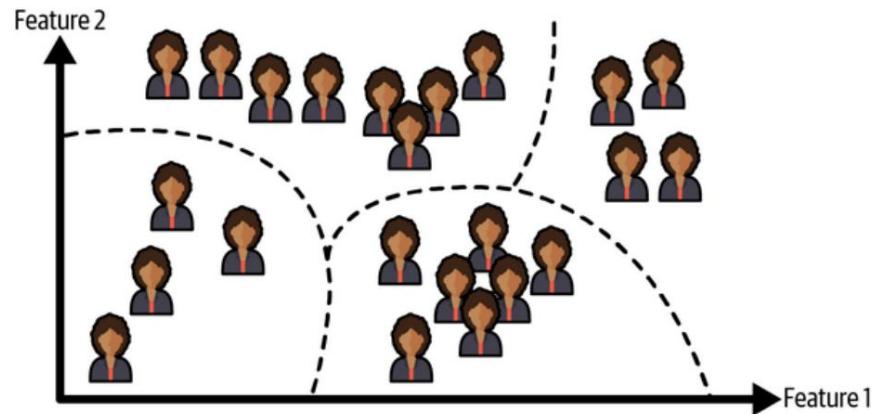


Figura 1-8. Agrupación

Los algoritmos de visualización también son buenos ejemplos de aprendizaje no supervisado: les proporcionas una gran cantidad de datos complejos y sin etiquetar, y ellos generan una representación 2D o 3D de tus datos que se puede analizar fácilmente.

trazado ([Figura 1-9](#)). Estos algoritmos intentan preservar tanta estructura como pueden (por ejemplo, tratando de evitar que grupos separados en el espacio de entrada se superpongan en la visualización) para que usted pueda comprender cómo se organizan los datos y tal vez identificar patrones insospechados.

Una tarea relacionada es la reducción de dimensionalidad, en la que el objetivo es simplificar los datos sin perder demasiada información. Una forma de hacerlo es fusionar varias funciones correlacionadas en una. Por ejemplo, el kilometraje de un automóvil puede estar fuertemente correlacionado con su edad, por lo que el algoritmo de reducción de dimensionalidad los fusionará en una característica que represente el desgaste del automóvil. Esto se llama extracción de características.

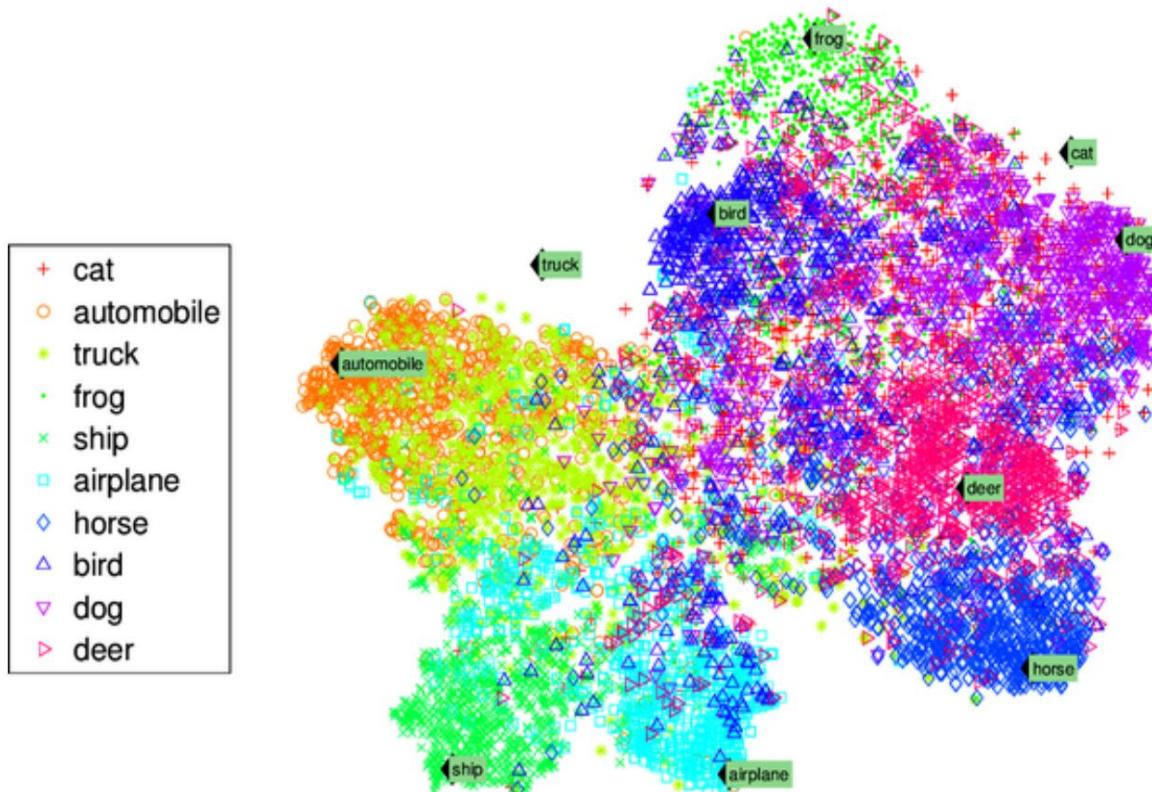


Figura 1-9. Ejemplo de una visualización t-SNE que destaca grupos semánticos<sup>2</sup>

## CONSEJO

A menudo es una buena idea intentar reducir la cantidad de dimensiones en sus datos de entrenamiento utilizando un algoritmo de reducción de dimensionalidad antes de alimentarlo con otro algoritmo de aprendizaje automático (como un algoritmo de aprendizaje supervisado). Se ejecutará mucho más rápido, los datos ocuparán menos espacio en disco y memoria y, en algunos casos, también puede funcionar mejor.

Otra tarea importante no supervisada es la detección de anomalías; por ejemplo, detectar transacciones inusuales con tarjetas de crédito para evitar fraudes, detectar defectos de fabricación o eliminar automáticamente valores atípicos de un conjunto de datos antes de alimentarlos a otro algoritmo de aprendizaje. Al sistema se le muestran principalmente instancias normales durante el entrenamiento, por lo que aprende a reconocerlas; luego, cuando ve una nueva instancia, puede decir si parece normal o si es probable que se trate de una anomalía (consulte la [Figura 1-10](#)). Una tarea muy similar es la detección de novedades: su objetivo es detectar nuevas instancias que se ven diferentes de todas las instancias del conjunto de entrenamiento. Esto requiere tener un conjunto de entrenamiento muy "limpio", desprovisto de cualquier instancia que deseé que detecte el algoritmo. Por ejemplo, si tiene miles de fotografías de perros y el 1% de estas imágenes representan chihuahuas, entonces un algoritmo de detección de novedades no debería tratar las nuevas imágenes de chihuahuas como novedades. Por otro lado, los algoritmos de detección de anomalías pueden considerar a estos perros como tan raros y diferentes de otros perros que probablemente los clasificarían como anomalías (sin ofender a los perros).

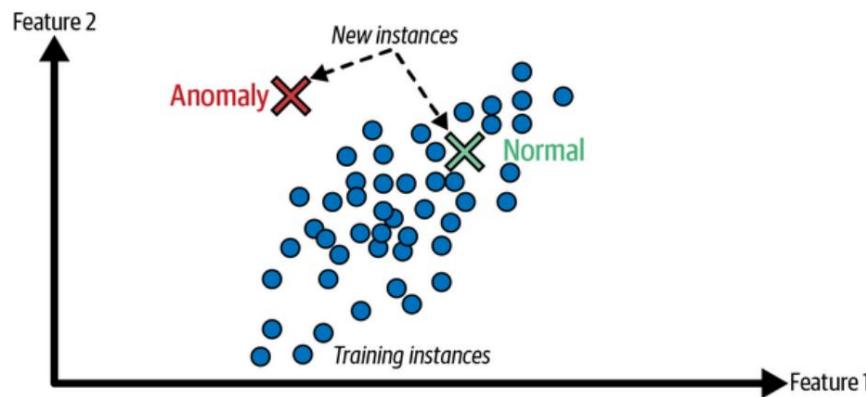


Figura 1-10. Detección de anomalías

Finalmente, otra tarea común no supervisada es el aprendizaje de reglas de asociación, en el que el objetivo es profundizar en grandes cantidades de datos y descubrir relaciones interesantes entre atributos. Por ejemplo, supongamos que usted es dueño de un supermercado. Ejecutar una regla de asociación en sus registros de ventas puede revelar que las personas que compran salsa barbacoa y patatas fritas también tienden a comprar bistec. Por lo tanto, es posible que desee colocar estos elementos uno cerca del otro.

### Aprendizaje semisupervisado

Dado que el etiquetado de datos suele llevar mucho tiempo y ser costoso, a menudo tendrá muchas instancias sin etiquetar y pocas instancias etiquetadas. Algunos algoritmos pueden manejar datos parcialmente etiquetados. Esto se llama aprendizaje semisupervisado ([Figura 1-11](#)).

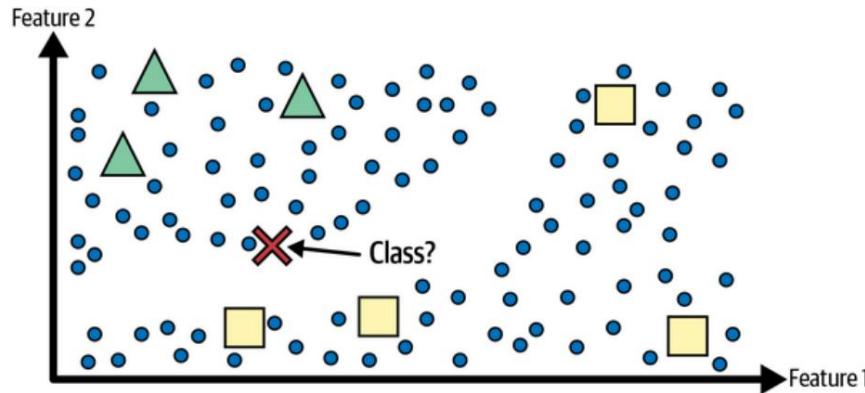


Figura 1-11. Aprendizaje semisupervisado con dos clases (triángulos y cuadrados): los ejemplos sin etiquetar (círculos) ayudan a clasificar una nueva instancia (la cruz) en la clase de triángulo en lugar de en la clase de cuadrado, aunque esté más cerca de los cuadrados etiquetados.

Algunos servicios de alojamiento de fotografías, como Google Photos, son buenos ejemplos de esto. Una vez que subes todas tus fotos familiares al servicio, este reconoce automáticamente que aparece la misma persona A en las fotos 1, 5 y 11, mientras que otra persona B aparece en las fotos 2, 5 y 7. Esta es la parte sin supervisión. del algoritmo (agrupación). Ahora todo lo que el sistema necesita es que usted le diga quiénes son estas personas. Simplemente agregue una etiqueta por persona y podrá nombrar <sup>3</sup> a todos en cada foto, lo cual es útil para buscar fotos.

La mayoría de los algoritmos de aprendizaje semisupervisados son combinaciones de algoritmos supervisados y no supervisados. Por ejemplo, se puede utilizar un algoritmo de agrupamiento para agrupar instancias similares y luego cada instancia sin etiquetar se puede etiquetar con la etiqueta más común en su grupo. Una vez etiquetado todo el conjunto de datos, es posible utilizar cualquier algoritmo de aprendizaje supervisado.

### Aprendizaje autosupervisado

Otro enfoque del aprendizaje automático implica generar un conjunto de datos completamente etiquetado a partir de uno sin etiquetar. Nuevamente, una vez etiquetado todo el conjunto de datos, se puede utilizar cualquier algoritmo de aprendizaje supervisado. Este enfoque se llama aprendizaje autosupervisado.

Por ejemplo, si tiene un conjunto de datos grande de imágenes sin etiquetar, puede enmascarar aleatoriamente una pequeña parte de cada imagen y luego entrenar un modelo para recuperar la imagen original (**Figura 1-12**). Durante el entrenamiento, las imágenes enmascaradas se utilizan como entradas para el modelo y las imágenes originales como etiquetas.

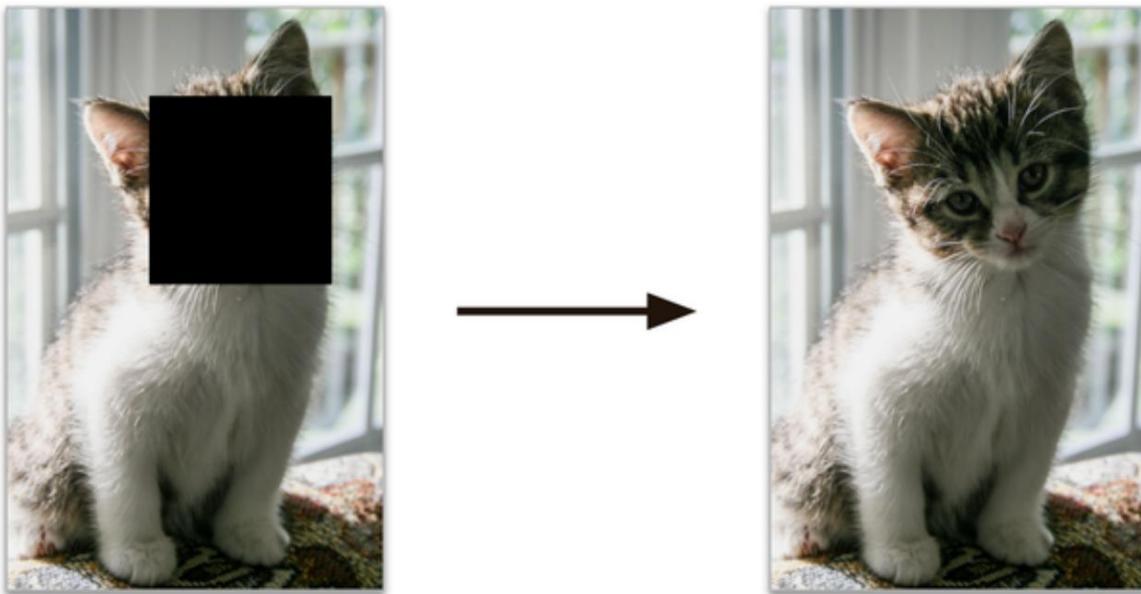


Figura 1-12. Ejemplo de aprendizaje autosupervisado: entrada (izquierda) y objetivo (derecha)

El modelo resultante puede ser bastante útil en sí mismo, por ejemplo, para reparar imágenes dañadas o borrar objetos no deseados de las imágenes. Pero la mayoría de las veces, un modelo entrenado mediante aprendizaje autosupervisado no es el objetivo final. Por lo general, querrás modificar y afinar el modelo para una tarea ligeramente diferente, una que realmente te interese.

Por ejemplo, supongamos que lo que realmente quieres es tener un modelo de clasificación de mascotas: dada una imagen de cualquier mascota, te dirá a qué especie pertenece. Si tiene un gran conjunto de datos de fotografías de mascotas sin etiquetar, puede comenzar entrenando un modelo de reparación de imágenes mediante el aprendizaje autosupervisado. Una vez que funcione bien, debería poder distinguir diferentes especies de mascotas: cuando repara una imagen de un gato cuya cara está enmascarada, debe saber que no debe agregar la cara de un perro.

Suponiendo que la arquitectura de su modelo lo permita (y la mayoría de las funciones neuronales)

como lo hacen las arquitecturas de red), entonces es posible modificar el modelo para que prediga las especies de mascotas en lugar de reparar imágenes. El último paso consiste en ajustar el modelo en un conjunto de datos etiquetado: el modelo ya sabe cómo son los gatos, los perros y otras especies de mascotas, por lo que este paso solo es necesario para que el modelo pueda aprender el mapeo entre las especies que ya conoce. y las etiquetas que esperamos de él.

## NOTA

Transferir conocimiento de una tarea a otra se llama transferencia de aprendizaje y es una de las técnicas más importantes del aprendizaje automático actual, especialmente cuando se utilizan redes neuronales profundas (es decir, redes neuronales compuestas por muchas capas de neuronas). Discutiremos esto en detalle en la [Parte II](#).

Algunas personas consideran que el aprendizaje autosupervisado es parte del aprendizaje no supervisado, ya que se trata de conjuntos de datos sin etiquetar. Pero el aprendizaje autosupervisado utiliza etiquetas (generadas) durante el entrenamiento, por lo que en ese sentido está más cerca del aprendizaje supervisado. Y el término "aprendizaje no supervisado" se utiliza generalmente cuando se trata de tareas como agrupación, reducción de dimensionalidad o detección de anomalías, mientras que el aprendizaje autosupervisado se centra en las mismas tareas que el aprendizaje supervisado: principalmente clasificación y regresión. En resumen, es mejor tratar el aprendizaje autosupervisado como una categoría propia.

## Aprendizaje reforzado

El aprendizaje por refuerzo es una bestia muy diferente. El sistema de aprendizaje, llamado agente en este contexto, puede observar el entorno, seleccionar y realizar acciones y obtener recompensas a cambio (o penalizaciones en forma de recompensas negativas, como se muestra en la [Figura 1-13](#)). Luego debe aprender por sí mismo cuál es la mejor estrategia, llamada política, para obtener la mayor recompensa con el tiempo. Una política define qué acción debe elegir el agente cuando se encuentra en una situación determinada.

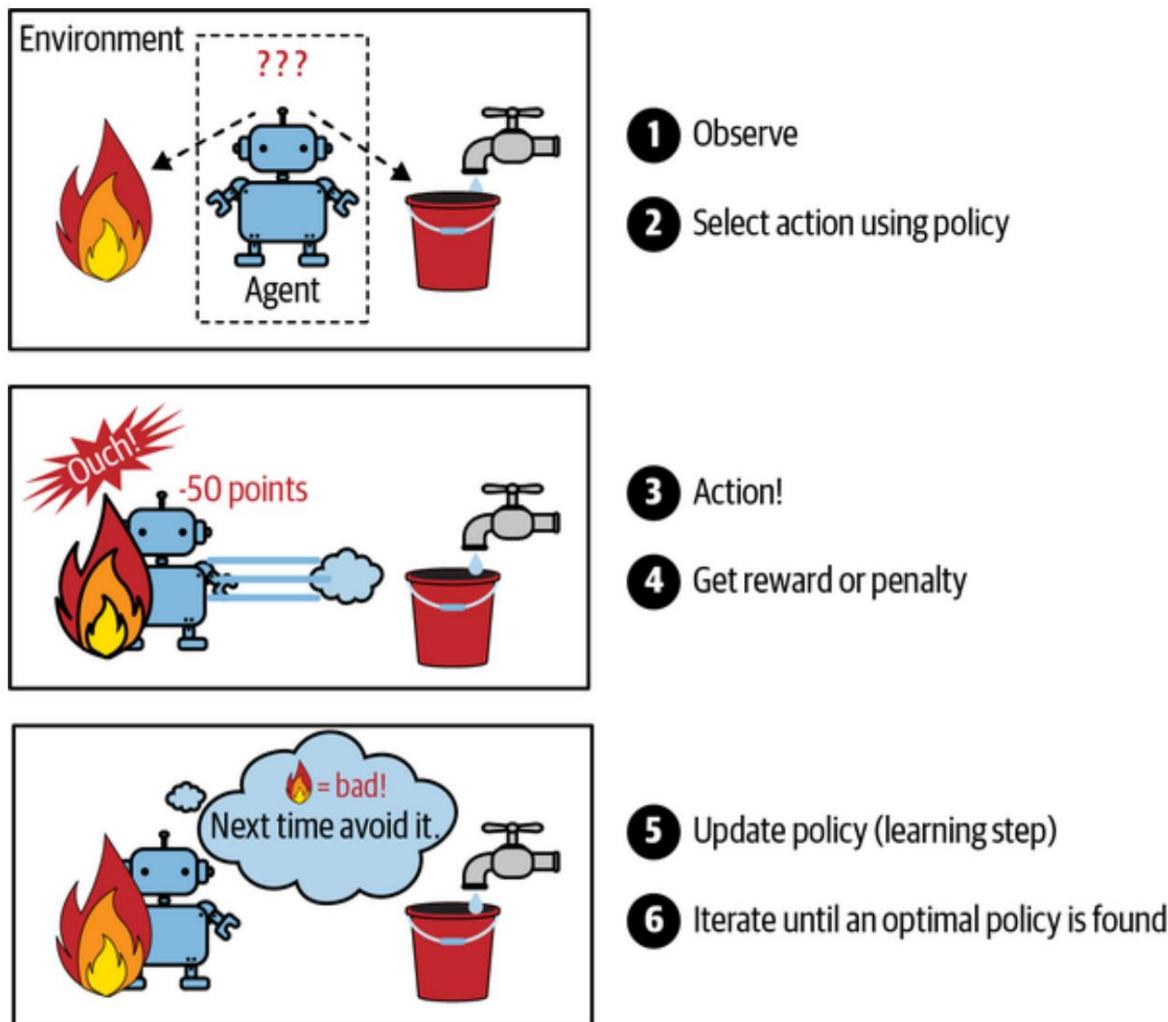


Figura 1-13. Aprendizaje reforzado

Por ejemplo, muchos robots implementan algoritmos de aprendizaje por refuerzo para aprender a caminar. El programa AlphaGo de DeepMind también es un buen ejemplo de aprendizaje por refuerzo: llegó a los titulares en mayo de 2017 cuando venció a Ke Jie, el jugador número uno del mundo en ese momento, en el juego de Go. Aprendió su política ganadora analizando millones de partidas y luego jugando muchas partidas contra sí mismo. Tenga en cuenta que el aprendizaje se desactivó durante los juegos contra el campeón; AlphaGo simplemente estaba aplicando la política que había aprendido. Como verá en la siguiente sección, esto se llama aprendizaje fuera de línea.

## Aprendizaje por lotes versus aprendizaje en

Línea Otro criterio utilizado para clasificar los sistemas de aprendizaje automático es si el sistema puede aprender incrementalmente a partir de un flujo de datos entrantes.

### Aprendizaje por

lotes En el aprendizaje por lotes, el sistema es incapaz de aprender de forma incremental: debe entrenarse utilizando todos los datos disponibles. Por lo general, esto requerirá mucho tiempo y recursos informáticos, por lo que normalmente se realiza sin conexión. Primero se entrena el sistema, luego se lanza a producción y se ejecuta sin aprender más; simplemente aplica lo que ha aprendido. A esto se le llama aprendizaje fuera de línea.

Desafortunadamente, el rendimiento de un modelo tiende a decaer lentamente con el tiempo, simplemente porque el mundo continúa evolucionando mientras el modelo permanece sin cambios. Este fenómeno a menudo se denomina deterioro del modelo o deriva de datos. La solución es volver a entrenar periódicamente el modelo con datos actualizados. La frecuencia con la que debe hacerlo depende del caso de uso: si el modelo clasifica imágenes de perros y gatos, su rendimiento decaerá muy lentamente, pero si el modelo trata con sistemas en rápida evolución, por ejemplo haciendo predicciones en el mercado financiero, entonces es probable que se descomponga bastante rápido.

### ADVERTENCIA

Incluso un modelo entrenado para clasificar imágenes de perros y gatos puede necesitar ser reentrenado periódicamente, no porque los perros y los gatos muten de la noche a la mañana, sino porque las cámaras siguen cambiando, junto con los formatos de imagen, la nitidez, el brillo y las proporciones de tamaño. Además, es posible que a las personas les gusten diferentes razas el próximo año o que decidan vestir a sus mascotas con pequeños sombreros, ¿quién sabe?

Si desea que un sistema de aprendizaje por lotes conozca nuevos datos (como un nuevo tipo de spam), debe entrenar una nueva versión del

sistema desde cero en el conjunto de datos completo (no solo los datos nuevos, sino también los datos antiguos), luego reemplace el modelo antiguo por el nuevo.

Afortunadamente, todo el proceso de capacitación, evaluación y lanzamiento de un sistema de aprendizaje automático se puede automatizar con bastante facilidad (como vimos en la [Figura 1-3](#)), por lo que incluso un sistema de aprendizaje por lotes puede adaptarse al cambio. Simplemente actualice los datos y entrene una nueva versión del sistema desde cero tantas veces como sea necesario.

Esta solución es simple y a menudo funciona bien, pero entrenar utilizando el conjunto completo de datos puede llevar muchas horas, por lo que normalmente entrenaría un nuevo sistema solo cada 24 horas o incluso semanalmente. Si su sistema necesita adaptarse a datos que cambian rápidamente (por ejemplo, para predecir los precios de las acciones), entonces necesita una solución más reactiva.

Además, el entrenamiento en el conjunto completo de datos requiere una gran cantidad de recursos informáticos (CPU, espacio de memoria, espacio en disco, E/S de disco, E/S de red, etc.). Si tienes muchos datos y automatizas tu sistema para entrenar desde cero todos los días, te acabará costando mucho dinero. Si la cantidad de datos es enorme, puede resultar incluso imposible utilizar un algoritmo de aprendizaje por lotes.

Finalmente, si su sistema necesita poder aprender de forma autónoma y tiene recursos limitados (por ejemplo, una aplicación de teléfono inteligente o un rover en Marte), entonces debe llevar consigo grandes cantidades de datos de entrenamiento y consumir muchos recursos para entrenar durante horas cada día. El día es espectacular.

Una mejor opción en todos estos casos es utilizar algoritmos que sean capaces de aprender de forma incremental.

### Aprendizaje en línea

En el aprendizaje en línea, usted entrena el sistema de manera incremental alimentándolo con instancias de datos de manera secuencial, ya sea individualmente o en pequeños grupos llamados minilotes. Cada paso de aprendizaje es rápido y barato, por lo que el sistema puede aprender sobre nuevos datos sobre la marcha, a medida que llegan (consulte [la Figura 1-14](#)).

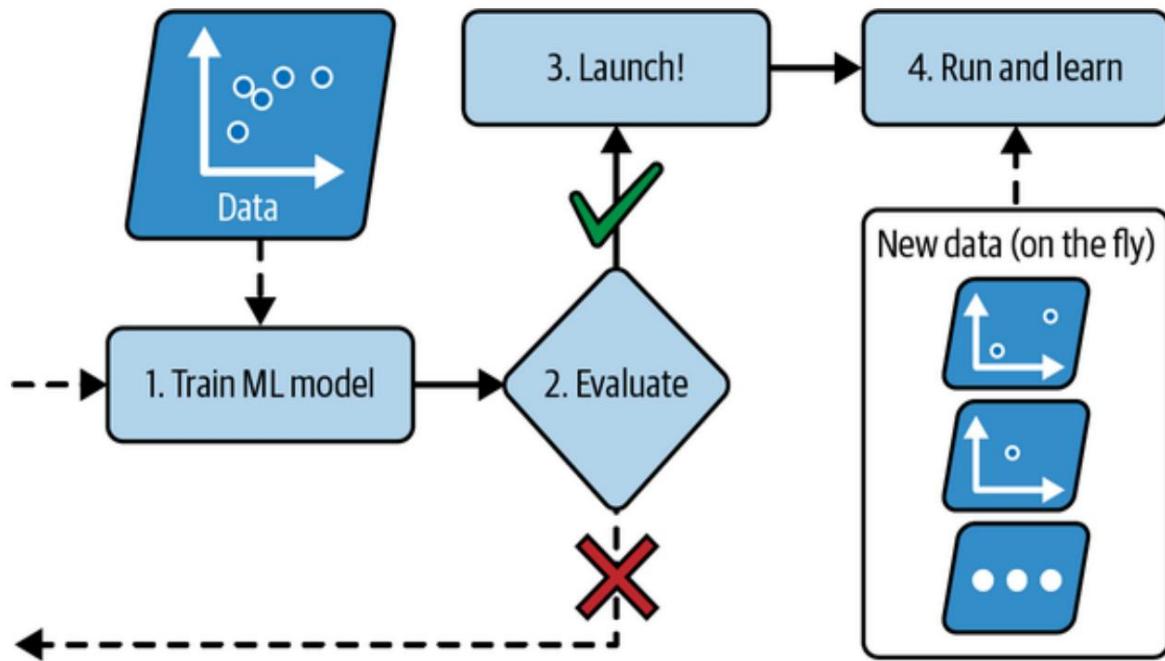


Figura 1-14. En el aprendizaje en línea, un modelo se entrena y se lanza a producción, y luego sigue aprendiendo a medida que llegan nuevos datos.

El aprendizaje en línea es útil para sistemas que necesitan adaptarse a cambios extremadamente rápidos (por ejemplo, para detectar nuevos patrones en el mercado de valores). También es una buena opción si tienes recursos informáticos limitados; por ejemplo, si el modelo se entrena en un dispositivo móvil.

Además, los algoritmos de aprendizaje en línea se pueden utilizar para entrenar modelos en enormes conjuntos de datos que no caben en la memoria principal de una máquina (esto se denomina aprendizaje fuera del núcleo). El algoritmo carga parte de los datos, ejecuta un paso de entrenamiento con esos datos y repite el proceso hasta que se ejecuta con todos los datos (consulte la [Figura 1-15](#)).

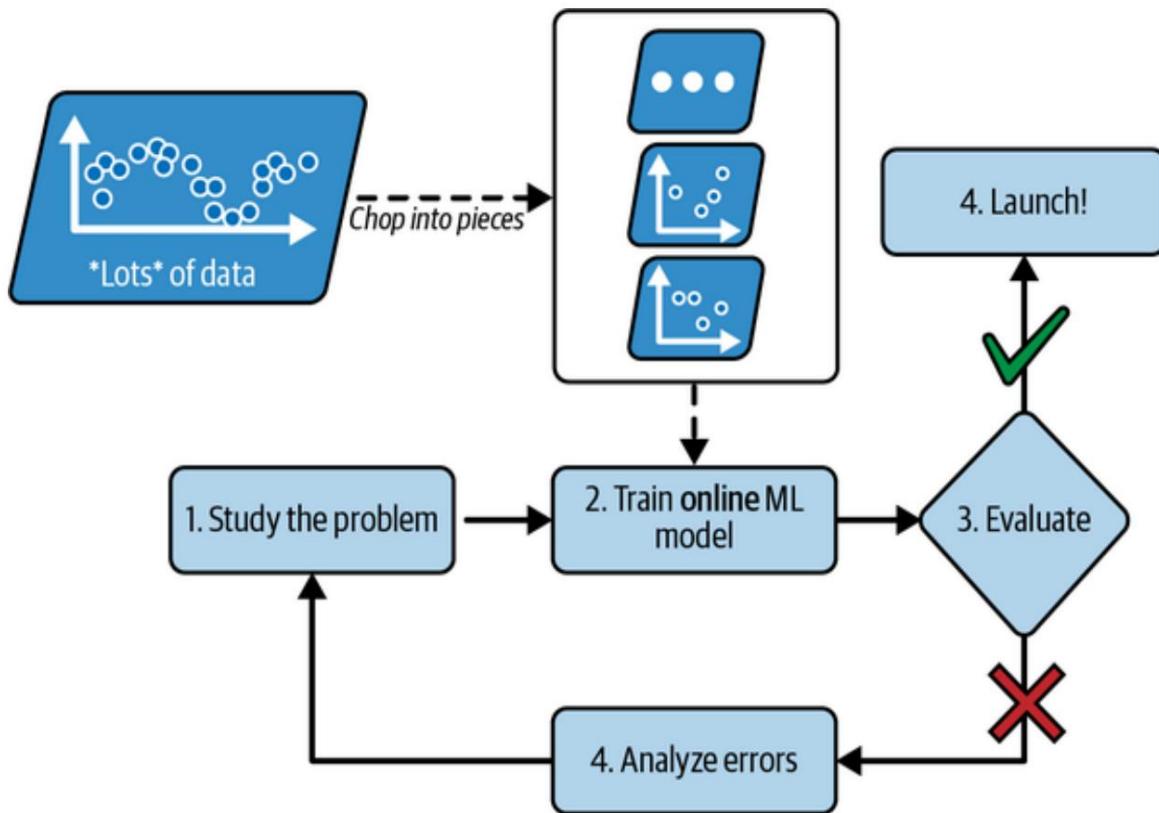


Figura 1-15. Uso del aprendizaje en línea para manejar enormes conjuntos de datos

Un parámetro importante de los sistemas de aprendizaje en línea es la rapidez con la que deben adaptarse a los datos cambiantes: esto se denomina tasa de aprendizaje. Si establece una tasa de aprendizaje alta, su sistema se adaptará rápidamente a los nuevos datos, pero también tenderá a olvidar rápidamente los datos antiguos (y no querrá que un filtro de spam marque solo los últimos tipos de spam que se mostraron). . Por el contrario, si establece una tasa de aprendizaje baja, el sistema tendrá más inercia; es decir, aprenderá más lentamente, pero también será menos sensible al ruido en los nuevos datos o a secuencias de puntos de datos no representativos (valores atípicos).

#### ADVERTENCIA

El aprendizaje fuera del núcleo generalmente se realiza fuera de línea (es decir, no en el sistema en vivo), por lo que el aprendizaje en línea puede ser un nombre confuso. Piense en ello como un aprendizaje incremental.

Un gran desafío con el aprendizaje en línea es que si se introducen datos incorrectos en el sistema, el rendimiento del sistema disminuirá, posiblemente rápidamente (dependiendo de la calidad de los datos y la tasa de aprendizaje). Si es un sistema en vivo, sus clientes lo notarán. Por ejemplo, los datos incorrectos podrían provenir de un error (p. ej., un sensor defectuoso en un robot) o podrían provenir de alguien que intenta engañar al sistema (p. ej., enviar spam a un motor de búsqueda para tratar de obtener una clasificación alta en los resultados de búsqueda). Para reducir este riesgo, debe monitorear su sistema de cerca y desactivar rápidamente el aprendizaje (y posiblemente volver a un estado de funcionamiento anterior) si detecta una caída en el rendimiento. Es posible que también desee monitorear los datos de entrada y reaccionar ante datos anormales; por ejemplo, utilizando un algoritmo de detección de anomalías (consulte [el Capítulo 9](#)).

## Aprendizaje basado en instancias versus aprendizaje basado en modelos

Una forma más de categorizar los sistemas de aprendizaje automático es por cómo se generalizan. La mayoría de las tareas de aprendizaje automático consisten en hacer predicciones. Esto significa que, dada una serie de ejemplos de entrenamiento, el sistema debe poder hacer buenas predicciones para (generalizar) ejemplos que nunca antes ha visto. Tener una buena medida del rendimiento de los datos de entrenamiento es bueno, pero insuficiente; el verdadero objetivo es tener un buen rendimiento en instancias nuevas.

Hay dos enfoques principales para la generalización: aprendizaje basado en instancias y aprendizaje basado en modelos.

### Aprendizaje basado en instancias

Posiblemente la forma más trivial de aprender sea simplemente aprender de memoria. Si creara un filtro de spam de esta manera, simplemente marcaría todos los correos electrónicos que sean idénticos a los que ya han sido marcados por los usuarios; no es la peor solución, pero ciertamente no es la mejor.

En lugar de simplemente marcar correos electrónicos que son idénticos a correos electrónicos no deseados conocidos, su filtro de spam podría programarse para marcar también correos electrónicos que son muy similares a correos electrónicos no deseados conocidos. Esto requiere una medida de

similitud entre dos correos electrónicos. Una medida de similitud (muy básica) entre dos correos electrónicos podría ser contar la cantidad de palabras que tienen en común. El sistema marcará un correo electrónico como spam si tiene muchas palabras en común con un correo electrónico no deseado conocido.

Esto se llama aprendizaje basado en instancias: el sistema aprende los ejemplos de memoria y luego los generaliza a nuevos casos utilizando una medida de similitud para compararlos con los ejemplos aprendidos (o un subconjunto de ellos). Por ejemplo, en la Figura 1-16 la nueva instancia se clasificaría como un triángulo porque la mayoría de las instancias más similares pertenecen a esa clase.

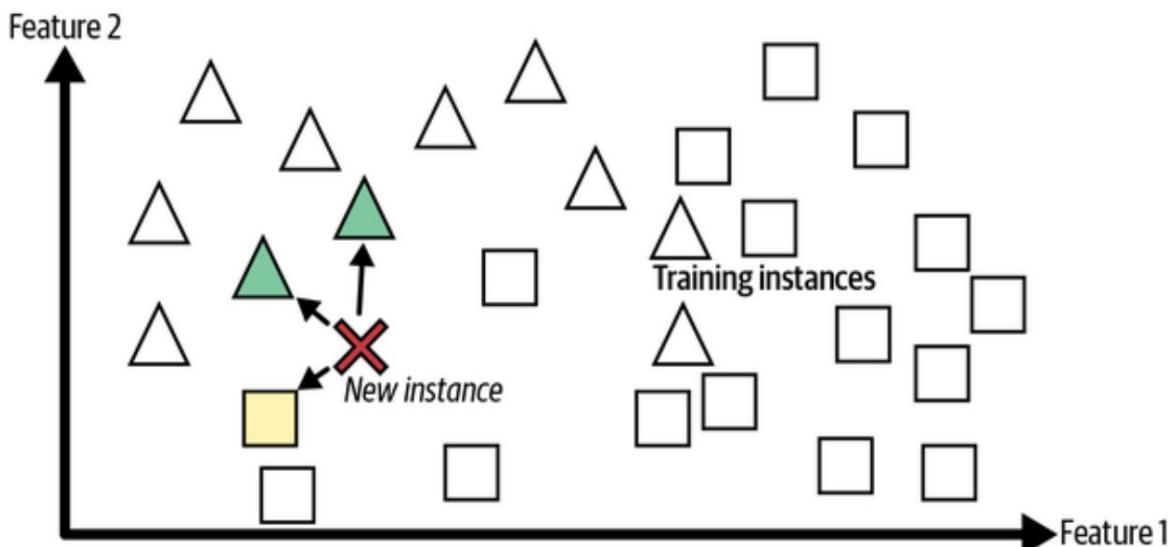


Figura 1-16. Aprendizaje basado en instancias

Aprendizaje basado en modelos y un flujo de trabajo típico de aprendizaje automático Otra forma de generalizar a partir de un conjunto de ejemplos es construir un modelo de estos ejemplos y luego usarlo para hacer predicciones. Esto se llama aprendizaje basado en modelos (Figura 1-17).

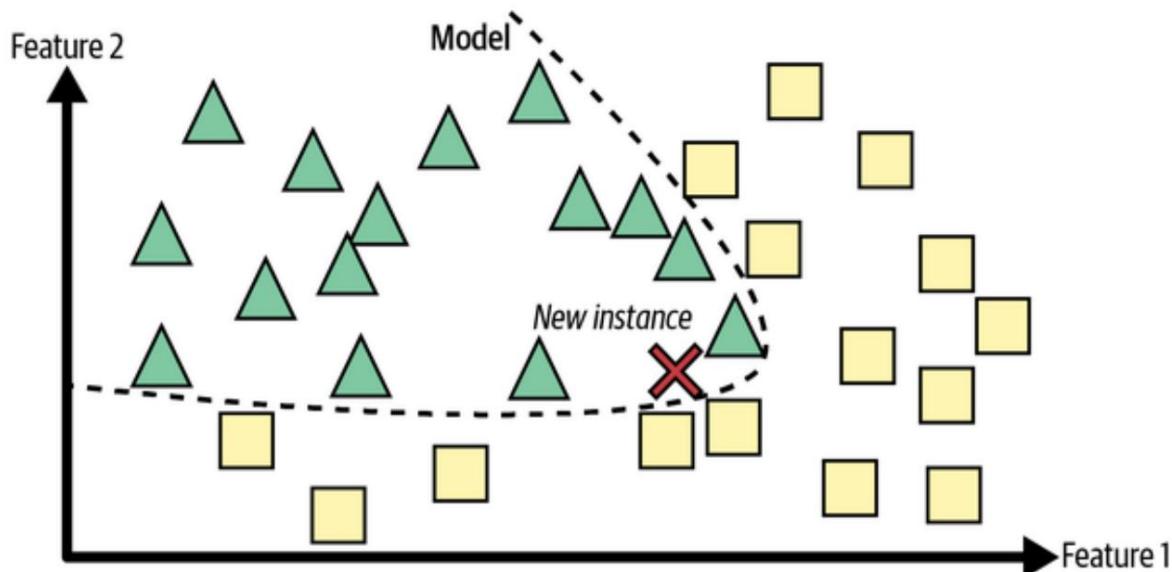


Figura 1-17. Aprendizaje basado en modelos

Por ejemplo, suponga que quiere saber si el dinero hace que la gente feliz, entonces descarga los datos del Índice de Vida Mejor de la **OCDE** sitio web y **estadísticas del Banco Mundial** sobre el producto interno bruto (PIB) per cápita. Luego unes las tablas y ordenas por PIB per cápita. La Tabla 1-1 muestra un extracto de lo que obtiene.

Tabla 1-1. ¿El dinero hace más feliz a la gente?

País	PIB per cápita (USD)	Satisfacción con la vida
Pavo	28.384	5.5
Hungría	31.008	5.6
Francia	42.026	6.5
Estados Unidos	60.236	6.9
Nueva Zelanda	42.404	7.3
Australia	48.698	7.3
Dinamarca	55.938	7.6

Tracemos los datos de estos países (Figura 1-18).

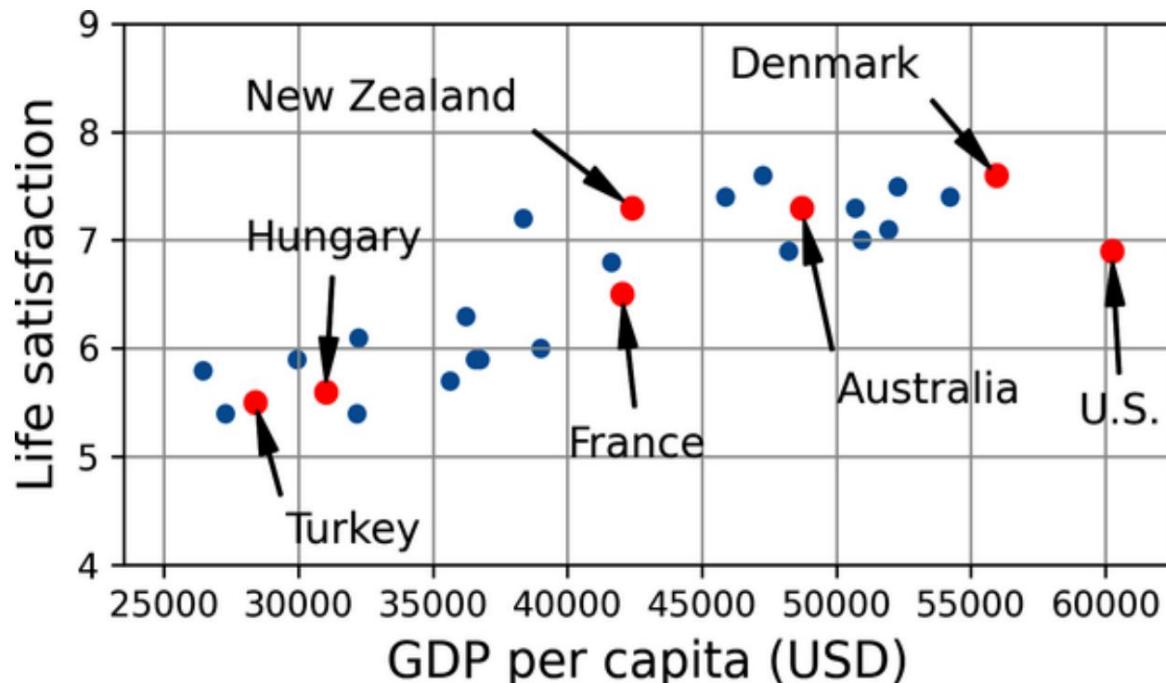


Figura 1-18. ¿Ve una tendencia aquí?

¡Parece haber una tendencia aquí! Aunque los datos son ruidosos (es decir, parcialmente aleatorios), parece que la satisfacción con la vida aumenta más o menos linealmente a medida que aumenta el PIB per cápita del país. Entonces decide modelar la satisfacción con la vida como una función lineal del PIB per cápita. Este paso se llama selección de modelo: usted seleccionó un modelo lineal de satisfacción con la vida con un solo atributo, el PIB per cápita (Ecuación 1-1).

Ecuación 1-1. Un modelo lineal simple

$$\text{satisfacción\_vida} = \theta_0 + \theta_1 \times \text{PIB\_per\_cápita}$$

Este modelo tiene dos parámetros de modelo,  $\theta_0$  y  $\theta_1$ . Al ajustar estos parámetros, puede hacer que su modelo represente cualquier función lineal, como se muestra en la Figura 1-19.

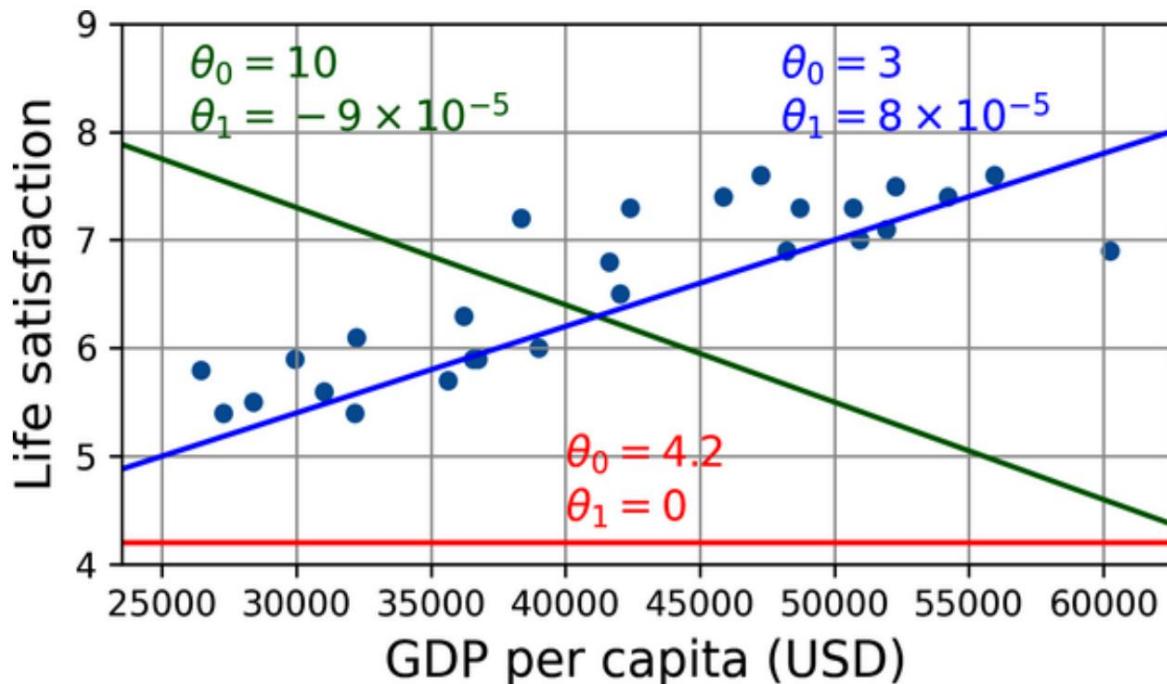


Figura 1-19. Algunos modelos lineales posibles

Antes de poder utilizar su modelo, debe definir los valores de los parámetros  $\theta_0$  y  $\theta_1$ . ¿Cómo puede saber qué valores harán que su modelo funcione mejor? Para responder a esta pregunta, es necesario especificar una medida de desempeño. Puede definir una función de utilidad (o función de aptitud) que mida qué tan bueno es su modelo, o puede definir una función de costo que mida qué tan malo es. Para los problemas de regresión lineal, la gente suele utilizar una función de costos que mide la distancia entre las predicciones del modelo lineal y los ejemplos de entrenamiento; el objetivo es minimizar esta distancia.

Aquí es donde entra en juego el algoritmo de regresión lineal: le proporciona sus ejemplos de entrenamiento y encuentra los parámetros que hacen que el modelo lineal se ajuste mejor a sus datos. A esto se le llama entrenar el modelo. En nuestro caso, el algoritmo encuentra que los valores óptimos de los parámetros son  $\theta_0 = 3,175$  y  $\theta_1 = -5,078 \times 10^{-5}$ .

### ADVERTENCIA

De manera confusa, la palabra "modelo" puede referirse a un tipo de modelo (p. ej., regresión lineal), a una arquitectura de modelo completamente especificada (p. ej., regresión lineal con una entrada y una salida) o al modelo entrenado final listo para ser utilizado. para predicciones (por ejemplo, regresión lineal con una entrada y una salida, usando  $\theta_0 = 3,75$  y  $\theta_1 = 6,78 \times 10^{-5}$ ). La selección del modelo consiste en elegir el tipo de modelo y especificar completamente su arquitectura. Entrenar un modelo significa ejecutar un algoritmo para encontrar los parámetros del modelo que harán que se ajuste mejor a los datos de entrenamiento y, con suerte, hacer buenas predicciones sobre nuevos datos.

Ahora el modelo se ajusta lo más posible a los datos de entrenamiento (para un modelo lineal), como se puede ver en [la Figura 1-20](#).

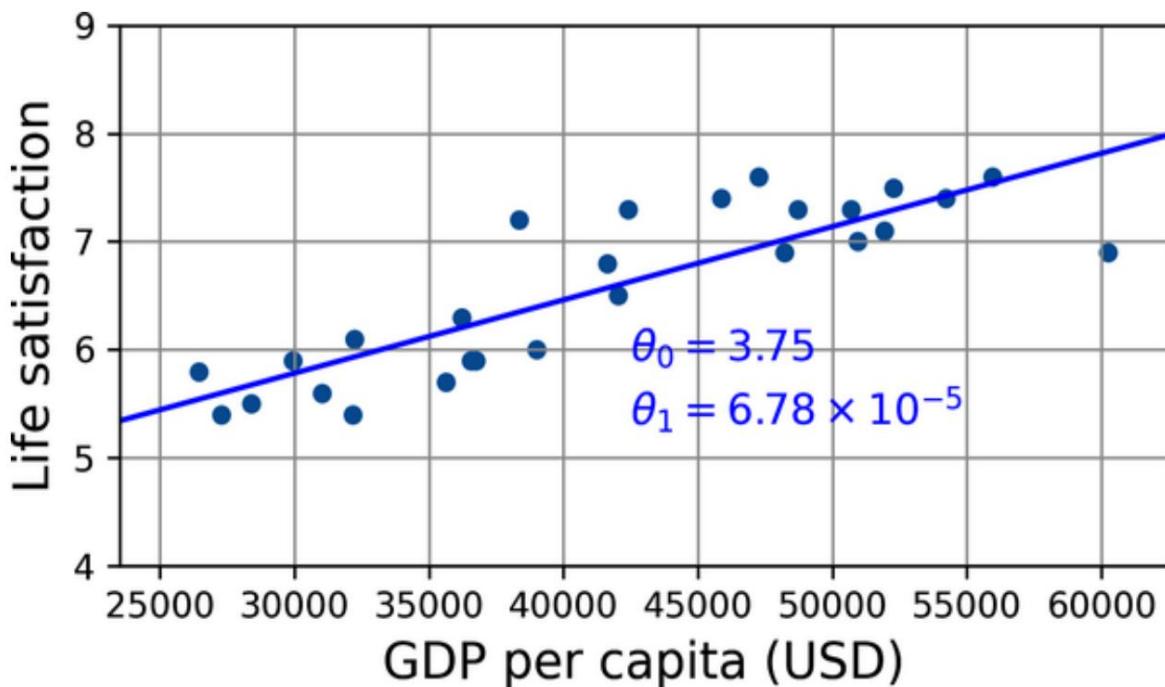


Figura 1-20. El modelo lineal que mejor se ajusta a los datos de entrenamiento.

Finalmente está listo para ejecutar el modelo para hacer predicciones. Por ejemplo, supongamos que quiere saber qué tan felices son los chipriotas y los datos de la OCDE no tienen la respuesta. Afortunadamente, puedes usar tu modelo para hacer una buena predicción: buscas el PIB per cápita de Chipre, encuentras \$37,655, luego aplicas tu modelo y encuentras que la vida

Es probable que la satisfacción esté alrededor de  $3,75 + 37.655 \times 6,78 \times -10^5 = 6,30$ .

Para abrirle el apetito, el [Ejemplo 1-1](#) muestra el código Python que carga los datos, separa las entradas X de las etiquetas y, crea un diagrama de dispersión para visualización y luego entrena un modelo lineal y hace una predicción.

[Ejemplo 1-1. Entrenar y ejecutar un modelo lineal usando Scikit-Learn](#)

```
import matplotlib.pyplot
as plt import numpy as np import pandas as pd
from sklearn.linear_model
import LinearRegression
```

```
# Descargue y prepare los datos data_root =
"https://github.com/ageron/data/raw/main/" lifesat = pd.read_csv(data_root + "lifesat/
lifesat.csv")
X = vidasat[["PIB per cápita (USD)"]].valores y = vidasat[[
Satisfacción con la vida"]].valores
```

```
# Visualizar los datos
lifesat.plot(kind='scatter', grid=True,
             x="PIB per cápita (USD)", y="Satisfacción con la vida")
plt.axis([23_500, 62_500, 4, 9]) plt.show()
```

```
# Seleccione un modelo lineal
modelo = LinearRegression()
```

```
# Entrena el modelo
modelo.fit(X, y)
```

```
# Haga una predicción para Chipre X_new =
[[37_655.2]] # PIB per cápita de Chipre en 2020 print(model.predict(X_new)) #
salida: [[6.30165767]]
```

## NOTA

Si en su lugar hubiera utilizado un algoritmo de aprendizaje basado en instancias, habría descubierto que Israel tiene el PIB per cápita más cercano al de Chipre (38.341 dólares), y dado que los datos de la OCDE nos dicen que la satisfacción con la vida de los israelíes es de 7,2, habría predijo una satisfacción con la vida de 7,2 para Chipre. Si te alejas un poco y miras los dos países más cercanos, encontrarás Lituania y Eslovenia, ambos con una satisfacción con la vida de 5,9. Al promediar estos tres valores, se obtiene 6,33, que se acerca bastante a la predicción basada en el modelo. Este algoritmo simple se llama regresión de k vecinos más cercanos (en este ejemplo, k = 3).

Reemplazar el modelo de regresión lineal con la regresión de k vecinos más cercanos en el código anterior es tan fácil como reemplazar estas líneas:

```
de sklearn.linear_model importar modelo LinearRegression =  
LinearRegression()
```

con estos dos:

```
de sklearn.neighbors importar modelo KNeighborsRegressor =  
KNeighborsRegressor(n_neighbors=3)
```

Si todo salió bien, su modelo hará buenas predicciones. De lo contrario, es posible que necesite utilizar más atributos (tasa de empleo, salud, contaminación del aire, etc.), obtener más datos de capacitación o de mejor calidad, o tal vez seleccionar un modelo más potente (por ejemplo, un modelo de regresión polinómica).

En resumen:

- Estudiaste los datos.
- Seleccionaste un modelo.
- Lo entrenó con los datos de entrenamiento (es decir, el algoritmo de aprendizaje buscó los valores de los parámetros del modelo que minimizan una función de costo).

- Finalmente, aplicó el modelo para hacer predicciones sobre casos nuevos (esto se llama inferencia), con la esperanza de que este modelo se generalice bien.

Así es como se ve un proyecto típico de aprendizaje automático. En [el Capítulo 2](#) experimentará esto de primera mano al recorrer un proyecto de principio a fin.

Hemos cubierto mucho terreno hasta ahora: ahora sabe de qué se trata realmente el aprendizaje automático, por qué es útil, cuáles son algunas de las categorías más comunes de sistemas ML y cómo es un flujo de trabajo de proyecto típico. Ahora veamos qué puede salir mal en el aprendizaje y evitar que usted haga predicciones precisas.

## Principales desafíos del aprendizaje automático

En resumen, dado que su tarea principal es seleccionar un modelo y entrenarlo con algunos datos, las dos cosas que pueden salir mal son un "modelo incorrecto" y "datos incorrectos". Comencemos con ejemplos de datos incorrectos.

### Cantidad insuficiente de datos de entrenamiento

Para que un niño pequeño aprenda qué es una manzana, todo lo que necesita es señalar una manzana y decir "manzana" (posiblemente repitiendo este procedimiento varias veces). Ahora el niño puede reconocer manzanas de todo tipo de colores y formas. Genio.

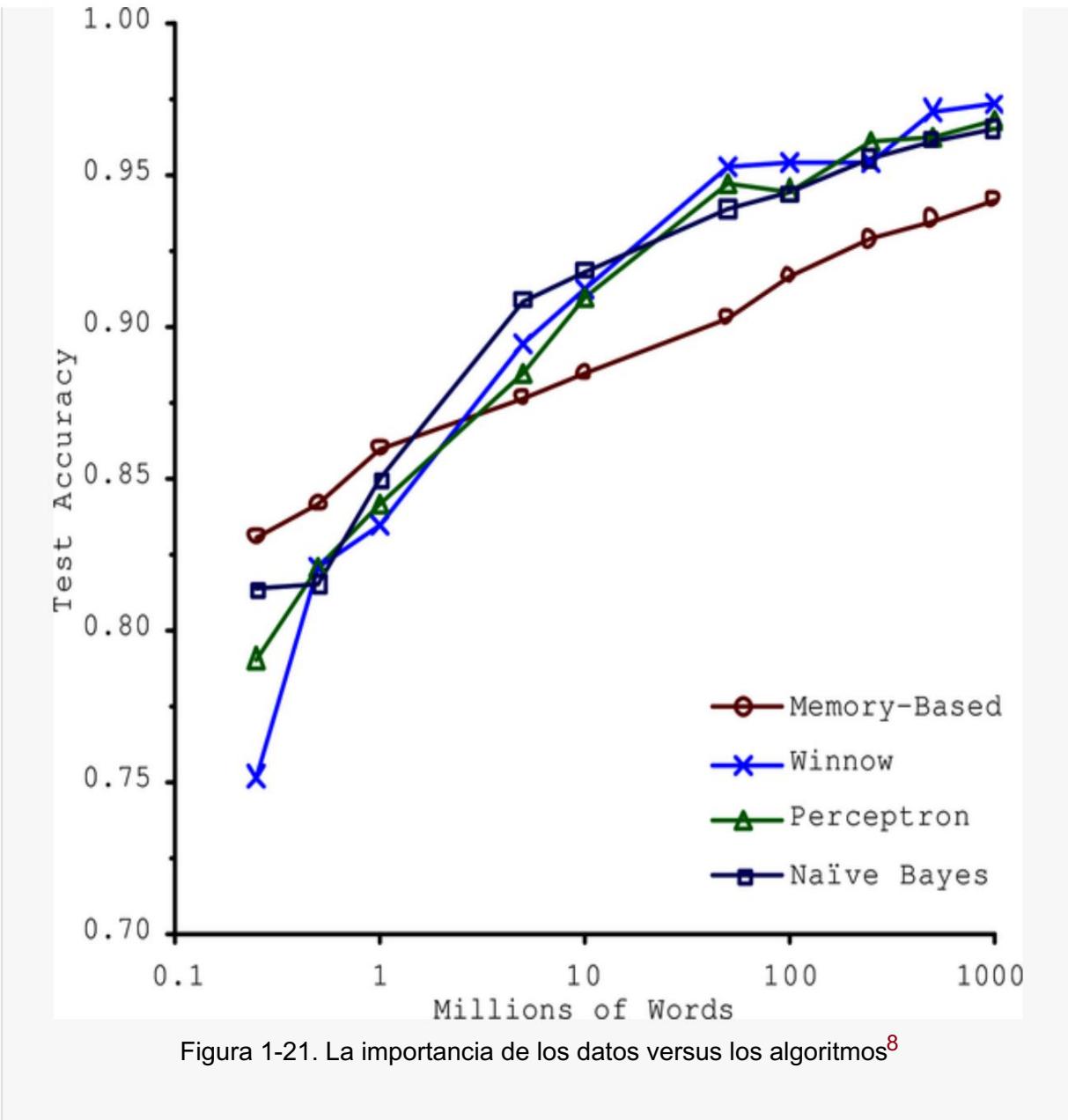
El aprendizaje automático aún no ha llegado a ese punto; Se necesitan muchos datos para que la mayoría de los algoritmos de aprendizaje automático funcionen correctamente. Incluso para problemas muy simples normalmente se necesitan miles de ejemplos, y para problemas complejos como el reconocimiento de imágenes o de voz es posible que se necesiten millones de ejemplos (a menos que se puedan reutilizar partes de un modelo existente).

## LA IRRAZONABLE EFECTIVIDAD DE LOS DATOS

En un [periódico famoso](#) publicado en 2001, investigadores de Microsoft Michele Banko y Eric Brill demostraron que algoritmos de aprendizaje automático muy diferentes, incluidos los bastante simples, funcionaban casi idénticamente bien en un problema complejo de lenguaje natural.<sup>6</sup> Una vez que se les proporcionaron suficientes datos (como se puede ver en [la Figura 1-21](#)).

Como lo expresaron los autores, "estos resultados sugieren que es posible que deseemos reconsiderar el equilibrio entre gastar tiempo y dinero en el desarrollo de algoritmos versus gastarlo en el desarrollo de corpus".

La idea de que los datos importan más que los algoritmos para problemas complejos fue popularizada aún más por Peter Norvig et al. en un artículo titulado "[La eficacia irrazonable de los datos](#)", publicado en 2009. Cabe señalar, sin embargo,<sup>7</sup> que los conjuntos de datos pequeños y medianos siguen siendo muy comunes, y no siempre es fácil o barato obtener datos de entrenamiento adicionales, así que no abandone los algoritmos todavía.



## Datos de formación no representativos

Para poder generalizar bien, es fundamental que los datos de entrenamiento sean representativos de los nuevos casos a los que desea generalizar. Esto es cierto ya sea que utilice el aprendizaje basado en instancias o el aprendizaje basado en modelos.

Por ejemplo, el conjunto de países que utilizó anteriormente para entrenar el modelo lineal no era perfectamente representativo; no contenía ningún país con un PIB per cápita inferior a 23.500 dólares o superior a 62.500 dólares. La Figura 1-22 muestra cómo se ven los datos cuando se agregan dichos países.

Si entrena un modelo lineal con estos datos, obtendrá la línea continua, mientras que el modelo anterior está representado por la línea de puntos. Como puede ver, agregar algunos países faltantes no sólo altera significativamente el modelo, sino que deja claro que un modelo lineal tan simple probablemente nunca funcionará bien. Parece que los países muy ricos no son más felices que los moderadamente ricos (¡de hecho, parecen un poco más infelices!) y, a la inversa, algunos países pobres parecen más felices que muchos países ricos.

Al utilizar un conjunto de entrenamiento no representativo, entrenó un modelo que probablemente no haga predicciones precisas, especialmente para países muy pobres y muy ricos.

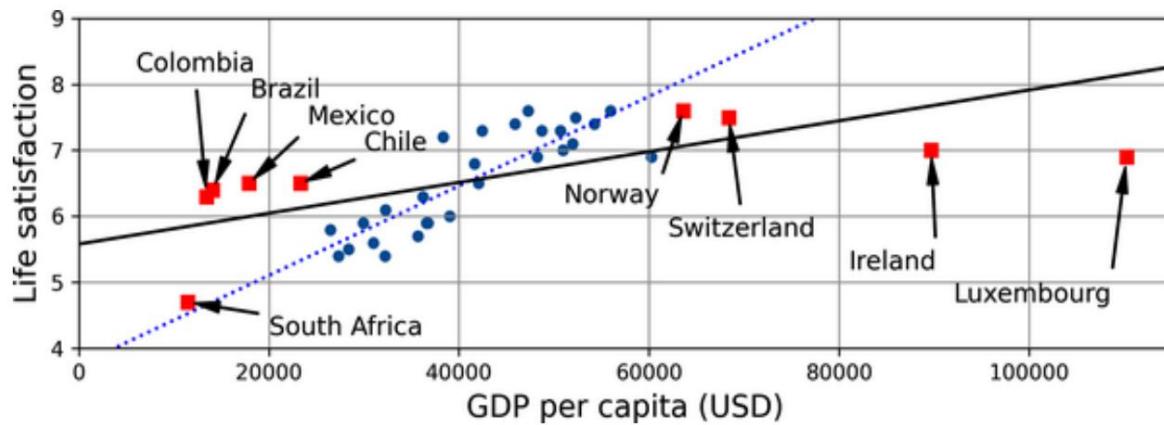


Figura 1-22. Una muestra de formación más representativa

Es fundamental utilizar un conjunto de entrenamiento que sea representativo de los casos a los que desea generalizar. Esto suele ser más difícil de lo que parece: si la muestra es demasiado pequeña, habrá ruido de muestreo (es decir, datos no representativos como resultado del azar), pero incluso muestras muy grandes pueden ser no representativas si el método de muestreo es defectuoso. A esto se le llama sesgo de muestreo.

## EJEMPLOS DE SESGO DE MUESTREO

Quizás el ejemplo más famoso de sesgo de muestreo ocurrió durante las elecciones presidenciales estadounidenses de 1936, que enfrentaron a Landon contra Roosevelt: el Literary Digest realizó una encuesta muy grande, enviando correo a unos 10 millones de personas. Obtuvo 2,4 millones de respuestas y predijo con gran confianza que Landon obtendría el 57% de los votos. En cambio, Roosevelt ganó con el 62% de los votos. El error estaba en el método de muestreo del Literary Digest :

- En primer lugar, para obtener las direcciones a las que enviar las encuestas, el Literary Digest utilizó guías telefónicas, listas de suscriptores de revistas, listas de miembros de clubes y similares. Todas estas listas tendían a favorecer a las personas más ricas, que tenían más probabilidades de votar por los republicanos (de ahí a Landon).
- En segundo lugar, menos del 25% de las personas encuestadas respondieron. Una vez más, esto introdujo un sesgo de muestreo, al descartar potencialmente a personas a las que no les importaba mucho la política, personas a las que no les gustaba el Literary Digest y otros grupos clave. Este es un tipo especial de sesgo de muestreo llamado sesgo de falta de respuesta.

Aquí hay otro ejemplo: supongamos que desea crear un sistema para reconocer videos musicales funk. Una forma de crear tu conjunto de entrenamiento es buscar “música funk” en YouTube y utilizar los videos resultantes. Pero esto supone que el motor de búsqueda de YouTube devuelve un conjunto de videos que son representativos de todos los videos de música funk de YouTube. En realidad, es probable que los resultados de búsqueda estén sesgados hacia artistas populares (y si vives en Brasil encontrarás muchos videos de “funk carioca”, que no se parecen en nada a James Brown).

Por otro lado, ¿de qué otra manera se puede conseguir un gran conjunto de entrenamiento?

## Datos de mala calidad

Obviamente, si sus datos de entrenamiento están llenos de errores, valores atípicos y ruido (por ejemplo, debido a mediciones de mala calidad), al sistema le resultará más difícil detectar los patrones subyacentes, por lo que es menos probable que su sistema funcione bien. A menudo vale la pena dedicar tiempo a limpiar los datos de entrenamiento. La verdad es que la mayoría de los científicos de datos dedican una parte importante de su tiempo a hacer precisamente eso. Los siguientes son un par de ejemplos de cuándo desearía limpiar los datos de entrenamiento:

- Si algunos casos son claramente atípicos, puede ser útil simplemente descartarlos o intentar corregir los errores manualmente.
- Si a algunos casos les faltan algunas características (por ejemplo, el 5 % de sus clientes no especificaron su edad), debe decidir si desea ignorar este atributo por completo, ignorar estos casos, completar los valores faltantes (por ejemplo, con la mediana edad), o entrenar un modelo con la característica y un modelo sin ella.

## Características irrelevantes

Como dice el refrán: basura entra, basura sale. Su sistema sólo será capaz de aprender si los datos de entrenamiento contienen suficientes características relevantes y no demasiadas irrelevantes. Una parte fundamental del éxito de un proyecto de aprendizaje automático es crear un buen conjunto de funciones sobre las que entrenar. Este proceso, llamado ingeniería de características, implica los siguientes pasos:

- Selección de funciones (seleccionar las funciones más útiles para entrenar entre las funciones existentes)
- Extracción de características (combinar características existentes para producir una más útil; como vimos anteriormente, los algoritmos de reducción de dimensionalidad pueden ayudar)
- Crear nuevas funciones recopilando nuevos datos

Ahora que hemos visto muchos ejemplos de datos incorrectos, veamos un par de ejemplos de algoritmos incorrectos.

## Sobreajustar los datos de entrenamiento

Digamos que estás visitando un país extranjero y el taxista te estafa.

Podría sentirse tentado a decir que todos los taxistas de ese país son ladrones. Generalizar demasiado es algo que los humanos hacemos con demasiada frecuencia y, lamentablemente, las máquinas pueden caer en la misma trampa si no tenemos cuidado. En aprendizaje automático, esto se llama sobreajuste: significa que el modelo funciona bien con los datos de entrenamiento, pero no se generaliza bien.

**La Figura 1-23** muestra un ejemplo de un modelo polinómico de satisfacción con la vida de alto grado que sobreajusta fuertemente los datos de entrenamiento. Aunque funciona mucho mejor con los datos de entrenamiento que el modelo lineal simple, ¿realmente confiaría en sus predicciones?

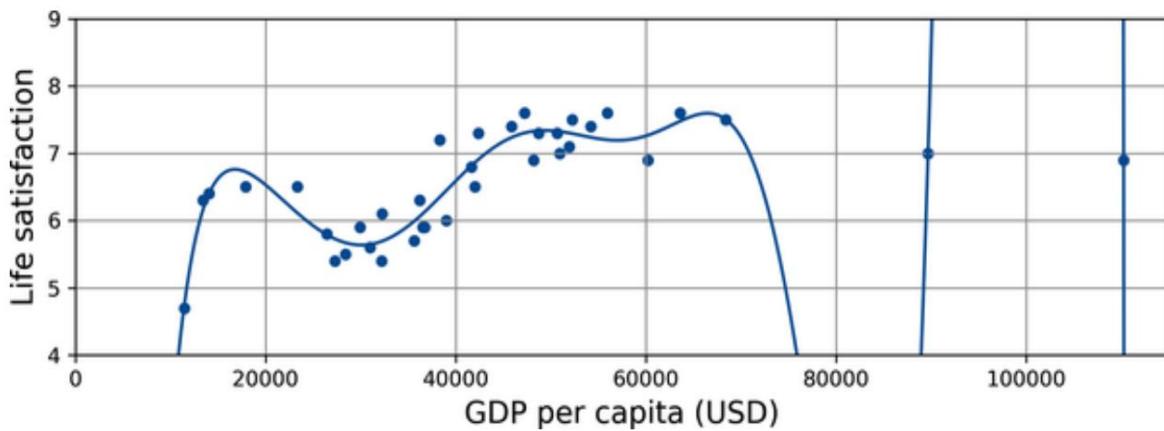


Figura 1-23. Sobreajustar los datos de entrenamiento

Los modelos complejos, como las redes neuronales profundas, pueden detectar patrones sutiles en los datos, pero si el conjunto de entrenamiento es ruidoso o si es demasiado pequeño, lo que introduce ruido de muestreo, entonces es probable que el modelo detecte patrones en el ruido mismo (como en el ejemplo del taxista). Obviamente estos patrones no se generalizarán a nuevos casos. Por ejemplo, supongamos que alimenta su modelo de satisfacción con la vida con muchos más atributos, incluidos los poco informativos, como el nombre del país. En ese caso, un modelo complejo puede detectar patrones como el hecho de que todos los países en los datos de entrenamiento con una w en su nombre tienen una satisfacción con la vida superior a 7: Nueva Zelanda (7,3), Noruega (7,6), Suecia (7,3), y

Suiza (7,5). ¿Qué tan seguro está de que la regla de satisfacción w se generalice a Ruanda o Zimbabwe? Obviamente, este patrón ocurrió en los datos de entrenamiento por pura casualidad, pero el modelo no tiene forma de saber si un patrón es real o simplemente el resultado del ruido en los datos.

#### ADVERTENCIA

El sobreajuste ocurre cuando el modelo es demasiado complejo en relación con la cantidad y el ruido de los datos de entrenamiento. Aquí hay posibles soluciones:

- Simplifique el modelo seleccionando uno con menos parámetros (por ejemplo, un modelo lineal en lugar de un modelo polinomial de alto grado), reduciendo la cantidad de atributos en los datos de entrenamiento o restringiendo el modelo.
- Reúna más datos de entrenamiento.
- Reducir el ruido en los datos de entrenamiento (por ejemplo, corregir errores de datos y eliminar valores atípicos).

Restringir un modelo para hacerlo más simple y reducir el riesgo de sobreajuste se llama regularización. Por ejemplo, el modelo lineal que definimos anteriormente tiene dos parámetros,  $\theta_0$  y  $\theta_1$ . Esto le da al algoritmo de aprendizaje dos grados de libertad para adaptar el modelo a los datos de entrenamiento: puede modificar tanto la altura ( $\theta_0$ ) como la pendiente ( $\theta_1$ ) de la línea. Si forzamos  $\theta_0 = 0$ , el algoritmo tendría solo un grado de libertad y le resultaría mucho más difícil ajustar los datos correctamente: todo lo que podría hacer es mover la línea hacia arriba o hacia abajo para acercarse lo más posible a las instancias de entrenamiento. entonces terminaría alrededor de la media. ¡Un modelo muy simple por cierto! Si permitimos que el algoritmo modifique  $\theta_0$  pero lo obligamos a mantenerlo pequeño, entonces el algoritmo de aprendizaje tendrá efectivamente entre uno y dos grados de libertad. Producirá un modelo que es más simple que uno con dos grados de libertad, pero más complejo que un

Quiere encontrar el equilibrio adecuado entre ajustar perfectamente los datos de entrenamiento y mantener el modelo lo suficientemente simple como para garantizar que se generalice bien.

**La Figura 1-24** muestra tres modelos. La línea de puntos representa el modelo original que se entrenó en los países representados como círculos (sin los países representados como cuadrados), la línea continua es nuestro segundo modelo entrenado con todos los países (círculos y cuadrados) y la línea discontinua es un modelo entrenado con los mismos datos que el primer modelo pero con una restricción de regularización.

Puede ver que la regularización obligó al modelo a tener una pendiente menor: este modelo no se ajusta a los datos de entrenamiento (círculos) tan bien como el primer modelo, pero en realidad se generaliza mejor a nuevos ejemplos que no vio durante el entrenamiento (cuadrados). .

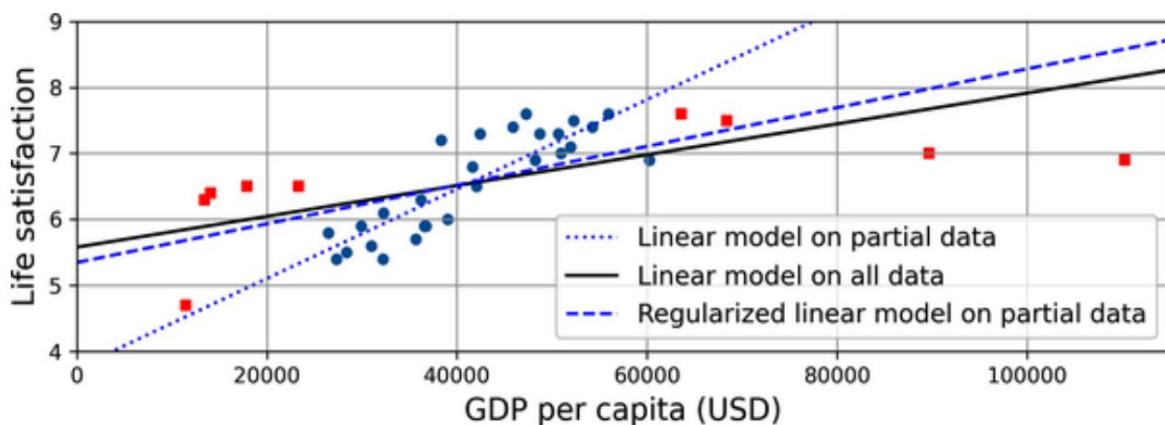


Figura 1-24. La regularización reduce el riesgo de sobreajuste

La cantidad de regularización que se aplicará durante el aprendizaje se puede controlar mediante un hiperparámetro. Un hiperparámetro es un parámetro de un algoritmo de aprendizaje (no del modelo). Como tal, no se ve afectado por el propio algoritmo de aprendizaje; debe establecerse antes del entrenamiento y permanece constante durante el entrenamiento. Si establece el hiperparámetro de regularización en un valor muy grande, obtendrá un modelo casi plano (una pendiente cercana a cero); Es casi seguro que el algoritmo de aprendizaje no se ajustará demasiado a los datos de entrenamiento, pero será menos probable que encuentre una buena solución. El ajuste de los hiperparámetros es una parte importante de

construir un sistema de aprendizaje automático (verá un ejemplo detallado en el siguiente capítulo).

### Ajuste insuficiente de los datos de entrenamiento Como

puede imaginar, el ajuste insuficiente es lo opuesto al sobreajuste: ocurre cuando su modelo es demasiado simple para aprender la estructura subyacente de los datos. Por ejemplo, un modelo lineal de satisfacción con la vida tiende a no adaptarse; La realidad es simplemente más compleja que el modelo, por lo que sus predicciones seguramente serán inexactas, incluso en los ejemplos de entrenamiento.

Estas son las principales opciones para solucionar este problema:

- Seleccione un modelo más potente, con más parámetros.
- Introducir mejores funciones en el algoritmo de aprendizaje (ingeniería de funciones).
- Reducir las restricciones del modelo (por ejemplo, reduciendo el hiperparámetro de regularización).

## Dar un paso atrás

A estas alturas ya sabes mucho sobre el aprendizaje automático. Sin embargo, analizamos tantos conceptos que es posible que te sientas un poco perdido, así que retrocedamos y miremos el panorama general:

- El aprendizaje automático consiste en hacer que las máquinas mejoren en alguna tarea aprendiendo de los datos, en lugar de tener que codificar reglas explícitamente.
- Hay muchos tipos diferentes de sistemas de aprendizaje automático: supervisados o no, por lotes o en línea, basados en instancias o basados en modelos.
- En un proyecto de ML, usted recopila datos en un conjunto de capacitación y alimenta el conjunto de capacitación a un algoritmo de aprendizaje. Si el algoritmo está basado en modelos, ajusta algunos parámetros para adaptar el modelo al entrenamiento.

conjunto (es decir, hacer buenas predicciones sobre el conjunto de entrenamiento en sí) y luego, con suerte, también podrá hacer buenas predicciones sobre casos nuevos. Si el algoritmo está basado en instancias, simplemente aprende los ejemplos de memoria y los generaliza a nuevas instancias utilizando una medida de similitud para compararlas con las instancias aprendidas.

- El sistema no funcionará bien si su conjunto de entrenamiento es demasiado pequeño, o si los datos no son representativos, tienen ruido o están contaminados con características irrelevantes (basura que entra, basura sale). Por último, su modelo no debe ser ni demasiado simple (en cuyo caso no se ajustará) ni demasiado complejo (en cuyo caso se ajustará demasiado).

Solo queda un último tema importante que cubrir: una vez que haya entrenado un modelo, no querrá simplemente “esperar” que se generalice a nuevos casos. Quieres evaluarlo y ajustarlo si es necesario. Veamos cómo hacerlo.

## Pruebas y Validaciones

La única forma de saber qué tan bien se generalizará un modelo a casos nuevos es probarlo en casos nuevos. Una forma de hacerlo es poner su modelo en producción y monitorear su rendimiento. Esto funciona bien, pero si su modelo es terriblemente malo, sus usuarios se quejarán; no es la mejor idea.

Una mejor opción es dividir los datos en dos conjuntos: el conjunto de entrenamiento y el conjunto de prueba. Como lo implican estos nombres, usted entrena su modelo usando el conjunto de entrenamiento y lo prueba usando el conjunto de prueba. La tasa de error en casos nuevos se denomina error de generalización (o error fuera de muestra) y, al evaluar su modelo en el conjunto de prueba, obtiene una estimación de este error. Este valor le indica qué tan bien funcionará su modelo en instancias que nunca antes había visto.

Si el error de entrenamiento es bajo (es decir, su modelo comete pocos errores en el conjunto de entrenamiento) pero el error de generalización es alto, significa que

su modelo está sobreajustando los datos de entrenamiento.

## CONSEJO

Es común utilizar el 80% de los datos para entrenamiento y reservar el 20% para pruebas. Sin embargo, esto depende del tamaño del conjunto de datos: si contiene 10 millones de instancias, entonces reservar el 1% significa que su conjunto de prueba contendrá 100.000 instancias, probablemente más que suficiente para obtener una buena estimación del error de generalización.

## Ajuste de hiperparámetros y selección de modelo

Evaluar un modelo es bastante simple: basta con utilizar un conjunto de prueba. Pero supongamos que usted duda entre dos tipos de modelos (digamos, un modelo lineal y un modelo polinomial): ¿cómo puede decidir entre ellos? Una opción es entrenar a ambos y comparar qué tan bien se generalizan usando el conjunto de prueba.

Ahora supongamos que el modelo lineal se generaliza mejor, pero desea aplicar cierta regularización para evitar el sobreajuste. La pregunta es, ¿cómo se elige el valor del hiperparámetro de regularización?

Una opción es entrenar 100 modelos diferentes utilizando 100 valores diferentes para este hiperparámetro. Supongamos que encuentra el mejor valor de hiperparámetro que produce un modelo con el error de generalización más bajo; digamos, solo un error del 5 %. Lanzas este modelo a producción, pero lamentablemente no funciona tan bien como se esperaba y produce un 15% de errores. ¿Lo que acaba de suceder?

El problema es que midió el error de generalización varias veces en el conjunto de prueba y adaptó el modelo y los hiperparámetros para producir el mejor modelo para ese conjunto en particular.

Esto significa que es poco probable que el modelo funcione tan bien con datos nuevos.

Una solución común a este problema se llama validación de exclusión ([Figura 1-25](#)): simplemente se reserva parte del conjunto de entrenamiento para evaluar varios modelos candidatos y seleccionar el mejor. El nuevo retenido

El conjunto se llama conjunto de validación (o conjunto de desarrollo, o conjunto de prueba).  
Más específicamente, entrena varios modelos con varios hiperparámetros en el conjunto de entrenamiento reducido (es decir, el conjunto de entrenamiento completo menos el conjunto de validación) y selecciona el modelo que funciona mejor en el conjunto de validación. Después de este proceso de validación de reserva, usted entrena el mejor modelo en el conjunto de entrenamiento completo (incluido el conjunto de validación) y esto le brinda el modelo final. Por último, se evalúa este modelo final en el conjunto de prueba para obtener una estimación del error de generalización.

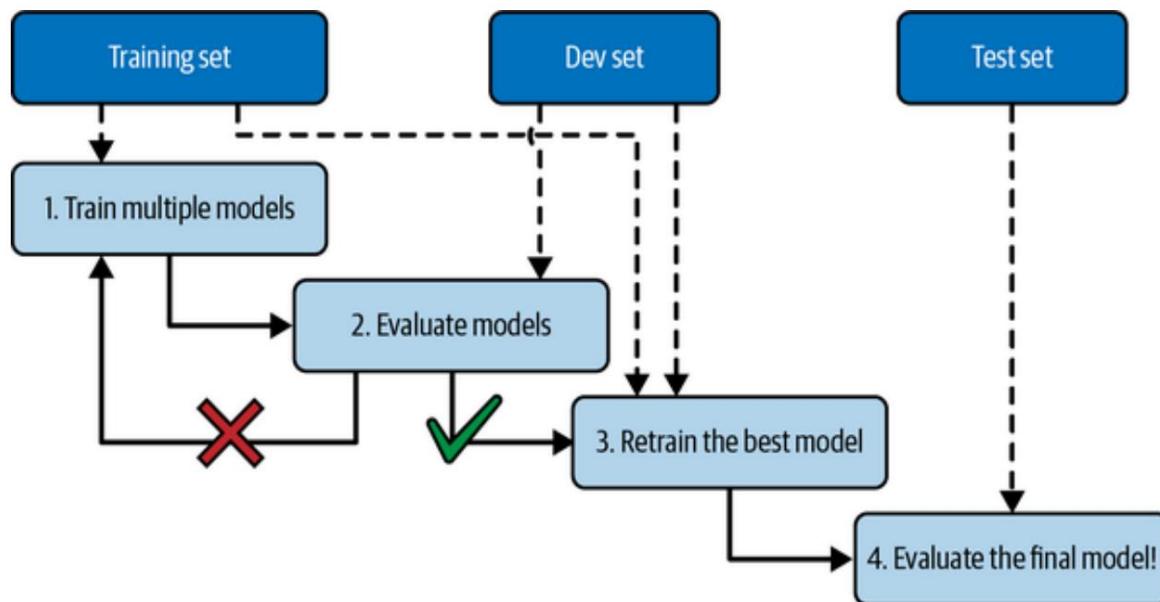


Figura 1-25. Selección de modelo mediante validación de reserva

Esta solución suele funcionar bastante bien. Sin embargo, si el conjunto de validación es demasiado pequeño, las evaluaciones del modelo serán imprecisas: puede terminar seleccionando un modelo subóptimo por error. Por el contrario, si el conjunto de validación es demasiado grande, el conjunto de entrenamiento restante será mucho más pequeño que el conjunto de entrenamiento completo. ¿Por qué es esto malo? Bueno, dado que el modelo final se entrenará en el conjunto de entrenamiento completo, no es ideal comparar modelos candidatos entrenados en un conjunto de entrenamiento mucho más pequeño. Sería como seleccionar al velocista más rápido para participar en una maratón. Una forma de resolver este problema es realizar validaciones cruzadas repetidas, utilizando muchos conjuntos de validación pequeños. Cada modelo se evalúa una vez por conjunto de validación después de entrenarse con el resto de los datos. Al promediar todas las evaluaciones de un modelo, se obtiene una

una medida mucho más precisa de su desempeño. Sin embargo, existe un inconveniente: el tiempo de entrenamiento se multiplica por el número de conjuntos de validación.

## No coinciden los datos

En algunos casos, es fácil obtener una gran cantidad de datos para el entrenamiento, pero estos datos probablemente no serán perfectamente representativos de los datos que se utilizarán en producción. Por ejemplo, supongamos que desea crear una aplicación móvil para tomar fotografías de flores y determinar automáticamente sus especies. Puedes descargar fácilmente millones de imágenes de flores en la web, pero no serán perfectamente representativas de las imágenes que realmente se tomarán usando la aplicación en un dispositivo móvil. Quizás sólo tengas 1000 fotografías representativas (es decir, tomadas realmente con la aplicación).

En este caso, la regla más importante a recordar es que tanto el conjunto de validación como el conjunto de prueba deben ser lo más representativos posible de los datos que espera utilizar en producción, por lo que deben estar compuestos exclusivamente por imágenes representativas: puede mezclarlas y coloque la mitad en el conjunto de validación y la otra mitad en el conjunto de prueba (asegurándose de que no terminen duplicados o casi duplicados en ambos conjuntos). Después de entrenar su modelo en las imágenes web, si observa que el rendimiento del modelo en el conjunto de validación es decepcionante, no sabrá si esto se debe a que su modelo se ha sobreajustado al conjunto de entrenamiento o si se debe simplemente a una falta de coincidencia entre las imágenes web y las imágenes de la aplicación móvil.

Una solución es mostrar algunas de las imágenes de entrenamiento (de la web) en otro conjunto que Andrew Ng denominó conjunto tren-desarrollo ([Figura 1-26](#)). Una vez entrenado el modelo (en el conjunto de entrenamiento, no en el conjunto de desarrollo de entrenamiento), puede evaluarlo en el conjunto de desarrollo de entrenamiento. Si el modelo funciona mal, entonces debe haber sobreajustado el conjunto de entrenamiento, por lo que debe intentar simplificar o regularizar el modelo, obtener más datos de entrenamiento y limpiarlos. Pero si funciona bien en el

train-dev set, luego puede evaluar el modelo en el conjunto de desarrollo. Si funciona mal, entonces el problema debe deberse a que los datos no coinciden. Puede intentar solucionar este problema preprocesando las imágenes web para que se parezcan más a las imágenes que tomará la aplicación móvil y luego volviendo a entrenar el modelo. Una vez que tenga un modelo que funcione bien tanto en el conjunto de desarrollo como en el conjunto de desarrollo, puede evaluarlo por última vez en el conjunto de prueba para saber qué tan bien es probable que funcione en producción.



Figura 1-26. Cuando los datos reales son escasos (derecha), puede utilizar datos abundantes similares (izquierda) para el entrenamiento y conservar algunos de ellos en un conjunto de desarrollo de tren para evaluar el sobreajuste; Luego, los datos reales se utilizan para evaluar la discrepancia de datos (conjunto de desarrollo) y para evaluar el rendimiento del modelo final (conjunto de prueba).

## TEOREMA DEL ALMUERZO GRATIS

Un modelo es una representación simplificada de los datos. Las simplificaciones pretenden descartar los detalles superfluos que es poco probable que se generalicen a nuevos casos. Cuando selecciona un tipo particular de modelo, implícitamente está haciendo suposiciones sobre los datos. Por ejemplo, si elige un modelo lineal, está asumiendo implícitamente que los datos son fundamentalmente lineales y que la distancia entre las instancias y la línea recta es solo ruido, que puede ignorarse con seguridad.

En un [famoso artículo de 1996](#), David Wolpert demostró que si no se hace ninguna suposición sobre los datos, entonces no hay razón para preferir un modelo sobre otro. Esto se llama el teorema del almuerzo gratis (NFL). Para algunos conjuntos de datos, el mejor modelo es un modelo lineal, mientras que para otros conjuntos de datos es una red neuronal. No existe ningún modelo que a priori garantice que funcione mejor (de ahí el nombre del teorema). La única forma de saber con seguridad qué modelo es mejor es evaluarlos todos. Como esto no es posible, en la práctica se hacen algunas suposiciones razonables sobre los datos y se evalúan sólo unos pocos modelos razonables. Por ejemplo, para tareas simples puedes evaluar modelos lineales con varios niveles de regularización y para un problema complejo puedes evaluar varias redes neuronales.

## Ejercicios

En este capítulo hemos cubierto algunos de los conceptos más importantes del aprendizaje automático. En los próximos capítulos profundizaremos y escribiremos más código, pero antes de hacerlo, asegúrese de que puede responder las siguientes preguntas:

1. ¿Cómo definirías el aprendizaje automático?
2. ¿Puedes nombrar cuatro tipos de aplicaciones en las que destaca?

3. ¿Qué es un conjunto de entrenamiento etiquetado?
4. ¿Cuáles son las dos tareas supervisadas más comunes?
5. ¿Puedes nombrar cuatro tareas comunes no supervisadas?
6. ¿Qué tipo de algoritmo usarías para permitir que un robot camine por varios terrenos desconocidos?
7. ¿Qué tipo de algoritmo usarías para segmentar tu clientes en varios grupos?
8. ¿Enmarcaría el problema de la detección de spam como un problema de aprendizaje supervisado o problema de aprendizaje no supervisado?
9. ¿Qué es un sistema de aprendizaje en línea?
10. ¿Qué es el aprendizaje fuera del núcleo?
11. ¿Qué tipo de algoritmo se basa en una medida de similitud para hacer predicciones?
12. ¿Cuál es la diferencia entre un parámetro de modelo y un hiperparámetro de modelo?
13. ¿Qué buscan los algoritmos basados en modelos? ¿Cuál es la estrategia más común que utilizan para tener éxito? ¿Cómo hacen predicciones?
14. ¿Puedes nombrar cuatro de los principales desafíos del aprendizaje automático?
15. Si su modelo funciona muy bien con los datos de entrenamiento pero no se generaliza bien a nuevas instancias, ¿qué está sucediendo? ¿Puedes nombrar tres posibles soluciones?
16. ¿Qué es un equipo de prueba y por qué querrías usarlo?
17. ¿Cuál es el propósito de un conjunto de validación?

18. ¿Qué es el conjunto de desarrollo de trenes, cuándo lo necesita y cómo lo necesita usarlo?

19. ¿Qué puede salir mal si ajustas los hiperparámetros mediante la prueba?

¿colocar?

Las soluciones a estos ejercicios están disponibles al final del cuaderno de este capítulo, en <https://homl.info/colab3>.

---

**1** Dato curioso: este nombre que suena extraño es un término estadístico introducido por Francis Galton mientras estudiaba el hecho de que los hijos de personas altas tienden a ser más bajos que sus padres. Como los niños eran más bajos, llamó a esta regresión a la media. Este nombre se aplicó luego a los métodos que utilizó para analizar correlaciones entre variables.

**2** Observe cómo los animales están bastante bien separados de los vehículos y cómo los caballos están cerca de los ciervos pero lejos de las aves. Figura reproducida con autorización de Richard Socher et al., “Zero-Shot Learning Through Cross-Modal Transfer”, Actas de la 26<sup>a</sup> Conferencia Internacional sobre Sistemas de Procesamiento de Información Neural 1 (2013): 935–943.

**3** Entonces es cuando el sistema funciona perfectamente. En la práctica, a menudo crea algunos grupos por persona y, a veces, mezcla a dos personas que se parecen, por lo que es posible que deba proporcionar algunas etiquetas por persona y limpiar manualmente algunos grupos.

**4** Por convención, la letra griega  $\theta$  (theta) se utiliza frecuentemente para representar parámetros del modelo.

**5** Está bien si aún no comprende todo el código; Presentaré Scikit-Learn en los siguientes capítulos.

**6** Por ejemplo, saber si escribir “a”, “dos” o “también”, dependiendo de el contexto.

**7** Peter Norvig et al., “La eficacia irrazonable de los datos”, IEEE Sistemas Inteligentes 24, núm. 2 (2009): 8-12.

**8** Figura reproducida con autorización de Michele Banko y Eric Brill, “Escalado a corpus muy grandes para la desambiguación del lenguaje natural”, Actas de la 39<sup>a</sup> reunión anual de la Asociación de Lingüística Computacional (2001): 26–33.

**9** David Wolpert, “La falta de distinciones a priori entre algoritmos de aprendizaje”, *Computación neuronal* 8, no. 7 (1996): 1341-1390.

## Capítulo 2. Proyecto de aprendizaje automático de un extremo a otro

---

En este capítulo, trabajará en un proyecto de ejemplo de principio a fin, pretendiendo ser un científico de datos contratado recientemente en una empresa de bienes raíces.

Este ejemplo es ficticio; El objetivo es ilustrar los pasos principales de un proyecto de aprendizaje automático, no aprender nada sobre el negocio inmobiliario. Estos son los pasos principales que seguiremos:

1. Mire el panorama general.
2. Obtenga los datos.
3. Explore y visualice los datos para obtener información.
4. Prepare los datos para los algoritmos de aprendizaje automático.
5. Seleccione un modelo y entrénelo.
6. Ajusta tu modelo.
7. Presente su solución.
8. Inicie, supervise y mantenga su sistema.

### Trabajar con datos reales

Cuando aprende sobre aprendizaje automático, es mejor experimentar con datos del mundo real, no con conjuntos de datos artificiales. Afortunadamente, hay miles de conjuntos de datos abiertos para elegir, que abarcan todo tipo de dominios. Aquí hay algunos lugares donde puede buscar para obtener datos:

- Repositorios de datos abiertos populares:
  - [OpenML.org](#)
  - [Kaggle.com](#)
  - [PapelesConCode.com](#)

- [Repositorio de aprendizaje automático de UC Irvine](#)
- [Conjuntos de datos de AWS de Amazon](#)
- [Conjuntos de datos de TensorFlow](#)
- Metaportales (enumeran repositorios de datos abiertos):
  - [PortalesdeDatos.org](#)
  - [OpenDataMonitor.eu](#)
- Otras páginas que enumeran muchos repositorios de datos abiertos populares:
  - [Lista de Wikipedia de conjuntos de datos de aprendizaje automático](#)
  - [Quora.com](#)
  - [El subreddit de conjuntos de datos.](#)

En este capítulo usaremos el conjunto de datos de Precios de Vivienda de California del <sup>1</sup> Repositorio StatLib (ver Figura 2-1). Este conjunto de datos se basa en datos del censo de California de 1990. No es exactamente reciente (una bonita casa en el Área de la Bahía todavía era asequible en ese momento), pero tiene muchas cualidades para aprender, por lo que haremos como si fueran datos recientes. Para fines didácticos, agregué un atributo categórico y eliminé algunas funciones.

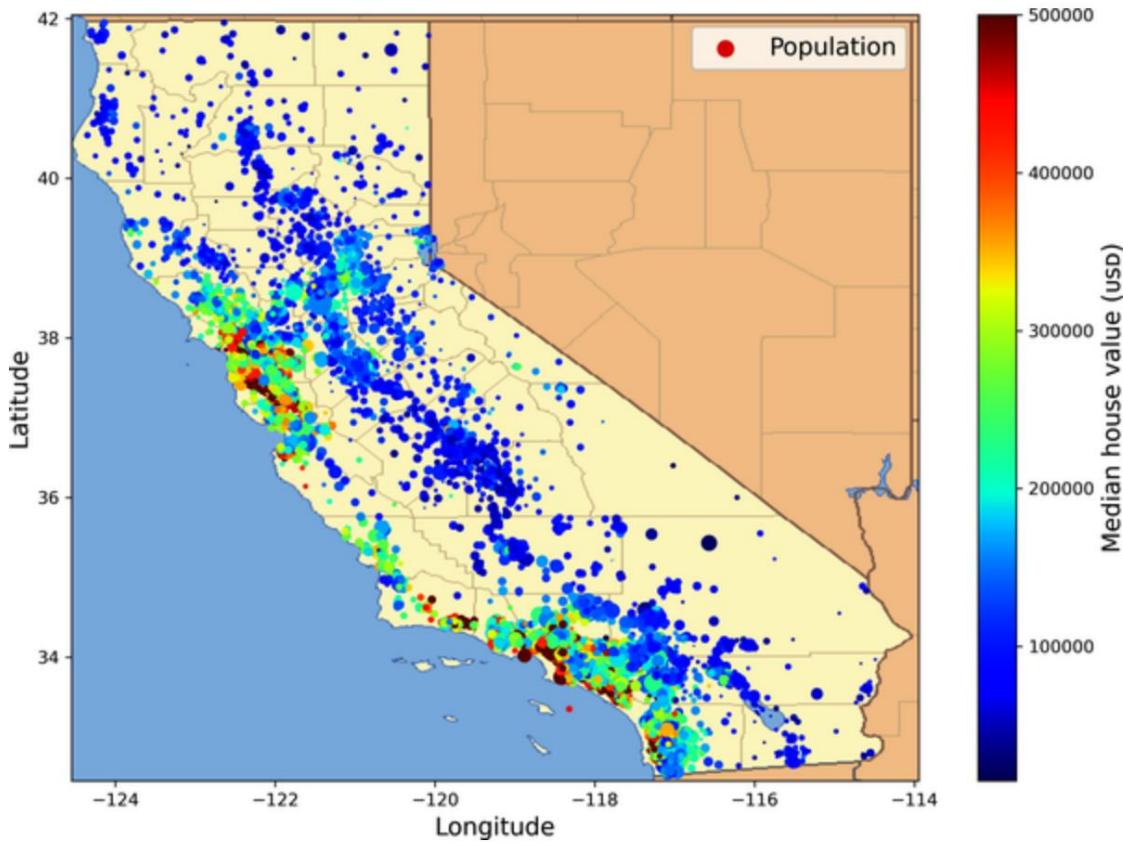


Figura 2-1. Precios de la vivienda en California

## Mire el panorama general ¡Bienvenido

a Machine Learning Housing Corporation! Su primera tarea es utilizar los datos del censo de California para construir un modelo de precios de vivienda en el estado. Estos datos incluyen métricas como la población, el ingreso medio y el precio medio de la vivienda para cada grupo de bloques en California. Los grupos de bloques son la unidad geográfica más pequeña para la cual la Oficina del Censo de EE. UU. publica datos de muestra (un grupo de bloques suele tener una población de 600 a 3000 personas). Los llamaré "distritos" para abreviar.

Su modelo debería aprender de estos datos y poder predecir el precio medio de la vivienda en cualquier distrito, teniendo en cuenta todas las demás métricas.

## CONSEJO

Dado que es un científico de datos bien organizado, lo primero que debe hacer es sacar la lista de verificación de su proyecto de aprendizaje automático. Puede comenzar con el del [Apéndice A](#); Debería funcionar razonablemente bien para la mayoría de los proyectos de aprendizaje automático, pero asegúrese de adaptarlo a sus necesidades. En este capítulo repasaremos muchos elementos de la lista de verificación, pero también omitiremos algunos, ya sea porque se explican por sí solos o porque se analizarán en capítulos posteriores.

## Enmarque el problema

La primera pregunta que debe hacerle a su jefe es cuál es exactamente el objetivo comercial. Probablemente construir un modelo no sea el objetivo final. ¿Cómo espera la empresa utilizar y beneficiarse de este modelo? Conocer el objetivo es importante porque determinará cómo planteará el problema, qué algoritmos seleccionará, qué medida de rendimiento utilizará para evaluar su modelo y cuánto esfuerzo dedicará a modificarlo.

Su jefe responde que el resultado de su modelo (una predicción del precio medio de la vivienda en un distrito) se enviará a otro sistema de aprendizaje automático (consulte la [Figura 2-2](#)), junto con muchas otras señales. Este sistema posterior determinará si vale la pena invertir en un área determinada. Hacer esto bien es fundamental, ya que afecta directamente a los ingresos.

La siguiente pregunta que debe hacerle a su jefe es cómo es la solución actual (si corresponde). La situación actual a menudo le brindará una referencia de desempeño, así como también ideas sobre cómo resolver el problema. Su jefe responde que actualmente los precios de las viviendas en los distritos los calculan manualmente expertos: un equipo recopila información actualizada sobre un distrito y, cuando no pueden obtener el precio medio de las viviendas, lo estiman utilizando reglas complejas.

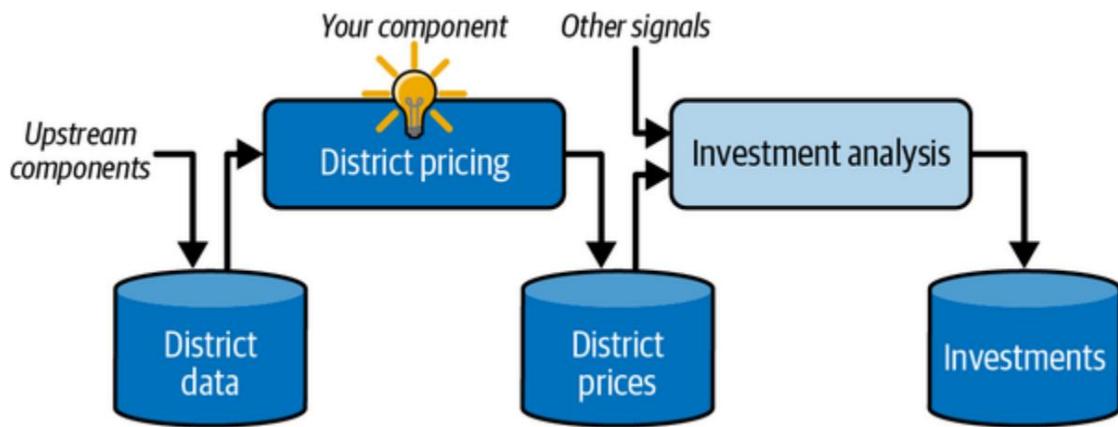


Figura 2-2. Un canal de aprendizaje automático para inversiones inmobiliarias

Este es costoso y lleva mucho tiempo, y sus estimaciones no son muy buenas; en los casos en que logran averiguar el precio medio real de la vivienda, a menudo se dan cuenta de que sus estimaciones estaban equivocadas en más del 30%. Por este motivo, la empresa cree que sería útil entrenar un modelo para predecir el precio medio de la vivienda en un distrito, teniendo en cuenta otros datos sobre ese distrito. Los datos del censo parecen un gran conjunto de datos para explotar con este propósito, ya que incluyen los precios medios de la vivienda de miles de distritos, así como otros datos.

## TUBERÍAS

Una secuencia de componentes de procesamiento de datos se denomina canalización de datos.

Las canalizaciones son muy comunes en los sistemas de aprendizaje automático, ya que hay muchos datos que manipular y muchas transformaciones de datos que aplicar.

Los componentes normalmente se ejecutan de forma asíncrona. Cada componente extrae una gran cantidad de datos, los procesa y arroja el resultado en otro almacén de datos. Luego, algún tiempo después, el siguiente componente en el proceso extrae estos datos y genera su propia salida. Cada componente es bastante autónomo: la interfaz entre componentes es simplemente el almacén de datos. Esto hace que el sistema sea fácil de entender (con la ayuda de un gráfico de flujo de datos) y diferentes equipos pueden centrarse en diferentes componentes.

Además, si un componente se estropea, los componentes posteriores a menudo pueden continuar funcionando normalmente (al menos por un tiempo) simplemente utilizando la última salida del componente roto. Esto hace que la arquitectura sea bastante robusta.

Por otro lado, un componente roto puede pasar desapercibido durante algún tiempo si no se implementa un seguimiento adecuado. Los datos se vuelven obsoletos y el rendimiento general del sistema cae.

Con toda esta información, ya está listo para comenzar a diseñar su sistema.

Primero, determine qué tipo de supervisión de capacitación necesitará el modelo: ¿es una tarea de aprendizaje supervisada, no supervisada, semisupervisada, autosupervisada o de refuerzo? ¿Y es una tarea de clasificación, una tarea de regresión o algo más? ¿Debería utilizar técnicas de aprendizaje por lotes o de aprendizaje en línea? Antes de seguir leyendo, haga una pausa e intente responder estas preguntas usted mismo.

¿Has encontrado las respuestas? Vamos a ver. Esta es claramente una tarea típica de aprendizaje supervisado, ya que el modelo se puede entrenar con ejemplos etiquetados (cada instancia viene con el resultado esperado, es decir, el precio medio de la vivienda en el distrito). Es una tarea de regresión típica, ya que se le pedirá al modelo que prediga un valor. Más específicamente, este es un problema de regresión múltiple , ya que el sistema utilizará múltiples características para hacer una predicción (la población del distrito, el ingreso medio, etc.). También es un problema de regresión univariante , ya que sólo intentamos predecir una única

valor para cada distrito. Si intentáramos predecir múltiples valores por distrito, sería un problema de regresión multivariada . Finalmente, no hay un flujo continuo de datos que ingresa al sistema, no hay una necesidad particular de adaptarse a los cambios rápidos de los datos y los datos son lo suficientemente pequeños como para caber en la memoria, por lo que el aprendizaje por lotes simple debería funcionar bien.

## CONSEJO

Si los datos fueran enormes, podría dividir su trabajo de aprendizaje por lotes en varios servidores (usando la técnica MapReduce) o utilizar una técnica de aprendizaje en línea.

## Seleccione una medida de desempeño

El siguiente paso es seleccionar una medida de desempeño. Una medida de rendimiento típica para problemas de regresión es la raíz del error cuadrático medio (RMSE). Da una idea de cuánto error suele cometer el sistema en sus predicciones, dando mayor importancia a los errores grandes. [La ecuación 2-1](#) muestra la fórmula matemática para calcular el RMSE.

### Ecuación 2-1. Error cuadrático medio (RMSE)

$$\text{RMSE}(X, h) = \sqrt{\frac{1}{\text{metros}} \sum_{y_0=1}^{\text{metro}} (h(x(i)) - y(i))^2}$$

## NOTACIONES

Esta ecuación presenta varias notaciones de aprendizaje automático muy comunes que usaré a lo largo de este libro:

- $m$  es el número de instancias en el conjunto de datos en el que está midiendo el RMSE.
- Por ejemplo, si está evaluando el RMSE en un conjunto de validación de 2000 distritos, entonces  $m = 2000$ .
- $(i)$   $x$  es un vector de todos los valores de características (excluyendo la etiqueta  $y$ ) de la  $(i)$  instancia  $i$  en el conjunto de datos, y  $y$  es su etiqueta (el valor de salida deseado para esa instancia).
- Por ejemplo, si el primer distrito del conjunto de datos está ubicado en longitud  $-118,29^\circ$ , latitud  $33,91^\circ$ , y tiene 1.416 habitantes con un ingreso medio de \$38.372, y el valor medio de la casa es \$156.400 (ignorando otras características por ahora), entonces :

$$\begin{matrix} & -118,29 \\ x(1) = & \begin{matrix} 33,91 \\ 1.416 \\ 38.372 \end{matrix} \end{matrix}$$

y:

$$(1) = 156.400 \text{ años}$$

- $X$  es una matriz que contiene todos los valores de las características (excluidas las etiquetas) de todas las instancias del conjunto de datos. Hay una fila por instancia y  $(i)$  la  $i$  fila es igual a la transpuesta de  $x$ , 3 anotado  $(^T_x)$ .
- Por ejemplo, si el primer distrito es como se acaba de describir, entonces la matriz  $X$  se verá así:

$$\begin{aligned}
 & (x(1)) \\
 & (x(2)) \\
 X = & -118,29 \ 33,91 \ 1.416 \ 38.372 \\
 & = ( \\
 & (x(1999)) \\
 & (x(2000))
 \end{aligned}$$

- $h$  es la función de predicción de su sistema, también llamada hipótesis. ( $i$ )  
Cuando a su sistema se le proporciona el vector de características de  $x^{(i)}$ , él genera un valor predicho  $\hat{y} = h(x^{(i)})$  para esa instancia ( $\hat{y}$  se pronuncia “y-hat”).
- Por ejemplo, si su sistema predice que el precio medio de la vivienda ( $1^{(1)}$ ) en el primer distrito es \$158,400, entonces  $\hat{y} = h(x^{(1)}) = 158.400$ . El error de predicción para este distrito es  $\hat{y} - y^{(1)} = 2000$ .
- $RMSE(X, h)$  es la función de costo medida en el conjunto de ejemplos utilizando su hipótesis  $h$ .

Usamos fuente en cursiva minúscula para valores escalares (como  $m$  o  $y$ )  $\hat{y}^{(i)}$   
nombres de funciones (como  $h$ ), fuente en negrita minúscula para vectores (como  $x^{(i)}$ ) y fuente en negrita mayúscula para matrices (como  $X$ ).

Aunque el RMSE es generalmente la medida de rendimiento preferida para tareas de regresión, en algunos contextos es posible que prefiera utilizar otra función. Por ejemplo, si hay muchos distritos atípicos. En ese caso, puede considerar utilizar el error absoluto medio (MAE, también llamado desviación absoluta promedio), que se muestra en [la Ecuación 2-2](#):

Ecuación 2-2. Error absoluto medio (MAE)

$$MAE(X, h) = \frac{1}{\text{metro}} \sum_{i=1}^{\text{metro}} |h(x^{(i)}) - y^{(i)}|$$

Tanto el RMSE como el MAE son formas de medir la distancia entre dos vectores: el vector de predicciones y el vector de valores objetivo.

Son posibles varias medidas o normas de distancia:

- Calcular la raíz de una suma de cuadrados (RMSE) corresponde a la norma euclídea: esta es la noción de distancia con la que todos estamos familiarizados.  
También se le llama norma  $\ell_2$ , anotada  $\|\cdot\|_2$  (o simplemente  $\|\cdot\|$ ).
- Calcular la suma de absolutos (MAE) corresponde a la norma  $\ell_1$ , anotada  $\|\cdot\|_1$ . A esto a veces se le llama norma de Manhattan porque mide la distancia entre dos puntos de una ciudad si solo se puede viajar a lo largo de manzanas ortogonales.
- De manera más general, la norma  $\ell_p$  de un vector  $v$  que contiene  $n$  elementos se define como  $\|v\|_p = (\sum_{k=1}^n |v_k|^p)^{1/p}$ .  $\ell_1$  da el número de elementos distintos de cero en el vector, y  $\ell_\infty$  da el valor absoluto máximo en el vector.

Cuanto mayor es el índice normal, más se centra en los valores grandes y se descuidan los pequeños. Por eso el RMSE es más sensible a los valores atípicos que el MAE. Pero cuando los valores atípicos son exponencialmente raros (como en una curva en forma de campana), el RMSE funciona muy bien y generalmente es el preferido.

## Verifique las suposiciones

Por último, es una buena práctica enumerar y verificar las suposiciones que se han hecho hasta ahora (por usted u otros); esto puede ayudarle a detectar problemas graves desde el principio. Por ejemplo, los precios de distrito que genera su sistema se introducirán en un sistema de aprendizaje automático posterior, y usted supone que estos precios se utilizarán como tales. Pero ¿qué pasa si el sistema descendente convierte los precios en categorías (por ejemplo, "barato", "medio" o "caro") y luego utiliza esas categorías en lugar de los precios mismos? En este caso, acertar perfectamente con el precio no tiene ninguna importancia; su sistema sólo necesita obtener la categoría correcta. Si es así, entonces el problema debería haberse planteado como una tarea de clasificación, no como una tarea de regresión. No querrás descubrir esto después de trabajar en un sistema de regresión durante meses.

Afortunadamente, después de hablar con el equipo a cargo del sistema downstream, está seguro de que realmente necesitan los precios reales, no solo las categorías. ¡Excelente! ¡Ya está todo listo, las luces están en verde y puede comenzar a codificar ahora!

## Obtener los datos

Es hora de ensuciarse las manos. No dude en tomar su computadora portátil y leer los ejemplos de código. Como mencioné en el prefacio, todos los ejemplos de código de este libro son de código abierto y están disponibles [en línea](#), como los cuadernos de Jupyter, que son documentos interactivos que contienen texto, imágenes y fragmentos de código ejecutable (Python en nuestro caso). En este libro asumiré que está ejecutando estos cuadernos en Google Colab, un servicio gratuito que le permite ejecutar cualquier cuaderno Jupyter directamente en línea, sin tener que instalar nada en su máquina. Si desea utilizar otra plataforma en línea (por ejemplo, Kaggle) o si desea instalar todo localmente en su propia máquina, consulte las instrucciones en la página de GitHub del libro.

## Ejecutar los ejemplos de código usando Google Colab

Primero, abra un navegador web y visite <https://homl.info/colab3>: Esto lo llevará a Google Colab y mostrará la lista de cuadernos Jupyter para este libro (consulte la [Figura 2-3](#)). Encontrará un cuaderno por capítulo, además de algunos cuadernos y tutoriales adicionales para NumPy, Matplotlib, Pandas, álgebra lineal y cálculo diferencial. Por ejemplo, si hace clic en 02\_end\_to\_end\_machine\_learning\_project.ipynb, el cuaderno del [Capítulo 2](#) se abrirá en Google Colab (consulte [la Figura 2-4](#)).

Un cuaderno de Jupyter se compone de una lista de celdas. Cada celda contiene código ejecutable o texto. Intente hacer doble clic en la primera celda de texto (que contiene la frase "¡Bienvenido a Machine Learning Housing Corp.!"). Esto abrirá la celda para editarla. Tenga en cuenta que los cuadernos de Jupyter utilizan la sintaxis de Markdown para formatear (p. ej., **negrita**, *cursiva*, # Título, [url](texto del enlace), etc.). Intente modificar este texto, luego presione Shift-Enter para ver el resultado.

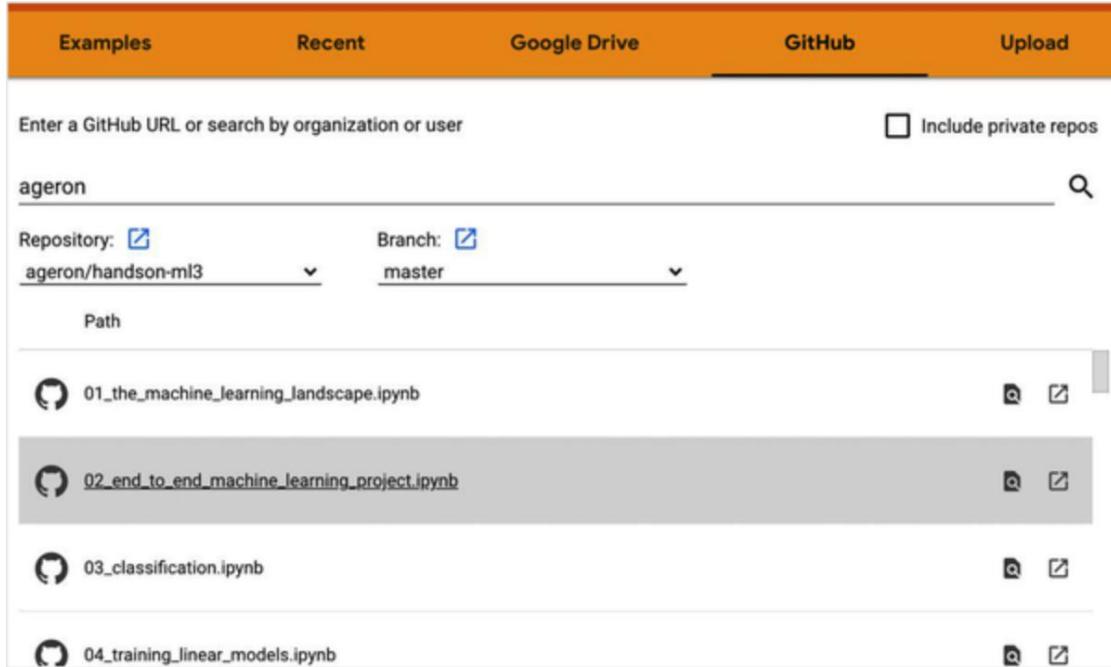


Figura 2-3. Lista de cuadernos en Google Colab

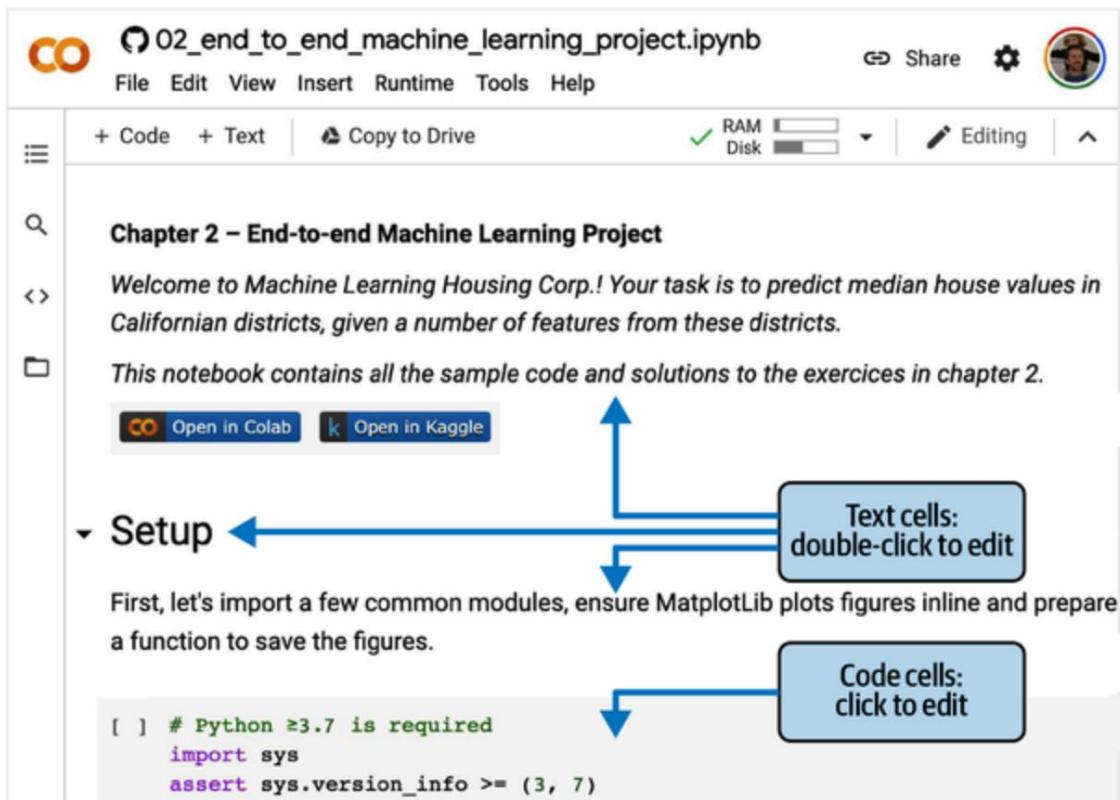


Figura 2-4. Tu cuaderno en Google Colab

A continuación, cree una nueva celda de código seleccionando Insertar → “Celda de código” en el menú. Alternativamente, puede hacer clic en el botón + Código en la barra de herramientas, o

Pase el mouse sobre la parte inferior de una celda hasta que vea aparecer + Código y + Texto, luego haga clic en + Código. En la nueva celda de código, escriba algún código Python, como `print("Hello World")`, luego presione Shift-Enter para ejecutar este código (o haga clic en el botón  en el lado izquierdo de la celda).

Si no ha iniciado sesión en su cuenta de Google, se le pedirá que inicie sesión ahora (si aún no tiene una cuenta de Google, deberá crear una). Una vez que haya iniciado sesión, cuando intente ejecutar el código, verá una advertencia de seguridad que le indica que este cuaderno no fue creado por Google. Una persona malintencionada podría crear un cuaderno que intente engañarlo para que ingrese sus credenciales de Google para poder acceder a sus datos personales, por lo que antes de ejecutar un cuaderno, asegúrese siempre de confiar en su autor (o verifique dos veces qué hará cada celda de código). antes de ejecutarlo). Suponiendo que confías en mí (o planeas verificar cada celda de código), ahora puedes hacer clic en "Ejecutar de todos modos".

Luego, Colab le asignará un nuevo tiempo de ejecución : se trata de una máquina virtual gratuita ubicada en los servidores de Google que contiene un montón de herramientas y bibliotecas de Python, incluido todo lo que necesitará para la mayoría de los capítulos (en algunos capítulos, deberá ejecutar un comando para instalar bibliotecas adicionales). Esto tardará unos segundos. A continuación, Colab se conectará automáticamente a este tiempo de ejecución y lo utilizará para ejecutar su nueva celda de código. Es importante destacar que el código se ejecuta en tiempo de ejecución, no en su máquina. La salida del código se mostrará debajo de la celda. ¡Felicitaciones, ha ejecutado código Python en Colab!

#### CONSEJO

Para insertar una nueva celda de código, también puede escribir Ctrl-M (o Cmd-M en macOS) seguido de A (para insertar encima de la celda activa) o B (para insertar debajo). Hay muchos otros atajos de teclado disponibles: puede verlos y editarlos escribiendo Ctrl-M (o Cmd-M) y luego H. Si elige ejecutar los cuadernos en Kaggle o en su propia máquina usando JupyterLab o un IDE como Visual Studio Code con la extensión Jupyter, verá algunas diferencias menores (los tiempos de ejecución se denominan kernels, la interfaz de usuario y los atajos de teclado son ligeramente diferentes, etc.), pero cambiar de un entorno Jupyter a otro no es demasiado difícil.

## Guardar los cambios de su código y sus datos

Puede realizar cambios en un cuaderno de Colab y persistirán mientras mantenga abierta la pestaña del navegador. Pero una vez que lo cierres, los cambios se perderán. Para evitar esto, asegúrese de guardar una copia del cuaderno en su Google Drive seleccionando Archivo → “Guardar una copia en Drive”.

Alternativamente, puede descargar la computadora portátil a su computadora seleccionando Archivo → Descargar → “Descargar .ipynb”. Luego podrás visitar <https://colab.research.google.com> y abre la libreta nuevamente (ya sea desde Google Drive o subiéndola desde tu computadora).

#### ADVERTENCIA

Google Colab está diseñado únicamente para uso interactivo: puede jugar en los cuadernos y modificar el código como desee, pero no puede dejar que los cuadernos se ejecuten sin supervisión durante un largo período de tiempo, de lo contrario, el tiempo de ejecución se cerrará y todos sus datos se perderán.

Si el portátil genera datos que le interesan, asegúrese de descargarlos antes de que se cierre el tiempo de ejecución. Para hacer esto, haga clic en el ícono Archivos (consulte el paso 1 en [la Figura 2-5](#)), busque el archivo que desea descargar, haga clic en los puntos verticales al lado (paso 2) y haga clic en Descargar (paso 3).

Alternativamente, puede montar su Google Drive en el tiempo de ejecución, permitiendo que la computadora portátil lea y escriba archivos directamente en Google Drive como si fuera un directorio local. Para esto, haga clic en el ícono Archivos (paso 1), luego haga clic en el ícono de Google Drive (encerrado en un círculo en la [Figura 2-5](#)) y siga las instrucciones en pantalla.

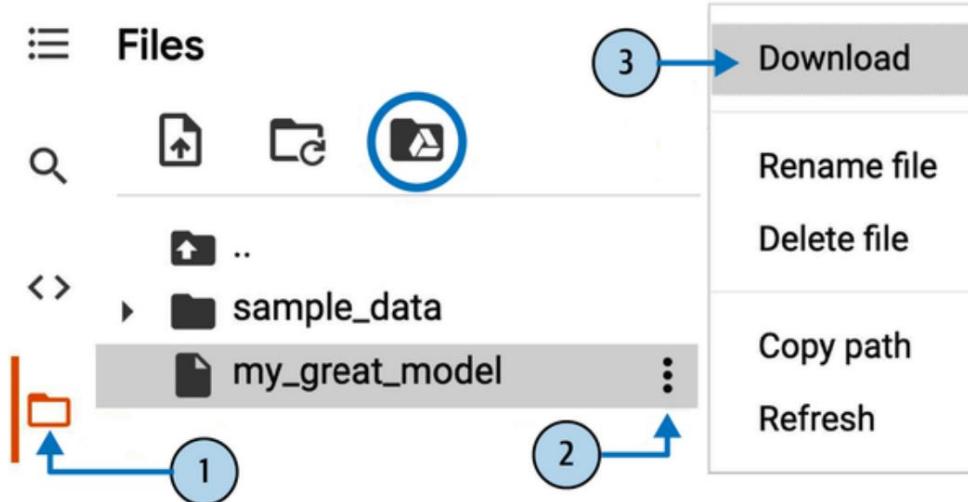


Figura 2-5. Descargar un archivo desde un tiempo de ejecución de Google Colab (pasos 1 a 3) o montar su Google Drive (ícono de círculo)

De forma predeterminada, su Google Drive se montará en /content/drive/MyDrive. Si desea hacer una copia de seguridad de un archivo de datos, simplemente cópielo a este directorio ejecutando !cp /content/my\_great\_model /content/drive/MyDrive. Cualquier comando que comience con bang (!) se trata como un comando de shell, no como código Python: cp es el comando de shell de Linux para copiar un archivo de una ruta a otra. Tenga en cuenta que los tiempos de ejecución de Colab se ejecutan en Linux (específicamente, Ubuntu).

## El poder y el peligro de la interactividad

Los cuadernos de Jupyter son interactivos, y eso es genial: puedes ejecutar cada celda una por una, detenerte en cualquier punto, insertar una celda, jugar con el código, regresar y ejecutar la misma celda nuevamente, etc., y te recomiendo encarecidamente que lo hagas. Si simplemente ejecuta las celdas una por una sin jugar con ellas, no aprenderá tan rápido. Sin embargo, esta flexibilidad tiene un precio: es muy fácil ejecutar celdas en el orden incorrecto u olvidarse de ejecutar una celda. Si esto sucede, es probable que las celdas de código posteriores fallen. Por ejemplo, la primera celda de código de cada cuaderno contiene código de configuración (como importaciones), así que asegúrese de ejecutarlo primero o nada funcionará.

CONSEJO

Si alguna vez se encuentra con un error extraño, intente reiniciar el tiempo de ejecución (seleccionando Tiempo de ejecución → “Reiniciar tiempo de ejecución” en el menú) y luego ejecute todas las celdas nuevamente desde el principio del cuaderno. Esto suele resolver el problema. De lo contrario, es probable que uno de los cambios que realizó haya dañado la libreta: simplemente vuelva a la libreta original e intente nuevamente. Si aún falla, presente un problema en GitHub.

## Código de libro versus código de cuaderno

A veces puedes notar algunas pequeñas diferencias entre el código de este libro y el código de los cuadernos. Esto puede suceder durante varios razones:

- Es posible que una biblioteca haya cambiado ligeramente en el momento en que leas estas líneas, o tal vez a pesar de mis mejores esfuerzos cometí un error en el libro. Lamentablemente, no puedo arreglar mágicamente el código de tu copia de este libro (a menos que estés leyendo una copia electrónica y puedas descargar la última versión), pero puedo arreglar los cuadernos. Por lo tanto, si se encuentra con un error después de copiar el código de este libro, busque el código fijo en los cuadernos: me esforzaré por mantenerlos libres de errores y actualizados con las últimas versiones de la biblioteca.
- Los cuadernos contienen código adicional para embellecer las figuras (agregando etiquetas, configurando tamaños de fuente, etc.) y guardarlas en alta resolución para este libro. Puede ignorar con seguridad este código adicional si lo desea.

Optimicé el código para que fuera legible y simple: lo hice lo más lineal y plano posible, definiendo muy pocas funciones o clases. El objetivo es garantizar que el código que está ejecutando esté generalmente justo frente a usted y no anidado dentro de varias capas de abstracciones en las que tenga que buscar. Esto también te facilita jugar con el código. Para simplificar, el manejo de errores es limitado y coloqué algunas de las importaciones menos comunes justo donde se necesitan (en lugar de colocarlas en la parte superior del archivo, como recomienda la guía de estilo PEP 8 Python).

Dicho esto, su código de producción no será muy diferente: solo un poco más modular y con pruebas adicionales y manejo de errores.

¡DE ACUERDO! Una vez que se sienta cómodo con Colab, estará listo para descargar los datos.

## Descargar los datos

En entornos típicos, sus datos estarían disponibles en una base de datos relacional o algún otro almacén de datos común, y estarían distribuidos en múltiples tablas/documentos/archivos. Para acceder a él, primero deberá obtener su

4 credenciales y autorizaciones de acceso y familiarizarse con el esquema de datos. En este proyecto, sin embargo, las cosas son mucho más simples: simplemente descargará un único archivo comprimido, housing.tgz, que contiene un archivo de valores separados por comas (CSV) llamado housing.csv con todos los datos.

En lugar de descargar y descomprimir los datos manualmente, normalmente es preferible escribir una función que lo haga por usted. Esto es útil en particular si los datos cambian regularmente: puede escribir un pequeño script que use la función para recuperar los datos más recientes (o puede configurar un trabajo programado para que lo haga automáticamente a intervalos regulares). Automatizar el proceso de obtención de datos también es útil si necesita instalar el conjunto de datos en varias máquinas.

Aquí está la función para buscar y cargar los datos:

```
desde pathlib importar ruta importar
pandas como pd importar
archivo tar importar
urllib.request

def cargar_datos_vivienda():
    tarball_path = Ruta("datasets/housing.tgz") si no tarball_path.is_file():
        Ruta("datasets").mkdir(parents=True,
                               exist_ok=True)
        URL =
            "https://github.com/ageron/data/raw/main/housing.tgz" urllib.request.urlretrieve(url,
                tarball_path) con tarfile.open(tarball_path) como housing_tarball:
                housing_tarball.extractall(path="datasets" )

    devolver pd.read_csv(Ruta("conjuntos de datos/vivienda/vivienda.csv"))

vivienda = load_housing_data()
```

Cuando se llama a `load_housing_data()`, busca el archivo datasets/housing.tgz . Si no lo encuentra, crea el directorio de conjuntos de datos dentro del directorio actual (que es /content de forma predeterminada, en Colab), descarga el archivo housing.tgz del repositorio de GitHub ageron/data y extrae su contenido en el directorio de conjuntos de datos . ; esto crea el directorio datasets/housing con el archivo housing.csv en su interior. Por último, la función carga este archivo CSV en un objeto Pandas DataFrame que contiene todos los datos y lo devuelve.

## Eche un vistazo rápido a la estructura de datos

Comienza mirando las cinco filas superiores de datos usando el DataFrame. método head() (consulte la Figura 2-6).

	longitude	latitude	housing_median_age	median_income	ocean_proximity	median_house_value
0	-122.23	37.88	41.0	8.3252	NEAR BAY	452600.0
1	-122.22	37.86	21.0	8.3014	NEAR BAY	358500.0
2	-122.24	37.85	52.0	7.2574	NEAR BAY	352100.0
3	-122.25	37.85	52.0	5.6431	NEAR BAY	341300.0
4	-122.25	37.85	52.0	3.8462	NEAR BAY	342200.0

Figura 2-6. Las cinco primeras filas del conjunto de datos

Cada fila representa un distrito. Hay 10 atributos (no son todos mostrado en la captura de pantalla): longitud, latitud, edad\_mediana\_vivienda, habitaciones\_total, habitaciones\_total, población, hogares, ingreso\_mediano, valor\_casa\_mediana, y proximidad\_oceánica.

El método info() es útil para obtener una descripción rápida de los datos, en particular el número total de filas, el tipo de cada atributo y el número de valores no nulos:

```
>>> vivienda.info()
<clase 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entradas, 0 a 20639
Columnas de datos (un total de 10 columnas):
 # Columna          Tipo D de recuento no nulo
_____
0 longitud    20640 float64 no nulo
1 latitud     20640 float64 no nulo
2 housing_median_age 20640 float64 no nulo
3 total_rooms   20640 float64 no nulo
4 total_dormitorios 5           20433 flotador no nulo64
población      6 hogares       20640 flotador no nulo64
                20640 flotador no nulo64
7 ingreso_mediano 8           20640 flotador no nulo64
valor_casa_mediana 20640 float64 no nulo
9 ocean_proximity 20640 objeto no nulo
tipos de datos: float64(9), objeto(1)
Uso de memoria: 1,6+ MB
```

## NOTA

En este libro, cuando un ejemplo de código contiene una combinación de código y resultados, como es el caso aquí, se formatea como en el intérprete de Python, para una mejor legibilidad: las líneas de código tienen el prefijo >>> (o ... para bloques sangrados), y las salidas no tienen prefijo.

Hay 20.640 instancias en el conjunto de datos, lo que significa que es bastante pequeño según los estándares del aprendizaje automático, pero es perfecto para comenzar. Observa que el atributo total\_bedrooms tiene solo 20 433 valores no nulos, lo que significa que a 207 distritos les falta esta característica. Tendrás que encargarte de esto más tarde.

Todos los atributos son numéricos, excepto ocean\_proximity. Su tipo es objeto, por lo que podría contener cualquier tipo de objeto Python. Pero como cargaste estos datos desde un archivo CSV, sabes que debe ser un atributo de texto. Cuando miró las cinco filas superiores, probablemente notó que los valores en la columna ocean\_proximity eran repetitivos, lo que significa que probablemente sea un atributo categórico. Puede averiguar qué categorías existen y cuántos distritos pertenecen a cada categoría utilizando el método value\_counts():

```
>>> vivienda["ocean_proximity"].value_counts()
<1H OCÉANO      9136
INTERIOR        6551
CERCA DEL OCÉANO    2658
CERCA DE LA BAHÍA   2290
ISLA
Nombre: ocean_proximity, tipo: int64
```

Miremos los otros campos. El método describe() muestra un resumen de los atributos numéricos ([Figura 2-7](#)).

housing.describe()						
	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	median_house_value
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000	20640.000000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553	206855.816909
std	2.003532	2.135952	12.585558	2181.615252	421.385070	115395.615874
min	-124.350000	32.540000	1.000000	2.000000	1.000000	14999.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000	119600.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000	179700.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000	264725.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	500001.000000

Figura 2-7. Resumen de cada atributo numérico

Las filas de recuento, media, mínimo y máximo se explican por sí mismas. Tenga en cuenta que los valores nulos se ignoran (por ejemplo, el recuento de total\_dormitorios es 20.433, no 20.640). La fila estándar muestra la desviación estándar, que mide qué tan dispersos están los <sup>5</sup> valores. Las filas 25%, 50% y 75% muestran los percentiles correspondientes: un percentil indica el valor por debajo del cual cae un porcentaje determinado de observaciones en un grupo de observaciones. Por ejemplo, el 25% de los distritos tienen una edad\_mediana\_de\_vivienda inferior a 18, mientras que el 50% es inferior a 29 y el 75% es inferior a 37. Estos a menudo se denominan percentil 25 (o primer cuartil), mediana y percentil 75 . (o tercer cuartil).

Otra forma rápida de tener una idea del tipo de datos que está tratando es trazar un histograma para cada atributo numérico. Un histograma muestra el número de instancias (en el eje vertical) que tienen un rango de valores determinado (en el eje horizontal). Puede trazar este atributo a la vez o puede llamar al método hist() en todo el conjunto de datos (como se muestra en el siguiente ejemplo de código) y trazará un histograma para cada atributo numérico (consulte la Figura 2-8) . ):

```
importar matplotlib.pyplot como plt
```

```
vivienda.hist(bins=50, figsize=(12, 8)) plt.show()
```

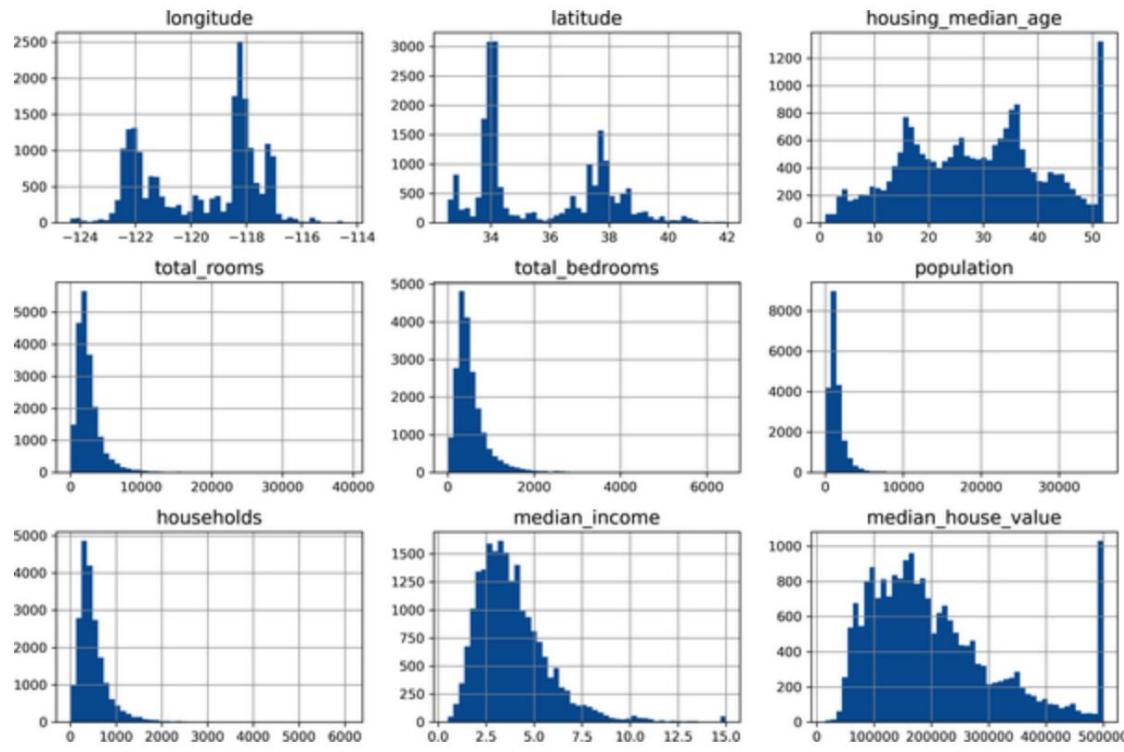


Figura 2-8. Un histograma para cada atributo numérico.

Al observar estos histogramas, notas algunas cosas:

- En primer lugar, el atributo de ingreso medio no parece estar expresado en dólares estadounidenses (USD). Después de consultar con el equipo que recopiló los datos, le dicen que los datos se han escalado y limitado a 15 (en realidad, 15,0001) para ingresos medios más altos, y a 0,5 (en realidad, 0,4999) para ingresos medios más bajos. Los números representan aproximadamente decenas de miles de dólares (por ejemplo, 3 en realidad significa alrededor de \$30,000). Trabajar con atributos preprocesados es común en el aprendizaje automático y no es necesariamente un problema, pero debes intentar comprender cómo se calcularon los datos.
- También se limitaron la edad media de la vivienda y el valor medio de la vivienda. Esto último puede ser un problema grave ya que es su atributo objetivo (sus etiquetas). Sus algoritmos de aprendizaje automático pueden aprender que los precios nunca superan ese límite. Debe consultar con el equipo de su cliente (el equipo que utilizará la salida de su sistema) para ver si esto es un problema o no. Si le dicen que necesitan predicciones precisas incluso superiores a 500.000 dólares, entonces tiene dos opciones:

- Recopile etiquetas adecuadas para los distritos cuyas etiquetas fueron limitadas.
- Elimine esos distritos del conjunto de entrenamiento (y también del conjunto de prueba, ya que su sistema no debería evaluarse mal si predice valores superiores a \$500 000).
- Estos atributos tienen escalas muy diferentes. Discutiremos esto más adelante en este capítulo, cuando exploremos el escalado de funciones.
- Finalmente, muchos histogramas están sesgados hacia la derecha: se extienden mucho más hacia la derecha de la mediana que hacia la izquierda. Esto puede hacer que a algunos algoritmos de aprendizaje automático les resulte un poco más difícil detectar patrones. Más adelante, intentará transformar estos atributos para que tengan distribuciones más simétricas y en forma de campana.

Ahora debería comprender mejor el tipo de datos con los que está tratando.

#### ADVERTENCIA

¡Esperar! Antes de seguir analizando los datos, debe crear un conjunto de prueba, dejarlo a un lado y nunca mirarlo.

### Crear un conjunto de prueba

Puede parecer extraño dejar de lado voluntariamente parte de los datos en esta etapa. Después de todo, solo has echado un vistazo rápido a los datos y seguramente deberías aprender mucho más sobre ellos antes de decidir qué algoritmos usar, ¿verdad? Esto es cierto, pero su cerebro es un sorprendente sistema de detección de patrones, lo que también significa que es muy propenso al sobreajuste: si observa el conjunto de pruebas, puede toparse con algún patrón aparentemente interesante en los datos de la prueba que le lleve a seleccionar un tipo particular de modelo de aprendizaje automático. Cuando calcule el error de generalización utilizando el conjunto de prueba, su estimación será demasiado optimista y lanzará un sistema que no funcionará tan bien como se esperaba. Esto se llama sesgo de espionaje de datos .

Crear un conjunto de pruebas es teóricamente sencillo; Elija algunas instancias al azar, normalmente el 20% del conjunto de datos (o menos si su conjunto de datos es muy grande) y déjelas a un lado:

```

importar numpy como np

def shuffle_and_split_data(datos, test_ratio):
    índices_shuffled = np.random.permutation(len(datos)) test_set_size =
        int(len(datos) * test_ratio) test_indices = índices_shuffled[:test_set_size]
    train_indices = índices_shuffled[test_set_size:] return
    datos.iloc[train_indices], datos.iloc[índices_prueba]

```

Luego puedes usar esta función de esta manera:

```

>>> conjunto_tren, conjunto_prueba = shuffle_and_split_data(vivienda, 0.2) >>>
len(conjunto_tren) 16512
>>>
len(conjunto_prueba)
4128

```

Bueno, esto funciona, pero no es perfecto: si ejecutas el programa nuevamente, ¡generará un conjunto de pruebas diferente! Con el tiempo, usted (o sus algoritmos de aprendizaje automático) podrán ver todo el conjunto de datos, que es lo que desea evitar.

Una solución es guardar el conjunto de prueba en la primera ejecución y luego cargarlo en ejecuciones posteriores. Otra opción es configurar la semilla del generador de números aleatorios (por ejemplo, con `np.random.seed(42)`) antes de llamar a `np.random.permutation()` para que siempre genere los mismos índices mezclados.<sup>6</sup>

Sin embargo, ambas soluciones fallarán la próxima vez que obtenga un conjunto de datos actualizado. Para tener una división de tren/prueba estable incluso después de actualizar el conjunto de datos, una solución común es usar el identificador de cada instancia para decidir si debe incluirse o no en el conjunto de prueba (asumiendo que las instancias tienen identificadores únicos e inmutables). Por ejemplo, podría calcular un hash del identificador de cada instancia y colocar esa instancia en el conjunto de prueba si el hash es menor o igual al 20 % del valor de hash máximo. Esto garantiza que el conjunto de pruebas se mantendrá coherente en varias ejecuciones, incluso si actualiza el conjunto de datos. El nuevo conjunto de prueba contendrá el 20 % de las nuevas instancias, pero no contendrá ninguna instancia que estuviera previamente en el conjunto de entrenamiento.

Aquí hay una posible implementación:

```
desde zlib importar crc32

def is_id_in_test_set(identificador, test_ratio): return
    crc32(np.int64(identificador)) < test_ratio * 2**32

def split_data_with_id_hash(datos, test_ratio, id_column): ids = datos[id_column]
    in_test_set = ids.apply(lambda
        id_: is_id_in_test_set(id_,
    test_ratio))
    devuelve datos.loc[~in_test_set], datos.loc[in_test_set]
```

Lamentablemente, el conjunto de datos de vivienda no tiene una columna de identificador. La solución más sencilla es utilizar el índice de fila como ID:

```
housing_with_id = housing.reset_index() # agrega un `índice` columna
train_set, test_set = split_data_with_id_hash(housing_with_id, 0.2, "índice")
```

Si utiliza el índice de fila como identificador único, debe asegurarse de que se agreguen datos nuevos al final del conjunto de datos y que nunca se elimine ninguna fila. Si esto no es posible, puede intentar utilizar las funciones más estables para crear un identificador único. Por ejemplo, se garantiza que la latitud y longitud de un distrito serán estables durante unos pocos millones de años, por lo que podrías **combinarlas** en un ID de la siguiente manera:

```
vivienda_con_id["id"] = vivienda["longitud"] * 1000 + vivienda["latitud"] train_set,
test_set =
split_data_with_id_hash(vivienda_con_id, 0.2, "id")
```

Scikit-Learn proporciona algunas funciones para dividir conjuntos de datos en múltiples subconjuntos de varias maneras. La función más simple es `train_test_split()`, que hace prácticamente lo mismo que la función `shuffle_and_split_data()` que definimos anteriormente, con un par de características adicionales. Primero, hay un parámetro `random_state` que le permite configurar la semilla del generador aleatorio. En segundo lugar, puede pasarle varios conjuntos de datos con un número idéntico de filas y los dividirá en los mismos índices (esto es muy útil, por ejemplo, si tiene un DataFrame separado para las etiquetas):

```
desde sklearn.model_selection importar train_test_split  
  
train_set, test_set = train_test_split(vivienda, test_size=0.2, random_state=42)
```

Hasta ahora hemos considerado métodos de muestreo puramente aleatorios. En general, esto está bien si su conjunto de datos es lo suficientemente grande (especialmente en relación con la cantidad de atributos), pero si no lo es, corre el riesgo de introducir un sesgo de muestreo significativo. Cuando los empleados de una empresa de encuestas deciden llamar a 1.000 personas para hacerles algunas preguntas, no eligen simplemente a 1.000 personas al azar en una guía telefónica. Intentan asegurarse de que estas 1.000 personas sean representativas de toda la población, en lo que respecta a las preguntas que quieren plantear. Por ejemplo, la población de EE.UU. está formada por un 51,1% de mujeres y un 48,9% de hombres, por lo que una encuesta bien realizada en EE.UU. intentaría mantener esta proporción en la muestra: 511 mujeres y 489 hombres (al menos si parece posible que las respuestas puedan varían según el género). Esto se llama muestreo estratificado: la población se divide en subgrupos homogéneos llamados estratos, y se muestrea el número correcto de instancias de cada estrato para garantizar que el conjunto de pruebas sea representativo de la población general. Si las personas que realizaron la encuesta utilizaron un muestreo puramente aleatorio, habría aproximadamente un 10,7% de posibilidades de muestrear un conjunto de pruebas sesgado con menos del 48,5% de mujeres o más del 53,5% de mujeres participantes. De cualquier manera, los resultados de la encuesta probablemente estarían Supongamos que ha conversado con algunos expertos que le dijeron que el ingreso medio es un atributo muy importante para predecir los precios medios de la vivienda. Es posible que desee asegurarse de que el conjunto de prueba sea representativo de las distintas categorías de ingresos en todo el conjunto de datos. Dado que el ingreso medio es un atributo numérico continuo, primero debe crear un atributo de categoría de ingreso. Miremos más de cerca el histograma del ingreso mediano (en [la Figura 2-8](#)): la mayoría de los valores del ingreso mediano se agrupan alrededor de 1,5 a 6 (es decir, entre \$15 000 y \$60 000), pero algunos ingresos medianos van mucho más allá de 6. Es importante tener un número suficiente de instancias en su conjunto de datos para cada estrato; de lo contrario, la estimación de la importancia de un estrato puede estar sesgada. Esto significa que no debe tener demasiados estratos y cada estrato debe ser lo suficientemente grande. El siguiente código utiliza la función pd.cut() para crear un atributo de categoría de ingresos con cinco categorías.

(etiquetado del 1 al 5); la categoría 1 varía de 0 a 1,5 (es decir, menos de \$15 000), la categoría 2 de 1,5 a 3, y así sucesivamente:

```
vivienda["ingresos_cat"] = pd.cut(vivienda["ingresos_medianos"],
                                    contenedores = [0., 1.5, 3.0, 4.5, 6.,
                                    np.inf],
                                    etiquetas = [1, 2, 3, 4, 5])
```

Estas categorías de ingresos están representadas en [la Figura 2-9](#):

```
vivienda["ingresos_cat"].value_counts().sort_index().plot.bar(rot= 0, grid=True) plt.xlabel(" Categoría de
ingresos ")
plt.ylabel("Número de distritos") plt.show( )
```

Ahora está listo para realizar un muestreo estratificado según la categoría de ingresos. Scikit-Learn proporciona una serie de clases de división en el paquete `sklearn.model_selection` que implementan varias estrategias para dividir su conjunto de datos en un conjunto de entrenamiento y un conjunto de prueba. Cada divisor tiene un método `split()` que devuelve un iterador sobre diferentes divisiones de entrenamiento/prueba de los mismos datos.

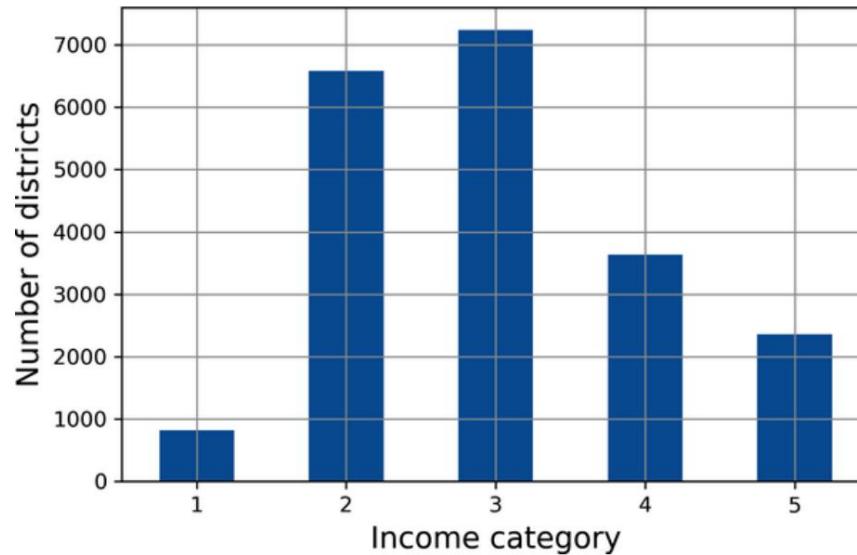


Figura 2-9. Histograma de categorías de ingresos

Para ser precisos, el método `split()` genera los índices de entrenamiento y prueba, no los datos en sí. Tener múltiples divisiones puede ser útil si quieras mejorar

estime el rendimiento de su modelo, como verá cuando analicemos la validación cruzada más adelante en este capítulo. Por ejemplo, el siguiente código genera 10 divisiones estratificadas diferentes del mismo conjunto de datos:

```
desde sklearn.model_selection importar StratifiedShuffleSplit

splitter = StratifiedShuffleSplit(n_splits=10, test_size=0.2, random_state=42) strat_splits = []
para train_index,
test_index en
splitter.split(vivienda, vivienda["ingresos_cat"]):

    strat_train_set_n = vivienda.iloc[train_index] strat_test_set_n =
    vivienda.iloc[test_index] strat_splits.append([strat_train_set_n,
    strat_test_set_n])
```

Por ahora, puedes usar sólo la primera división:

```
estrat_train_set, estrat_test_set = estrat_splits[0]
```

O, dado que el muestreo estratificado es bastante común, existe una forma más corta de obtener una división única usando la función `train_test_split()` con el argumento `estratificar`:

```
estrat_tren_set, estrat_test_set = tren_prueba_split(
    vivienda, test_size=0.2, estratificar=vivienda["ingresos_cat"], random_state=42)
```

Veamos si esto funcionó como se esperaba. Puede comenzar observando las proporciones de las categorías de ingresos en el conjunto de prueba:

```
>>> estrat_test_set["ingresos_cat"].value_counts() / len(strat_test_set)

3      0.350533
2      0.318798
4      0.176357
5      0.114341
1      0.039971

Nombre: ingresos_cat, tipo d: float64
```

Con un código similar puede medir las proporciones de las categorías de ingresos en el conjunto de datos completo. [La Figura 2-10](#) compara las proporciones de las categorías de ingresos en el

conjunto de datos general, en el conjunto de prueba generado con muestreo estratificado y en un conjunto de prueba generado utilizando muestreo puramente aleatorio. Como puede ver, el conjunto de pruebas generado mediante muestreo estratificado tiene proporciones de categorías de ingresos casi idénticas a las del conjunto de datos completo, mientras que el conjunto de pruebas generado mediante muestreo puramente aleatorio está sesgado.

Income Category	Overall %	Stratified %	Random %	Strat. Error %	Rand. Error %
1	3.98	4.00	4.24	0.36	6.45
2	31.88	31.88	30.74	-0.02	-3.59
3	35.06	35.05	34.52	-0.01	-1.53
4	17.63	17.64	18.41	0.03	4.42
5	11.44	11.43	12.09	-0.08	5.63

Figura 2-10. Comparación del sesgo de muestreo del muestreo estratificado versus puramente aleatorio

No volverás a utilizar la columna `ingresos_cat`, por lo que también puedes eliminarla y revertir los datos a su estado original:

```
para set_in (strat_train_set, strat_test_set):
    set_.drop("ingresos_cat", eje=1, inplace=True)
```

Dedicamos bastante tiempo a la generación de conjuntos de pruebas por una buena razón: esta es una parte a menudo descuidada pero crítica de un proyecto de aprendizaje automático.

Además, muchas de estas ideas serán útiles más adelante cuando analicemos la validación cruzada. Ahora es el momento de pasar a la siguiente etapa: explorar los datos.

## Explore y visualice los datos para obtener información valiosa

Hasta ahora sólo ha echado un vistazo rápido a los datos para obtener una comprensión general del tipo de datos que está manipulando. Ahora el objetivo es profundizar un poco más.

Primero, asegúrese de haber dejado a un lado el conjunto de pruebas y de que solo esté explorando el conjunto de entrenamiento. Además, si el conjunto de entrenamiento es muy grande, es posible que desee probar un conjunto de exploración para que las manipulaciones sean fáciles y rápidas durante la fase de exploración. En este caso, el conjunto de entrenamiento es bastante pequeño, por lo que puedes trabajar directamente en el conjunto completo. Ya que vas a

Experimente con varias transformaciones del conjunto de entrenamiento completo, debe hacer una copia del original para poder volver a él después:

```
vivienda = strat_train_set.copy()
```

## Visualización de datos geográficos

Debido a que el conjunto de datos incluye información geográfica (latitud y longitud), es una buena idea crear un diagrama de dispersión de todos los distritos para visualizar los datos ([Figura 2-11](#)):

```
vivienda.plot(kind="dispersión", x="longitud", y="latitud", grid=True) plt.show()
```

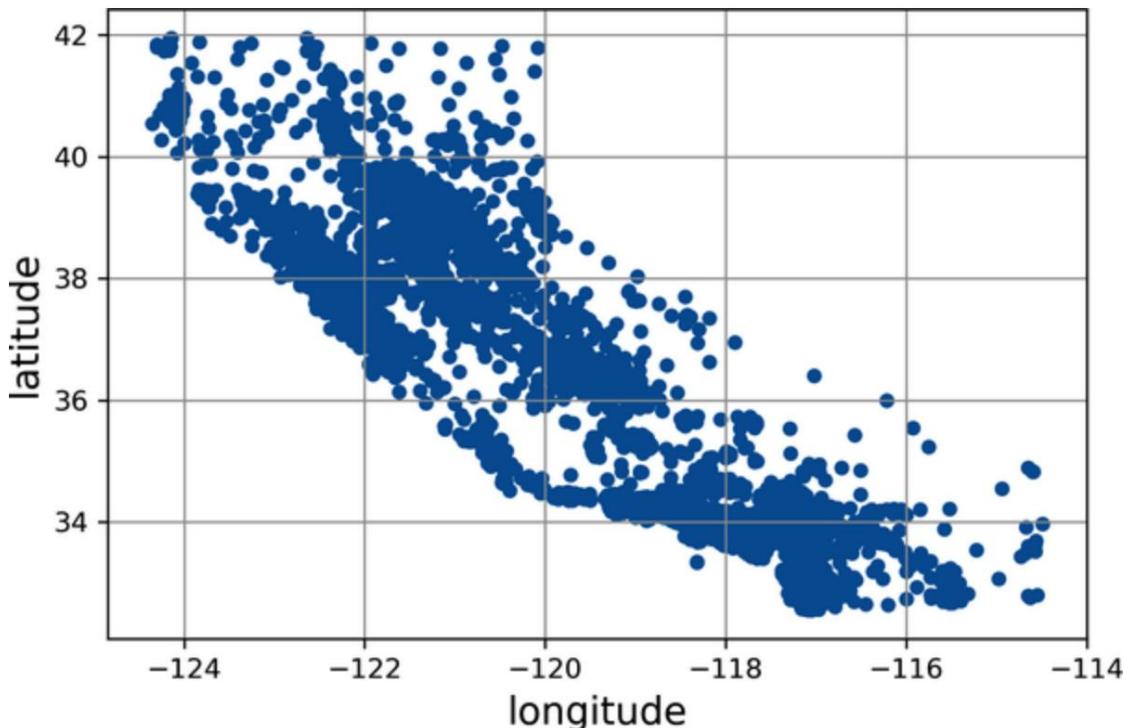


Figura 2-11. Un diagrama de dispersión geográfico de los datos.

Esto se parece a California, pero aparte de eso es difícil ver algún patrón en particular. Establecer la opción alfa en 0,2 hace que sea mucho más fácil visualizar los lugares donde hay una alta densidad de puntos de datos ([Figura 2-12](#)):

```
vivienda.plot(kind="scatter", x="longitud", y="latitud", grid=True, alpha=0.2) plt.show()
```

Eso es mucho mejor: se pueden ver claramente las áreas de alta densidad, es decir, el Área de la Bahía y alrededor de Los Ángeles y San Diego, además de una larga línea de áreas de densidad bastante alta en el Valle Central (en particular, alrededor de Sacramento y Fresno). .

Nuestros cerebros son muy buenos para detectar patrones en imágenes, pero es posible que tengas que jugar con los parámetros de visualización para que los patrones se destaque.

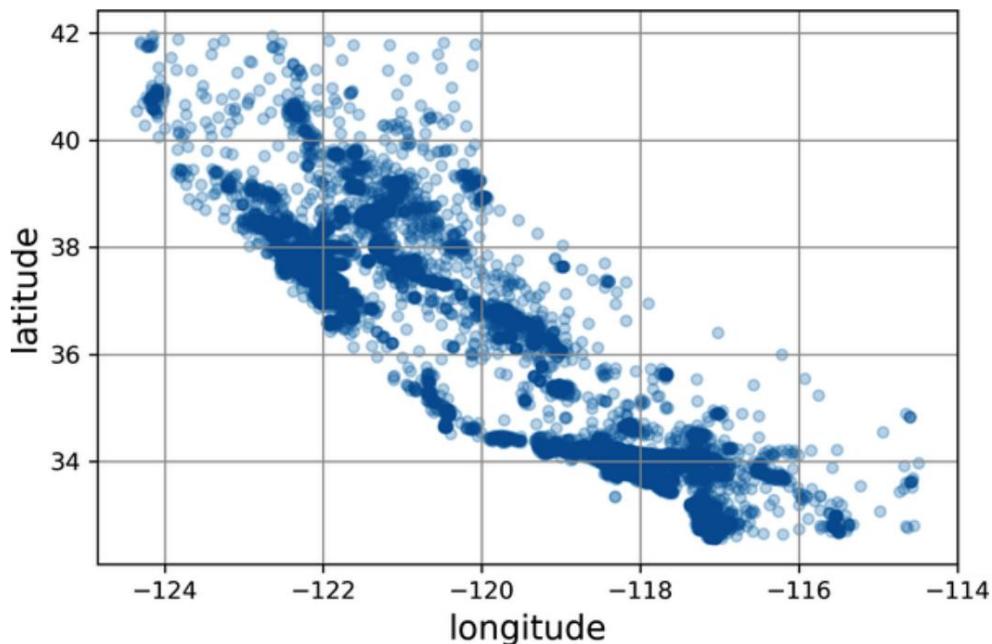


Figura 2-12. Una mejor visualización que resalta las áreas de alta densidad.

A continuación, nos fijamos en los precios de la vivienda ([Figura 2-13](#)). El radio de cada círculo representa la población del distrito (opción s) y el color representa el precio (opción c). Aquí se utiliza un mapa de colores predefinido (opción cmap) llamado jet, que va del azul (valores bajos) al rojo (precios altos):

8

```
vivienda.plot(kind="dispersión", x="longitud", y="latitud", grid=True,
s=vivienda["población"] / 100, etiqueta="población", c="valor_casa_mediana",
cmap="jet", colorbar=True,
```

```
leyenda=True, sharex=False, figsize=(10, 7))
plt.show()
```

Esta imagen le indica que los precios de la vivienda están muy relacionados con la ubicación (por ejemplo, cerca del océano) y con la densidad de población, como probablemente ya sabía. Un algoritmo de agrupamiento debería ser útil para detectar el grupo principal y agregar nuevas características que midan la proximidad a los centros del grupo. El atributo de proximidad al océano también puede ser útil, aunque en el norte de California los precios de la vivienda en los distritos costeros no son demasiado altos, por lo que no es una regla sencilla.

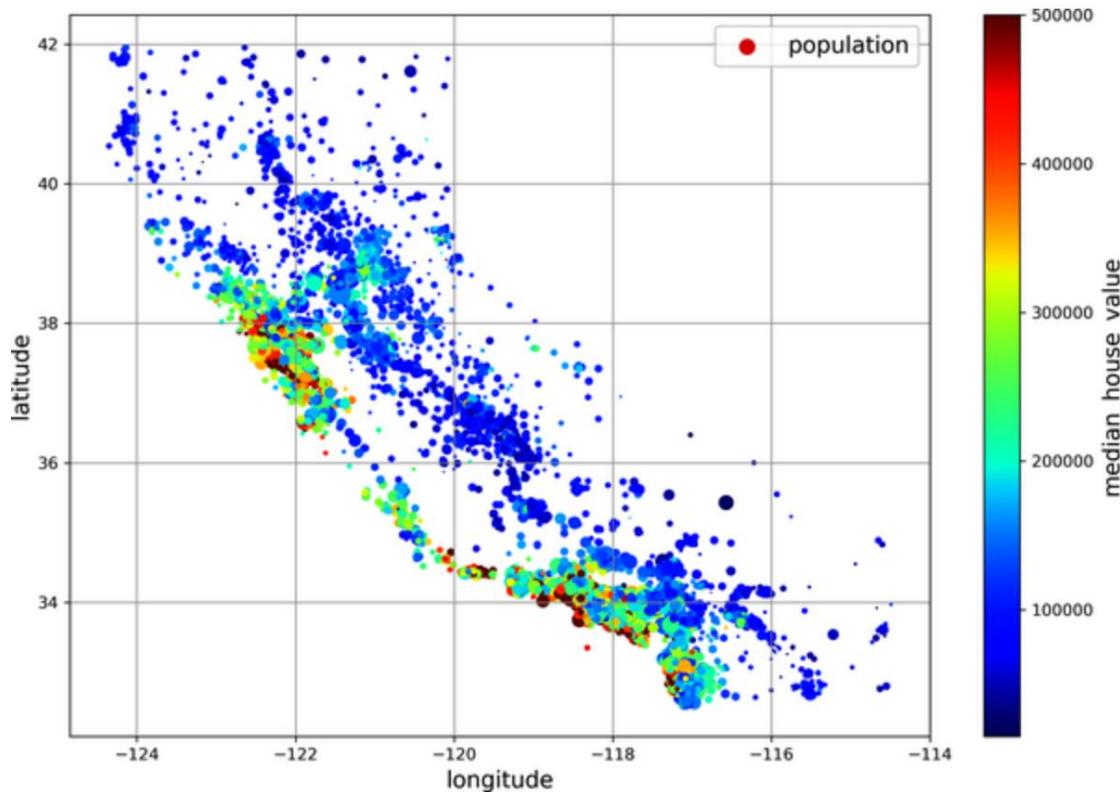


Figura 2-13. Precios de la vivienda en California: el rojo es caro, el azul es barato, los círculos más grandes indican áreas con mayor población

## Busque correlaciones

Dado que el conjunto de datos no es demasiado grande, puedes calcular fácilmente el coeficiente de correlación estándar (también llamado  $r$  de Pearson ) entre cada par de atributos usando el método `corr()`:

```
corr_matrix = vivienda.corr()
```

Ahora puedes ver cuánto se correlaciona cada atributo con la mediana. valor de la casa:

```
>>>
corr_matrix["median_house_value"].sort_values(ascendente=False)
valor_mediano de la casa      1.000000
ingreso_media                  0.688380
habitaciones_total              0.137455
edad_mediana de la vivienda   0.102175
hogares                         0.071426
población_total_dormitorios    0.054635
-0,020153
longitud -0.050859
latitud -0.139584
Nombre: median_house_value, tipo d: float64
```

El coeficiente de correlación varía de –1 a 1. Cuando está cerca de 1, significa que existe una fuerte correlación positiva; por ejemplo, la mediana El valor de la vivienda tiende a aumentar cuando aumenta el ingreso medio. Cuando el El coeficiente es cercano a –1, significa que hay un fuerte negativo. correlación; Puedes ver una pequeña correlación negativa entre la latitud. y el valor medio de la vivienda (es decir, los precios tienen una ligera tendencia a subir abajo cuando vas hacia el norte). Finalmente, coeficientes cercanos a 0 significan que hay sin correlación lineal.

Otra forma de comprobar la correlación entre atributos es utilizar el Función Pandas scatter\_matrix(), que traza cada número atributo frente a cualquier otro atributo numérico. Ya que ahora hay 11 atributos numéricos, obtendría  $11 = 121$  parcelas,<sup>2</sup> que no encajarían en una página, por lo que decide centrarse en algunos atributos prometedores que parecen más correlacionado con el valor medio de la vivienda ([Figura 2-14](#)):

```
de pandas.plotting importar scatter_matrix

atributos = ["valor_mediano_casa", "ingreso_mediano",
"total_habitaciones",
"edad_mediana_vivienda"]
scatter_matrix(vivienda[atributos], tamaño de figura=(12, 8))
plt.mostrar()
```

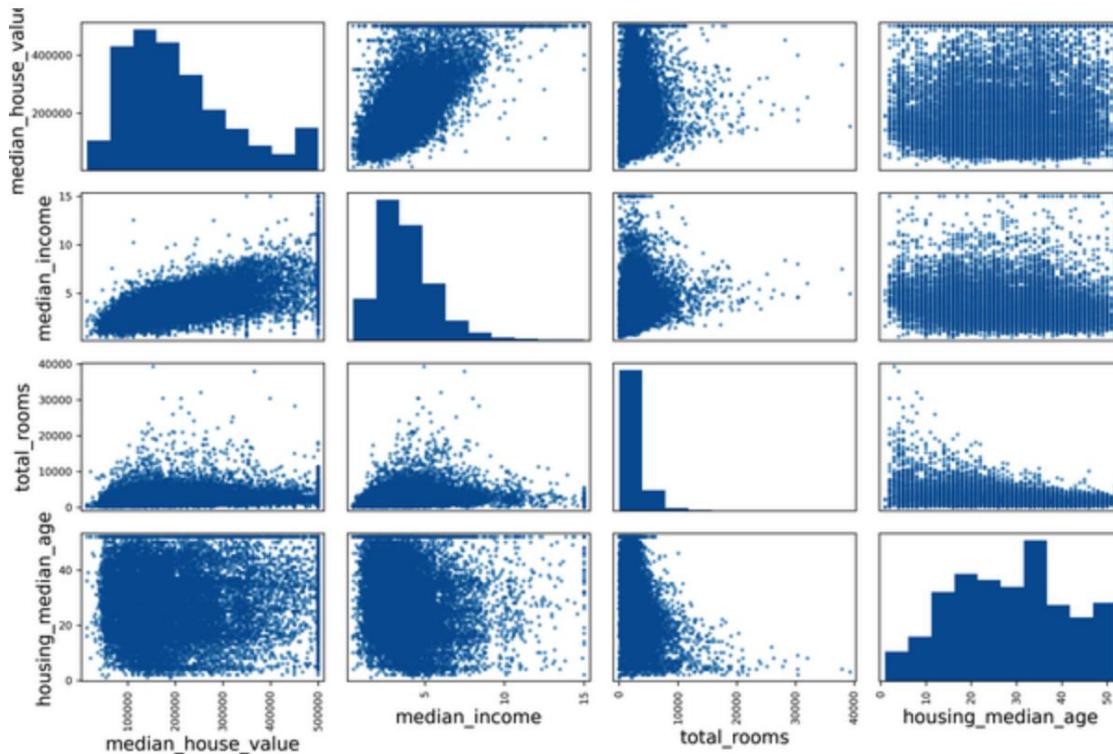


Figura 2-14. Esta matriz de dispersión traza cada atributo numérico frente a todos los demás atributos numéricos, además de un histograma de los valores de cada atributo numérico en la diagonal principal (de arriba a la izquierda a abajo a la derecha).

La diagonal principal estaría llena de líneas rectas si Pandas trazara cada variable contra sí misma, lo que no sería muy útil. Entonces, Pandas muestra un histograma de cada atributo (hay otras opciones disponibles; consulte la documentación de Pandas para obtener más detalles).

Al observar los diagramas de dispersión de correlación, parece que el atributo más prometedor para predecir el valor medio de una vivienda es el ingreso medio, por lo que se amplía su diagrama de dispersión ([Figura 2-15](#)):

```
vivienda.plot(kind="scatter", x="ingreso_mediano", y="valor_casa_mediano",
alpha=0.1, grid=True)
```

```
plt.mostrar()
```

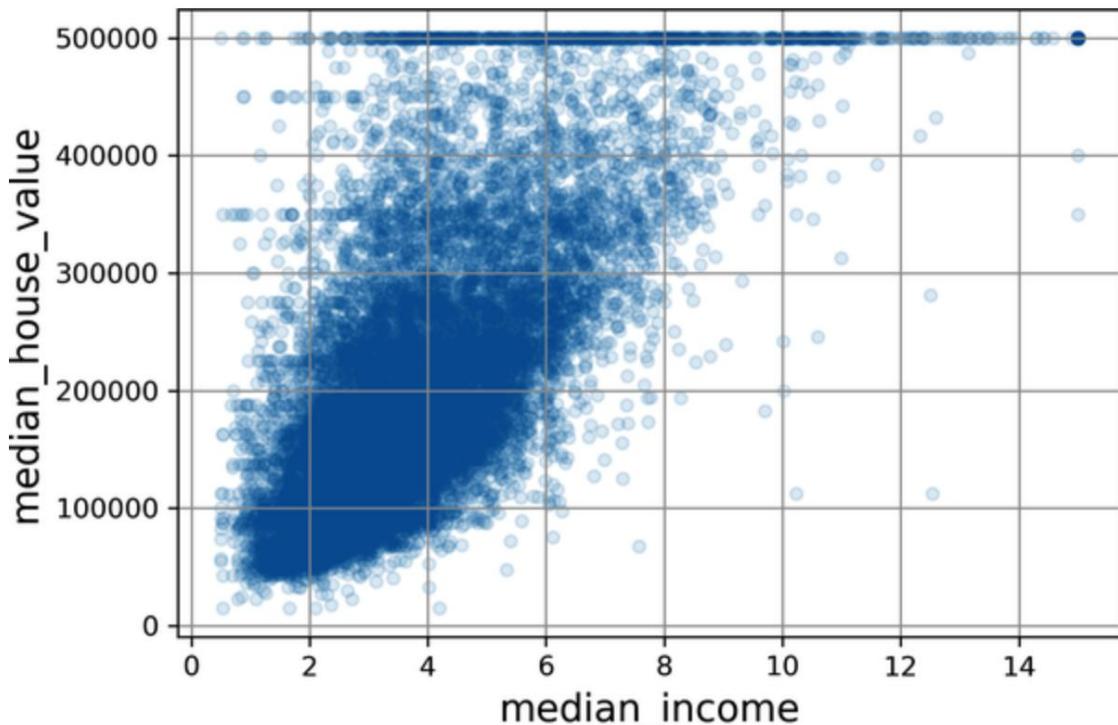


Figura 2-15. Ingreso medio versus valor medio de la vivienda

Esta trama revela algunas cosas. En primer lugar, la correlación es bastante fuerte; Se puede ver claramente la tendencia alcista y los puntos no están demasiado dispersos. En segundo lugar, el límite de precio que notó anteriormente es claramente visible como una línea horizontal en \$500,000. Pero el gráfico también revela otras líneas rectas menos obvias: una línea horizontal alrededor de \$450.000, otra alrededor de \$350.000, quizás una alrededor de \$280.000, y algunas más por debajo de esa. Es posible que desee intentar eliminar los distritos correspondientes para evitar que sus algoritmos aprendan a reproducir estas peculiaridades de los datos.

**ADVERTENCIA**

El coeficiente de correlación solo mide correlaciones lineales (“a medida que x aumenta, y generalmente sube/baja”). Es posible que pase por alto por completo las relaciones no lineales (por ejemplo, “a medida que x se acerca a 0, y generalmente aumenta”). La Figura 2-16 muestra una variedad de conjuntos de datos junto con su coeficiente de correlación. Observe cómo todos los gráficos de la fila inferior tienen un coeficiente de correlación igual a 0, a pesar de que sus ejes claramente no son independientes: estos son ejemplos de relaciones no lineales. Además, la segunda fila muestra ejemplos donde el coeficiente de correlación es igual a 1 o -1; Note que esto no tiene nada que ver con la pendiente. Por ejemplo, su altura en pulgadas tiene un coeficiente de correlación de 1 con su altura en pies o en nanómetros.

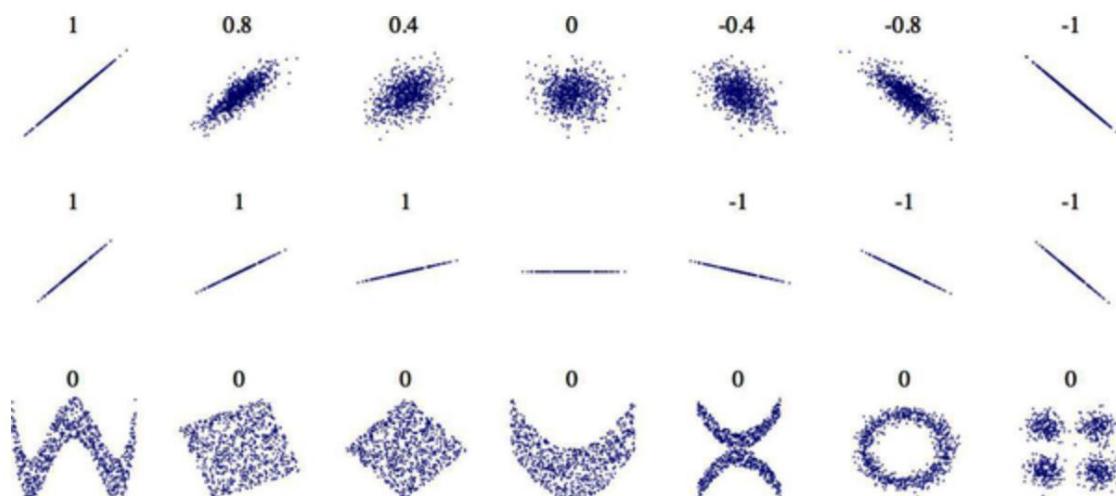


Figura 2-16. Coeficiente de correlación estándar de varios conjuntos de datos (fuente: Wikipedia; imagen de dominio público)

## Experimente con combinaciones de atributos

Esperamos que las secciones anteriores le hayan dado una idea de algunas formas en las que puede explorar los datos y obtener información. Identificó algunas peculiaridades de los datos que quizás desee limpiar antes de alimentar los datos a un algoritmo de aprendizaje automático y encontró correlaciones interesantes entre los atributos, en particular con el atributo objetivo. También notó que algunos atributos tienen una distribución asimétrica a la derecha, por lo que es posible que desee transformarlos (por ejemplo, calculando su logaritmo o raíz cuadrada). Por supuesto, su kilometraje variará considerablemente con cada proyecto, pero las ideas generales son similares.

Una última cosa que quizás quieras hacer antes de preparar los datos para la máquina. Aprender algoritmos consiste en probar varias combinaciones de atributos. Para Por ejemplo, el número total de habitaciones en un distrito no es muy útil si No sé cuántos hogares hay. Lo que realmente quieras es el número de habitaciones por hogar. Del mismo modo, el número total de dormitorios por sí solo no es muy útil: probablemente quieras compararlo con el número de habitaciones. Y la población por hogar también parece un dato interesante. combinación de atributos para mirar. Estos nuevos atributos se crean como sigue:

```
vivienda["habitaciones_por_casa"] = vivienda["total_habitaciones"] /  
vivienda["hogares"]  
vivienda["dormitorios_ratio"] = vivienda["total_dormitorios"] /  
vivienda["total_habitaciones"]  
vivienda["personas_por_casa"] = vivienda["población"] /  
vivienda["hogares"]
```

Y luego vuelves a mirar la matriz de correlación:

```
>>> corr_matrix = vivienda.corr()  
>>>  
corr_matrix["median_house_value"].sort_values(ascendente=False)  
median_house_value           1.000000  
ingresos_medianos          0.688380  
habitaciones_por_casa        0,143663  
total_habitaciones          0,137455  
vivienda_edad_media_hogares 0.102175  
                           0.071426  
población_total_dormitorios 0.054635  
                           -0,020153  
personas_por_casa           -0.038224  
longitud                      -0.050859  
latitud                      -0.139584  
proporción_dormitorios       -0.256397  
Nombre: median_house_value, tipo d: float64
```

¡Oye, no está mal! El nuevo atributo beds\_ratio es mucho más correlacionada con el valor medio de la casa que el número total de habitaciones o dormitorios. Aparentemente, las casas con una menor proporción de dormitorios por habitación tienden a ser más caro. El número de habitaciones por hogar también es mayor

más informativo que el número total de habitaciones en un distrito; obviamente, cuanto más grandes son las casas, más caras son.

Esta ronda de exploración no tiene por qué ser absolutamente exhaustiva; La cuestión es empezar con el pie derecho y obtener rápidamente información que le ayude a conseguir un primer prototipo razonablemente bueno. Pero este es un proceso iterativo: una vez que tienes un prototipo en funcionamiento, puedes analizar su resultado para obtener más información y volver a este paso de exploración.

**Prepare los datos para los algoritmos de aprendizaje automático** Es hora de preparar los datos para sus algoritmos de aprendizaje automático. En lugar de hacer esto manualmente, deberías escribir funciones para este propósito, por varias buenas razones:

- Esto le permitirá reproducir estas transformaciones fácilmente en cualquier conjunto de datos (por ejemplo, la próxima vez que obtenga un conjunto de datos nuevo).
- Gradualmente creará una biblioteca de funciones de transformación que podrá reutilizar en proyectos futuros.
- Puede utilizar estas funciones en su sistema en vivo para transformar los nuevos datos antes de alimentarlos a sus algoritmos.
- Esto le permitirá probar fácilmente varias transformaciones y ver qué combinación de transformaciones funciona mejor.

Pero primero, vuelva a un conjunto de entrenamiento limpio (copiando `strat_train_set` una vez más). También debes separar los predictores y las etiquetas, ya que no necesariamente deseas aplicar las mismas transformaciones a los predictores y los valores objetivo (nota que `drop()` crea una copia de los datos y no afecta a `strat_train_set`):

```
vivienda = strat_train_set.drop("valor_mediano_casa", eje=1) etiquetas_vivienda = estrat_train_set["valor_mediano_casa"].copiar()
```

## Limpiar los datos

La mayoría de los algoritmos de aprendizaje automático no pueden funcionar con funciones faltantes, por lo que deberás encargarte de ellas. Por ejemplo, usted notó antes que el

Al atributo `total_bedrooms` le faltan algunos valores. Tienes tres opciones para solucionar este problema:

1. Deshacerse de los distritos correspondientes.
2. Deshazte de todo el atributo.
3. Establezca los valores faltantes en algún valor (cero, media, mediana, etc.). Esto se llama imputación.

Puede lograr esto fácilmente utilizando los métodos `dropna()`, `drop()` y `fillna()` de Pandas DataFrame:

```
vivienda.dropna(subset=["total_dormitorios"], inplace=True) # opción 1
```

```
vivienda.drop("total_dormitorios", eje=1) # opción 2
```

```
mediana = vivienda["total_dormitorios"].median() # opción 3
vivienda["total_dormitorios"].fillna(mediana, inplace=True)
```

Decides optar por la opción 3 porque es la menos destructiva, pero en lugar del código anterior, utilizarás una práctica clase de Scikit-Learn: `SimpleImputer`. El beneficio es que almacenará el valor mediano de cada característica: esto permitirá imputar valores faltantes no solo en el conjunto de entrenamiento, sino también en el conjunto de validación, el conjunto de prueba y cualquier dato nuevo introducido en el modelo. Para usarlo, primero necesita crear una instancia de `SimpleImputer`, especificando que desea reemplazar los valores faltantes de cada atributo con la mediana de ese atributo:

```
desde sklearn.impute importar SimpleImputer
imputador = SimpleImputer(estrategia="mediana")
```

Dado que la mediana solo se puede calcular con atributos numéricos, deberá crear una copia de los datos solo con los atributos numéricos (esto excluirá el atributo de texto `ocean_proximity`):

```
num_vivienda = vivienda.select_dtypes(include=[np.number])
```

Ahora puede ajustar la instancia del imputador a los datos de entrenamiento usando el método de ajuste():

```
imputer.fit(num_vivienda)
```

El imputador simplemente ha calculado la mediana de cada atributo y almacenó el resultado en su variable de instancia stats\_. Solo el Al atributo total\_bedrooms le faltaban valores, pero no puede estar seguro que no habrá valores faltantes en los datos nuevos después de que el sistema funcione. live, por lo que es más seguro aplicar el imputador a todos los atributos numéricos:

```
>>> imputador.estadisticas_
matriz([-118.51, 34.26 3.5385])    , 29.    , 2125.    , 434.    , 1167.    , 408.    ,
>>> num_vivienda.mediana().valores
matriz([-118.51, 2125. 3.5385])    , 29.    ,           , 434.    , 1167.    , 408.    ,
```

Ahora puede utilizar este imputador "entrenado" para transformar el conjunto de entrenamiento mediante reemplazando los valores faltantes con las medianas aprendidas:

```
X = imputador.transformar(num_vivienda)
```

Los valores faltantes también se pueden reemplazar con el valor medio. (estrategia="media"), o con el valor más frecuente (estrategia="most\_frequent"), o con un valor constante (estrategia="constante", valor\_relleno=...). Las dos últimas estrategias Admite datos no numéricos.

CONSEJO

También hay imputadores más potentes disponibles en el paquete `sklearn.impute` (ambos solo para funciones numéricas):

- `KNNImputer` reemplaza cada valor faltante con la media de los valores de los  $k$  vecinos más cercanos para esa característica. La distancia se basa en todas las funciones disponibles.
- `IterativeImputer` entrena un modelo de regresión por función para predecir los valores faltantes en función de todas las demás funciones disponibles. Luego entrena el modelo nuevamente con los datos actualizados y repite el proceso varias veces, mejorando los modelos y los valores de reemplazo en cada iteración.

## DISEÑO DE APRENDIZAJE SCIKIT

La API de Scikit-Learn está notablemente bien diseñada. Estos son los **principales principios de diseño:**<sup>9</sup>

### Consistencia

Todos los objetos comparten una interfaz consistente y simple:

### Estimadores

Cualquier objeto que pueda estimar algunos parámetros basándose en un conjunto de datos se denomina estimador (por ejemplo, un SimpleImputer es un estimador). La estimación en sí se realiza mediante el método fit() y toma un conjunto de datos como parámetro, o dos para los algoritmos de aprendizaje supervisado; el segundo conjunto de datos contiene las etiquetas. Cualquier otro parámetro necesario para guiar el proceso de estimación se considera un hiperparámetro (como la estrategia de SimpleImputer) y debe establecerse como una variable de instancia (generalmente mediante un parámetro de constructor).

### transformadores

Algunos estimadores (como SimpleImputer) también pueden transformar un conjunto de datos; estos se llaman transformadores. Una vez más, la API es simple: la transformación se realiza mediante el método transform() con el conjunto de datos a transformar como parámetro. Devuelve el conjunto de datos transformado. Esta transformación generalmente se basa en los parámetros aprendidos, como es el caso de SimpleImputer. Todos los transformadores también tienen un método conveniente llamado fit\_transform(), que equivale a llamar a fit() y luego a transform() (pero a veces fit\_transform() está optimizado y se ejecuta mucho más rápido).

### Predictores

Finalmente, algunos estimadores, dado un conjunto de datos, son capaces de hacer predicciones; se les llama predictores. Por ejemplo, el modelo de Regresión Lineal del capítulo anterior era un

Predictor: dado el PIB per cápita de un país, predijo la satisfacción con la vida. Un predictor tiene un método `predict()` que toma un conjunto de datos de nuevas instancias y devuelve un conjunto de datos de las predicciones correspondientes. También cuenta con un método `score()` que mide la calidad de las predicciones, dado un conjunto de pruebas (y las etiquetas correspondientes, en el caso de aprendizaje supervisado).  
algoritmos). <sup>10</sup>

### Inspección

Se puede acceder a todos los hiperparámetros del estimador directamente a través de variables de instancia públicas (por ejemplo, `imputer.strategy`), y a todos los parámetros aprendidos del estimador se puede acceder a través de variables de instancia públicas con un sufijo de subrayado (por ejemplo, `imputer.statistics_`).

### No proliferación de clases

Los conjuntos de datos se representan como matrices NumPy o matrices dispersas SciPy, en lugar de clases caseras. Los hiperparámetros son simplemente cadenas o números normales de Python.

### Composición

Los bloques de construcción existentes se reutilizan tanto como sea posible. Por ejemplo, es fácil crear un estimador Pipeline a partir de una secuencia arbitraria de transformadores seguida de un estimador final, como verá.

#### Valores predeterminados sensatos

Scikit-Learn proporciona valores predeterminados razonables para la mayoría de los parámetros, lo que facilita la creación rápida de un sistema de trabajo básico.

Los transformadores Scikit-Learn generan matrices NumPy (o, a veces, matrices dispersas SciPy) incluso cuando reciben Pandas DataFrames como entrada.

Entonces, la salida de `imputer.transform(housing_num)` es un NumPy

<sup>11</sup>

matriz: X no tiene nombres de columnas ni índice. Por suerte, no es demasiado difícil envuelva X en un DataFrame y recupere los nombres de las columnas y el índice de número\_vivienda:

```
vivienda_tr = pd.DataFrame(X, columnas=vivienda_num.columnas,
                            índice=número_vivienda.índice)
```

## Manejo de texto y atributos categóricos

Hasta ahora sólo hemos tratado con atributos numéricos, pero sus datos también pueden contener atributos de texto. En este conjunto de datos, solo hay uno: el atributo ocean\_proximity. Veamos su valor durante los primeros instancias:

```
>>> vivienda_cat = vivienda[["ocean_proximity"]]
>>> vivienda_cat.head(8)
   proximidad_oceánica
13096      CERCA DE LA BAHÍA
14973      <1H OCÉANO
3785        INTERIOR
14689        INTERIOR
20507      CERCA DEL OCÉANO
1286        INTERIOR
18078      <1H OCÉANO
4396      CERCA DE LA BAHÍA
```

No es texto arbitrario: hay un número limitado de valores posibles, cada uno de los cuales representa una categoría. Entonces este atributo es un atributo categórico. La mayoría de los algoritmos de aprendizaje automático prefieren trabajar con números, así que convierta estas categorías de texto a números. Para esto, podemos usar la clase OrdinalEncoder de Scikit-Learn:

```
desde sklearn.preprocesamiento importar OrdinalEncoder
```

```
ordinal_encoder = Codificador ordinal()
vivienda_cat_codificada =
ordinal_encoder.fit_transform(vivienda_cat)
```

Así es como se ven los primeros valores codificados en housing\_cat\_encoded como:

```
>>> vivienda_cat_encoded[:8]
array([[3.], [0.],
       [1.],
       [1.],
       [4.],
       [1.],
       [0.],
       [3.]])
```

Puede obtener la lista de categorías utilizando la variable de instancia `categorías_`. Es una lista que contiene una matriz 1D de categorías para cada atributo categórico (en este caso, una lista que contiene una matriz única ya que solo hay un atributo categórico):

```
>>> ordinal_encoder.categories_
array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'], dtype=object)
```

Un problema con esta representación es que los algoritmos de ML asumirán que dos valores cercanos son más similares que dos valores distantes. Esto puede estar bien en algunos casos (por ejemplo, para categorías ordenadas como "malo", "promedio", "bueno" y "excelente"), pero obviamente no es el caso para la columna `ocean_proximity` (por ejemplo, categorías 0 y 4 son claramente más similares que las categorías 0 y 1). Para solucionar este problema, una solución común es crear un atributo binario por categoría: un atributo igual a 1 cuando la categoría es "<1H OCEAN" (y 0 en caso contrario), otro atributo igual a 1 cuando la categoría es "INLAND" (y 0 en caso contrario), y así sucesivamente. Esto se llama codificación one-hot, porque solo un atributo será igual a 1 (caliente), mientras que los demás serán 0 (frío). Los nuevos atributos a veces se denominan atributos ficticios. Scikit-Learn proporciona una clase `OneHotEncoder` para convertir valores categóricos en vectores one-hot:

```
desde sklearn.preprocessing importar OneHotEncoder

cat_encoder = OneHotEncoder()
vivienda_cat_1hot = cat_encoder.fit_transform(vivienda_cat)
```

De forma predeterminada, la salida de OneHotEncoder es una matriz dispersa de SciPy, en lugar de una matriz NumPy:

```
>>> housing_cat_1hot <16512x5
matriz dispersa de tipo '<clase 'numpy.float64'>
con 16512 elementos almacenados en formato de fila dispersa comprimida>
```

Una matriz dispersa es una representación muy eficiente para matrices que contienen principalmente ceros. De hecho, internamente sólo almacena los valores distintos de cero y sus posiciones. Cuando un atributo categórico tiene cientos o miles de categorías, la codificación one-hot da como resultado una matriz muy grande llena de ceros, excepto un único 1 por fila. En este caso, una matriz dispersa es exactamente lo que necesita: ahorrará mucha memoria y acelerará los cálculos. Tú

Puede usar una matriz dispersa principalmente como una matriz 2D <sup>12</sup> normal, pero si desea convertirla en una matriz NumPy (densa), simplemente llame al método toarray():

```
>>> housing_cat_1hot.toarray() matriz([[0., 0.,
0., 1., 0.], [1., 0., 0., 0., 0.], [0., 1., 0., 0., 0.],
..., [0., 0., 0., 0., 1.], [1., 0., 0., 0., 0.],
[0., 0., 0., 0., 1.]])
```

Alternativamente, puede establecer sparse=False al crear OneHotEncoder, en cuyo caso el método transform() devolverá una matriz NumPy regular (densa) directamente.

Al igual que con OrdinalEncoder, puede obtener la lista de categorías utilizando la variable de instancia groups\_ del codificador:

```
>>> cat_encoder.categories_ [array(['<1H
OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'], dtype=object)]
```

Pandas tiene una función llamada get\_dummies(), que también convierte cada característica categórica en una representación única, con una característica binaria por categoría:

```
>>> df_test = pd.DataFrame({"ocean_proximity": ["INLAND", "NEAR BAY"]}) >>>

pd.get_dummies(df_test)
ocean_proximity_INLAND ocean_proximity_NEAR BAY 0 1
1
0 1
0 0
```

Se ve bonito y simple, así que ¿por qué no usarlo en lugar de OneHotEncoder? Bueno, la ventaja de OneHotEncoder es que recuerda en qué categorías fue entrenado. Esto es muy importante porque una vez que su modelo esté en producción, debe recibir exactamente las mismas características que durante el entrenamiento: ni más ni menos. Mire lo que genera nuestro cat\_encoder entrenado cuando lo hacemos transformar el mismo df\_test (usando transform(), no fit\_transform()):

```
>>> cat_encoder.transform(df_test) matriz([[0., 1.,
0., 0., 0.], [0., 0., 0., 1., 0.]])
```

¿Ver la diferencia? get\_dummies() vio solo dos categorías, por lo que generó dos columnas, mientras que OneHotEncoder generó una columna por categoría aprendida, en el orden correcto. Además, si alimenta a get\_dummies() con un DataFrame que contiene una categoría desconocida (por ejemplo, "<2H OCEAN"), felizmente generará una columna para ello:

```
>>> df_test_unknown = pd.DataFrame({"ocean_proximity": ["<2H OCEAN", "ISLAND"]}) >>>
pd.get_dummies(df_test_unknown)

ocean_proximity_<2H OCEAN ocean_proximity_ISLAND 0
0
1
1
0
```

Pero OneHotEncoder es más inteligente: detectará la categoría desconocida y generará una excepción. Si lo prefiere, puede configurar el hiperparámetro handle\_unknown en "ignorar", en cuyo caso simplemente representará la categoría desconocida con ceros:

```
>>> cat_encoder.handle_unknown = "ignorar" >>>
cat_encoder.transform(df_test_unknown)
```

```
matriz([[0., 0., 0., 0., 0.], [0., 0., 1., 0., 0.]])
```

## CONSEJO

Si un atributo categórico tiene una gran cantidad de categorías posibles (por ejemplo, código de país, profesión, especie), entonces la codificación one-hot dará como resultado una gran cantidad de características de entrada. Esto puede ralentizar el entrenamiento y degradar el rendimiento. Si esto sucede, es posible que desee reemplazar la entrada categórica con características numéricas útiles relacionadas con las categorías: por ejemplo, podría reemplazar la característica ocean\_proximity con la distancia al océano (de manera similar, un código de país podría reemplazarse con la población del país y PIB per cápita). Alternativamente, puede utilizar uno de los codificadores proporcionados por el paquete Category\_encoders en [GitHub](#). O, cuando se trata de redes neuronales, puede reemplazar cada categoría con un vector de baja dimensión que se puede aprender llamado incrustación. Este es un ejemplo de aprendizaje de representación (véanse los Capítulos 13 y 17 para obtener más detalles).

Cuando ajusta cualquier estimador de Scikit-Learn usando un DataFrame, el estimador almacena los nombres de las columnas en el atributo feature\_names\_in\_. Scikit-Learn luego garantiza que cualquier DataFrame enviado a este estimador después de eso (por ejemplo, para transformar() o predecir()) tenga los mismos nombres de columna. Los transformadores también proporcionan un método get\_feature\_names\_out() que puedes usar para construir un DataFrame alrededor de la salida del transformador:

```
>>> cat_encoder.feature_names_in_
array(['ocean_proximity'], dtype=object) >>>
cat_encoder.get_feature_names_out()
array(['ocean_proximity_<1H OCEAN', 'ocean_proximity_INLAND',
       'ocean_proximity_ISLAND', 'ocean_proximity_NEAR BAY',
       'ocean_proximity_NEAR OCEAN'], dtype=object) >>>
df_output =
pd.DataFrame(cat_encoder.transform(df_test_unknown),
...
columnas=cat_encoder.get_feature_names_out(),
...                                         indice=df_test_unknown.index)
... 
```

## Escalado y transformación de funciones

Una de las transformaciones más importantes que debe aplicar a sus datos es el escalado de funciones. Con pocas excepciones, los algoritmos de aprendizaje automático no funcionan bien cuando los atributos numéricos de entrada tienen escalas muy diferentes. Este es el caso de los datos de vivienda: el número total de habitaciones oscila entre 6 y 39.320, mientras que los ingresos medios sólo oscilan entre 0 y 15. Sin ninguna escala, la mayoría de los modelos estarán sesgados hacia ignorar el ingreso medio y centrarse más en el número de habitaciones.

Hay dos formas comunes de conseguir que todos los atributos tengan la misma escala: escalamiento mínimo-máximo y estandarización.

#### ADVERTENCIA

Como ocurre con todos los estimadores, es importante ajustar los escaladores únicamente a los datos de entrenamiento: nunca use `fit()` o `fit_transform()` para nada más que el conjunto de entrenamiento. Una vez que tenga un escalador capacitado, podrá usarlo para `transform()` cualquier otro conjunto, incluido el conjunto de validación, el conjunto de prueba y los datos nuevos. Tenga en cuenta que, si bien los valores del conjunto de entrenamiento siempre se escalarán al rango especificado, si los datos nuevos contienen valores atípicos, estos pueden terminar escalados fuera del rango. Si desea evitar esto, simplemente establezca el hiperparámetro `clip` en Verdadero.

La escala mínima-máxima (muchas personas llaman a esto normalización) es la más simple: para cada atributo, los valores se desplazan y se reescalan para que terminen oscilando entre 0 y 1. Esto se realiza restando el valor mínimo y dividiéndolo por la diferencia entre el mínimo y el máximo. Scikit-Learn proporciona un transformador llamado `MinMaxScaler` para esto. Tiene un hiperparámetro `feature_range` que le permite cambiar el rango si, por alguna razón, no desea 0–1 (por ejemplo, las redes neuronales funcionan mejor con entradas de media cero, por lo que es preferible un rango de –1 a 1). Es bastante fácil usar:

```
desde sklearn.preprocessing importar MinMaxScaler
```

```
min_max_scaler = MinMaxScaler(feature_range=(-1, 1))
num_vivienda_min_max_scaled =
min_max_scaler.fit_transform(num_vivienda)
```

La estandarización es diferente: primero resta el valor medio (por lo que los valores estandarizados tienen una media cero), luego divide el resultado por la desviación estándar (por lo que los valores estandarizados tienen una desviación estándar igual a 1). A diferencia del escalamiento mínimo-máximo, la estandarización no restringe los valores a un rango específico. Sin embargo, la estandarización se ve mucho menos afectada por los valores atípicos. Por ejemplo, supongamos que un distrito tiene un ingreso medio igual a 100 (por error), en lugar del habitual 0-15. La escala mínima-máxima al rango de 0 a 1 asignaría este valor atípico a 1 y aplastaría todos los demás valores a 0 a 0,15, mientras que la estandarización no se vería muy afectada. Scikit-Learn proporciona un transformador llamado StandardScaler para la estandarización:

```
desde sklearn.preprocessing importar StandardScaler
```

```
std_scaler = StandardScaler()
num_vivienda_std_scaled = std_scaler.fit_transform(num_vivienda)
```

#### CONSEJO

Si desea escalar una matriz dispersa sin convertirla primero en una matriz densa, puede usar un StandardScaler con su hiperparámetro `with_mean` establecido en `False`: solo dividirá los datos por la desviación estándar, sin restar la media (ya que esto rompería escasez).

Cuando la distribución de una característica tiene una cola pesada (es decir, cuando los valores alejados de la media no son exponencialmente raros), tanto el escalamiento mínimo-máximo como la estandarización reducirán la mayoría de los valores a un rango pequeño. A los modelos de aprendizaje automático generalmente no les gusta esto en absoluto, como verá en [el Capítulo 4](#). Entonces, antes de escalar la característica, primero debes transformarla para reducir la cola pesada y, si es posible, hacer que la distribución sea aproximadamente simétrica. Por ejemplo, una forma común de hacer esto para entidades positivas con una cola pesada hacia la derecha es reemplazar la entidad con su raíz cuadrada (o elevar la entidad a una potencia entre 0 y 1). Si la característica tiene una cola muy larga y pesada, como una distribución de ley de potencia, entonces puede ser útil reemplazar la característica con su logaritmo. Por ejemplo, la característica de población sigue aproximadamente una ley potencial: los distritos con 10.000 habitantes son sólo 10 veces menos frecuentes que los distritos con 1.000 habitantes, no exponencialmente menos.

frecuente. La Figura 2-17 muestra cuánto mejor se ve esta característica cuando se calcula su registro: está muy cerca de una distribución gaussiana (es decir, tiene forma de campana).

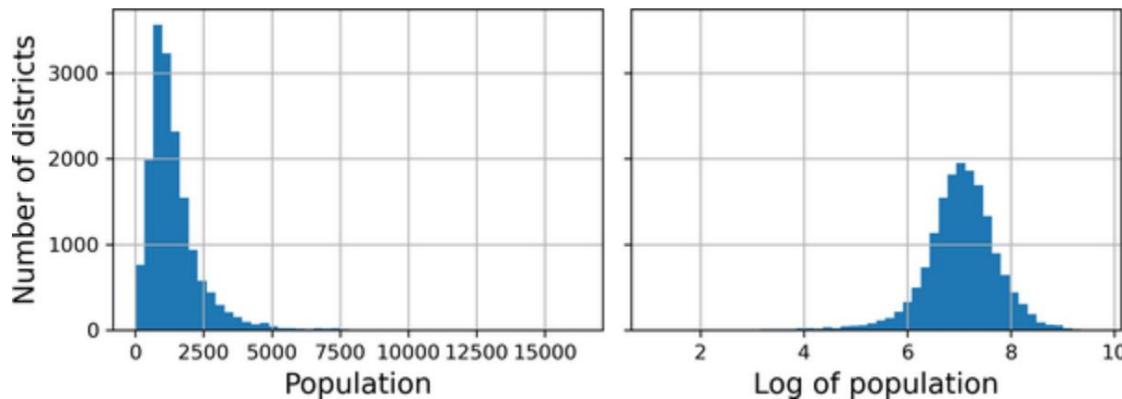


Figura 2-17. Transformar una característica para acercarla a una distribución gaussiana

Otro enfoque para manejar características de cola pesada consiste en agrupar la característica en cubos . Esto significa dividir su distribución en grupos de aproximadamente el mismo tamaño y reemplazar cada valor de característica con el índice del grupo al que pertenece, de manera muy similar a lo que hicimos para crear la característica `ingresos_cat` (aunque solo la usamos para muestreo estratificado). Por ejemplo, podrías reemplazar cada valor con su percentil. La agrupación en depósitos con depósitos del mismo tamaño da como resultado una característica con una distribución casi uniforme, por lo que no es necesario realizar más escalas, o simplemente puede dividir por el número de depósitos para forzar los valores al rango de 0 a 1.

Cuando una característica tiene una distribución multimodal (es decir, con dos o más picos claros, llamados modos), como la característica `housing_median_age`, también puede ser útil agruparla, pero esta vez tratando los ID de los depósitos como categorías, en lugar de números. Esto significa que los índices de los depósitos deben codificarse, por ejemplo, utilizando un OneHotEncoder (por lo que normalmente no querrás utilizar demasiados depósitos). Este enfoque permitirá que el modelo de regresión aprenda más fácilmente diferentes reglas para diferentes rangos de este valor de característica. Por ejemplo, quizás las casas construidas hace unos 35 años tengan un estilo peculiar que pasó de moda y, por lo tanto, sean más baratas de lo que sugeriría su antigüedad.

Otro enfoque para transformar las distribuciones multimodales es agregar una característica para cada uno de los modos (al menos los principales), que represente la similitud entre la edad media de la vivienda y ese modo en particular. El

La medida de similitud generalmente se calcula utilizando una función de base radial (RBF), cualquier función que dependa únicamente de la distancia entre el valor de entrada y un punto fijo. El RBF más comúnmente utilizado es el RBF gaussiano, cuyo valor de salida decrece exponencialmente a medida que el valor de entrada se aleja del punto fijo. Por ejemplo, la similitud Gaussiana RBF entre la edad de vivienda  $x$  y 35 viene dada por la ecuación  $\exp(-\gamma(x - 35)^2)$ . El hiperparámetro  $\gamma$  (gamma) determina qué tan rápido decrece la medida de similitud a medida que  $x$  se aleja de 35. Usando la función `rbf_kernel()` de Scikit-Learn, puede crear una nueva característica Gaussiana RBF que mida la similitud entre la edad media de la vivienda y 35:

```
desde sklearn.metrics.pairwise importar rbf_kernel  
  
edad_simil_35 = rbf_kernel(vivienda[["vivienda_median_age"]], [[35]], gamma=0.1)
```

La Figura 2-18 muestra esta nueva característica en función de la edad media de la vivienda (línea continua). También muestra cómo se vería la característica si usara un valor  $\gamma$  más pequeño. Como muestra el gráfico, la nueva característica de similitud de edad alcanza su punto máximo a los 35 años, justo alrededor del pico en la distribución de edad media de la vivienda: si este grupo de edad en particular está bien correlacionado con precios más bajos, hay muchas posibilidades de que esta nueva característica ayude.

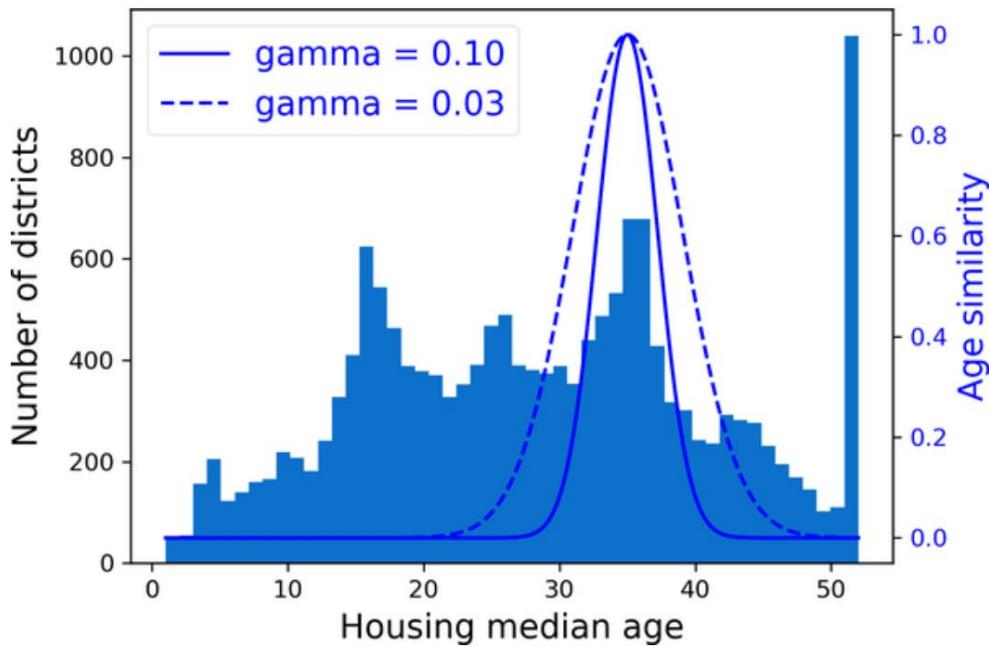


Figura 2-18. Función RBF gaussiana que mide la similitud entre la edad media de la vivienda y los 35 años

Hasta ahora solo hemos analizado las características de entrada, pero es posible que también sea necesario transformar los valores objetivo. Por ejemplo, si la distribución objetivo tiene una cola pesada, puede optar por reemplazar el objetivo con su logaritmo. Pero si lo hace, el modelo de regresión ahora predecirá el logaritmo del valor medio de la vivienda, no el valor medio de la vivienda en sí. Deberá calcular la exponencial de la predicción del modelo si desea obtener el valor medio de la vivienda previsto.

Afortunadamente, la mayoría de los transformadores de Scikit-Learn tienen un método `inverse_transform()`, lo que facilita calcular la inversa de sus transformaciones. Por ejemplo, el siguiente ejemplo de código muestra cómo escalar las etiquetas usando un `StandardScaler` (tal como lo hicimos para las entradas), luego entrenar un modelo de regresión lineal simple en las etiquetas escaladas resultantes y usarlo para hacer predicciones sobre algunos datos nuevos, que transforme de nuevo a la escala original utilizando el método `inverse_transform()` del escalador entrenado. Tenga en cuenta que convertimos las etiquetas de una serie Pandas a un `DataFrame`, ya que `StandardScaler` espera entradas 2D. Además, en este ejemplo simplemente entrenamos el modelo en una única característica de entrada sin procesar (ingreso medio), por simplicidad:

```

de sklearn.linear_model importar LinearRegression

target_scaler = StandardScaler()
etiquetas_escaladas =
target_scaler.fit_transform(housing_labels.to_frame())

model = LinearRegression()
model.fit(housing[["median_Income"]], scaled_labels) some_new_data =
vivienda[["median_Income"]].iloc[:5] # finge que estos son datos nuevos

predicciones_escaladas = model.predict(algunos_nuevos_datos)
predicciones =
target_scaler.inverse_transform(predicciones_escaladas)

```

Esto funciona bien, pero una opción más sencilla es utilizar `TransformedTargetRegressor`. Sólo necesitamos construirlo, dándole el modelo de regresión y el transformador de etiquetas, luego ajustarlo al conjunto de entrenamiento, usando las etiquetas originales sin escala. Utilizará automáticamente el transformador para escalar las etiquetas y entrenar el modelo de regresión en las etiquetas escaladas resultantes, tal como lo hicimos anteriormente. Luego, cuando queramos hacer una predicción, llamará al método `predict()` del modelo de regresión y utilizará el método `inverse_transform()` del escalador para producir la predicción:

```

desde sklearn.compose importar TransformedTargetRegressor

modelo = TransformedTargetRegressor (LinearRegression(),

transformador=StandardScaler())
model.fit(vivienda[["ingresos_medianos"]], etiquetas_vivienda) predicciones =
model.predict(algunos_nuevos_datos)

```

## Transformadores personalizados

Aunque Scikit-Learn proporciona muchos transformadores útiles, necesitará escribir los suyos propios para tareas como transformaciones personalizadas, operaciones de limpieza o combinación de atributos específicos.

Para transformaciones que no requieren ningún entrenamiento, simplemente puede escribir una función que tome una matriz NumPy como entrada y genere la matriz transformada. Por ejemplo, como se analizó en la sección anterior, a menudo es una buena

idea de transformar características con distribuciones de colas pesadas reemplazándolas con su logaritmo (asumiendo que la característica es positiva y la cola está a la derecha). Creemos un transformador logarítmico y apliquémoslo a la característica de población:

```
de sklearn.preprocessing importar FunctionTransformer

log_transformer = FunctionTransformer(np.log, inverse_func=np.exp)
log_pop =
log_transformer.transform(vivienda[["población"]])
```

El argumento `inverse_func` es opcional. Le permite especificar una función de transformación inversa, por ejemplo, si planea usar su transformador en un `TransformedTargetRegressor`.

Su función de transformación puede tomar hiperparámetros como argumentos adicionales. Por ejemplo, aquí se explica cómo crear un transformador que calcule la misma medida de similitud Gaussiana RBF que antes:

```
rbf_transformer = FunctionTransformer(rbf_kernel, kw_args=dict(Y=[[35.]],
gamma=0.1))
edad_simil_35 =
rbf_transformer.transform(vivienda[["vivienda_edad_mediana"]])
```

Tenga en cuenta que no existe una función inversa para el núcleo RBF, ya que siempre hay dos valores a una distancia determinada de un punto fijo (excepto a la distancia 0). También tenga en cuenta que `rbf_kernel()` no trata las funciones por separado. Si le pasa una matriz con dos características, medirá la distancia 2D (euclíadiana) para medir la similitud. Por ejemplo, aquí se explica cómo agregar una característica que medirá la similitud geográfica entre cada distrito y San Francisco:

```
sf_coords = 37.7749, -122.41
sf_transformer = FunctionTransformer(rbf_kernel, kw_args=dict(Y=
[sf_coords], gamma=0.1)) sf_simil
= sf_transformer.transform(vivienda[["latitud", "longitud"]])
```

Los transformadores personalizados también son útiles para combinar funciones. Por ejemplo, aquí hay un FunctionTransformer que calcula la relación entre las características de entrada 0 y 1:

```
>>> ratio_transformer = FunctionTransformer(lambda X: X[:, [0]]  
 / X[:, [1]]) >>>  
ratio_transformer.transform(np.array([[1., 2.], [3., 4.]])) array([[0.5], [0.75]])
```

FunctionTransformer es muy útil, pero ¿qué sucede si desea que su transformador sea entrenable, aprendiendo algunos parámetros en el método fit() y usándolos más adelante en el método transform()? Para ello, necesita escribir una clase personalizada. Scikit-Learn se basa en la escritura pato, por lo que esta clase no tiene que heredar de ninguna clase base en particular. Todo lo que necesita son tres métodos: fit() (que debe devolver self), transform() y fit\_transform().

Puede obtener fit\_transform() gratis simplemente agregando TransformerMixin como clase base: la implementación predeterminada simplemente llamará a fit() y luego transform(). Si agrega BaseEstimator como clase base (y evita usar \*args y \*\*kwargs en su constructor), también obtendrá dos métodos adicionales: get\_params() y set\_params().

Estos serán útiles para el ajuste automático de hiperparámetros.

Por ejemplo, aquí hay un transformador personalizado que actúa de manera muy similar al Escalador estándar:

```
desde sklearn.base importe BaseEstimator, TransformerMixin desde  
sklearn.utils.validation importe check_array, check_is_fitted  
  
clase StandardScalerClone(BaseEstimator, TransformerMixin): def __init__(self,  
with_mean=True): # ¡sin *args ni **kwargs! self.con_media = con_media  
  
    def fit(self, X, y=None): # y es necesario aunque  
    no lo uses  
        X = check_array(X) # comprueba que X es una matriz con  
        valores flotantes finitos
```

```

        self.mean_ = X.mean(axis=0) self.scale_ =
        X.std(axis=0) self.n_features_in_ =
        X.shape[1] # cada estimador
almacena esto en fit()
regresar a uno mismo # ¡siempre regresar a uno mismo!

def transform(self, X):
    check_is_fitted(self) # busca atributos aprendidos (con _ final)

    X = check_array(X) afirmar
    self.n_features_in_ == X.shape[1] if self.with_mean: X = X -
    self.mean_ return X /
    self.scale_

```

Aquí hay algunas cosas a tener en cuenta:

- El paquete `sklearn.utils.validation` contiene varias funciones que podemos usar para validar las entradas. Para simplificar, omitiremos dichas pruebas en el resto de este libro, pero el código de producción debería incluirlas.
- Las canalizaciones de Scikit-Learn requieren que el método `fit()` tenga dos argumentos `X` e `y`, razón por la cual necesitamos el argumento `y=None` aunque no usemos `y`.
- Todos los estimadores de Scikit-Learn configuran `n_features_in_` en el método `fit()` y garantizan que los datos pasados a `transform()` o `predict()` tengan esta cantidad de características.
- El método `fit()` debe devolver `self`.
- Esta implementación no está 100% completa: todos los estimadores deben configurar `feature_names_in_` en el método `fit()` cuando se les pasa un `DataFrame`. Además, todos los transformadores deben proporcionar un método `get_feature_names_out()`, así como un método `inverse_transform()` cuando su transformación se puede revertir. Consulte el último ejercicio al final de este capítulo para obtener más detalles.

Un transformador personalizado puede (y a menudo lo hace) utilizar otros estimadores en su implementación. Por ejemplo, el siguiente código muestra la personalización

transformador que utiliza un clusterer de KMeans en el método fit() para identificar los clusters principales en los datos de entrenamiento, y luego usa rbf\_kernel() en el método transform() para medir qué tan similar es cada muestra a cada centro de cluster:

```
desde sklearn.cluster importar KMeans

clase ClusterSimilarity(BaseEstimator, TransformerMixin): def __init__(self,
n_clusters=10, gamma=1.0, random_state=Ninguno):

    self.n_clusters = n_clusters self.gamma =
    gamma self.random_state
    = estado_aleatorio

    def fit(self, X, y=Ninguno, sample_weight=Ninguno): self.kmeans_ =
        KMeans(self.n_clusters, random_state=self.random_state)
    self.kmeans_.fit(X, sample_weight=sample_weight)
    return self # siempre devuelve self !

    def transform(self, X): devuelve
        rbf_kernel(X, self.kmeans_.cluster_centers_, gamma=self.gamma)

    def get_feature_names_out(self, nombres=Ninguno):
        devolver [f" Similitud del grupo {i} " para i en
rango(self.n_clusters)]
```

## CONSEJO

Puede comprobar si su estimador personalizado respeta la API de Scikit-Learn pasando una instancia a check\_estimator() desde el paquete sklearn.utils.estimator\_checks. Para obtener la API completa, consulte <https://scikit-learn.org/stable/developers>.

Como verá en [el Capítulo 9](#), k-means es un algoritmo de agrupamiento que ubica grupos en los datos. La cantidad que busca está controlada por el hiperparámetro n\_clusters. Después del entrenamiento, los centros de clúster están disponibles a través del atributo cluster\_centers\_. El método fit() de KMeans admite un argumento opcional sample\_weight, que permite

El usuario especifica los pesos relativos de las muestras. k-means es estocástico algoritmo, lo que significa que se basa en la aleatoriedad para localizar los grupos, por lo que si quieras resultados reproducibles, debes establecer el valor random\_state. parámetro. Como puede ver, a pesar de la complejidad de la tarea, el código es bastante sencillo. Ahora usemos este transformador personalizado:

```
cluster_simil = ClusterSimilarity(n_clusters=10, gamma=1.,
estado_aleatorio=42)
similitudes = cluster_simil.fit_transform(vivienda[["latitud",
"longitud"]],
peso_muestra=etiquetas_vivienda)
```

Este código crea un transformador ClusterSimilarity, configurando el número de clústeres a 10. Luego llama a fit\_transform() con el latitud y longitud de cada distrito en el conjunto de entrenamiento, ponderando cada distrito por el valor medio de la vivienda. El transformador utiliza k-means para localizar los grupos, luego mide la similitud gaussiana RBF entre cada uno distrito y los 10 centros agrupados. El resultado es una matriz con una fila por distrito y una columna por grupo. Veamos las primeras tres filas, redondeando a dos decimales:

```
>>> similitudes[:3].ronda(2)
matrix([[0. , 0.14, 0. 6 ], , 0. , 0. , 0.08, 0. , 0.99, 0. , ,
[0.63, 0. , 0.99, 0. , 0. , 0. , 0.04, 0. , 0.11,
0. ], [0. , 0.29, 0. , 0. , 0.01, 0.44, 0. , 0.7 , 0. , 0.3 ]])
```

**La Figura 2-19** muestra los 10 centros de conglomerados encontrados por k-medias. los distritos están coloreados según su similitud geográfica con su grupo más cercano centro. Como puede ver, la mayoría de los grupos están ubicados en zonas altamente pobladas y zonas caras.

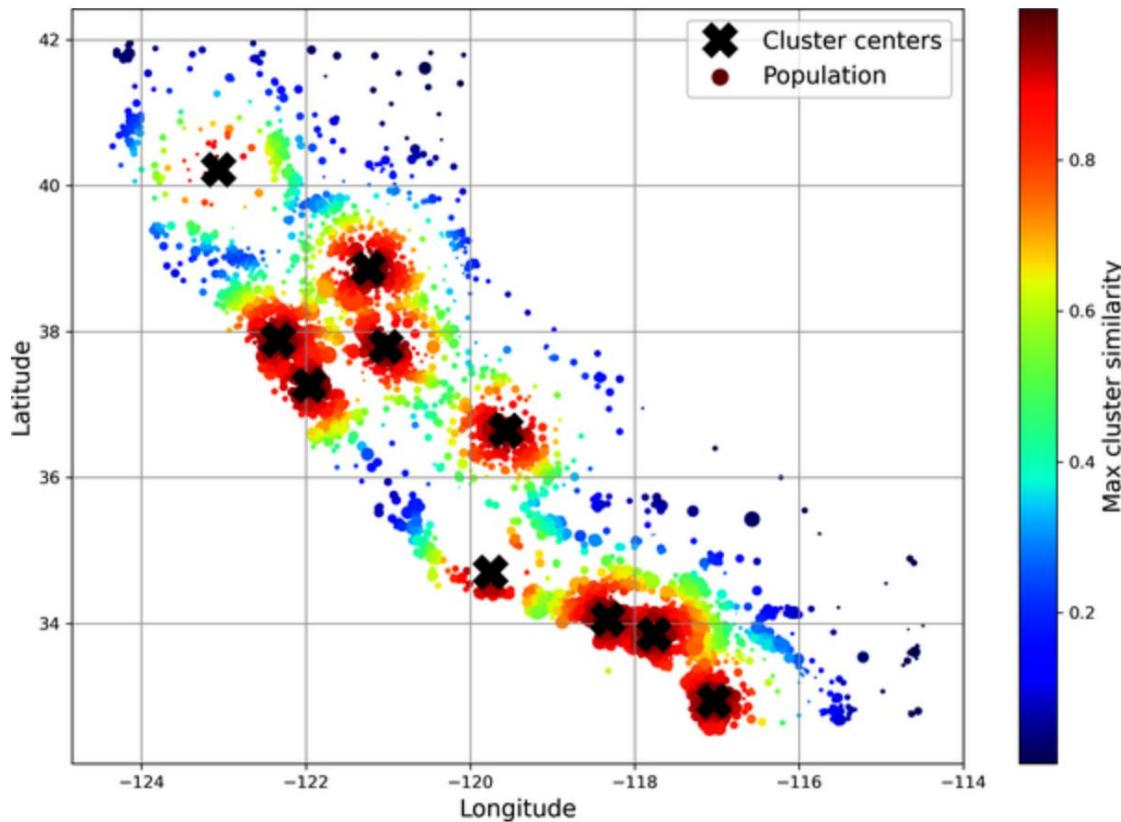


Figura 2-19. Similitud de RBF gaussiano con el centro del cúmulo más cercano

## Tuberías de transformación

Como puede ver, hay muchos pasos de transformación de datos que deben ejecutarse en el orden correcto. Afortunadamente, Scikit-Learn proporciona la clase Pipeline para ayudar con este tipo de secuencias de transformaciones. Aquí hay una pequeña canalización para atributos numéricos, que primero imputará y luego escalará las características de entrada:

```
desde sklearn.pipeline importar tubería

num_pipeline = Tubería([
    ("imputar", SimpleImputer(estrategia="mediana")),
    ("estandarizar", StandardScaler()),
])

```

El constructor Pipeline toma una lista de pares de nombre/estimador (2 tuplas) que definen una secuencia de pasos. Los nombres pueden ser los que desee, siempre que sean únicos y no contengan guiones bajos dobles (\_). Ellos

Será útil más adelante, cuando analicemos el ajuste de hiperparámetros. Todos los estimadores deben ser transformadores (es decir, deben tener un método `fit_transform()`), excepto el último, que puede ser cualquier cosa: un transformador, un predictor o cualquier otro tipo de estimador.

## CONSEJO

En un cuaderno Jupyter, si importa `sklearn` y ejecuta `sklearn.set_config(display="diagram")`, todos los estimadores de Scikit-Learn se representarán como diagramas interactivos. Esto es particularmente útil para visualizar tuberías. Para visualizar `num_pipeline`, ejecute una celda con `num_pipeline` como última línea. Al hacer clic en un estimador se mostrarán más detalles.

Si no desea nombrar los transformadores, puede utilizar la función `make_pipeline()` en su lugar; toma transformadores como argumentos posicionales y crea un Pipeline usando los nombres de las clases de los transformadores, en minúsculas y sin guiones bajos (por ejemplo, "`simpleimputer`"):

```
desde sklearn.pipeline importar make_pipeline

num_pipeline = make_pipeline(SimpleImputer(estrategia="mediana"),
Escalador estándar())
```

Si varios transformadores tienen el mismo nombre, se agrega un índice a sus nombres (por ejemplo, "`foo-1`", "`foo-2`", etc.).

Cuando llama al método `fit()` de la canalización, llama a `fit_transform()` secuencialmente en todos los transformadores, pasando la salida de cada llamada como parámetro a la siguiente llamada hasta que llega al estimador final, para lo cual simplemente llama a `fit()` método.

El pipeline expone los mismos métodos que el estimador final. En este ejemplo, el último estimador es un `StandardScaler`, que es un transformador, por lo que la tubería también actúa como un transformador. Si llama al método `transform()` de la canalización, aplicará secuencialmente todas las transformaciones a los datos. Si el último estimador fuera un predictor en lugar de un transformador, entonces la canalización tendría un método `predict()` en lugar de un

método transform(). Llamarlo aplicaría secuencialmente todas las transformaciones a los datos y pasaría el resultado al método predict() del predictor.

Llamemos al método fit\_transform() de la canalización y observemos las dos primeras filas de la salida, redondeadas a dos decimales:

```
>>> num_vivienda_preparada =
    num_pipeline.fit_transform(num_vivienda) >>>
    num_vivienda_preparada[:2].round(2) array([-1.42, 1.01, 1.86,
    0.31, 1.37, 0.14, 1.39, -0.94],
    [ 0.6      , -0.7      , 0.91, -0.31, -0.44, -0.69, -0.37,
    1.17]))
```

Como vio anteriormente, si desea recuperar un buen DataFrame, puede usar el método get\_feature\_names\_out() de la canalización:

```
df_housing_num_prepared = pd.DataFrame( vivienda_num_prepared,
    columns=num_pipeline.get_feature_names_out(), índice=housing_num.index)
```

Los oleoductos admiten la indexación; por ejemplo, pipeline[1] devuelve el segundo estimador del pipeline, y pipeline[:-1] devuelve un objeto Pipeline que contiene todo menos el último estimador. También puede acceder a los estimadores a través del atributo de pasos, que es una lista de pares de nombre/estimador, o mediante el atributo de diccionario llamado\_pasos, que asigna los nombres a los estimadores. Por ejemplo, num\_pipeline["simpleimputer"] devuelve el estimador llamado "simpleimputer".

Hasta ahora, hemos manejado las columnas categóricas y las columnas numéricas por separado. Sería más conveniente disponer de un único transformador capaz de manejar todas las columnas, aplicando a cada columna las transformaciones adecuadas. Para esto, puedes usar un ColumnTransformer. Por ejemplo, el siguiente ColumnTransformer aplicará num\_pipeline (el que acabamos de definir) a los atributos numéricos y cat\_pipeline al atributo categórico:

```

desde sklearn.compose importar ColumnTransformer

num_atribus = ["longitud", "latitud", "edad_media_vivienda", "habitaciones_total",
               "total_dormitorios", "población", "hogares", "ingresos_medianos"] cat_atribus
= ["proximidad_océano"]

cat_pipeline = hacer_pipeline(
    SimpleImputer(estrategia="más_frecuente"),
    OneHotEncoder(handle_unknown="ignorar"))

preprocesamiento = ColumnTransformer([
    ("num", num_pipeline, num_atribus), ("cat", cat_pipeline,
                                         cat_atribus),
])

```

Primero importamos la clase ColumnTransformer, luego definimos la lista de nombres de columnas numéricas y categóricas y construimos una canalización simple para atributos categóricos. Por último, construimos un ColumnTransformer. Su constructor requiere una lista de tripletes (3-tuplas), cada uno de los cuales contiene un nombre (que debe ser único y no contener guiones bajos dobles), un transformador y una lista de nombres (o índices) de columnas a las que se debe aplicar el transformador. .

## CONSEJO

En lugar de utilizar un transformador, puede especificar la cadena "soltar" si desea que las columnas se eliminen, o puede especificar "paso a través" si desea que las columnas no se modifiquen. De forma predeterminada, las columnas restantes (es decir, las que no estaban en la lista) se eliminarán, pero puede configurar el hiperparámetro restante en cualquier transformador (o en "transmisión") si desea que estas columnas se manejen de manera diferente.

Dado que enumerar todos los nombres de las columnas no es muy conveniente, Scikit-Learn proporciona una función make\_column\_selector() que devuelve una función selectora que puede usar para seleccionar automáticamente todas las características de un tipo determinado, como numéricas o categóricas. Puede pasar esta función de selección a ColumnTransformer en lugar de nombres de columnas o índices.

Además, si no le importa nombrar los transformadores, puede usar `make_column_transformer()`, que elige los nombres por usted, tal como lo hace `make_pipeline()`. Por ejemplo, el siguiente código crea el mismo ColumnTransformer que antes, excepto que los transformadores se denominan automáticamente "pipeline-1" y "pipeline-2" en lugar de "num" y "cat":

```
desde sklearn.compose importe make_column_selector,  
make_column_transformer  
  
preprocesamiento = make_column_transformer(  
    (num_pipeline,  
     make_column_selector(dtype_=include=np.number)),  
    (cat_pipeline, make_column_selector(dtype_=objeto)),  
)
```

Ahora estamos listos para aplicar este ColumnTransformer a los datos de vivienda:

```
vivienda_preparada = preprocesamiento.fit_transform(vivienda)
```

¡Excelente! Tenemos una canalización de preprocesamiento que toma todo el conjunto de datos de entrenamiento y aplica cada transformador a las columnas apropiadas, luego concatena las columnas transformadas horizontalmente (los transformadores nunca deben cambiar el número de filas). Una vez más, esto devuelve una matriz NumPy, pero puedes obtener los nombres de las columnas usando `preprocessing.get_feature_names_out()` y envolver los datos en un bonito DataFrame como lo hicimos antes.

### NOTA

OneHotEncoder devuelve una matriz dispersa y num\_pipeline devuelve una matriz densa. Cuando existe una combinación de matrices densas y dispersas, ColumnTransformer estima la densidad de la matriz final (es decir, la proporción de celdas distintas de cero) y devuelve una matriz dispersa si la densidad es inferior a un umbral determinado (de forma predeterminada, `umbral_disperso = 0,3`). En este ejemplo, devuelve una matriz densa.

¡Tu proyecto va muy bien y estás casi listo para entrenar algunos modelos! Ahora deseas crear una canalización única que realizará todas las transformaciones con las que ha experimentado hasta ahora. Recapitulemos qué hará el oleoducto y por qué:

- Los valores faltantes en las características numéricas se imputarán reemplazándolos con la mediana, ya que la mayoría de los algoritmos de ML no esperan valores faltantes. En las características categóricas, los valores faltantes serán reemplazados por la categoría más frecuente.
- La característica categórica se codificará en caliente, ya que la mayoría de los algoritmos de ML solo aceptan entradas numéricas.
- Se calcularán y agregarán algunas características de proporción: proporción\_dormitorios, habitaciones\_por\_casa y personas\_por\_casa. Con suerte, estos se correlacionarán mejor con el valor medio de la vivienda y, por lo tanto, ayudarán a los modelos ML.
- También se agregarán algunas características de similitud de grupos. Probablemente serán más útiles para el modelo que la latitud y la longitud.
- Las características con una cola larga serán reemplazadas por su logaritmo, ya que la mayoría de los modelos prefieren características con distribuciones aproximadamente uniformes o gaussianas.
- Todas las características numéricas estarán estandarizadas, ya que la mayoría de los algoritmos de ML prefieren que todas las características tengan aproximadamente la misma escala.

El código que construye la canalización para hacer todo esto ya debería resultarle familiar:

```
def column_ratio(X): devolver X[:,  
[0]] / X[:, [1]]  
  
def ratio_name(function_transformer, feature_names_in): return ["ratio"] # nombres de  
características fuera  
  
def ratio_pipeline(): devuelve  
    make_pipeline(  
        SimpleImputer(estrategia="mediana"),  
        FunctionTransformer(column_ratio,  
        feature_names_out=ratio_name),  
        Escalador estándar())
```

```

log_pipeline =
    make_pipeline( SimpleImputer(estrategia="mediana"),
    FunctionTransformer(np.log, feature_names_out="uno a uno"), StandardScaler())

cluster_simil = ClusterSimilarity(n_clusters=10, gamma=1., random_state=42) default_num_pipeline
=

make_pipeline(SimpleImputer(estrategia="mediana"),
    Escalador estándar())
preprocesamiento = ColumnTransformer([ ("dormitorios",
    ratio_pipeline(), ["total_dormitorios", "total_habitaciones"]),
    ("habitaciones_por_casa",
    ratio_pipeline(),
    ["total_habitaciones",
    "hogares"]),
    ("personas_por_casa", ratio_pipeline(), ["población",
    "hogares"]),
    ("log", log_pipeline, ["total_dormitorios", "total_habitaciones", "población",
    "hogares", "ingreso_mediano"]),
    ("geo", cluster_simil, ["latitud", "longitud"]),
    ("cat", cat_pipeline,
    make_column_selector(dtype_include=objeto)),
    ],
    resto=default_num_pipeline) # queda una columna: housing_median_age

```

Si ejecuta este ColumnTransformer, realiza todas las transformaciones y genera una matriz NumPy con 24 funciones:

```

>>> vivienda_prepared = preprocesamiento.fit_transform(vivienda) >>> vivienda_prepared.shape
(16512, 24) >>>

preprocesamiento.get_feature_names_out() array(['dormitorios__ratio',
'habitaciones_por_casa__ratio',
'people_per_house__ratio', 'log__total_dormitorios', 'log__total_habitaciones',
'log__población', 'log__hogares',
'log__median_ Income', 'geo__Cluster 0 similitud',
[...],
'geo__Similitud del grupo 9', 'cat__ocean_proximity_<1H
OCÉANO',
'cat__ocean_proximity_INLAND',
'cat__ocean_proximity_ISLAND',

```

```
'cat_ocean_proximity_NEAR BAY',
'cat_ocean_proximity_NEAR OCEAN',
'resto_vivienda_edad_media'], tipod=objeto)
```

## Seleccionar y entrenar un modelo

¡Por fin! Planteó el problema, obtuvo los datos y los exploró, tomó muestras de un conjunto de entrenamiento y de prueba, y escribió un proceso de preprocesamiento para limpiar y preparar automáticamente sus datos para los algoritmos de aprendizaje automático. Ahora está listo para seleccionar y entrenar un modelo de aprendizaje automático.

## Entrene y evalúe en el conjunto de entrenamiento

¡La buena noticia es que gracias a todos estos pasos previos ahora todo va a ser fácil! Para comenzar, decides entrenar un modelo de regresión lineal muy básico:

```
de sklearn.linear_model importar LinearRegression
```

```
lin_reg = make_pipeline(preprocesamiento, LinearRegression())
lin_reg.fit(vivienda, etiquetas_vivienda)
```

¡Hecho! Ahora tiene un modelo de regresión lineal funcional. Lo prueba en el conjunto de entrenamiento, observa las primeras cinco predicciones y las compara con las etiquetas:

```
>>> predicciones_vivienda = lin_reg.predict(vivienda) >>>
predicciones_vivienda[:5].round(-2) # -2 = redondeado a la centena más cercana
matriz([243700.,
372400., 128800., 94400., 328300 .]) >>> etiquetas_vivienda.iloc[:5] .matriz
de valores([458300., 483800., 101700., 96100.,
361800.])
```

Bueno, funciona, pero no siempre: la primera predicción está muy equivocada (¡en más de 200.000 dólares!), mientras que las otras predicciones son mejores: dos están equivocadas en aproximadamente un 25% y dos están equivocadas en menos del 10%. Recuerde que eligió utilizar el RMSE como medida de rendimiento, por lo que desea medir el RMSE de este modelo de regresión en todo el conjunto de entrenamiento utilizando Scikit-Learn.

función `mean_squared_error()`, con el argumento `al_cuadrado` establecido en `FALSO`:

```
>>> de sklearn.metrics importar error_cuadrado_medio >>> lin_rmse =  
error_cuadrado_medio(etiquetas_vivienda, predicciones_vivienda,  
...                                         al cuadrado = Falso)  
...  
>>> lin_rmse  
68687.89176589991
```

Este es mejor que nada, pero claramente no es una gran puntuación: los valores\_de\_vivienda\_medianos de la mayoría de los distritos oscilan entre \$120.000 y \$265.000, por lo que un error de predicción típico de \$68.628 realmente no es muy satisfactorio. Este es un ejemplo de un modelo que no se adapta adecuadamente a los datos de entrenamiento. Cuando esto sucede, puede significar que las funciones no proporcionan suficiente información para hacer buenas predicciones o que el modelo no es lo suficientemente potente. Como vimos en el capítulo anterior, las principales formas de solucionar el desajuste son seleccionar un modelo más potente, alimentar el algoritmo de entrenamiento con mejores características o reducir las restricciones del modelo.

Este modelo no está regularizado, lo que descarta la última opción. Podrías intentar agregar más funciones, pero primero debes probar un modelo más complejo para ver cómo funciona.

Decide probar un DecisionTreeRegressor, ya que es un modelo bastante potente capaz de encontrar relaciones no lineales complejas en los datos (los árboles de decisión se presentan con más detalle en el [Capítulo 6](#)):

```
desde sklearn.tree importar DecisionTreeRegressor

tree_reg = make_pipeline(preprocesamiento,
DecisionTreeRegressor(random_state=42))
tree_reg.fit(vivienda, etiquetas_vivienda)
```

Ahora que el modelo está entrenado, lo evalúa en el conjunto de entrenamiento:

```
>>> árbol_rmse 0.0
```

¿¡Esperar lo!? ¿Ningún error? ¿Podría este modelo ser realmente absolutamente perfecto? Por supuesto, es mucho más probable que el modelo se haya sobreajustado gravemente a los datos. ¿Como puedes estar seguro? Como vio anteriormente, no desea tocar el conjunto de prueba hasta que esté listo para lanzar un modelo en el que esté seguro, por lo que debe usar parte del conjunto de entrenamiento para el entrenamiento y parte para la validación del modelo.

Mejor evaluación mediante validación cruzada Una forma de evaluar el modelo de árbol de decisión sería utilizar la función `train_test_split()` para dividir el conjunto de entrenamiento en un conjunto de entrenamiento más pequeño y un conjunto de validación, luego entrenar sus modelos con el conjunto de entrenamiento más pequeño y evaluar compararlos con el conjunto de validación. Es un poco de esfuerzo, pero nada demasiado difícil y funcionaría bastante bien.

Una gran alternativa es utilizar la función de validación cruzada `k_fold` de Scikit-Learn . El siguiente código divide aleatoriamente el conjunto de entrenamiento en 10 subconjuntos no superpuestos llamados pliegues, luego entrena y evalúa el modelo de árbol de decisión 10 veces, eligiendo un pliegue diferente para la evaluación cada vez y usando los otros 9 pliegues para el entrenamiento. El resultado es una matriz que contiene las 10 puntuaciones de evaluación:

```
desde sklearn.model_selection importar cross_val_score
tree_rmses = -cross_val_score(tree_reg, vivienda, viviendas_labels,
puntuación="neg_root_mean_squared_error", cv=10)
```

#### ADVERTENCIA

Las características de validación cruzada de Scikit-Learn esperan una función de utilidad (cuanto mayor es mejor) en lugar de una función de costo (cuanto menor es mejor), por lo que la función de puntuación es en realidad lo opuesto al RMSE. Es un valor negativo, por lo que debe cambiar el signo de la salida para obtener las puntuaciones RMSE.

Veamos los resultados:

```
>>> pd.Series(tree_rmses).describe()
contar           10.000000
significar      66868.027288
                2060.966425
mín.            63649.536493
25%             65338.078316
50%             66801.953094
75%             68229.934454
máximo          70094.778246
tipo de letra: float64
```

Ahora el árbol de decisiones no parece tan bueno como antes. De hecho, ¡Parece funcionar casi tan mal como el modelo de regresión lineal! Aviso esa validación cruzada permite obtener no sólo una estimación del rendimiento de su modelo, sino también una medida de cuán preciso es este estimación es (es decir, su desviación estándar). El árbol de decisión tiene un RMSE de alrededor de 66.868, con una desviación estándar de alrededor de 2.061. No podrías tener esta información si solo usó un conjunto de validación. Pero la validación cruzada tiene el costo de entrenar el modelo varias veces, por lo que no es siempre factible.

Si calcula la misma métrica para el modelo de regresión lineal, obtendrá Encuentre que el RMSE medio es 69,858 y la desviación estándar es 4,182. Por lo tanto, el modelo de árbol de decisión parece funcionar ligeramente mejor que el modelo lineal, pero la diferencia es mínima debido a un sobreajuste severo. Nosotros saber que hay un problema de sobreajuste porque el error de entrenamiento es bajo (en realidad cero) mientras que el error de validación es alto.

Probemos ahora un último modelo: RandomForestRegressor. Como tu quieras ver en [el Capítulo 7](#), los bosques aleatorios funcionan entrenando muchos árboles de decisión en subconjuntos aleatorios de las características y luego promedian sus predicciones. Semejante Los modelos compuestos de muchos otros modelos se llaman conjuntos: son capaz de impulsar el rendimiento del modelo subyacente (en este caso, árboles de decisión). El código es muy parecido al anterior:

```
desde sklearn.ensemble importar RandomForestRegressor
```

```
forest_reg = make_pipeline(preprocesamiento,
```

```
RandomForestRegressor(random_state=42)) forest_rmses =
-cross_val_score(forest_reg, vivienda, viviendas_labels,
                  puntuación="neg_root_mean_squared_error", cv=10)
```

Veamos las puntuaciones:

```
>>> pd.Series(forest_rmses).describe() 10.000000
contar           47019.561281
significar      1033.957120
                45458.112527
                46464.031184
                46967.596354
estándar mínimo 25% 50%
75%             47325.694987
máximo          49243.765795
tipo de letra: float64
```

Vaya, esto es mucho mejor: ¡los bosques aleatorios realmente parecen muy prometedores para esta tarea! Sin embargo, si entrena un RandomForest y mide el RMSE en el conjunto de entrenamiento, encontrará aproximadamente 17,474: eso es mucho más bajo, lo que significa que todavía hay bastante sobreajuste. Las posibles soluciones son simplificar el modelo, restringirlo (es decir, regularizarlo) u obtener muchos más datos de entrenamiento. Sin embargo, antes de profundizar mucho más en los bosques aleatorios, debería probar muchos otros modelos de varias categorías de algoritmos de aprendizaje automático (por ejemplo, varias máquinas de vectores de soporte con diferentes núcleos y posiblemente una red neuronal), sin perder demasiado tiempo modificando el hiperparámetros. El objetivo es preseleccionar algunos (de dos a cinco) modelos prometedores.

## Ajusta tu modelo

Supongamos que ahora tiene una lista corta de modelos prometedores. Ahora necesitas ajustarlos. Veamos algunas formas en las que puedes hacerlo.

Búsqueda de cuadrícula

Una opción sería jugar con los hiperparámetros manualmente, hasta encontrar una excelente combinación de valores de hiperparámetros. Esto sería

Es un trabajo muy tedioso y es posible que no tengas tiempo para explorar muchas combinaciones.

En su lugar, puede utilizar la clase GridSearchCV de Scikit-Learn para buscarlo. Todo lo que necesita hacer es decirle con qué hiperparámetros desea que experimente y qué valores probar, y utilizará la validación cruzada para evaluar todas las combinaciones posibles de valores de hiperparámetros. Por ejemplo, el siguiente código busca la mejor combinación de valores de hiperparámetros para RandomForestRegressor:

```
desde sklearn.model_selection importar GridSearchCV

full_pipeline =
    Pipeline([ ("preprocesamiento",
    preprocesamiento), ("random_forest", RandomForestRegressor(random_state=42)),
])
param_grid =
    [ {'preprocessing_geo_n_clusters': [5, 8, 10],
    'random_forest_max_features': [4, 6, 8]},
    {'preprocessing_geo_n_clusters': [10, 15],
    'random_forest_max_features': [6, 8 , 10]},]

grid_search = GridSearchCV(full_pipeline, param_grid, cv=3,
    puntuación='neg_root_mean_squared_error')
grid_search.fit(vivienda, etiquetas_vivienda)
```

Tenga en cuenta que puede hacer referencia a cualquier hiperparámetro de cualquier estimador en una tubería, incluso si este estimador está anidado en lo profundo de varias tuberías y transformadores de columna. Por ejemplo, cuando Scikit-Learn ve "preprocesamiento\_geo\_n\_clusters", divide esta cadena en guiones bajos dobles, luego busca un estimador llamado "preprocesamiento" en la canalización y encuentra el ColumnTransformer de preprocesamiento. A continuación, busca un transformador llamado "geo" dentro de este ColumnTransformer y encuentra el transformador ClusterSimilarity que utilizamos en los atributos de latitud y longitud. Luego encuentra el hiperparámetro n\_clusters de este transformador. De manera similar, random\_forest\_max\_features se refiere al hiperparámetro max\_features del estimador llamado "random\_forest", que es de

Por supuesto, el modelo RandomForest (el hiperparámetro `max_features` se explicará en [el Capítulo 7](#)).

## CONSEJO

Incluir los pasos de preprocessamiento en una canalización de Scikit-Learn le permite ajustar los hiperparámetros de preprocessamiento junto con los hiperparámetros del modelo. Esto es bueno ya que interactúan a menudo. Por ejemplo, tal vez aumentar `n_clusters` requiera aumentar también `max_features`. Si instalar los transformadores de la tubería es computacionalmente costoso, puede configurar el hiperparámetro de memoria de la tubería en la ruta de un directorio de almacenamiento en caché: cuando instale la tubería por primera vez, Scikit-Learn guardará los transformadores instalados en este directorio. Si luego vuelve a ajustar la canalización con los mismos hiperparámetros, Scikit-Learn simplemente cargará los transformadores almacenados en caché.

Hay dos diccionarios en este `param_grid`, por lo que GridSearchCV primero evaluará todas las  $3 \times 3 = 9$  combinaciones de valores de hiperparámetros `n_clusters` y `max_features` especificados en el primer diccionario, luego probará todas las  $2 \times 3 = 6$  combinaciones de valores de hiperparámetros en el segundo diccionario. . Entonces, en total, la búsqueda de cuadrícula explorará  $9 + 6 = 15$  combinaciones de valores de hiperparámetros y entrenará la canalización 3 veces por combinación, ya que estamos usando una validación cruzada triple. ¡Esto significa que habrá un total de  $15 \times 3 = 45$  rondas de entrenamiento! Puede que tarde un poco, pero cuando esté hecho podrás obtener la mejor combinación de parámetros como este:

```
>>> grid_search.best_params_
{'preprocesamiento_geo_n_clusters': 15,
 'random_forest_max_features': 6}
```

En este ejemplo, el mejor modelo se obtiene configurando `n_clusters` en 15 y `max_features` en 8.

## CONSEJO

Dado que 15 es el valor máximo evaluado para n\_clusters, usted probablemente debería intentar buscar nuevamente con valores más altos; la puntuación puede seguir mejorando.

Puedes acceder al mejor estimador usando

`grid_search.best_estimator_`. Si GridSearchCV se inicializa con `refit=True` (que es el valor predeterminado), una vez que encuentre el mejor estimador mediante validación cruzada, lo vuelve a entrenar en todo el conjunto de entrenamiento. Esto es suele ser una buena idea, ya que alimentarlo con más datos probablemente mejorará su actuación.

Los puntajes de evaluación están disponibles usando `grid_search.cv_results_`.

Este es un diccionario, pero si lo envuelves en un DataFrame obtendrás una buena lista de todas las puntuaciones de las pruebas para cada combinación de hiperparámetros y para cada división de validación cruzada, así como la puntuación media de la prueba en todas las divisiones:

```
>>> cv_res = pd.DataFrame(grid_search.cv_results_)
>>> cv_res.sort_values(by="mean_test_score", ascending=False,
en el lugar = Verdadero)
>>> [...] # cambiar los nombres de las columnas para que quepan en esta página y mostrar
rmse = -puntuación
>>> cv_res.head() # nota: la primera columna es el ID de la fila
   n_clusters max_features split0 split1 split2
media_prueba_rmse
12            15          6 43460 43919 44748
44042
13            15          8 44132 44075 45010
44406
14            15         10 44374 44286 45316
44659
7 44999           10          6 44683 44655 45657
9            10          6 44683 44655 45657
44999
```

La puntuación media de la prueba RMSE para el mejor modelo es 44.042, que es mejor que la puntuación que obtuvo anteriormente utilizando los valores de hiperparámetro predeterminados

(que fue 47.019). ¡Felicitaciones, ha perfeccionado con éxito su mejor modelo!

## Búsqueda aleatoria

El enfoque de búsqueda en cuadrícula está bien cuando se exploran relativamente pocas combinaciones, como en el ejemplo anterior, pero RandomizedSearchCV suele ser preferible, especialmente cuando el espacio de búsqueda de hiperparámetros es grande. Esta clase se puede utilizar de forma muy similar a la clase GridSearchCV, pero en lugar de probar todas las combinaciones posibles, evalúa un número fijo de combinaciones y selecciona un valor aleatorio para cada hiperparámetro en cada iteración. Esto puede parecer sorprendente, pero este enfoque tiene varios beneficios:

- Si algunos de sus hiperparámetros son continuos (o discretos pero con muchos valores posibles) y deja que la búsqueda aleatoria se ejecute durante, digamos, 1000 iteraciones, entonces explorará 1000 valores diferentes para cada uno de estos hiperparámetros, mientras que la búsqueda en cuadrícula solo explorará los algunos valores que enumeró para cada uno.
- Supongamos que un hiperparámetro en realidad no hace mucha diferencia, pero aún no lo sabe. Si tiene 10 valores posibles y lo agregas a tu búsqueda en la grilla, entonces el entrenamiento tomará 10 veces más. Pero si lo agregas a una búsqueda aleatoria, no hará ninguna diferencia.
- Si hay 6 hiperparámetros para explorar, cada uno con 10 valores posibles, entonces la búsqueda en cuadrícula no ofrece otra opción que entrenar el modelo un millón de veces, mientras que la búsqueda aleatoria siempre puede ejecutarse para cualquier número de iteraciones que elija.

Para cada hiperparámetro, debe proporcionar una lista de valores posibles o una distribución de probabilidad:

```
desde sklearn.model_selection importar RandomizedSearchCV desde scipy.stats
importar randint

param_dists = {'preprocessing__geo__n_clusters': randint(bajo=3, alto=50),
'random_forest__max_features':
                    randint(bajo=2,
alto = 20)}
```

```
rnd_search = RandomizedSearchCV( full_pipeline,
    param_distributions=param_distrib, n_iter=10, cv=3,
    scoring='neg_root_mean_squared_error', random_state=42)

rnd_search.fit(vivienda, etiquetas_vivienda)
```

Scikit-Learn también tiene clases de búsqueda de hiperparámetros

HalvingRandomSearchCV y HalvingGridSearchCV. Su objetivo es utilizar los recursos computacionales de manera más eficiente, ya sea para entrenar más rápido o para explorar un espacio de hiperparámetros más grande. Así es como funcionan: en la primera ronda, se generan muchas combinaciones de hiperparámetros (llamadas "candidatos") utilizando el enfoque de cuadrícula o el enfoque aleatorio. Luego, estos candidatos se utilizan para entrenar modelos que se evalúan mediante validación cruzada, como es habitual. Sin embargo, la formación utiliza recursos limitados, lo que acelera considerablemente esta primera ronda. De forma predeterminada, "recursos limitados" significa que los modelos se entran en una pequeña parte del conjunto de entrenamiento. Sin embargo, son posibles otras limitaciones, como reducir el número de iteraciones de entrenamiento si el modelo tiene un hiperparámetro para configurarlo. Una vez evaluados todos los candidatos, sólo los mejores pasan a la segunda vuelta, donde se les permiten más recursos para competir. Después de varias rondas, los candidatos finales son evaluados utilizando todos los recursos. Esto puede ahorrarle algo de tiempo ajustando los hiperparámetros.

## Métodos de conjunto

Otra forma de ajustar su sistema es intentar combinar los modelos que funcionan mejor. El grupo (o "conjunto") a menudo funcionará mejor que el mejor modelo individual (al igual que los bosques aleatorios funcionan mejor que los árboles de decisión individuales en los que se basan), especialmente si los modelos individuales cometan tipos de errores muy diferentes. Por ejemplo, podría entrenar y ajustar un modelo de k vecinos más cercanos y luego crear un modelo de conjunto que simplemente prediga la media de la predicción del bosque aleatorio y la predicción de ese modelo. Cubriremos este tema con más detalle en el [Capítulo 7](#).

## Analizando los mejores modelos y sus errores

A menudo obtendrá buenos conocimientos sobre el problema inspeccionando los mejores modelos. Por ejemplo, RandomForestRegressor puede indicar la importancia relativa de cada atributo para realizar predicciones precisas:

```
>>> final_model = rnd_search.best_estimator_ # incluye preprocesamiento
>>>
feature_importances =
final_model["random_forest"].feature_importances_
>>>
feature_importances.round(2) array([0.07,
0.05, 0.05, 0.01, 0.01, 0.01, 0.01, 0.19, [...], 0.01])
```

Clasifiquemos estas puntuaciones de importancia en orden descendente y mostrémoslas junto a los nombres de sus atributos correspondientes:

```
>>> ordenado(zip(características_importancias,
...
final_model["preprocesamiento"].get_feature_names_out(),
...           inversa=True)
...
[(0.18694559869103852, 'log__ingreso_mediano'),
 (0.0748194905715524, 'cat__ocean_proximity_INLAND'),
 (0.06926417748515576, 'dormitorios__proporción'),
 (0.05446998753775219, 'habitaciones_por_house_ratio'),
 (0.05262301809680712, 'people_per_house_ratio'),
 (0.03819415873915732, 'geo_Cluster 0 similitud') , [...]
(0.00015061247730531558, 'cat__ocean_proximity_NEAR BAY'),
(7.301686597099842e-05, 'cat__ocean_proximity_ISLAND')]
```

Con esta información, es posible que desee intentar eliminar algunas de las funciones menos útiles (por ejemplo, aparentemente sólo una categoría ocean\_proximity es realmente útil, por lo que podría intentar eliminar las demás).

#### CONSEJO

El transformador sklearn.feature\_selection.SelectFromModel puede eliminar automáticamente las funciones menos útiles: cuando lo ajusta, entrena un modelo (normalmente un bosque aleatorio), mira su atributo feature\_importances\_ y selecciona las funciones más útiles. Luego, cuando llamas a transform(), elimina las otras funciones.

También debe observar los errores específicos que comete su sistema, luego tratar de comprender por qué los comete y qué podría solucionar el problema: agregar funciones adicionales o deshacerse de las que no son informativas, limpiar los valores atípicos, etc.

Ahora también es un buen momento para garantizar que su modelo no sólo funcione bien en promedio, sino también en todas las categorías de distritos, ya sean rurales o urbanos, ricos o pobres, del norte o del sur, minoritarios o no, etc. de su conjunto de validación para cada categoría requiere un poco de trabajo, pero es importante: si su modelo funciona mal en toda una categoría de distritos, entonces probablemente no debería implementarse hasta que se resuelva el problema, o al menos no debería usarse hacer predicciones para esa categoría, ya que puede hacer más daño que bien.

Evalúe su sistema en el conjunto de prueba Después de modificar sus modelos por un tiempo, eventualmente tendrá un sistema que funciona suficientemente bien. Está listo para evaluar el modelo final en el conjunto de prueba. No hay nada especial en este proceso; simplemente obtenga los predictores y las etiquetas de su conjunto de pruebas y ejecute su final\_model para transformar los datos y hacer predicciones, luego evalúe estas predicciones:

```
X_test = strat_test_set.drop("valor_mediano_casa", eje=1) y_test =  
strat_test_set["valor_mediano_casa"].copiar()  
  
prediccciones_finales = modelo_final.predict(X_test)  
  
final_rmse = mean_squared_error(y_test, final_predictions, squared=False)  
print(final_rmse) #  
imprime 41424.40026462184
```

En algunos casos, una estimación puntual del error de generalización no será suficiente para convencerle de lanzarlo: ¿qué pasa si es sólo un 0,1% mejor que el modelo actualmente en producción? Es posible que desee tener una idea de cuán precisa es esta estimación. Para ello, puede calcular un intervalo de confianza del 95% para el error de generalización utilizando `scipy.stats.t.interval()`.

Obtienes un intervalo bastante grande de 39,275 a 43,467, y tu estimación puntual anterior de 41,424 está aproximadamente en el medio:

```
>>> de las estadísticas de importación
de scipy >>> confianza = 0,95
>>> errores_cuadrados = (predicciones_finales - prueba_y) ** 2 >>>
np.sqrt(stats.t.interval(confianza, len(errores_cuadrados) - 1,
...
...                                loc=errores_cuadrados.mean(),
...                                escala=stats.sem(errores_cuadrados)))
...
matriz([39275.40861216, 43467.27680583])
```

Si realizó muchos ajustes de hiperparámetros, el rendimiento generalmente será ligeramente peor que lo que midió mediante validación cruzada. Esto se debe a que su sistema termina ajustado para funcionar bien con los datos de validación y probablemente no funcionará tan bien con conjuntos de datos desconocidos. Ese no es el caso en este ejemplo, ya que el RMSE de prueba es menor que el RMSE de validación, pero cuando esto sucede, debe resistir la tentación de modificar los hiperparámetros para que los números se vean bien en el conjunto de prueba; Es poco probable que las mejoras se generalicen a nuevos datos.

Ahora viene la fase previa al lanzamiento del proyecto: debe presentar su solución (destacando lo que ha aprendido, lo que funcionó y lo que no, las suposiciones que se hicieron y cuáles son las limitaciones de su sistema), documentar todo y crear presentaciones agradables con visualizaciones claras. y declaraciones fáciles de recordar (por ejemplo, “el ingreso medio es el predictor número uno de los precios de la vivienda”). En este ejemplo de vivienda de California, el rendimiento final del sistema no es mucho mejor que las estimaciones de precios de los expertos, que a menudo estaban desviadas en un 30%, pero aun así puede ser una buena idea lanzarlo, especialmente si esto libera algo de tiempo. para que los expertos puedan trabajar en tareas más interesantes y productivas.

## Inicie, supervise y mantenga su sistema

¡Perfecto, tienes aprobación para lanzar! Ahora necesita preparar su solución para producción (por ejemplo, pulir el código, escribir documentación y pruebas, etc.). Luego puede implementar su modelo en su entorno de producción. La forma más básica de hacer esto es simplemente guardar el mejor modelo que entrenó, transferir el archivo a su entorno de producción y cargarlo. Para guardar el modelo, puede utilizar la biblioteca joblib de esta manera:

```
importar biblioteca de trabajos

joblib.dump(final_model, "my_california_housing_model.pkl")
```

## CONSEJO

A menudo es una buena idea guardar cada modelo con el que experimentas para poder volver fácilmente a cualquier modelo que deseas. También puede guardar las puntuaciones de validación cruzada y quizás las predicciones reales en el conjunto de validación. Esto le permitirá comparar fácilmente puntuaciones entre tipos de modelos y comparar los tipos de errores que cometen.

Una vez que su modelo se transfiera a producción, podrá cargarlo y usarlo. Para esto, primero debe importar las clases y funciones personalizadas en las que se basa el modelo (lo que significa transferir el código a producción), luego cargar el modelo usando joblib y usarlo para hacer predicciones:

```
importar joblib [...]
# importar KMeans, BaseEstimator, TransformerMixin, rbf_kernel, etc.

def column_ratio(X): [...] def
ratio_name(function_transformer, feature_names_in): [...] clase
ClusterSimilarity(BaseEstimator, TransformerMixin): [...]

final_model_reloaded =
joblib.load("mi_modelo_de_vivienda_california.pkl")

new_data = [...] # algunos distritos nuevos para hacer predicciones para predicciones =
final_model_reloaded.predict(new_data)
```

Por ejemplo, tal vez el modelo se utilice dentro de un sitio web: el usuario escribirá algunos datos sobre un nuevo distrito y hará clic en el botón Estimar precio. Esto enviará una consulta que contiene los datos al servidor web, que los reenviará a su aplicación web y, finalmente, su código simplemente llamará al método predict() del modelo (desea cargar el modelo al iniciar el servidor, en lugar de cada vez). se utiliza el modelo). Alternativamente, puede envolver el modelo dentro de un servicio web dedicado que su aplicación web puede consultar a través de una API REST (consulte [Figura 2-20](#)). Esto hace que sea más fácil actualizar su modelo a nuevas versiones sin interrumpir el proceso principal.

solicitud. También simplifica el escalado, ya que puede iniciar tantos servicios web como necesite y equilibrar la carga de las solicitudes provenientes de su aplicación web entre estos servicios web. Además, permite que su aplicación web utilice cualquier lenguaje de programación, no solo Python.



Figura 2-20. Un modelo implementado como un servicio web y utilizado por una aplicación web.

Otra estrategia popular es implementar su modelo en la nube, por ejemplo en Vertex AI de Google (anteriormente conocido como Google Cloud AI Platform y Google Cloud ML Engine): simplemente guarde su modelo usando joblib y cárguelo en Google Cloud Storage (GCS). luego dirígete a Vertex AI y crea una nueva versión del modelo, apuntándola al archivo GCS. ¡Eso es todo! Esto le brinda un servicio web simple que se encarga del equilibrio y el escalado de carga por usted. Toma solicitudes JSON que contienen los datos de entrada (por ejemplo, de un distrito) y devuelve respuestas JSON que contienen las predicciones. Luego puede utilizar este servicio web en su sitio web (o en cualquier entorno de producción que esté utilizando). Como verá en [el Capítulo 19](#), implementar modelos TensorFlow en Vertex AI no es muy diferente de implementar modelos Scikit-Learn.

Pero el despliegue no es el final de la historia. También necesita escribir un código de monitoreo para verificar el rendimiento en vivo de su sistema a intervalos regulares y activar alertas cuando falla. Puede caer muy rápidamente, por ejemplo, si un componente de su infraestructura se estropea, pero tenga en cuenta que también podría decaer muy lentamente, lo que fácilmente puede pasar desapercibido durante mucho tiempo. Esto es bastante común debido a la descomposición del modelo: si el modelo se entrenó con los datos del año pasado, es posible que no se adapte a los datos actuales.

Por lo tanto, necesita monitorear el desempeño en vivo de su modelo. ¿Pero cómo haces eso? Bueno, eso depende. En algunos casos, el rendimiento del modelo se puede inferir a partir de métricas posteriores. Por ejemplo, si su modelo es parte de un sistema de recomendación y sugiere productos que pueden interesar a los usuarios, entonces es fácil monitorear la cantidad de productos recomendados vendidos cada día. Si este número cae (en comparación con productos no recomendados), entonces el principal sospechoso es el modelo. Esto puede deberse a que la canalización de datos está rota o tal vez sea necesario volver a entrenar el modelo con datos nuevos (como veremos en breve).

Sin embargo, es posible que también necesite un análisis humano para evaluar el rendimiento del modelo. Por ejemplo, supongamos que entrenó un modelo de clasificación de imágenes (los veremos en el [Capítulo 3](#)) para detectar varios defectos del producto en una línea de producción. ¿Cómo puede recibir una alerta si el rendimiento del modelo disminuye, antes de que se envíen miles de productos defectuosos a sus clientes?

Una solución es enviar a los evaluadores humanos una muestra de todas las imágenes que clasificó el modelo (especialmente las imágenes de las que el modelo no estaba tan seguro). Dependiendo de la tarea, es posible que los evaluadores deban ser expertos o no especialistas, como trabajadores de una plataforma de crowdsourcing (por ejemplo, Amazon Mechanical Turk). En algunas aplicaciones, incluso podrían ser los propios usuarios, respondiendo, por ejemplo, mediante encuestas o captchas reutilizados.

14

De cualquier manera, es necesario implementar un sistema de monitoreo (con o sin evaluadores humanos para evaluar el modelo en vivo), así como todos los procesos relevantes para definir qué hacer en caso de fallas y cómo prepararse para ellas.

Desafortunadamente, esto puede suponer mucho trabajo. De hecho, suele ser mucho más trabajo que construir y entrenar un modelo.

Si los datos siguen evolucionando, deberá actualizar sus conjuntos de datos y volver a entrenar su modelo con regularidad. Probablemente deberías automatizar todo el proceso tanto como sea posible. A continuación se muestran algunas cosas que puede automatizar:

- Recopile datos nuevos periódicamente y etiquételos (por ejemplo, utilizando evaluadores humanos).
- Escriba un script para entrenar el modelo y ajustar los hiperparámetros automáticamente. Este script podría ejecutarse automáticamente, por ejemplo, todos los días o todas las semanas, según sus necesidades.
- Escriba otro script que evalúe tanto el modelo nuevo como el modelo anterior en el conjunto de prueba actualizado e implemente el modelo en producción si el rendimiento no ha disminuido (si así fue, asegúrese de investigar por qué). El script probablemente debería probar el rendimiento de su modelo en varios subconjuntos del conjunto de prueba, como distritos pobres o ricos, distritos rurales o urbanos, etc.

También debe asegurarse de evaluar la calidad de los datos de entrada del modelo.

A veces, el rendimiento se degradará levemente debido a una señal de mala calidad (por ejemplo, un sensor defectuoso que envía valores aleatorios o la salida de otro equipo se vuelve obsoleta), pero puede pasar un tiempo antes de que su

El rendimiento del sistema se degrada lo suficiente como para activar una alerta. Si monitorea las entradas de su modelo, puede detectar esto antes. Por ejemplo, podría activar una alerta si a más y más entradas les falta una característica, o si la media o la desviación estándar se aleja demasiado del conjunto de entrenamiento, o si una característica categórica comienza a contener nuevas categorías.

Finalmente, asegúrese de mantener copias de seguridad de cada modelo que cree y de contar con el proceso y las herramientas para volver rápidamente a un modelo anterior, en caso de que el nuevo modelo comience a fallar gravemente por algún motivo. Tener copias de seguridad también permite comparar fácilmente los modelos nuevos con los anteriores.

De manera similar, debe mantener copias de seguridad de cada versión de sus conjuntos de datos para poder volver a un conjunto de datos anterior si el nuevo alguna vez se corrompe (por ejemplo, si los datos nuevos que se le agregan resultan estar llenos de valores atípicos).

Tener copias de seguridad de sus conjuntos de datos también le permite evaluar cualquier modelo con respecto a cualquier conjunto de datos anterior.

Como puede ver, el aprendizaje automático implica bastante infraestructura.

[El Capítulo 19](#) analiza algunos aspectos de esto, pero es un tema muy amplio llamado Operaciones de ML (MLOps), que merece su propio libro. Por lo tanto, no se sorprenda si su primer proyecto de aprendizaje automático requiere mucho esfuerzo y tiempo para construirlo e implementarlo en producción. Afortunadamente, una vez que toda la infraestructura esté en su lugar, pasar de la idea a la producción será mucho más rápido.

## ¡Pruébalo!

Esperamos que este capítulo le haya dado una buena idea de cómo es un proyecto de aprendizaje automático, además de mostrarle algunas de las herramientas que puede utilizar para entrenar un gran sistema. Como puede ver, gran parte del trabajo consiste en el paso de preparación de datos: crear herramientas de monitoreo, configurar canales de evaluación humana y automatizar la capacitación regular de modelos. Los algoritmos de aprendizaje automático son importantes, por supuesto, pero probablemente sea preferible sentirse cómodo con el proceso general y conocer bien tres o cuatro algoritmos en lugar de dedicar todo el tiempo a explorar algoritmos avanzados.

Entonces, si aún no lo ha hecho, ahora es un buen momento para tomar una computadora portátil, seleccionar un conjunto de datos que le interese e intentar realizar todo el proceso de la A a la Z. Un buen lugar para comenzar es una competencia

sitio web como [Kaggle](#): Tendrás un conjunto de datos con el que jugar, un objetivo claro y personas con quienes compartir la experiencia. ¡Divertirse!

## Ejercicios

Los siguientes ejercicios se basan en el conjunto de datos sobre vivienda de este capítulo:

1. Pruebe un regresor de máquina de vectores de soporte (sklearn.svm.SVR) con varios hiperparámetros, como kernel="linear" (con varios valores para el hiperparámetro C) o kernel="rbf" (con varios valores para los hiperparámetros C y gamma). Tenga en cuenta que las máquinas de vectores de soporte no se adaptan bien a conjuntos de datos grandes, por lo que probablemente debería entrenar su modelo solo en las primeras 5000 instancias del conjunto de entrenamiento y usar solo validación cruzada triple; de lo contrario, llevará horas. No se preocupe por lo que significan los hiperparámetros por ahora; Los analizaremos en [el Capítulo 5](#). ¿Cómo funciona el mejor predictor de RVS?
2. Intente reemplazar GridSearchCV con RandomizedSearchCV.
3. Intente agregar un transformador SelectFromModel en el proceso de preparación para seleccionar solo los atributos más importantes.
4. Intente crear un transformador personalizado que entrene un regresor de k vecinos más cercanos (sklearn.neighbors.KNeighborsRegressor) en su método fit() y genere las predicciones del modelo en su método transform(). Luego agregue esta característica a la canalización de preprocessamiento, utilizando la latitud y la longitud como entradas a este transformador.  
Esto agregará una característica en el modelo que corresponde al precio medio de la vivienda de los distritos más cercanos.
5. Explore automáticamente algunas opciones de preparación usando GridSearchCV.
6. Intente implementar la clase StandardScalerClone nuevamente desde scratch, luego agregue soporte para el método inverse\_transform(): ejecutar escalador.  
`inverse_transform(scaler.fit_transform(X))` debería devolver una matriz muy cercana a X. Luego agregue soporte para nombres de funciones:

establezca `feature_names_in_` en el método `fit()` si la entrada es un `DataFrame`. Este atributo debe ser una matriz NumPy de nombres de columnas. Por último, implemente el método `get_feature_names_out()`: debe tener un argumento opcional `input_features=None`. Si se aprueba, el método debe verificar que su longitud coincida con `n_features_in_`, y debe coincidir con `feature_names_in_` si está definido; entonces se deben devolver `input_features`. Si `input_features` es `Ninguno`, entonces el método debe devolver `feature_names_in_` si está definido o `np.array(["x0", "x1", ...])` con longitud `n_features_in_` en caso contrario.

Las soluciones a estos ejercicios están disponibles al final del cuaderno de este capítulo, en <https://homl.info/colab3>.

<sup>1</sup> El conjunto de datos original apareció en R. Kelley Pace y Ronald Barry, “Sparse Spatial Autoregressions”, *Statistics & Probability Letters* 33, no. 3 (1997): 291–297.

<sup>2</sup> Una porción de información enviada a un sistema de aprendizaje automático a menudo se denomina señal , en referencia a la teoría de la información de Claude Shannon, que desarrolló en los Laboratorios Bell para mejorar las telecomunicaciones. Su teoría: quieres una alta relación señal-ruido.

<sup>3</sup> Recuerde que el operador de transposición convierte un vector de columna en un vector de fila (y viceversa).

<sup>4</sup> Es posible que también deba verificar las restricciones legales, como los campos privados que nunca deben copiarse en almacenes de datos no seguros.

<sup>5</sup> La desviación estándar generalmente se denota por  $\sigma$ (la letra griega sigma) y es la raíz cuadrada de la varianza, que es el promedio de la desviación al cuadrado de la media. Cuando una característica tiene una distribución normal en forma de campana (también llamada distribución gaussiana), lo cual es muy común, se aplica la regla “68-95-99.7”: alrededor del 68% de los valores caen dentro de 1 $\sigma$  de la media, el 95% dentro de 2 $\sigma$  y 99,7% dentro de 3 $\sigma$

<sup>6</sup> A menudo verás a personas establecer la semilla aleatoria en 42. Este número no tiene ninguna propiedad especial, aparte de ser la respuesta a la pregunta fundamental de la vida, el universo y todo.

<sup>7</sup> La información de ubicación es en realidad bastante aproximada y, como resultado, muchos distritos tendrán exactamente el mismo ID, por lo que terminarán en el mismo conjunto (prueba o tren). Esto introduce un desafortunado sesgo de muestreo.

<sup>8</sup> Si estás leyendo esto en escala de grises, toma un bolígrafo rojo y garabatea la mayor parte del texto. costa desde el Área de la Bahía hasta San Diego (como era de esperar). También puedes agregar una mancha amarilla alrededor de Sacramento.

**9** Para obtener más detalles sobre los principios de diseño, consulte Lars Buitinck et al., “API Design for Machine Learning Software: Experiences from the Scikit-Learn Project”, preimpresión de arXiv arXiv:1309.0238 (2013).

**10** Algunos predictores también proporcionan métodos para medir la confianza de sus predicciones

**11** Para cuando lea estas líneas, es posible que todos los transformadores genera Pandas DataFrames cuando reciben un DataFrame como entrada: Pandas dentro, Pandas fuera. Probablemente habrá una opción de configuración global para esto: `sklearn.set_config(pandas_in_out=True)`.

**12** Consulte la documentación de SciPy para obtener más detalles.

**13** En pocas palabras, una API REST (o RESTful) es una API basada en HTTP que sigue algunas convenciones, como el uso de verbos HTTP estándar para leer, actualizar, crear o eliminar recursos (GET, POST, PUT y DELETE) y usando JSON para las entradas y salidas.

**14** Un captcha es una prueba para garantizar que un usuario no es un robot. Estas pruebas a menudo han sido

Se utiliza como una forma económica de etiquetar los datos de entrenamiento.

# Capítulo 3. Clasificación

---

En el Capítulo 1 mencioné que las tareas de aprendizaje supervisado más comunes son la regresión (predecir valores) y la clasificación (predecir clases).

En el Capítulo 2 exploramos una tarea de regresión, prediciendo valores de vivienda, utilizando varios algoritmos como regresión lineal, árboles de decisión y bosques aleatorios (que se explicarán con más detalle en capítulos posteriores).

Ahora centraremos nuestra atención en los sistemas de clasificación.

## MNIST

En este capítulo utilizaremos el conjunto de datos MNIST, que es un conjunto de 70.000 imágenes pequeñas de dígitos escritos a mano por estudiantes de secundaria y empleados de la Oficina del Censo de EE. UU. Cada imagen está etiquetada con el dígito que representa. Este conjunto se ha estudiado tanto que a menudo se le llama el “hola mundo” del aprendizaje automático: cada vez que a las personas se les ocurre un nuevo algoritmo de clasificación, sienten curiosidad por ver cómo funcionará en MNIST, y cualquiera que aprenda aprendizaje automático se enfrenta a esto. conjunto de datos tarde o temprano.

Scikit-Learn proporciona muchas funciones de ayuda para descargar conjuntos de datos populares. MNIST es uno de ellos. El siguiente código recupera el conjunto de datos MNIST de OpenML.org:

1

```
desde sklearn.datasets importar fetch_openml  
  
mnist = fetch_openml('mnist_784', as_frame=False)
```

El paquete `sklearn.datasets` contiene principalmente tres tipos de funciones: funciones `fetch_*` como `fetch_openml()` para descargar conjuntos de datos de la vida real, funciones `load_*` para cargar pequeños conjuntos de datos de juguetes incluidos con Scikit-Learn (por lo que no es necesario descargarlos), a través de Internet) y funciones `make_*` para generar conjuntos de datos falsos, útiles para pruebas.

Los conjuntos de datos generados generalmente se devuelven como una tupla (`X, y`) que contiene los datos de entrada y los objetivos, ambos como matrices NumPy. Otros conjuntos de datos son

devueltos como objetos `sklearn.utils.Bunch`, que son diccionarios a cuyas entradas también se puede acceder como atributos. Generalmente contienen las siguientes entradas:

"DESCR"

Una descripción del conjunto de datos.

"datos"

Los datos de entrada, generalmente como una matriz NumPy 2D.

"objetivo"

Las etiquetas, generalmente como una matriz NumPy 1D

La función `fetch_openml()` es un poco inusual ya que de forma predeterminada devuelve las entradas como un Pandas DataFrame y las etiquetas como una Pandas Series (a menos que el conjunto de datos sea escaso). Pero el conjunto de datos MNIST contiene imágenes y los DataFrames no son ideales para eso, por lo que es preferible establecer `as_frame=False` para obtener los datos como matrices NumPy. Veamos estas matrices:

```
>>> X, y = mnist.datos, mnist.objetivo
>>> X
matrix([[0., 0., 0., ..., 0., 0., 0.], [0., 0., 0., ..., 0., 0., 0.], [0.,
0., 0., ..., 0., 0., 0.],
        ...,
        [0., 0., 0., ..., 0., 0., 0.], [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]])
```

```
>>> X.shape
(70000, 784) >>>
y
matrix(['5', '0', '4', ..., '4', '5', '6'], dtype=object ) >>> forma y (70000,
```

Hay 70.000 imágenes y cada imagen tiene 784 características. Esto se debe a que cada imagen tiene  $28 \times 28$  píxeles y cada característica simplemente representa la intensidad de un píxel, de 0 (blanco) a 255 (negro). tomemos un

eche un vistazo a un dígito del conjunto de datos ([Figura 3-1](#)). Todo lo que necesitamos hacer es tomar el vector de características de una instancia, remodelarlo en una matriz de  $28 \times 28$  y mostrarlo usando la función imshow() de Matplotlib. Usamos cmap="binary" para obtener un mapa de color en escala de grises donde 0 es blanco y 255 es negro:

```
importar matplotlib.pyplot como plt

def plot_digit(image_data): imagen =
    image_data.reshape(28, 28) plt.imshow(imagen,
    cmap="binary") plt.axis("off")

algún_dígito = X[0]
plot_digit(algun_dígito) plt.show()
```



Figura 3-1. Ejemplo de una imagen MNIST

Esto parece un 5, y de hecho eso es lo que nos dice la etiqueta:

```
>>> y[0] '5'
```

Para darle una idea de la complejidad de la tarea de clasificación, [la Figura 3-2](#) muestra algunas imágenes más del conjunto de datos MNIST.

¡Pero espera! Siempre debes crear un conjunto de prueba y dejarlo a un lado antes de inspeccionar los datos de cerca. El conjunto de datos MNIST devuelto por `fetch_openml()` en realidad ya está dividido en un conjunto de entrenamiento (las primeras 60 000 imágenes) y un conjunto de prueba (las últimas 10 000 imágenes):<sup>2</sup>

```
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

El conjunto de entrenamiento ya está mezclado para nosotros, lo cual es bueno porque garantiza que todos los pliegues de validación cruzada serán similares (no queremos que a un pliegue le falten algunos dígitos). Además, algunos algoritmos de aprendizaje son sensibles al orden de las instancias de entrenamiento y su rendimiento es deficiente si obtienen muchas instancias similares seguidas. Mezclar el conjunto de datos garantiza que esto<sup>3</sup> no suceda.



Figura 3-2. Dígitos del conjunto de datos MNIST

## Entrenando un clasificador binario

Simplifiquemos el problema por ahora e intentemos identificar solo un dígito, por ejemplo, el número 5. Este “detector de 5” será un ejemplo de clasificador binario, capaz de distinguir entre solo dos clases, 5 y no 5. Primero crearemos los vectores objetivo para esta tarea de clasificación:

```
y_train_5 = (y_train == '5') # Verdadero para todos los 5, Falso para todos los demás
dígitos y_test_5
= (y_test == '5')
```

Ahora elijamos un clasificador y entrenémoslo. Un buen lugar para comenzar es con un clasificador de descenso de gradiente estocástico (SGD o GD estocástico), utilizando la clase SGDClassifier de Scikit-Learn. Este clasificador es capaz de manejar conjuntos de datos muy grandes de manera eficiente. Esto se debe en parte a que SGD se ocupa de las instancias de capacitación de forma independiente, una a la vez, lo que también hace que SGD sea muy adecuado para el aprendizaje en línea, como verá más adelante. Creemos un SGDClassifier y entrenémoslo en todo el conjunto de entrenamiento:

```
desde sklearn.linear_model importar SGDClassifier
sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train, y_train_5)
```

Ahora podemos usarlo para detectar imágenes del número 5:

```
>>> sgd_clf.predict([algún_dígito])
matriz([Verdadero])
```

El clasificador supone que esta imagen representa un 5 (Verdadero). ¡Parece que acertó en este caso particular! Ahora, evaluemos el rendimiento de este modelo.

## Medidas de desempeño

Evaluar un clasificador suele ser mucho más complicado que evaluar un regresor, por lo que dedicaremos gran parte de este capítulo a este tema. Allá

Hay muchas medidas de desempeño disponibles, así que toma otro café y prepárate para aprender un montón de conceptos y acrónimos nuevos.

## Medición de la precisión mediante validación cruzada

Una buena manera de evaluar un modelo es usar validación cruzada, tal como lo hizo en el Capítulo 2. Usemos la función `cross_val_score()` para evaluar nuestro modelo `SGDClassifier`, usando validación cruzada de k veces con tres pliegues.

Recuerde que la validación cruzada de k veces significa dividir el conjunto de entrenamiento en k veces (en este caso, tres), luego entrenar el modelo k veces, manteniendo un pliegue diferente cada vez para su evaluación (consulte el Capítulo 2):

```
>>> de sklearn.model_selection importar cross_val_score >>> cross_val_score(sgd_clf,
X_train, y_train_5, cv=3, scoring="accuracy") array([0.95035, 0.96035, 0.9604])
```

¡Guau! ¿Más del 95% de precisión (proporción de predicciones correctas) en todos los pliegues de validación cruzada? Esto parece increíble, ¿no? Bueno, antes de que te emociones demasiado, veamos un clasificador ficticio que simplemente clasifica cada imagen en la clase más frecuente, que en este caso es la clase negativa (es decir, no 5) :

```
desde sklearn.dummy importar DummyClassifier

dummy_clf = DummyClassifier()
dummy_clf.fit(X_train, y_train_5)
print(any(dummy_clf.predict(X_train))) # imprime Falso: no 5s
detectado
```

¿Puedes adivinar la precisión de este modelo? Vamos a averiguar:

```
>>> cross_val_score(dummy_clf, X_train, y_train_5, cv=3, scoring="accuracy")
matriz([0.90965, 0.90965,
0.90965])
```

Así es, ¡tiene más del 90% de precisión! Esto se debe simplemente a que sólo alrededor del 10% de las imágenes son 5, por lo que si siempre adivinas que una imagen no es un 5, acertarás aproximadamente el 90% de las veces. Supera a Nostradamus.

Esto demuestra por qué la precisión generalmente no es la medida de desempeño preferida para los clasificadores, especialmente cuando se trata de conjuntos de datos sesgados (es decir, cuando algunas clases son mucho más frecuentes que otras). Una forma mucho mejor de evaluar el desempeño de un clasificador es observar la matriz de confusión (CM).

## IMPLEMENTACIÓN DE LA VALIDACIÓN CRUZADA

Ocasionalmente, necesitará más control sobre el proceso de validación cruzada que el que ofrece Scikit-Learn. En estos casos, usted mismo puede implementar la validación cruzada. El siguiente código hace aproximadamente lo mismo que la función `cross_val_score()` de Scikit-Learn e imprime el mismo resultado:

```

desde sklearn.model_selection importar StratifiedKFold desde
sklearn.base importar clon

skfolds = StratifiedKFold(n_splits=3) # agrega shuffle=True si el conjunto de datos
es
# no todavía
barajado
para train_index, test_index en skfolds.split(X_train, y_train_5): clone_clf
= clon(sgd_clf)

    X_train_folds = X_train[índice_tren] y_train_folds
    = y_train_5[índice_tren]
    X_test_fold = X_train[test_index] y_test_fold
    = y_train_5[test_index]

    clone_clf.fit(X_train_folds, y_train_folds) y_pred =
    clone_clf.predict(X_test_fold) n_correct = suma(y_pred
    == y_test_fold) print(n_correct / len(y_pred)) #
    imprime 0.95035, 0.96035 y 0.9604

```

La clase `StratifiedKFold` realiza un muestreo estratificado (como se explica en [el Capítulo 2](#)) para producir pliegues que contienen una proporción representativa de cada clase. En cada iteración, el código crea un clon del clasificador, lo entrena en los pliegues de entrenamiento y hace predicciones en el pliegue de prueba. Luego cuenta el número de predicciones correctas y genera la proporción de predicciones correctas.

## Matrices de confusión

La idea general de una matriz de confusión es contar el número de veces que las instancias de clase A se clasifican como clase B, para todos los pares A/B. Para

Por ejemplo, para saber la cantidad de veces que el clasificador confundió imágenes de 8 con 0, miraría la fila 8, columna 0 de la matriz de confusión.

Para calcular la matriz de confusión, primero es necesario tener un conjunto de predicciones para poder compararlas con los objetivos reales. Puede hacer predicciones en el conjunto de prueba, pero es mejor dejarlo intacto por ahora (recuerde que desea usar el conjunto de prueba solo al final de su proyecto, una vez que tenga un clasificador que esté listo para lanzar).

En su lugar, puedes usar la función `cross_val_predict()`:

```
desde sklearn.model_selection importar cross_val_predict

y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
```

Al igual que la función `cross_val_score()`, `cross_val_predict()` realiza una validación cruzada de  $k$  veces, pero en lugar de devolver las puntuaciones de la evaluación, devuelve las predicciones realizadas en cada pliegue de la prueba. Esto significa que obtienes una predicción limpia para cada instancia del conjunto de entrenamiento (por "limpia" me refiero a "fuera de muestra": el modelo hace predicciones sobre datos que nunca vio durante el entrenamiento).

Ahora está listo para obtener la matriz de confusión usando la función `confusion_matrix()`. Simplemente pásele las clases objetivo (`y_train_5`) y las clases predichas (`y_train_pred`):

```
>>> desde sklearn.metrics importar confusion_matrix >>> cm =
confusion_matrix(y_train_5, y_train_pred)
>>> centímetros
matriz ([[53892, 687], [1891,
3530]])
```

Cada fila de una matriz de confusión representa una clase real, mientras que cada columna representa una clase prevista. La primera fila de esta matriz considera imágenes no 5 (la clase negativa): 53.892 de ellas fueron clasificadas correctamente como no 5 (se llaman verdaderos negativos), mientras que las 687 restantes fueron clasificadas erróneamente como 5 (falsos positivos, también llamados errores tipo I). La segunda fila considera las imágenes de 5 (la clase positiva): 1.891 fueron clasificadas erróneamente como no 5 (falsos negativos, también llamados tipo

11 errores), mientras que los 3.530 restantes se clasificaron correctamente como 5 (verdaderos positivos). Un clasificador perfecto solo tendría verdaderos positivos y verdaderos negativos, por lo que su matriz de confusión tendría valores distintos de cero solo en su diagonal principal (de arriba a la izquierda a abajo a la derecha):

```
>>> y_train_perfect_predictions = y_train_5 # finge que alcanzamos la perfección >>>
confusion_matrix(y_train_5, y_train_perfect_predictions) array([[54579, [0], 0, 5421]])
```

La matriz de confusión le brinda mucha información, pero a veces es posible que prefiera una métrica más concisa. Una cuestión interesante de observar es la precisión de las predicciones positivas; esto se llama precisión del clasificador ([Ecuación 3-1](#)).

### Ecuación 3-1. Precisión

$$\text{precisión} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

TP es el número de verdaderos positivos y FP es el número de falsos positivos.

Una forma trivial de tener una precisión perfecta es crear un clasificador que siempre haga predicciones negativas, excepto una única predicción positiva en la instancia en la que tiene más confianza. Si esta predicción es correcta, entonces el clasificador tiene una precisión del 100% (precisión = 1/1 = 100%). Obviamente, un clasificador de este tipo no sería muy útil, ya que ignoraría todos los casos positivos excepto uno. Por lo tanto, la precisión generalmente se usa junto con otra métrica llamada recuperación, también llamada sensibilidad o tasa de verdaderos positivos (TPR): esta es la proporción de instancias positivas que el clasificador detecta correctamente ([Ecuación 3-2](#)).

### Ecuación 3-2. Recordar

$$\text{recordar} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

FN es, por supuesto, el número de falsos negativos.

Si está confundido acerca de la matriz de confusión, la Figura 3-3 puede ayudarle.

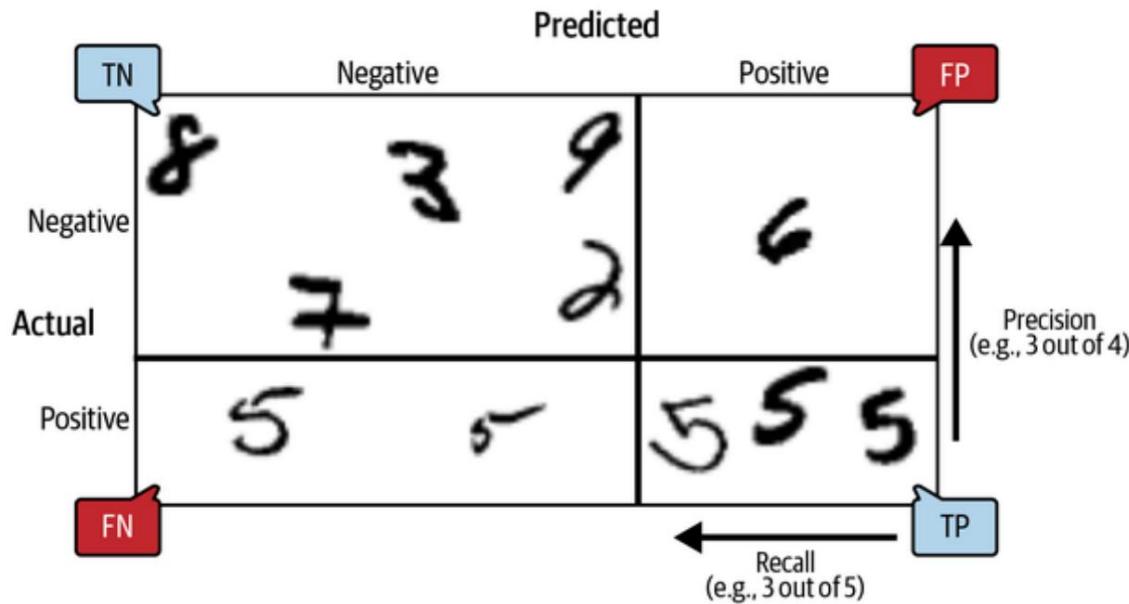


Figura 3-3. Una matriz de confusión ilustrada que muestra ejemplos de verdaderos negativos (arriba a la izquierda), falsos positivos (arriba a la derecha), falsos negativos (abajo a la izquierda) y verdaderos positivos (abajo a la derecha).

## Precisión y recuperación

Scikit-Learn proporciona varias funciones para calcular métricas de clasificador, incluida la precisión y la recuperación:

```
>>> de sklearn.metrics importar puntuación_precisión, puntuación_recuperación >>>
puntuación_precisión(y_train_5, y_train_pred) # == 3530 / (687 + 3530)

0.8370879772350012
>>> puntuación_recuperación(y_train_5, y_train_pred) # == 3530 / (1891 + 3530)

0.6511713705958311
```

Ahora nuestro detector de 5 no se ve tan brillante como cuando analizamos su precisión. Cuando afirma que una imagen representa un 5, es correcto sólo el 83,7% de las veces. Además, sólo detecta el 65,1% de los 5.

A menudo resulta conveniente combinar precisión y recuperación en una única métrica llamada puntuación F, especialmente cuando se necesita una única métrica para comparar dos clasificadores. La puntuación F es la media armónica de precisión y recuperación.

(Ecuación 3-3). Mientras que la media regular trata todos los valores por igual, la media armónica da mucho más peso a los valores bajos. Como resultado, el clasificador sólo obtendrá una puntuación F alta si tanto la recuperación como la precisión son altas.

Ecuación 3-3. puntuación F

$$F1 = \frac{2}{\frac{1}{\text{precisión}} + \frac{1}{\text{recuperación}}} = 2 \times \frac{\text{precisión} \times \text{recuperación}}{\text{precisión} + \text{recuperación}} = \frac{\text{TP}}{\text{TP} + \frac{\text{FN} + \text{FP}}{2}}$$

Para calcular la puntuación F, simplemente llame a la función `f1_score()`:

```
>>> desde sklearn.metrics importar f1_score >>>
f1_score(y_train_5, y_train_pred) 0.7325171197343846
```

La puntuación F favorece a los clasificadores que tienen una precisión y un recuerdo similares. Esto no siempre es lo que desea: en algunos contextos lo que más le importa es la precisión, y en otros contextos lo que realmente le importa es la recuperación. Por ejemplo, si entrenó un clasificador para detectar videos que son seguros para los niños, probablemente prefiera un clasificador que rechace muchos videos buenos (baja recuperación) pero mantenga solo los seguros (alta precisión), en lugar de un clasificador que tenga mucha mayor recuerdo, pero permite que aparezcan algunos videos realmente malos en su producto (en tales casos, es posible que incluso desee agregar una canalización humana para verificar la selección de videos del clasificador). Por otro lado, supongamos que entrena a un clasificador para detectar ladrones en imágenes de vigilancia: probablemente esté bien si su clasificador solo tiene un 30% de precisión siempre que tenga un 99% de recuperación (seguro, los guardias de seguridad recibirán algunas alertas falsas, pero atrapan a casi todos los ladrones).

Desafortunadamente, no se pueden tener ambas cosas: aumentar la precisión reduce la recuperación y viceversa. Esto se llama equilibrio precisión/recuperación.

### La compensación entre precisión y recuperación

Para comprender esta compensación, veamos cómo SGDClassifier toma sus decisiones de clasificación. Para cada instancia, calcula una puntuación basada en una función de decisión. Si esa puntuación es mayor que un umbral, asigna la instancia a la clase positiva; en caso contrario lo asigna al negativo

clase. La Figura 3-4 muestra algunos dígitos ubicados desde la puntuación más baja a la izquierda hasta la puntuación más alta a la derecha. Supongamos que el umbral de decisión está ubicado en la flecha central (entre los dos 5): encontrará 4 verdaderos positivos (5 reales) a la derecha de ese umbral y 1 falso positivo (en realidad, un 6). Por tanto, con ese umbral, la precisión es del 80% (4 sobre 5). Pero de 6 5 reales, el clasificador solo detecta 4, por lo que la recuperación es del 67% (4 de 6). Si eleva el umbral (muévalo hacia la flecha de la derecha), el falso positivo (el 6) se convierte en un verdadero negativo, aumentando así la precisión (hasta el 100% en este caso), pero un verdadero positivo se convierte en un falso negativo., disminuyendo el retiro hasta un 50%. Por el contrario, reducir el umbral aumenta la recuperación y reduce la precisión.

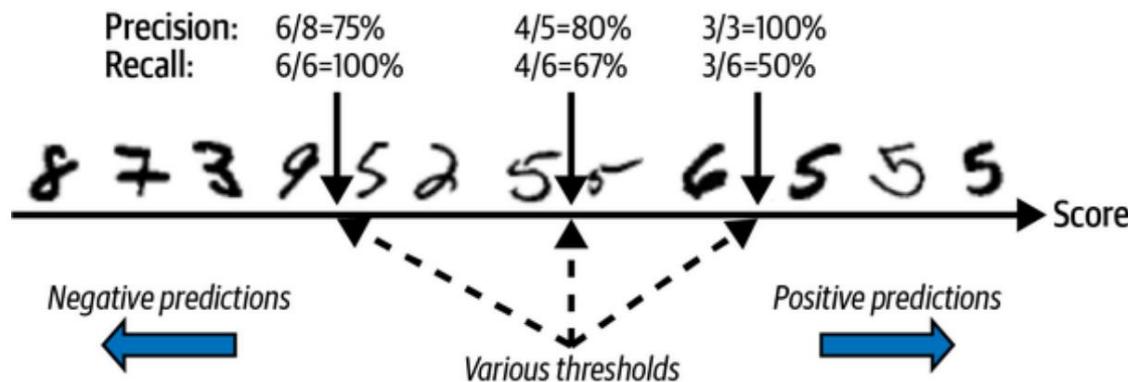


Figura 3-4. La compensación precisión/recuperación: las imágenes se clasifican según su puntuación del clasificador, y aquellas que están por encima del umbral de decisión elegido se consideran positivas; cuanto más alto es el umbral, menor es la recuperación, pero (en general) mayor es la precisión

Scikit-Learn no le permite establecer el umbral directamente, pero sí le brinda acceso a las puntuaciones de decisión que utiliza para hacer predicciones. En lugar de llamar al método `predict()` del clasificador, puede llamar a su método `decision_function()`, que devuelve una puntuación para cada instancia, y luego usar cualquier umbral que desee para hacer predicciones basadas en esas puntuaciones:

```
>>> y_scores = sgd_clf.decision_function([algún_dígito]) >>> y_scores

array([2164.22030239]) >>>
umbral = 0 >>>

y_some_digit_pred = (y_scores > umbral) array([ Verdadero])
```

El SGDClassifier utiliza un umbral igual a 0, por lo que el código anterior devuelve el mismo resultado que el método predict() (es decir, Verdadero). Elevemos el umbral:

```
>>> umbral = 3000 >>>
y_some_digit_pred = (y_scores > umbral) >>> y_some_digit_pred
array([False])
```

Esto confirma que elevar el umbral disminuye la recuperación. La imagen en realidad representa un 5 y el clasificador lo detecta cuando el umbral es 0, pero lo omite cuando el umbral se incrementa a 3000.

¿Cómo se decide qué umbral utilizar? Primero, use la función cross\_val\_predict() para obtener las puntuaciones de todas las instancias en el conjunto de entrenamiento, pero esta vez especifique que desea devolver puntuaciones de decisión en lugar de predicciones:

```
puntuaciones_y = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
                                    método="función_decision")
```

Con estos puntajes, use la función precision\_recall\_curve() para calcular la precisión y la recuperación de todos los umbrales posibles (la función agrega una última precisión de 0 y una última recuperación de 1, correspondiente a un umbral infinito):

```
desde sklearn.metrics importe precision_recall_curve
precisiones, retiros, umbrales =
precision_recall_curve(y_train_5, y_scores)
```

Finalmente, use Matplotlib para trazar la precisión y la recuperación como funciones del valor umbral ([Figura 3-5](#)). Mostremos el umbral de 3000 que seleccionamos:

```
plt.plot(umbrales, precisiones[:-1], "b--", etiqueta="Precisión", ancho de línea=2) plt.plot(umbrales,
recuperaciones[:-1],
"g-", etiqueta=" Recordar", ancho de línea=2) plt.vlines(umbral, 0, 1.0, "k", "punteado",
etiqueta="umbral")
```

[...] # embellecer la figura: agregar cuadrícula, leyenda, eje, etiquetas y círculos

```
plt.mostrar()
```

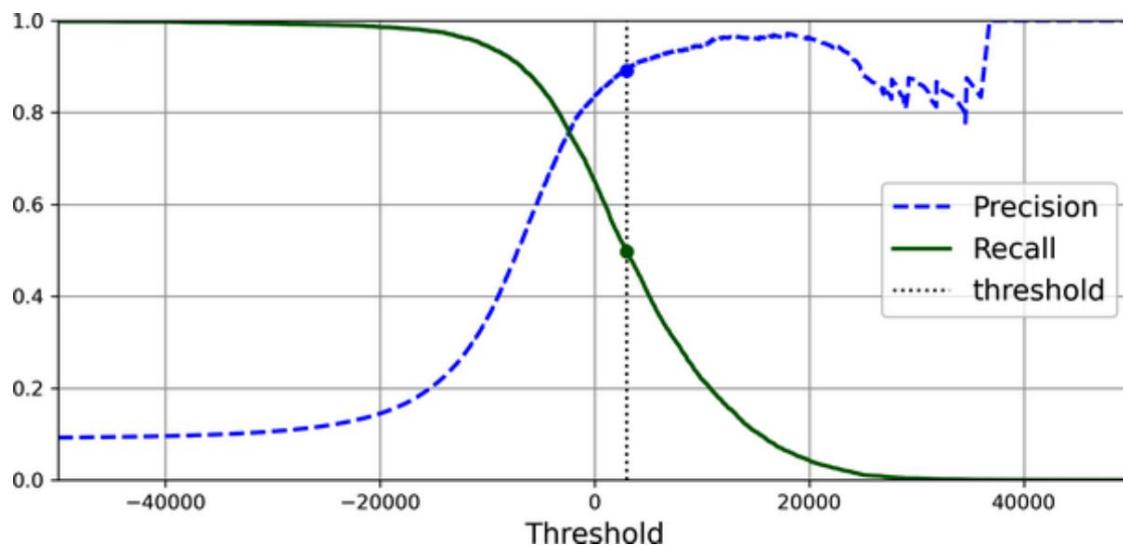


Figura 3-5. Precisión y recuperación frente al umbral de decisión

### NOTA

Quizás se pregunte por qué la curva de precisión tiene más obstáculos que la curva de recuperación de la Figura 3-5. La razón es que a veces la precisión puede disminuir cuando se eleva el umbral (aunque en general aumentará). Para entender por qué, mire nuevamente la Figura 3-4 y observe lo que sucede cuando comienza desde el umbral central y lo mueve solo un dígito hacia la derecha: la precisión va de 4/5 (80%) a 3/4 (75%). Por otro lado, la recuperación sólo puede disminuir cuando se aumenta el umbral, lo que explica por qué su curva parece suave.

En este valor umbral, la precisión es cercana al 90% y la recuperación es de alrededor del 50%. Otra forma de seleccionar una buena compensación entre precisión y recuperación es trazar la precisión directamente frente a la recuperación, como se muestra en la Figura 3-6 (se muestra el mismo umbral):

```
plt.plot(recuerdos, precisiones, ancho de línea=2, etiqueta="Curva  
de precisión/recuerdo") [...] # embellecer la  
figura: agregar etiquetas, cuadrícula, leyenda, flecha y texto
```

```
plt.mostrar()
```

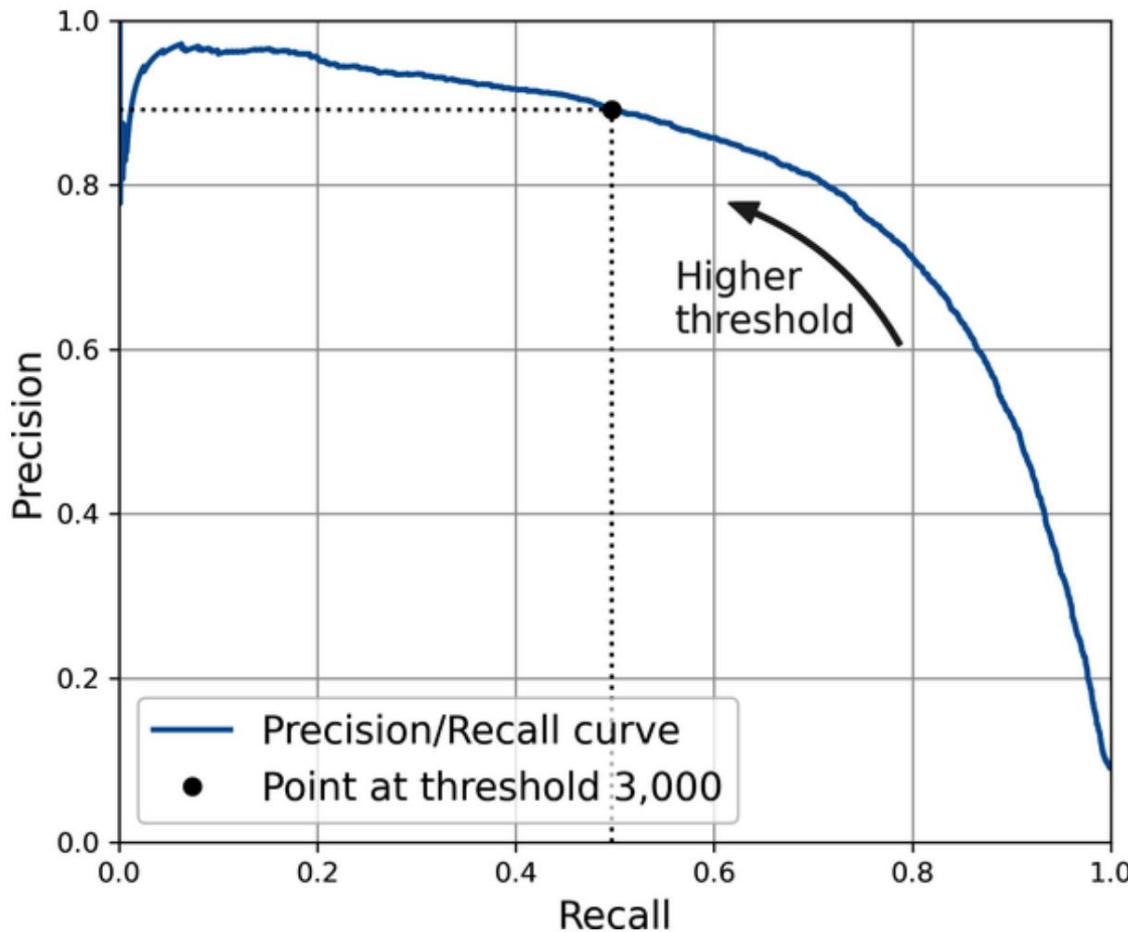


Figura 3-6. Precisión versus recuperación

Puede ver que la precisión realmente comienza a caer drásticamente alrededor del 80% de recuperación. Probablemente desee seleccionar un equilibrio entre precisión y recuperación justo antes de esa caída; por ejemplo, alrededor del 60% de recuperación. Pero claro, la elección depende de tu proyecto.

Suponga que decide aspirar a una precisión del 90%. Podrías usar el primer gráfico para encontrar el umbral que necesitas usar, pero eso no es muy preciso. Alternativamente, puede buscar el umbral más bajo que le brinde al menos un 90% de precisión. Para esto, puede utilizar el método `argmax()` de la matriz NumPy. Esto devuelve el primer índice del valor máximo, que en este caso significa el primer valor Verdadero:

```
>>> idx_for_90_precision = (precisiones >= 0.90).argmax() >>
umbral_for_90_precision =
umbrales[idx_for_90_precision] >>
umbral_for_90_precision
3370.0194991439557
```

Para hacer predicciones (en el conjunto de entrenamiento por ahora), en lugar de llamar al método predict() del clasificador, puedes ejecutar este código:

```
y_train_pred_90 = (y_scores >= umbral_para_90_precision)
```

Comprobemos la precisión de estas predicciones y recordemos:

```
>>> puntuación_precision(y_train_5, y_train_pred_90)
0.9000345901072293
>>> recuperación_en_90_precision = recuperación_score(y_train_5,
y_tren_pred_90) >>>
recuperación_en_90_precision
0.4799852425751706
```

¡Genial, tienes un clasificador de precisión del 90%! Como puede ver, es bastante fácil crear un clasificador con prácticamente cualquier precisión que desee: simplemente establezca un umbral suficientemente alto y listo. Pero espera, no tan rápido: ¡un clasificador de alta precisión no es muy útil si su recuperación es demasiado baja! Para muchas aplicaciones, un 48% de recuperación no sería nada bueno.

#### CONSEJO

Si alguien dice: "Alcancemos el 99% de precisión", debería preguntar: "¿En qué retirada?".

## La curva ROC

La curva de característica operativa del receptor (ROC) es otra herramienta común utilizada con clasificadores binarios. Es muy similar a la curva de precisión/recuperación, pero en lugar de representar la precisión versus la recuperación, la curva ROC traza la tasa de verdaderos positivos (otro nombre para la recuperación) frente a la tasa de falsos positivos (FPR). El FPR (también llamado fall-out) es la proporción de casos negativos que se clasifican incorrectamente como positivos. Es igual a 1: la tasa de verdaderos negativos (TNR), que es la proporción de casos negativos que se clasifican correctamente como negativos. La TNR también se llama especificidad. Por lo tanto, la curva ROC representa la sensibilidad (recuerdo) versus 1 – especificidad.

Para trazar la curva ROC, primero use la función `roc_curve()` para calcular el TPR y el FPR para varios valores de umbral:

```
desde sklearn.metrics importar roc_curve  
  
fpr, tpr, umbrales = roc_curve(y_train_5, y_scores)
```

Luego puedes trazar el FPR contra el TPR usando Matplotlib. El siguiente código produce el gráfico de **la Figura 3-7**. Para encontrar el punto que corresponde al 90% de precisión, debemos buscar el índice del umbral deseado. Dado que en este caso los umbrales se enumeran en orden decreciente, usamos `<=` en lugar de `>=` en la primera línea:

```
idx_for_threshold_at_90 = (umbrales <=  
umbral_for_90_precision).argmax() tpr_90, fpr_90  
= tpr[idx_for_threshold_at_90], fpr[idx_for_threshold_at_90]  
  
plt.plot(fpr, tpr, linewidth=2, label=" Curva ROC") plt.plot([0, 1], [0, 1],  
'k:', label="Curva ROC del clasificador aleatorio") plt .plot([fpr_90], [tpr_90], "ko",  
label="Umbral para 90% de precisión") [...] # embellecer la figura: agregar etiquetas,  
cuadrícula,  
leyenda, flecha y texto plt.show( )
```

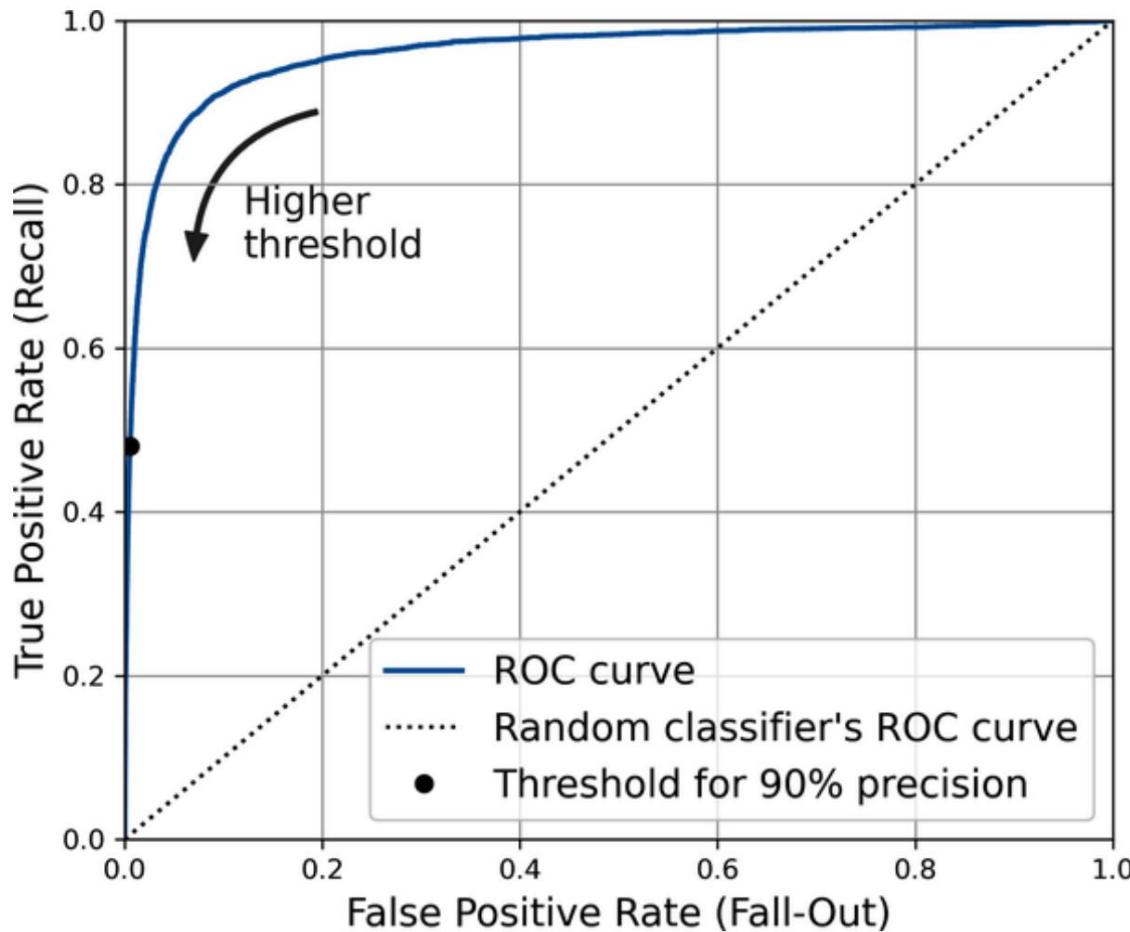


Figura 3-7. Una curva ROC que representa la tasa de falsos positivos frente a la tasa de verdaderos positivos para todos los umbrales posibles; el círculo negro resalta la proporción elegida (con un 90% de precisión y un 48% de recuperación)

Una vez más, hay una compensación: cuanto mayor es el recuerdo (TPR), más falsos positivos (FPR) produce el clasificador. La línea de puntos representa la curva ROC de un clasificador puramente aleatorio; un buen clasificador se mantiene lo más lejos posible de esa línea (hacia la esquina superior izquierda).

Una forma de comparar clasificadores es medir el área bajo la curva (AUC). Un clasificador perfecto tendrá un ROC AUC igual a 1, mientras que un clasificador puramente aleatorio tendrá un ROC AUC igual a 0,5. Scikit-Learn proporciona una función para estimar el AUC de la República de China:

```
>>> desde sklearn.metrics importar roc_auc_score >>>
roc_auc_score(y_train_5, y_scores)
0.9604938554008616
```

## CONSEJO

Dado que la curva ROC es tan similar a la curva de precisión/recuperación (PR), quizás se pregunte cómo decidir cuál usar. Como regla general, debería preferir la curva PR siempre que la clase positiva sea rara o cuando le importen más los falsos positivos que los falsos negativos. De lo contrario, utilice la curva ROC.

Por ejemplo, al observar la curva ROC anterior (y la puntuación AUC de ROC), puede pensar que el clasificador es realmente bueno. Pero esto se debe principalmente a que hay pocos aspectos positivos (5) en comparación con los negativos (no 5). En contraste, la curva PR deja claro que el clasificador tiene margen de mejora: la curva realmente podría estar más cerca de la esquina superior derecha (consulte nuevamente [la Figura 3-6](#) ).

Creemos ahora un RandomForestClassifier, cuya curva PR y puntuación F podemos comparar [1](#) con las del SGDClassifier:

```
de sklearn.ensemble importar RandomForestClassifier
```

```
forest_clf = RandomForestClassifier(random_state=42)
```

La función precision\_recall\_curve() espera etiquetas y puntuaciones para cada instancia, por lo que necesitamos entrenar el clasificador de bosque aleatorio y hacer que asigne una puntuación a cada instancia. Pero la clase RandomForestClassifier no tiene un método decision\_function(), debido a la forma en que

funciona (cubriremos esto en el [Capítulo 7](#)). Afortunadamente, tiene un método predict\_proba() que devuelve probabilidades de clase para cada instancia, y podemos usar la probabilidad de la clase positiva como puntuación, por lo que funcionará bien. Podemos llamar a la función cross\_val\_predict() para entrenar RandomForestClassifier usando validación cruzada <sup>4</sup> y hacer que prediga las probabilidades de clase para cada imagen de la siguiente manera:

```
y_probas_forest = cross_val_predict(forest_clf, X_train, y_train_5, cv=3,
método='predict_proba')
```

Veamos las probabilidades de clase para las dos primeras imágenes del conjunto de entrenamiento:

```
>>> y_probas_forest[:2]
array([[0.11, 0.89], [0.99,
0.01]])
```

El modelo predice que la primera imagen es positiva con un 89% de probabilidad y predice que la segunda imagen es negativa con un 99% de probabilidad. Dado que cada imagen es positiva o negativa, las probabilidades en cada fila suman 100%.

#### ADVERTENCIA

Éstas son probabilidades estimadas , no probabilidades reales. Por ejemplo, si observa todas las imágenes que el modelo clasificó como positivas con una probabilidad estimada entre el 50% y el 60%, aproximadamente el 94% de ellas son realmente positivas. Así pues, las probabilidades estimadas del modelo eran demasiado bajas en este caso, pero los modelos también pueden ser demasiado confiados. El paquete sklearn.calibration contiene herramientas para calibrar las probabilidades estimadas y acercarlas mucho más a las probabilidades reales. Consulte la sección de material adicional en [el cuaderno de este capítulo](#) para obtener más detalles.

La segunda columna contiene las probabilidades estimadas para la clase positiva, así que pasémoslas a la función precision\_recall\_curve():

```
y_scores_forest = y_probas_forest[:, 1] precisions_forest,
recuerda_bosque, umbrales_bosque = precision_recall_curve( y_train_5,
y_scores_forest)
```

Ahora estamos listos para trazar la curva PR. También resulta útil trazar la primera curva PR para ver cómo se comparan ([Figura 3-8](#)):

```
plt.plot(recalls_forest, precisions_forest, "b-", linewidth=2, label="Random Forest")
plt.plot(recuerdos, precisiones,
"--", linewidth=2, label="SGD") [...] # embellecer la figura: agregar etiquetas, cuadrícula
y leyenda plt.show()
```

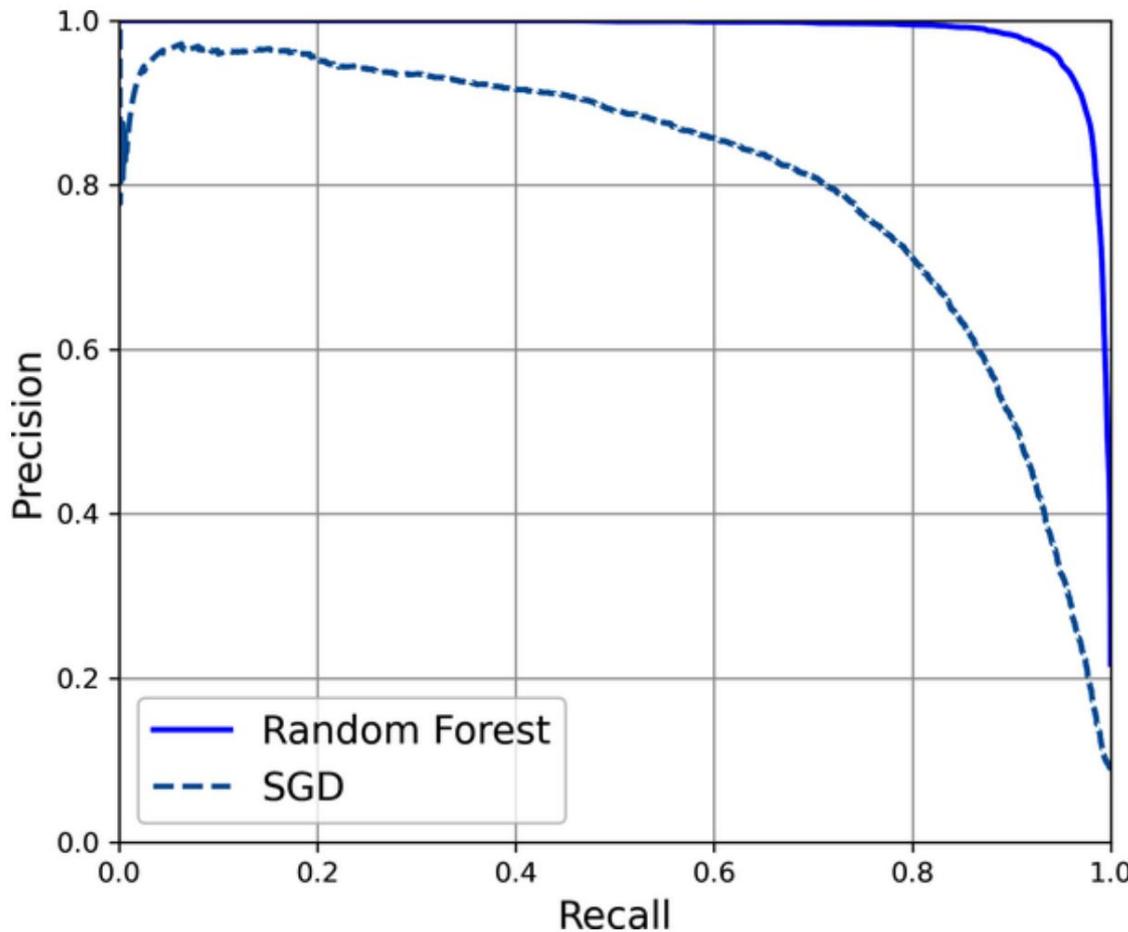


Figura 3-8. Comparación de curvas PR: el clasificador de bosque aleatorio es superior al clasificador SGD porque su curva PR está mucho más cerca de la esquina superior derecha y tiene un AUC mayor

Como puede ver en [la Figura 3-8](#), la curva PR de RandomForestClassifier se ve mucho mejor que la del SGDClassifier: se acerca mucho más a la esquina superior derecha. Su puntuación F y su puntuación ROC AUC también son significativamente mejores:

```
>>> y_train_pred_forest = y_probas_forest[:, 1] >= 0.5 # proba positiva ≥ 50%
>>> f1_score(y_train_5,
y_pred_forest) 0.9242275142688446

>>> roc_auc_score(y_train_5, y_scores_forest)
0.9983436731328145
```

Intente medir la precisión y las puntuaciones de recuperación: debería encontrar aproximadamente un 99,1 % de precisión y un 86,6 % de recuperación. ¡No está mal!

Ahora sabe cómo entrenar clasificadores binarios, elegir la métrica adecuada para su tarea, evaluar sus clasificadores mediante validación cruzada, seleccionar el equilibrio precisión/recuperación que se ajuste a sus necesidades y utilizar varias métricas y curvas para comparar varios modelos. Estás listo para intentar detectar algo más que los 5.

## Clasificación multiclas

Mientras que los clasificadores binarios distinguen entre dos clases, los clasificadores multiclas (también llamados clasificadores multinomiales) pueden distinguir entre más de dos clases.

Algunos clasificadores de Scikit-Learn (por ejemplo, LogisticRegression, RandomForestClassifier y GaussianNB) son capaces de manejar múltiples clases de forma nativa. Otros son clasificadores estrictamente binarios (por ejemplo, SGDClassifier y SVC). Sin embargo, existen varias estrategias que puede utilizar para realizar una clasificación multiclas con varios clasificadores binarios.

Una forma de crear un sistema que pueda clasificar las imágenes de dígitos en 10 clases (de 0 a 9) es entrenar 10 clasificadores binarios, uno para cada dígito (un detector de 0, un detector de 1, un detector de 2, etc.). Luego, cuando desee clasificar una imagen, obtendrá la puntuación de decisión de cada clasificador para esa imagen y seleccionará la clase cuyo clasificador genere la puntuación más alta.  
Esto se denomina estrategia uno contra el resto (OvR) o, a veces, uno contra todos (OvA).

Otra estrategia es entrenar un clasificador binario para cada par de dígitos: uno para distinguir 0 y 1, otro para distinguir 0 y 2, otro para 1 y 2, y así sucesivamente. Esto se llama estrategia uno contra uno (OvO). Si hay N clases, necesita entrenar  $N \times (N - 1) / 2$  clasificadores. Para el problema MNIST, esto significa entrenar 45 clasificadores binarios. Cuando quieras clasificar una imagen, debes pasar la imagen por los 45 clasificadores y ver qué clase gana más duelos. La principal ventaja de OvO es que cada clasificador sólo necesita ser entrenado en la parte del conjunto de entrenamiento que contiene las dos clases que debe distinguir.

Algunos algoritmos (como los clasificadores de máquinas de vectores de soporte) escalan mal con el tamaño del conjunto de entrenamiento. Para estos algoritmos OvO es

Se prefiere porque es más rápido entrenar muchos clasificadores en conjuntos de entrenamiento pequeños que entrenar pocos clasificadores en conjuntos de entrenamiento grandes. Sin embargo, para la mayoría de los algoritmos de clasificación binaria se prefiere OvR.

Scikit-Learn detecta cuando intenta utilizar un algoritmo de clasificación binaria para una tarea de clasificación multiclase y ejecuta automáticamente OvR u OvO, según el algoritmo. Probemos esto con un clasificador de máquina de vectores de soporte usando la clase `sklearn.svm.SVC` (consulte [el Capítulo 5](#)). Solo entrenaremos con las primeras 2000 imágenes o, de lo contrario, llevará mucho tiempo:

```
desde sklearn.svm importar SVC
```

```
svm_clf = SVC(random_state=42)
svm_clf.fit(X_train[:2000], y_train[:2000]) # y_train, no y_train_5
```

¡Eso fue fácil! Entrenamos el SVC usando las clases objetivo originales de 0 a 9 (`y_train`), en lugar de las clases objetivo 5 versus el resto (`y_train_5`). Dado que hay 10 clases (es decir, más de 2), Scikit-Learn utilizó la estrategia OvO y entrenó 45 clasificadores binarios. Ahora hagamos una predicción sobre una imagen:

```
>>> svm_clf.predict([algún_dígito]) matriz(['5'],
tipod=objeto)
```

¡Eso es correcto! En realidad, este código hizo 45 predicciones, una por par de clases, y seleccionó la clase que ganó más duelos. Si llama al método `decision_function()`, verá que devuelve 10 puntuaciones por instancia: una por clase. Cada clase obtiene una puntuación igual al número de duelos ganados más o menos un pequeño ajuste (máximo  $\pm 0,33$ ) para romper empates, según las puntuaciones del clasificador:

```
>>> some_digit_scores = svm_clf.decision_function([some_digit]) >>> some_digit_scores.round(2)
matriz([[ 3.79, 0.73, 6.06, 8.3 7.21,
, -0,29, 9,3 , 1,75, 2,77,
4.82]])
```

La puntuación más alta es 9,3, y efectivamente es la correspondiente a la clase 5:

```
>>> class_id = some_digit_scores.argmax() >>> class_id 5
```

Cuando se entrena un clasificador, almacena la lista de clases objetivo en su atributo `clases_`, ordenadas por valor. En el caso de MNIST, el índice de cada clase en la matriz `clases_` coincide convenientemente con la clase misma (por ejemplo, la clase en el índice 5 resulta ser la clase '5'), pero en general no tendrá tanta suerte; necesitarás buscar la etiqueta de clase de esta manera:

```
>>> svm_clf.classes_ array(['0',  
'1', '2', '3', '4', '5', '6', '7', '8', '9'], dtype=object) >>> svm_clf.classes_[class_id]
```

'5'

Si desea forzar a Scikit-Learn a usar uno contra uno o uno contra el resto, puede usar las clases `OneVsOneClassifier` o `OneVsRestClassifier`.

Simplemente cree una instancia y pase un clasificador a su constructor (ni siquiera tiene que ser un clasificador binario).

Por ejemplo, este código crea un clasificador multiclas utilizando la estrategia OvR, basado en un SVC:

```
desde sklearn.multiclass importar OneVsRestClassifier
```

```
ovr_clf = OneVsRestClassifier(SVC(random_state=42)) ovr_clf.fit(X_train[:2000],  
y_train[:2000])
```

Hagamos una predicción y verifiquemos la cantidad de clasificadores entrenados:

```
>>> ovr_clf.predict([algún_dígito]) array(['5'],  
dtype='<U1') >>> len(ovr_clf.estimators_)  
10
```

Entrenar un SGDClassifier en un conjunto de datos multiclas y usarlo para hacer predicciones es igual de fácil:

```
>>> sgd_clf = SGDClassifier(estado_aleatorio=42) >>>  
sgd_clf.fit(X_train, y_train)
```

```
>>> sgd_clf.predict([algún_dígito]) matriz(['3'],
   dtype=<U1')
```

Vaya, eso es incorrecto. ¡Los errores de predicción ocurren! Esta vez Scikit-Learn utilizó la estrategia OvR internamente: dado que hay 10 clases, entrenó 10 clasificadores binarios. El método `decision_function()` ahora devuelve un valor por clase. Veamos las puntuaciones que el clasificador SGD asignó a cada clase:

```
>>> sgd_clf.decision_function([algún_dígito]).round() matriz([-31893., -34420.,
   -9531., 1824., -22320., -1386., -26189., -16148., -4604., -12051.])
```

Puede ver que el clasificador no confía mucho en su predicción: casi todas las puntuaciones son muy negativas, mientras que la clase 3 tiene una puntuación de +1.824 y la clase 5 no se queda atrás con -1.386. Por supuesto, querrás evaluar este clasificador en más de una imagen. Dado que hay aproximadamente la misma cantidad de imágenes en cada clase, la métrica de precisión está bien. Como de costumbre, puedes usar la función `cross_val_score()` para evaluar el modelo:

```
>>> cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy") matriz([0.87365,
   0.85835, 0.8689])
```

Obtiene más del 85,8% en todos los pliegues de prueba. Si utilizara un clasificador aleatorio, obtendría un 10% de precisión, por lo que no es una puntuación tan mala, pero aún puede hacerlo mucho mejor. Simplemente escalar las entradas (como se analiza en [el Capítulo 2](#)) aumenta la precisión por encima del 89,1%:

```
>>> de sklearn.preprocessing importar StandardScaler >>> escalador =
StandardScaler()
>>> X_train_scaled =
scaler.fit_transform(X_train.astype("float64")) >>> cross_val_score(sgd_clf,
X_train_scaled, y_train, cv=3, scoring="accuracy") array([0.8983, 0.891
   , 0.9018])
```

## Análisis de errores

Si este fuera un proyecto real, ahora seguiría los pasos de la lista de verificación de su proyecto de aprendizaje automático (consulte el Apéndice A).

Exploraría las opciones de preparación de datos, probaría varios modelos, seleccionaría los mejores, ajustaría sus hiperparámetros utilizando GridSearchCV y automatizaría tanto como fuera posible. Aquí asumiremos que ha encontrado un modelo prometedor y desea encontrar formas de mejorarlo. Una forma de hacerlo es analizar los tipos de errores que comete.

Primero, observe la matriz de confusión. Para esto, primero necesita hacer predicciones usando la función `cross_val_predict()`; luego puedes pasar las etiquetas y predicciones a la función `confusion_matrix()`, tal como lo hiciste antes. Sin embargo, dado que ahora hay 10 clases en lugar de 2, la matriz de confusión contendrá bastantes números y puede resultar difícil de leer.

Un diagrama coloreado de la matriz de confusión es mucho más fácil de analizar.

Para trazar un diagrama de este

tipo, utilice la función `ConfusionMatrixDisplay.from_predictions()` como esta:

```
desde sklearn.metrics import ConfusionMatrixDisplay  
  
y_train_pred = cross_val_predict(sgd_clf, X_train_scaled, y_train, cv=3)  
  
ConfusionMatrixDisplay.from_predictions(y_train, y_train_pred).plt.show()
```

Esto produce el diagrama de la izquierda en la Figura 3-9. Esta matriz de confusión se ve bastante bien: la mayoría de las imágenes están en la diagonal principal, lo que significa que fueron clasificadas correctamente. Observe que la celda en la diagonal de la fila n.º 5 y la columna n.º 5 se ve un poco más oscura que los otros dígitos. Esto podría deberse a que el modelo cometió más errores en los 5 o a que hay menos 5 en el conjunto de datos que los otros dígitos. Por eso es importante normalizar la matriz de confusión dividiendo cada valor por el número total de imágenes en la clase correspondiente (verdadera) (es decir, dividiéndola por la suma de la fila). Esto se puede hacer simplemente configurando `normalize="true"`. También podemos especificar el argumento `value_format=".0%"` para mostrar porcentajes con

sin decimales. El siguiente código produce el diagrama de la derecha en la Figura 3-9:

```
ConfusionMatrixDisplay.from_predictions(y_train, y_train_pred,
                                         normalizar="verdadero",
                                         valores_formato=".0%")
plt.show()
```

Ahora podemos ver fácilmente que sólo el 82% de las imágenes de 5 se clasificaron correctamente. El error más común que cometió el modelo con imágenes de 5 fue clasificarlas erróneamente como 8: esto sucedió con el 10% de todos los 5. Pero sólo el 2% de los 8 fueron clasificados erróneamente como 5; ¡Las matrices de confusión generalmente no son simétricas! Si observa con atención, notará que muchos dígitos se han clasificado erróneamente como 8, pero esto no es inmediatamente obvio en este diagrama. Si desea que los errores se destaquen más, puede intentar poner cero peso en las predicciones correctas. El siguiente código hace precisamente eso y produce el diagrama de la izquierda en la Figura 3-10:

```
peso_muestra = (y_train_pred != y_train)
ConfusionMatrixDisplay.from_predictions(y_train, y_train_pred,
                                         peso_muestra=peso_muestra,
                                         normalizar="verdadero",
                                         valores_formato=".0%")
plt.show()
```

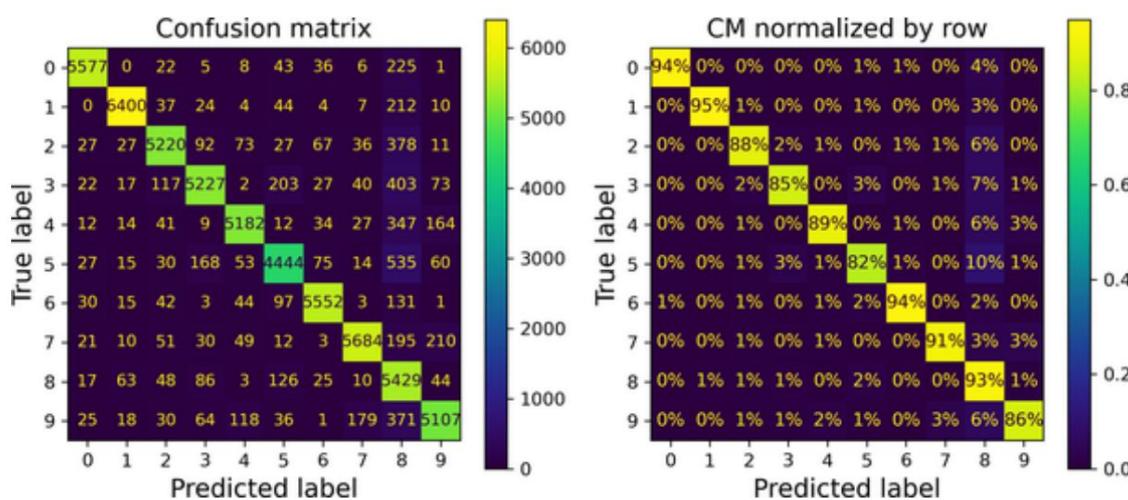


Figura 3-9. Matriz de confusión (izquierda) y el mismo CM normalizado por fila (derecha)

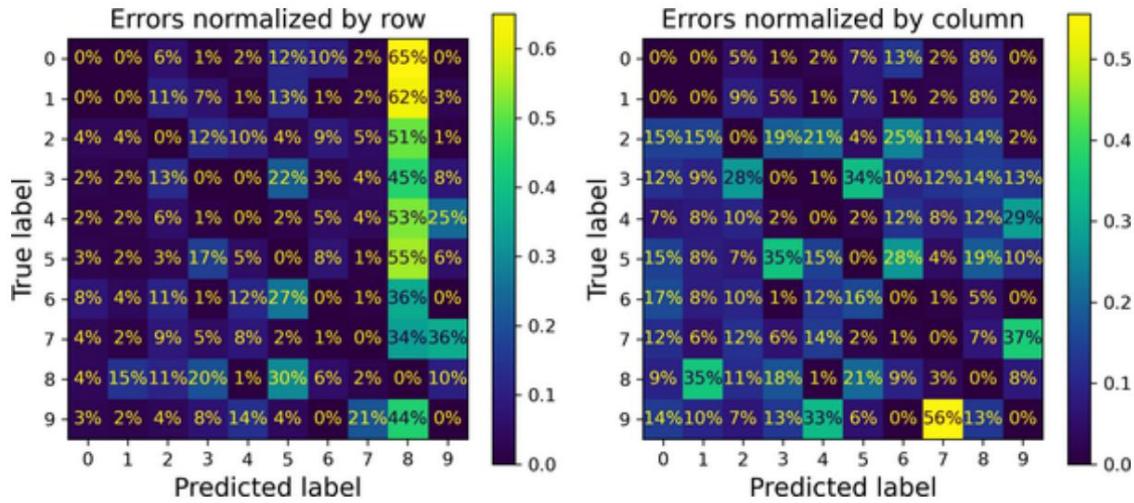


Figura 3-10. Matriz de confusión solo con errores, normalizada por fila (izquierda) y por columna (derecha)

Ahora puedes ver mucho más claramente los tipos de errores que comete el clasificador. La columna de la clase 8 ahora es muy brillante, lo que confirma que muchas imágenes se clasificaron erróneamente como 8. De hecho, esta es la clasificación errónea más común en casi todas las clases. Pero tenga cuidado al interpretar los porcentajes de este diagrama: recuerde que hemos excluido las predicciones correctas. Por ejemplo, el 36% en la fila 7, columna 9 no significa que el 36% de todas las imágenes de 7 se clasificaron erróneamente como 9. Significa que el 36% de los errores que cometió el modelo en imágenes de 7 fueron clasificaciones erróneas como 9. En realidad, sólo el 3% de las imágenes de 7 se clasificaron erróneamente como 9, como se puede ver en el diagrama de la derecha de la Figura 3-9.

También es posible normalizar la matriz de confusión por columna en lugar de por fila: si establece `normalize="pred"`, obtendrá el diagrama de la derecha en la Figura 3-10. Por ejemplo, puedes ver que el 56% de los 7 mal clasificados son en realidad 9.

El análisis de la matriz de confusión a menudo le brinda información sobre formas de mejorar su clasificador. Al observar estos gráficos, parece que sus esfuerzos deberían dedicarse a reducir los falsos 8. Por ejemplo, podría intentar recopilar más datos de entrenamiento para dígitos que parecen 8 (pero no lo son) para que el clasificador pueda aprender a distinguirlos de los 8 reales. O podría diseñar nuevas funciones que ayudarían al clasificador; por ejemplo, escribir un algoritmo para contar el número de bucles cerrados (por ejemplo, 8 tiene dos, 6 tiene uno, 5 no tiene ninguno). O podría preprocesar las imágenes (por ejemplo, usando Scikit-

Image, Pillow u OpenCV) para hacer que algunos patrones, como los bucles cerrados, se destaquen más.

Analizar errores individuales también puede ser una buena manera de obtener información sobre lo que hace su clasificador y por qué falla. Por ejemplo, tracemos ejemplos de 3 y 5 en un estilo de matriz de confusión ([Figura 3-11](#)):

```

cl_a, cl_b = '3', '5'
X_aa = X_train[(y_train == cl_a) & (y_train_pred == cl_a)]
X_ab = X_tren[(y_tren == cl_a) & (y_tren_pred == cl_b)]
X_ba = X_tren[(y_tren == cl_b) & (y_tren_pred == cl_a)]
X_bb = X_train[(y_train == cl_b) & (y_train_pred == cl_b)] [...] # traza todas las
imágenes en X_aa, X_ab, X_ba, X_bb en un estilo de matriz de confusión

```

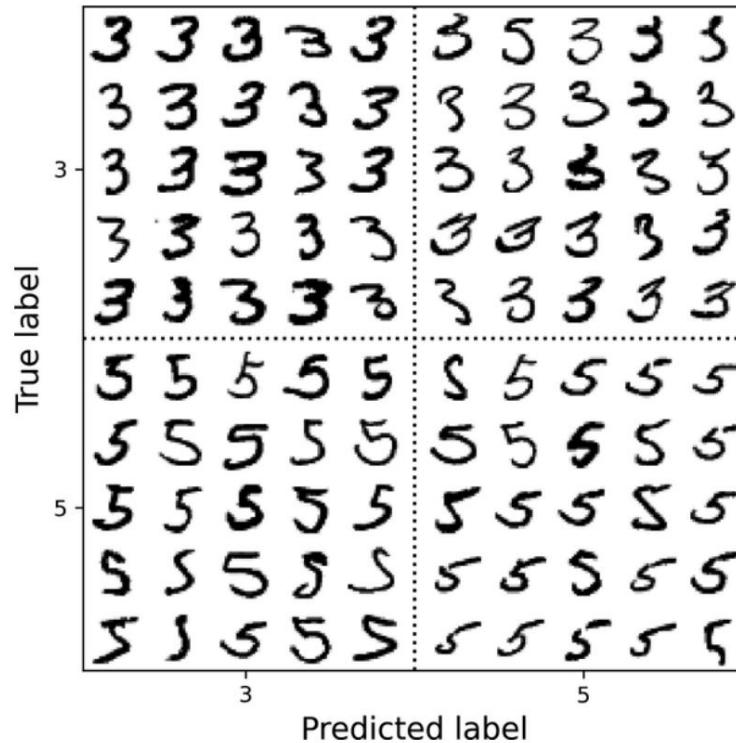


Figura 3-11. Algunas imágenes de 3 y 5 organizadas como una matriz de confusión.

Como puede ver, algunos de los dígitos en los que el clasificador se equivoca (es decir, en los bloques inferior izquierdo y superior derecho) están tan mal escritos que incluso un humano tendría problemas para clasificarlos. Sin embargo, la mayoría de las imágenes mal clasificadas nos parecen errores obvios. Puede ser difícil entender por qué el clasificador cometió los errores que cometió, pero recuerde que el cerebro humano es un fantástico sistema de reconocimiento de patrones, y nuestro sistema visual también lo hace.

mucho preprocessamiento complejo antes de que cualquier información llegue siquiera a nuestra conciencia. Entonces, el hecho de que esta tarea parezca simple no significa que lo sea. Recuerde que usamos un SGDClassifier simple, que es solo un modelo lineal: todo lo que hace es asignar un peso por clase a cada píxel, y cuando ve una nueva imagen simplemente suma las intensidades ponderadas de los píxeles para obtener una puntuación para cada clase. . Dado que los 3 y 5 difieren sólo en unos pocos píxeles, este modelo los confundirá fácilmente.

La principal diferencia entre 3 y 5 es la posición de la pequeña línea que une la línea superior con el arco inferior. Si dibuja un 3 con el cruce ligeramente desplazado hacia la izquierda, el clasificador podría clasificarlo como un 5 y viceversa. En otras palabras, este clasificador es bastante sensible al cambio y la rotación de imágenes. Una forma de reducir la confusión 3/5 es preprocessar las imágenes para asegurarse de que estén bien centradas y no demasiado rotadas. Sin embargo, esto puede no ser fácil ya que requiere predecir la rotación correcta de cada imagen. Un enfoque mucho más simple consiste en aumentar el conjunto de entrenamiento con variantes de las imágenes de entrenamiento ligeramente desplazadas y rotadas. Esto obligará al modelo a aprender a ser más tolerante a tales variaciones. Esto se llama aumento de datos (lo cubriremos en [el Capítulo 14](#); consulte también el ejercicio 2 al final de este capítulo).

## Clasificación multietiqueta

Hasta ahora, cada instancia siempre ha sido asignada a una sola clase. Pero en algunos casos es posible que desee que su clasificador genere múltiples clases para cada instancia. Consideremos un clasificador de reconocimiento facial: ¿qué debería hacer si reconoce a varias personas en la misma imagen? Debe adjuntar una etiqueta por persona que reconozca. Digamos que el clasificador ha sido entrenado para reconocer tres caras: Alice, Bob y Charlie. Luego, cuando al clasificador se le muestra una imagen de Alice y Charlie, debería generar [Verdadero, Falso, Verdadero] (que significa "Alice sí, Bob no, Charlie sí"). Un sistema de clasificación de este tipo que genera múltiples etiquetas binarias se denomina sistema de clasificación de etiquetas múltiples .

No entraremos en el reconocimiento facial todavía, pero veamos un ejemplo más simple, sólo con fines ilustrativos:

```

importar numpy como np
desde sklearn.neighbors importar KNeighborsClassifier

y_train_large = (y_train >= '7') y_train_odd =
(y_train.astype('int8') % 2 == 1) y_multilabel = np.c_[y_train_large,
y_train_odd]

knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train, y_multilabel)

```

Este código crea una matriz `y_multilabel` que contiene dos etiquetas de destino para cada imagen de dígito: la primera indica si el dígito es grande o no (7, 8 o 9) y la segunda indica si es impar o no. Luego, el código crea una instancia de `KNeighborsClassifier`, que admite la clasificación de etiquetas múltiples (no todos los clasificadores lo hacen) y entrena este modelo utilizando la matriz de múltiples objetivos. Ahora puede hacer una predicción y observar que genera dos etiquetas:

```
>>> knn_clf.predict([algún_dígito]) matriz([[Falso,
Verdadero]])
```

¡Y lo hace bien! De hecho, el dígito 5 no es grande (Falso) ni impar (Verdadero).

Hay muchas formas de evaluar un clasificador de etiquetas múltiples y seleccionar la métrica correcta realmente depende de su proyecto. Un enfoque es medir la puntuación F para cada etiqueta individual (o cualquier otra métrica de clasificador binario analizada anteriormente) y luego simplemente calcular la puntuación promedio. El siguiente código calcula la puntuación F promedio en todas las etiquetas:

```
>>> y_train_knn_pred = cross_val_predict(knn_clf, X_train, y_multilabel, cv=3) >>>
f1_score(y_multilabel,
y_train_knn_pred, Average="macro") 0.976410265560605
```

Este enfoque supone que todas las etiquetas son igualmente importantes, lo que puede no ser el caso. En particular, si tiene muchas más imágenes de Alice que de Bob o Charlie, es posible que desee darle más peso a la puntuación del clasificador en las imágenes de Alice. Una opción simple es darle a cada etiqueta un peso igual a su soporte (es decir, el número de instancias con esa

etiqueta de destino). Para hacer esto, simplemente establezca promedio="ponderado" al llamar a la función f1\_score(). 5

Si desea utilizar un clasificador que no admite de forma nativa la clasificación de etiquetas múltiples, como SVC, una estrategia posible es entrenar un modelo por etiqueta. Sin embargo, esta estrategia puede tener dificultades para capturar las dependencias entre las etiquetas. Por ejemplo, un dígito grande (7, 8 o 9) tiene el doble de probabilidades de ser impar que par, pero el clasificador de la etiqueta "impar" no sabe lo que predijo el clasificador de la etiqueta "grande". Para resolver este problema, los modelos se pueden organizar en una cadena: cuando un modelo hace una predicción, utiliza las características de entrada más todas las predicciones de los modelos que le preceden en la cadena.

¡La buena noticia es que Scikit-Learn tiene una clase llamada ChainClassifier que hace precisamente eso! De forma predeterminada, utilizará las etiquetas verdaderas para el entrenamiento y alimentará a cada modelo con las etiquetas apropiadas según su posición en la cadena. Pero si configura el hiperparámetro cv, utilizará validación cruzada para obtener predicciones "limpias" (fuera de muestra) de cada modelo entrenado para cada instancia del conjunto de entrenamiento, y estas predicciones luego se usarán para entrenar todos los modelos posteriores en la cadena. A continuación se muestra un ejemplo que muestra cómo crear y entrenar un ChainClassifier utilizando la estrategia de validación cruzada. Como antes, solo usaremos las primeras 2000 imágenes en el conjunto de entrenamiento para acelerar las cosas:

```
desde sklearn.multioutput importar ClassifierChain

chain_clf = ClassifierChain(SVC(), cv=3, random_state=42) chain_clf.fit(X_train[:2000],
y_multilabel[:2000])
```

Ahora podemos usar este ChainClassifier para hacer predicciones:

```
>>> chain_clf.predict([algún_dígito]) matriz([[0., 1.]])
```

## Clasificación multisalida

El último tipo de tarea de clasificación que discutiremos aquí se llama clasificación multisalida-moniclase (o simplemente clasificación multisalida). Es un

Generalización de la clasificación de etiquetas múltiples donde cada etiqueta puede ser multiclase (es decir, puede tener más de dos valores posibles).

Para ilustrar esto, construyamos un sistema que elimine el ruido de las imágenes. Tomará como entrada una imagen de dígitos ruidosa y (con suerte) generará una imagen de dígitos limpia, representada como una matriz de intensidades de píxeles, al igual que las imágenes MNIST. Observe que la salida del clasificador es multietiqueta (una etiqueta por píxel) y cada etiqueta puede tener múltiples valores (la intensidad de los píxeles varía de 0 a 255). Éste es, por tanto, un ejemplo de un sistema de clasificación de múltiples resultados.

### NOTA

La línea entre clasificación y regresión a veces es borrosa, como en este ejemplo. Podría decirse que predecir la intensidad de los píxeles se parece más a una regresión que a una clasificación. Además, los sistemas de múltiples salidas no se limitan a tareas de clasificación; incluso podría tener un sistema que genere múltiples etiquetas por instancia, incluidas etiquetas de clase y etiquetas de valor.

Comencemos creando los conjuntos de entrenamiento y prueba tomando las imágenes MNIST y agregando ruido a sus intensidades de píxeles con la función randint() de NumPy. Las imágenes de destino serán las imágenes originales:

```
np.random.seed(42) # para hacer que este ejemplo de código sea reproducible
ruido = np.random.randint(0, 100, (len(X_train), 784))
X_train_mod = X_train + ruido
ruido =
np.random.randint(0, 100, (len(X_test), 784))
X_test_mod = X_test + ruido
y_train_mod = y_train
y_test_mod = y_test
```

Echemos un vistazo a la primera imagen del conjunto de prueba ([Figura 3-12](#)). Sí, estamos husmeando en los datos de las pruebas, por lo que deberías estar frunciendo el ceño ahora mismo.

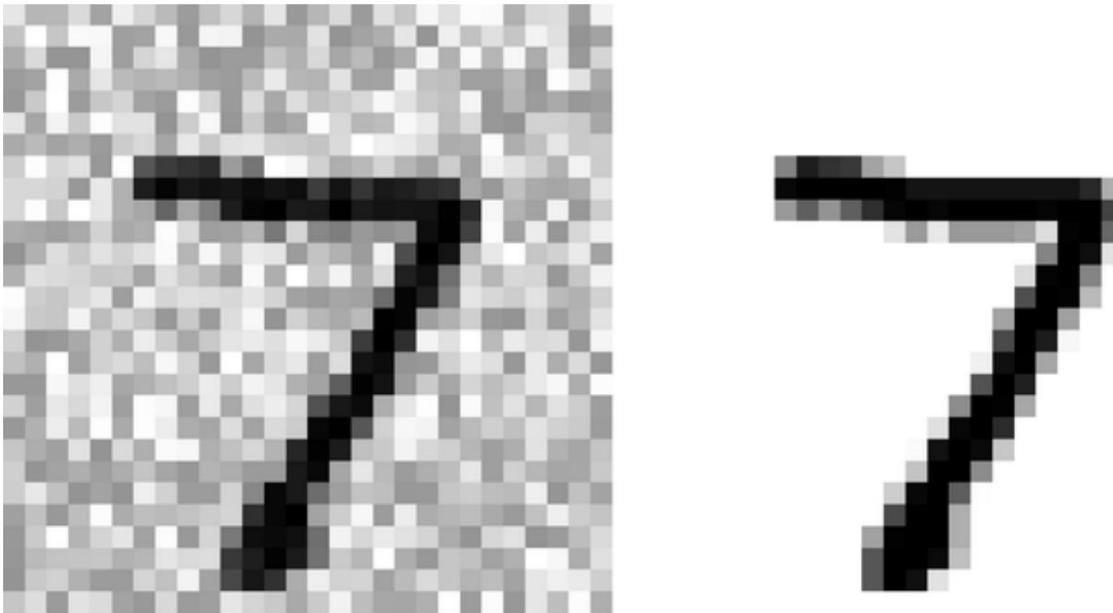


Figura 3-12. Una imagen ruidosa (izquierda) y la imagen limpia del objetivo (derecha)

A la izquierda está la imagen de entrada ruidosa y a la derecha está la imagen de destino limpia. Ahora entrenemos al clasificador y hagamos que limpie esta imagen ([Figura 3-13](#)):

```
knn_clf = KNeighborsClassifier()  
knn_clf.fit(X_train_mod, y_train_mod) clean_digit =  
knn_clf.predict([X_test_mod[0]]) plot_digit(clean_digit) plt.show()
```

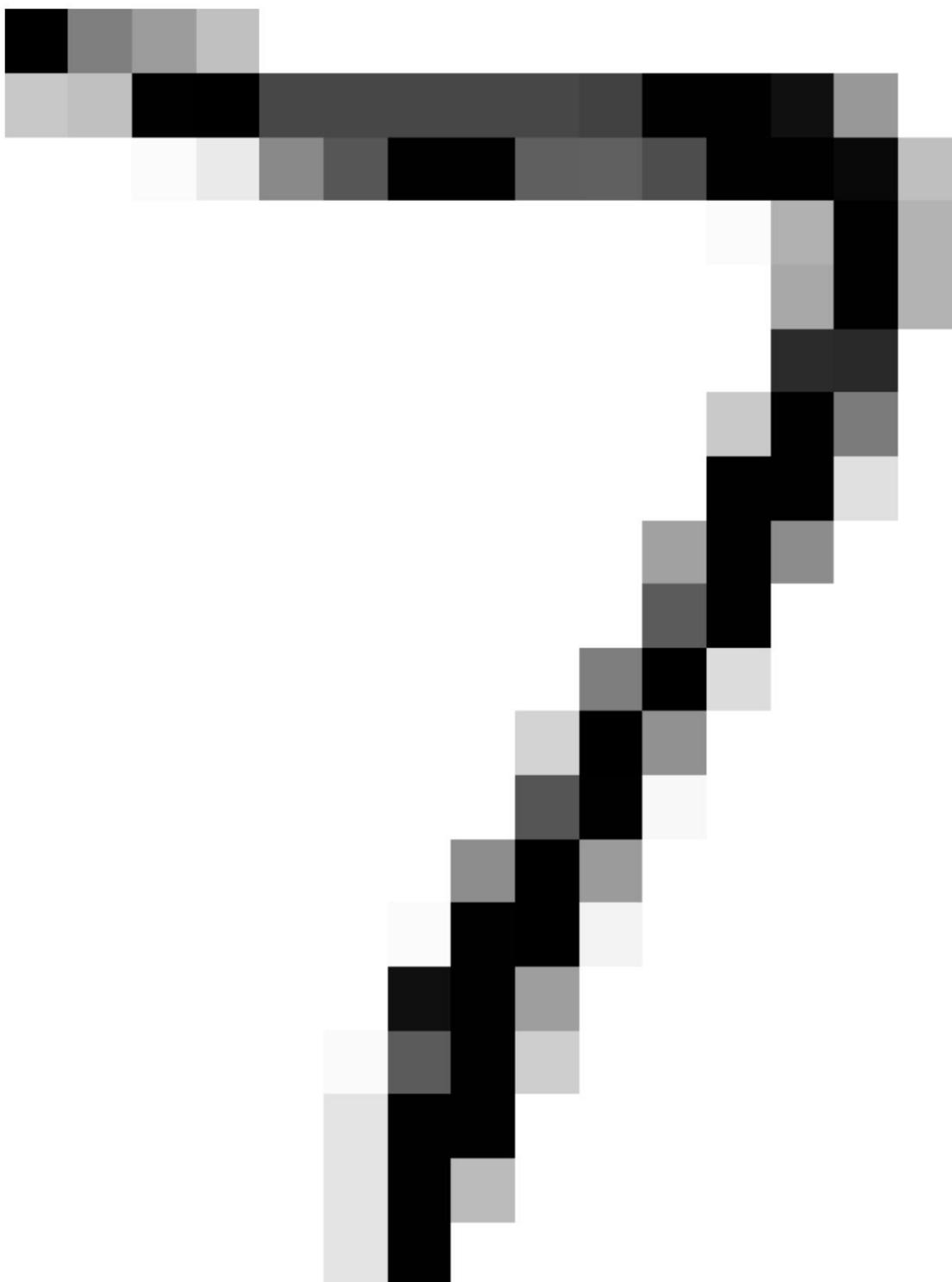


Figura 3-13. La imagen limpia

¡Parece lo suficientemente cerca del objetivo! Con esto concluye nuestro recorrido por la clasificación. Ahora sabe cómo seleccionar buenas métricas para las tareas de clasificación, elegir la compensación adecuada entre precisión y recuperación, comparar clasificadores,

y, en términos más generales, construir buenos sistemas de clasificación para una variedad de tareas. En los próximos capítulos, aprenderá cómo funcionan realmente todos estos modelos de aprendizaje automático que ha estado utilizando.

## Ejercicios

1. Intente crear un clasificador para el conjunto de datos MNIST que logre más del 97% de precisión en el conjunto de prueba. Sugerencia: KNeighborsClassifier funciona bastante bien para esta tarea; solo necesita encontrar buenos valores de hiperparámetros (pruebe una búsqueda en cuadrícula en los hiperparámetros pesos y n\_vecinos).
2. Escriba una función que pueda desplazar una imagen MNIST en cualquier dirección (izquierda, derecha, arriba o abajo) en un píxel. Luego,<sup>6</sup> para cada imagen en el conjunto de entrenamiento, cree cuatro copias desplazadas (una por dirección) y agréguelas al conjunto de entrenamiento. Finalmente, entrene su mejor modelo en este conjunto de entrenamiento ampliado y mida su precisión en el conjunto de prueba. ¡Deberías observar que tu modelo funciona aún mejor ahora! Esta técnica de hacer crecer artificialmente el conjunto de entrenamiento se llama aumento de datos o expansión del conjunto de entrenamiento.
3. Aborde el conjunto de datos del Titanic. Un excelente lugar para comenzar es [Kaggle](#). Alternativamente, puede descargar los datos desde <https://homl.info/titanic.tgz> y descomprima este tarball como lo hizo con los datos de vivienda en el Capítulo 2. Esto le dará dos archivos CSV, train.csv y test.csv, que puede cargar usando pandas.read\_csv(). El objetivo es entrenar un clasificador que pueda predecir la columna Sobrevivido en función de las otras columnas.
4. Cree un clasificador de spam (un ejercicio más desafiante):
  - a. Descargue ejemplos de spam y jamón de los [conjuntos de datos públicos de Apache SpamAssassin](#).
  - b. Descomprima los conjuntos de datos y familiarícese con el formato de los datos.
  - C. Divida los datos en un conjunto de entrenamiento y un conjunto de prueba.

d. Escriba un proceso de preparación de datos para convertir cada correo electrónico en un vector de características. Su proceso de preparación debe transformar un correo electrónico en un vector (escaso) que indique la presencia o ausencia de cada palabra posible. Por ejemplo, si todos los correos electrónicos solo contienen cuatro palabras, "Hola", "cómo", "eres", "tú", entonces el correo electrónico "Hola, hola, hola" se convertiría en un vector [1, 0, 0, 1] (que significa ["Hola" está presente, "cómo" está ausente, "estás" está ausente, "tú" está presente]), o [3, 0, 0, 2] si prefieres contar el número de apariciones de cada palabra.

Es posible que desee agregar hiperparámetros a su canal de preparación para controlar si eliminar o no los encabezados de los correos electrónicos, convertir cada correo electrónico a minúsculas, eliminar la puntuación, reemplazar todas las URL con "URL", reemplazar todos los números con "NÚMERO" o incluso realizar derivaciones . (es decir, recortar las terminaciones de palabras; hay bibliotecas de Python disponibles para hacer esto).

mi. Finalmente, pruebe varios clasificadores y vea si puede crear un excelente clasificador de spam, con alta recuperación y alta precisión.

Las soluciones a estos ejercicios están disponibles al final del cuaderno de este capítulo, en <https://homl.info/colab3>.

---

<sup>1</sup> De forma predeterminada, Scikit-Learn almacena en caché los conjuntos de datos descargados en un directorio llamado scikit\_learn\_data en su directorio de inicio.

<sup>2</sup> Los conjuntos de datos devueltos por fetch\_openml() no siempre se mezclan ni dividen.

<sup>3</sup> Mezclar datos puede ser una mala idea en algunos contextos; por ejemplo, si se trabaja con datos de series temporales (como los precios del mercado de valores o las condiciones climáticas). Exploraremos esto en [el Capítulo 15](#).

<sup>4</sup> Los clasificadores de Scikit-Learn siempre tienen un método decision\_function() o un método predict\_proba() o, a veces, ambos.

<sup>5</sup> Scikit-Learn ofrece algunas otras opciones de promedio y métricas de clasificador multietiqueta; consulte la documentación para obtener más detalles.

<sup>6</sup> Puede utilizar la función shift() del módulo scipy.ndimage.interpolation. Por ejemplo, shift(image, [2, 1], cval=0) desplaza la imagen dos píxeles hacia abajo y un píxel hacia la derecha.

## Capítulo 4. Modelos de entrenamiento

---

Hasta ahora hemos tratado los modelos de aprendizaje automático y sus algoritmos de entrenamiento principalmente como cajas negras. Si realizó algunos de los ejercicios de los capítulos anteriores, es posible que se haya sorprendido de lo mucho que puede hacer sin saber nada sobre lo que hay debajo del capó: optimizó un sistema de regresión, mejoró un clasificador de imágenes de dígitos e incluso creó un clasificador de spam desde cero, todo sin saber cómo funciona realmente. De hecho, en muchas situaciones no es necesario conocer los detalles de implementación.

Sin embargo, tener una buena comprensión de cómo funcionan las cosas puede ayudarle a encontrar rápidamente el modelo adecuado, el algoritmo de entrenamiento adecuado a utilizar y un buen conjunto de hiperparámetros para su tarea. Comprender lo que hay debajo del capó también lo ayudará a depurar problemas y realizar análisis de errores de manera más eficiente. Por último, la mayoría de los temas discutidos en este capítulo serán esenciales para comprender, construir y entrenar redes neuronales (que se analizan en la [Parte II](#) de este libro).

En este capítulo comenzaremos analizando el modelo de regresión lineal, uno de los modelos más simples que existen. Discutiremos dos formas muy diferentes de entrenarlo:

- Usar una ecuación de “forma cerrada” <sup>1</sup> que calcula directamente los parámetros del modelo que mejor se ajustan al conjunto de entrenamiento (es decir, los parámetros del modelo que minimizan la función de costo sobre el conjunto de entrenamiento).
- Utilizando un enfoque de optimización iterativo llamado descenso de gradiente (GD) que modifica gradualmente los parámetros del modelo para minimizar la función de costo en el conjunto de entrenamiento, y eventualmente convergiendo al mismo conjunto de parámetros que el primer método. Veremos algunas variantes del descenso de gradiente que usaremos una y otra vez cuando estudiemos redes neuronales en la [Parte II](#): GD por lotes, GD por mini lotes y GD estocástico.

A continuación veremos la regresión polinómica, un modelo más complejo que puede ajustarse a conjuntos de datos no lineales. Dado que este modelo tiene más parámetros que la regresión lineal, es más propenso a sobreajustar los datos de entrenamiento. Exploraremos cómo detectar si este es el caso utilizando curvas de aprendizaje y luego veremos varias técnicas de regularización que pueden reducir el riesgo de sobreajuste del conjunto de entrenamiento.

Finalmente, examinaremos dos modelos más que se usan comúnmente para tareas de clasificación: regresión logística y regresión softmax.

### ADVERTENCIA

En este capítulo habrá bastantes ecuaciones matemáticas, utilizando nociones básicas de álgebra lineal y cálculo. Para comprender estas ecuaciones, necesitarás saber qué son los vectores y las matrices; cómo transponerlos, multiplicarlos e invertirlos; y qué son las derivadas parciales. Si no está familiarizado con estos conceptos, consulte los tutoriales introductorios de álgebra lineal y cálculo disponibles como cuadernos de Jupyter en el material complementario [en línea](#). Para aquellos que son verdaderamente alérgicos a las matemáticas, aún así deberían leer este capítulo y simplemente saltarse las ecuaciones; Con suerte, el texto será suficiente para ayudarle a comprender la mayoría de los conceptos.

## Regresión lineal

En el [Capítulo 1](#) analizamos un modelo de regresión simple de satisfacción con la vida:

$$\text{satisfacción\_vida} = \theta_0 + \theta_1 \text{PIB_per_cápita}$$

Este modelo es solo una función lineal de la característica de entrada `GDP_per_capita`.  $\theta_0$  y  $\theta_1$  son los parámetros del modelo.

De manera más general, un modelo lineal hace una predicción simplemente calculando una suma ponderada de las características de entrada, más una constante llamada término de sesgo (también llamado término de intersección), como se muestra en la [Ecuación 4-1](#).

Ecuación 4-1. Predicción del modelo de regresión lineal.

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n$$

En esta ecuación:

- $\hat{y}$  es el valor previsto.
- $n$  es el número de características.
- $x_i$  es el valor de la característica  $i$ .
- $\theta_j$  es el parámetro del modelo  $j$ , incluido el término de sesgo  $\theta_0$  y los pesos de las características  $j$   $\theta_1, \theta_2, \dots, \theta_n$ .

Esto se puede escribir de manera mucho más concisa usando una forma vectorizada, como se muestra en la [Ecuación 4-2](#).

Ecuación 4-2. Predicción del modelo de regresión lineal (forma vectorizada)

$$\hat{y} = h(x) = \theta^T x$$

En esta ecuación:

- $h$  es la función de hipótesis, utilizando los parámetros del modelo  $\theta$ .

- $\theta$  es el vector de parámetros del modelo, que contiene el término de sesgo  $\theta_0$  y los pesos de característica  $\theta$  a  $\theta_n$ .
- $x$  es el vector de características de la instancia, que contiene  $x$  a  $x_n$ , siendo  $x_0$  siempre igual a 1.
- $\theta \cdot x$  es el producto escalar de los vectores  $\theta$  y  $x$ , que es igual a  $\theta_0 x_0 + \theta_1 x_1 + \dots + \theta_n x_n$ .

#### NOTA

En el aprendizaje automático, los vectores suelen representarse como vectores de columna, que son matrices 2D con una sola columna. Si  $\theta$  y  $x$  son vectores columna, entonces la predicción es  $\hat{y} = \theta^T x$ , donde  $\theta^T$  es la transpuesta de  $\theta$  (un vector fila en lugar de un vector columna) y  $\theta^T x$  es la multiplicación matricial de  $\theta^T$  y  $x$ . Por supuesto, es la misma predicción, excepto que ahora se representa como una matriz unicelular en lugar de un valor escalar. En este libro usaré esta notación para evitar cambiar entre productos escalares y multiplicaciones de matrices.

Bien, ese es el modelo de regresión lineal, pero ¿cómo lo entrenamos? Bueno, recuerde que entrenar un modelo significa configurar sus parámetros para que el modelo se ajuste mejor al conjunto de entrenamiento. Para este propósito, primero necesitamos una medida de qué tan bien (o mal) se ajusta el modelo a los datos de entrenamiento. En [el Capítulo 2](#) vimos que la medida de desempeño más común de un modelo de regresión es la raíz del error cuadrático medio ([Ecuación 2-1](#)). Por lo tanto, para entrenar un modelo de regresión lineal, necesitamos encontrar el valor de  $\theta$  que minimice el RMSE. En la práctica, es más sencillo minimizar el error cuadrático medio (MSE) que el RMSE, y conduce al mismo resultado (porque el valor que minimiza una función positiva también minimiza su raíz cuadrada).

#### ADVERTENCIA

Los algoritmos de aprendizaje a menudo optimizarán una función de pérdida diferente durante el entrenamiento que la medida de rendimiento utilizada para evaluar el modelo final. Esto generalmente se debe a que la función es más fácil de optimizar y/o porque tiene términos adicionales necesarios solo durante el entrenamiento (por ejemplo, para la regularización). Una buena métrica de desempeño es lo más cercana posible al objetivo comercial final. Una buena pérdida de entrenamiento es fácil de optimizar y está fuertemente correlacionada con la métrica. Por ejemplo, los clasificadores a menudo se entrena usando una función de costo como la pérdida logarítmica (como verá más adelante en este capítulo), pero se evalúan usando precisión/recuperación. La pérdida de registros es fácil de minimizar y, al hacerlo, normalmente mejorará la precisión/recuperación.

El MSE de una hipótesis de regresión lineal  $h$  en un conjunto de entrenamiento  $X$  se calcula utilizando [la Ecuación 4-3](#).

Ecuación 4-3. Función de costo MSE para un modelo de regresión lineal

$$\text{MSE}(X, h) = \frac{1}{\text{metro}} \sum_{i=1}^m (x(i) - y(i))^2$$

La mayoría de estas notaciones se presentaron en el Capítulo 2 (ver "Notaciones"). La única diferencia es que escribimos  $h$  en lugar de solo  $h$  para dejar claro que el modelo está parametrizado por el vector  $\theta$ . Para simplificar las notaciones, simplemente escribiremos  $\text{MSE}(\theta)$  en lugar de  $\text{MSE}(X, h)$ .

## La ecuación normal

Para encontrar el valor de  $\theta$  que minimiza el MSE, existe una solución de forma cerrada; en otras palabras, una ecuación matemática que da el resultado directamente. Esto se llama ecuación normal ([Ecuación 4-4](#)).

Ecuación 4-4. ecuación normal

$$\hat{\theta} = (X^T X)^{-1} X^T y$$

En esta ecuación:

- $\hat{\theta}$  es el valor de  $\theta$  que minimiza la función de costos.
- $(m) y$  es el vector de valores objetivo que contiene  $y_1, y_2, \dots, y_m$ .

Generemos algunos datos de apariencia lineal para probar esta ecuación ([Figura 4-1](#)):

```
importar numpy como np
```

```
np.random.seed(42) # para hacer reproducible este ejemplo de código m = 100 # número de instancias # vector de columna X = 2 * np.random.rand(m, 1)
y = 4 + 3 * X + np.random.randn(m, 1) # np.random.rand(m, 1) y vector de columna
```

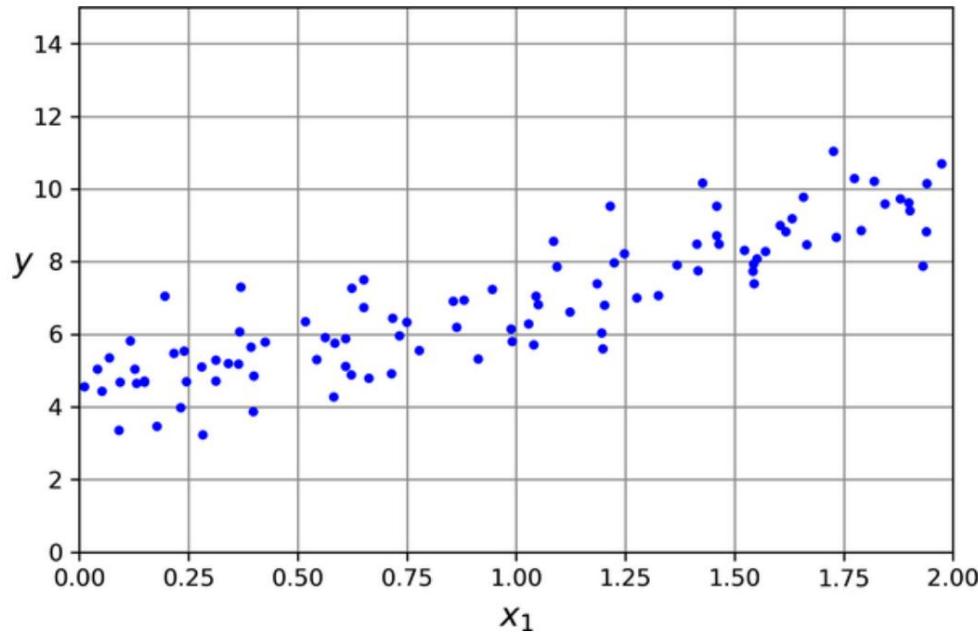


Figura 4-1. Un conjunto de datos lineal generado aleatoriamente

Ahora calculemos  $\hat{y}$  usando la ecuación normal. Usaremos la función `inv()` de el módulo de álgebra lineal de NumPy (`np.linalg`) para calcular la inversa de una matriz y el método `dot()` para la multiplicación de matrices:

```
desde sklearn.preprocessing importar add_dummy_feature
X_b = add_dummy_feature(X) # agrega x0 = 1 a cada instancia
theta_best = np.linalg.inv(X_b.T @ X_b) @ X_b.T @ y
```

#### NOTA

El operador `@` realiza la multiplicación de matrices. Si A y B son matrices NumPy, entonces `A @ B` es equivalente a `np.matmul(A, B)`. Muchas otras bibliotecas, como TensorFlow, PyTorch y JAX, también admiten el operador `@`. Sin embargo, no puede utilizar `@` en matrices de Python puras (es decir, listas de listas).

La función que utilizamos para generar los datos es  $y = 4 + 3x + \text{ruido gaussiano}$ . Veamos qué encontró la ecuación:

```
>>> matriz theta_best
([[4.21509616],
 [2.77011339]])
```

Habríamos esperado  $\theta_0 = 4$  y  $\theta_1 = 3$  en lugar de  $\theta_0 = 4,215$  y  $\theta_1 = 2,770$ .  
Lo suficientemente cerca, pero el ruido hizo imposible recuperar los parámetros exactos de

la función original. Cuanto más pequeño y ruidoso sea el conjunto de datos, más difícil se vuelve.

Ahora podemos hacer predicciones usando  $\hat{\theta}$ :

```
>>> X_nuevo = np.array([[0], [2]])
>>> X_new_b = add_dummy_feature(X_new) # agrega x0 = 1 a cada instancia >>> y_predict = X_new_b @
theta_best >>> y_predict array([4.21509616,
[9.75532293]])
```

Tracemos las predicciones de este modelo (Figura 4-2):

```
importar matplotlib.pyplot como plt
plt.plot(X_new, y_predict, "r-", label="Predicciones") plt.plot(X, y, "b.") [...] # embellecer
la figura: agregar etiquetas, ejes,
cuadrículas y leyenda plt.show()
```

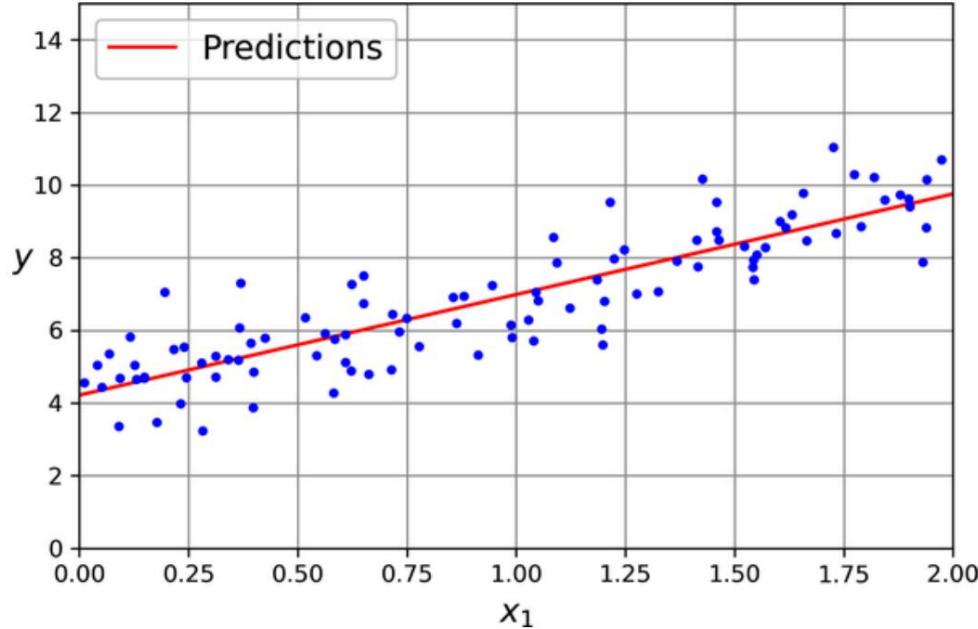


Figura 4-2. Predicciones del modelo de regresión lineal

Realizar una regresión lineal utilizando Scikit-Learn es relativamente sencillo:

```
>>> de sklearn.linear_model import LinearRegression >>> lin_reg = LinearRegression()
>>> lin_reg.fit(X, y) >>> lin_reg.intercept_,
lin_reg.coef_ (matriz([4.21509616]),
matriz([[ 2.77011339]])) >>> lin_reg.predict(X_nuevo)
```

```
matriz([[4.21509616],
       [9.75532293]])
```

Observe que Scikit-Learn separa el término de sesgo (`intercept_`) de los pesos de las características (`coef_`). La clase `LinearRegression` se basa en la función `scipy.linalg.lstsq()` (el nombre significa “mínimos cuadrados”), a la que puedes llamar directamente:

```
>>> theta_best_svd, residuales, rango, s = np.linalg.lstsq(X_b, y, rcond=1e- 6) >>> theta_best_svd array([[4.21509616],
       [2.77011339]])
```

Esta función calcula la  $\hat{y} = X\hat{y}$ , donde  $X\hat{y}$  es la pseudoinversa de  $X$  (específicamente, inversa de Moore-Penrose). Puedes usar `np.linalg.pinv()` para calcular el pseudoinverso directamente:

```
>>> np.linalg.pinv(X_b) @ y
matriz([[4.21509616],
       [2.77011339]])
```

El pseudoinverso en sí se calcula utilizando una técnica estándar de factorización matricial llamada descomposición de valores singulares (SVD) que puede descomponer la matriz  $X$  del conjunto de entrenamiento en la multiplicación matricial de tres matrices  $U \Sigma V$  (ver `numpy.linalg.svd()`).

La pseudoinversa se calcula como  $X\hat{y} = V\Sigma^{-1}U^T$ . Para calcular la matriz  $\Sigma^{-1}$ , el algoritmo toma  $\Sigma$  y pone a cero todos los valores menores que un pequeño valor umbral, luego reemplaza todos los valores distintos de cero con su inverso y finalmente transpone la matriz resultante. Este enfoque es más eficiente que calcular la ecuación normal, además maneja bien los casos extremos: de hecho, la ecuación normal puede no funcionar si la matriz  $XX^T$  no es invertible (es decir, singular), como si  $m < n$  o si algunas características son redundante, pero la pseudoinversa siempre está definida.

## Complejidad computacional

La ecuación normal calcula la inversa de  $XX^T$ , que es una matriz  $(n + 1) \times (n + 1)$  (donde  $n$  es el número de características). La complejidad computacional de invertir una matriz de este tipo suele ser de  $O(n^3)$  a  $O(n^4)$ , dependiendo de la implementación. En otras palabras, si duplica la cantidad de funciones, multiplica el tiempo de cálculo aproximadamente por 2 = 5,3 a 2 = 8.

2.4

3

El enfoque SVD utilizado por la clase `LinearRegression` de Scikit-Learn es aproximadamente  $O(\sqrt{n}^2)$ . Si duplica la cantidad de funciones, multiplica el tiempo de cálculo por aproximadamente 4.

**ADVERTENCIA**

Tanto la ecuación normal como el enfoque SVD se vuelven muy lentos cuando el número de características aumenta (por ejemplo, 100.000). En el lado positivo, ambos son lineales con respecto al número de instancias en el conjunto de entrenamiento (son  $O(m)$ ), por lo que manejan grandes conjuntos de entrenamiento de manera eficiente, siempre que quepan en la memoria.

Además, una vez que haya entrenado su modelo de regresión lineal (usando la ecuación normal o cualquier otro algoritmo), las predicciones son muy rápidas: la complejidad computacional es lineal tanto con respecto al número de instancias sobre las que desea hacer predicciones como al número de características. . En otras palabras, hacer predicciones sobre el doble de instancias (o el doble de características) llevará aproximadamente el doble de tiempo.

Ahora veremos una forma muy diferente de entrenar un modelo de regresión lineal, que es más adecuada para casos en los que hay una gran cantidad de características o demasiadas instancias de entrenamiento para caber en la memoria.

## Descenso de gradiente

El descenso de gradiente es un algoritmo de optimización genérico capaz de encontrar soluciones óptimas a una amplia gama de problemas. La idea general del descenso de gradiente es modificar los parámetros de forma iterativa para minimizar una función de costo.

Supongamos que estás perdido en las montañas en una densa niebla y solo puedes sentir la pendiente del suelo bajo tus pies. Una buena estrategia para llegar rápidamente al fondo del valle es descender en dirección a la pendiente más pronunciada. Esto es exactamente lo que hace el descenso de gradiente: mide el gradiente local de la función de error con respecto al vector de parámetros  $\theta$  y va en la dirección del gradiente descendente. Una vez que el gradiente es cero, ¡has alcanzado el mínimo!

En la práctica, se comienza llenando  $\theta$  con valores aleatorios (esto se llama inicialización aleatoria). Luego lo mejora gradualmente, dando un pequeño paso a la vez, cada paso intentando disminuir la función de costo (por ejemplo, el MSE), hasta que el algoritmo converja a un mínimo (consulte la Figura 4-3).

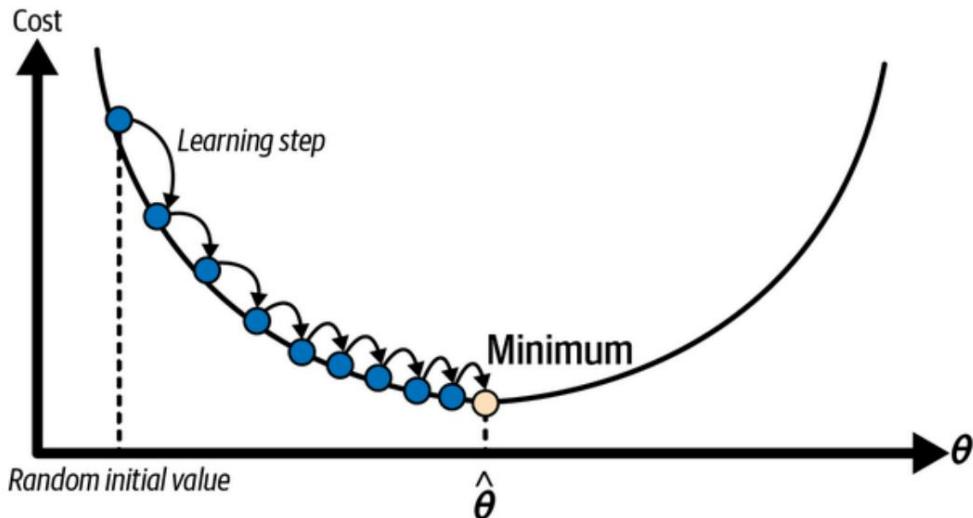


Figura 4-3. En esta representación del descenso de gradiente, los parámetros del modelo se inicializan aleatoriamente y se modifican repetidamente para minimizar la función de costo; El tamaño del paso de aprendizaje es proporcional a la pendiente de la función de costos, por lo que los pasos se hacen gradualmente más pequeños a medida que el costo se acerca al mínimo.

Un parámetro importante en el descenso de gradiente es el tamaño de los pasos, determinado por el hiperparámetro de tasa de aprendizaje . Si la tasa de aprendizaje es demasiado pequeña, entonces el algoritmo tendrá que pasar por muchas iteraciones para converger, lo que llevará mucho tiempo (consulte la Figura 4-4).

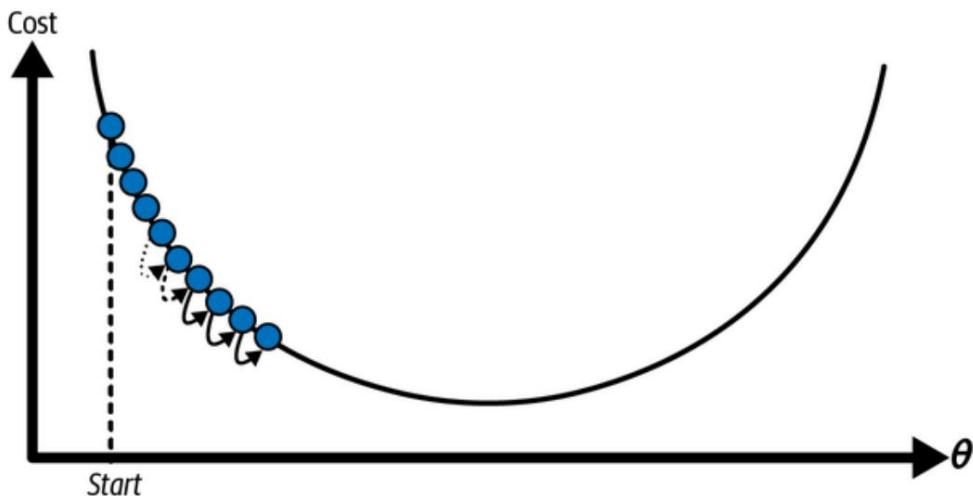


Figura 4-4. Tasa de aprendizaje demasiado pequeña

Por otro lado, si la tasa de aprendizaje es demasiado alta, podrías saltar a través del valle y terminar en el otro lado, posiblemente incluso más arriba que antes. Esto podría hacer que el algoritmo diverja, con valores cada vez mayores, y no pueda encontrar una buena solución (consulte la Figura 4-5).

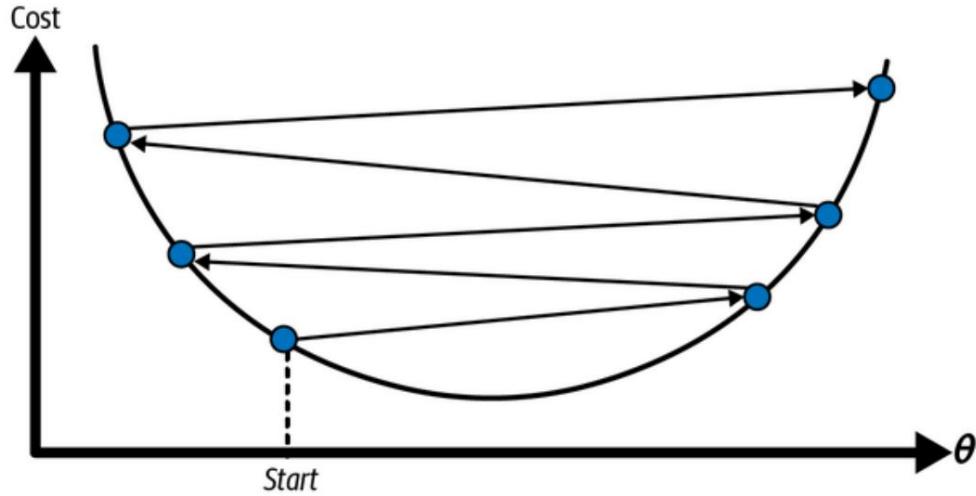


Figura 4-5. Tasa de aprendizaje demasiado alta

Además, no todas las funciones de costo parecen tazones bonitos y normales. Puede haber agujeros, crestas, mesetas y todo tipo de terreno irregular, dificultando la convergencia al mínimo. La Figura 4-6 muestra los dos desafíos principales del descenso de gradientes. Si la inicialización aleatoria inicia el algoritmo por la izquierda, entonces convergerá a un mínimo local, que no es tan bueno como el mínimo global. Si empieza por la derecha, tardará mucho en cruzar la meseta. Y si se detiene demasiado pronto, nunca se alcanzará el mínimo global.

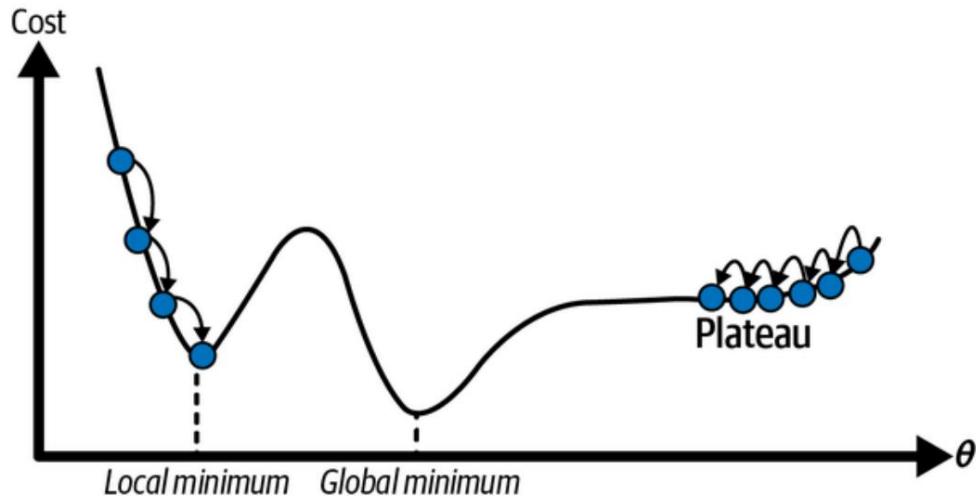


Figura 4-6. Errores del descenso de gradientes

Afortunadamente, la función de costos MSE para un modelo de regresión lineal resulta ser una función convexa, lo que significa que si se eligen dos puntos cualesquiera en la curva, el segmento de línea que los une nunca está debajo de la curva. Esto implica que no existen mínimos locales, sólo un mínimo global. También es una función continua con una pendiente que nunca cambia abruptamente. Estos dos hechos tienen una gran consecuencia: se garantiza que el descenso de gradiente se acercará arbitrariamente al mínimo global (si se espera lo suficiente y si la tasa de aprendizaje no es demasiado alta).

Si bien la función de costo tiene la forma de un cuenco, puede ser un cuenco alargado si las características tienen escalas muy diferentes. La Figura 4-7 muestra el descenso de gradiente en un conjunto de entrenamiento donde las características 1 y 2 tienen la misma escala (a la izquierda), y en un conjunto de entrenamiento donde la característica 1 tiene valores mucho más pequeños que la característica 2 (a la derecha).<sup>3</sup>

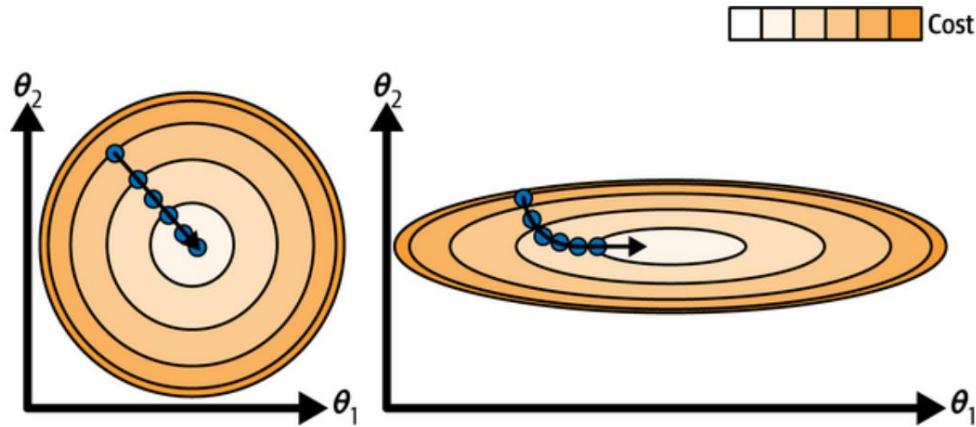


Figura 4-7. Descenso de gradiente con (izquierda) y sin escala de características (derecha)

Como puede ver, a la izquierda el algoritmo de descenso de gradiente va directamente hacia el mínimo, alcanzándolo rápidamente, mientras que a la derecha primero va en una dirección casi ortogonal a la dirección del mínimo global, y termina con una larga marcha por un valle casi llano. Al final llegará al mínimo, pero llevará mucho tiempo.

#### ADVERTENCIA

Al utilizar el descenso de gradiente, debe asegurarse de que todas las características tengan una escala similar (por ejemplo, usando la clase StandardScaler de Scikit-Learn), o de lo contrario tomará mucho más tiempo converger.

Este diagrama también ilustra el hecho de que entrenar un modelo significa buscar una combinación de parámetros del modelo que minimice una función de costo (sobre el conjunto de entrenamiento). Es una búsqueda en el espacio de parámetros del modelo. Cuantos más parámetros tiene un modelo, más dimensiones tiene este espacio y más difícil es la búsqueda: buscar una aguja en un pajar de 300 dimensiones es mucho más complicado que en 3 dimensiones.

Afortunadamente, dado que la función de costos es convexa en el caso de la regresión lineal, la aguja está simplemente en el fondo del recipiente.

## Descenso de gradiente por lotes

Para implementar el descenso de gradiente, es necesario calcular el gradiente de la función de costo con respecto a cada parámetro del modelo  $\theta$ . En otras palabras, necesita calcular cuánto cambiará  $J$  la función de costo si cambia  $\theta$  solo un poco. Esto se llama derivada parcial. Es como preguntar: “¿Cuál es la pendiente de la montaña bajo mis pies si

mira hacia el este"? y luego hacer la misma pregunta mirando hacia el norte (y así sucesivamente para todas las demás dimensiones, si puedes imaginar un universo con más de tres dimensiones). La ecuación 4-5 calcula la derivada parcial del MSE con respecto al parámetro  $\theta_j$ , anotó  $\partial \text{MSE}(\theta) / \partial \theta_j$ .

Ecuación 4-5. Derivadas parciales de la función de costos

$$\frac{\partial}{\partial \theta_j} \text{MSE}(\theta) = \frac{2}{\text{metro}} \sum_{y=1}^n (x(i) - y(i)) \times \sum_j^{(i)}$$

En lugar de calcular estas derivadas parciales individualmente, puedes usar la ecuación 4-6 para calcularlas todas de una sola vez. El vector gradiente, denominado  $\nabla \text{MSE}(\theta)$ , contiene todas las derivadas parciales de la función de costo (una para cada parámetro del modelo).

Ecuación 4-6. Vector gradiente de la función de costo.

$$\begin{aligned} \frac{\partial}{\partial \theta_0} \text{EEM}(\theta) \\ \frac{\partial}{\partial \theta_1} \text{EEM}(\theta) \\ \text{EEM}(\theta) = & \frac{2}{\text{metro}} X (X - y) \\ \frac{\partial}{\partial \theta_n} \text{EEM}(\theta) \end{aligned}$$

#### ADVERTENCIA

Tenga en cuenta que esta fórmula implica cálculos sobre el conjunto de entrenamiento completo  $X$ , en cada paso de descenso de gradiente. Es por eso que el algoritmo se llama descenso de gradiente por lotes: utiliza todo el lote de datos de entrenamiento en cada paso (en realidad, descenso de gradiente completo probablemente sería un mejor nombre). Como resultado, es terriblemente lento en conjuntos de entrenamiento muy grandes (en breve veremos algunos algoritmos de descenso de gradiente mucho más rápidos). Sin embargo, el descenso de gradiente se adapta bien a la cantidad de entidades; entrenar un modelo de regresión lineal cuando hay cientos de miles de características es mucho más rápido usando el descenso de gradiente que usando la ecuación normal o la descomposición SVD.

Una vez que tengas el vector de gradiente, que apunta hacia arriba, simplemente ve en la dirección opuesta para ir cuesta abajo. Esto significa restar  $\nabla \text{MSE}(\theta)$  de  $\theta$ . Aquí es donde entra en juego la tasa de aprendizaje  $\eta$ <sup>4</sup>: multiplique el vector gradiente por  $\eta$  para determinar el tamaño del paso cuesta abajo (Ecuación 4-7).

Ecuación 4-7. Paso de descenso de gradiente

$$(\text{próximo paso}) = -\eta \nabla \text{MSE}(\theta)$$

Veamos una implementación rápida de este algoritmo:

```

eta = 0.1 # tasa de aprendizaje n_epochs
= 1000 m = len(X_b) #
número de instancias

np.random.seed(42) theta
= np.random.randn(2, 1) # parámetros del modelo inicializados aleatoriamente

para época en el rango (n_epochs):
    gradientes = 2 / m * X_b.T @ (X_b @ theta - y) theta = theta - eta *
    gradientes

```

¡Eso no fue demasiado difícil! Cada iteración sobre el conjunto de entrenamiento se denomina época. Veamos el theta resultante:

```

>>> matriz
theta ([[4.21509616],
[2.77011339]])

```

¡Oye, eso es exactamente lo que encontró la ecuación normal! El descenso de gradiente funcionó perfectamente. Pero, ¿qué pasaría si hubieras utilizado una tasa de aprendizaje (eta) diferente? La Figura 4-8 muestra los primeros 20 pasos del descenso de gradiente utilizando tres tasas de aprendizaje diferentes. La línea en la parte inferior de cada gráfico representa el punto de inicio aleatorio, luego cada época está representada por una línea cada vez más oscura.

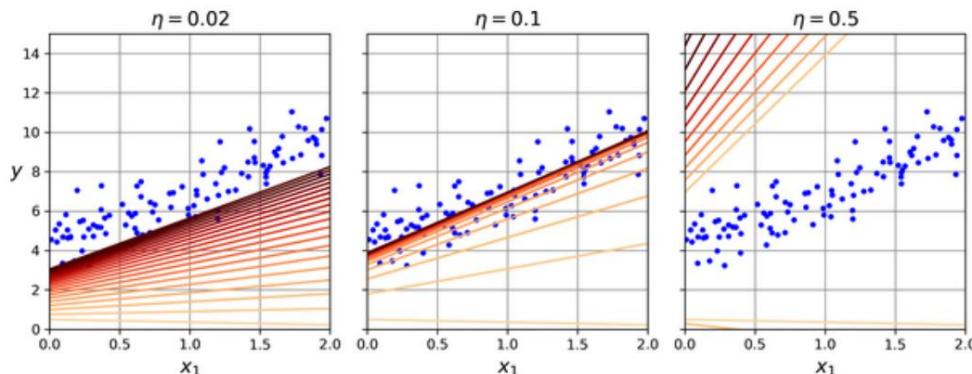


Figura 4-8. Descenso de gradiente con varias tasas de aprendizaje.

A la izquierda, la tasa de aprendizaje es demasiado baja: el algoritmo eventualmente llegará a la solución, pero llevará mucho tiempo. En el medio, la tasa de aprendizaje parece bastante buena: en solo unas pocas épocas, ya ha convergido a la solución. A la derecha, la tasa de aprendizaje es demasiado alta: el algoritmo diverge, salta por todos lados y, de hecho, se aleja cada vez más de la solución en cada paso.

Para encontrar una buena tasa de aprendizaje, puede utilizar la búsqueda en cuadrícula (consulte el Capítulo 2). Sin embargo, es posible que desee limitar el número de épocas para que la búsqueda en cuadrícula pueda eliminar los modelos que tardan demasiado en converger.

Quizás se pregunte cómo establecer el número de épocas. Si es demasiado bajo, aún estará lejos de la solución óptima cuando el algoritmo se detenga; pero si es demasiado alto, lo harás

perder el tiempo mientras los parámetros del modelo no cambian más. Una solución simple es establecer un número muy grande de épocas pero interrumpir el algoritmo cuando el vector de gradiente se vuelve pequeño, es decir, cuando su norma se vuelve más pequeña que un número pequeño (llamado tolerancia), porque esto sucede cuando el descenso del gradiente tiene (casi) alcanzó el mínimo.

### TASA DE CONVERGENCIA

Cuando la función de costo es convexa y su pendiente no cambia abruptamente (como es el caso de la función de costo MSE), el descenso del gradiente por lotes con una tasa de aprendizaje fija eventualmente convergerá a la solución óptima, pero es posible que deba esperar un tiempo: pueden ser necesarias  $O(1/\epsilon)$  iteraciones para alcanzar el óptimo dentro de un rango de  $\epsilon$ , dependiendo de la forma de la función de costos. Si divide la tolerancia por 10 para tener una solución más precisa, es posible que el algoritmo deba ejecutarse unas 10 veces más.

## Descenso del gradiente estocástico

El principal problema con el descenso de gradientes por lotes es el hecho de que utiliza todo el conjunto de entrenamiento para calcular los gradientes en cada paso, lo que lo hace muy lento cuando el conjunto de entrenamiento es grande. En el extremo opuesto, el descenso de gradiente estocástico selecciona una instancia aleatoria en el conjunto de entrenamiento en cada paso y calcula los gradientes basándose únicamente en esa única instancia. Obviamente, trabajar en una sola instancia a la vez hace que el algoritmo sea mucho más rápido porque tiene muy pocos datos para manipular en cada iteración. También hace posible entrenar en conjuntos de entrenamiento enormes, ya que solo es necesario que haya una instancia en la memoria en cada iteración (GD estocástico se puede implementar como un algoritmo fuera del núcleo; consulte el Capítulo 1).

Por otro lado, debido a su naturaleza estocástica (es decir, aleatoria), este algoritmo es mucho menos regular que el descenso de gradiente por lotes: en lugar de disminuir suavemente hasta alcanzar el mínimo, la función de costo rebotará hacia arriba y hacia abajo, disminuyendo solo en promedio. Con el tiempo terminará muy cerca del mínimo, pero una vez que llegue allí continuará rebotando sin estabilizarse nunca (ver [Figura 4-9](#)). Una vez que el algoritmo se detiene, los valores finales de los parámetros serán buenos, pero no óptimos.

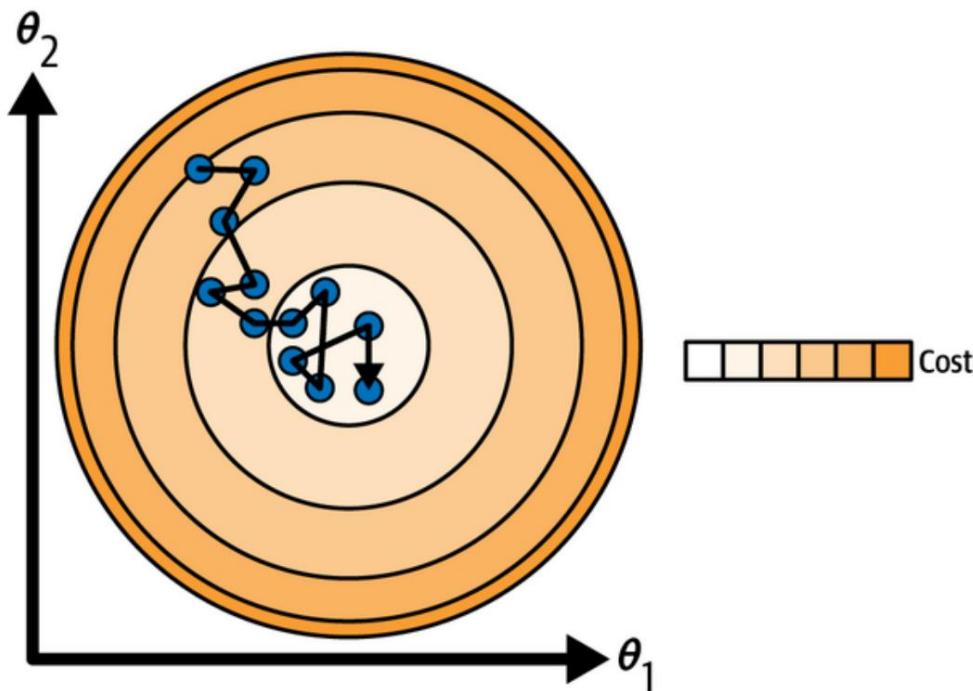


Figura 4-9. Con el descenso de gradiente estocástico, cada paso de entrenamiento es mucho más rápido pero también mucho más estocástico que cuando se utiliza el descenso de gradiente por lotes.

Cuando la función de costo es muy irregular (como en [la Figura 4-6](#)), esto en realidad puede ayudar al algoritmo a saltar de los mínimos locales, por lo que el descenso de gradiente estocástico tiene más posibilidades de encontrar el mínimo global que el descenso de gradiente por lotes.

Por lo tanto, la aleatoriedad es buena para escapar de los óptimos locales, pero es mala porque significa que el algoritmo nunca puede establecerse en el mínimo. Una solución a este dilema es reducir gradualmente la tasa de aprendizaje. Los pasos comienzan siendo grandes (lo que ayuda a avanzar rápidamente y escapar de los mínimos locales) y luego se hacen cada vez más pequeños, lo que permite que el algoritmo se establezca en el mínimo global. Este proceso es similar al recocido simulado, un algoritmo inspirado en el proceso metalúrgico de recocido, donde el metal fundido se enfria lentamente. La función que determina la tasa de aprendizaje en cada iteración se llama programa de aprendizaje. Si la tasa de aprendizaje se reduce demasiado rápido, es posible que se quede atrapado en un mínimo local o incluso termine congelado a mitad del mínimo. Si la tasa de aprendizaje se reduce demasiado lentamente, puede saltarse el mínimo durante mucho tiempo y terminar con una solución subóptima si detiene el entrenamiento demasiado pronto.

Este código implementa el descenso de gradiente estocástico utilizando un programa de aprendizaje simple:

```
n_epochs = 50 t0,
t1 = 5, 50 # hiperparámetros del programa de aprendizaje

def horario_aprendizaje(t): devolver
    t0 / (t + t1)
```

```

np.random.seed(42) theta =
np.random.randn(2, 1) # inicialización aleatoria

para época en el rango (n_epochs):
    para iteración en rango (m): índice_aleatorio
        = np.random.randint(m) xi = X_b[índice_aleatorio :
        índice_aleatorio + 1] yi = y[índice_aleatorio : índice_aleatorio + 1]
        gradientes = 2 * xi.T @ (xi @ theta - yi) # para SGD, no dividir

    por m
        eta = calendario_aprendizaje(época * theta = theta  m + iteración)
        - eta * gradientes

```

Por convención iteramos por rondas de iteraciones ; cada ronda se llama época, como antes. Mientras que el código de descenso de gradiente por lotes se repitió 1000 veces a través de todo el conjunto de entrenamiento, este código pasa por el conjunto de entrenamiento solo 50 veces y llega a una solución bastante buena:

```

>>> theta
matrix([[4.21076011],
       [2.74856079]])

```

La Figura 4-10 muestra los primeros 20 pasos del entrenamiento (observe cuán irregulares son los pasos).

Tenga en cuenta que, dado que las instancias se seleccionan al azar, algunas instancias pueden seleccionarse varias veces por época, mientras que otras pueden no seleccionarse en absoluto. Si desea estar seguro de que el algoritmo pasa por todas las instancias en cada época, otro método es mezclar el conjunto de entrenamiento (asegurándose de mezclar las características de entrada y las etiquetas de manera conjunta), luego revisarlo instancia por instancia y luego mezclarlo. otra vez, y así sucesivamente. Sin embargo, este enfoque es más complejo y generalmente no mejora el resultado.

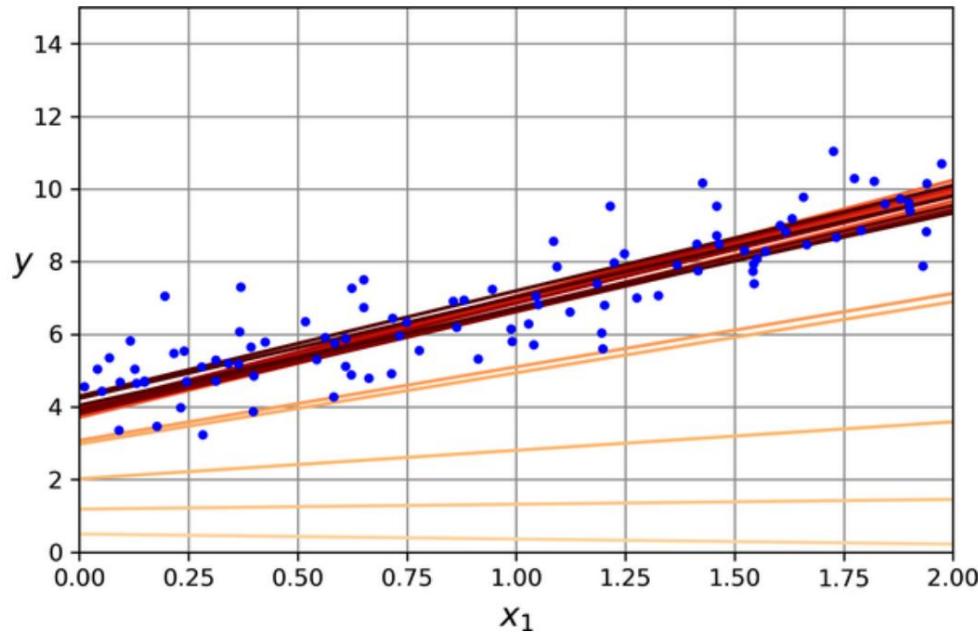


Figura 4-10. Los primeros 20 pasos del descenso de gradiente estocástico

**ADVERTENCIA**

Cuando se utiliza el descenso de gradiente estocástico, las instancias de entrenamiento deben ser independientes y estar distribuidas de manera idéntica (IID) para garantizar que los parámetros se acerquen al óptimo global, en promedio. Una forma sencilla de garantizar esto es mezclar las instancias durante el entrenamiento (por ejemplo, elegir cada instancia al azar o mezclar el conjunto de entrenamiento al comienzo de cada época). Si no mezcla las instancias (por ejemplo, si las instancias se ordenan por etiqueta), SGD comenzará optimizando para una etiqueta, luego para la siguiente, y así sucesivamente, y no se acercará al mínimo global.

Para realizar una regresión lineal usando GD estocástico con Scikit-Learn, puede usar la clase SGDRegressor, que de forma predeterminada optimiza la función de costo MSE. El siguiente código se ejecuta durante un máximo de 1000 épocas (max\_iter) o hasta que la pérdida se reduzca a  $\leq 10^{-5}$  (tol) durante 100 épocas (n\_iter\_no\_change). Comienza con una tasa de aprendizaje de 0,01 (eta0), utilizando el programa de aprendizaje predeterminado (diferente al que usamos). Por último, no utiliza ninguna regularización (penalización=Ninguna; más detalles sobre esto en breve):

```
desde sklearn.linear_model importar SGDRegressor
sgd_reg = SGDRegressor(max_iter=1000, tol=1e-5, penalización=Ninguna, eta0=0.01,
                      n_iter_no_change=100, estado_aleatorio=42)
sgd_reg.fit(X, y.ravel()) # y.ravel() porque fit() espera objetivos 1D
```

Una vez más, encuentras una solución bastante cercana a la que arroja la ecuación normal:

```
>>> sgd_reg.intercept_, sgd_reg.coef_
(matriz([4.21278812]), matriz([2.77270267]))
```

## CONSEJO

Todos los estimadores de Scikit-Learn se pueden entrenar usando el método `fit()`, pero algunos estimadores también tienen un método `partial_fit()` al que se puede llamar para ejecutar una única ronda de entrenamiento en una o más instancias (ignora hiperparámetros como `max_iter` o `tol`). . Llamar repetidamente a `partial_fit()` entrenará gradualmente el modelo. Esto resulta útil cuando necesita más control sobre el proceso de formación. Otros modelos tienen en su lugar un hiperparámetro `warm_start` (y algunos tienen ambos): si estableces `warm_start=True`, llamar al método `fit()` en un modelo entrenado no restablecerá el modelo; simplemente continuará entrenando donde lo dejó, respetando hiperparámetros como `max_iter` y `tol`. Tenga en cuenta que `fit()` restablece el contador de iteraciones utilizado por el programa de aprendizaje, mientras que `partial_fit()` no lo hace.

## Descenso de gradiente de mini lotes

El último algoritmo de descenso de gradiente que veremos se llama descenso de gradiente por mini lotes. Es sencillo una vez que conoces el descenso de gradiente estocástico y por lotes: en cada paso, en lugar de calcular los gradientes en función del conjunto de entrenamiento completo (como en GD por lotes) o en base a una sola instancia (como en GD estocástico), se utiliza GD en mini lotes. calcula los gradientes en pequeños conjuntos aleatorios de instancias llamados minilotes. La principal ventaja del GD mini-batch sobre el GD estocástico es que puede obtener un aumento de rendimiento mediante la optimización del hardware de las operaciones matriciales, especialmente cuando se utilizan GPU.

El progreso del algoritmo en el espacio de parámetros es menos errático que con GD estocástico, especialmente con minilotes bastante grandes. Como resultado, el GD de mini lotes terminará caminando un poco más cerca del mínimo que el GD estocástico, pero puede resultarle más difícil escapar de los mínimos locales (en el caso de problemas que sufren de mínimos locales, a diferencia de la regresión lineal). con la función de costos MSE). [La Figura 4-11](#) muestra los caminos tomados por los tres algoritmos de descenso de gradiente en el espacio de parámetros durante el entrenamiento. Todos terminan cerca del mínimo, pero la ruta del GD por lotes en realidad se detiene en el mínimo, mientras que tanto el GD estocástico como el GD mini-batch continúan caminando. Sin embargo, no olvide que el GD por lotes requiere mucho tiempo para realizar cada paso, y el GD estocástico y el GD por mini lotes también alcanzarían el mínimo si utilizara un buen programa de aprendizaje.

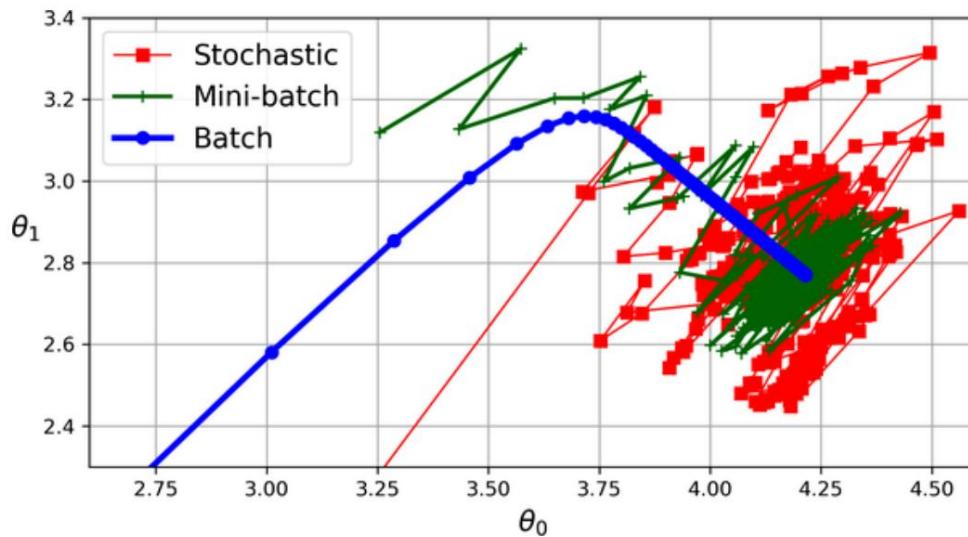


Figura 4-11. Rutas de descenso de gradiente en el espacio de parámetros

La tabla 4-1 compara los algoritmos que hemos analizado hasta ahora para la regresión lineal.

5

(recuerde que  $m$  es el número de instancias de entrenamiento y  $n$  es el número de características).

Tabla 4-1. Comparación de algoritmos de regresión lineal.

Algoritmo	Grande $m$	Fuera del núcleo	apoyo	grande norte	hiperparámetros	Escalada requerido
Normal ecuación	Rápido	No		Lento	0	No
SVD	Rápido	No		Lento	0	No
Lote GD	Lento	No		Rápido	2	Sí
Estocástico GD rápido		Sí		Rápido	$\geq 2$	Sí
Mini lote GD Fast		Sí		Rápido	$\geq 2$	Sí

Casi no hay diferencia después del entrenamiento: todos estos algoritmos terminan con resultados muy similares y hacer predicciones exactamente de la misma manera.

## Regresión polinomial

¿Qué pasa si tus datos son más complejos que una línea recta? Sorprendentemente, puedes utilizar un modelo lineal para ajustar datos no lineales. Una forma sencilla de hacer esto es sumar potencias de cada característica como características nuevas, luego entrena un modelo lineal en este conjunto extendido de características.

Esta técnica se llama regresión polinómica.

Veamos un ejemplo. Primero, generaremos algunos datos no lineales (consulte [la Figura 4-12](#)), basados en una ecuación cuadrática simple (es decir, una ecuación de la forma  $y = ax^2 + bx + c$ ) , más algo de ruido:

```
np.semilla.aleatoria(42) m
= 100
X = 6 * np.random.rand(m, 1) - 3 y = 0.5 * X ** 2
+ X + 2 + np.random.randn(m, 1)
```

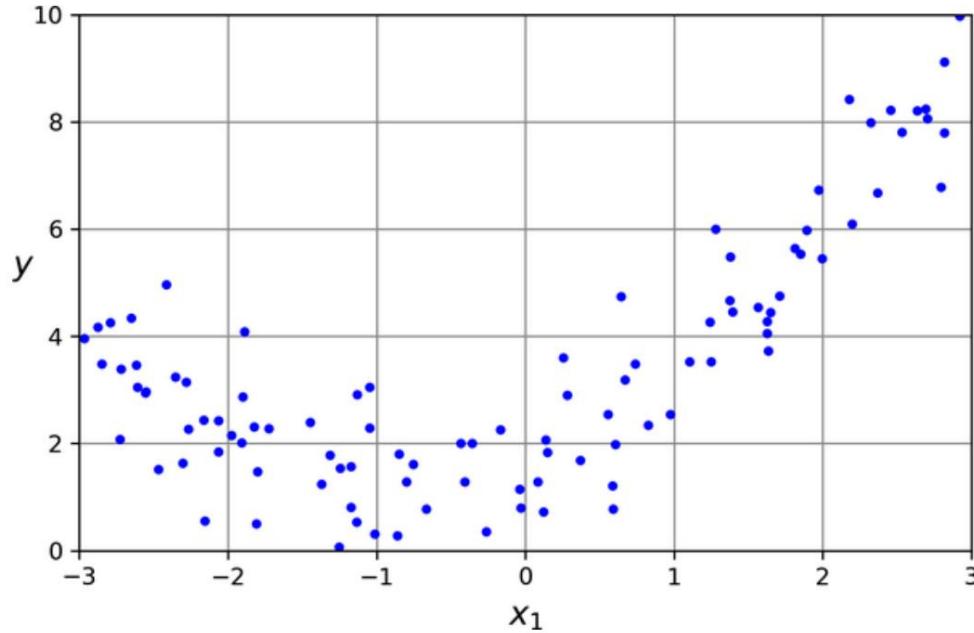


Figura 4-12. Conjunto de datos no lineal y ruidoso generado.

Es evidente que una línea recta nunca se ajustará correctamente a estos datos. Entonces, usemos la clase `PolynomialFeatures` de Scikit-Learn para transformar nuestros datos de entrenamiento, agregando el cuadrado (polinomio de segundo grado) de cada característica en el conjunto de entrenamiento como una nueva característica (en este caso solo hay una característica):

```
>>> de sklearn.preprocessing importar PolynomialFeatures >>> poli_características =
PolynomialFeatures(grado=2, include_bias=False)
>>> X_poli = poli_características.fit_transform(X)
>>> X[0]
matriz([-0.75275929])
>>> Matriz X_poly[0]
([-0.75275929, 0.56664654])
```

$X_{\text{poly}}$  ahora contiene la característica original de  $X$  más el cuadrado de esta característica. Ahora podemos ajustar un modelo de Regresión Lineal a estos datos de entrenamiento extendidos ([Figura 4-13](#)):

```
>>> lin_reg = Regresión Lineal() >>>
lin_reg.fit(X_poly, y)
```

```
>>> lin_reg.intercept_, lin_reg.coef_
(matriz ([1.78134581]), matriz ([[0.93366893, 0.56456263]]))
```

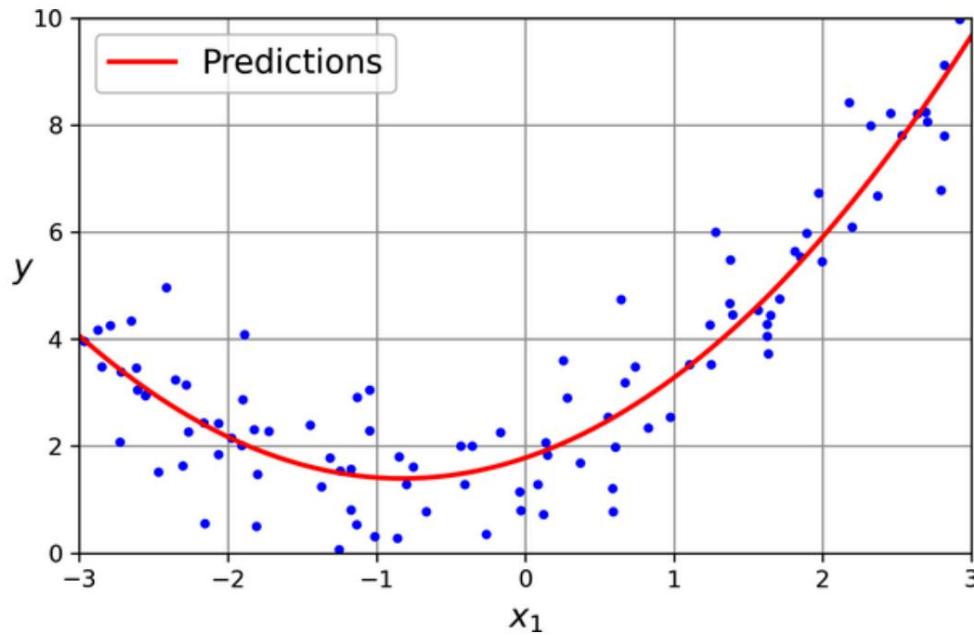


Figura 4-13. Predicciones del modelo de regresión polinómica

Nada mal: el modelo estima  $\hat{y} = 0,56x_1^2 + 0,93x_1 + 1,78$  cuando en realidad el función original era  $y = 0.5x_1^2 + 1,0x_1 + 2,0 + \text{ruido gaussiano}$ .

Tenga en cuenta que cuando hay múltiples características, la regresión polinómica es capaz de encontrar relaciones entre características, que es algo que un simple modelo de regresión lineal no puedo hacer. Esto es posible gracias al hecho de que PolynomialFeatures también agrega todas las combinaciones de características hasta el grado dado. Por ejemplo, si hubiera dos características  $a$  y  $b$ , PolynomialFeatures con grado = 3 no solo agregaría el  $a^2, 3a^2, 3^2, ab, b^2$ , pero también las combinaciones  $ab, a^2b, ab^2$ .

#### ADVERTENCIA

PolynomialFeatures(grado=d) transforma una matriz que contiene  $n$  características en una matriz que contiene  $(n + d)! / n!$  características, donde  $n!$  es el factorial de  $n$ , igual a  $1 \times 2 \times 3 \times \dots \times n$ . ¡Cuidado con la explosión combinatoria del número de funciones!

## Curvas de aprendizaje

Si realiza una regresión polinómica de alto grado, probablemente se ajuste a los datos de entrenamiento mucho mejor que con la simple regresión lineal. Por ejemplo, la Figura 4-14 aplica un 300-grado polinómico de grado con los datos de entrenamiento anteriores y compara el resultado con un modelo lineal puro y un modelo cuadrático (polinomio de segundo grado). Date cuenta cómo

el modelo polinomial de 300 grados se mueve para acercarse lo más posible a las instancias de entrenamiento.

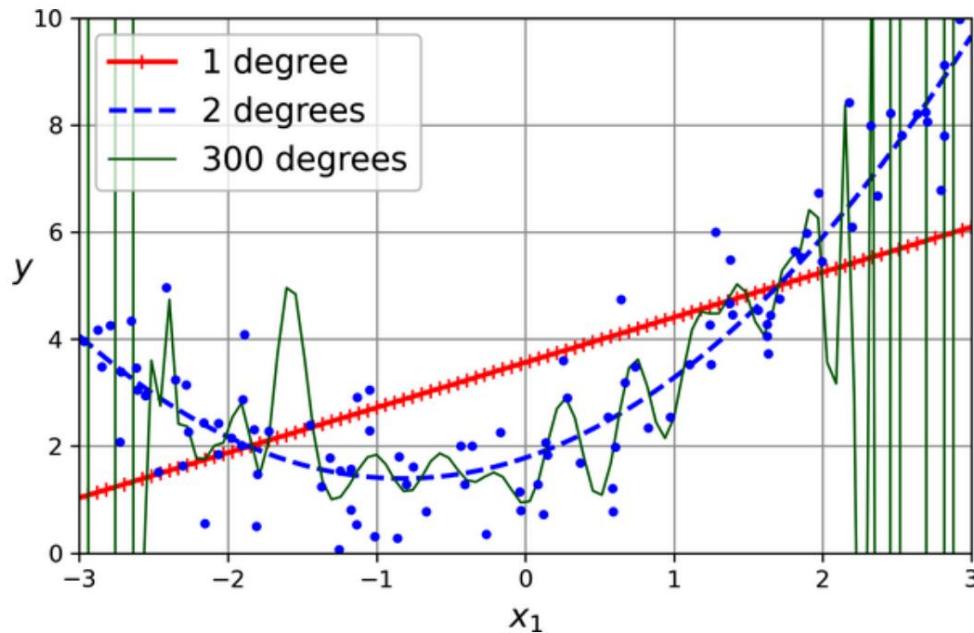


Figura 4-14. Regresión polinómica de alto grado

Este modelo de regresión polinómica de alto grado sobreajusta gravemente los datos de entrenamiento, mientras que el modelo lineal los sobreajusta. El modelo que se generalizará mejor en este caso es el modelo cuadrático, que tiene sentido porque los datos se generaron mediante un modelo cuadrático. Pero, en general, no sabrá qué función generó los datos, entonces, ¿cómo puede decidir qué tan complejo debe ser su modelo? ¿Cómo puede saber si su modelo se está sobreajustando o no ajustando los datos?

En el Capítulo 2 utilizó la validación cruzada para obtener una estimación del rendimiento de generalización de un modelo. Si un modelo funciona bien con los datos de entrenamiento pero se generaliza mal según las métricas de validación cruzada, entonces su modelo está sobreajustado. Si funciona mal en ambos, entonces no es adecuado. Ésta es una forma de saber cuándo un modelo es demasiado simple o demasiado complejo.

Otra forma de saberlo es observar las curvas de aprendizaje, que son gráficos del error de entrenamiento y del error de validación del modelo en función de la iteración de entrenamiento: simplemente evalúe el modelo a intervalos regulares durante el entrenamiento tanto en el conjunto de entrenamiento como en el conjunto de validación, y trazar los resultados. Si el modelo no se puede entrenar de forma incremental (es decir, si no admite `partial_fit()` o `warm_start`), entonces debes entrenarlo varias veces en subconjuntos gradualmente más grandes del conjunto de entrenamiento.

Scikit-Learn tiene una útil función `learning_curve()` para ayudar con esto: entrena y evalúa el modelo mediante validación cruzada. De forma predeterminada, vuelve a entrenar el modelo en subconjuntos crecientes del conjunto de entrenamiento, pero si el modelo admite el aprendizaje incremental, puede configurar `exploit_incremental_learning=True` al llamar a `learning_curve()` y

En su lugar, entrenará el modelo de forma incremental. La función devuelve los tamaños del conjunto de entrenamiento en los que evaluó el modelo y las puntuaciones de entrenamiento y validación que midió para cada tamaño y para cada pliegue de validación cruzada. Usemos esta función para observar las curvas de aprendizaje del modelo de regresión lineal simple (ver Figura 4-15):

```
desde sklearn.model_selection importar curva_aprendizaje

tamaños_de_tren, puntuaciones_de_tren, puntuaciones_válidas = curva_de_aprendizaje(
    LinearRegression(), X, y, train_sizes=np.linspace(0.01, 1.0, 40), cv=5,
    scoring="neg_root_mean_squared_error") train_errors =
    -train_scores.mean(axis=1) valid_errors =
    -valid_scores.mean(axis =1)

plt.plot(train_sizes, train_errors, "r-+", linewidth=2, label="train") plt.plot(train_sizes, valid_errors,
    "b-", linewidth=3, label="valid") [...] # embellecer la figura: agregar etiquetas, eje, cuadrícula y
leyenda plt.show()
```

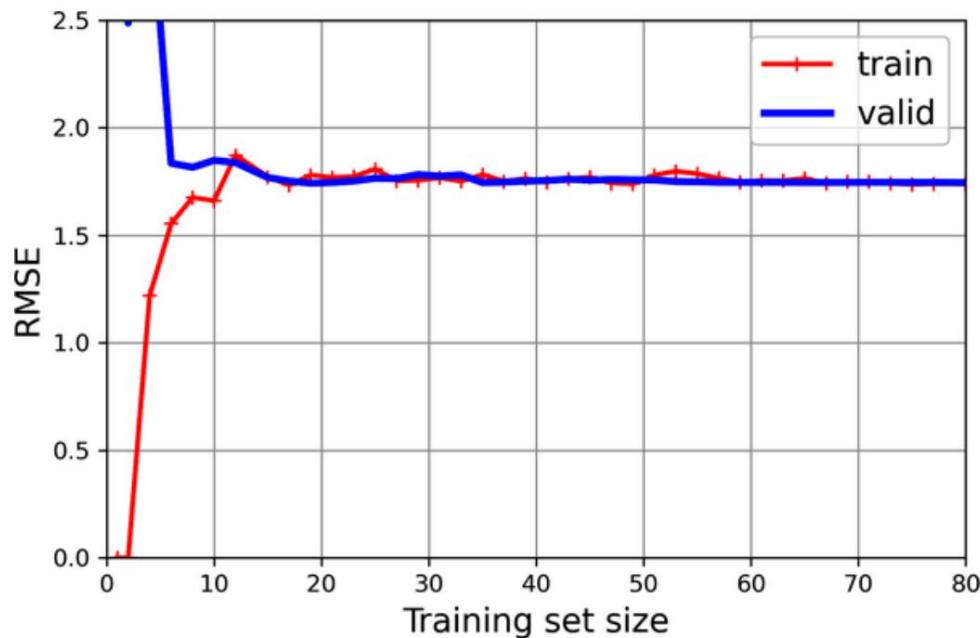


Figura 4-15. Curvas de aprendizaje

Este modelo no es adecuado. Para ver por qué, primero veamos el error de entrenamiento. Cuando hay sólo una o dos instancias en el conjunto de entrenamiento, el modelo puede ajustarlas perfectamente, razón por la cual la curva comienza en cero. Pero a medida que se agregan nuevas instancias al conjunto de entrenamiento, resulta imposible que el modelo se ajuste perfectamente a los datos de entrenamiento, tanto porque los datos son ruidosos como porque no son lineales en absoluto. Entonces, el error en los datos de entrenamiento aumenta hasta llegar a una meseta, momento en el cual agregar nuevas instancias al conjunto de entrenamiento no mejora ni empeora el error promedio. Ahora veamos el error de validación. Cuando el modelo se entrena en muy pocas instancias de entrenamiento, es incapaz de generalizarse adecuadamente, razón por la cual el error de validación es inicialmente bastante grande. Entonces como

Al modelo se le muestran más ejemplos de entrenamiento, aprende y, por lo tanto, el error de validación disminuye lentamente. Sin embargo, una vez más, una línea recta no puede modelar bien los datos, por lo que el error termina en una meseta, muy cerca de la otra curva.

Estas curvas de aprendizaje son típicas de un modelo insuficientemente adaptado. Ambas curvas han llegado a una meseta; están cerca y bastante altos.

## CONSEJO

Si su modelo no se ajusta adecuadamente a los datos de entrenamiento, agregar más ejemplos de entrenamiento no ayudará. Necesita utilizar un modelo mejor o crear mejores funciones.

Ahora veamos las curvas de aprendizaje de un modelo polinómico de décimo grado con los mismos datos ([Figura 4-16](#)):

```
desde sklearn.pipeline importar make_pipeline

regresión_polinomial = make_pipeline(
    Características polinómicas (grado = 10, incluir_bias = Falso),
    Regresión lineal())

tamaños_de_tren, puntuaciones_de_tren, puntuaciones_válidas = curva_de_aprendizaje(
    regresión_polinomial, X, y, train_sizes=np.linspace(0.01, 1.0, 40), cv=5, scoring="neg_root_mean_squared_error")

[...] # igual que antes
```

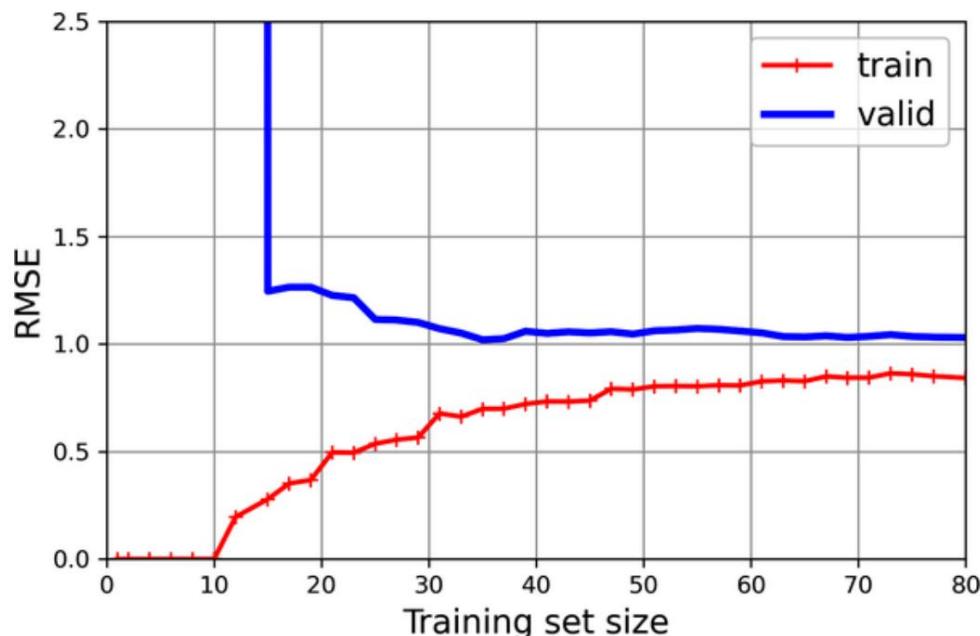


Figura 4-16. Curvas de aprendizaje para el modelo polinómico de décimo grado

Estas curvas de aprendizaje se parecen un poco a las anteriores, pero hay dos diferencias muy importantes:

- El error en los datos de entrenamiento es mucho menor que antes.
- Hay una brecha entre las curvas. Esto significa que el modelo funciona significativamente mejor con los datos de entrenamiento que con los datos de validación, lo cual es el sello distintivo de un modelo de sobreajuste. Sin embargo, si utilizaras un conjunto de entrenamiento mucho más grande, las dos curvas seguirían acercándose.

#### CONSEJO

Una forma de mejorar un modelo de sobreajuste es alimentarlo con más datos de entrenamiento hasta que el error de validación alcance el error de entrenamiento.

## LA COMPENSACIÓN SESGO/VARIANZA

Un resultado teórico importante de la estadística y el aprendizaje automático es el hecho de que el error de generalización de un modelo se puede expresar como la suma de tres factores muy diferentes. errores:

#### Inclinación

Esta parte del error de generalización se debe a suposiciones erróneas, como suponer que los datos son lineales cuando en realidad son cuadráticos. Es más probable que un modelo con alto sesgo no se ajuste a los datos de entrenamiento.<sup>6</sup>

#### Diferencia

Esta parte se debe a la excesiva sensibilidad del modelo a pequeñas variaciones en los datos de entrenamiento. Es probable que un modelo con muchos grados de libertad (como un modelo polinomial de alto grado) tenga una varianza alta y, por lo tanto, sobreajuste los datos de entrenamiento.

#### error irreducible

Esta parte se debe al ruido de los propios datos. La única forma de reducir esta parte del error es limpiar los datos (por ejemplo, arreglar las fuentes de datos, como sensores rotos, o detectar y eliminar valores atípicos).

Aumentar la complejidad de un modelo normalmente aumentará su varianza y reducirá su sesgo. Por el contrario, reducir la complejidad de un modelo aumenta su sesgo y reduce su varianza. Por eso se le llama compensación.

## Modelos lineales regularizados

Como vio en los Capítulos 1 y 2, una buena manera de reducir el sobreajuste es regularizar el modelo (es decir, restringirlo): cuantos menos grados de libertad tenga, más difícil le resultará sobreajustar los datos. Una forma sencilla de regularizar un modelo polinómico es reducir el número de grados del polinomio.

Para un modelo lineal, la regularización normalmente se logra restringiendo los pesos del modelo. Ahora veremos la regresión de crestas, la regresión de lazo y la regresión neta elástica, que implementan tres formas diferentes de restringir los pesos.

### Regresión de crestas La

regresión de crestas (también llamada regularización de Tikhonov) es una versión regularizada de la regresión lineal: un término de regularización igual a Esto  $\frac{\alpha}{\text{metro}} \sum_{y_0=1} \theta_i^2$  se agrega al MSE. obliga al algoritmo de aprendizaje no solo a ajustar los datos sino también a mantener los pesos del modelo lo más pequeños posible. Tenga en cuenta que el término de regularización solo debe agregarse a la función de costos durante la capacitación. Una vez que se entrena el modelo, desea utilizar el MSE no regularizado (o RMSE) para evaluar el rendimiento del modelo.

El hiperparámetro  $\alpha$  controla cuánto desea regularizar el modelo. Si  $\alpha = 0$ , entonces la regresión de crestas es simplemente una regresión lineal. Si  $\alpha$  es muy grande, entonces todos los pesos terminan muy cerca de cero y el resultado es una línea plana que pasa por la media de los datos.

La ecuación 4-8 presenta la función de costo de regresión de cresta. <sup>7</sup>

Ecuación 4-8. Función de costo de regresión de cresta

$$J(\theta) = \text{MSE}(\theta) + \frac{\alpha}{\text{metro}} \sum_{y_0=1} \theta_i^2$$

Tenga en cuenta que el término de sesgo  $\theta_0$  no está regularizado (la suma comienza en  $i = 1$ , no en 0). Si definimos  $w$  como el vector de pesos de características ( $\theta_1$  a  $\theta_n$ ), entonces el término de regularización es igual a  $\alpha \|w\|_2^2 / m$ , donde  $\|w\|_2$  representa la norma  $\ell_2$  del vector de peso. Para el descenso de gradiente por lotes, simplemente agregue  $2\alpha w / m$  a la parte del vector de gradiente MSE que corresponde a los pesos de las características, sin agregar nada al gradiente del término de sesgo (consulte la Ecuación 4-6).

#### ADVERTENCIA

Es importante escalar los datos (por ejemplo, usando un StandardScaler) antes de realizar la regresión de crestas, ya que es sensible a la escala de las características de entrada. Esto es cierto para la mayoría de los modelos regularizados.

La Figura 4-17 muestra varios modelos de crestas que se entrenaron con datos lineales muy ruidosos utilizando diferentes valores de  $\alpha$ . A la izquierda, se utilizan modelos de crestas planas, lo que lleva a lineales.

predicciones A la derecha, los datos primero se expanden usando `PolynomialFeatures` (grado = 10), luego se escalan usando un `StandardScaler` y, finalmente, los modelos de cresta se aplican a las características resultantes: esto es regresión polinómica con regularización de cresta. Observe cómo el aumento de  $\alpha$  conduce a predicciones más planas (es decir, menos extremas, más razonables), reduciendo así la varianza del modelo pero aumentando su sesgo.

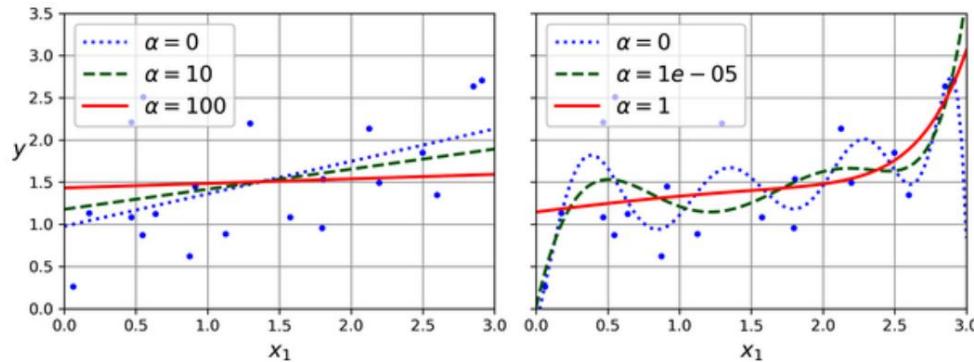


Figura 4-17. Modelos lineales (izquierda) y polinomiales (derecha), ambos con varios niveles de regularización de crestas.

Al igual que con la regresión lineal, podemos realizar una regresión de crestas calculando una ecuación de forma cerrada o realizando un descenso de gradiente. Los pros y los contras son los mismos. La ecuación 4-9 muestra la solución de forma cerrada, donde  $A$  es la matriz identidad  $(n + 1) \times (n + 1)$ , excepto con un 0 en la celda superior izquierda, correspondiente al término de sesgo.

Ecuación 4-9. Solución de forma cerrada de regresión de crestas

$$\hat{y} = (\mathbf{X}^\top \mathbf{X} + \alpha \mathbf{A})^{-1} \mathbf{X}^\top \mathbf{y}$$

A continuación se explica cómo realizar una regresión de crestas con Scikit-Learn utilizando una solución de forma cerrada (una variante de la Ecuación 4-9 que utiliza una técnica de factorización matricial de André-Louis Cholesky):

```
>>> from sklearn.linear_model import Ridge >>> ridge_reg = Ridge(alpha=0.1, solver="cholesky") >>> ridge_reg.fit(X, y) >>>
ridge_reg.predict([[1.5]])
array([1.55325833])
```

Y usando descenso de gradiente estocástico:

10

```
>>> sgd_reg = SGDRegressor(penalty="l2", alpha=0.1 / m, tol=None, max_iter=1000,
...                         eta0=0.01, random_state=42)
...
>>> sgd_reg.fit(X, y.ravel()) # y.ravel() porque fit() espera objetivos 1D >>>
sgd_reg.predict([[1.5]])
array([1.55302613])
```

El hiperparámetro de penalización establece el tipo de término de regularización que se utilizará. Especificar "l2" indica que desea que SGD agregue un término de regularización a la función de costo de MSE igual a alfa multiplicado por el cuadrado de la norma  $\ell_2$  del vector de peso. Esto es como la regresión de crestas, excepto que en este caso no hay división por m ; es por eso que pasamos alfa=0,1/m, para obtener el mismo resultado que Ridge(alfa=0,1).

## CONSEJO

La clase RidgeCV también realiza una regresión de crestas, pero ajusta automáticamente los hiperparámetros mediante validación cruzada. Es más o menos equivalente a usar GridSearchCV, pero está optimizado para la regresión de crestas y se ejecuta mucho más rápido. Varios otros estimadores (en su mayoría lineales) también tienen variantes de CV eficientes, como LassoCV y ElasticNetCV.

## Regresión de lazo

La regresión del operador de selección y contracción mínima absoluta (generalmente llamada simplemente regresión de lazo) es otra versión regularizada de la regresión lineal: al igual que la regresión de cresta, agrega un término de regularización a la función de costo, pero usa la norma  $\ell_1$  del vector de peso en lugar de la cuadrado de la norma  $\ell_2$  (ver [Ecuación 4-10](#)). Observe que la norma  $\ell_1$  se multiplica por  $2\alpha$ , mientras que la norma  $\ell_2$  se multiplica por  $\alpha / m$  en la regresión de crestas. Estos factores se eligieron para garantizar que el valor  $\alpha$  óptimo sea independiente del tamaño del conjunto de entrenamiento: diferentes normas conducen a diferentes factores (consulte [el número 15657 de Scikit-Learn](#) para más detalles).

Ecuación 4-10. Función de costo de regresión de lazo

$$J(\theta) = \text{MSE}(\theta) + 2\alpha \sum_{i=1}^n |\theta_i|$$

[La Figura 4-18](#) muestra lo mismo que [la Figura 4-17](#), pero reemplaza los modelos de cresta con modelos de lazo y usa diferentes valores de  $\alpha$ .

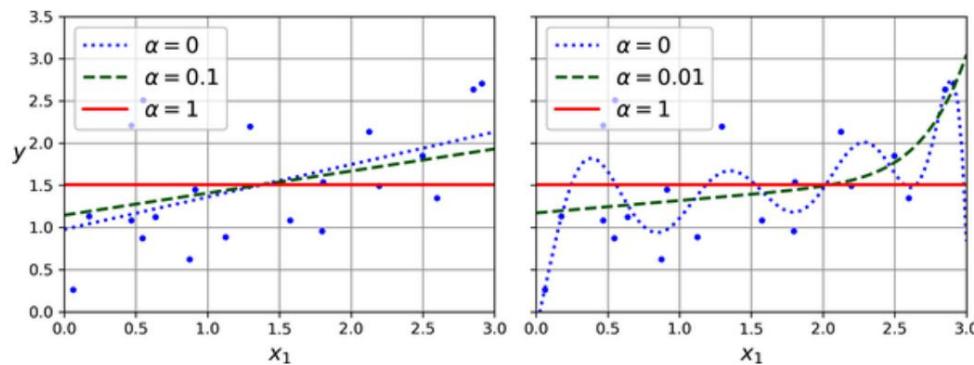


Figura 4-18. Modelos lineales (izquierda) y polinomiales (derecha), ambos utilizando varios niveles de regularización de lazo.

Una característica importante de la regresión con lazo es que tiende a eliminar los pesos de las características menos importantes (es decir, a ponerlos en cero). Por ejemplo, la línea discontinua en

el gráfico de la derecha en la Figura 4-18 (con  $\alpha = 0,01$ ) parece aproximadamente cúbico: todos los pesos de las características polinómicas de alto grado son iguales a cero. En otras palabras, la regresión de lazo realiza automáticamente la selección de características y genera un modelo disperso con pocos pesos de características distintos de cero.

Puede tener una idea de por qué es así al observar la Figura 4-19: los ejes representan dos parámetros del modelo y los contornos del fondo representan diferentes funciones de pérdida. En el gráfico superior izquierdo, los contornos representan la pérdida  $\ell_1 (|\theta_1| + |\theta_2|)$ , que cae linealmente a medida que te acercas a cualquier eje. Por ejemplo, si inicializa los parámetros del modelo en  $\theta_1 = 2$  y  $\theta_2 = 0,5$ , ejecutar el descenso de gradiente disminuirá ambos parámetros por igual (como lo representa la línea amarilla discontinua); por lo tanto,  $\theta_1$  llegará a 0 primero (ya que, para empezar, estaba más cerca de 0). Despues de eso, el descenso del gradiente descenderá por el canal hasta alcanzar  $\theta_2 = 0$  (con un poco de rebote, ya que los gradientes de  $\ell_1$  nunca se acercan a 0: son -1 o 1 para cada parámetro). En el gráfico superior derecho, los contornos representan la función de costos de la regresión de lazo (es decir, una función de costos de MSE más una pérdida de  $\ell_1$ ). Los pequeños círculos blancos muestran el camino que toma el descenso del gradiente para optimizar algunos parámetros del modelo que se inicializaron alrededor de  $\theta_1 = 0,25$  y  $\theta_2 = -1$ : observe una vez más cómo el camino alcanza rápidamente  $\theta_2 = 0$ , luego rueda por el canal y termina rebotando alrededor del óptimo global (representado por el cuadrado rojo). Si aumentamos  $\alpha$ , el óptimo global se movería hacia la izquierda a lo largo de la línea amarilla discontinua, mientras que si disminuimos  $\alpha$ , el óptimo global se movería hacia la derecha (en este ejemplo, los parámetros óptimos para el MSE no regularizado son  $\theta_1 = 2$  y  $\theta_2 = 0,5$ ).

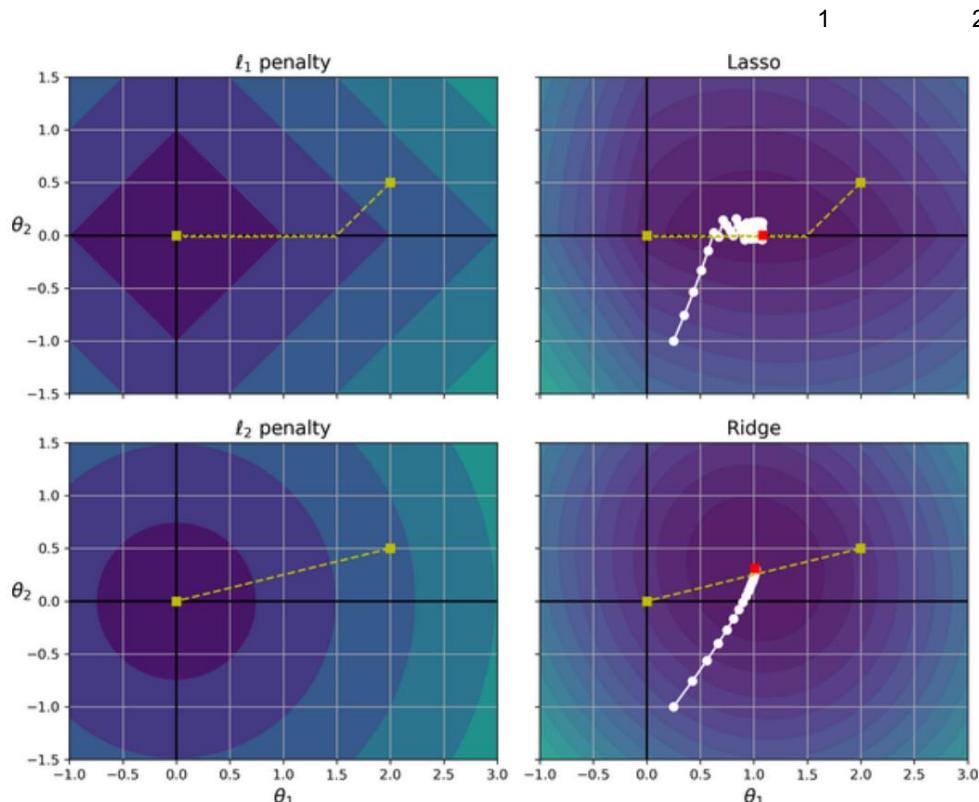


Figura 4-19. Regularización de lazo versus cresta

Los dos gráficos inferiores muestran lo mismo pero con una penalización de  $\ell_2$ . En el gráfico inferior izquierdo, puede ver que la pérdida de  $\ell_2$  disminuye a medida que nos acercamos al origen, por lo que el descenso del gradiente simplemente toma un camino recto hacia ese punto. En el gráfico inferior derecho, los contornos representan la función de costos de la regresión de cresta (es decir, una función de costos MSE más una pérdida  $\ell_2$ ). Como puede ver, los gradientes se vuelven más pequeños a medida que los parámetros se acercan al óptimo global, por lo que el descenso del gradiente naturalmente se ralentiza. Esto limita el rebote, lo que ayuda a que las crestas converjan más rápido que la regresión con lazo. También tenga en cuenta que los parámetros óptimos (representados por el cuadrado rojo) se acercan cada vez más al origen cuando aumenta  $\alpha$ , pero nunca se eliminan por completo.

## CONSEJO

Para evitar que el descenso de gradiente rebote alrededor del óptimo al final cuando se utiliza la regresión de lazo, es necesario reducir gradualmente la tasa de aprendizaje durante el entrenamiento. Seguirá rebotando alrededor del óptimo, pero los pasos serán cada vez más pequeños, por lo que convergerá.

La función de costo del lazo no es diferenciable en  $\theta_i = 0$  (para  $i = 1, 2, \dots, n$ ), pero el descenso de gradiente aún funciona si usa un vector de subgradiente  $g$  cuando cualquier  $\theta_i = 0$ .

**La ecuación 4-11** muestra una ecuación vectorial de subgradiente que puede utilizar para el descenso de gradiente con la función de costo de lazo.

Ecuación 4-11. Vector subgradiente de regresión de lazo

$$g(\theta, J) = \text{MSE}(\theta) + 2\alpha$$

signo ( $\theta_1$ )		$-1 \text{ si } \theta_1 < 0 \quad 0 \text{ si } \theta_1 = 0$
signo ( $\theta_2$ )		$+1 \text{ si } \theta_2 > 0$
donde signo ( $\theta_i$ ) =		$\theta_i = 0$
signo ( $\theta_n$ )		

Aquí hay un pequeño ejemplo de Scikit-Learn usando la clase Lasso:

```
>>> de sklearn.linear_model importar Lasso >>> lasso_reg =
Lasso(alpha=0.1) >>> lasso_reg.fit(X, y) >>>
lasso_reg.predict([[1.5]])
matriz([1.53788174])
```

Tenga en cuenta que, en su lugar, podría utilizar SGDRegressor(penalty="l1", alpha=0.1).

## Regresión neta elástica

La regresión neta elástica es un término medio entre la regresión de cresta y la regresión de lazo. El término de regularización es una suma ponderada de los términos de regularización de cresta y lazo, y puede controlar la proporción de mezcla  $r$ . Cuando  $r = 0$ , la red elástica es

equivalente a la regresión de cresta, y cuando  $r = 1$ , es equivalente a la regresión de lazo ([Ecuación 4-12](#)).

Ecuación 4-12. Función de costo neto elástico

$$J(\theta) = \text{MSE}(\theta) + r(2\alpha \sum_{i=1}^n |\theta_i|) + (1-r)(\alpha \overline{\sum_{i=1}^n y_{i0}} \sum_{i=1}^n y_{i0} - \theta_0)^2)$$

Entonces, ¿cuándo debería utilizar la regresión neta elástica, o la regresión lineal simple, de cresta o de lazo (es decir, sin ninguna regularización)? Casi siempre es preferible tener al menos un poco de regularización, por lo que generalmente se debe evitar la regresión lineal simple.

Ridge es un buen valor predeterminado, pero si sospecha que solo unas pocas funciones son útiles, debería preferir lazo o red elástica porque tienden a reducir el peso de las funciones inútiles a cero, como se analizó anteriormente. En general, se prefiere la red elástica al lazo porque el lazo puede comportarse de manera errática cuando el número de características es mayor que el número de instancias de entrenamiento o cuando varias características están fuertemente correlacionadas.

A continuación se muestra un breve ejemplo que utiliza ElasticNet de Scikit-Learn (`l1_ratio` corresponde a la relación de mezcla  $r$ ):

```
>>> de sklearn.linear_model importar ElasticNet >>> elastic_net
= ElasticNet(alpha=0.1, l1_ratio=0.5) >>> elastic_net.fit(X, y) >>>
elastic_net.predict([[1.5]])
matriz([1,54333232])
```

### Detención temprana Una

forma muy diferente de regularizar algoritmos de aprendizaje iterativo, como el descenso de gradiente, es detener el entrenamiento tan pronto como el error de validación alcance un mínimo. A esto se le llama parada anticipada. [La Figura 4-20](#) muestra un modelo complejo (en este caso, un modelo de regresión polinómica de alto grado) entrenado con descenso de gradiente por lotes en el conjunto de datos cuadrático que utilizamos anteriormente. A medida que pasan las épocas, el algoritmo aprende y su error de predicción (RMSE) en el conjunto de entrenamiento disminuye, junto con su error de predicción en el conjunto de validación. Sin embargo, después de un tiempo, el error de validación deja de disminuir y comienza a volver a aumentar. Esto indica que el modelo ha comenzado a sobreajustar los datos de entrenamiento. Con la parada anticipada, simplemente dejas de entrenar tan pronto como el error de validación alcance el mínimo. Es una técnica de regularización tan simple y eficiente que Geoffrey Hinton la llamó un “hermoso almuerzo gratis”.

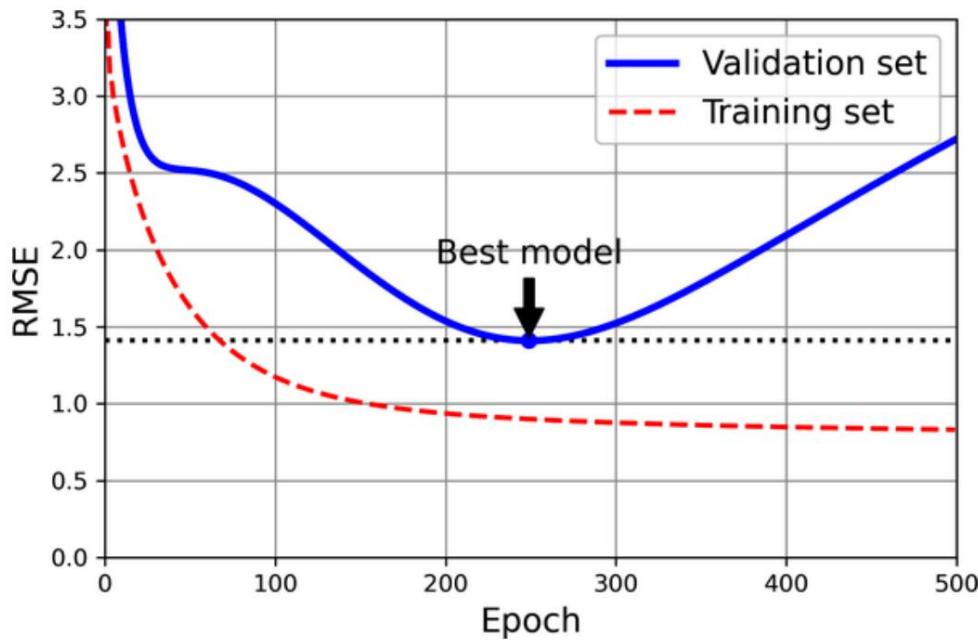


Figura 4-20. Regularización de parada anticipada

## CONSEJO

Con el descenso de gradiente estocástico y de mini lotes, las curvas no son tan suaves y puede resultar difícil saber si se ha alcanzado el mínimo o no. Una solución es detenerse solo después de que el error de validación haya estado por encima del mínimo durante algún tiempo (cuando esté seguro de que el modelo no funcionará mejor) y luego revertir los parámetros del modelo hasta el punto donde el error de validación fue mínimo..

A continuación se muestra una implementación básica de la parada anticipada:

```

desde copia importar copia profunda desde
sklearn.metrics importar mean_squared_error desde sklearn.preprocesamiento
importar StandardScaler

X_train, y_train, X_valid, y_valid = [...] # dividir el conjunto de datos cuadrático

preprocesamiento = make_pipeline(PolynomialFeatures(grado=90, include_bias=False),
                                 Escalador estándar())
X_train_prep = preprocesamiento.fit_transform(X_train)
X_valid_prep = preprocesamiento.transform(X_valid)
sgd_reg =
SGDRegressor(penalidad=Ninguna, eta0=0.002, random_state=42)
n_epochs = 500
best_valid_rmse =
float('inf')

para época en el rango (n_epochs):
    sgd_reg.partial_fit(X_train_prep, y_train)
    y_valid_predict =
    sgd_reg.predict(X_valid_prep)

```

```

    val_error = mean_squared_error(y_valid, y_valid_predict, squared=False) si val_error
< best_valid_rmse:
    best_valid_rmse = val_error best_model =
        deepcopy(sgd_reg)

```

Este código primero agrega las características polinómicas y escala todas las características de entrada, tanto para el conjunto de entrenamiento como para el conjunto de validación (el código supone que ha dividido el conjunto de entrenamiento original en un conjunto de entrenamiento más pequeño y un conjunto de validación). Luego crea un modelo SGDRegressor sin regularización y con una pequeña tasa de aprendizaje. En el ciclo de entrenamiento, llama a `partial_fit()` en lugar de `fit()`, para realizar un aprendizaje incremental. En cada época, mide el RMSE en el conjunto de validación. Si es inferior al RMSE más bajo visto hasta ahora, guarda una copia del modelo en la variable `best_model`. En realidad, esta implementación no detiene el entrenamiento, pero le permite volver al mejor modelo después del entrenamiento. Tenga en cuenta que el modelo se copia utilizando `copy.deepcopy()`, porque copia tanto los hiperparámetros del modelo como los parámetros aprendidos. Por el contrario, `sklearn.base.clone()` solo copia los hiperparámetros del modelo.

### Regresión logística

Como se

analizó en [el Capítulo 1](#), algunos algoritmos de regresión se pueden utilizar para la clasificación (y viceversa). La regresión logística (también llamada regresión logit) se usa comúnmente para estimar la probabilidad de que una instancia pertenezca a una clase particular (por ejemplo, ¿cuál es la probabilidad de que este correo electrónico sea spam?). Si la probabilidad estimada es mayor que un umbral determinado (normalmente 50%), entonces el modelo predice que la instancia pertenece a esa clase (llamada clase positiva, etiquetada como "1") y, en caso contrario, predice que no (es decir, pertenece a la clase negativa, denominada "0"). Esto lo convierte en un clasificador binario.

### Estimación de probabilidades

Entonces, ¿cómo funciona la regresión logística? Al igual que un modelo de regresión lineal, un modelo de regresión logística calcula una suma ponderada de las características de entrada (más un término de sesgo), pero en lugar de generar el resultado directamente como lo hace el modelo de regresión lineal, genera la logística de este resultado (ver [Ecuación 4-13](#)).

Ecuación 4-13. Probabilidad estimada del modelo de regresión logística (forma vectorizada)

$$\hat{p} = h(x) = \sigma(x)$$

La logística, denominada  $\sigma(\cdot)$ , es una función sigmoidea (es decir, en forma de S) que genera un número entre 0 y 1. Se define como se muestra en [la ecuación 4-14](#) y [la figura 4-21](#).

Ecuación 4-14. Función logística

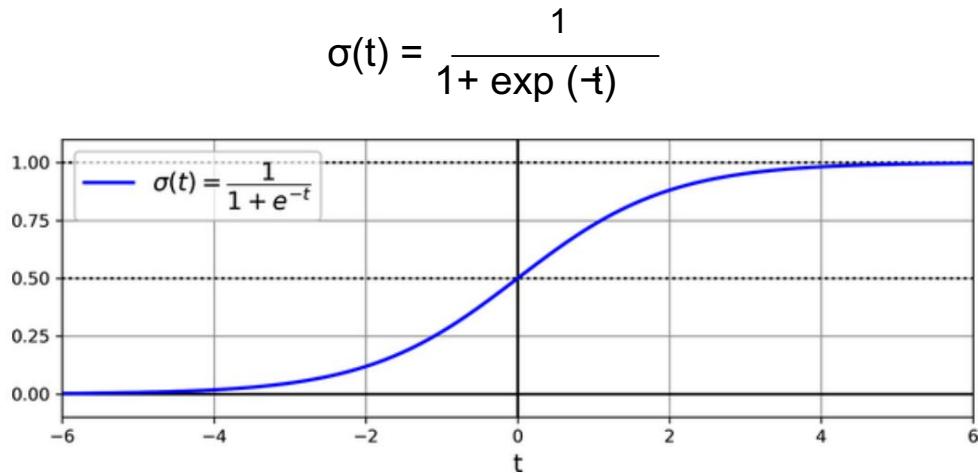


Figura 4-21. Función logística

Una vez que el modelo de regresión logística ha estimado la probabilidad  $\hat{p} = h(x)$  de que una instancia  $x$  pertenezca a la clase positiva, puede hacer su predicción  $\hat{y}$  fácilmente (ver [Ecuación 4-15](#)).

Ecuación 4-15. Predicción del modelo de regresión logística utilizando un umbral de probabilidad del 50%

$$\begin{aligned} & 0 \text{ si } \hat{p} < 0,5 \\ & \hat{y} = \{ 1 \text{ si } \hat{p} \geq 0,5 \end{aligned}$$

Observe que  $\sigma(t) < 0,5$  cuando  $t < 0$ , y  $\sigma(t) \geq 0,5$  cuando  $t \geq 0$ , por lo que un modelo de regresión logística que utiliza el umbral predeterminado del 50% de probabilidad predice 1 si  $\theta^T x$  es positivo y 0 si lo es negativo.

### NOTA

La puntuación  $t$  a menudo se denomina logit. El nombre proviene del hecho de que la función logit, definida como  $\text{logit}(p) = \log(p / (1 - p))$ , es la inversa de la función logística. De hecho, si calcula el logit de la probabilidad estimada  $p$ , encontrará que el resultado es  $t$ . El logit también se denomina log-odds, ya que es el logaritmo de la relación entre la probabilidad estimada para la clase positiva y la probabilidad estimada para la clase negativa.

### Función de entrenamiento y costos Ahora

ya sabe cómo un modelo de regresión logística estima probabilidades y hace predicciones. ¿Pero cómo se entrena? El objetivo del entrenamiento es establecer el vector de parámetros  $\theta$  de modo que el modelo estime altas probabilidades para instancias positivas ( $y = 1$ ) y bajas probabilidades para instancias negativas ( $y = 0$ ). Esta idea se captura en la función de costo que se muestra en [la ecuación 4-16](#) para una única instancia de entrenamiento  $x$ .

Ecuación 4-16. Función de costo de una sola instancia de capacitación

$$-\log(p^{\hat{}}) \text{ si } y = 1 -$$

$$c( ) = \begin{cases} -\log(1 - p^{\hat{}}) & \text{si } y = 0 \end{cases}$$

Esta función de costo tiene sentido porque  $-\log(t)$  crece mucho cuando  $t$  tiende a 0, por lo que el costo será grande si el modelo estima una probabilidad cercana a 0 para una instancia positiva, y también será grande si el modelo estima una probabilidad cercana a 0 para una instancia negativa. Probabilidad cercana a 1 para una instancia negativa. Por otro lado,  $-\log(t)$  es cercano a 0 cuando  $t$  es cercano a 1, por lo que el costo será cercano a 0 si la probabilidad estimada es cercana a 0 para una instancia negativa o cercana a 1 para una instancia positiva. que es precisamente lo que queremos.

La función de costo de todo el conjunto de capacitación es el costo promedio de todas las instancias de capacitación. Puede escribirse en una sola expresión llamada pérdida logarítmica, que se muestra en [la ecuación 4-17](#).

Ecuación 4-17. Función de costo de regresión logística (pérdida logarítmica)

$$J( ) = - \frac{1}{\text{metro}} \sum_{i=1}^m [y(i)\log(p^{\hat{}}(i)) + (1 - y(i))\log(1 - p^{\hat{}}(i))]$$

#### ADVERTENCIA

La pérdida de troncos no surgió de la casualidad. Se puede demostrar matemáticamente (usando inferencia bayesiana) que minimizar esta pérdida dará como resultado el modelo con la máxima probabilidad de ser óptimo, asumiendo que las instancias siguen una distribución gaussiana alrededor de la media de su clase. Cuando utiliza la pérdida logarítmica, esta es la suposición implícita que está haciendo. Cuanto más errónea sea esta suposición, más sesgado será el modelo. De manera similar, cuando utilizamos el MSE para entrenar modelos de regresión lineal, asumimos implícitamente que los datos eran puramente lineales, además de algo de ruido gaussiano. Entonces, si los datos no son lineales (por ejemplo, si son cuadráticos) o si el ruido no es gaussiano (por ejemplo, si los valores atípicos no son exponencialmente raros), entonces el modelo estará sesgado.

La mala noticia es que no existe ninguna ecuación de forma cerrada conocida para calcular el valor de  $\theta$  que minimice esta función de costos (no existe un equivalente de la ecuación normal). Pero la buena noticia es que esta función de costo es convexa, por lo que se garantiza que el descenso de gradiente (o cualquier otro algoritmo de optimización) encontrará el mínimo global (si la tasa de aprendizaje no es demasiado grande y se espera lo suficiente). Las derivadas parciales de la función de costos con respecto al parámetro  $\theta$  del modelo  $j$  vienen dadas por [la Ecuación 4-18](#).

Ecuación 4-18. Derivadas parciales de la función de costos logísticos

$$\frac{\partial}{\partial \theta_j} J( ) = \frac{1}{\text{metro}} \sum_{i=1}^m (\sigma(x(i)) - y(i)) x_j^{(i)}$$

Esta ecuación se parece mucho a [la ecuación 4-5](#): para cada instancia calcula el error de predicción y lo multiplica por el valor de la característica  $j$ , y luego calcula el

promedio de todas las instancias de capacitación. Una vez que tenga el vector de gradiente que contiene todos las derivadas parciales, puede usarlas en el algoritmo de descenso de gradiente por lotes. Eso es todo: Ahora sabes cómo entrenar un modelo de regresión logística. Para GD estocástico usted tome una instancia a la vez, y para GD de mini lotes usaría un mini lote a la vez. tiempo.

## Límites de decisión

Podemos utilizar el conjunto de datos del iris para ilustrar la regresión logística. Este es un conjunto de datos famoso. que contiene el largo y ancho del sépalo y pétalo de 150 flores de iris de tres diferentes especies: Iris setosa, Iris versicolor e Iris virginica (ver Figura 4-22).



Figura 4-22. Flores de tres especies de plantas de iris.<sup>12</sup>

Intentemos construir un clasificador para detectar el tipo Iris virginica basado solo en el pétalo. característica de ancho. El primer paso es cargar los datos y echar un vistazo rápido:

```
>>> desde sklearn.datasets importar load_iris
>>> iris = cargar_iris(as_frame=True)
>>> lista(iris)
['datos', 'objetivo', 'marco', 'nombres_objetivo', 'DESCR', 'nombres_características',
 'nombre de archivo', 'módulo_datos']
>>> iris.datos.cabeza(3)
      largo del sépalo (cm) ancho del sépalo (cm) largo del pétalo (cm) ancho del pétalo
      (cm)
0 0.2 5.1 3.5 1.4
1 0.2 4.9 3.0 1.4
2 0.2 4.7 3.2 1.3
>>> iris.target.head(3) # tenga en cuenta que las instancias no se mezclan
0    0
```

```

1      0
2      0
Nombre: objetivo, dtype: int64 >>>
iris.target_names array(['setosa',
'versicolor', 'virginica'], dtype='|<U10')

```

A continuación dividiremos los datos y entrenaremos un modelo de regresión logística en el conjunto de entrenamiento:

```

de sklearn.linear_model importar LogisticRegression de sklearn.model_selection
importar train_test_split

X = iris.data[[" ancho del pétalo (cm)"]].valores y =
iris.target_names[iris.target] == 'virginica'
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

log_reg = Regresión logística(estado_aleatorio=42) log_reg.fit(X_train,
y_train)

```

Veamos las probabilidades estimadas del modelo para flores con anchos de pétalos que varían de 0 cm a 3 cm (Figura 4-23):

13

```

X_new = np.linspace(0, 3, 1000).reshape(-1, 1) # remodelar para obtener un vector de columna

y_proba = log_reg.predict_proba(X_new) decision_boundary
= X_new[y_proba[:, 1] >= 0.5][0, 0]

plt.plot(X_new, y_proba[:, 0], "b-", ancho de línea=2, label="Not Iris virginica
proba") plt.plot(X_new, y_proba[:, 1], "g-", ancho
de línea=2, etiqueta="Iris virginica proba") plt.plot([límite_decision, límite_decision], [0, 1], "k:", ancho de línea=2,
etiqueta="Límite de decisión")
[...] # embellecer la figura: agregar cuadrícula, etiquetas, eje, leyenda, flechas y muestras plt.show()

```

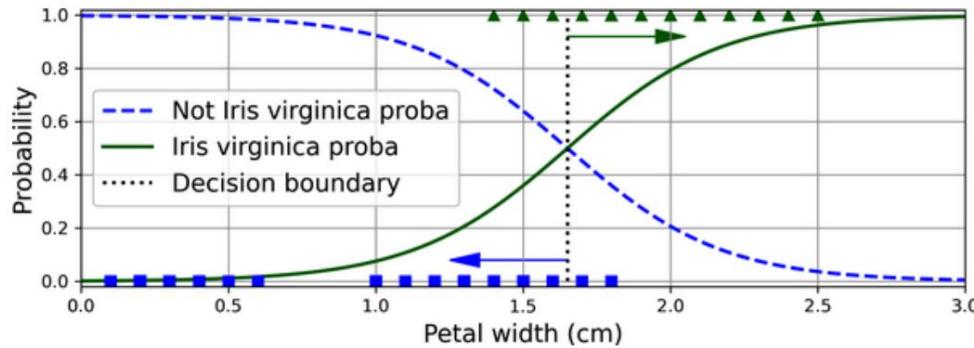


Figura 4-23. Probabilidades estimadas y límite de decisión.

El ancho de los pétalos de las flores de Iris virginica (representadas como triángulos) varía de 1,4 cm a 2,5 cm, mientras que las otras flores de iris (representadas por cuadrados) generalmente tienen un ancho de pétalos más pequeño, que oscila entre 0,1 cm y 1,8 cm. Observe que hay un poco de superposición. Por encima de aproximadamente 2 cm, el clasificador tiene mucha confianza en que la flor es Iris virginica (da una alta probabilidad para esa clase), mientras que por debajo de 1 cm tiene mucha confianza en que no es una Iris virginica (alta probabilidad para la clase "No Iris"). Entre estos extremos, el clasificador no está seguro. Sin embargo, si le pide que prediga la clase (usando el método predict()) en lugar del método predict\_proba(), devolverá la clase que sea más probable. Por lo tanto, existe un límite de decisión en torno a 1,6 cm donde ambas probabilidades son iguales al 50%: si el ancho del pétalo es mayor que 1,6 cm el clasificador predecirá que la flor es una Iris virginica, y en caso contrario predecirá que no lo es. (incluso si no tiene mucha confianza):

```
>>> límite_decision
1.6516516516516517 >>>
log_reg.predict([[1.7], [1.5]]) matriz([ Verdadero, Falso])
```

**La Figura 4-24** muestra el mismo conjunto de datos, pero esta vez con dos características: ancho y largo de los pétalos. Una vez entrenado, el clasificador de regresión logística puede, basándose en estas dos características, estimar la probabilidad de que una nueva flor sea una Iris virginica. La línea discontinua representa los puntos donde el modelo estima una probabilidad del 50%: este es el límite de decisión del modelo. Tenga en cuenta que es un límite lineal. Cada línea paralela <sup>14</sup> a la discontinua representa los puntos donde el modelo genera una probabilidad específica, desde el 15% (abajo a la izquierda) hasta el 90% (arriba a la derecha). Según el modelo, todas las flores más allá de la línea superior derecha tienen más del 90% de posibilidades de ser Iris virginica .

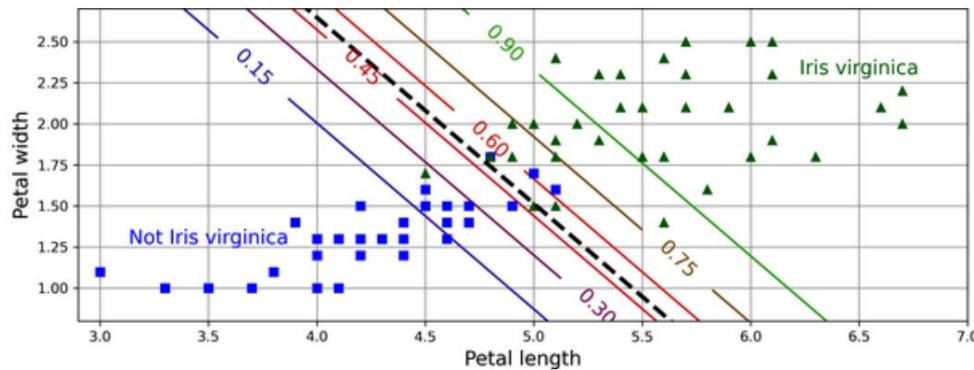


Figura 4-24. Límite de decisión lineal

### NOTA

El hiperparámetro que controla la fuerza de regularización de un modelo Scikit-Learn LogisticRegression no es alfa (como en otros modelos lineales), sino su inverso: C. Cuanto mayor sea el valor de C, menos regularizado estará el modelo.

Al igual que los otros modelos lineales, los modelos de regresión logística se pueden regularizar utilizando penalizaciones de  $\ell_1$  o  $\ell_2$ . Scikit-Learn en realidad agrega una penalización de  $\ell_2$  de forma predeterminada.

## Regresión Softmax

El modelo de regresión logística se puede generalizar para admitir múltiples clases directamente, sin tener que entrenar y combinar múltiples clasificadores binarios (como se analiza en el [Capítulo 3](#)). Esto se llama regresión softmax o regresión logística multinomial.

La idea es simple: cuando se le da una instancia  $x$ , el modelo de regresión softmax primero calcula una puntuación  $s(x)$  para cada clase  $k$ , luego estima la probabilidad de cada clase aplicando la función softmax (también llamada exponencial normalizada) a las puntuaciones. La ecuación para calcular  $s(x)$  debería resultar familiar, ya que es igual a la ecuación para la predicción de regresión lineal (consulte [la ecuación 4-19](#)).

Ecuación 4-19. Puntuación Softmax para la clase  $k$ .

$$s_k(x) = \sigma(s(x))$$

Tenga en cuenta que cada clase tiene su propio vector de parámetros dedicado  $\theta_k$ . Todos estos vectores normalmente se almacenan como filas en una matriz de parámetros  $\Theta$

Una vez que haya calculado la puntuación de cada clase para la instancia  $x$ , puede estimar la probabilidad  $p_k^*$  de que la instancia pertenezca a la clase  $k$  ejecutando las puntuaciones a través de la función softmax ([Ecuación 4-20](#)). La función calcula el exponencial de cada puntuación y luego los normaliza (dividiendo por la suma de todos los exponentiales). Las puntuaciones generalmente se denominan logits o log-odds (aunque en realidad son log-odds no normalizadas).

Ecuación 4-20. función softmax

$$p_k^* = \sigma(s(x)) = \frac{\exp(s_k(x))}{\sum_{j=1}^K \exp(s_j(x))}$$

En esta ecuación:

- $K$  es el número de clases.
- $s(x)$  es un vector que contiene las puntuaciones de cada clase para la instancia  $x$ .
- $\sigma(s(x))$  es la probabilidad estimada de que la instancia  $x$  pertenezca a la clase  $k$ , dadas las puntuaciones de cada clase para esa instancia.

Al igual que el clasificador de regresión logística, de forma predeterminada el clasificador de regresión softmax predice la clase con la probabilidad estimada más alta (que es simplemente la clase con

la puntuación más alta), como se muestra en la Ecuación 4-21.

Ecuación 4-21. Predicción del clasificador de regresión Softmax

$$k = \operatorname{argmax}_k \sigma(s(x)) \quad s_k(x) = \operatorname{argmax}_k y^{\hat{}} = \operatorname{argmax}_k ((k)) \quad X$$

El operador  $\operatorname{argmax}$  devuelve el valor de una variable que maximiza una función. En esta ecuación, devuelve el valor de  $k$  que maximiza la probabilidad estimada  $\sigma(s(x))$ .

#### CONSEJO

El clasificador de regresión softmax predice sólo una clase a la vez (es decir, es multiclase, no multisalida), por lo que debe usarse sólo con clases mutuamente excluyentes, como diferentes especies de plantas. No puedes usarlo para reconocer a varias personas en una imagen.

Ahora que sabes cómo el modelo estima probabilidades y hace predicciones, echemos un vistazo al entrenamiento. El objetivo es tener un modelo que estime una probabilidad alta para la clase objetivo (y en consecuencia una probabilidad baja para las otras clases). Minimizar la función de costos que se muestra en la ecuación 4-22, llamada entropía cruzada, debería conducir a este objetivo porque penaliza al modelo cuando estima una probabilidad baja para una clase objetivo. La entropía cruzada se utiliza con frecuencia para medir qué tan bien coincide un conjunto de probabilidades de clase estimadas con las clases objetivo.

Ecuación 4-22. Función de costo de entropía cruzada

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^{K_{\text{clases}}} y_i^{(k)} \log(p_i^{(k)})$$

En esta ecuación,  $y_i^{(k)}$  es la probabilidad objetivo de que la instancia  $i$  pertenezca a la clase  $k$ . En general, es igual a 1 o 0, dependiendo de si la instancia pertenece a la clase o no.

Observe que cuando hay sólo dos clases ( $K = 2$ ), esta función de costos es equivalente a la función de costos de regresión logística (pérdida logarítmica; consulte la ecuación 4-17).

## ENTROPÍA CRUZADA

La entropía cruzada se originó a partir de la teoría de la información de Claude Shannon . Supongamos que desea transmitir de manera eficiente información sobre el clima todos los días. Si hay ocho opciones (soleado, lluvioso, etc.), podrías codificar cada opción usando 3 bits, porque  $2^3 = 8$ . Sin embargo, si crees que hará sol casi todos los días, sería mucho más eficiente codificar "sunny" en solo un bit (0) y las otras siete opciones en cuatro bits (comenzando con un 1). La entropía cruzada mide la cantidad promedio de bits que realmente envía por opción. Si su suposición sobre el clima es perfecta, la entropía cruzada será igual a la entropía del clima mismo (es decir, su imprevisibilidad intrínseca). Pero si su suposición es errónea (por ejemplo, si llueve con frecuencia), la entropía cruzada será mayor en una cantidad llamada divergencia Kullback-Leibler (KL).

La entropía cruzada entre dos distribuciones de probabilidad  $p$  y  $q$  se define como  $H(p,q) = -\sum p(x) \log q(x)$  (al menos cuando las distribuciones son discretas). Para más detalles, mira [mi video sobre el tema](#).

El vector gradiente de esta función de costos con respecto a  $\theta$  viene dado por [la Ecuación 4-23](#).

Ecuación 4-23. Vector de gradiente de entropía cruzada para la clase  $k$

$$(k) J(\Theta) = \frac{1}{m} \sum_{i=1}^m (p^{(i)} - y^{(i)})x^{(i)}$$

Ahora puede calcular el vector de gradiente para cada clase y luego usar el descenso de gradiente (o cualquier otro algoritmo de optimización) para encontrar la matriz de parámetros  $\Theta$  que minimiza la función de costo.

Usemos la regresión softmax para clasificar las plantas de iris en las tres clases. El clasificador LogisticRegression de Scikit-Learn usa la regresión softmax automáticamente cuando lo entrenas en más de dos clases (suponiendo que uses solver="lbfgs", que es el valor predeterminado). También aplica la regularización  $\ell_2$  de forma predeterminada, que puedes controlar usando el hiperparámetro C, como se mencionó anteriormente:

```
X = iris.data[["largo del pétalo (cm)", "ancho del pétalo (cm)"]].valores y = iris["objetivo"]
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
```

```
softmax_reg = Regresión logística(C=30, estado_aleatorio=42) softmax_reg.fit(X_train, y_train)
```

Entonces, la próxima vez que encuentres un iris con pétalos de 5 cm de largo y 2 cm de ancho, puedes pedirle a tu modelo que te diga qué tipo de iris es y te responderá Iris virginica (clase 2) con un 96% de probabilidad (o Iris versicolor con 4% de probabilidad):

```
>>> softmax_reg.predict([[5, 2]]) matriz([2])
>>>
softmax_reg.predict_proba([[5, 2]]).round(2) matriz([[0. , 0.04,
0.96]])
```

La Figura 4-25 muestra los límites de decisión resultantes, representados por los colores de fondo. Observe que los límites de decisión entre dos clases cualesquiera son lineales. La figura también muestra las probabilidades para la clase Iris versicolor, representada por las líneas curvas (por ejemplo, la línea etiquetada con 0,30 representa el límite de probabilidad del 30%).

Observe que el modelo puede predecir una clase que tiene una probabilidad estimada inferior al 50%. Por ejemplo, en el punto donde se encuentran todos los límites de decisión, todas las clases tienen una probabilidad estimada igual del 33%.

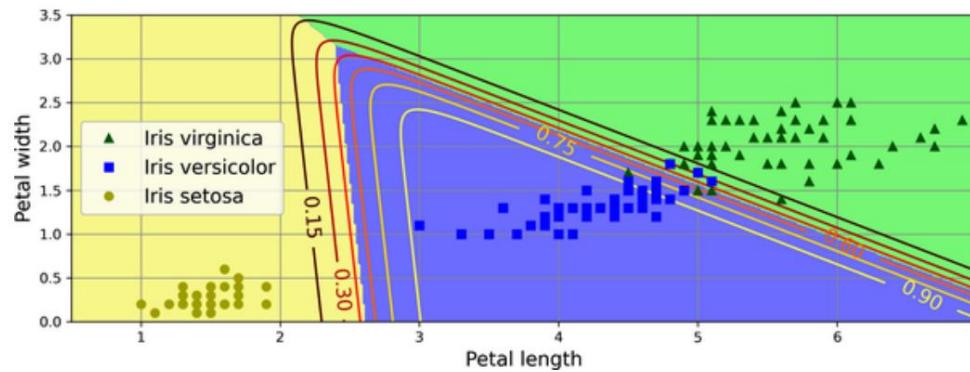


Figura 4-25. Límites de decisión de regresión Softmax

En este capítulo, aprendió varias formas de entrenar modelos lineales, tanto para regresión como para clasificación. Utilizó una ecuación de forma cerrada para resolver la regresión lineal, así como el descenso de gradiente, y aprendió cómo se pueden agregar varias penalizaciones a la función de costos durante el entrenamiento para regularizar el modelo. A lo largo del camino, también aprendió a trazar curvas de aprendizaje y analizarlas, y a implementar una parada anticipada.

Finalmente, aprendió cómo funcionan la regresión logística y la regresión softmax. ¡Hemos abierto las primeras cajas negras de aprendizaje automático! En los próximos capítulos abriremos muchos más, comenzando con las máquinas de vectores de soporte.

## Ejercicios

1. ¿Qué algoritmo de entrenamiento de regresión lineal puedes usar si tienes un conjunto de entrenamiento con millones de funciones?
2. Suponga que las funciones de su conjunto de entrenamiento tienen escalas muy diferentes. ¿Qué algoritmos podrían verse afectados por esto y cómo? ¿Qué puedes hacer al respecto?
3. ¿Puede el descenso de gradiente quedarse atascado en un mínimo local al entrenar una logística? ¿Modelo de regresión?

4. ¿Todos los algoritmos de descenso de gradiente conducen al mismo modelo, siempre que los dejes funcionar durante el tiempo suficiente?
  5. Suponga que utiliza el descenso de gradiente por lotes y traza el error de validación en cada época. Si nota que el error de validación aumenta constantemente, ¿qué es probable que esté pasando? ¿Cómo puedes arreglar esto?
  6. ¿Es una buena idea detener el descenso del gradiente del mini lote inmediatamente cuando el error de validación aumenta?
  7. ¿Qué algoritmo de descenso de gradiente (entre los que discutimos) alcanzará el la vecindad de la solución óptima es la más rápida? ¿Cuáles realmente convergerán? ¿Cómo puedes hacer que los demás también converjan?
  8. Suponga que está utilizando regresión polinómica. Trazas las curvas de aprendizaje y notas que hay una gran brecha entre el error de entrenamiento y el error de validación. ¿Lo que está sucediendo? ¿Cuáles son las tres formas de resolver esto?
  9. Suponga que está utilizando la regresión de crestas y observa que el error de entrenamiento y el error de validación son casi iguales y bastante altos. ¿Diría que el modelo sufre de un alto sesgo o una alta varianza? ¿Debería aumentar el hiperparámetro de regularización  $\alpha$  o reducirlo?
10. ¿Por qué querrías usar:
- a. ¿Regresión de cresta en lugar de regresión lineal simple (es decir, sin ninguna regularización)?
  - b. ¿Lazo en lugar de regresión de crestas?
  - c. ¿Red elástica en lugar de regresión con lazo?
11. Suponga que desea clasificar las imágenes como exteriores/internos y diurnas/nocturnas. ¿Debería implementar dos clasificadores de regresión logística o un clasificador de regresión softmax?
12. Implementar el descenso de gradiente por lotes con parada temprana para la regresión softmax sin usar Scikit-Learn, solo NumPy. Úselo en una tarea de clasificación como el conjunto de datos del iris.

Las soluciones a estos ejercicios están disponibles al final del cuaderno de este capítulo, en <https://homl.info/colab3>.

---

<sup>1</sup> Una ecuación de forma cerrada sólo se compone de un número finito de constantes, variables y operaciones estándar: por ejemplo,  $a = \sin(b - c)$ . Sin sumas infinitas, sin límites, sin integrales, etc.

<sup>2</sup> Técnicamente hablando, su derivada es la continua de Lipschitz.

- 3 Dado que la característica 1 es más pequeña, se necesita un cambio mayor en  $\theta$  para afectar la función de costos, razón por la cual el cuenco se alarga a lo largo del eje  $\theta$ . 1
- 4 Eta ( $\eta$ ) es la séptima letra del alfabeto griego.
- 5 Mientras que la ecuación normal solo puede realizar regresión lineal, los algoritmos de descenso de gradiente se puede utilizar para entrenar muchos otros modelos, como verá.
- 6 Esta noción de sesgo no debe confundirse con el término de sesgo de los modelos lineales.
- 7 Es común utilizar la notación  $J(\theta)$  para funciones de costos que no tienen un nombre corto; A menudo usaré esta notación a lo largo del resto de este libro. El contexto dejará claro qué función de costos se está discutiendo.
- 8 Las normas se analizan en [el Capítulo 2](#).
- 9 Una matriz cuadrada llena de ceros excepto unos en la diagonal principal (de arriba a la izquierda a abajo a la derecha).
- 10 Alternativamente, puede utilizar la clase Ridge con el solucionador de "hundimiento". El GD promedio estocástico es un Variante del GD estocástico. Para obtener más detalles, consulte la presentación "[Minimización de sumas finitas con el algoritmo de gradiente promedio estocástico](#)" de Mark Schmidt et al. de la Universidad de Columbia Británica.
- 11 Puedes pensar en un vector subgradiente en un punto no diferenciable como un vector intermedio entre los vectores de gradiente alrededor de ese punto.
- 12 fotografías reproducidas de las páginas correspondientes de Wikipedia. Fotografía de Iris virginica de Frank Mayfield ([Creative Commons BY-SA 2.0](#)), Fotografía de iris versicolor de D. Gordon E. Robertson ([Creative Commons BY-SA 3.0](#)), Foto de Iris setosa de dominio público.
- 13 La función reshape() de NumPy permite que una dimensión sea -1, lo que significa "automático":  
El valor se infiere de la longitud de la matriz y las dimensiones restantes.
- 14 Es el conjunto de puntos  $x$  tales que  $\theta_0 + \theta_1 x_1 + \theta_2 x_2 = 0$ , que define una recta.

# Capítulo 5. Máquinas de vectores de soporte

Una máquina de vectores de soporte (SVM) es un modelo de aprendizaje automático potente y versátil, capaz de realizar clasificación, regresión e incluso detección de novedades lineales o no lineales. Las SVM brillan con conjuntos de datos no lineales de tamaño pequeño y mediano (es decir, de cientos a miles de instancias), especialmente para tareas de clasificación. Sin embargo, como verá, no se adaptan muy bien a conjuntos de datos muy grandes.

Este capítulo explicará los conceptos básicos de las SVM, cómo usarlas y cómo funcionan. ¡Vamos a entrar!

## Clasificación SVM lineal

La idea fundamental detrás de las SVM se explica mejor con algunas imágenes. La Figura 5-1 muestra parte del conjunto de datos de iris que se presentó al final del Capítulo 4. Las dos clases se pueden separar fácilmente con una línea recta (son linealmente separables). El gráfico de la izquierda muestra los límites de decisión de tres posibles clasificadores lineales. El modelo cuyo límite de decisión está representado por la línea discontinua es tan malo que ni siquiera separa las clases adecuadamente. Los otros dos modelos funcionan perfectamente en este conjunto de entrenamiento, pero sus límites de decisión se acercan tanto a las instancias que estos modelos probablemente no funcionarán tan bien en instancias nuevas. Por el contrario, la línea continua en el gráfico de la derecha representa el límite de decisión de un clasificador SVM; esta línea no sólo separa las dos clases sino que también se mantiene lo más alejada posible de las instancias de entrenamiento más cercanas posible. Puede pensar que un clasificador SVM se ajusta a la calle más ancha posible (representada por líneas discontinuas paralelas) entre las clases. A esto se le llama clasificación de gran margen.

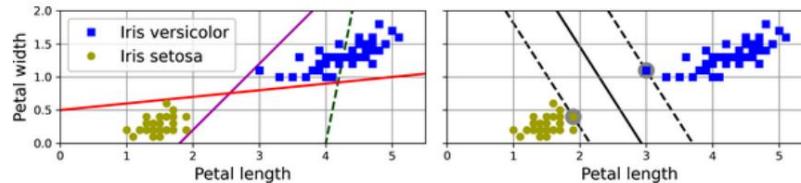


Figura 5-1. Clasificación de gran margen

Tenga en cuenta que agregar más instancias de entrenamiento "fuera de la calle" no afectará en absoluto el límite de decisión: está completamente determinado (o "respaldado") por las instancias ubicadas en el borde de la calle. Estos casos se denominan vectores de soporte (están rodeados por un círculo en la Figura 5-1).

### ADVERTENCIA

Las SVM son sensibles a las escalas de características, como puede ver en la Figura 5-2. En el gráfico de la izquierda, la escala vertical es mucho mayor que la escala horizontal, por lo que la calle más ancha posible está cerca de la horizontal. Después del escalado de características (por ejemplo, usando StandardScaler de Scikit-Learn), el límite de decisión en el gráfico correcto se ve mucho mejor.

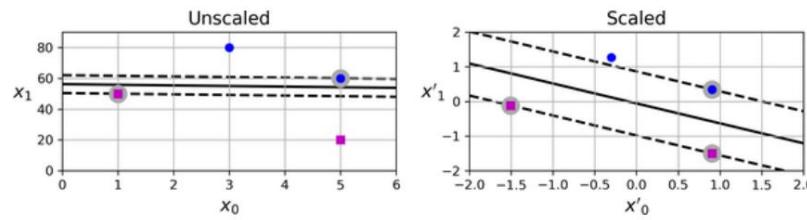


Figura 5-2. Sensibilidad a las escalas de características.

## Clasificación de margen blando

Si imponemos estrictamente que todas las instancias deben estar fuera de la calle y en el lado correcto, esto se denomina clasificación de margen estricto. Hay dos problemas principales con la clasificación de margen estricto. En primer lugar, sólo funciona si los datos son linealmente separables. En segundo lugar, es sensible a los valores atípicos. La Figura 5-3 muestra el conjunto de datos del iris con sólo un valor atípico adicional: a la izquierda, es imposible encontrar un margen definido; a la derecha, el límite de decisión termina siendo muy diferente del que vimos en la Figura 5-1 sin el valor atípico, y el modelo probablemente tampoco se generalizará.

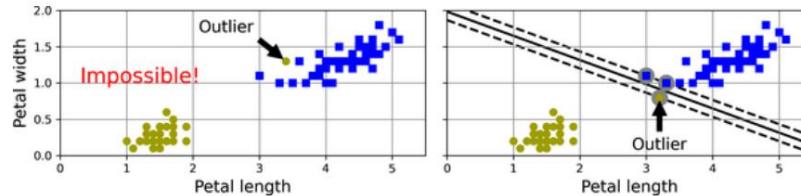


Figura 5-3. Sensibilidad de margen duro a valores atípicos

Para evitar estos problemas, necesitamos utilizar un modelo más flexible. El objetivo es encontrar un buen equilibrio entre mantener la calle lo más grande posible y limitar las violaciones de los márgenes (es decir, instancias que terminan en el medio de la calle o incluso en el lado equivocado). Esto se llama clasificación de margen suave.

Al crear un modelo SVM usando Scikit-Learn, puede especificar varios hiperparámetros, incluido el hiperparámetro de regularización C. Si lo establece en un valor bajo, terminará con el modelo a la izquierda de la Figura 5-4 . Con un valor alto, obtienes el modelo de la derecha. Como puede ver, reducir C hace que la calle sea más grande, pero también genera más violaciones de márgenes. En otras palabras, reducir C da como resultado que más instancias apoyen la calle, por lo que hay menos riesgo de sobreajuste. Pero si lo reduce demasiado, entonces el modelo termina por no ajustarse lo suficiente, como parece ser el caso aquí: parece que el modelo con C=100 se generalizará mejor que el modelo con C=1.

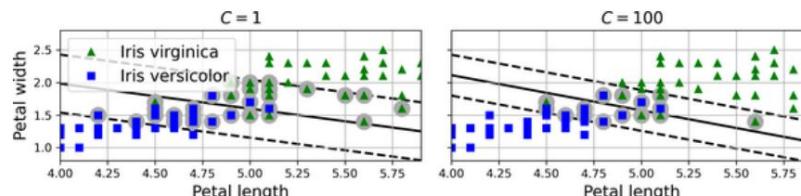


Figura 5-4. Gran margen (izquierda) versus menos violaciones de margen (derecha)

### CONSEJO

Si su modelo SVM está sobreajustado, puede intentar regularizarlo reduciendo C.

El siguiente código Scikit-Learn carga el conjunto de datos de iris y entrena un clasificador SVM lineal para detectar flores de Iris virginica . La canalización primero escala las características y luego usa un LinearSVC con C=1:

```
desde sklearn.datasets importar load_iris desde
sklearn.pipeline importar make_pipeline desde
sklearn.preprocessing importar StandardScaler desde sklearn.svm
importar LinearSVC

iris = cargar_iris(as_frame=True)
X = iris.data[["largo del pétalo (cm)", "ancho del pétalo (cm)"]].valores y = (iris.target == 2) # Iris
virginica

svm_clf = make_pipeline(StandardScaler(),
```

```
LinealSVC(C=1, estado_aleatorio=42))
svm_clf.fit(X, y)
```

El modelo resultante se representa a la izquierda en la [Figura 5-4](#).

Luego, como de costumbre, puedes usar el modelo para hacer predicciones:

```
>>> X_new = [[5.5, 1.7], [5.0, 1.5]] >>>
svm_clf.predict(X_new)
[[ Verdadero, Falso]]
```

La primera planta está clasificada como Iris virginica, mientras que la segunda no. Veamos las puntuaciones que utilizó SVM para hacer estas predicciones. Estos miden la distancia con signo entre cada instancia y el límite de decisión:

```
>>> matriz svm_clf.decision_function(X_new)
[[ 0.66163411, -0.22036063]]
```

A diferencia de LogisticRegression, LinearSVC no tiene un método predict\_proba() para estimar las probabilidades de clase. Dicho esto, si usa la clase SVC (que se analiza en breve) en lugar de LinearSVC, y si establece su hiperparámetro de probabilidad en Verdadero, entonces el modelo se ajustará a un modelo adicional al final del entrenamiento para asignar las puntuaciones de la función de decisión SVM a las probabilidades estimadas. . Básicamente, esto requiere utilizar una validación cruzada quíntuple para generar predicciones fuera de la muestra para cada instancia en el conjunto de entrenamiento y luego entrenar un modelo de Regresión Logística, por lo que ralentizará considerablemente el entrenamiento.

Después de eso, los métodos predict\_proba() y predict\_log\_proba() estarán disponibles.

## Clasificación SVM no lineal

Aunque los clasificadores SVM lineales son eficientes y a menudo funcionan sorprendentemente bien, muchos conjuntos de datos ni siquiera están cerca de ser linealmente separables. Un enfoque para manejar conjuntos de datos no lineales es agregar más características, como características polinómicas (como hicimos en el [Capítulo 4](#)); en algunos casos, esto puede dar como resultado un conjunto de datos linealmente separable. Considere el gráfico de la izquierda en la [Figura 5-5](#): representa un conjunto de datos simple con una sola característica,  $x$ . Este conjunto de datos no es linealmente separable, como puede ver. Pero si agrega una segunda característica  $x = (x_1, x_2)$ , el conjunto de datos 2D resultante es perfectamente separable linealmente.

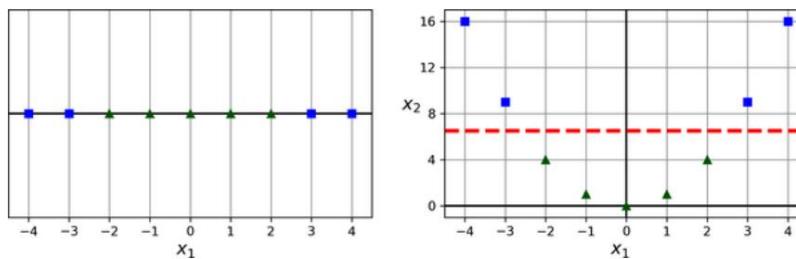


Figura 5-5. Agregar características para hacer que un conjunto de datos sea linealmente separable

Para implementar esta idea usando Scikit-Learn, puede crear una canalización que contenga un transformador PolynomialFeatures (que se analiza en [“Regresión polinomial”](#)), seguido de un StandardScaler y un clasificador LinearSVC. Probemos esto en el conjunto de datos de las lunas, un conjunto de datos de juguete para clasificación binaria en el que los puntos de datos tienen la forma de dos lunas crecientes entrelazadas (consulte la [Figura 5-6](#)). Puedes generar este conjunto de datos usando la función make\_moons():

```
desde sklearn.datasets importar make_moons desde
sklearn.preprocessing importar PolynomialFeatures
```

```
X, y = hacer_lunas(n_muestras=100, ruido=0.15, estado_aleatorio=42)
```

```

polinomio_svm_clf = make_pipeline(
    Características polinómicas (grado = 3),
    Escalador estándar(),
    LinealSVC(C=10, max_iter=10_000, random_state=42)

) polinomio_svm_clf.fit(X, y)

```

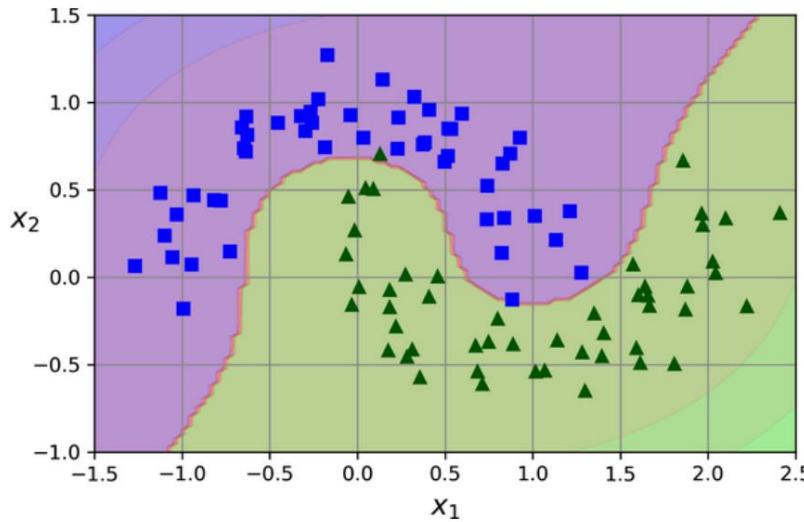


Figura 5-6. Clasificador SVM lineal que utiliza características polinómicas

## Núcleo polinomial

Agregar características polinómicas es fácil de implementar y puede funcionar muy bien con todo tipo de algoritmos de aprendizaje automático (no solo con SVM). Dicho esto, con un grado polinomial bajo, este método no puede manejar conjuntos de datos muy complejos, y con un grado polinomial alto crea una gran cantidad de características, lo que hace que el modelo sea demasiado lento.

Afortunadamente, cuando se utilizan SVM se puede aplicar una técnica matemática casi milagrosa llamada truco del núcleo (que se explica más adelante en este capítulo). El truco del kernel permite obtener el mismo resultado que si hubiera agregado muchas características polinómicas, incluso con un grado muy alto, sin tener que agregarlas realmente. Esto significa que no hay una explosión combinatoria del número de funciones. Este truco lo implementa la clase SVC. Probémoslo en el conjunto de datos de las lunas:

```

desde sklearn.svm importar SVC

poly_kernel_svm_clf = make_pipeline(StandardScaler(),
                                    SVC(núcleo="poli", grado=3, coef0=1, C=5))

poly_kernel_svm_clf.fit(X, y)

```

Este código entrena un clasificador SVM utilizando un núcleo polinomial de tercer grado, representado a la izquierda en la **Figura 5-7**. A la derecha hay otro clasificador SVM que utiliza un núcleo polinomial de décimo grado. Obviamente, si su modelo está sobreajustado, es posible que desee reducir el grado del polinomio. Por el contrario, si no es suficiente, puedes intentar aumentarlo. El hiperparámetro `coef0` controla en qué medida el modelo está influenciado por términos de alto grado versus términos de bajo grado.

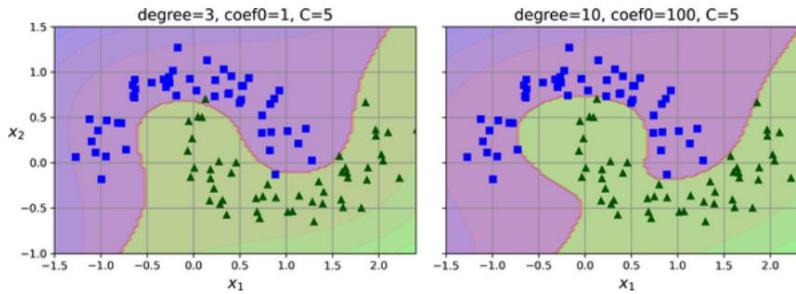


Figura 5-7. Clasificadores SVM con un núcleo polinomial

## CONSEJO

Aunque los hiperparámetros generalmente se ajustan automáticamente (por ejemplo, mediante búsqueda aleatoria), es bueno tener una idea de lo que realmente hace cada hiperparámetro y cómo puede interactuar con otros hiperparámetros: de esta manera, puede limitar la búsqueda a un espacio mucho más pequeño.

## Características de similitud

Otra técnica para abordar problemas no lineales es agregar características calculadas usando una función de similitud, que mide cuánto se parece cada instancia a un punto de referencia particular, como hicimos en el Capítulo 2 cuando agregamos las características de similitud geográfica. Por ejemplo, tomemos el conjunto de datos 1D anterior y agreguemos dos puntos de referencia en  $x = -2$  y  $x = 1$  (consulte el gráfico de la izquierda en la Figura 5-8). A continuación, definiremos la función de similitud como el RBF gaussiano con  $y = 0.3$ . Esta es una función en forma de campana que varía de 0 (muy lejos del punto de referencia) a 1 (en el punto de referencia).

Ahora estamos listos para calcular las nuevas características. Por ejemplo, miremos la instancia  $x = -1$ : está ubicada a una distancia de 1 del primer punto de referencia y de 2 del segundo punto de referencia. Por lo tanto, sus nuevas características son  $x = \exp(-0.3 \times 1) \approx 0.74$  y  $x = \exp(-0.3^2 \times 2) \approx 0.30$ . El gráfico de la derecha en la Figura 5-8 muestra el conjunto de datos transformado (eliminando las características originales). Como puede ver, ahora es linealmente separable.

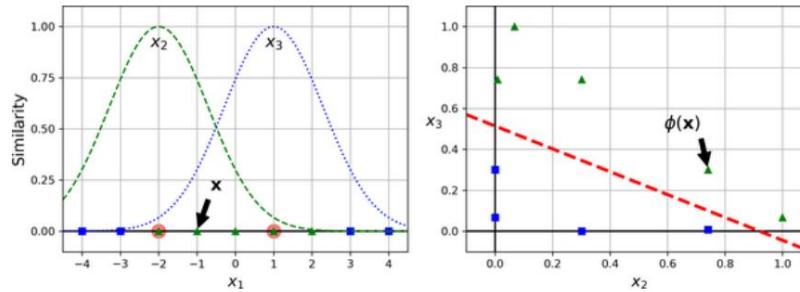


Figura 5-8. Características de similitud utilizando el RBF gaussiano

Quizás se pregunte cómo seleccionar los puntos de referencia. El enfoque más simple es crear un punto de referencia en la ubicación de todas y cada una de las instancias del conjunto de datos. Hacer eso crea muchas dimensiones y, por lo tanto, aumenta las posibilidades de que el conjunto de entrenamiento transformado sea linealmente separable. La desventaja es que un conjunto de entrenamiento con  $m$  instancias y  $n$  características se transforma en un conjunto de entrenamiento con  $m$  instancias y  $m$  características (suponiendo que elimine las características originales). Si su conjunto de entrenamiento es muy grande, terminará con una cantidad igualmente grande de funciones.

## Núcleo RBF gaussiano

Al igual que el método de características polinómicas, el método de características de similitud puede ser útil con cualquier algoritmo de aprendizaje automático, pero puede resultar costoso desde el punto de vista computacional calcular todas las características adicionales (especialmente en conjuntos de entrenamiento grandes). Una vez más el truco del kernel hace su magia SVM, permitiendo obtener un resultado similar como si hubieras añadido muchas características de similitud, pero sin hacerlo realmente. Probemos la clase SVC con el núcleo Gaussiano RBF:

```
rbf_kernel_svm_clf = make_pipeline(StandardScaler(),
                                    SVC(núcleo="rbf", gamma=5, C=0.001))
rbf_kernel_svm_clf.fit(X, y)
```

Este modelo está representado en la parte inferior izquierda de la [Figura 5-9](#). Los otros gráficos muestran modelos entrenados con diferentes valores de los hiperparámetros gamma ( $\gamma$ ) y  $C$ . El aumento de gamma hace que la curva en forma de campana sea más estrecha (consulte los gráficos de la izquierda en la [Figura 5-8](#)). Como resultado, el rango de influencia de cada instancia es menor: el límite de decisión termina siendo más irregular, moviéndose alrededor de instancias individuales. Por el contrario, un valor gamma pequeño hace que la curva en forma de campana sea más ancha: las instancias tienen un rango de influencia más amplio y el límite de decisión termina siendo más suave. Entonces  $\gamma$  actúa como un hiperparámetro de regularización: si su modelo está sobreajustado, debe reducir  $\gamma$ ; si no es adecuado, debe aumentar  $\gamma$  (similar al hiperparámetro  $C$ ).

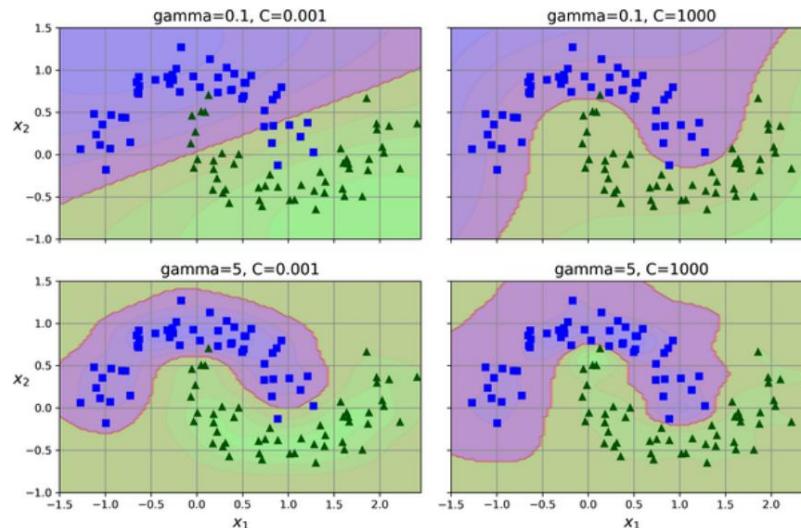


Figura 5-9. Clasificadores SVM que utilizan un kernel RBF

Existen otros núcleos, pero se utilizan mucho menos. Algunos núcleos están especializados en estructuras de datos específicas. Los núcleos de cadena se utilizan a veces al clasificar documentos de texto o secuencias de ADN (por ejemplo, utilizando el núcleo de subsecuencia de cadena o núcleos basados en la distancia de Levenshtein).

#### CONSEJO

Con tantos núcleos para elegir, ¿cómo puedes decidir cuál usar? Como regla general, siempre debes probar primero el kernel lineal. La clase LinearSVC es mucho más rápida que SVC(kernel="linear"), especialmente si el conjunto de entrenamiento es muy grande. Si no es demasiado grande, también debería probar SVM kernelizadas, comenzando con el kernel Gaussiano RBF; muchas veces funciona muy bien. Luego, si tiene tiempo libre y capacidad informática, puede experimentar con algunos otros núcleos utilizando la búsqueda de hiperparámetros. Si existen núcleos especializados para la estructura de datos de su conjunto de entrenamiento, asegúrese de probarlos también.

## Clases SVM y complejidad computacional

La clase LinearSVC se basa en la biblioteca liblinear, que implementa un [algoritmo optimizado](#), para SVM lineales. No es compatible con el truco del kernel, pero escala casi linealmente con la cantidad de entrenamiento. instancias y el número de características. La complejidad del tiempo de entrenamiento es aproximadamente  $O(m \times n)$ . el algoritmo lleva más tiempo si requiere una precisión muy alta. Esto está controlado por el hiperparámetro de tolerancia (llamado tol en Scikit-Learn). En la mayoría de las tareas de clasificación, la tolerancia predeterminada está bien.

La clase SVC se basa en la biblioteca libsvm, que implementa un [algoritmo que admite el kernel](#). truco. La complejidad del tiempo de entrenamiento suele estar entre  $O(m \times n)$  y  $O(m^2 \times n)$ . Desafortunadamente, esto significa que se vuelve terriblemente lento cuando el número de instancias de entrenamiento aumenta (por ejemplo, cientos de miles de instancias), por lo que este algoritmo es mejor para conjuntos de entrenamiento no lineales pequeños o medianos. Él escala bien con el número de características, especialmente con características escasas (es decir, cuando cada instancia tiene pocas características distintas de cero). En este caso, el algoritmo escala aproximadamente con el número promedio de valores distintos de cero. características por instancia.

La clase SGDClassifier también realiza una clasificación de margen grande de forma predeterminada, y su hiperparámetros, especialmente los hiperparámetros de regularización (alfa y penalización) y los learning\_rate: se puede ajustar para producir resultados similares a los de las SVM lineales. Para entrenar utiliza descenso de gradiente estocástico (ver [Capítulo 4](#)), que permite el aprendizaje incremental y usa poca memoria, por lo que puede usarlo para entrenar un modelo en un conjunto de datos grande que no cabe en la RAM (es decir, para sistemas fuera del núcleo). aprendiendo). Además, se escala muy bien, ya que su complejidad computacional es  $O(m \times n)$ . La tabla 5-1 compara Clases de clasificación SVM de Scikit-Learn.

Tabla 5-1. Comparación de clases de Scikit-Learn para la clasificación SVM

Clase	Complejidad del tiempo	Soporte fuera del núcleo	El escalado requiere truco del kernel
LinearSVC	$O(m \times n)$	No	Sí
SVC	$O(m^2 \times n)$ a $O(m^3 \times n)$	No	Sí
Clasificador SGD	$O(m \times n)$	Sí	Sí

Ahora veamos cómo los algoritmos SVM también se pueden utilizar para regresión lineal y no lineal.

## Regresión SVM

Para utilizar SVM para regresión en lugar de clasificación, el truco consiste en modificar el objetivo: en lugar de intentar para ajustar la calle más grande posible entre dos clases y al mismo tiempo limitar las violaciones de márgenes, regresión SVM intenta encajar tantas instancias como sea posible en la calle mientras limita las violaciones de márgenes (es decir, instancias fuera de la calle). El ancho de la calle está controlado por un hiperparámetro, . La Figura 5-10 muestra dos lineales. Modelos de regresión SVM entrenados con algunos datos lineales, uno con un margen pequeño ( $\gamma = 0,5$ ) y el otro con un margen mayor ( $\gamma = 1,2$ ).

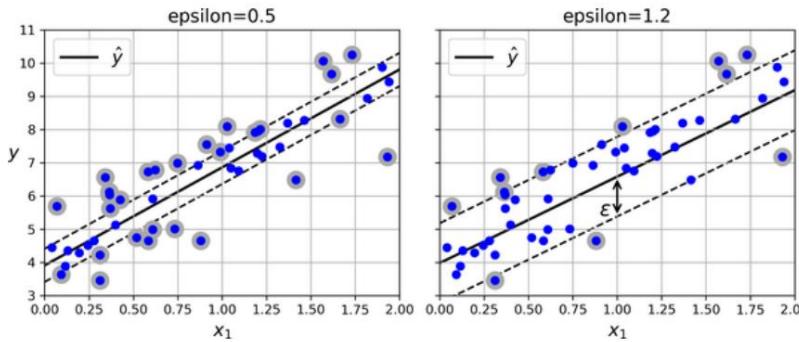


Figura 5-10. regresión SVM

Reducir  $\epsilon$  aumenta el número de vectores de soporte, lo que regulariza el modelo. Además, si agrega más instancias de entrenamiento dentro del margen, no afectará las predicciones del modelo; por tanto, se dice que el modelo es  $\epsilon$ -insensible.

Puede utilizar la clase LinearSVR de Scikit-Learn para realizar una regresión SVM lineal. El siguiente código produce el modelo representado a la izquierda en la [Figura 5-10](#):

```
desde sklearn.svm importar LinearSVR

X, y = [...] # un conjunto de datos lineal svm_reg
= make_pipeline(StandardScaler(),
                LinearSVR(épsilon=0.5, estado_aleatorio=42))
svm_reg.fit(X, y)
```

Para abordar tareas de regresión no lineal, puede utilizar un modelo SVM kernelizado. [La Figura 5-11](#) muestra la regresión SVM en un conjunto de entrenamiento cuadrático aleatorio, utilizando un núcleo polinomial de segundo grado. Hay cierta regularización en el gráfico de la izquierda (es decir, un valor C pequeño) y mucha menos en el gráfico de la derecha (es decir, un valor C grande).

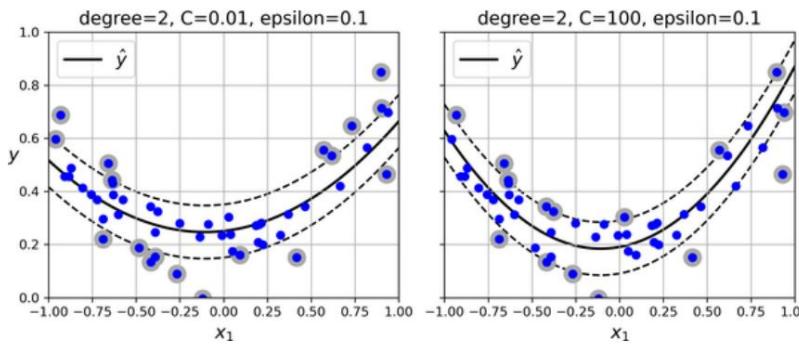


Figura 5-11. Regresión SVM utilizando un núcleo polinómico de segundo grado

El siguiente código utiliza la clase SVR de Scikit-Learn (que admite el truco del kernel) para producir el modelo representado a la izquierda en la [Figura 5-11](#):

```
desde sklearn.svm importar SVR

X, y = [...] # un conjunto de datos cuadrático
svm_poly_reg = make_pipeline(StandardScaler(),
                             SVR(kernel="poli", grado=2, C=0.01, épsilon=0.1))
svm_poly_reg.fit(X, y)
```

La clase SVR es el equivalente de regresión de la clase SVC y la clase LinearSVR es el equivalente de regresión de la clase LinearSVC. La clase LinearSVR escala linealmente con el tamaño del entrenamiento

conjunto (al igual que la clase LinearSVC), mientras que la clase SVR se vuelve demasiado lenta cuando el conjunto de entrenamiento crece muy grande (al igual que la clase SVC).

### NOTA

Las SVM también se pueden utilizar para la detección de novedades, como verá en [el Capítulo 9](#).

El resto de este capítulo explica cómo las SVM hacen predicciones y cómo funcionan sus algoritmos de entrenamiento. comenzando con clasificadores SVM lineales. Si recién está comenzando con el aprendizaje automático, puede hacerlo de manera segura omita esto y vaya directamente a los ejercicios al final de este capítulo, y vuelva más tarde cuando lo desee. para obtener una comprensión más profunda de las SVM.

## Bajo el capó de los clasificadores SVM lineales

Un clasificador SVM lineal predice la clase de una nueva instancia  $x$  calculando primero la función de decisión  $\theta \cdot x + b$ . Si el resultado es positivo, entonces el predicha  $\hat{y}$  es la clase positiva (1); de lo contrario es la clase negativa (0). Esto es exactamente como Regresión logística (discutida en [el Capítulo 4](#)).

### NOTA

Hasta ahora, he usado la convención de poner todos los parámetros del modelo en un vector  $\theta$ , incluido el término de sesgo.  $\theta$  y la característica de entrada pondera  $\theta_0$  a  $\theta_1$ . Esto requirió agregar una entrada de sesgo  $x = 1$  a todas las instancias. Otro muy común es separar el término de sesgo  $b$  (igual a  $\theta_0$ ) y el vector de ponderaciones de características  $w$  (que contiene  $\theta_1$  a  $\theta_n$ ). En este caso, no es necesario agregar ninguna característica de sesgo a los vectores de características de entrada, y la decisión del SVM lineal la función es igual a  $wx + b = w_1x_1 + \dots + w_nx_n + b$ . Usaré esta convención a lo largo del resto de este libro.

Por tanto, hacer predicciones con un clasificador SVM lineal es bastante sencillo. ¿Qué tal entrenar? Este requiere encontrar el vector de pesos  $w$  y el término de sesgo  $b$  que hacen que la calle, o margen, sea tan ancha como posible y al mismo tiempo limitar el número de violaciones de márgenes. Empecemos por el ancho de la calle: para que sea más grande, necesitamos hacer  $w$  más pequeño. Esto puede ser más fácil de visualizar en 2D, como se muestra en [la Figura 5-12](#). vamos definir los bordes de la calle como los puntos donde la función de decisión es igual a  $-1$  o  $+1$ . En la izquierda Grafique el peso  $w$  es  $1$ , por lo que los puntos en los que  $wx = -1$  o  $+1$  son  $x = -1$  y  $+1$ ; por lo tanto, el margen el tamaño es  $2$ . En el gráfico de la derecha, el peso es  $0.5$ , por lo que los puntos en los que  $wx = -1$  o  $+1$  son  $x = -2$  y  $+2$ ; el tamaño del margen es  $4$ . Por lo tanto, debemos mantener  $w$  lo más pequeño posible. Tenga en cuenta que el término de sesgo  $b$  no tiene influencia en el tamaño del margen: ajustarlo simplemente desplaza el margen, sin afectar su tamaño.

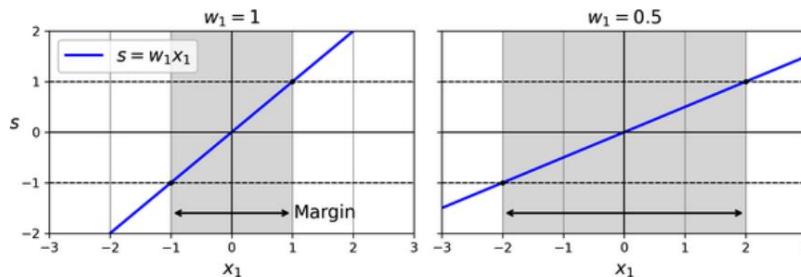


Figura 5-12. Un vector de peso más pequeño da como resultado un margen mayor

También queremos evitar violaciones de márgenes, por lo que necesitamos que la función de decisión sea mayor que  $1$  para todos las instancias de entrenamiento positivas y menor que  $-1$  para instancias de entrenamiento negativas. Si definimos  $t = -1$  para

instancias negativas (cuando  $y = -1$ ) y  $t = 1$  para las instancias positivas (cuando  $y = 1$ ), entonces podemos escribir esta restricción como  $(w \cdot x + b) \geq 1$  para todas las instancias.

Por lo tanto, podemos expresar el objetivo del clasificador SVM lineal de margen duro como el problema de optimización restringida en la Ecuación 5-1.

Ecuación 5-1. Objetivo del clasificador SVM lineal de margen duro

$$\begin{array}{ll} \text{minimizar} & \frac{1}{2} w \cdot w \\ w, b & \\ \text{sujeto a } t(i)(w \cdot x(i) + b) \geq 1 & \text{para } i = 1, 2, \dots, m \end{array}$$

#### NOTA

Estamos minimizando  $\frac{1}{2} w \cdot w$ , que es igual a  $\frac{1}{2} \|w\|^2$ , tiene una derivada agradable y simple (es solo  $w$ ), mientras que  $\|w\|$  no es diferenciable en  $w = 0$ . Los algoritmos de optimización a menudo funcionan mucho mejor en funciones diferenciables.

Para obtener el objetivo del margen suave, necesitamos introducir una variable de holgura  $\zeta \geq 0$  para cada instancia:  $\zeta$  mide cuánto se le permite a la instancia  $i$  violar el margen. Ahora tenemos dos objetivos en conflicto: hacer que las variables de holgura sean lo más pequeñas posible para reducir las violaciones de margen y hacer que  $\frac{1}{2} w \cdot w$  sea lo más pequeña posible para aumentar el margen. Aquí es donde entra en juego el hiperparámetro  $C$ : nos permite definir el equilibrio entre estos dos objetivos. Esto nos da el problema de optimización restringida en la Ecuación 5-2.

Ecuación 5-2. Objetivo clasificador SVM lineal de margen suave

$$\begin{array}{ll} \text{minimizar} & \frac{1}{2} w \cdot w + C \sum_{y_i=1} \zeta_i \\ w, b, \zeta & \\ \text{sujeto a } t(i)(w \cdot x(i) + b) \geq 1 - \zeta_i & \text{y } \zeta_i \geq 0 \text{ para } i = 1, 2, \dots, m \end{array}$$

Los problemas de margen duro y margen blando son problemas de optimización cuadrática convexa con restricciones lineales. Estos problemas se conocen como problemas de programación cuadrática (QP). Hay muchos solucionadores disponibles para resolver problemas de QP mediante el uso de una variedad de técnicas que están fuera del alcance de este libro. <sup>4</sup>

Usar un solucionador de QP es una forma de entrenar una SVM. Otra es utilizar el descenso de gradiente para minimizar la pérdida de bisagra o la pérdida de bisagra al cuadrado (consulte la Figura 5-13). Dada una instancia  $x$  de la clase positiva (es decir, con  $t = 1$ ), la pérdida es 0 si la salida  $s$  de la función de decisión ( $s = w \cdot x + b$ ) es mayor o igual a 1. Esto sucede cuando la instancia es Fuera de la calle y en el lado positivo. Dada una instancia de la clase negativa (es decir, con  $t = -1$ ), la pérdida es 0 si  $s \leq -1$ . Esto sucede cuando la instancia está fuera de la calle y en el lado negativo. Cuanto más alejada esté una instancia del lado correcto del margen, mayor será la pérdida: crece linealmente para la pérdida de bisagra y cuadráticamente para la pérdida de bisagra al cuadrado. Esto hace que la pérdida de bisagra al cuadrado sea más sensible a los valores atípicos. Sin embargo, si el conjunto de datos está limpio, tiende a converger más rápido. De forma predeterminada, LinearSVC usa la pérdida de bisagra al cuadrado, mientras que SGDClassifier usa la pérdida de bisagra. Ambas clases le permiten elegir la pérdida configurando el hiperparámetro de pérdida en "hinge" o "squared\_hinge". El algoritmo de optimización de la clase SVC encuentra una solución similar a la de minimizar la pérdida de bisagra.

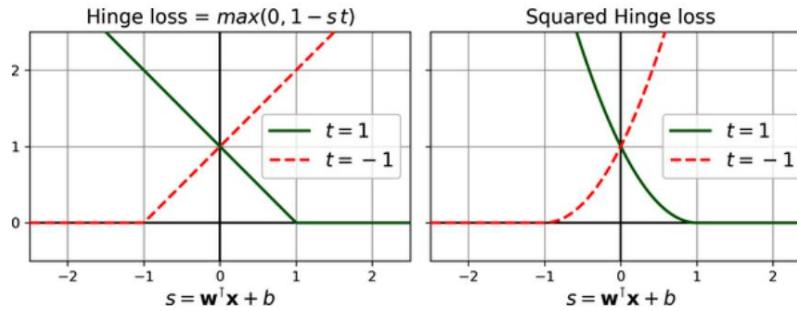


Figura 5-13. La pérdida de bisagra (izquierda) y la pérdida de bisagra al cuadrado (derecha)

A continuación, veremos otra forma de entrenar un clasificador SVM lineal: resolviendo el problema dual.

### El doble problema

Dado un problema de optimización restringida, conocido como problema primario, es posible expresar un problema diferente pero estrechamente relacionado, llamado problema dual. La solución al problema dual normalmente da un límite inferior a la solución del problema primario, pero bajo algunas condiciones puede tener la misma solución que el problema primario.

Afortunadamente, el problema SVM cumple estas condiciones, por lo que puedes elegir resolver el problema primario o el problema <sup>5</sup>dual; ambos tendrán la misma solución.

[La ecuación 5-3](#) muestra la forma dual del objetivo SVM lineal. Si está interesado en saber cómo derivar el problema dual del problema primario, consulte la sección de material adicional en el cuaderno [de este capítulo](#).

Ecuación 5-3. Forma dual del objetivo SVM lineal

$$\text{minimizar } \frac{1}{2} \sum_{\substack{\text{yo=1} \\ j=1}}^m \alpha(i)\alpha(j)t(i)t(j)x(i) - \sum_{\substack{\text{yo=1} \\ j=1}}^m \alpha(i) \text{ sujeto a } \alpha(i) \geq 0 \text{ para todo } i = 1, 2, \dots$$

Una vez que encuentre el vector  $w^*$  que minimiza esta ecuación (usando un solucionador QP), use [la ecuación 5-4](#) para calcular  $w^*$  y  $b^*$  que minimicen el problema primal. En esta ecuación,  $n$  representa el número de vectores de soporte.

Ecuación 5-4. De la solución dual a la solución primaria

$$w^* = \sum_{\substack{\text{yo=1} \\ i=1}}^n \alpha^*(i)t(i)x(i)$$

$$b^* = \frac{1}{n_s} \sum_{i=1}^{n_s} (t(i) - w^* \cdot x(i)) \quad \alpha^*(i) > 0$$

El problema dual es más rápido de resolver que el principal cuando el número de instancias de entrenamiento es menor que el número de características. Más importante aún, el problema dual hace posible el truco del núcleo, mientras que el problema primario no. Entonces, ¿cuál es este truco del kernel?

### SVM kernelizadas

Suponga que desea aplicar una transformación polinómica de segundo grado a un conjunto de entrenamiento bidimensional (como el conjunto de entrenamiento de lunas) y luego entrenar un clasificador SVM lineal en el conjunto de entrenamiento transformado. [La ecuación 5-5](#) muestra la función de mapeo polinómico de segundo grado que desea aplicar.

Ecuación 5-5. Mapeo de polinomios de segundo grado

$$(x) = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \begin{pmatrix} & & 2x_1^2 \\ & & \sqrt{2}x_1x_2 \\ & & 2x_2^2 \end{pmatrix}$$

Observe que el vector transformado es 3D en lugar de 2D. Ahora veamos qué sucede con un par de imágenes 2D. vectores, a y b, si aplicamos este mapeo polinómico de segundo grado y luego calculamos el producto escalar de los vectores transformados (ver Ecuación 5-6). <sup>6</sup>

Ecuación 5-6. Truco del kernel para un mapeo polinómico de segundo grado

$$\begin{aligned} (a) \cdot (b) &= \frac{\sqrt{2}a_1a_2}{\sqrt{2}b_1b_2} = a_1^2b_1^2 + 2a_1b_1a_2b_2 + a_2^2b_2^2 \\ &= (a_1b_1 + a_2b_2)^2 \\ &= \begin{pmatrix} a_1 & b_1 \\ a_2 & b_2 \end{pmatrix}^2 \end{aligned}$$

¿Qué hay sobre eso? El producto escalar de los vectores transformados es igual al cuadrado del producto escalar de los vectores originales:  $(a) \cdot (b) = (a \cdot b)$ .

Aquí está la idea clave: si aplica la transformación a todas las instancias de entrenamiento, entonces el problema dual (i) (ver Ecuación 5-3) contendrá el producto escalar  $\langle x_i | x_j \rangle$ . Pero si (j) es la transformación polinómica de segundo grado definida en la Ecuación 5-5, entonces puedes reemplazar este producto escalar de los vectores transformados

simplemente por  $\langle x(i) | x(j) \rangle$ . Por lo tanto, no es necesario transformar las instancias de capacitación en absoluto; simplemente reemplaza el punto producto por su cuadrado en la ecuación 5-3. El resultado será estrictamente el mismo que si se hubiera tomado la molestia de transformar el conjunto de entrenamiento y luego ajustar un algoritmo SVM lineal, pero este truco hace que todo el proceso sea mucho más eficiente desde el punto de vista computacional.

La función  $K(a, b) = (a \cdot b)$  es un núcleo polinomial de segundo grado. En el aprendizaje automático, un núcleo es una función capaz de calcular el producto escalar  $(a) \cdot (b)$ , basándose únicamente en los vectores originales a y b, sin tener que calcular (o incluso conocer) la transformación . La ecuación 5-7 enumera algunos de los núcleos más utilizados.

Ecuación 5-7. grano comunes

$$\text{Lineal: } K(a, b) = a \cdot b$$

$$\text{Polinomio: } K(a, b) = (\gamma a \cdot b + r)^d$$

$$\text{RBF gaussiano: } K(a, b) = \exp(-\gamma \|a - b\|^2)$$

$$\text{Sigmoide: } K(a, b) = \tanh(\gamma a \cdot b + r)$$

## TEOREMA DE MERCER

Según el teorema de Mercer, si una función  $K(a, b)$  respeta algunas condiciones matemáticas llamadas condiciones de Mercer (por ejemplo,  $K$  debe ser continua y simétrica en sus argumentos para que  $K(a, b) = K(b, a)$ , etc.), entonces existe una función  $\hat{K}$  que asigna a  $a$  y  $b$  a otro espacio (posiblemente con dimensiones mucho más altas) tal que  $K(a, b) = \hat{K}(a, b)$ .

(b). Puedes usar  $K$  como núcleo porque sabes que  $\hat{K}$  existe, incluso si no sabes qué es  $\hat{K}$ . En el caso del núcleo Gaussiano RBF, se puede demostrar que  $\hat{K}$  asigna cada instancia de entrenamiento a un espacio de dimensión infinita, por lo que es bueno que no necesites realizar el mapeo.

Tenga en cuenta que algunos núcleos utilizados con frecuencia (como el núcleo sigmoideo) no respetan todas las condiciones de Mercer, pero generalmente funcionan bien en la práctica.

Aún queda un cabo suelto que debemos atar. La ecuación 5-4 muestra cómo pasar de la solución dual a la solución primaria en el caso de un clasificador SVM lineal. Pero si aplicas el truco del kernel, terminarás con ecuaciones que incluyen  $\hat{K}(x)$ . De hecho,  $\hat{w}$  debe tener el mismo número de dimensiones que  $\hat{K}(x)$ , que puede ser enorme o incluso infinita, por lo que no se puede calcular. Pero ¿cómo se pueden hacer predicciones sin saber  $\hat{w}$ ? Bueno, la buena noticia es que puedes introducir la fórmula para  $\hat{w}$  de la ecuación 5-4 en la función de decisión ( $\hat{y}$ ) para una nueva instancia  $x$ . Esto hace posible usar el truco del núcleo (ecuación 5-8), y obtienes una ecuación con solo productos escalares entre vectores de entrada.

Ecuación 5-8. Hacer predicciones con una SVM kernelizada

$$\begin{aligned}
 \hat{y} &= \hat{w}^T \hat{b}(\hat{K}(x)) = \hat{w}^T (\hat{K}(x)) + \hat{b} = (\sum_{i=1}^m \alpha_i t(i) \hat{K}(x, x(i))) + \hat{b} \\
 &= \sum_{i=1}^m \alpha_i t(i) (\hat{K}(x, x(i))) + \hat{b} \\
 &= \sum_{\substack{i=1 \\ \alpha_i > 0}} \alpha_i t(i) K(x(i), x(n)) + \hat{b}
 \end{aligned}$$

Tenga en cuenta que, dado que  $\alpha \neq 0$  solo para los vectores de soporte, hacer predicciones implica calcular el producto escalar ( $n$ ) del nuevo vector de entrada  $x$  solo con los vectores de soporte, no con todas las instancias de entrenamiento. Por supuesto, es necesario utilizar el mismo truco para calcular el término de sesgo ( $\hat{b}$  (Ecuación 5-9)).

Ecuación 5-9. Usando el truco del kernel para calcular el término de sesgo

$$\begin{aligned}
 \hat{b} &= \frac{1}{n_s} \sum_{\substack{i=1 \\ \alpha_i > 0}}^m (t(i) - \hat{w}^T \hat{b}(\hat{K}(x(i)))) = \frac{1}{n_s} \sum_{\substack{i=1 \\ \alpha_i > 0}}^m (t(i) - (\sum_{j=1}^m \alpha_j t(j) \hat{K}(x(i), x(j)))) \\
 &= \frac{1}{n_s} \sum_{\substack{i=1 \\ \alpha_i > 0}}^m -\sum_{\substack{j=1 \\ \alpha_j > 0}}^m \alpha_j t(j) K(x(i), x(j))
 \end{aligned}$$

Si empieza a tener dolor de cabeza, es perfectamente normal: es un efecto secundario desafortunado del truco del kernel.

## NOTA

También es posible implementar SVM kernelizadas en línea, capaces de realizar un aprendizaje incremental, como se describe en los artículos "[Incremental and Decremental Support Vector Machine Learning](#)" y "[Fast Kernel Classifiers with Online and Active Learning](#)". Estas SVM kernelizadas se implementan en Matlab y C++. Pero para problemas no lineales a gran escala, es posible que desee considerar el uso de bosques aleatorios (consulte [el Capítulo 7](#)) o redes neuronales (consulte [la Parte II](#)).

## Ejercicios

1. ¿Cuál es la idea fundamental detrás de las máquinas de vectores de soporte?
2. ¿Qué es un vector de soporte?
3. ¿Por qué es importante escalar las entradas cuando se utilizan SVM?
4. ¿Puede un clasificador SVM generar una puntuación de confianza cuando clasifica una instancia? Que tal un  $\zeta$  probabilidad?
5. ¿Cómo se puede elegir entre LinearSVC, SVC y SGDClassifier?
6. Supongamos que ha entrenado un clasificador SVM con un kernel RBF, pero parece no adaptarse al conjunto de entrenamiento. ¿Debería aumentar o disminuir  $\gamma$  (gamma)? ¿Qué pasa con  $C$ ?
7. ¿Qué significa que un modelo sea  $\epsilon$ -insensible?
8. ¿Cuál es el punto de utilizar el truco del kernel?
9. Entrene un LinearSVC en un conjunto de datos linealmente separable. Luego entrene un SVC y un SGDClassifier en el mismo conjunto de datos. Vea si puede lograr que produzcan aproximadamente el mismo modelo.
10. Entrene un clasificador SVM en el conjunto de datos de vino, que puede cargar usando `sklearn.datasets.load_wine()`. Este conjunto de datos contiene los análisis químicos de 178 muestras de vino producidas por 3 cultivadores diferentes: el objetivo es entrenar un modelo de clasificación capaz de predecir el cultivador basándose en el análisis químico del vino. Dado que los clasificadores SVM son clasificadores binarios, deberá utilizar uno contra todos para clasificar las tres clases. ¿Qué precisión puedes alcanzar?
11. Entrene y ajuste un regresor SVM en el conjunto de datos de vivienda de California. Puede usar el conjunto de datos original en lugar de la versión modificada que usamos en [el Capítulo 2](#), que puede cargar usando `sklearn.datasets.fetch_california_housing()`. Los objetivos representan cientos de miles de dólares. Dado que hay más de 20 000 instancias, las SVM pueden ser lentas, por lo que para el ajuste de hiperparámetros debe usar muchas menos instancias (por ejemplo, 2000) para probar muchas más combinaciones de hiperparámetros. ¿Cuál es el RMSE de tu mejor modelo?

Las soluciones a estos ejercicios están disponibles al final del cuaderno de este capítulo, en <https://homl.info/colab3>.

<sup>1</sup> Chih-Jen Lin et al., "A Dual Coordinate Descent Method for Large-Scale Linear SVM", Actas del 25 Conferencia internacional sobre aprendizaje automático (2008): 408–415.

<sup>2</sup> John Platt, "Optimización mínima secuencial: un algoritmo rápido para entrenar máquinas de vectores de soporte" (Microsoft Informe técnico de investigación, 21 de abril de 1998).

<sup>3</sup> Zeta ( $\zeta$ ) es la sexta letra del alfabeto griego.

<sup>4</sup> Para aprender más sobre programación cuadrática, puedes comenzar leyendo el libro de Stephen Boyd y Lieven Vandenberghe. [Optimización convexa](#) (Cambridge University Press) o viendo [la serie de videoconferencias](#) de Richard Brown .

<sup>5</sup> La función objetivo es convexa y las restricciones de desigualdad son funciones continuamente diferenciables y convexas.

<sup>6</sup> Como se explicó en el capítulo 4, el producto escalar de dos vectores  $a$  y  $b$  normalmente se anota como  $a \cdot b$ . Sin embargo, en la máquina de aprendizaje, los vectores se representan frecuentemente como vectores de columna (es decir, matrices de una sola columna), por lo que el producto escalar se logra calculando  $a^T b$ . Para mantener la coherencia con el resto del libro, usaremos esta notación aquí, ignorando el hecho de que técnicamente esto da como resultado una matriz unicelular en lugar de un valor escalar.

<sup>7</sup> Gert Cauwenberghs y Tomaso Poggio, “Aprendizaje automático con vectores de soporte incremental y decremental”, Actas de la 13.<sup>a</sup> Conferencia internacional sobre sistemas de procesamiento de información neuronal (2000): 388–394.

<sup>8</sup> Antoine Bordes et al., “Clasificadores rápidos de kernel con aprendizaje activo y en línea”, Journal of Machine Learning Research 6 (2005): 1579–1619.

# Capítulo 6. Árboles de decisión

---

Los árboles de decisión son algoritmos versátiles de aprendizaje automático que pueden realizar tareas de clasificación y regresión, e incluso tareas de múltiples salidas. Son algoritmos potentes, capaces de ajustar conjuntos de datos complejos. Por ejemplo, en [el Capítulo 2](#) entrenó un modelo DecisionTreeRegressor en el conjunto de datos de vivienda de California, ajustándolo perfectamente (en realidad, sobreajustándolo).

Los árboles de decisión también son los componentes fundamentales de los bosques aleatorios (ver [Capítulo 7](#)), que se encuentran entre los algoritmos de aprendizaje automático más poderosos disponibles en la actualidad.

En este capítulo comenzaremos discutiendo cómo entrenar, visualizar y hacer predicciones con árboles de decisión. Luego, revisaremos el algoritmo de entrenamiento CART utilizado por Scikit-Learn y exploraremos cómo regularizar árboles y usarlos para tareas de regresión. Finalmente, discutiremos algunas de las limitaciones de los árboles de decisión.

## Entrenamiento y visualización de un árbol de decisiones

Para comprender los árboles de decisión, construyamos uno y echemos un vistazo a cómo hace predicciones. El siguiente código entrena un DecisionTreeClassifier en el conjunto de datos del iris (consulte [el Capítulo 4](#)):

```
desde sklearn.datasets importar load_iris desde
sklearn.tree importar DecisionTreeClassifier

iris = cargar_iris(as_frame=True)
X_iris = iris.data[[" largo del pétalo (cm)", "ancho del pétalo (cm)"]].values y_iris = iris.target

tree_clf = DecisionTreeClassifier(max_profundidad=2, random_state=42) tree_clf.fit(X_iris,
y_iris)
```

Puede visualizar el árbol de decisión entrenado usando primero la función `export_graphviz()` para generar un archivo de definición de gráfico llamado `iris_tree.dot`:

```
de sklearn.tree importar export_graphviz

export_graphviz( tree_clf,
    out_file="iris_tree.dot",
    feature_names=[" largo del pétalo (cm)", "ancho del pétalo (cm)"],
    class_names=iris.target_names,
    redondeado=True, relleno=True
)
```

Luego puedes usar `Graphviz.Source.from_file()` para cargar y mostrar el archivo en un cuaderno Jupyter:

de la fuente de importación de Graphviz

```
Fuente.from_file("iris_tree.dot")
```

Graviz es un paquete de software de visualización de gráficos de código abierto. También incluye una herramienta de línea de comandos de puntos para convertir archivos .dot a una variedad de formatos, como PDF o PNG.

Su primer árbol de decisiones se parece a la Figura 6-1.

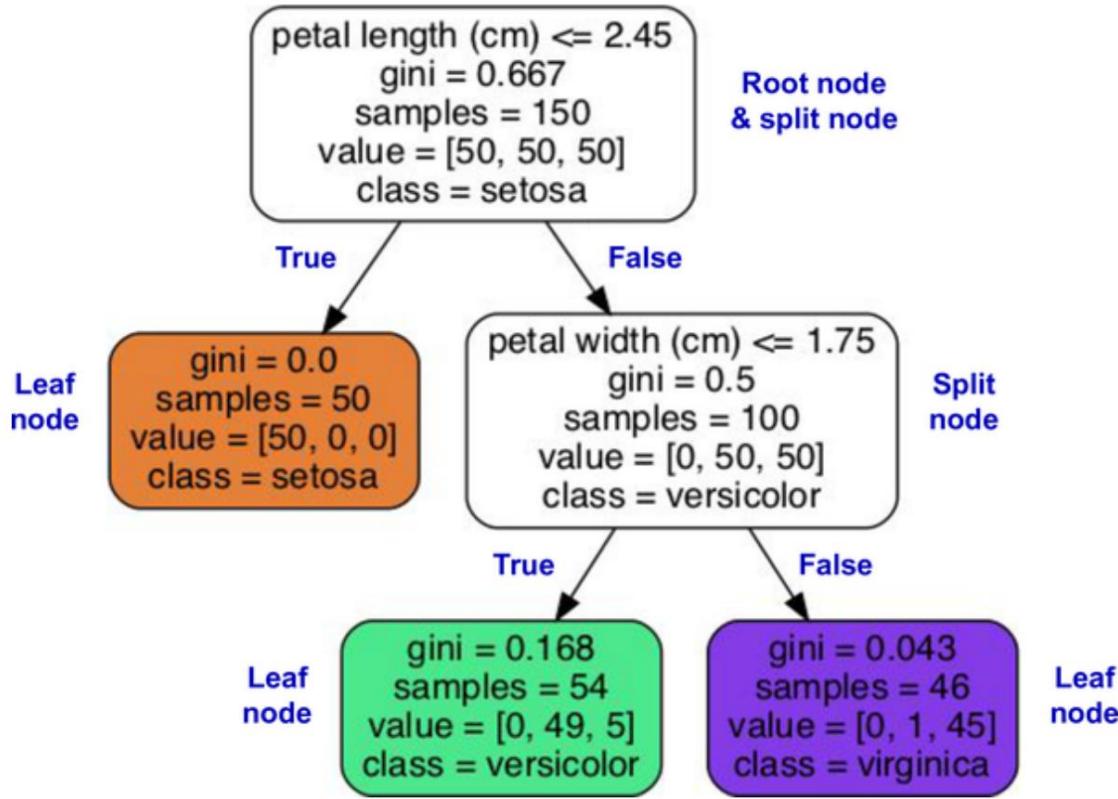


Figura 6-1. Árbol de decisión del iris

### Haciendo predicciones Veamos

cómo el árbol representado en la Figura 6-1 hace predicciones. Supongamos que encuentras una flor de iris y quieres clasificarla según sus pétalos. Se comienza en el nodo raíz (profundidad 0, en la parte superior): este nodo pregunta si la longitud del pétalo de la flor es inferior a 2,45 cm. Si es así, baje al nodo secundario izquierdo de la raíz (profundidad 1, izquierda). En este caso, es un nodo hoja (es decir, no tiene ningún nodo hijo), por lo que no hace ninguna pregunta: simplemente mire la clase predicha para ese nodo y el árbol de decisión predice que su flor es un Iris . setosa (clase=setosa).

Ahora supongamos que encuentras otra flor y esta vez la longitud del pétalo es mayor que 2,45 cm.

Nuevamente comienza en la raíz, pero ahora baja hasta su nodo secundario derecho (profundidad 1, derecha). Este no es un nodo de hoja, es un nodo dividido, por lo que plantea otra pregunta: ¿el ancho del pétalo es menor que 1,75 cm? Si es así, lo más probable es que tu flor sea un Iris versicolor (profundidad 2, izquierda). De lo contrario, es probable que se trate de una Iris virginica (profundidad 2, derecha). Es realmente así de simple.

### NOTA

Una de las muchas cualidades de los árboles de decisión es que requieren muy poca preparación de datos. De hecho, no requieren ningún escalado ni centrado de funciones.

El atributo de muestras de un nodo cuenta a cuántas instancias de entrenamiento se aplica. Por ejemplo, 100 instancias de entrenamiento tienen una longitud de pétalo superior a 2,45 cm (profundidad 1, derecha), y de esas 100, 54 tienen una anchura de pétalo inferior a 1,75 cm (profundidad 2, izquierda). El atributo de valor de un nodo le indica a cuántas instancias de entrenamiento de cada clase se aplica este nodo: por ejemplo, el nodo inferior derecho se aplica a 0 Iris setosa, 1 Iris versicolor y 45 Iris virginica. Finalmente, el atributo gini de un nodo mide su impureza Gini: un nodo es “puro” ( $\text{gini}=0$ ) si todas las instancias de entrenamiento a las que aplica pertenecen a la misma clase. Por ejemplo, dado que el nodo izquierdo de profundidad 1 se aplica solo a instancias de entrenamiento de Iris setosa, es puro y su impureza de Gini es 0. [La ecuación 6-1](#) muestra cómo el algoritmo de entrenamiento calcula la impureza de Gini  $G_i$  del nodo  $i$ . El nodo izquierdo de profundidad 2 tiene una impureza de Gini igual a  $1 - \left( \frac{0}{54} + \frac{49}{54} \right) \approx 0,168$ .

Ecuación 6-1. impureza de gini

$$G_i = 1 - \sum_{k=1}^n p_{i,k}^2$$

En esta ecuación:

- $G_i$  es la impureza de Gini del nodo  $i$ .
- $p_{y_0, k}$  es la proporción de instancias de clase  $k$  entre las instancias de entrenamiento en el nodo  $i$ .

### NOTA

Scikit-Learn utiliza el algoritmo CART, que produce sólo árboles binarios, es decir, árboles donde los nodos divididos siempre tienen exactamente dos hijos (es decir, las preguntas sólo tienen respuestas de sí/no). Sin embargo, otros algoritmos, como ID3, pueden producir árboles de decisión con nodos que tienen más de dos hijos.

[La Figura 6-2](#) muestra los límites de decisión de este árbol de decisión. La línea vertical gruesa representa el límite de decisión del nodo raíz (profundidad 0): longitud del pétalo = 2,45 cm. Dado que el área de la izquierda es pura (sólo Iris setosa), no se puede dividir más. Sin embargo, el área de la derecha es impura, por lo que el nodo derecho de profundidad 1 la divide en un ancho de pétalo = 1,75 cm (representado por la línea discontinua). Dado que `max_depth` se estableció en 2, el árbol de decisión se detiene allí. Si establece `max_depth` en 3, entonces los dos nodos de profundidad 2 agregarían cada uno otro límite de decisión (representado por las dos líneas de puntos verticales).

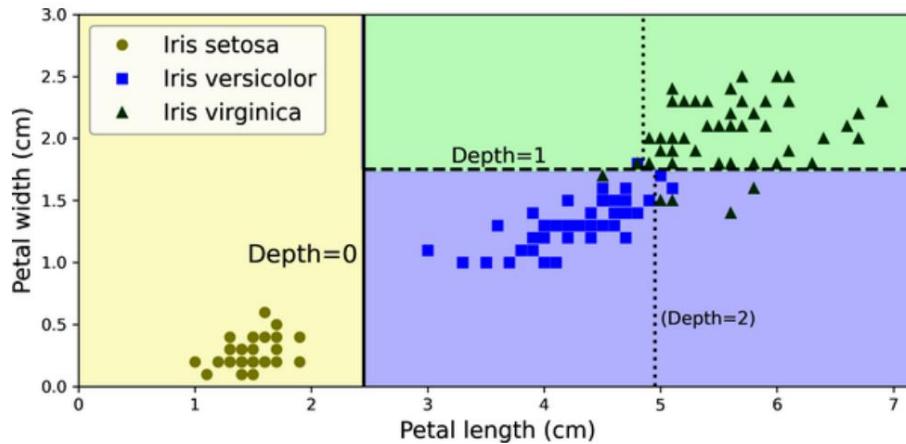


Figura 6-2. Límites de decisión del árbol de decisión

CONSEJO

La estructura de árbol, incluida toda la información que se muestra en la [Figura 6-1](#), está disponible a través del atributo `tree_.tree_` del clasificador. Escriba `help(tree_clf.tree_)` para obtener más detalles y consulte el [cuaderno de este capítulo](#) para un ejemplo.

### INTERPRETACIÓN DEL MODELO: CAJA BLANCA VERSUS CAJA NEGRA

Los árboles de decisión son intuitivos y sus decisiones son fáciles de interpretar. Estos modelos suelen denominarse modelos de caja blanca. Por el contrario, como verá, los bosques aleatorios y las redes neuronales generalmente se consideran modelos de caja negra. Hacen grandes predicciones y usted puede comprobar fácilmente los cálculos que realizaron para hacer estas predicciones; sin embargo, suele ser difícil explicar en términos sencillos por qué se hicieron las predicciones. Por ejemplo, si una red neuronal dice que una persona en particular aparece en una imagen, es difícil saber qué contribuyó a esta predicción: ¿reconoció el modelo los ojos de esa persona? ¿Su boca? ¿Su nariz? ¿Sus zapatos? ¿O incluso el sofá en el que estaban sentados? Por el contrario, los árboles de decisión proporcionan reglas de clasificación sencillas y agradables que incluso pueden aplicarse manualmente si es necesario (por ejemplo, para la clasificación de flores). El campo del ML interpretable tiene como objetivo crear sistemas de ML que puedan explicar sus decisiones de una manera que los humanos puedan entender. Esto es importante en muchos ámbitos; por ejemplo, para garantizar que el sistema no tome decisiones injustas.

### Estimación de probabilidades de clase

Un árbol de decisión también puede estimar la probabilidad de que una instancia pertenezca a una clase k particular. Primero atraviesa el árbol para encontrar el nodo hoja para esta instancia y luego devuelve la proporción de instancias de entrenamiento de clase k en este nodo. Por ejemplo, supongamos que ha encontrado una flor cuyos pétalos miden 5 cm de largo y 1,5 cm de ancho. El nodo de hoja correspondiente es el nodo izquierdo de profundidad 2, por lo que el árbol de decisión genera las siguientes probabilidades: 0% para Iris setosa (0/54), 90,7% para Iris versicolor (49/54) y 9,3% para Iris virginica ( 5/54). Y si lo pides

predecir la clase, genera Iris versicolor (clase 1) porque tiene la mayor probabilidad.

Comprobemos esto:

```
>>> tree_clf.predict_proba([[5, 1.5]]).round(3)
array([[0. , 0.907,
       0.093]])
>>> tree_clf.predict([[5, 1.5]])
array([1])
```

¡Perfecto! Observe que las probabilidades estimadas serían idénticas en cualquier otro lugar del rectángulo inferior derecho de la [Figura 6-2](#); por ejemplo, si los pétalos midieran 6 cm de largo y 1,5 cm de ancho (aunque parece obvio que lo más probable es que sea un Iris) . virginica en este caso).

## El algoritmo de entrenamiento CART

Scikit-Learn utiliza el algoritmo de árbol de clasificación y regresión (CART) para entrenar árboles de decisión (también llamados árboles "en crecimiento"). El algoritmo funciona dividiendo primero el conjunto de entrenamiento en dos subconjuntos utilizando una única característica  $k$  y un umbral  $t$  (por ejemplo, "longitud del pétalo  $\leq 2.45$  cm"). ¿ Cómo elige  $k$  y  $t$  ? Busca el par  $(k, t)$  que produce los subconjuntos más puros, ponderados por su tamaño. [La ecuación 6-2](#) proporciona la función de costo que el algoritmo intenta minimizar.

Ecuación 6-2. Función de costo CART para clasificación

$$J(k, t_k) = \frac{\text{izquierda}}{\text{metro}} G_{\text{left}} + \frac{\text{derecha}}{\text{metro}} G_{\text{right}}$$

$G_{\text{left/right}}$  mide la impureza del subconjunto izquierdo/derecho

dónde {  $m_{\text{left/right}}$  es el número de instancias en el subconjunto izquierdo/derecho}

Una vez que el algoritmo CART ha dividido con éxito el conjunto de entrenamiento en dos, divide los subconjuntos usando la misma lógica, luego los subconjuntos, y así sucesivamente, de forma recursiva. Deja de recurrir una vez que alcanza la profundidad máxima (definida por el hiperparámetro `max_depth`), o si no puede encontrar una división que reduzca la impureza. Algunos otros hiperparámetros (descritos en un momento) controlan condiciones de parada adicionales: `min_samples_split`, `min_samples_leaf`, `min_weight_fraction_leaf` y `max_leaf_nodes`.

### ADVERTENCIA

Como puede ver, el algoritmo CART es un algoritmo codicioso: busca con avidez una división óptima en el nivel superior y luego repite el proceso en cada nivel posterior. No comprueba si la división conducirá a la menor impureza posible varios niveles más abajo. Un algoritmo codicioso a menudo produce una solución que es razonablemente buena pero que no garantiza que sea óptima.

Desafortunadamente, se sabe que encontrar el árbol óptimo es un problema NP completo . Requiere tiempo  $O(\exp(m))$  , lo que hace que el problema sea intratable incluso para conjuntos de entrenamiento pequeños. Es por eso que debemos conformarnos con una solución "razonablemente buena" al entrenar árboles de decisión.

## Complejidad computacional

Para hacer predicciones es necesario recorrer el árbol de decisión desde la raíz hasta la hoja. Los árboles de decisión generalmente están aproximadamente equilibrados, por lo que atravesar el árbol de decisión requiere pasar aproximadamente por nodos  $O(\log(m))$ , donde  $\log_2(m)$  es el logaritmo binario de  $m$  igual a  $\log(m) / \log(2)$ . Dado que cada nodo solo requiere verificar el valor de una característica, la complejidad general de la predicción es  $O(\log(m))$ , independientemente de la cantidad de características. Por tanto, las predicciones son muy rápidas, incluso cuando se trata de grandes conjuntos de entrenamiento.

El algoritmo de entrenamiento compara todas las características (o menos si se establece `max_features`) en todas las muestras en cada nodo. La comparación de todas las características en todas las muestras en cada nodo da como resultado una complejidad de entrenamiento de  $O(n \times m \log(m))$ .

## ¿Impureza de Gini o entropía?

De forma predeterminada, la clase `DecisionTreeClassifier` utiliza la medida de impureza de Gini, pero puede seleccionar la medida de impureza de entropía estableciendo el hiperparámetro de criterio en "entropía". El concepto de entropía se originó en la termodinámica como una medida del desorden molecular: la entropía tiende a cero cuando las moléculas están quietas y bien ordenadas.

Posteriormente, la entropía se extendió a una amplia variedad de dominios, incluida la teoría de la información de Shannon, donde mide el contenido de información promedio de un mensaje, como vimos en el capítulo 4. La entropía es cero cuando todos los mensajes son idénticos. En el aprendizaje automático, la entropía se utiliza con frecuencia como medida de impureza: la entropía de un conjunto es cero cuando contiene instancias de una sola clase. La ecuación 6-3 muestra la definición de la entropía del nodo  $i$ . Por ejemplo, el nodo izquierdo de profundidad 2 en la Figura 6-1 tiene una entropía igual a  $-(49/54) \log(49/54) - (5/54) \log(5/54) \approx 0,445$ .

2

2

Ecuación 6-3. entropía

$$H_{\text{Gini}} = -n \sum_{k=1}^K p_{i,k} \log_2(p_{i,k})$$

Entonces, ¿deberías utilizar la impureza o la entropía de Gini? La verdad es que la mayoría de las veces no supone una gran diferencia: dan lugar a árboles similares. La impureza de Gini es un poco más rápida de calcular, por lo que es un buen valor predeterminado. Sin embargo, cuando difieren, la impureza de Gini tiende a aislar la clase más frecuente en su propia rama del árbol, mientras que la entropía tiende a producir 2 árboles ligeramente más equilibrados.

## Hiperparámetros de regularización

Los árboles de decisión hacen muy pocas suposiciones sobre los datos de entrenamiento (a diferencia de los modelos lineales, que suponen que los datos son lineales, por ejemplo). Si no se restringe, la estructura del árbol se adaptará a los datos de entrenamiento, ajustándolos muy estrechamente; de hecho, probablemente sobreajustándolos. Un modelo de este tipo suele denominarse modelo no paramétrico, no porque no tenga ningún parámetro (a menudo tiene muchos), sino porque el número de parámetros no es el mismo.

determinado antes del entrenamiento, por lo que la estructura del modelo puede adherirse estrechamente a los datos. Por el contrario, un modelo paramétrico, como un modelo lineal, tiene un número predeterminado de parámetros, por lo que su grado de libertad es limitado, lo que reduce el riesgo de sobreajuste (pero aumenta el riesgo de desajuste).

Para evitar un ajuste excesivo de los datos de entrenamiento, es necesario restringir la libertad del árbol de decisiones durante el entrenamiento. Como ya sabes, esto se llama regularización. Los hiperparámetros de regularización dependen del algoritmo utilizado, pero generalmente se puede al menos restringir la profundidad máxima del árbol de decisión. En Scikit-Learn, esto está controlado por el hiperparámetro `max_depth`. El valor predeterminado es Ninguno, lo que significa ilimitado. Reducir `max_depth` regularizará el modelo y, por lo tanto, reducirá el riesgo de sobreajuste.

La clase `DecisionTreeClassifier` tiene algunos otros parámetros que restringen de manera similar la forma del árbol de decisión:

`max_features`

Número máximo de características que se evalúan para dividir en cada nodo

`max_leaf_nodes`

Número máximo de nodos de hoja

`min_samples_split`

Número mínimo de muestras que debe tener un nodo antes de poder dividirlo

`min_samples_leaf`

Número mínimo de muestras que debe tener un nodo hoja para crear

`min_weight_fraction_leaf`

Igual que `min_samples_leaf` pero expresado como una fracción del número total de instancias ponderadas

Aumentar los hiperparámetros `min_*` o reducir los hiperparámetros `max_*` regularizará el modelo.

### NOTA

Otros algoritmos funcionan entrenando primero el árbol de decisión sin restricciones y luego podando (eliminando) los nodos innecesarios. Un nodo cuyos hijos sean todos nodos hoja se considera innecesario si la mejora de pureza que proporciona no es estadísticamente significativa. Las pruebas estadísticas estándar,<sup>2</sup> como la prueba de  $\chi^2$  (prueba de chi cuadrado), se utilizan para estimar la probabilidad de que la mejora sea puramente resultado del azar (lo que se denomina hipótesis nula). Si esta probabilidad, denominada valor  $p$ , es superior a un umbral determinado (normalmente 5 %, controlado por un hiperparámetro), entonces el nodo se considera innecesario y sus hijos se eliminan. La poda continúa hasta que se hayan podado todos los nodos innecesarios.

Probemos la regularización en el conjunto de datos de lunas, presentado en [el Capítulo 5](#). Entrenaremos un árbol de decisión sin regularización y otro con `min_samples_leaf=5`. Aquí está el código; [La Figura 6-3](#) muestra los límites de decisión de cada árbol:

```
desde sklearn.datasets importar make_moons

X_lunas, y_lunas = make_moons(n_muestras=150, ruido=0.2, estado_aleatorio=42)

tree_clf1 = DecisionTreeClassifier(random_state=42) tree_clf2 =
DecisionTreeClassifier(min_samples_leaf=5, random_state=42) tree_clf1.fit(X_lunas, y_lunas)
tree_clf2.fit(X_lunas, y_lunas)
```

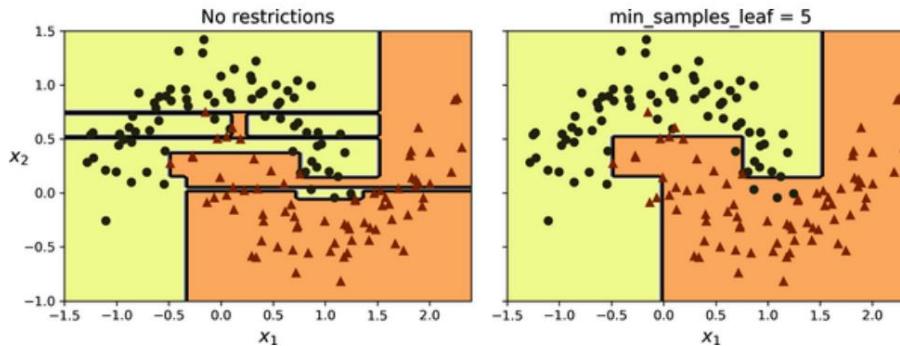


Figura 6-3. Límites de decisión de un árbol no regularizado (izquierda) y un árbol regularizado (derecha)

El modelo no regularizado de la izquierda está claramente sobreajustado, y el modelo regularizado de la derecha probablemente se generalizará mejor. Podemos verificar esto evaluando ambos árboles en un conjunto de prueba generado usando una semilla aleatoria diferente:

```
>>> X_moons_test, y_moons_test = make_moons(n_samples=1000, ruido=0.2, estado_aleatorio=43)
...
...
>>> tree_clf1.score(X_moons_test, y_moons_test) 0.898
>>> tree_clf2.score(X_moons_test, y_moons_test) 0.92
```

De hecho, el segundo árbol tiene una mayor precisión en el conjunto de prueba.

## Regresión

Los árboles de decisión también son capaces de realizar tareas de regresión. Construyamos un árbol de regresión usando la clase `DecisionTreeRegressor` de Scikit-Learn, entrenándolo en un conjunto de datos cuadrático ruidoso con `max_depth=2`:

```
importar numpy como np
desde sklearn.tree importar DecisionTreeRegressor

np.semilla.aleatoria(42)
X_quad = np.random.rand(200, 1) - 0.5 # una única característica de entrada aleatoria y_quad = X_quad
** 2 + 0.025 * np.random.randn(200, 1)
```

```
tree_reg = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg.fit(X_quad, y_quad)
```

El árbol resultante se representa en la Figura 6-4.

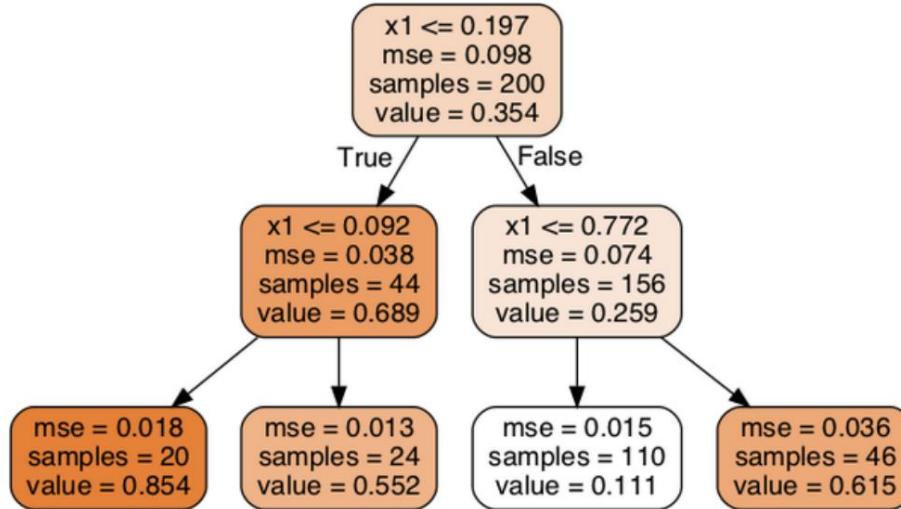


Figura 6-4. Un árbol de decisión para la regresión

Este árbol se parece mucho al árbol de clasificación que creó anteriormente. La principal diferencia es que en lugar de predecir una clase en cada nodo, predice un valor. Por ejemplo, supongamos que desea hacer una predicción para una nueva instancia con  $x = 0.2$ . El nodo raíz pregunta si  $x \leq 0.197$ . Como no es así, el algoritmo va al nodo secundario derecho, que pregunta si  $x \leq 0.772$ . Como es así, el algoritmo va al nodo secundario izquierdo. Este es un nodo hoja y predice un valor = 0,111. Esta predicción es el valor objetivo promedio de las 110 instancias de entrenamiento asociadas con este nodo hoja y da como resultado un error cuadrático medio igual a 0,015 en estas 110 instancias.

Las predicciones de este modelo se representan a la izquierda en la Figura 6-5. Si establece `max_depth = 3`, obtendrá las predicciones representadas a la derecha. Observe cómo el valor previsto para cada región es siempre el valor objetivo promedio de las instancias en esa región. El algoritmo divide cada región de manera que la mayoría de las instancias de entrenamiento se acerquen lo más posible al valor previsto.

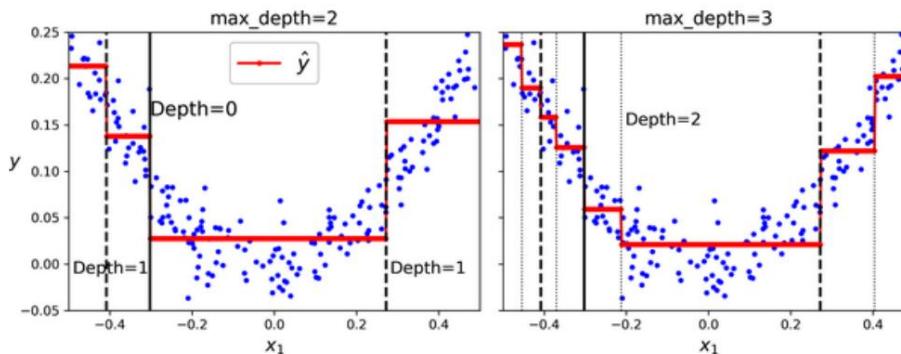


Figura 6-5. Predicciones de dos modelos de regresión de árboles de decisión.

El algoritmo CART funciona como se describió anteriormente, excepto que en lugar de intentar dividir el conjunto de entrenamiento de una manera que minimice la impureza, ahora intenta dividir el conjunto de entrenamiento de una manera que minimice el MSE. [La ecuación 6-4](#) muestra la función de costos que el algoritmo intenta minimizar.

Ecuación 6-4. Función de costo CART para regresión

$$J(k, t_k) = \frac{\text{izquierda}}{\text{metro}} \text{MSEizquierda} + \frac{\text{derecha}}{\text{metro}} \text{MSEjusto} \text{ donde}$$

$$\text{MSEnodo} = \frac{\sum_i \text{nodo} (\hat{y}^{\text{nodo}} - y(i))^2}{\text{nodo m}}$$

$$\text{y } \hat{y}^{\text{nodo}} = \frac{\sum_i \text{nodo} (i)}{\text{nodo m}}$$

Al igual que las tareas de clasificación, los árboles de decisión tienden a sobreajustarse cuando se trata de tareas de regresión. Sin ninguna regularización (es decir, utilizando los hiperparámetros predeterminados), se obtienen las predicciones a la izquierda en la [Figura 6-6](#). Obviamente, estas predicciones están sobreajustando muy mal el conjunto de entrenamiento. Simplemente establecer `min_samples_leaf=10` da como resultado un modelo mucho más razonable, representado a la derecha en la [Figura 6-6](#).

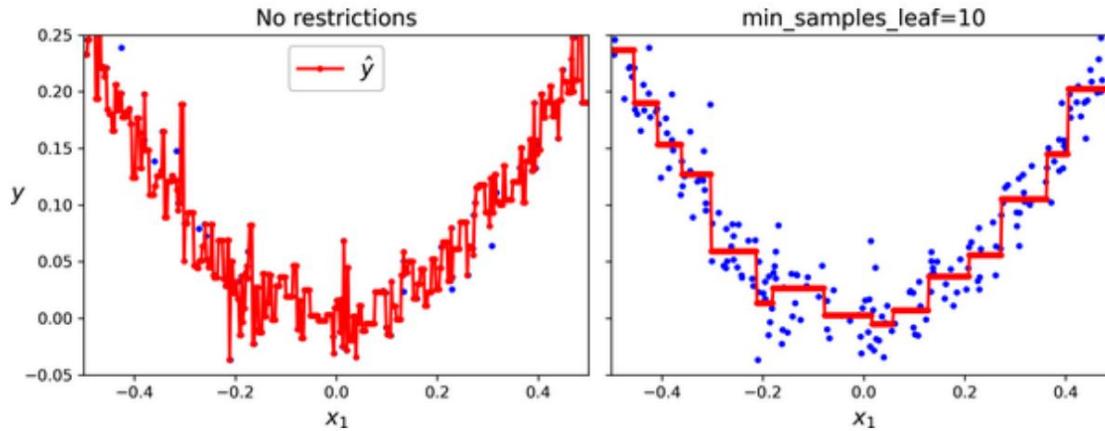


Figura 6-6. Predicciones de un árbol de regresión no regularizado (izquierda) y un árbol regularizado (derecha)

## Sensibilidad a la orientación del eje

Esperemos que a estas alturas ya esté convencido de que los árboles de decisión tienen mucho que ofrecer: son relativamente fáciles de entender e interpretar, fáciles de usar, versátiles y potentes. Sin embargo, tienen algunas limitaciones. Primero, como habrás notado, a los árboles de decisión les encantan los límites de decisión ortogonales (todas las divisiones son perpendiculares a un eje), lo que los hace sensibles a la orientación de los datos. Por ejemplo, [la Figura 6-7](#) muestra un conjunto de datos linealmente separable simple: a la izquierda, un árbol de decisión puede dividirlo fácilmente, mientras que a la derecha, después de girar el conjunto de datos 45°, el límite de decisión parece innecesariamente complicado. Aunque ambos árboles de decisión se ajustan perfectamente al conjunto de entrenamiento, es muy probable que el modelo de la derecha no se generalice bien.

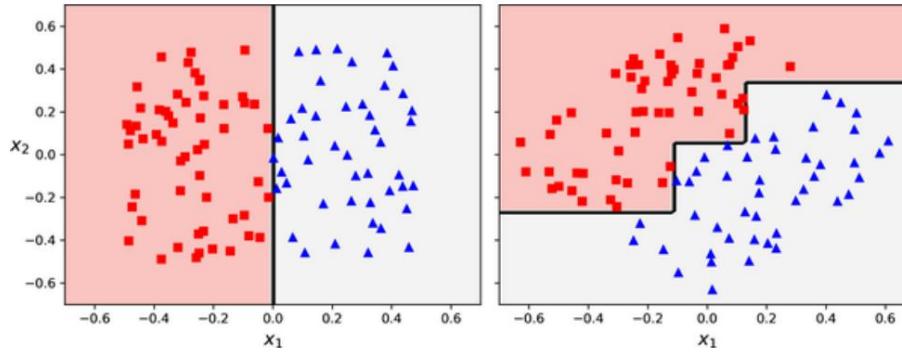


Figura 6-7. Sensibilidad a la rotación del conjunto de entrenamiento.

Una forma de limitar este problema es escalar los datos y luego aplicar una transformación de análisis de componentes principales. Analizaremos PCA en detalle en [el Capítulo 8](#), pero por ahora sólo necesita saber que rota los datos de una manera que reduce la correlación entre las características, lo que a menudo (no siempre) facilita las cosas para los árboles.

Creemos una pequeña canalización que escala los datos y los rote usando PCA, luego entrenemos un `DecisionTreeClassifier` con esos datos. [La Figura 6-8](#) muestra los límites de decisión de ese árbol: como puede ver, la rotación permite ajustar bastante bien el conjunto de datos usando solo una característica, que es una función lineal de la longitud y el ancho del pétalo original. Aquí está el código:

```
desde sklearn.decomposition importar PCA desde
sklearn.pipeline importar make_pipeline desde
sklearn.preprocessing importar StandardScaler

pca_pipeline = make_pipeline(StandardScaler(), PCA())
X_iris_rotated = pca_pipeline.fit_transform(X_iris) tree_clf_pca =
DecisionTreeClassifier(max_profundidad=2, random_state=42) tree_clf_pca.fit(X_iris_rotated,
y_iris)
```

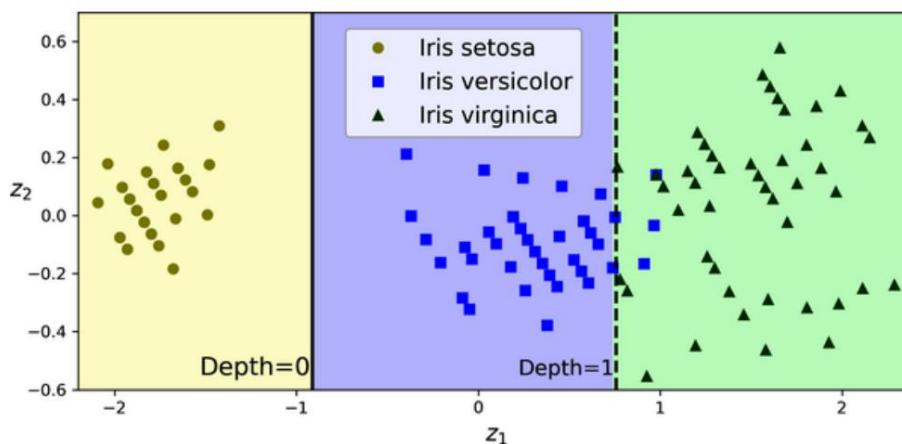


Figura 6-8. Los límites de decisión de un árbol en el conjunto de datos de iris escalado y rotado por PCA

**Los árboles de decisión tienen una alta variación**

En términos más generales, el principal problema de los árboles de decisión es que tienen una varianza bastante alta: pequeños cambios en los hiperparámetros o en los datos pueden producir modelos muy diferentes. De hecho, dado que el algoritmo de entrenamiento utilizado por Scikit-Learn es estocástico (selecciona aleatoriamente el conjunto de características para evaluar en cada nodo), incluso volver a entrenar el mismo árbol de decisión con exactamente los mismos datos puede producir un modelo muy diferente, como el representado en la Figura 6-9 (a menos que establezca el hiperparámetro `random_state`). Como puede ver, se ve muy diferente del árbol de decisión anterior (Figura 6-2).

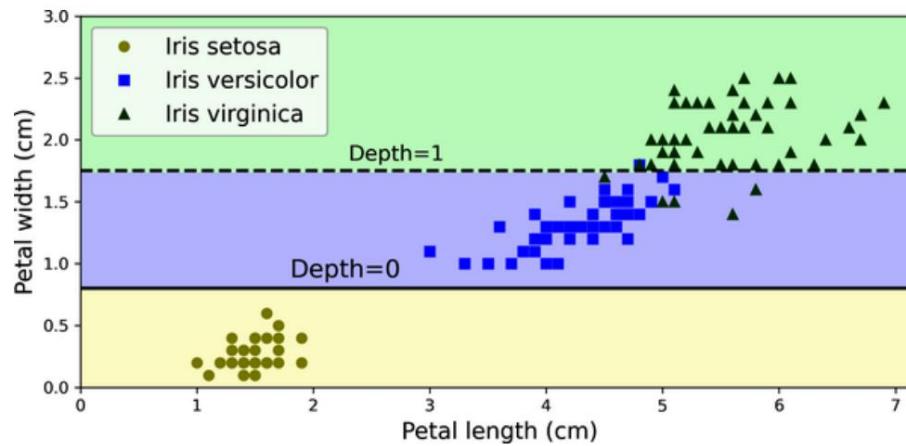


Figura 6-9. Volver a entrenar el mismo modelo con los mismos datos puede producir un modelo muy diferente

Afortunadamente, al promediar las predicciones de muchos árboles, es posible reducir significativamente la varianza. Este conjunto de árboles se denomina bosque aleatorio y es uno de los tipos de modelos más potentes disponibles en la actualidad, como verá en el próximo capítulo.

## Ejercicios

1. ¿Cuál es la profundidad aproximada de un árbol de decisión entrenado (sin restricciones) en un conjunto de entrenamiento con un millón de instancias?
2. ¿La impureza Gini de un nodo es generalmente menor o mayor que la de su padre? ¿Es generalmente más bajo/más alto, o siempre más bajo/más alto?
3. Si un árbol de decisión sobreajusta el conjunto de entrenamiento, ¿es una buena idea intentar disminuirlo? ¿máxima profundidad?
4. Si un árbol de decisión no se adapta adecuadamente al conjunto de entrenamiento, ¿es una buena idea intentar escalar la entrada? ¿características?
5. Si se necesita una hora para entrenar un árbol de decisiones en un conjunto de entrenamiento que contiene un millón de instancias, ¿cuánto tiempo aproximadamente llevará entrenar otro árbol de decisión en un conjunto de entrenamiento que contiene diez millones de instancias? Sugerencia: considere la complejidad computacional del algoritmo CART.
6. Si se necesita una hora para entrenar un árbol de decisión en un conjunto de entrenamiento determinado, ¿cuánto tiempo aproximadamente tomará si se duplica la cantidad de funciones?
7. Entrene y ajuste un árbol de decisión para el conjunto de datos de las lunas siguiendo estos pasos:

- a. Utilice `make_moons(n_samples=10000, noise=0.4)` para generar lunas conjunto de datos.
- b. Utilice `train_test_split()` para dividir el conjunto de datos en un conjunto de entrenamiento y un conjunto de prueba.
- C. Utilice la búsqueda de cuadrícula con validación cruzada (con la ayuda de la clase `GridSearchCV`) para encontrar buenos valores de hiperparámetros para un `DecisionTreeClassifier`. Sugerencia: pruebe con varios valores para `max_leaf_nodes`.
- d. Entrénelo en el conjunto de entrenamiento completo utilizando estos hiperparámetros y mida el rendimiento de su modelo en el conjunto de prueba. Debería obtener aproximadamente entre un 85% y un 87% de precisión.

8. Haga crecer un bosque siguiendo estos pasos:

- a. Continuando con el ejercicio anterior, genere 1000 subconjuntos del conjunto de entrenamiento, cada uno de los cuales contenga 100 instancias seleccionadas al azar. Sugerencia: puedes usar la clase `ShuffleSplit` de Scikit-Learn para esto.
- b. Entrene un árbol de decisión en cada subconjunto, utilizando los mejores valores de hiperparámetros encontrado en el ejercicio anterior. Evalúe estos 1000 árboles de decisión en el conjunto de prueba. Dado que fueron entrenados en conjuntos más pequeños, estos árboles de decisión probablemente funcionarán peor que el primer árbol de decisión, logrando solo alrededor del 80% de precisión.
- C. Ahora viene la magia. Para cada instancia del conjunto de pruebas, genere las predicciones de los 1000 árboles de decisión y mantenga solo la predicción más frecuente (puede usar la función `mode()` de SciPy para esto). Este enfoque le brinda predicciones de voto mayoritario sobre el conjunto de prueba.
- d. Evalúe estas predicciones en el conjunto de prueba: debería obtener una puntuación ligeramente mayor que su primer modelo (entre un 0,5 y un 1,5 % mayor). ¡Felicitaciones, ha entrenado un clasificador forestal aleatorio!

Las soluciones a estos ejercicios están disponibles al final del cuaderno de este capítulo, en <https://homl.info/colab3>.

---

<sup>1</sup> P es el conjunto de problemas que se pueden resolver en tiempo polinómico (es decir, un polinomio del tamaño del conjunto de datos). NP es el conjunto de problemas cuyas soluciones pueden verificarse en tiempo polinomial. Un problema NP-difícil es un problema que se puede reducir a un problema NP-difícil conocido en tiempo polinomial. Un problema NP-completo es tanto NP como NP-difícil. Una importante cuestión matemática abierta es si P = NP o no. Si P ≠ NP (lo que parece probable), entonces nunca se encontrará ningún algoritmo polinomial para ningún problema NP completo (excepto quizás algún día en una computadora cuántica).

<sup>2</sup> Véase [el interesante análisis](#) de Sebastian Raschka para más detalles.

# Capítulo 7. Aprendizaje conjunto y bosques aleatorios

---

Suponga que plantea una pregunta compleja a miles de personas al azar y luego agrega sus respuestas. En muchos casos encontrará que esta respuesta agregada es mejor que la respuesta de un experto. A esto se le llama la sabiduría de la multitud. De manera similar, si agrega las predicciones de un grupo de predictores (como clasificadores o regresores), a menudo obtendrá mejores predicciones que con el mejor predictor individual. Un grupo de predictores se llama conjunto; por lo tanto, esta técnica se denomina aprendizaje conjunto y un algoritmo de aprendizaje conjunto se denomina método conjunto.

Como ejemplo de método de conjunto, puede entrenar un grupo de clasificadores de árboles de decisión, cada uno en un subconjunto aleatorio diferente del conjunto de entrenamiento. Luego puede obtener las predicciones de todos los árboles individuales, y la clase que obtiene más votos es la predicción del conjunto (consulte el último ejercicio en el [Capítulo 6](#)). Este conjunto de árboles de decisión se denomina bosque aleatorio y, a pesar de su simplicidad, es uno de los algoritmos de aprendizaje automático más potentes disponibles en la actualidad.

Como se analizó en el [Capítulo 2](#), a menudo utilizará métodos de conjunto cerca del final de un proyecto, una vez que ya haya creado algunos buenos predictores, para combinarlos en un predictor aún mejor. De hecho, las soluciones ganadoras en concursos de aprendizaje automático a menudo implican varios métodos conjuntos, el más famoso en el [concurso del Premio Netflix](#).

En este capítulo examinaremos los métodos de conjuntos más populares, incluidos clasificadores de votación, conjuntos de embolsado y pegado, bosques aleatorios y conjuntos de refuerzo y apilamiento.

## Clasificadores de votación

Suponga que ha entrenado algunos clasificadores y cada uno logra aproximadamente un 80% de precisión. Es posible que tenga un clasificador de regresión logística, un clasificador SVM, un clasificador de bosque aleatorio, un clasificador de k vecinos más cercanos y quizás algunos más (consulte [la Figura 7-1](#)).

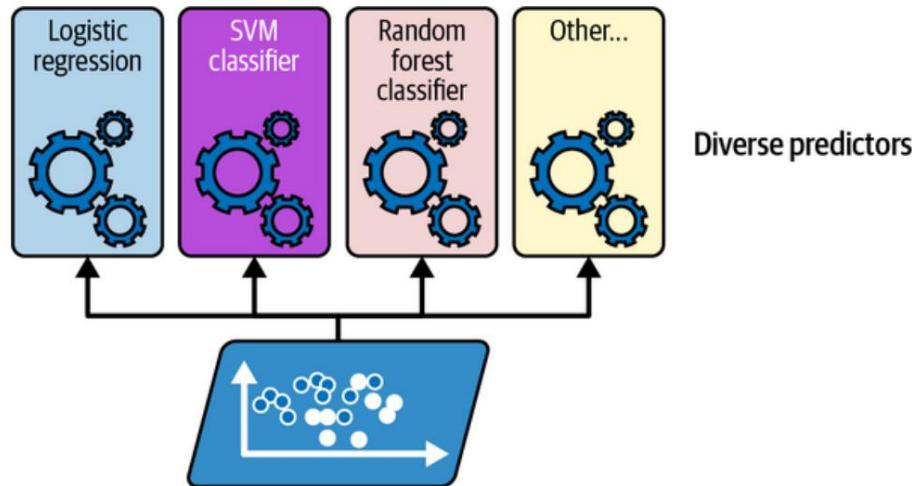


Figura 7-1. Entrenamiento de diversos clasificadores

Una forma muy sencilla de crear un clasificador aún mejor es agregar las predicciones de cada clasificador: la clase que obtiene la mayor cantidad de votos es la predicción del conjunto. Este clasificador de voto mayoritario se denomina clasificador de voto duro (consulte [la Figura 7-2](#)).

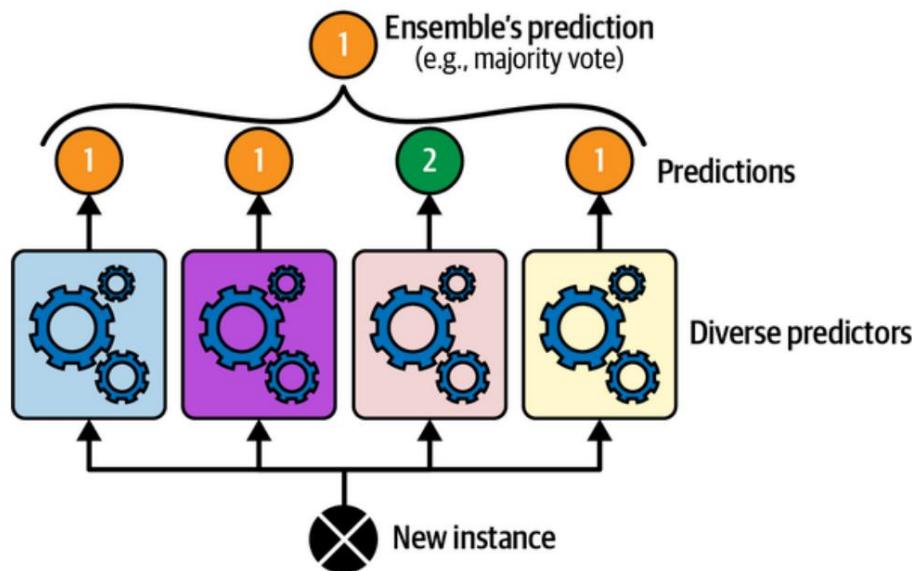


Figura 7-2. Predicciones del clasificador de votación dura

Sorprendentemente, este clasificador de votación a menudo logra una mayor precisión que el mejor clasificador del conjunto. De hecho, incluso si cada clasificador es un alumno débil (lo que significa que lo hace sólo un poco mejor que las adivinanzas aleatorias), el conjunto puede seguir siendo un alumno fuerte (logrando una alta precisión), siempre que haya un número suficiente de alumnos débiles en el conjunto y son suficientemente diversos.

¿Cómo es esto posible? La siguiente analogía puede ayudar a arrojar algo de luz sobre este misterio. Supongamos que tiene una moneda ligeramente sesgada que tiene un 51% de posibilidades de salir cara y un 49% de posibilidades de salir cruz. Si lo lanza 1000 veces, generalmente obtendrás más o menos 510 caras y 490 cruces y, por tanto, la mayoría de caras. Si haces los cálculos, encontrarás que la probabilidad de obtener la mayoría de caras después de 1.000 lanzamientos es cercana al 75%. Cuanto más

Cuando lanzas la moneda, mayor es la probabilidad (por ejemplo, con 10.000 lanzamientos, la probabilidad supera el 97%). Esto se debe a la ley de los grandes números: a medida que se sigue lanzando la moneda, la proporción de caras se acerca cada vez más a la probabilidad de que salga cara (51%). La figura 7-3 muestra 10 series de lanzamientos de monedas sesgados. Puedes ver que a medida que aumenta el número de lanzamientos, la proporción de caras se acerca al 51%. Al final, las 10 series terminan tan cerca del 51% que están constantemente por encima del 50%.

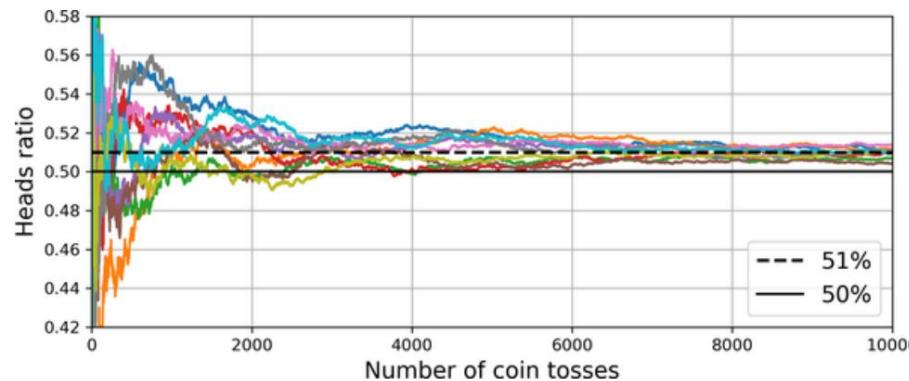


Figura 7-3. La ley de los grandes números.

De manera similar, supongamos que construye un conjunto que contiene 1000 clasificadores que son correctos individualmente solo el 51% de las veces (apenas mejor que las conjeturas aleatorias). Si predices la clase votada por mayoría, ¡puedes esperar una precisión de hasta el 75%! Sin embargo, esto sólo es cierto si todos los clasificadores son perfectamente independientes y cometan errores no correlacionados, lo que claramente no es el caso porque están entrenados con los mismos datos. Es probable que cometan el mismo tipo de errores, por lo que habrá muchos votos mayoritarios para la clase equivocada, lo que reducirá la precisión del conjunto.

#### CONSEJO

Los métodos de conjunto funcionan mejor cuando los predictores son lo más independientes posible entre sí. Una forma de conseguir clasificadores diversos es entrenarlos utilizando algoritmos muy diferentes. Esto aumenta la posibilidad de que cometan tipos de errores muy diferentes, mejorando la precisión del conjunto.

Scikit-Learn proporciona una clase `VotingClassifier` que es bastante fácil de usar: simplemente proporcionele una lista de pares de nombre/predictor y úsela como un clasificador normal. Probémoslo en el conjunto de datos de lunas (presentado en el Capítulo 5). Cargaremos y dividiremos el conjunto de datos de las lunas en un conjunto de entrenamiento y un conjunto de prueba, luego crearemos y entrenaremos un clasificador de votación compuesto por tres clasificadores diversos:

```
de sklearn.datasets importar make_moons de
sklearn.ensemble importar RandomForestClassifier, VotingClassifier de sklearn.linear_model importar
LogisticRegression de sklearn.model_selection importar train_test_split de
sklearn.svm importar SVC
```

```
X, y = hacer_lunas(n_muestras=500, ruido=0,30, estado_aleatorio=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
```

```

vote_clf = VotingClassifier( estimadores=[

    ('lr', LogisticRegression(random_state=42)), ('rf',
    RandomForestClassifier(random_state=42)), ('svc',
    SVC(random_state=42))
]

) votación_clf.fit(X_train, y_train)

```

Cuando ajustas un `VotingClassifier`, clona cada estimador y ajusta los clones. Los estimadores originales están disponibles a través del atributo `estimadores`, mientras que los clones ajustados están disponibles a través del atributo `estimadores_`. Si prefieres un dictado en lugar de una lista, puedes usar `llamados_estimadores` o `nombrados_estimadores_` en su lugar. Para comenzar, veamos la precisión de cada clasificador ajustado en el conjunto de prueba:

```

>>> para nombre, clf en vote_clf.named_estimators_.items(): print(nombre, "=",
...         clf.score(X_test, y_test))
...
I = 0,864
rf = 0,896
servicio = 0,896

```

Cuando llama al método `predict()` del clasificador de votación, realiza una votación estricta. Por ejemplo, el clasificador de votación predice la clase 1 para la primera instancia del conjunto de prueba, porque dos de cada tres clasificadores predicen esa clase:

```

>>> vote_clf.predict(X_test[:1]) array([1]) >>>
[clf.predict(X_test[:1]) para clf en vote_clf.estimators_] [array([1]), array([1]), matriz ([0])]

```

Ahora veamos el rendimiento del clasificador de votación en el conjunto de prueba:

```
>>> votación_clf.score(prueba_X, prueba_y) 0.912
```

¡Ahí tienes! El clasificador de votación supera a todos los clasificadores individuales.

Si todos los clasificadores pueden estimar las probabilidades de clase (es decir, si todos tienen un método `predict_proba()`), entonces puede decirle a Scikit-Learn que prediga la clase con la probabilidad de clase más alta, promediada entre todos los clasificadores individuales. A esto se le llama voto blando. A menudo logra un mayor rendimiento que la votación dura porque da más peso a los votos con mucha confianza. Todo lo que necesita hacer es configurar el hiperparámetro de votación del clasificador de votación en "suave" y asegurarse de que todos los clasificadores puedan estimar las probabilidades de clase. Este no es el caso de la clase SVC de forma predeterminada, por lo que debe establecer su hiperparámetro de probabilidad en Verdadero (esto hará que la clase SVC use validación cruzada para estimar las probabilidades de la clase, lo que ralentizará el entrenamiento y agregará un `predict_proba()` método). Probemos eso:

```
>>> vote_clf.voting = "soft" >>>
vote_clf.named_estimators["svc"].probabilidad = Verdadero >>>
vote_clf.fit(X_train, y_train) >>>
vote_clf.score(X_test, y_test) 0.92
```

Alcanzamos una precisión del 92 % simplemente utilizando la votación suave: ¡nada mal!

## Embolsar y pegar

Una forma de obtener un conjunto diverso de clasificadores es utilizar algoritmos de entrenamiento muy diferentes, como se acaba de comentar. Otro enfoque es utilizar el mismo algoritmo de entrenamiento para cada predictor, pero entrenarlos en diferentes subconjuntos aleatorios del conjunto de entrenamiento. Cuando el muestreo se realiza con reposición, este método se denomina **embolsado**. (abreviatura de agregación de arranque). Cuando <sup>1</sup>  
<sup>2</sup> El muestreo se realiza sin reemplazo, se llama **pegado**. <sup>3</sup>  
<sup>4</sup>

En otras palabras, tanto el empaquetado como el pegado permiten que las instancias de entrenamiento se muestren varias veces en múltiples predictores, pero solo el empaquetado permite que las instancias de entrenamiento se muestren varias veces para el mismo predictor. Este proceso de muestreo y capacitación se representa en la Figura 7-4.

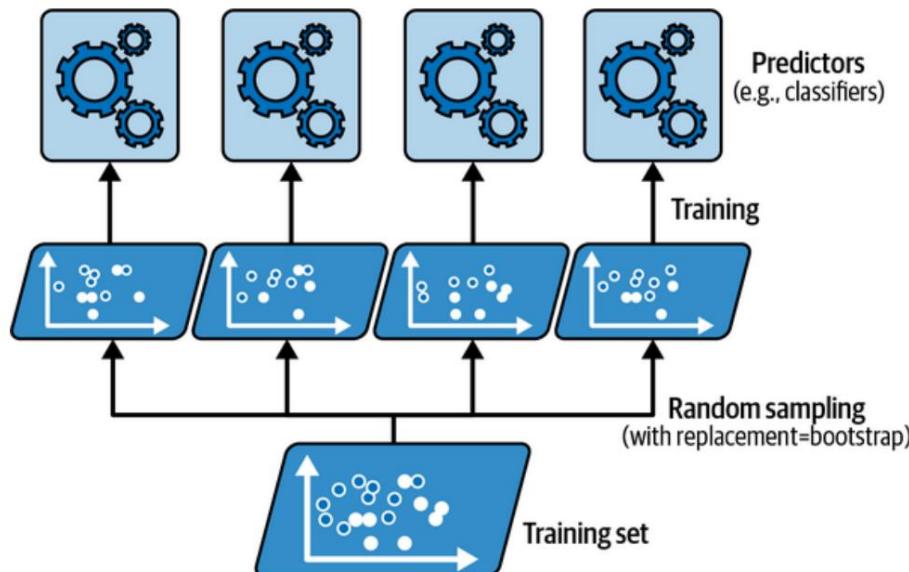


Figura 7-4. Embolsar y pegar implica entrenar varios predictores en diferentes muestras aleatorias del conjunto de entrenamiento.

Una vez que todos los predictores estén entrenados, el conjunto puede hacer una predicción para una nueva instancia simplemente agregando las predicciones de todos los predictores. La función de agregación suele ser el modo estadístico para la clasificación (es decir, la predicción más frecuente, como ocurre con un clasificador de votación dura), o el promedio para la regresión. Cada predictor individual tiene un sesgo mayor que si estuviera entrenado en el conjunto de entrenamiento original, pero la agregación reduce tanto el sesgo como la varianza. Generalmente, el resultado neto es que el conjunto tiene un sesgo similar pero una varianza menor que un único predictor entrenado en el conjunto de entrenamiento original. <sup>5</sup>

Como puede ver en [la Figura 7-4](#), todos los predictores se pueden entrenar en paralelo, a través de diferentes núcleos de CPU o incluso diferentes servidores. De manera similar, se pueden hacer predicciones en paralelo. Ésta es una de las razones por las que embolsar y pegar son métodos tan populares: se adaptan muy bien.

## Embolsar y pegar en Scikit-Learn

Scikit-Learn ofrece una API simple para empaquetar y pegar: clase BaggingClassifier (o BaggingRegressor para regresión). El siguiente código entrena un conjunto de 500 clasificadores de árboles de decisión: cada uno se entrena en 100 instancias de <sup>6</sup>entrenamiento tomadas aleatoriamente del conjunto de entrenamiento con reemplazo (este es un ejemplo de embolsado, pero si desea utilizar el pegado en su lugar, simplemente configure bootstrap=False) . El parámetro n\_jobs le dice a Scikit-Learn la cantidad de núcleos de CPU que se usarán para el entrenamiento y las predicciones, y -1 le dice a Scikit-Learn que use todos los núcleos disponibles:

```
desde sklearn.ensemble importar BaggingClassifier desde sklearn.tree
importar DecisionTreeClassifier
```

```
bag_clf = BaggingClassifier(DecisionTreeClassifier(), n_estimators=500,
                           max_samples=100, n_jobs=-1, random_state=42) bag_clf.fit(X_train,
y_train)
```

### NOTA

Un BaggingClassifier realiza automáticamente una votación suave en lugar de una votación dura si el clasificador base puede estimar las probabilidades de clase (es decir, si tiene un método predict\_proba()), que es el caso de los clasificadores de árbol de decisión.

[La Figura 7-5](#) compara el límite de decisión de un único árbol de decisión con el límite de decisión de un conjunto de 500 árboles (del código anterior), ambos entrenados en el conjunto de datos de las lunas. Como puede ver, las predicciones del conjunto probablemente se generalizarán mucho mejor que las predicciones del árbol de decisión único: el conjunto tiene un sesgo comparable pero una varianza menor (comete aproximadamente el mismo número de errores en el conjunto de entrenamiento, pero el límite de decisión es menor). irregular).

El embolsado introduce un poco más de diversidad en los subconjuntos en los que se entrena cada predictor, por lo que el embolsado termina con un sesgo ligeramente mayor que el pegado; pero la diversidad adicional también significa que los predictores terminan estando menos correlacionados, por lo que la varianza del conjunto se reduce. En general, el embolsado suele dar como resultado mejores modelos, lo que explica por qué generalmente se prefiere. Pero si tiene tiempo libre y potencia de CPU, puede utilizar la validación cruzada para evaluar tanto el embolsado como el pegado y seleccionar el que funcione mejor.

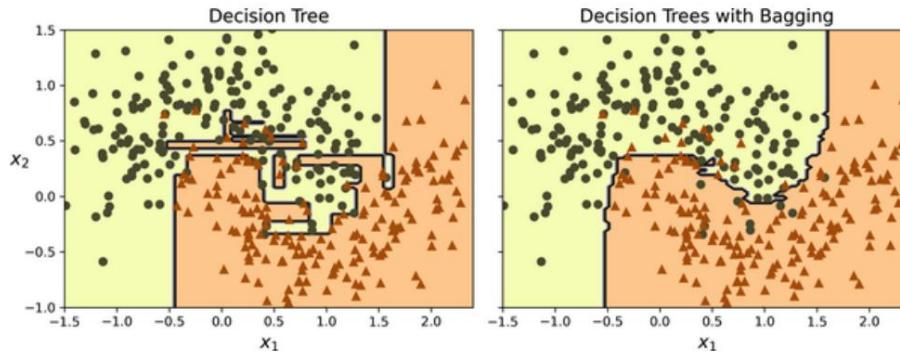


Figura 7-5. Un único árbol de decisión (izquierda) versus un conjunto de embolsado de 500 árboles (derecha)

## Evaluación fuera de bolsa

Con el embolsado, es posible que algunas instancias de entrenamiento se muestren varias veces para cualquier predictor determinado, mientras que es posible que otras no se muestren en absoluto. De forma predeterminada, un BaggingClassifier muestra  $m$  instancias de entrenamiento con reemplazo (`bootstrap=True`), donde  $m$  es el tamaño del conjunto de entrenamiento. Con este proceso, se puede demostrar matemáticamente que, en promedio, solo alrededor del 63% de las instancias de entrenamiento se muestran <sup>7</sup> para cada predictor. El 37% restante de las instancias de capacitación que no se muestran se denominan instancias listas para usar (OOB). Tenga en cuenta que no son los mismos 37% para todos los predictores.

Un conjunto de ensacado se puede evaluar usando instancias OOB, sin la necesidad de un conjunto de validación separado: de hecho, si hay suficientes estimadores, entonces cada instancia en el conjunto de entrenamiento probablemente será una instancia OOB de varios estimadores, por lo que estos estimadores se pueden usar para hacer una predicción conjunta justa para ese caso. Una vez que tenga una predicción para cada instancia, puede calcular la precisión de la predicción del conjunto (o cualquier otra métrica).

En Scikit-Learn, puede configurar `oob_score=True` al crear un BaggingClassifier para solicitar una evaluación OOB automática después del entrenamiento. El siguiente código demuestra esto.

La puntuación de evaluación resultante está disponible en el atributo `oob_score_`:

```
>>> bag_clf = BaggingClassifier(DecisionTreeClassifier(), n_estimators=500, oob_score=True, n_jobs=-1,
...                                random_state=42)
...
>>> bag_clf.fit(X_train, y_train) >>>
bag_clf.oob_score_ 0.896
```

Según esta evaluación OOB, es probable que este BaggingClassifier logre aproximadamente un 89,6% de precisión en el conjunto de prueba. Verifiquemos esto:

```
>>> from sklearn.metrics import precision_score >>>
y_pred = bag_clf.predict(X_test) >>>
precision_score(y_test, y_pred) 0.92
```

Obtenemos un 92% de precisión en la prueba. La evaluación OOB fue demasiado pesimista, poco más del 2% por debajo.

La función de decisión OOB para cada instancia de entrenamiento también está disponible a través del atributo `oob_decision_function_`. Dado que el estimador base tiene un método `predict_proba()`, la función de decisión devuelve las probabilidades de clase para cada instancia de entrenamiento. Por ejemplo, la evaluación OOB estima que la primera instancia de entrenamiento tiene un 67,6% de probabilidad de pertenecer a la clase positiva y un 32,4% de probabilidad de pertenecer a la clase negativa:

```
>>> bag_clf.oob_decision_function_[:3] # probas para las primeras 3 instancias de la matriz ([[0.32352941,
0.67647059], [0.3375 ], [1. ]])
      , 0.6625
      , 0.
```

## Parches aleatorios y subespacios aleatorios

La clase `BaggingClassifier` también admite el muestreo de características. El muestreo está controlado por dos hiperparámetros: `max_features` y `bootstrap_features`. Funcionan de la misma manera que `max_samples` y `bootstrap`, pero para muestreo de características en lugar de muestreo de instancias. Por lo tanto, cada predictor se entrenará en un subconjunto aleatorio de características de entrada.

Esta técnica es particularmente útil cuando se trata de entradas de alta dimensión (como imágenes), ya que puede acelerar considerablemente el entrenamiento. El muestreo de instancias y características de entrenamiento se denomina método [de parches aleatorios](#)<sup>8</sup>. Mantener todas las instancias de entrenamiento (estableciendo `bootstrap=False` y `max_samples=1.0`) pero muestreando características (estableciendo `bootstrap_features` en `True` y/o `max_features` en un valor menor que 1.0) se denomina método de [subespacios aleatorios](#).

Las características de muestreo dan como resultado una diversidad de predictores aún mayor, intercambiando un poco más de sesgo por una varianza más baja.

## Bosques aleatorios

Como hemos comentado, un [bosque aleatorio](#)<sup>10</sup> es un conjunto de árboles de decisión, generalmente entrenados mediante el método de embolsado (o, a veces, pegado), normalmente con `max_samples` establecido en el tamaño del conjunto de entrenamiento. En lugar de crear un `BaggingClassifier` y pasarlo un `DecisionTreeClassifier`, puede usar la clase `RandomForestClassifier`, que es más conveniente y está optimizada para árboles de decisión (de manera similar, existe<sup>11</sup> una clase `RandomForestRegressor` para tareas de regresión). El siguiente código entrena un clasificador de bosque aleatorio con 500 árboles, cada uno limitado a un máximo de 16 nodos hoja, utilizando todos los núcleos de CPU disponibles:

```
de sklearn.ensemble importar RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16,
                                  n_trabajos=-1, estado_aleatorio=42)
rnd_clf.fit(tren_X, tren_y)

y_pred_rf = rnd_clf.predict(X_test)
```

Con algunas excepciones, un RandomForestClassifier tiene todos los hiperparámetros de un DecisionTreeClassifier (para controlar cómo crecen los árboles), además de todos los hiperparámetros de un BaggingClassifier para controlar el conjunto en sí.

El algoritmo de bosque aleatorio introduce una aleatoriedad adicional al cultivar árboles; en lugar de buscar la mejor característica al dividir un nodo (ver [Capítulo 6](#)), busca la mejor característica entre un subconjunto aleatorio de características. De forma predeterminada, muestra  $\sqrt{n}$  características (donde  $n$  es el número total de características). El algoritmo da como resultado una mayor diversidad de árboles, lo que (nuevamente) intercambia un mayor sesgo por una menor varianza, lo que generalmente produce un mejor modelo en general.

Entonces, el siguiente BaggingClassifier es equivalente al anterior Clasificador de bosque aleatorio:

```
bag_clf = Clasificador de embolsado(
    DecisionTreeClassifier(max_features="sqrt", max_leaf_nodes=16), n_estimators=500,
    n_jobs=-1, random_state=42)
```

### Árboles adicionales

Cuando se cultiva un árbol en un bosque aleatorio, en cada nodo solo se considera para dividir un subconjunto aleatorio de características (como se analizó anteriormente). Es posible hacer que los árboles sean aún más aleatorios utilizando también umbrales aleatorios para cada característica en lugar de buscar los mejores umbrales posibles (como lo hacen los árboles de decisión normales). Para esto, simplemente configure splitter="random" al crear un DecisionTreeClassifier.

Un bosque de árboles tan extremadamente aleatorios se llama **árboles extremadamente aleatorios**. (o árboles <sup>12</sup> adicionales para abreviar) conjunto. Una vez más, esta técnica cambia más sesgo por una varianza más baja.

También hace que los clasificadores de árboles adicionales sean mucho más rápidos de entrenar que los bosques aleatorios normales, porque encontrar el mejor umbral posible para cada característica en cada nodo es una de las tareas que más tiempo consumen al hacer crecer un árbol.

Puede crear un clasificador de árboles adicionales utilizando la clase ExtraTreesClassifier de Scikit-Learn. Su API es idéntica a la clase RandomForestClassifier, excepto que el valor predeterminado de arranque es False. De manera similar, la clase ExtraTreesRegressor tiene la misma API que la clase RandomForestRegressor, excepto que el valor predeterminado de arranque es False.

#### CONSEJO

Es difícil saber de antemano si RandomForestClassifier funcionará mejor o peor que ExtraTreesClassifier. Generalmente, la única forma de saberlo es probar ambos y compararlos mediante validación cruzada.

### Importancia de la característica

Otra gran cualidad de los bosques aleatorios es que facilitan la medición de la importancia relativa de cada característica. Scikit-Learn mide la importancia de una característica observando cuánto reducen la impureza los nodos del árbol que usan esa característica en promedio, en todos los árboles del

bosque. Más precisamente, es un promedio ponderado, donde el peso de cada nodo es igual al número de muestras de entrenamiento asociadas con él (consulte el [Capítulo 6](#)).

Scikit-Learn calcula esta puntuación automáticamente para cada característica después del entrenamiento, luego escala los resultados para que la suma de todas las importancias sea igual a 1. Puede acceder al resultado utilizando la variable `feature_importances_`. Por ejemplo, el siguiente código entrena un `RandomForestClassifier` en el conjunto de datos del iris (presentado en el [Capítulo 4](#)) y genera la importancia de cada característica. Parece que las características más importantes son la longitud del pétalo (44%) y el ancho (42%), mientras que la longitud y el ancho del sépalo son poco importantes en comparación (11% y 2%, respectivamente):

```
>>> de sklearn.datasets importar load_iris >>> iris =
load_iris(as_frame=True) >>> rnd_clf =
RandomForestClassifier(n_estimators=500, random_state=42) >>> rnd_clf.fit(iris.data, iris.target)
>>> para puntuación, nombre en
zip(rnd_clf.feature_importances_, iris.data.columns):
...     imprimir(ronda(puntuación, 2), nombre)
...
0,11 largo de sépalo (cm) 0,02
ancho de sépalo (cm) 0,44
largo de pétalo (cm) 0,42 ancho
de pétalo (cm)
```

De manera similar, si entrena un clasificador de bosque aleatorio en el conjunto de datos MNIST (presentado en el [Capítulo 3](#)) y traza la importancia de cada píxel, obtendrá la imagen representada en la [Figura 7-6](#).

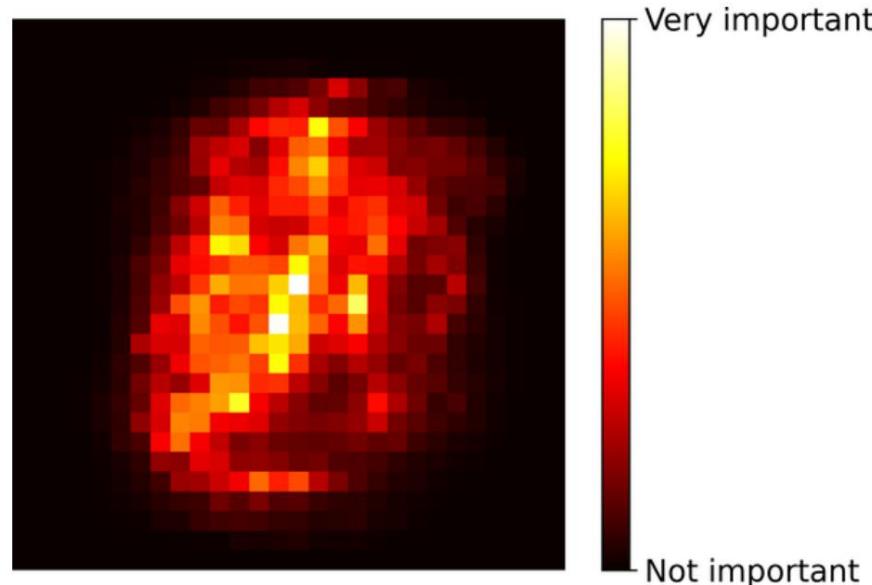


Figura 7-6. Importancia de píxeles MNIST (según un clasificador de bosque aleatorio)

Los bosques aleatorios son muy útiles para comprender rápidamente qué características realmente importan, en particular si necesita realizar una selección de características.

## Impulsando

El refuerzo (originalmente llamado refuerzo de hipótesis) se refiere a cualquier método conjunto que pueda combinar varios alumnos débiles en un alumno fuerte. La idea general de la mayoría de los métodos de impulso es entrenar predictores de forma secuencial, cada uno intentando corregir a su predecesor. Hay muchos métodos de refuerzo disponibles, pero los más populares son, con diferencia, AdaBoost. (abreviatura de <sup>13</sup>impulso adaptativo) y refuerzo de gradiente. Comencemos con AdaBoost.

## AdaBoost

Una forma de que un nuevo predictor corrija a su predecesor es prestar un poco más de atención a las instancias de entrenamiento que el predecesor no cumple con los requisitos. Esto da como resultado nuevos predictores que se centran cada vez más en los casos difíciles. Esta es la técnica utilizada por AdaBoost.

Por ejemplo, cuando se entrena un clasificador AdaBoost, el algoritmo primero entrena un clasificador base (como un árbol de decisión) y lo utiliza para hacer predicciones en el conjunto de entrenamiento. Luego, el algoritmo aumenta el peso relativo de las instancias de entrenamiento mal clasificadas. Luego entrena un segundo clasificador usando los pesos actualizados y nuevamente hace predicciones en el conjunto de entrenamiento, actualiza los pesos de las instancias, etc. (consulte la Figura 7-7).

La Figura 7-8 muestra los límites de decisión de cinco predictores consecutivos en el conjunto de datos de las lunas (en este ejemplo, cada predictor es un clasificador SVM altamente regularizado con un núcleo RBF). El primer clasificador se equivoca en muchas ocasiones, por lo que sus pesos aumentan. Por lo tanto, el segundo clasificador hace un mejor trabajo en estas instancias, y así sucesivamente. El gráfico de la derecha representa la misma secuencia de predictores, excepto que la tasa de aprendizaje se reduce a la mitad (es decir, los pesos de las instancias mal clasificadas aumentan mucho menos en cada iteración). Como puede ver, esta técnica de aprendizaje secuencial tiene algunas similitudes con el descenso de gradiente, excepto que en lugar de ajustar los parámetros de un único predictor para minimizar una función de costo, AdaBoost agrega predictores al conjunto, mejorándolo gradualmente.

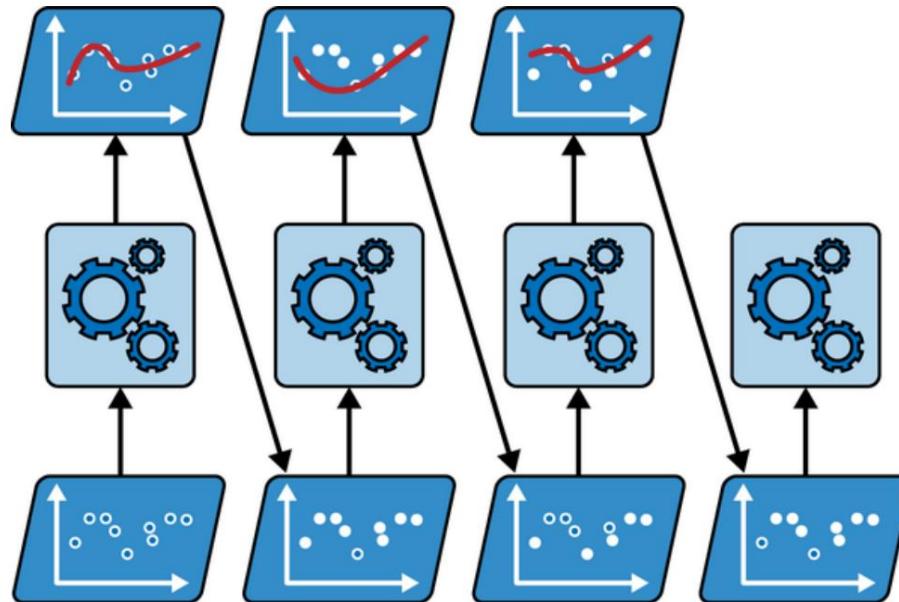


Figura 7-7. Entrenamiento secuencial AdaBoost con actualizaciones de peso de instancias

Una vez que todos los predictores están entrenados, el conjunto hace predicciones de manera muy similar a embolsar o pegar, excepto que los predictores tienen diferentes pesos dependiendo de su precisión general en

el conjunto de entrenamiento ponderado.

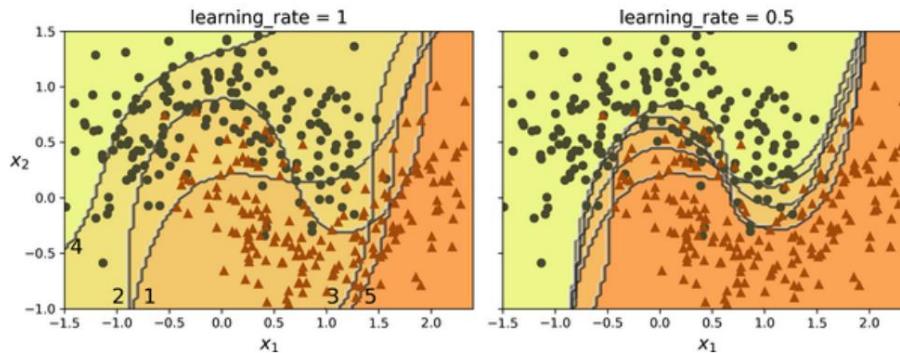


Figura 7-8. Límites de decisión de predictores consecutivos.

#### ADVERTENCIA

Hay un inconveniente importante en esta técnica de aprendizaje secuencial: el entrenamiento no se puede paralelizar ya que cada predictor sólo puede entrenarse después de que el predictor anterior haya sido entrenado y evaluado. Como resultado, no se escala tan bien como el embolsado o el pegado.

Echemos un vistazo más de cerca al algoritmo AdaBoost. El peso de cada instancia  $w$  se establece inicialmente en  $1/m$ . Se entrena un primer predictor y su tasa de error ponderada  $r$  se calcula en el conjunto de entrenamiento; ver [Ecuación 7-1](#).

Ecuación 7-1. Tasa de error ponderada del predictor  $j^{th}$

$$r_j = \frac{1}{m} \sum_{i=1}^m w(i) \text{ donde } y^{\hat{}}_j(i) \text{ es el } j^{th} \text{ predicción del predictor para el } i^{th} \text{ instancia}$$

$y^{\hat{}}_j(i) \neq y(i)$

Luego, el peso  $\alpha$  del predictor se calcula utilizando la [Ecuación 7-2](#), donde  $\eta$  es el hiperparámetro  $j$  de la tasa de aprendizaje (el valor predeterminado es  $\frac{1}{15}$ ). Cuanto más preciso sea el predictor, mayor será su peso. Si simplemente adivina al azar, entonces su peso será cercano a cero. Sin embargo, si la mayoría de las veces es incorrecta (es decir, es menos precisa que una conjetura aleatoria), entonces su peso será negativo.

Ecuación 7-2. Peso predictivo

$$\alpha_j = \eta \ln \frac{1 - r_j}{r_j}$$

A continuación, el algoritmo AdaBoost actualiza los pesos de las instancias mediante la [ecuación 7-3](#), que aumenta los pesos de las instancias mal clasificadas.

Ecuación 7-3. Regla de actualización de peso

$$\begin{aligned}
 & \text{para } i = 1, 2, \dots, m \\
 w^{(i)} &= y^{(i)} \frac{(j)}{\sum_{l=1}^m y^{(l)}} \text{ si } y^{(i)} \neq y^{(j)} \\
 &\leftarrow \{w^{(i)} \exp(\alpha_j) \text{ si } y^{(i)} \neq y^{(j)}\}
 \end{aligned}$$

Luego, todos los pesos de las instancias se normalizan (es decir, se dividen por  $\sum_{y_0=1}^m$  Wisconsin)).

Finalmente, se entrena un nuevo predictor utilizando los pesos actualizados y se repite todo el proceso: se calcula el peso del nuevo predictor, se actualizan los pesos de las instancias, luego se entrena otro predictor, y así sucesivamente. El algoritmo se detiene cuando se alcanza el número deseado de predictores o cuando se encuentra un predictor perfecto.

Para hacer predicciones, AdaBoost simplemente calcula las predicciones de todos los predictores y las pesa utilizando los pesos de los predictores  $\alpha$ . La clase prevista es la que recibe la mayoría de votos ponderados (ver [Ecuación 7-4](#)).

Ecuación 7-4. Predicciones de AdaBoost

$$\hat{y}(x) = \operatorname{argmax}_k \sum_{j=1}^N \alpha_j \text{ donde } N \text{ es el número de predictores}$$

Scikit-Learn utiliza una versión multiclase de AdaBoost llamada **SAMME** (que significa Modelado aditivo por etapas utilizando una función de pérdida exponencial multiclase). Cuando solo hay dos clases, SAMME equivale a AdaBoost. Si los predictores pueden estimar las probabilidades de clase (es decir, si tienen un método `predict_proba()`), Scikit-Learn puede usar una variante de SAMME llamada **SAMME.R** (la R significa "Real"), que se basa en probabilidades de clase en lugar de predicciones y generalmente funciona mejor.

El siguiente código entrena un clasificador AdaBoost basado en 30 muñones de decisión utilizando la clase `AdaBoostClassifier` de Scikit-Learn (como es de esperar, también hay una clase `AdaBoostRegressor`). Un muñón de decisión es un árbol de decisión con `profundidad_máxima=1`; en otras palabras, un árbol compuesto por un único nodo de decisión más dos nodos hoja. Este es el estimador base predeterminado para la clase `AdaBoostClassifier`:

```

desde sklearn.ensemble importar AdaBoostClassifier

ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(máx_profundidad=1), n_estimadores=30,
    tasa_de_aprendizaje=0.5, estado_aleatorio=42)
ada_clf.fit(X_train, y_train)

```

#### CONSEJO

Si su conjunto AdaBoost está sobreajustando el conjunto de entrenamiento, puede intentar reducir la cantidad de estimadores o regularizar más fuertemente el estimador base.

## Aumento de gradiente

Otro algoritmo de impulso muy popular es el **impulso de gradiente**. Al igual que AdaBoost, el aumento de gradiente funciona agregando predictores secuencialmente a un conjunto, cada uno corrigiendo a su predecesor. Sin embargo, en lugar de ajustar los pesos de las instancias en cada iteración como lo hace AdaBoost, este método intenta ajustar el nuevo predictor a los errores residuales cometidos por el predictor anterior.

Veamos un ejemplo de regresión simple, utilizando árboles de decisión como predictores base; esto se denomina refuerzo de árbol de gradiente o árboles de regresión impulsados por gradiente (GBRT). Primero, generaremos un conjunto de datos cuadrático ruidoso y le ajustemos un DecisionTreeRegressor:

```
importar numpy como np
desde sklearn.tree importar DecisionTreeRegressor

np.semilla.aleatoria(42)
X = np.random.rand(100, 1) - 0,5 y = 3 * X[:, 0] **
2 + 0,05 * np.random.randn(100) # y = 3x2 + ruido gaussiano

tree_reg1 = DecisionTreeRegressor(max_profundidad=2, random_state=42) tree_reg1.fit(X, y)
```

A continuación, entrenaremos un segundo DecisionTreeRegressor sobre los errores residuales cometidos por el primer predictor:

```
y2 = y - tree_reg1.predict(X) tree_reg2 =
DecisionTreeRegressor(max_profundidad=2, random_state=43) tree_reg2.fit(X, y2)
```

Y luego entrenaremos un tercer regresor sobre los errores residuales cometidos por el segundo predictor:

```
y3 = y2 - tree_reg2.predict(X) tree_reg3 =
DecisionTreeRegressor(max_profundidad=2, random_state=44) tree_reg3.fit(X, y3)
```

Ahora tenemos un conjunto que contiene tres árboles. Puede hacer predicciones sobre una nueva instancia simplemente sumando las predicciones de todos los árboles:

```
>>> X_new = np.array([-0.4], [0], [0.5]) >>>
suma(tree.predict(X_new) para árbol en (tree_reg1, tree_reg2, tree_reg3)) array([ 0,49484029, 0,04021166,
0,75026781])
```

La Figura 7-9 representa las predicciones de estos tres árboles en la columna de la izquierda y las predicciones del conjunto en la columna de la derecha. En la primera fila, el conjunto tiene un solo árbol, por lo que sus predicciones son exactamente las mismas que las del primer árbol. En la segunda fila, se entrena un nuevo árbol sobre los errores residuales del primer árbol. A la derecha puedes ver que las predicciones del conjunto son iguales a la suma de las predicciones de los dos primeros árboles.

De manera similar, en la tercera fila se entrena otro árbol con los errores residuales del segundo árbol.

Puede ver que las predicciones del conjunto mejoran gradualmente a medida que se agregan árboles al conjunto.

Puede utilizar la clase GradientBoostingRegressor de Scikit-Learn para entrenar conjuntos GBRT más fácilmente (también hay una clase GradientBoostingClassifier para clasificación). Al igual que la clase RandomForestRegressor, tiene hiperparámetros para controlar el crecimiento de los árboles de decisión (por ejemplo, `max_depth`, `min_samples_leaf`), así como hiperparámetros para controlar el entrenamiento del conjunto, como el número de árboles (`n_estimators`). El siguiente código crea el mismo conjunto que el anterior:

```
desde sklearn.ensemble importar GradientBoostingRegressor

gbdt = GradientBoostingRegressor(max_depth=2, n_estimators=3,
                                  learning_rate=1.0, random_state=42)
gbdt.fit(X, y)
```

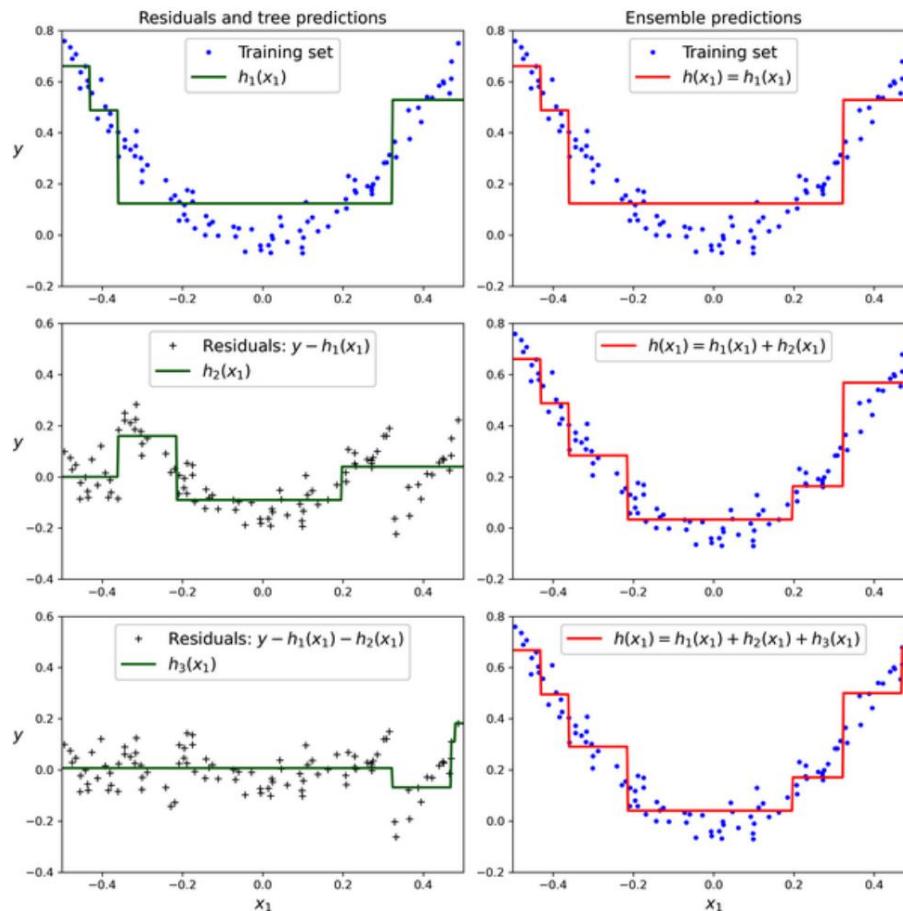


Figura 7-9. En esta representación del aumento de gradiente, el primer predictor (arriba a la izquierda) se entrena normalmente, luego cada predictor consecutivo (centro izquierdo e inferior izquierdo) se entrena con los residuos del predictor anterior; la columna de la derecha muestra las predicciones del conjunto resultante

El hiperparámetro `learning_rate` escala la contribución de cada árbol. Si lo establece en un valor bajo, como 0,05, necesitará más árboles en el conjunto para ajustarse al conjunto de entrenamiento, pero las predicciones normalmente se generalizarán mejor. Esta es una técnica de regularización llamada contracción. La Figura 7-10 muestra dos conjuntos de GBRT entrenados con diferentes hiperparámetros: el de la izquierda no tiene suficientes árboles para ajustarse al conjunto de entrenamiento, mientras que el de la izquierda

El derecho tiene aproximadamente la cantidad correcta. Si agregamos más árboles, el GBRT comenzaría a sobreajustar el conjunto de entrenamiento.

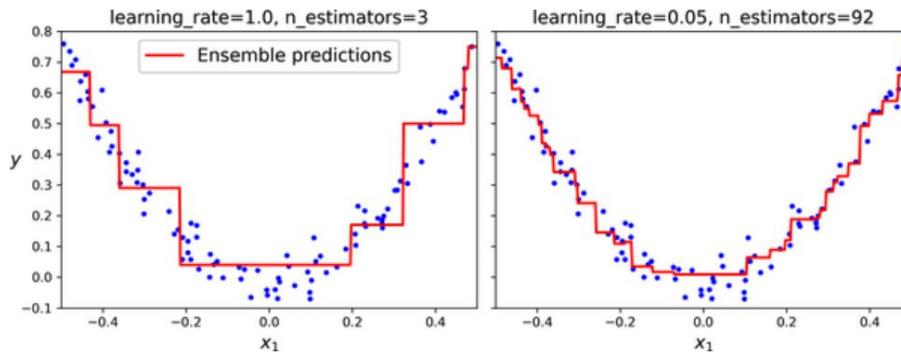


Figura 7-10. Conjuntos GBRT con insuficientes predictores (izquierda) y suficientes (derecha)

Para encontrar la cantidad óptima de árboles, puede realizar una validación cruzada usando GridSearchCV o RandomizedSearchCV, como de costumbre, pero hay una forma más sencilla: si configura el hiperparámetro `n_iter_no_change` en un valor entero, digamos 10, entonces el `GradientBoostingRegressor` dejará automáticamente de agregar más árboles durante el entrenamiento si ve que los últimos 10 árboles no ayudaron. Esto es simplemente una parada temprana (introducida en el Capítulo 4), pero con un poco de paciencia: tolera no tener progreso durante algunas iteraciones antes de detenerse. Entrenemos al conjunto usando paradas tempranas:

```
gbdt_best = GradientBoostingRegressor(profundidad_máxima=2,
                                       tasa_de_aprendizaje=0.05, n_estimadores=500, n_iter_no_change=10,
                                       estado_aleatorio=42) gbdt_best.fit(X, y)
```

Si establece `n_iter_no_change` demasiado bajo, el entrenamiento puede detenerse demasiado pronto y el modelo no se ajustará bien. Pero si lo configuras demasiado alto, se ajustará demasiado. También establecimos una tasa de aprendizaje bastante pequeña y una gran cantidad de estimadores, pero la cantidad real de estimadores en el conjunto entrenado es mucho menor, gracias a la detención temprana:

```
>>> gbdt_best.n_estimadores_ 92
```

Cuando se establece `n_iter_no_change`, el método `fit()` divide automáticamente el conjunto de entrenamiento en un conjunto de entrenamiento más pequeño y un conjunto de validación: esto le permite evaluar el rendimiento del modelo cada vez que agrega un nuevo árbol. El tamaño del conjunto de validación está controlado por el hiperparámetro `validation_fraction`, que es del 10% de forma predeterminada. El hiperparámetro `tol` determina la mejora máxima del rendimiento que aún se considera insignificante. El valor predeterminado es 0,0001.

La clase `GradientBoostingRegressor` también admite un hiperparámetro de submuestra, que especifica la fracción de instancias de entrenamiento que se utilizarán para entrenar cada árbol. Por ejemplo, si `submuestra = 0,25`, entonces cada árbol se entrena en el 25 % de las instancias de entrenamiento, seleccionadas al azar. Como probablemente ya puedas adivinar, esta técnica tiene un sesgo mayor.

para una variación menor. También acelera considerablemente el entrenamiento. Esto se llama aumento de gradiente estocástico.

## Aumento de gradiente basado en histograma

Scikit-Learn también proporciona otra implementación de GBRT, optimizada para grandes conjuntos de datos: aumento de gradiente basado en histogramas (HGB). Funciona agrupando las funciones de entrada y reemplazándolas con números enteros. El número de contenedores está controlado por el hiperparámetro `max_bins`, que por defecto es 255 y no se puede establecer en un valor superior a este. La agrupación puede reducir en gran medida la cantidad de umbrales posibles que el algoritmo de entrenamiento necesita evaluar. Además, trabajar con números enteros permite utilizar estructuras de datos más rápidas y con mayor eficiencia de memoria. Y la forma en que se construyen los contenedores elimina la necesidad de clasificar las características al entrenar cada árbol.

Como resultado, esta implementación tiene una complejidad computacional de  $O(b \cdot m)$  en lugar de  $O(n \cdot m \cdot \log(m))$ , donde  $b$  es el número de contenedores,  $m$  es el número de instancias de entrenamiento y  $n$  es el número de características. En la práctica, esto significa que HGB puede entrenar cientos de veces más rápido que GBRT normal en grandes conjuntos de datos. Sin embargo, la agrupación provoca una pérdida de precisión, que actúa como un regularizador: según el conjunto de datos, esto puede ayudar a reducir el sobreajuste o puede provocar un subajuste.

Scikit-Learn proporciona dos clases para HGB: `HistGradientBoostingRegressor` y `HistGradientBoostingClassifier`. Son similares a `GradientBoostingRegressor` y `GradientBoostingClassifier`, con algunas diferencias notables:

- La parada anticipada se activa automáticamente si el número de instancias es superior a 10.000. Puede activar o desactivar siempre la parada anticipada estableciendo el hiperparámetro `early_stopping` en Verdadero o Falso.
- No se admite el submuestreo.
- `n_estimators` pasa a llamarse `max_iter`.
- Los únicos hiperparámetros del árbol de decisión que se pueden modificar son `max_leaf_nodes`, `min_samples_leaf` y `max_depth`.

Las clases HGB también tienen dos características interesantes: admiten tanto características categóricas como valores faltantes. Esto simplifica bastante el preprocesamiento. Sin embargo, las características categóricas deben representarse como números enteros que van desde 0 hasta un número inferior a `max_bins`. Puede utilizar un `OrdinalEncoder` para esto. Por ejemplo, aquí se explica cómo construir y entrenar un proceso completo para el conjunto de datos de vivienda de California presentado en el [Capítulo 2](#):

```
desde sklearn.pipeline importar make_pipeline desde
sklearn.compose importar make_column_transformer desde
sklearn.ensemble importar HistGradientBoostingRegressor desde
sklearn.preprocessing importar OrdinalEncoder

hgb_reg = make_pipeline(
    make_column_transformer((OrdinalEncoder(), ["ocean_proximity"]),
                           ...)
```

```

resto="paso a través"),
HistGradientBoostingRegressor(categorical_features=[0], random_state=42)

) hgb_reg.fit(vivienda, etiquetas_vivienda)

```

¡Todo el proceso es tan corto como las importaciones! No se necesita un imputador, un escalador o un codificador one-hot, por lo que es realmente conveniente. Tenga en cuenta que `categorical_features` debe establecerse en los índices de columnas categóricas (o una matriz booleana). Sin ningún ajuste de hiperparámetros, este modelo produce un RMSE de aproximadamente 47.600, lo cual no está tan mal.

## CONSEJO

Varias otras implementaciones optimizadas de aumento de gradiente están disponibles en el ecosistema Python ML: en particular, [XGBoost](#), [impulso de gato](#), y [LightGBM](#). Estas bibliotecas existen desde hace varios años. Todos están especializados en aumentar el gradiente, sus API son muy similares a las de Scikit-Learn y proporcionan muchas funciones adicionales, incluida la aceleración de GPU; ¡Definitivamente deberías revisarlos! Además, la [biblioteca TensorFlow Random Forests](#) proporciona implementaciones optimizadas de una variedad de algoritmos de bosques aleatorios, incluidos bosques aleatorios simples, árboles adicionales, GBRT y muchos más.

## Apilado

El último método de conjunto que discutiremos en este capítulo se llama apilamiento (abreviatura de [generalización apilada](#)). Se basa en una idea simple: en lugar de utilizar funciones triviales (como la votación dura) para agregar las predicciones de todos los predictores en un conjunto, ¿por qué no entrenamos un modelo para realizar esta agregación? La Figura 7-11 muestra un conjunto de este tipo realizando una tarea de regresión en una nueva instancia. Cada uno de los tres predictores inferiores predice un valor diferente (3.1, 2.7 y 2.9), y luego el predictor final (llamado mezclador o metaaprendizaje) toma estas predicciones como entradas y realiza la predicción final (3.0).

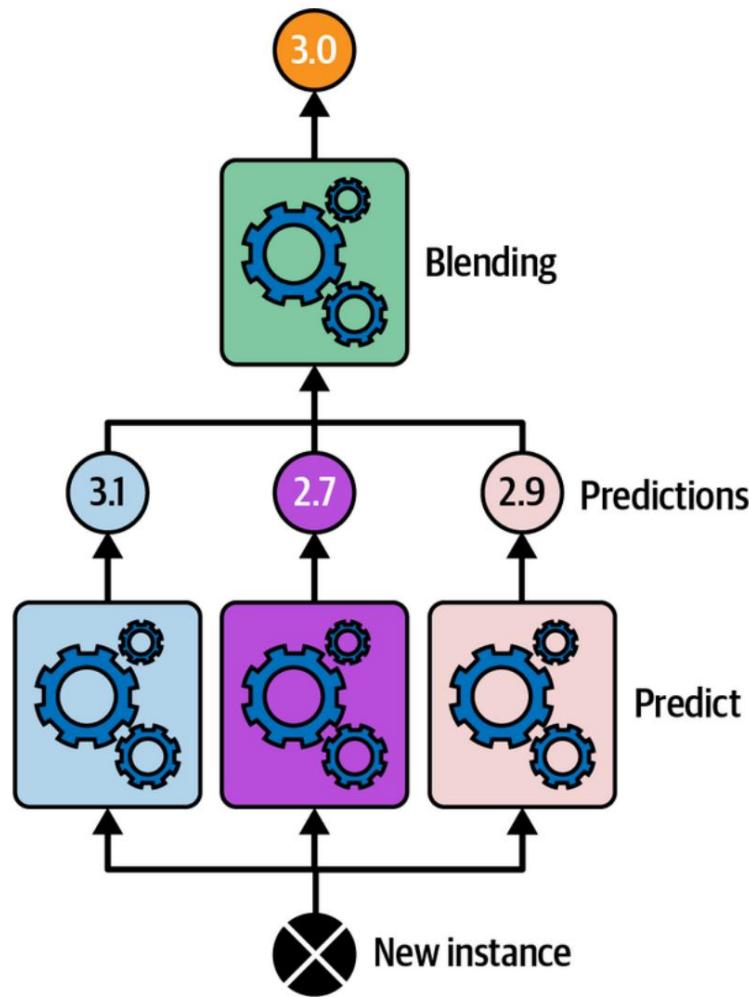


Figura 7-11. Agregar predicciones utilizando un predictor combinado

Para entrenar la licuadora, primero debe crear el conjunto de entrenamiento de mezcla. Puede usar `cross_val_predict()` en cada predictor del conjunto para obtener predicciones fuera de la muestra para cada instancia en el conjunto de entrenamiento original ([Figura 7-12](#)), y usarlas como características de entrada para entrenar el mezclador; y los objetivos pueden simplemente copiarse del conjunto de entrenamiento original. Tenga en cuenta que, independientemente de la cantidad de características en el conjunto de entrenamiento original (solo una en este ejemplo), el conjunto de entrenamiento combinado contendrá una característica de entrada por predictor (tres en este ejemplo). Una vez que se entrena el mezclador, los predictores base se vuelven a entrenar una última vez en el conjunto de entrenamiento original completo.

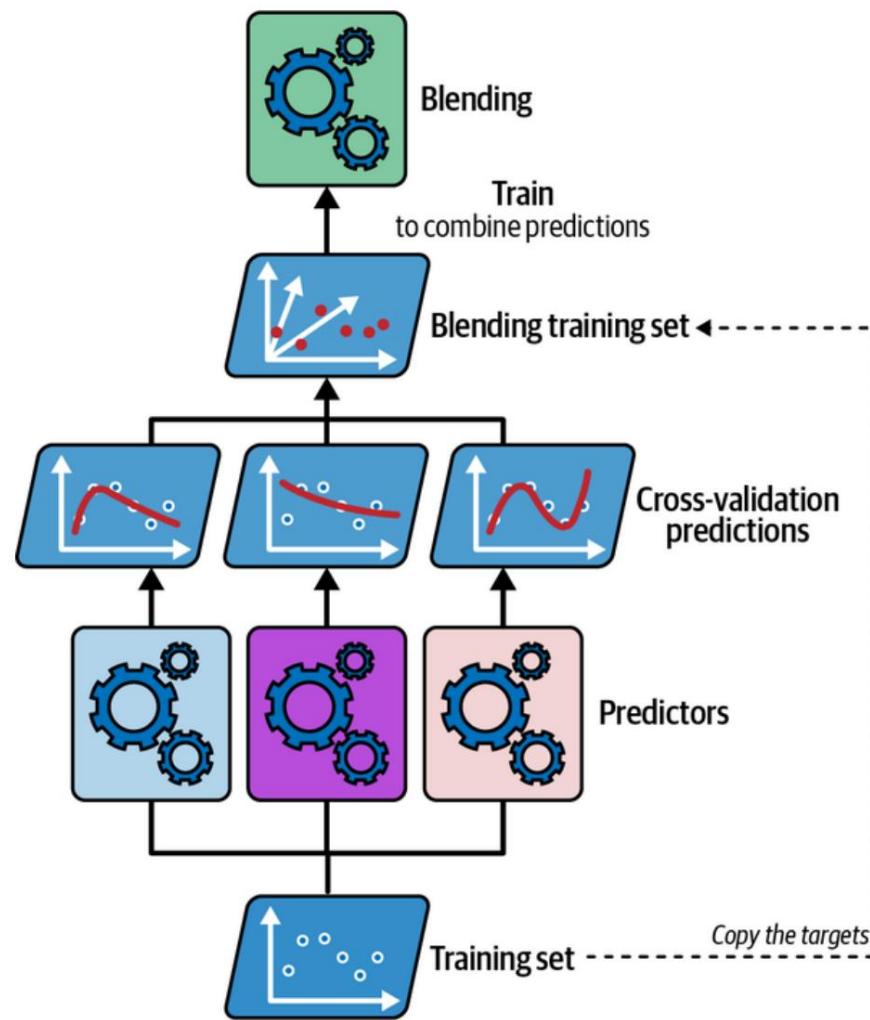


Figura 7-12. Entrenando la licuadora en un conjunto de apilamiento

En realidad, es posible entrenar varios mezcladores diferentes de esta manera (por ejemplo, uno usando regresión lineal, otro usando regresión forestal aleatoria) para obtener una capa completa de mezcladores y luego agregar otro mezclador encima para producir la predicción final, como se muestra en la Figura 7-13. Es posible que pueda obtener algunas caídas más de rendimiento al hacer esto, pero le costará tanto en tiempo de capacitación como en complejidad del sistema.

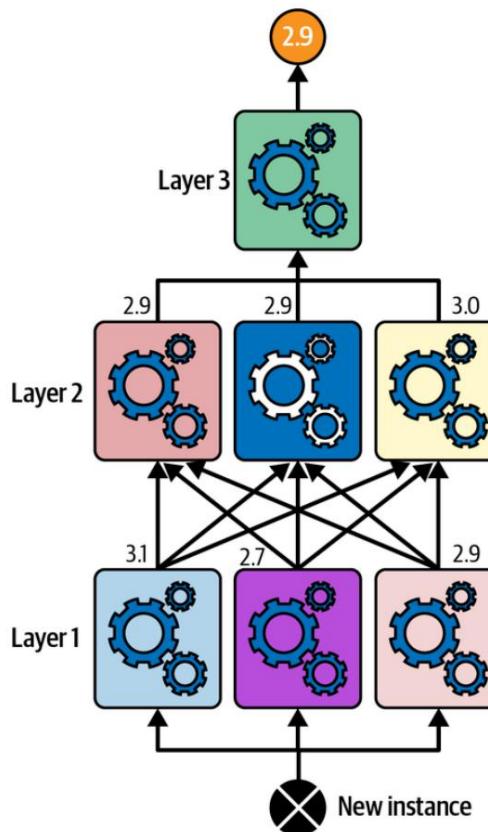


Figura 7-13. Predicciones en un conjunto de apilamiento multicapa

Scikit-Learn proporciona dos clases para apilar conjuntos: `StackingClassifier` y `StackingRegressor`. Por ejemplo, podemos reemplazar el `VotingClassifier` que usamos al comienzo de este capítulo en el conjunto de datos de las lunas con un `StackingClassifier`:

```
de sklearn.ensemble importar StackingClassifier

stacking_clf = ApilamientoClasificador(
    estimadores=[('lr', LogisticRegression(random_state=42)), ('rf',
        RandomForestClassifier(random_state=42)), ('svc', SVC(probabilidad=True,
        random_state=42))],
    final_estimator=RandomForestClassifier(random_state=43), cv=5 # número de pliegues
    de validación cruzada

) stacking_clf.fit(X_train, y_train)
```

Para cada predictor, el clasificador de apilamiento llamará a `predict_proba()` si está disponible; de lo contrario, recurrirá a `decision_function()` o, como último recurso, llamará a `predict()`. Si no proporciona un estimador final, `StackingClassifier` usará `LogisticRegression` y `StackingRegressor` usará `RidgeCV`.

Si evalúa este modelo de apilamiento en el conjunto de prueba, encontrará una precisión del 92,8%, que es un poco mejor que el clasificador de votación que utiliza votación suave, que obtuvo un 92%.

En conclusión, los métodos de conjunto son versátiles, potentes y bastante sencillos de utilizar. Los bosques aleatorios, AdaBoost y GBRT se encuentran entre los primeros modelos que debe probar para la mayoría de las tareas de aprendizaje automático y brillan particularmente con datos tabulares heterogéneos. Además, como requieren muy poco preprocessamiento, son excelentes para poner en marcha un prototipo rápidamente. Por último, los métodos de conjunto, como los clasificadores de votación y los clasificadores de apilamiento, pueden ayudar a llevar el rendimiento de su sistema al límite.

## Ejercicios

1. Si ha entrenado cinco modelos diferentes con exactamente los mismos datos de entrenamiento y todos logran una precisión del 95%, ¿existe alguna posibilidad de que pueda combinar estos modelos para obtener mejores resultados? ¿Si es así, cómo? Si no, ¿por qué?
2. ¿Cuál es la diferencia entre clasificadores de voto duro y blando?
3. ¿Es posible acelerar el entrenamiento de un conjunto de embolsado distribuyéndolo entre varios servidores? ¿Qué pasa con pegar conjuntos, impulsar conjuntos, bosques aleatorios o apilar conjuntos?
4. ¿Cuál es el beneficio de la evaluación inmediata?
5. ¿Qué hace que los conjuntos de árboles adicionales sean más aleatorios que los bosques aleatorios normales? Cómo Puede ayudar esta aleatoriedad adicional? ¿Los clasificadores de árboles adicionales son más lentos o más rápidos que los bosques aleatorios normales?
6. Si su conjunto AdaBoost no se ajusta a los datos de entrenamiento, ¿qué hiperparámetros deberían modificarse, y ¿cómo?
7. Si su conjunto de aumento de gradiente se adapta demasiado al conjunto de entrenamiento, ¿debería aumentar o disminuir la tasa de aprendizaje?
8. Cargue el conjunto de datos MNIST (presentado en [el Capítulo 3](#)) y divídalo en un conjunto de entrenamiento, un conjunto de validación y un conjunto de prueba (por ejemplo, use 50 000 instancias para entrenamiento, 10 000 para validación y 10 000 para pruebas). Luego entrene varios clasificadores, como un clasificador de bosque aleatorio, un clasificador de árboles adicionales y un clasificador SVM. A continuación, intente combinarlos en un conjunto que supere a cada clasificador individual en el conjunto de validación, mediante votación suave o dura. Una vez que haya encontrado uno, pruébelo en el equipo de prueba. ¿Cuánto mejor funciona en comparación con los clasificadores individuales?
9. Ejecute los clasificadores individuales del ejercicio anterior para hacer predicciones sobre el conjunto de validación y cree un nuevo conjunto de entrenamiento con las predicciones resultantes: cada instancia de entrenamiento es un vector que contiene el conjunto de predicciones de todos sus clasificadores para una imagen, y el objetivo es la clase de la imagen. Entrene a un clasificador en este nuevo conjunto de entrenamiento. Felicitaciones, acaba de entrenar una licuadora y, junto con los clasificadores, ¡forma un conjunto apilable! Ahora evalúe el conjunto en el conjunto de prueba. Para cada imagen en el conjunto de prueba, haga predicciones con todos sus clasificadores y luego envíe las predicciones al mezclador para obtener las predicciones del conjunto. ¿Cómo se compara con el clasificador de votación que entrenó anteriormente? Ahora intente nuevamente usando un `StackingClassifier`.  
¿Obtienes un mejor rendimiento? Si es así, ¿por qué?

Las soluciones a estos ejercicios están disponibles al final del cuaderno de este capítulo, en <https://homl.info/colab3>.

---

**1** Imagíñese tomar una carta al azar de una baraja de cartas, escribirla y luego volver a colocarla en la baraja antes de elegir la siguiente carta: la misma carta se podría probar varias veces.

**2** Leo Breiman, "Predictores de embolsado", *Machine Learning* 24, no. 2 (1996): 123–140.

**3** En estadística, el remuestreo con reemplazo se denomina bootstrapping.

**4** Leo Breiman, "Pegando pequeños votos para su clasificación en grandes bases de datos y en línea", *Machine Aprendizaje* 36, núm. 1–2 (1999): 85–103.

**5** El sesgo y la varianza se introdujeron en [el Capítulo 4](#).

Alternativamente, **6** `max_samples` se pueden establecer en un valor flotante entre 0,0 y 1,0, en cuyo caso el número máximo de instancias muestreadas es igual al tamaño del conjunto de entrenamiento multiplicado por `max_samples`.

**7** A medida que `m` crece, esta relación se acerca a  $1 - \exp(-1) \approx 63\%$ .

**8** Gilles Louppe y Pierre Geurts, "Conjuntos en parches aleatorios", *Lecture Notes in Computer Science* 7523 (2012): 346–361.

**9** Tin Kam Ho, "El método subespacial aleatorio para la construcción de bosques de decisión", *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20, no. 8 (1998): 832–844.

**10** Tin Kam Ho, "Random Decision Forests", *Actas de la Tercera Conferencia Internacional sobre Análisis y reconocimiento de documentos* 1 (1995): 278.

**11** La clase `BaggingClassifier` sigue siendo útil si desea una bolsa de algo más que decisión árboles.

**12** Pierre Geurts et al., "Árboles extremadamente aleatorios", *Machine Learning* 63, no. 1 (2006): 3–42.

**13** Yoav Freund y Robert E. Schapire, "Una generalización teórica del aprendizaje en línea y una aplicación al impulso", *Journal of Computer and System Sciences* 55, no. 1 (1997): 119–139.

**14** Esto es sólo para fines ilustrativos. Las SVM generalmente no son buenos predictores básicos para AdaBoost; ellos son lentos y tienden a ser inestables con él.

**15** El algoritmo AdaBoost original no utiliza un hiperparámetro de tasa de aprendizaje.

**16** Para obtener más detalles, consulte Ji Zhu et al., "Multi-Class AdaBoost", *Statistics and Its Interface* 2, no. 3 (2009): 349–360.

**17** El aumento de gradiente se introdujo por primera vez en [el artículo de Leo Breiman de 1997](#). "Arcing the Edge" y se desarrolló más en el [artículo de 1999](#). "Aproximación de funciones codiciosas: una máquina de aumento de gradiente" por Jerome H. Friedman.

**18** David H. Wolpert, "Generalización apilada", *Neural Networks* 5, no. 2 (1992): 241–259.

# Capítulo 8. Reducción de dimensionalidad

---

Muchos problemas de aprendizaje automático implican miles o incluso millones de funciones para cada instancia de formación. Todas estas características no sólo hacen que el entrenamiento sea extremadamente lento, sino que también pueden hacer que sea mucho más difícil encontrar una buena solución, como verás. Este problema a menudo se conoce como la maldición de la dimensionalidad.

Afortunadamente, en los problemas del mundo real, a menudo es posible reducir considerablemente el número de funciones, convirtiendo un problema intratable en uno tratable.

Por ejemplo, considere las imágenes MNIST (introducidas en [el Capítulo 3](#)): los píxeles en los bordes de la imagen casi siempre son blancos, por lo que podría eliminar completamente estos píxeles del conjunto de entrenamiento sin perder mucha información. Como vimos en el capítulo anterior, ([Figura 7-6](#)) confirma que estos píxeles no tienen ninguna importancia para la tarea de clasificación. Además, dos píxeles vecinos suelen estar altamente correlacionados: si los fusiona en un solo píxel (por ejemplo, tomando la media de las intensidades de los dos píxeles), no perderá mucha información.

## ADVERTENCIA

Reducir la dimensionalidad causa cierta pérdida de información, al igual que comprimir una imagen a JPEG puede degradar su calidad, por lo que aunque acelerará el entrenamiento, puede hacer que su sistema funcione un poco peor. También hace que sus tuberías sean un poco más complejas y, por lo tanto, más difíciles de mantener. Por lo tanto, le recomiendo que primero intente entrenar su sistema con los datos originales antes de considerar utilizar la reducción de dimensionalidad. En algunos casos, reducir la dimensionalidad de los datos de entrenamiento puede filtrar algo de ruido y detalles innecesarios y, por tanto, dar como resultado un mayor rendimiento, pero en general no es así; simplemente acelerará el entrenamiento.

Además de acelerar el entrenamiento, la reducción de dimensionalidad también es extremadamente útil para la visualización de datos. Reducir el número de dimensiones a dos (o tres) permite trazar una vista condensada de un conjunto de entrenamiento de alta dimensión en un gráfico y, a menudo, obtener información importante al detectar patrones visualmente, como grupos. Además, la visualización de datos es esencial.

comunicar sus conclusiones a personas que no son científicos de datos, en particular, a los tomadores de decisiones que utilizarán sus resultados.

En este capítulo primero discutiremos la maldición de la dimensionalidad y tendremos una idea de lo que sucede en el espacio de alta dimensión. Luego, consideraremos los dos enfoques principales para la reducción de dimensionalidad (proyección y aprendizaje múltiple) y repasaremos tres de las técnicas de reducción de dimensionalidad más populares: PCA, proyección aleatoria e incrustación localmente lineal (LLE).

## La maldición de la dimensionalidad

Estamos tan acostumbrados a vivir en tres dimensiones <sup>1</sup> que nuestra intuición nos falla cuando intentamos imaginar un espacio de altas dimensiones. Incluso un hipercubo 4D básico es increíblemente difícil de imaginar en nuestra mente (ver [Figura 8-1](#)), por no hablar de un elipsoide de 200 dimensiones doblado en un espacio de 1.000 dimensiones.

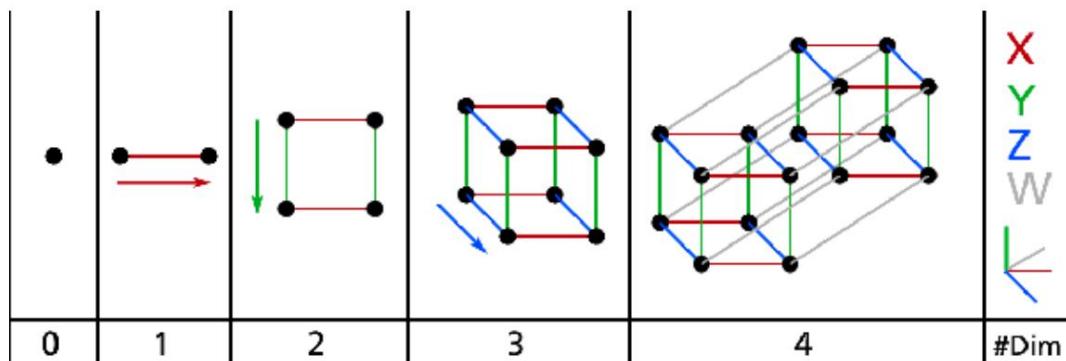


Figura 8-1. Punto, segmento, cuadrado, cubo y teseracto (hipercubos de 0D a 4D)<sup>2</sup>

Resulta que muchas cosas se comportan de manera muy diferente en el espacio de alta dimensión. Por ejemplo, si eliges un punto aleatorio en un cuadrado unitario (un cuadrado de  $1 \times 1$ ), tendrá sólo un 0,4% de probabilidad de estar ubicado a menos de 0,001 de un borde (en otras palabras, es muy improbable que un punto se encuentre a menos de 0,001 de un borde). El punto aleatorio será “extremo” en cualquier dimensión). Pero en un hipercubo unitario de 10.000 dimensiones, esta probabilidad es superior al 99,999999%. La mayoría de los puntos en un hipercubo de alta dimensión están muy cerca del borde.<sup>3</sup>

Aquí hay una diferencia más problemática: si eliges dos puntos al azar en un cuadrado unitario, la distancia entre estos dos puntos será, en promedio, aproximadamente 0,52. Si eliges dos puntos aleatorios en un cubo unitario 3D, la distancia promedio será aproximadamente 0,66. Pero ¿qué pasa con dos puntos elegidos al azar en un hipercubo unitario de 1.000.000 de dimensiones? La distancia promedio, lo creas o

---

no, ¡será aproximadamente 408,25 (aproximadamente  $\sqrt{1.000.000}/6$ )! Esto es contradictorio: ¿cómo ¿Pueden dos puntos estar tan separados cuando ambos se encuentran dentro del mismo hipercubo unitario? Bueno, en las dimensiones altas hay mucho espacio. Como resultado, los conjuntos de datos de alta dimensión corren el riesgo de ser muy escasos: es probable que la mayoría de las instancias de entrenamiento estén muy alejadas unas de otras. Esto también significa que una nueva instancia probablemente estará lejos de cualquier instancia de entrenamiento, lo que hace que las predicciones sean mucho menos confiables que en dimensiones más bajas, ya que se basarán en extrapolaciones mucho mayores. En resumen, cuantas más dimensiones tenga el conjunto de entrenamiento, mayor será el riesgo de sobreajustarlo.

En teoría, una solución a la maldición de la dimensionalidad podría ser aumentar el tamaño del conjunto de entrenamiento para alcanzar una densidad suficiente de instancias de entrenamiento. Desafortunadamente, en la práctica, la cantidad de instancias de entrenamiento necesarias para alcanzar una densidad determinada crece exponencialmente con la cantidad de dimensiones. Con solo 100 características (significativamente menos que en el problema MNIST), todas con un rango de 0 a 1, se necesitarían más instancias de entrenamiento que átomos en el universo observable para que las instancias de entrenamiento estén dentro de un promedio de 0,1 entre sí, suponiendo que fueran distribuirse uniformemente en todas las dimensiones.

## Principales enfoques para la reducción de dimensionalidad

Antes de sumergirnos en algoritmos de reducción de dimensionalidad específicos, echemos un vistazo a los dos enfoques principales para reducir la dimensionalidad: proyección y aprendizaje múltiple.

### Proyección

En la mayoría de los problemas del mundo real, las instancias de capacitación no se distribuyen uniformemente en todas las dimensiones. Muchas características son casi constantes, mientras que otras están altamente correlacionadas (como se analizó anteriormente para MNIST). Como resultado, todas las instancias de entrenamiento se encuentran dentro (o cerca de) un subespacio de dimensiones mucho más bajas del espacio de alta dimensión. Esto suena muy abstracto, así que veamos un ejemplo. En [la Figura 8-2](#) puede ver un conjunto de datos 3D representado por pequeñas esferas.

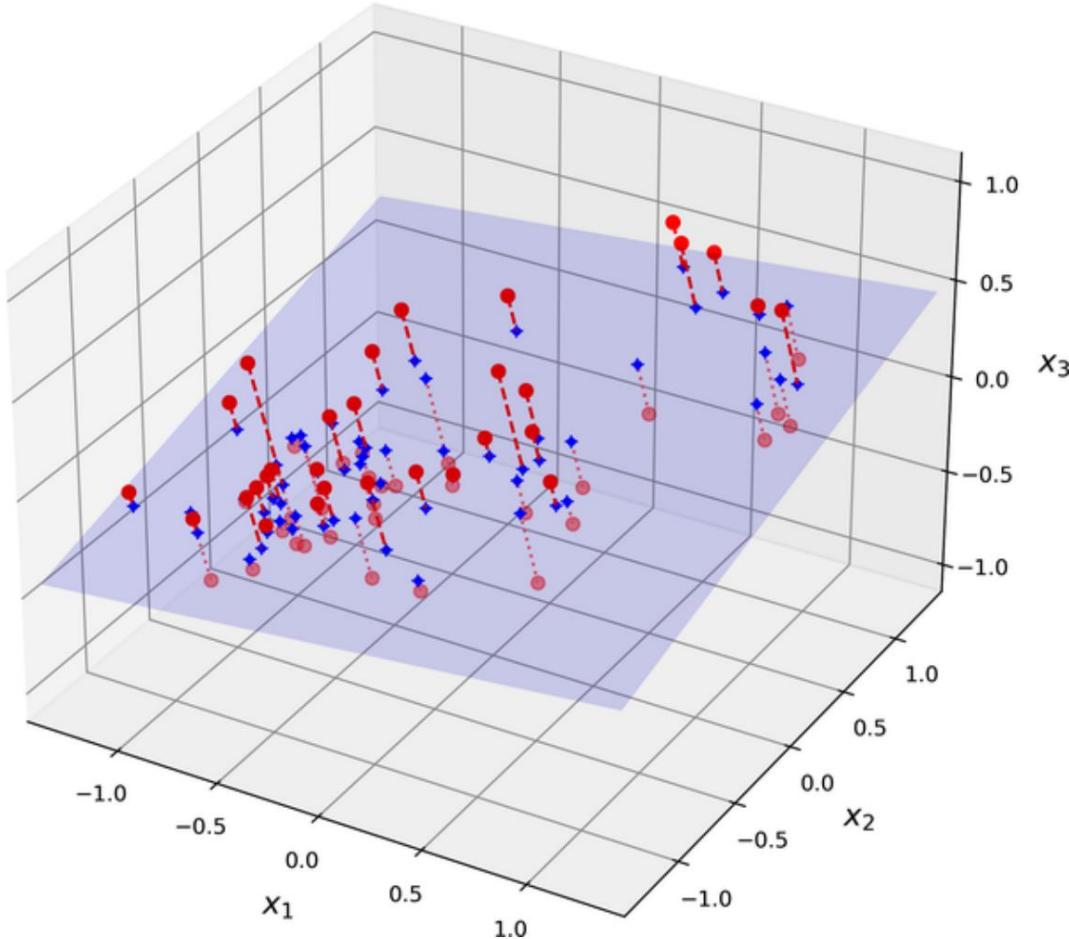


Figura 8-2. Un conjunto de datos 3D que se encuentra cerca de un subespacio 2D

Observe que todas las instancias de entrenamiento se encuentran cerca de un plano: este es un subespacio de dimensiones inferiores (2D) del espacio de dimensiones superiores (3D). Si proyectamos cada instancia de entrenamiento perpendicularmente a este subespacio (como lo representan las líneas discontinuas cortas que conectan las instancias con el plano), obtenemos el nuevo conjunto de datos 2D que se muestra en la [Figura 8-3](#). ¡Ta-dá! Acabamos de reducir la dimensionalidad del conjunto de datos de 3D a 2D. Tenga en cuenta que los ejes corresponden a las ~~z~~ nuevas características  $z$  y  $z$ : son las coordenadas de las proyecciones en el plano.

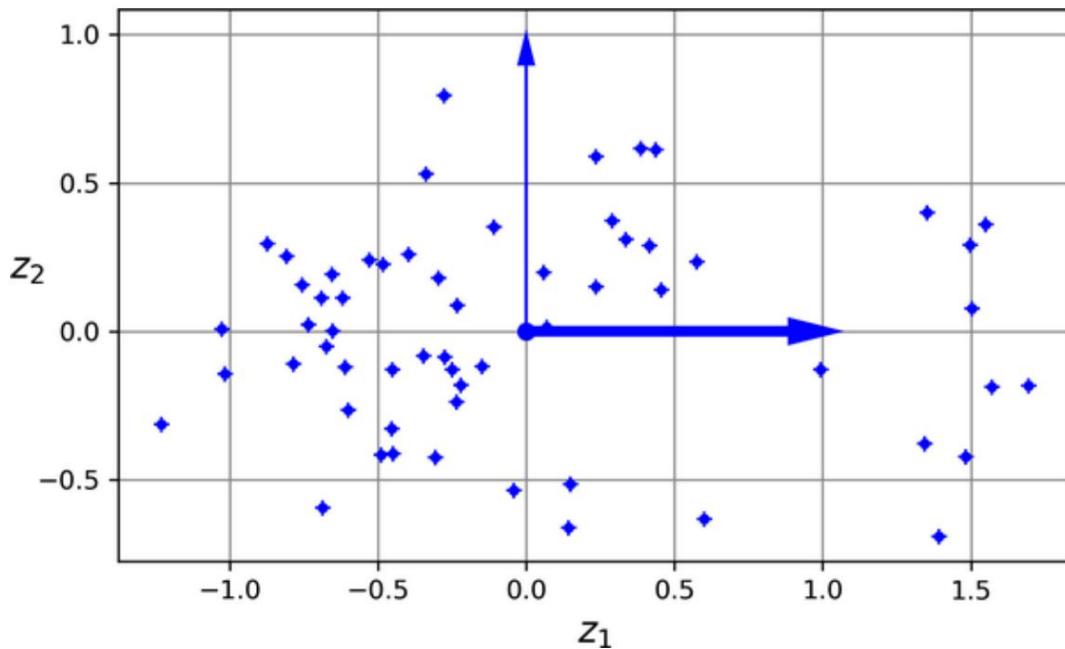


Figura 8-3. El nuevo conjunto de datos 2D después de la proyección.

### Aprendizaje múltiple Sin

embargo, la proyección no siempre es el mejor enfoque para la reducción de dimensionalidad. En muchos casos, el subespacio puede torcerse y girar, como en el famoso conjunto de datos de juguetes suizos representado en la Figura 8-4.

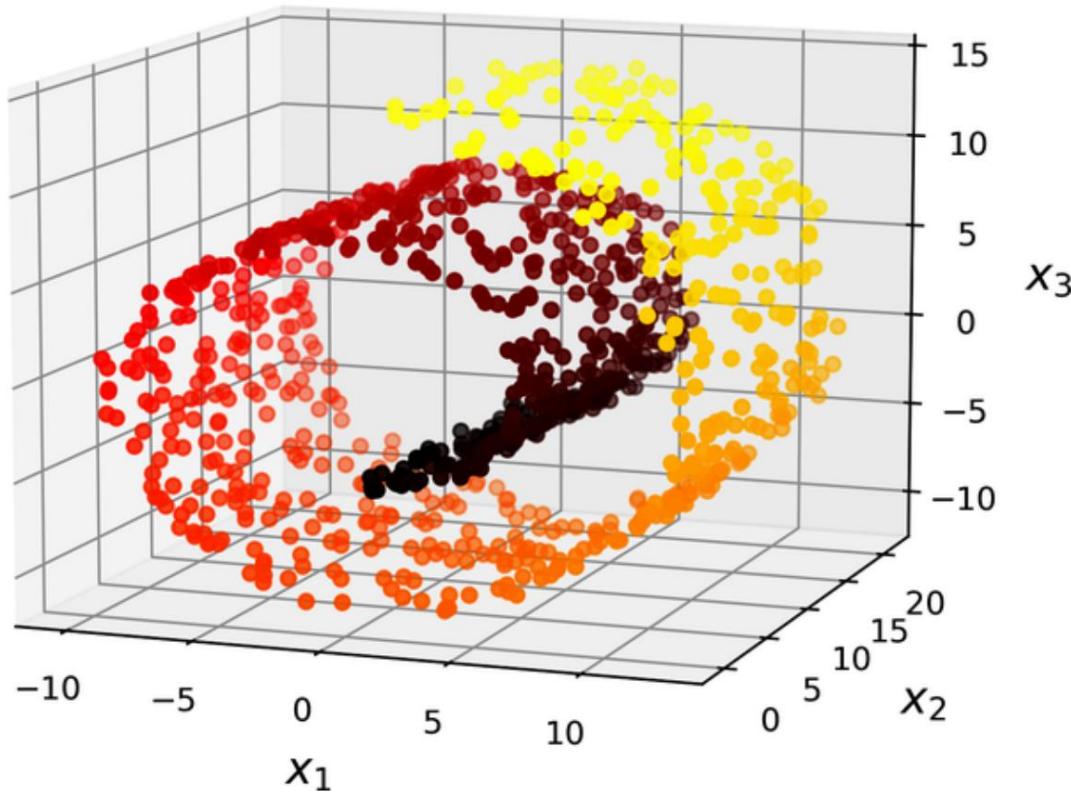


Figura 8-4. Conjunto de datos de rollo suizo

Simplemente proyectando sobre un plano (por ejemplo, dejando caer  $x_3$ ) se aplastarían diferentes capas del rollo suizo, como se muestra en el lado izquierdo de la [Figura 8-5](#). Lo que probablemente desee es desenrollar el rollo suizo para obtener el conjunto de datos 2D en el lado derecho de [la Figura 8-5](#).

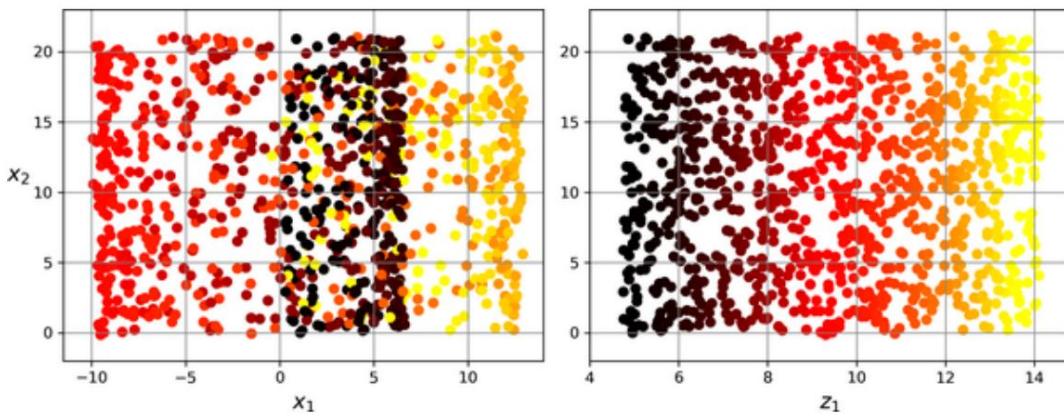


Figura 8-5. Aplastar proyectando sobre un plano (izquierda) versus desenrollar el rollo suizo (derecha)

El rollo suizo es un ejemplo de variedad 2D. En pocas palabras, una variedad 2D es una forma 2D que se puede doblar y torcer en un espacio de dimensiones superiores. De manera más general, una variedad  $d$ -dimensional es parte de un espacio  $n$ -dimensional (donde  $d < n$ ) que localmente se asemeja a un hiperplano  $d$ -dimensional. En el caso

del rollo suizo,  $d = 2$  y  $n = 3$ : localmente se parece a un plano 2D, pero está enrollado en la tercera dimensión.

Muchos algoritmos de reducción de dimensionalidad funcionan modelando la variedad en la que se encuentran las instancias de entrenamiento; esto se llama aprendizaje múltiple. Se basa en el supuesto múltiple, también llamado hipótesis múltiple, que sostiene que la mayoría de los conjuntos de datos de alta dimensión del mundo real se encuentran cerca de una variedad de dimensiones mucho más bajas. Esta suposición se observa muy a menudo empíricamente.

Una vez más, piense en el conjunto de datos MNIST: todas las imágenes de dígitos escritos a mano tienen algunas similitudes. Están formados por líneas conectadas, los bordes son blancos y están más o menos centrados. Si generaras imágenes aleatoriamente, sólo una fracción ridículamente pequeña de ellas parecería dígitos escritos a mano. En otras palabras, los grados de libertad disponibles para usted si intenta crear una imagen digital son dramáticamente más bajos que los grados de libertad que tiene si se le permite generar cualquier imagen que desee. Estas restricciones tienden a comprimir el conjunto de datos en una variedad de dimensiones inferiores.

El supuesto múltiple suele ir acompañado de otro supuesto implícito: que la tarea en cuestión (por ejemplo, clasificación o regresión) será más simple si se expresa en el espacio de dimensiones inferiores de la variedad. Por ejemplo, en la fila superior de [la Figura 8-6](#), el rollo suizo se divide en dos clases: en el espacio 3D (a la izquierda), el límite de decisión sería bastante complejo, pero en el espacio múltiple desenrollado 2D (a la derecha) el límite de decisión es una línea recta.

Sin embargo, esta suposición implícita no siempre se cumple. Por ejemplo, en la fila inferior de [la Figura 8-6](#), el límite de decisión está ubicado en  $x = 5$ . Este límite de decisión parece muy simple en el espacio 3D original (un plano vertical), pero parece más complejo en la variedad desenrollada (una colección de cuatro segmentos de línea independientes).

En resumen, reducir la dimensionalidad de su conjunto de entrenamiento antes de entrenar un modelo generalmente acelerará el entrenamiento, pero no siempre conducirá a una solución mejor o más simple; Todo depende del conjunto de datos.

Esperemos que ahora tenga una buena idea de cuál es la maldición de la dimensionalidad y cómo los algoritmos de reducción de dimensionalidad pueden combatirla, especialmente cuando se cumple la suposición múltiple. El resto de este capítulo analizará algunos de los algoritmos más populares para la reducción de dimensionalidad.

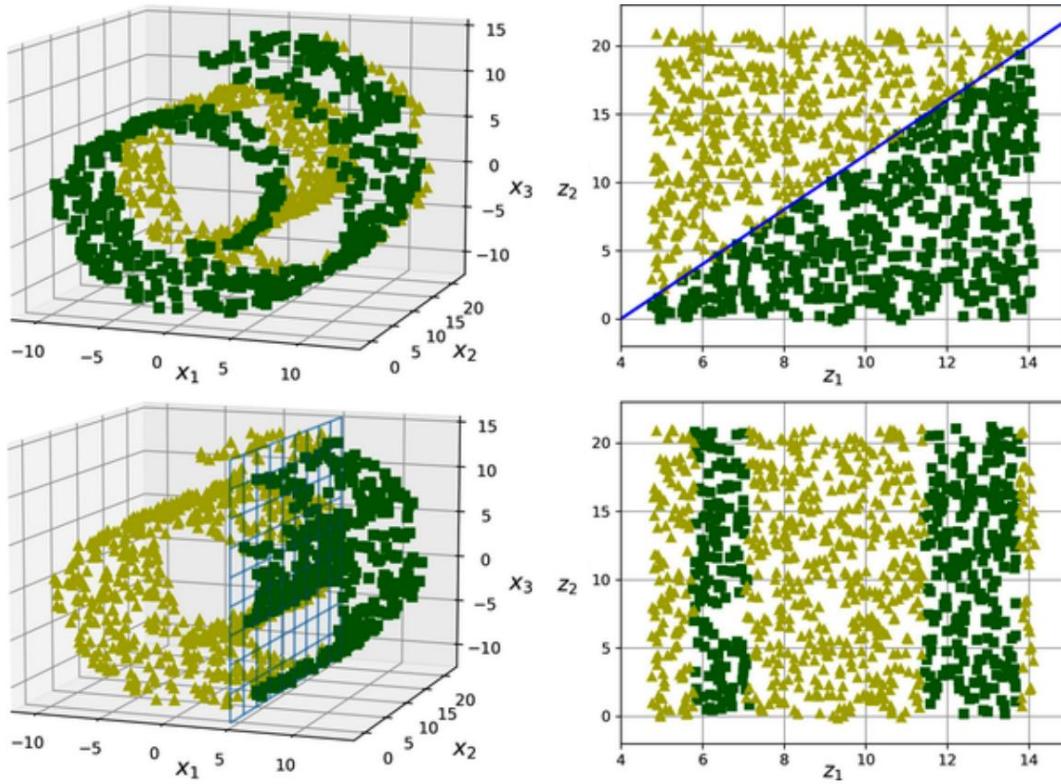


Figura 8-6. Es posible que el límite de decisión no siempre sea más simple con dimensiones más bajas

## PCA

El análisis de componentes principales (PCA) es, con diferencia, el algoritmo de reducción de dimensionalidad más popular. Primero identifica el hiperplano más cercano a los datos y luego proyecta los datos en él, como en la [Figura 8-2](#).

## Preservar la variación

Antes de poder proyectar el conjunto de entrenamiento en un hiperplano de dimensiones inferiores, primero debe elegir el hiperplano correcto. Por ejemplo, un conjunto de datos 2D simple se representa a la izquierda en la [Figura 8-7](#), junto con tres ejes diferentes (es decir, hiperplanos 1D). A la derecha está el resultado de la proyección del conjunto de datos sobre cada uno de estos ejes. Como puede ver, la proyección sobre la línea continua conserva la varianza máxima (arriba), mientras que la proyección sobre la línea de puntos conserva muy poca varianza (abajo) y la proyección sobre la línea discontinua conserva una cantidad intermedia de varianza (centro).

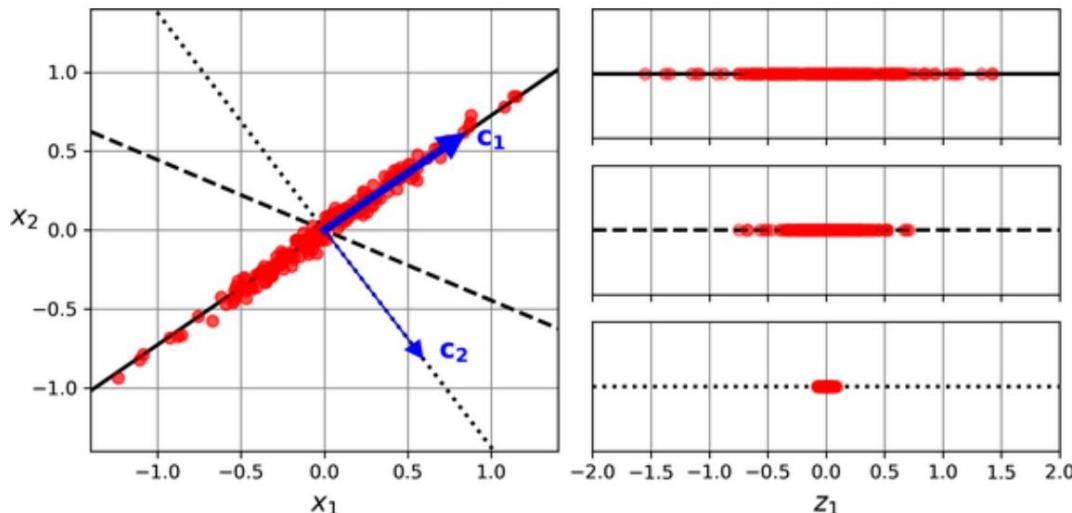


Figura 8-7. Seleccionar el subespacio sobre el que proyectar

Parece razonable seleccionar el eje que conserve la máxima cantidad de varianza, ya que probablemente perderá menos información que las otras proyecciones.

Otra forma de justificar esta elección es que es el eje que minimiza la distancia media cuadrática entre el conjunto de datos original y su proyección sobre ese eje. Ésta es la idea bastante simple detrás de **PCA**.<sup>4</sup>

### Componentes principales PCA

identifica el eje que representa la mayor cantidad de variación en el conjunto de entrenamiento. En la Figura 8-7, es la línea continua. También encuentra un segundo eje, ortogonal al primero, que representa la mayor cantidad de la varianza restante.

En este ejemplo 2D no hay elección: es la línea de puntos.

Si se tratara de un conjunto de datos de dimensiones superiores, PCA también encontraría un tercer eje, ortogonal a los dos ejes anteriores, y un cuarto, un quinto, etc., tantos ejes como el número de dimensiones del conjunto de datos.

El eje  $i^{\text{th}}$  se denomina componente principal  $i$  (PC) de los datos. En la Figura 8-7, el primer PC es el eje sobre el que se encuentra el vector  $c_1$  y el segundo PC es el eje sobre el que se encuentra el vector  $c_2$ . En la Figura 8-2, las dos primeras PC están en el plano de proyección y la tercera PC es el eje ortogonal a ese plano. Después de la proyección, en la Figura 8-3, la primera PC corresponde al eje  $z$  y la segunda PC corresponde al eje  $z$ .

## NOTA

Para cada componente principal, PCA encuentra un vector unitario centrado en cero que apunta en la dirección de la PC. Dado que dos vectores unitarios opuestos se encuentran en el mismo eje, la dirección de los vectores unitarios devueltos por PCA no es estable: si perturba ligeramente el conjunto de entrenamiento y ejecuta PCA nuevamente, los vectores unitarios pueden apuntar en la dirección opuesta a los vectores originales. Sin embargo, por lo general seguirán estando en los mismos ejes. En algunos casos, un par de vectores unitarios pueden incluso rotar o intercambiarse (si las varianzas a lo largo de estos dos ejes son muy cercanas), pero el plano que definen generalmente seguirá siendo el mismo.

Entonces, ¿cómo se pueden encontrar los componentes principales de un conjunto de entrenamiento? Afortunadamente, existe una técnica estándar de factorización matricial llamada descomposición de valores singulares (SVD) que puede descomponer la matriz  $X$  del conjunto de entrenamiento en la multiplicación  $U \Sigma V^T$ , donde  $V$  contiene la unidad matricial de tres matrices  $U \Sigma V$  vectores que definen todos los componentes principales que está buscando, como se muestra en [Ecuación 8-1](#).

Ecuación 8-1. Matriz de componentes principales

$$V = \begin{matrix} & c_1 & c_2 & \dots & c_n \end{matrix}$$

El siguiente código Python utiliza la función `svd()` de NumPy para obtener todos los componentes principales del conjunto de entrenamiento 3D representado en la [Figura 8-2](#), luego extrae los dos vectores unitarios que definen las dos primeras PC:

```
importar numpy como np

X = [...] # crear un pequeño conjunto de datos 3D
X_centered = X - X.mean(axis=0)
U, s, Vt = np.linalg.svd(X_centered)
c1 = Vt[0]
c2 = Vt[1]
```

**ADVERTENCIA**

PCA supone que el conjunto de datos se centra en el origen. Como verá, las clases PCA de Scikit-Learn se encargan de centrar los datos por usted. Si implementa PCA usted mismo (como en el ejemplo anterior) o si utiliza otras bibliotecas, no olvide centrar los datos primero.

Proyectar hasta d dimensiones Una vez que haya

identificado todos los componentes principales, puede reducir la dimensionalidad del conjunto de datos hasta d dimensiones proyectándolo en el hiperplano definido por los primeros d componentes principales. Seleccionar este hiperplano garantiza que la proyección conserve la mayor variación posible. Por ejemplo, en [la Figura 8-2](#), el conjunto de datos 3D se proyecta hasta el plano 2D definido por los dos primeros componentes principales, preservando una gran parte de la varianza del conjunto de datos. Como resultado, la proyección 2D se parece mucho al conjunto de datos 3D original.

Para proyectar el conjunto de entrenamiento en el hiperplano y obtener un conjunto de datos reducido X de dimensionalidad d, calcule la multiplicación matricial del conjunto de entrenamiento `proj` matriz X por la matriz W definida como la matriz que contiene las primeras d columnas de V, como se muestra en [Ecuación 8-2](#).

Ecuación 8-2. Proyectar el conjunto de entrenamiento en dimensiones d

$$X_d\text{-proyecto} = XW_d$$

El siguiente código Python proyecta el conjunto de entrenamiento en el plano definido por los dos primeros componentes principales:

```
W2 = Vt[:2].T X2D
= X_centerado en W2
```

¡Ahí tienes! Ahora sabe cómo reducir la dimensionalidad de cualquier conjunto de datos proyectándolo a cualquier número de dimensiones, preservando al mismo tiempo la mayor variación posible.

## Usando Scikit-Learn

La clase PCA de Scikit-Learn utiliza SVD para implementar PCA, tal como lo hicimos anteriormente en este capítulo. El siguiente código aplica PCA para reducir la dimensionalidad del conjunto de datos a dos dimensiones (tenga en cuenta que automáticamente se encarga de centrar los datos):

```
desde sklearn.decomposition importar PCA
```

```
pca = PCA(n_componentes=2)
X2D = pca.fit_transform(X)
```

Después de ajustar el transformador PCA al conjunto de datos, su atributo `componentes_` contiene la transpuesta de  $W^T$ : contiene una fila para cada uno de los primeros  $d$  componentes principales.

## Relación de varianza explicada

Otra información útil es el índice de varianza explicada de cada componente principal, disponible a través de la variable `explained_variance_ratio_`. La relación indica la proporción de la varianza del conjunto de datos que se encuentra a lo largo de cada componente principal. Por ejemplo, veamos las razones de varianza explicadas de los dos primeros componentes del conjunto de datos 3D representado en [la Figura 8-2](#):

```
>>> pca.explained_variance_ratio_
array([0.7578477, 0.15186921])
```

Este resultado nos dice que aproximadamente el 76 % de la variación del conjunto de datos se encuentra a lo largo de la primera PC y aproximadamente el 15 % se encuentra a lo largo de la segunda PC. Esto deja alrededor del 9% para la tercera PC, por lo que es razonable suponer que la tercera PC probablemente contenga poca información.

Elegir el número correcto de dimensiones En lugar de elegir arbitrariamente el número de dimensiones a reducir, es más sencillo elegir el número de dimensiones que sumen una porción suficientemente grande de la varianza, digamos, 95% (una excepción a esta regla). , por supuesto, es si está reduciendo la dimensionalidad para la visualización de datos, en cuyo caso querrá reducir la dimensionalidad a 2 o 3).

El siguiente código carga y divide el conjunto de datos MNIST (introducido en [Capítulo 3](#)) y realiza PCA sin reducir la dimensionalidad, luego calcula el número mínimo de dimensiones necesarias para preservar el 95% de la varianza del conjunto de entrenamiento:

```
desde sklearn.datasets importar fetch_openml

mnist = fetch_openml('mnist_784', as_frame=False)
X_train, y_train = mnist.data[:60_000], mnist.target[:60_000]
X_test, y_test = mnist.data[60_000:], mnist.target[60_000:]

pca = PCA()
pca.fit(X_train) cumsum
= np.cumsum(pca.explained_variance_ratio_) d = np.argmax(cumsum
>= 0,95) + 1 # d es igual a 154
```

Luego podría configurar `n_components=d` y ejecutar PCA nuevamente, pero hay una opción mejor. En lugar de especificar el número de componentes principales que desea conservar, puede configurar `n_components` para que sea un valor flotante entre 0,0 y 1,0, lo que indica la proporción de varianza que desea conservar:

```
pca = PCA(n_componentes=0,95)
X_reducido = pca.fit_transform(X_train)
```

La cantidad real de componentes se determina durante el entrenamiento y se almacena en el atributo `n_components_`:

```
>>> pca.n_components_
```

Otra opción más es trazar la varianza explicada como una función del número de dimensiones (simplemente trazar la suma acumulada; ver [la Figura 8-8](#)). Generalmente habrá un codo en la curva, donde la varianza explicada deja de crecer rápidamente. En este caso, puede ver que reducir la dimensionalidad a aproximadamente 100 dimensiones no perdería demasiada varianza explicada.

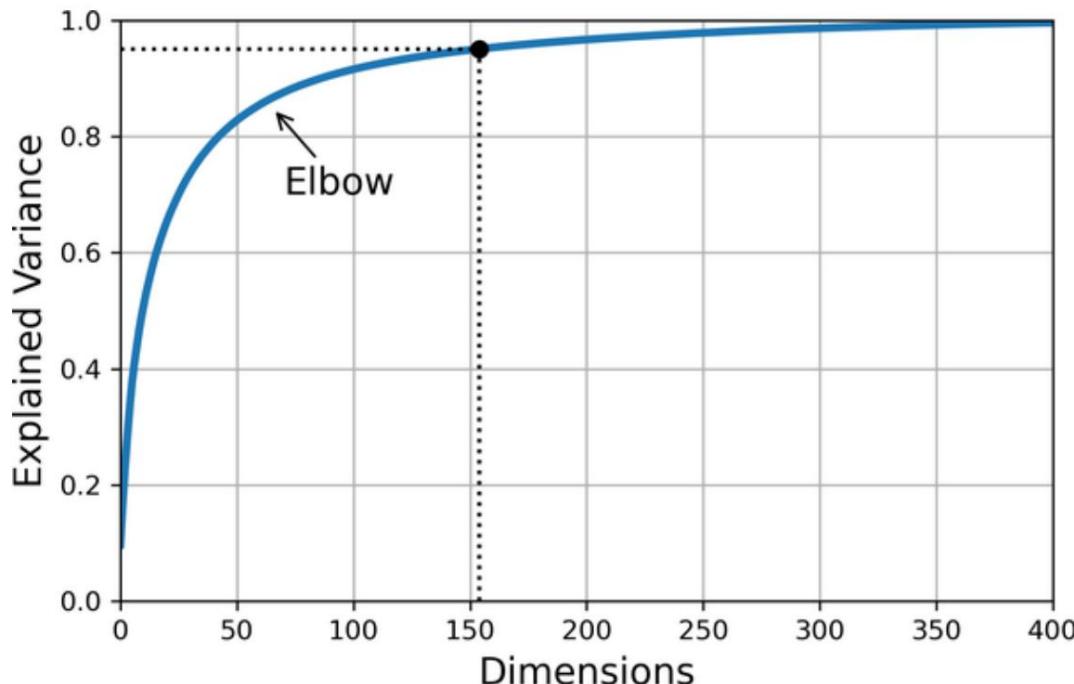


Figura 8-8. Varianza explicada en función del número de dimensiones.

Por último, si está utilizando la reducción de dimensionalidad como paso de preprocesamiento para una tarea de aprendizaje supervisado (por ejemplo, clasificación), puede ajustar el número de dimensiones como lo haría con cualquier otro hiperparámetro (consulte el Capítulo 2). Por ejemplo, el siguiente ejemplo de código crea una canalización de dos pasos: primero reduce la dimensionalidad mediante PCA y luego clasifica mediante un bosque aleatorio. A continuación, utiliza RandomizedSearchCV para encontrar una buena combinación de hiperparámetros tanto para PCA como para el clasificador de bosque aleatorio. Este ejemplo realiza una búsqueda rápida, ajusta solo 2 hiperparámetros, entrena en solo 1000 instancias y ejecuta solo 10 iteraciones, pero siéntete libre de hacer una búsqueda más exhaustiva si tienes tiempo:

```

desde sklearn.ensemble importar RandomForestClassifier desde
sklearn.model_selection importar RandomizedSearchCV desde sklearn.pipeline
importar make_pipeline

clf = make_pipeline(PCA(estado_aleatorio=42),
                    Clasificador de bosque aleatorio (estado_aleatorio = 42))
param_distrib =
    { "pca__n_components": np.arange(10, 80),
      "randomforestclassifier__n_estimators": np.arange(50, 500)}

} rnd_search = RandomizedSearchCV(clf, param_distrib, n_iter=10, cv=3,

```

```
rnd_search.fit(X_train[:1000], y_train[:1000],  
                estado_aleatorio=42)
```

Veamos los mejores hiperparámetros encontrados:

```
>>> print(rnd_search.best_params_)  
{'randomforestclassifier__n_estimators': 465, 'pca__n_components': 23}
```

Es interesante notar cuán bajo es el número óptimo de componentes: ¡redujimos un conjunto de datos de 784 dimensiones a solo 23 dimensiones! Esto está relacionado con el hecho de que utilizamos un bosque aleatorio, que es un modelo bastante potente. Si en su lugar usáramos un modelo lineal, como un SGDClassifier, la búsqueda encontraría que necesitamos preservar más dimensiones (alrededor de 70).

## PCA para compresión

Después de la reducción de dimensionalidad, el conjunto de entrenamiento ocupa mucho menos espacio. Por ejemplo, después de aplicar PCA al conjunto de datos MNIST preservando el 95% de su varianza, nos quedan 154 características, en lugar de las 784 características originales. Entonces, el conjunto de datos ahora tiene menos del 20% de su tamaño original y ¡solo perdemos el 5% de su variación! Esta es una relación de compresión razonable y es fácil ver cómo una reducción de tamaño de este tipo aceleraría enormemente un algoritmo de clasificación.

También es posible descomprimir el conjunto de datos reducido a 784 dimensiones aplicando la transformación inversa de la proyección PCA. Esto no le devolverá los datos originales, ya que la proyección perdió un poco de información (dentro de la variación del 5% que se eliminó), pero probablemente estará cerca de los datos originales. La distancia media cuadrática entre los datos originales y los datos reconstruidos (comprimidos y luego descomprimidos) se denomina error de reconstrucción.

El método `inverse_transform()` nos permite descomprimir el reducido Conjunto de datos MNIST de nuevo a 784 dimensiones:

```
X_recuperado = pca.inverse_transform(X_reducido)
```

**La Figura 8-9** muestra algunos dígitos del conjunto de entrenamiento original (a la izquierda) y los dígitos correspondientes después de la compresión y descompresión. Puede ver que hay una ligera pérdida de calidad de imagen, pero los dígitos aún están prácticamente intactos.

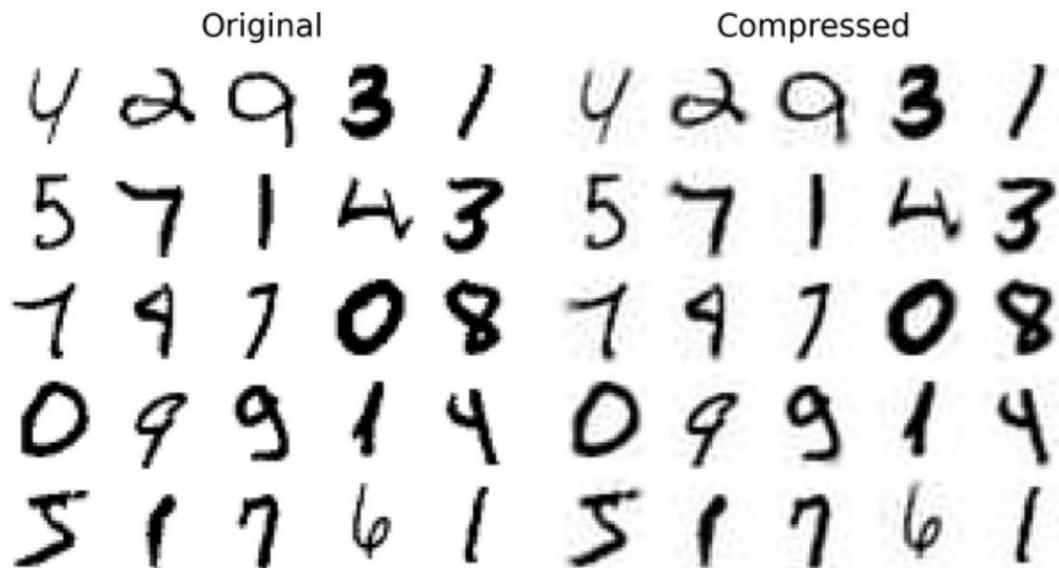


Figura 8-9. Compresión MNIST que preserva el 95% de la varianza

La ecuación para la transformación inversa se muestra en [la Ecuación 8-3](#).

Ecuación 8-3. Transformación inversa PCA, de vuelta al número original de dimensiones

$$X_{\text{recuperado}} = X_d \cdot \text{proj}W_d$$

## PCA aleatorizado

Si configura el hiperparámetro `svd_solver` en "aleatorio", Scikit-Learn utiliza un algoritmo estocástico llamado PCA aleatorio que encuentra rápidamente una aproximación de los primeros  $d$  componentes principales. Su complejidad computacional es  $O(m \times d) + O(d)$ , en lugar de  $O^3(m \times n) + O(n)$  para el enfoque SVD completo, por lo que es dramáticamente más rápido que el SVD completo cuando  $d$  es mucho menor que  $n$  :

```
rnd_pca = PCA(n_components=154, svd_solver="randomized",
               random_state=42)
X_reducido = rnd_pca.fit_transform(X_train)
```

## CONSEJO

De forma predeterminada, `svd_solver` en realidad está configurado en "auto": Scikit-Learn usa automáticamente el algoritmo PCA aleatorio si  $\max(m, n) > 500$  y `n_components` es un número entero menor que el 80% de  $\min(m, n)$ , o de lo contrario utiliza el enfoque SVD completo. Por lo tanto, el código anterior usaría el algoritmo PCA aleatorio incluso si eliminara el argumento `svd_solver="randomized"`, ya que  $154 < 0,8 \times 784$ . Si desea forzar a Scikit-Learn a usar SVD completo para obtener un resultado un poco más preciso, puede establecer el hiperparámetro `svd_solver` en "completo".

## PCA incremental

Un problema con las implementaciones anteriores de PCA es que requieren que todo el conjunto de entrenamiento quepa en la memoria para que se ejecute el algoritmo.

Afortunadamente, se han desarrollado algoritmos de PCA incremental (IPCA) que le permiten dividir el conjunto de entrenamiento en minilotes y alimentarlos en un minilote a la vez. Esto es útil para conjuntos de capacitación grandes y para aplicar PCA en línea (es decir, sobre la marcha, a medida que llegan nuevas instancias).

El siguiente código divide el conjunto de entrenamiento MNIST en 100 minilotes (usando la función `array_split()` de NumPy) y los envía a la clase `IncrementalPCA` de Scikit-Learn para reducir la dimensionalidad del conjunto <sup>5</sup> de datos MNIST a 154 dimensiones, como antes. Tenga en cuenta que debe llamar al método `partial_fit()` con cada mini-lote, en lugar del método `fit()` con todo el conjunto de entrenamiento:

```
desde sklearn.decomposition importar PCA incremental
```

```
n_batches = 100
inc_pca = PCA incremental (n_components=154) para
X_batch en np.array_split(X_train, n_batches): inc_pca.partial_fit(X_batch)

X_reducido = inc_pca.transform(X_train)
```

Alternativamente, puede usar la clase `memmap` de NumPy, que le permite manipular una matriz grande almacenada en un archivo binario en el disco como si estuviera completamente en la memoria; la clase carga solo los datos que necesita en la memoria, cuando los necesita. Para demostrar esto, primero creamos un archivo mapeado en memoria (`memmap`) y copiemos en él el conjunto de entrenamiento MNIST, luego llamemos a `flush()` para asegurarnos de que cualquier dato que aún esté en el caché se guarde en el disco. En la vida real, `X_train` normalmente no encajaría

en la memoria, por lo que lo cargarías fragmento por fragmento y guardarías cada fragmento en la parte derecha de la matriz memmap:

```
nombre de archivo = "my_mnist.memmap"
X_memmap = np.memmap(nombre de archivo, dtype='float32', modo='write',
forma=X_train.shape)
X_memmap[:] = X_train # podría ser un bucle en su lugar, guardando los datos fragmento por
fragmento
X_memmap.flush()
```

A continuación, podemos cargar el archivo memmap y usarlo como una matriz NumPy normal. Usemos la clase IncrementalPCA para reducir su dimensionalidad. Dado que este algoritmo utiliza sólo una pequeña parte de la matriz en un momento dado, el uso de la memoria permanece bajo control. Esto hace posible llamar al método fit() habitual en lugar de part\_fit(), lo cual es bastante conveniente:

```
X_memmap = np.memmap(nombre de archivo, dtype="float32",
modo="readonly").reshape(-1, 784) tamaño_lote =
X_memmap.shape[0] // n_batches inc_pca = PCA
incremental(n_components=154, tamaño_lote=tamaño_lote ) inc_pca.fit(X_memmap)
```

#### ADVERTENCIA

Sólo los datos binarios sin procesar se guardan en el disco, por lo que debe especificar el tipo de datos y la forma de la matriz cuando la carga. Si omite la forma, np.memmap() devuelve una matriz 1D.

Para conjuntos de datos de muy altas dimensiones, PCA puede ser demasiado lento. Como vio anteriormente, incluso si usa PCA aleatorio, su complejidad computacional sigue siendo  $O(m \times d^2 + O(d))$ , por lo que el número objetivo de dimensiones  $d$  no debe ser demasiado grande. Si se trata de un conjunto de datos con decenas de miles de características o más (por ejemplo, imágenes), entonces el entrenamiento puede volverse demasiado lento: en este caso, debería considerar utilizar una proyección aleatoria en su lugar.

## Proyección aleatoria

Como sugiere su nombre, el algoritmo de proyección aleatoria proyecta los datos a un espacio de dimensiones inferiores mediante una proyección lineal aleatoria. Esto puede parecer una locura, pero resulta que es muy probable que una proyección tan aleatoria

preservar las distancias bastante bien, como lo demostraron matemáticamente William B. Johnson y Joram Lindenstrauss en un famoso lema. Entonces, dos instancias similares seguirán siendo similares después de la proyección, y dos instancias muy diferentes seguirán siendo muy diferentes.

Obviamente, cuantas más dimensiones se eliminan, más información se pierde y más distancias se distorsionan. Entonces, ¿cómo se puede elegir el número óptimo de dimensiones? Bueno, a Johnson y Lindenstrauss se les ocurrió una ecuación que determina el número mínimo de dimensiones a preservar para garantizar, con alta probabilidad, que las distancias no cambien más que una tolerancia determinada. Por ejemplo, si tiene un conjunto de datos que contiene  $m = 5000$  instancias con  $n = 2000$  características cada una y no desea que la distancia al cuadrado entre dos instancias cambie en más de  $\epsilon = 10\%$ , entonces debe proyectar los datos hacia abajo.<sup>6</sup> a  $d$  dimensiones, con  $d \geq 4 \log(m) / (\frac{1}{2}\epsilon^2 - \frac{1}{3}\epsilon^3)$ , que son 7300 dimensiones. ¡Esa es una reducción de dimensionalidad bastante significativa! Observe que la ecuación no usa  $n$ , solo se basa en  $m$  y  $\epsilon$ .

Esta ecuación se implementa mediante la función `johnson_lindenstrauss_min_dim()`:

```
>>> de sklearn.random_projection importar
johnson_lindenstrauss_min_dim >>> m, ε =
5_000, 0.1 >>> d =
johnson_lindenstrauss_min_dim(m, eps=ε ) >>> d
```

7300

Ahora podemos simplemente generar una matriz aleatoria  $P$  de forma  $[d, n]$ , donde cada elemento se muestrea aleatoriamente de una distribución gaussiana con media 0 y varianza  $1/d$ , y usarla para proyectar un conjunto de datos desde  $n$  dimensiones hasta  $d$ :

```
n = 20_000
np.semilla.aleatoria(42)
P = np.random.randn(d, n) / np.sqrt(d) # std dev = raíz cuadrada de la varianza
```

```
X = np.random.randn(m, n) # generar un conjunto de datos falso X_reduced =
X @ PT
```

¡Eso es todo al respecto! Es simple y eficiente y no requiere capacitación: lo único que necesita el algoritmo para crear la matriz aleatoria es la forma del conjunto de datos. Los datos en sí no se utilizan en absoluto.

Scikit-Learn ofrece una clase GaussianRandomProjection para hacer exactamente lo que acabamos de hacer: cuando llamas a su método fit(), usa johnson\_lindenstrauss\_min\_dim() para determinar la dimensionalidad de salida, luego genera una matriz aleatoria, que almacena en el atributo componentes\_. Luego, cuando llamas a transform(), utiliza esta matriz para realizar la proyección. Al crear el transformador, puede configurar eps si desea modificar  $\epsilon$  (el valor predeterminado es 0.1) y n\_components si desea forzar una dimensionalidad objetivo específica d. El siguiente ejemplo de código proporciona el mismo resultado que el código anterior (también puede verificar que gaussian\_rnd\_proj.components\_ es igual a P):

```
de sklearn.random_projection importar GaussianRandomProjection

gaussian_rnd_proj = GaussianRandomProjection(eps=ε , random_state=42)

X_reduced = gaussian_rnd_proj.fit_transform(X) # mismo resultado que
arriba
```

Scikit-Learn también proporciona un segundo transformador de proyección aleatoria, conocido como SparseRandomProjection. Determina la dimensionalidad del objetivo de la misma manera, genera una matriz aleatoria de la misma forma y realiza la proyección de manera idéntica. La principal diferencia es que la matriz aleatoria es escasa. Esto significa que utiliza mucha menos memoria: unos 25 MB en lugar de casi 1,2 GB en el ejemplo anterior. Y también es mucho más rápido, tanto para generar la matriz aleatoria como para reducir la dimensionalidad: aproximadamente un 50% más rápido en este caso. Además, si la entrada es escasa, la transformación la mantiene escasa (a menos que establezca densa\_output=True). Por último, disfruta de la misma propiedad de conservación de la distancia que el enfoque anterior y la calidad de la reducción de dimensionalidad es comparable. En resumen, normalmente es preferible utilizar este transformador en lugar del primero, especialmente para conjuntos de datos grandes o dispersos.

La proporción  $r$  de elementos distintos de cero en la matriz aleatoria dispersa se llama densidad. Por defecto, es igual a  $1/\sqrt{n}$ . Con 20.000 características, esto significa que sólo 1 de cada 141 celdas en la matriz aleatoria es distinta de cero: ¡eso es bastante escaso! Puede establecer el hiperparámetro de densidad en otro valor si lo prefiere. Cada celda de la matriz aleatoria dispersa tiene una probabilidad  $r$  de ser distinta de cero, y cada celda distinta de cero El valor es  $-v$  o  $+v$  (ambos igualmente probables), donde  $v = 1/\sqrt{dr}$ .

Si desea realizar la transformación inversa, primero debe calcular la pseudoinversa de la matriz de componentes usando la función `pinv()` de SciPy, luego multiplicar los datos reducidos por la transposición de la pseudoinversa:

```
componentes_pinv = np.linalg.pinv(gaussian_rnd_proj.components_)
X_recuperado = X_reducido @ componentes_pinv.T
```

#### ADVERTENCIA

Calcular la pseudoinversa puede llevar mucho tiempo si la matriz de componentes es grande, ya que la complejidad computacional de `pinv()` es  $O(dn^2)$  si  $d < n$ , o  $O(nd^2)$  en caso contrario.

En resumen, la proyección aleatoria es un algoritmo de reducción de dimensionalidad simple, rápido, eficiente en memoria y sorprendentemente poderoso que debe tener en cuenta, especialmente cuando trabaja con conjuntos de datos de alta dimensión.

#### NOTA

La proyección aleatoria no siempre se utiliza para reducir la dimensionalidad de grandes conjuntos de datos. Por ejemplo, un [artículo de 2017](#) por Sanjoy Dasgupta et al. demostró que el cerebro de una mosca de la fruta implementa un análogo de la proyección aleatoria para mapear entradas olfativas densas de baja dimensión en salidas binarias escasas de alta dimensión: para cada olor, sólo se activa una pequeña fracción de las neuronas de salida, pero olores similares activan muchas de las mismas neuronas. Esto es similar a un algoritmo conocido llamado hash sensible a la localidad (LSH), que normalmente se utiliza en los motores de búsqueda para agrupar documentos similares.

## LLE

**Incrustación localmente lineal (LLE)**<sup>8</sup> es una técnica de reducción de dimensionalidad no lineal (NLDR). Es una técnica de aprendizaje múltiple que no se basa en proyecciones, a diferencia del PCA y la proyección aleatoria. En pocas palabras, LLE funciona midiendo primero cómo cada instancia de entrenamiento se relaciona linealmente con sus vecinos más cercanos y luego buscando una representación de baja dimensión del conjunto de entrenamiento donde estas relaciones locales se conservan mejor (más detalles en breve). Este enfoque lo hace particularmente bueno para desenrollar colectores retorcidos, especialmente cuando no hay demasiado ruido.

El siguiente código realiza una tirada suiza y luego utiliza Scikit-Learn. Clase LocallyLinearEmbedding para desenrollarlo:

```
desde sklearn.datasets importar make_swiss_roll desde
sklearn.manifold importar LocallyLinearEmbedding

X_suizo, t = make_swiss_roll(n_samples=1000, noise=0.2, random_state=42)
lle =
LocallyLinearEmbedding(n_components=2, n_neighbors=10, random_state=42)

X_unrolled = lle.fit_transform(X_suizo)
```

La variable `t` es una matriz NumPy 1D que contiene la posición de cada instancia a lo largo del eje enrollado del rollo suizo. No lo usamos en este ejemplo, pero puede usarse como objetivo para una tarea de regresión no lineal.

El conjunto de datos 2D resultante se muestra en [la Figura 8-10](#). Como puede ver, el rollo suizo está completamente desenrollado y las distancias entre copias están localmente bien conservadas. Sin embargo, las distancias no se mantienen a mayor escala: el rollo suizo desenrollado debe ser un rectángulo, no una especie de banda estirada y retorcida. Sin embargo, LLE hizo un buen trabajo modelando la variedad.

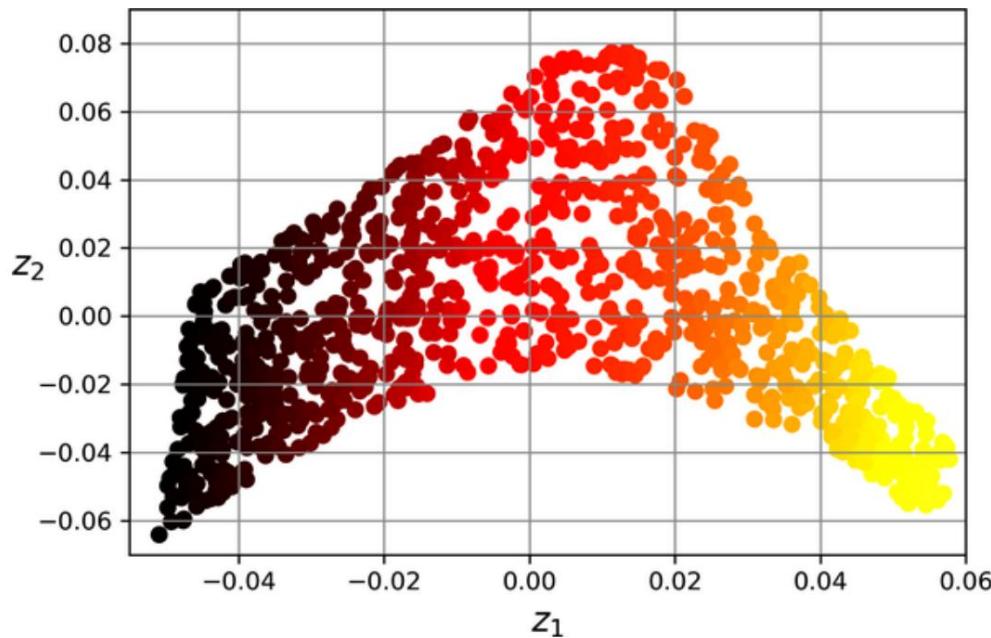


Figura 8-10. Rollo suizo desenrollado mediante LLE

Así es como funciona LLE: para cada instancia de  $(i)$ , el algoritmo identifica entrenamiento  $x$  sus  $k$  vecinos más cercanos (en el código anterior  $k = 10$ ), luego intenta

reconstruir  $x^{(j)}$  como una función lineal de estos vecinos. Más específicamente, intenta encontrar los pesos  $w_i$  tales que la distancia al cuadrado entre  $x^{(j)}$  y  $y_{o,j}$  sea lo más pequeña posible, asumiendo  $w_i = 0$  si  $x^{(j)}$  no es uno de los  $\sum_{mj=1}^k w_{i,j} x^{(j)}$ . Los  $k$  vecinos más cercanos de  $x^{(i)}$ . Así, el primer paso de LLE es el problema de optimización restringida descrito en [la ecuación 8-4](#), donde  $W$  es la matriz de pesos que contiene todos los pesos  $w_i$ . La segunda restricción simplemente normaliza los pesos  $i,j$  ( $i$ ) para cada instancia de entrenamiento  $x^{(i)}$ .

Ecuación 8-4. Paso 1 de LLE: modelar linealmente las relaciones locales

$$\begin{aligned} W^* = \arg \min \sum_{i=1}^n \|x^{(i)} - \sum_{j=1}^k w_{i,j} x^{(j)}\|^2 \\ \text{subject to } \{w_{i,j} = 0 \quad \sum_{mj=1}^k w_{i,j} = 1 \text{ para } i = 1, 2, \dots, m\} \end{aligned}$$

Después de este paso, la matriz de pesos  $W^*$  (que contiene los pesos  $w_{i,j}^*$ ) codifica las relaciones lineales locales entre las instancias de entrenamiento. El segundo paso es mapear las instancias de entrenamiento en un espacio d-dimensional (donde  $d < n$ ) preservando al mismo tiempo estas relaciones locales tanto como sea posible. Si  $z$  es la imagen de  $x$  en este espacio d-dimensional, entonces queremos la distancia al cuadrado entre  $z$  y  $\sum_{j=1}^k w_{i,j} z^{(j)}$  sea lo más pequeño posible. Esta idea lleva a el problema de optimización sin restricciones descrito en [la Ecuación 8-5](#). Se ve muy similar al primer paso, pero en lugar de mantener las instancias fijas y encontrar los pesos óptimos, estamos haciendo lo contrario: mantener los pesos fijos y encontrar la posición óptima de las imágenes de las instancias en la baja ( $i$ ) dimensión . espacio. Tenga en cuenta que  $Z$  es la matriz que contiene todos los  $z^{(i)}$ .

Ecuación 8-5. LLE paso 2: reducir la dimensionalidad preservando las relaciones

$$Z^* = \arg \min \sum_{i=1}^n \|z^{(i)} - \sum_{j=1}^k w_{i,j} z^{(j)}\|^2$$

La implementación LLE de Scikit-Learn tiene la siguiente complejidad computacional:  $O(m \log(m)n \log(k))$  para encontrar los  $k$  vecinos más cercanos,  $O(mnk^3)$  para optimizar los pesos, y  $O(dm^2)$  para construir el modelo de baja dimensión

representaciones. Desafortunadamente, la  $m^2$  en el último término hace que este algoritmo escale mal a conjuntos de datos muy grandes.

Como puede ver, LLE es bastante diferente de las técnicas de proyección y es significativamente más complejo, pero también puede construir representaciones de baja dimensión mucho mejores, especialmente si los datos no son lineales.

## Otras técnicas de reducción de dimensionalidad

Antes de concluir este capítulo, echemos un vistazo rápido a algunas otras técnicas populares de reducción de dimensionalidad disponibles en Scikit-Learn:

### `sklearn.manifold.MDS`

El escalado multidimensional (MDS) reduce la dimensionalidad al tiempo que intenta preservar las distancias entre las instancias. La proyección aleatoria hace eso con datos de alta dimensión, pero no funciona bien con datos de baja dimensión.

### `sklearn.manifold.Isomap`

Isomap crea un gráfico conectando cada instancia con sus vecinos más cercanos, luego reduce la dimensionalidad mientras intenta preservar las distancias geodésicas entre las instancias. La distancia geodésica entre dos nodos en un gráfico es el número de nodos en el camino más corto entre estos nodos.

### `sklearn.manifold.TSNE`

La incrustación de vecinos estocásticos distribuidos en t (t-SNE) reduce la dimensionalidad al intentar mantener instancias similares cercanas y instancias diferentes separadas. Se utiliza principalmente para visualización, en particular para visualizar grupos de instancias en un espacio de alta dimensión. Por ejemplo, en los ejercicios al final de este capítulo utilizará t-SNE para visualizar un mapa 2D de las imágenes MNIST.

### `sklearn.discriminant_analysis.LinearDiscriminantAnalysis`

El análisis discriminante lineal (LDA) es un algoritmo de clasificación lineal que, durante el entrenamiento, aprende los ejes más discriminativos entre las clases. Luego, estos ejes se pueden utilizar para definir un hiperplano en el que proyectar los datos. El beneficio de este enfoque es que la proyección se mantendrá

clases lo más separadas posible, por lo que LDA es una buena técnica para reducir la dimensionalidad antes de ejecutar otro algoritmo de clasificación (a menos que LDA por sí sola sea suficiente).

**La Figura 8-11** muestra los resultados de MDS, Isomap y t-SNE en el rollo suizo.

MDS consigue aplanar el rollo suizo sin perder su curvatura global, mientras que Isomap lo elimina por completo. Dependiendo de la tarea posterior, preservar la estructura a gran escala puede ser bueno o malo. t-SNE hace un trabajo razonable al aplanar el rollo suizo, preservando un poco de curvatura, y también amplifica los grupos, desgarrando el rollo. Nuevamente, esto puede ser bueno o malo, dependiendo de la tarea posterior.

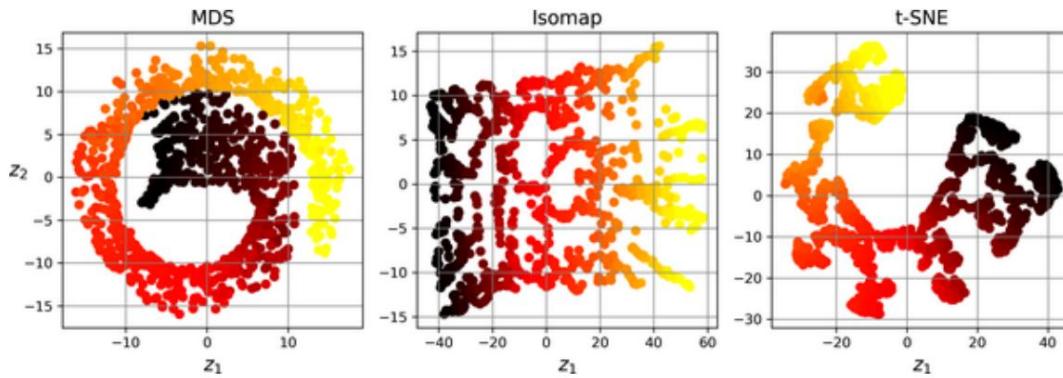


Figura 8-11. Usando varias técnicas para reducir el rollo suizo a 2D

## Ejercicios

1. ¿Cuáles son las principales motivaciones para reducir la dimensionalidad de un conjunto de datos? ¿Cuáles son los principales inconvenientes?
2. ¿Cuál es la maldición de la dimensionalidad?
3. Una vez que se ha reducido la dimensionalidad de un conjunto de datos, ¿es posible revertir la operación? ¿Si es así, cómo? Si no, ¿por qué?
4. ¿Se puede utilizar PCA para reducir la dimensionalidad de un sistema altamente no lineal? conjunto de datos?
5. Suponga que realiza PCA en un conjunto de datos de 1000 dimensiones y establece la relación de varianza explicada en 95 %. ¿Cuántas dimensiones tendrá el conjunto de datos resultante?

6. ¿En qué casos utilizaría PCA regular, PCA incremental, aleatorizado?  
¿PCA o proyección aleatoria?
7. ¿Cómo se puede evaluar el desempeño de una reducción de dimensionalidad?  
algoritmo en su conjunto de datos?
8. ¿Tiene algún sentido encadenar dos reducciones de dimensionalidad diferentes?  
algoritmos?
9. Cargue el conjunto de datos MNIST (presentado en [el Capítulo 3](#)) y divídalo en un conjunto de entrenamiento y un conjunto de prueba (tome las primeras 60.000 instancias para entrenamiento y las 10.000 restantes para prueba). Entrene un clasificador de bosque aleatorio en el conjunto de datos y cronometre cuánto tiempo lleva, luego evalúe el modelo resultante en el conjunto de prueba. A continuación, utilice PCA para reducir la dimensionalidad del conjunto de datos, con una tasa de varianza explicada del 95 %. Entrene un nuevo clasificador de bosque aleatorio en el conjunto de datos reducido y vea cuánto tiempo lleva. ¿El entrenamiento fue mucho más rápido? A continuación, evalúe el clasificador en el conjunto de prueba. ¿Cómo se compara con el clasificador anterior? Inténtelo de nuevo con un SGDClassifier. ¿Cuánto ayuda PCA ahora?
10. Utilice t-SNE para reducir las primeras 5000 imágenes del conjunto de datos MNIST a 2 dimensiones y trace el resultado utilizando Matplotlib. Puede utilizar un diagrama de dispersión con 10 colores diferentes para representar la clase objetivo de cada imagen. Alternativamente, puede reemplazar cada punto en el diagrama de dispersión con la clase de instancia correspondiente (un dígito del 0 al 9), o incluso trazar versiones reducidas de las imágenes de los dígitos (si traza todos los dígitos, la visualización estará demasiado desordenada, por lo que (debe extraer una muestra aleatoria o trazar un caso sólo si no se ha trazado ningún otro caso a una distancia cercana). Debería obtener una buena visualización con grupos de dígitos bien separados. Intente utilizar otros algoritmos de reducción de dimensionalidad, como PCA, LLE o MDS, y compare las visualizaciones resultantes.

Las soluciones a estos ejercicios están disponibles al final del cuaderno de este capítulo, en <https://homl.info/colab3>.

---

<sup>1</sup> Bueno, cuatro dimensiones si cuentas el tiempo, y algunas más si eres un teórico de cuerdas.

<sup>2</sup> Observe un teseracto giratorio proyectado en el espacio 3D en <https://homl.info/30>. Imagen del usuario de Wikipedia NerdBoy1392 ([Creative Commons BY-SA 3.0](#)). Reproducido de <https://en.wikipedia.org/wiki/Tesseract>.

- 3** Dato curioso: cualquiera que conozcas probablemente sea extremista en al menos una dimensión (por ejemplo, cuánta azúcar le ponen al café), si consideras suficientes dimensiones.
- 4** Karl Pearson, “On Lines and Planes of Closest Fit to Systems of Points in Space”, *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2, no. 11 (1901): 559–572.
- 5** Scikit-Learn utiliza el [algoritmo](#) descrito en David A. Ross et al., “Incremental Learning for Robust Visual Tracking”, *Revista Internacional de Visión por Computadora* 77, no. 1–3 (2008): 125–141.
- 6**  $\epsilon$  es la letra griega épsilon, utilizada a menudo para valores pequeños.
- 7** Sanjoy Dasgupta et al., “Un algoritmo neuronal para un problema informático fundamental”, *Science* 358, no. 6364 (2017): 793–796.
- 8** Sam T. Roweis y Lawrence K. Saul, “Reducción de dimensionalidad no lineal mediante Incrustación lineal”, *Science* 290, no. 5500 (2000): 2323–2326.

# Capítulo 9. Técnicas de aprendizaje no supervisado

---

Aunque la mayoría de las aplicaciones del aprendizaje automático hoy en día se basan en el aprendizaje supervisado (y como resultado, aquí es donde se destina la mayor parte de las inversiones), la gran mayoría de los datos disponibles no están etiquetados: tenemos las características de entrada  $X$ , pero no tengo las etiquetas  $y$ . El científico informático Yann LeCun dijo la famosa frase: "si la inteligencia fuera un pastel, el aprendizaje no supervisado sería el pastel, el aprendizaje supervisado sería la guinda del pastel y el aprendizaje por refuerzo sería la guinda del pastel". En otras palabras, existe un enorme potencial en el aprendizaje no supervisado al que apenas hemos empezado a hincarle el diente.

Supongamos que desea crear un sistema que tome algunas fotografías de cada artículo en una línea de producción y detecte qué artículos están defectuosos.

Puede crear con bastante facilidad un sistema que tome fotografías automáticamente, y esto podría brindarle miles de fotografías cada día. Luego podrá crear un conjunto de datos razonablemente grande en tan solo unas pocas semanas. Pero espera, ¡no hay etiquetas! Si desea entrenar un clasificador binario normal que prediga si un artículo es defectuoso o no, deberá etiquetar cada imagen como "defectuosa" o "normal".

Por lo general, esto requerirá que expertos humanos se sienten y revisen manualmente todas las imágenes. Esta es una tarea larga, costosa y tediosa, por lo que normalmente sólo se realizará en un pequeño subconjunto de las imágenes disponibles. Como resultado, el conjunto de datos etiquetado será bastante pequeño y el rendimiento del clasificador será decepcionante.

Además, cada vez que la empresa realiza algún cambio en sus productos, será necesario empezar de nuevo todo el proceso desde cero. ¿No sería fantástico si el algoritmo pudiera explotar los datos sin etiquetar sin necesidad de que los humanos etiqueten cada imagen? Ingrese al aprendizaje no supervisado.

En el Capítulo 8 analizamos la tarea de aprendizaje no supervisado más común: la reducción de dimensionalidad. En este capítulo veremos algunas tareas más no supervisadas:

### Agrupación

El objetivo es agrupar instancias similares en clústeres.

La agrupación en clústeres es una gran herramienta para análisis de datos, segmentación de clientes, sistemas de recomendación, motores de búsqueda, segmentación de imágenes, aprendizaje semisupervisado, reducción de dimensionalidad y más.

### Detección de anomalías (también llamada detección de valores atípicos)

El objetivo es aprender cómo son los datos "normales" y luego utilizarlos para detectar casos anormales. Estos casos se denominan anomalías o valores atípicos, mientras que los casos normales se denominan valores internos. La detección de anomalías es útil en una amplia variedad de aplicaciones, como la detección de fraudes, la detección de productos defectuosos en la fabricación, la identificación de nuevas tendencias en series temporales o la eliminación de valores atípicos de un conjunto de datos antes de entrenar otro modelo, lo que puede mejorar significativamente el rendimiento del modelo resultante. .

### Estimación de densidad

Esta es la tarea de estimar la función de densidad de probabilidad (PDF) del proceso aleatorio que generó el conjunto de datos. La estimación de densidad se utiliza comúnmente para la detección de anomalías: es probable que las instancias ubicadas en regiones de muy baja densidad sean anomalías. También es útil para el análisis y visualización de datos.

¿Listo para un poco de pastel? Comenzaremos con dos algoritmos de agrupamiento, k-medias y DBSCAN, luego discutiremos los modelos de mezcla gaussiana y veremos cómo se pueden usar para la estimación de densidad, agrupamiento y detección de anomalías.

## Algoritmos de agrupamiento: k-means y DBSCAN

Mientras disfrutas de una caminata por las montañas, te topas con una planta que nunca antes habías visto. Miras a tu alrededor y notas algunos más. No son idénticos, pero son lo suficientemente similares como para saber que lo más probable es que pertenezcan a la misma especie (o al menos al mismo género). Es posible que necesites que un botánico te diga qué especie es, pero ciertamente no necesitas un experto para identificar grupos de objetos de apariencia similar. A esto se le llama clustering: es la tarea de identificar instancias similares y asignarlas a clusters, o grupos de instancias similares.

Al igual que en la clasificación, cada instancia se asigna a un grupo.

Sin embargo, a diferencia de la clasificación, la agrupación es una tarea no supervisada.

Considere la [Figura 9-1](#): a la izquierda está el conjunto de datos de iris (presentado en el [Capítulo 4](#)), donde la especie de cada instancia (es decir, su clase) se representa con un marcador diferente. Es un conjunto de datos etiquetados, para el cual los algoritmos de clasificación como la regresión logística, los SVM o los clasificadores forestales aleatorios son muy adecuados. A la derecha está el mismo conjunto de datos, pero sin las etiquetas, por lo que ya no se puede utilizar un algoritmo de clasificación.

Aquí es donde intervienen los algoritmos de agrupación: muchos de ellos pueden detectar fácilmente el grupo inferior izquierdo. También es bastante fácil de ver con nuestros propios ojos, pero no es tan obvio que el grupo superior derecho esté compuesto por dos subgrupos distintos. Dicho esto, el conjunto de datos tiene dos características adicionales (largo y ancho del sépalo) que no se representan aquí, y los algoritmos de agrupamiento pueden hacer un buen uso de todas las características, por lo que, de hecho, identifican los tres grupos bastante bien (por ejemplo, usando un modelo de mezcla gaussiana). , solo 5 instancias de 150 están asignadas al clúster incorrecto).

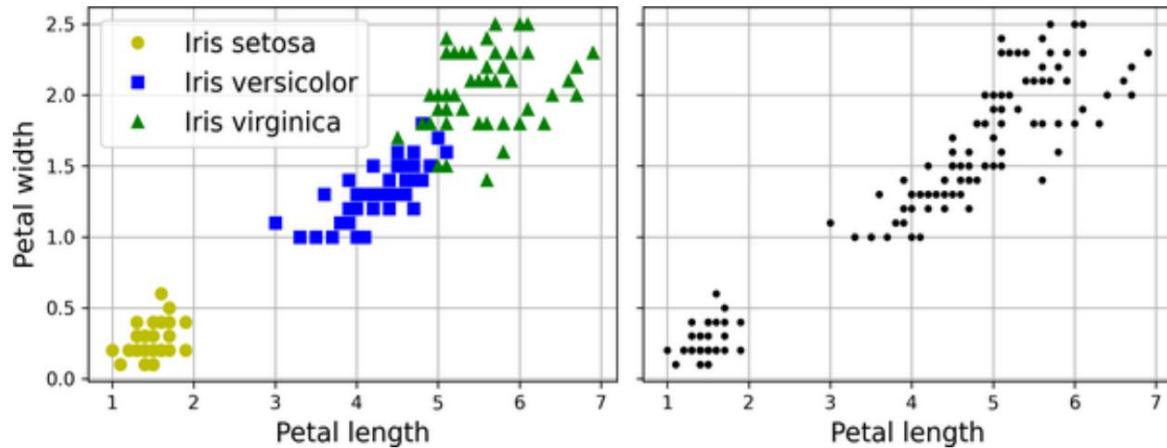


Figura 9-1. Clasificación (izquierda) versus agrupación (derecha)

La agrupación en clústeres se utiliza en una amplia variedad de aplicaciones, que incluyen:

#### Segmentación de clientes

Puede agrupar a sus clientes en función de sus compras y su actividad en su sitio web. Esto es útil para entender quiénes son tus clientes y qué necesitan, para que puedas adaptar tus productos y campañas de marketing a cada segmento. Por ejemplo, la segmentación de clientes puede resultar útil en los sistemas de recomendación para sugerir contenido que disfrutaron otros usuarios del mismo grupo.

#### Análisis de los datos

Cuando analiza un nuevo conjunto de datos, puede resultar útil ejecutar un algoritmo de agrupación y luego analizar cada grupo por separado.

#### Reducción de dimensionalidad

Una vez que se ha agrupado un conjunto de datos, generalmente es posible medir la afinidad de cada instancia con cada grupo; La afinidad es cualquier medida de qué tan bien encaja una instancia en un clúster. El vector de características  $x$  de cada instancia se puede reemplazar con el vector de sus afinidades de grupo. Si hay  $k$  grupos, entonces este vector es  $k$ -dimensional. El nuevo vector suele tener dimensiones mucho más bajas.

que el vector de características original, pero puede conservar suficiente información para su posterior procesamiento.

### Ingeniería de características

Las afinidades de clúster a menudo pueden resultar útiles como funciones adicionales. Por ejemplo, utilizamos k-means en [el Capítulo 2](#) para agregar características de afinidad de conglomerados geográficos al conjunto de datos de vivienda de California, y nos ayudaron a obtener un mejor rendimiento.

### Detección de anomalías (también llamada detección de valores atípicos)

Cualquier instancia que tenga baja afinidad con todos los clústeres probablemente sea una anomalía. Por ejemplo, si ha agrupado a los usuarios de su sitio web en función de su comportamiento, puede detectar usuarios con comportamientos inusuales, como una cantidad inusual de solicitudes por segundo.

### Aprendizaje semisupervisado

Si solo tiene unas pocas etiquetas, puede realizar una agrupación en clústeres y propagar las etiquetas a todas las instancias del mismo clúster. Esta técnica puede aumentar en gran medida la cantidad de etiquetas disponibles para un algoritmo de aprendizaje supervisado posterior y, por lo tanto, mejorar su rendimiento.

### Los motores de búsqueda

Algunos motores de búsqueda le permiten buscar imágenes similares a una imagen de referencia. Para construir un sistema de este tipo, primero aplicaría un algoritmo de agrupamiento a todas las imágenes de su base de datos; imágenes similares terminarían en el mismo grupo. Luego, cuando un usuario proporciona una imagen de referencia, todo lo que necesita hacer es usar el modelo de agrupamiento entrenado para encontrar el grupo de esta imagen y luego simplemente podría devolver todas las imágenes de este grupo.

### Segmentación de imagen

Al agrupar los píxeles según su color y luego reemplazar el color de cada píxel con el color medio de su grupo, es posible reducir considerablemente el número de colores diferentes en una imagen.

La segmentación de imágenes se utiliza en muchos sistemas de detección y seguimiento de objetos, ya que facilita la detección del contorno de cada objeto.

No existe una definición universal de qué es un clúster: realmente depende del contexto y diferentes algoritmos capturarán diferentes tipos de clústeres. Algunos algoritmos buscan instancias centradas alrededor de un punto particular, llamado centroide . Otros buscan regiones continuas de instancias densamente pobladas: estos grupos pueden adoptar cualquier forma.

Algunos algoritmos son jerárquicos y buscan grupos de grupos. Y la lista continúa.

En esta sección, veremos dos algoritmos de agrupamiento populares, k-means y DBSCAN, y exploraremos algunas de sus aplicaciones, como la reducción de dimensionalidad no lineal, el aprendizaje semisupervisado y la detección de anomalías.

## k-significa

Considere el conjunto de datos sin etiqueta representado en [la Figura 9-2](#): puede ver claramente cinco manchas de instancias. El algoritmo k-means es un algoritmo simple capaz de agrupar este tipo de conjunto de datos de manera muy rápida y eficiente, a menudo en tan solo unas pocas iteraciones. Fue propuesta por Stuart Lloyd en los Laboratorios Bell en 1957 como técnica para la modulación de código de pulso, pero sólo se [publicó](#) fuera de la empresa<sup>1</sup> en 1982. En 1965, Edward W. Forgy había publicado prácticamente el mismo algoritmo, por lo que a veces se hace referencia a k-means como algoritmo de Lloyd-Forgy.

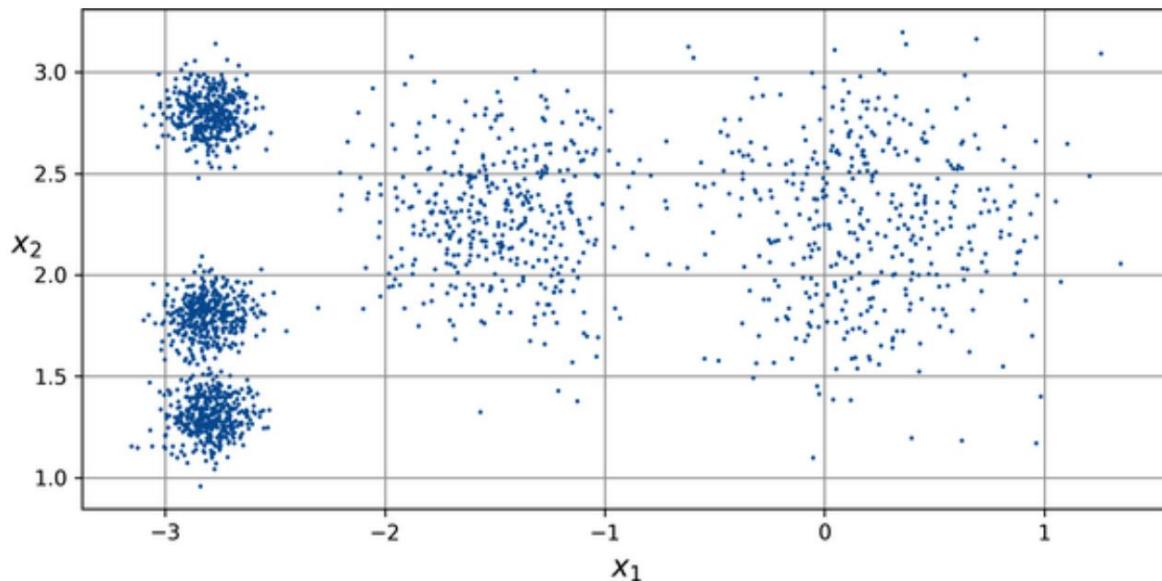


Figura 9-2. Un conjunto de datos sin etiquetar compuesto por cinco blobs de instancias.

Entrenemos un agrupador de k-medias en este conjunto de datos. Intentará encontrar el centro de cada blob y asignar cada instancia al blob más cercano:

```
desde sklearn.cluster importe KMeans desde
sklearn.datasets importe make_blobs

X, y = make_blobs([...]) # crea los blobs: y contiene los ID del clúster, pero nosotros
# no los usará; eso es lo que
queremos predecir k = 5

kmeans = KMeans(n_clusters=k, random_state=42) y_pred =
kmeans.fit_predict(X)
```

Tenga en cuenta que debe especificar el número de grupos  $k$  que debe encontrar el algoritmo. En este ejemplo, al observar los datos, resulta bastante obvio que  $k$  debe establecerse en 5, pero en general no es tan fácil. Discutiremos esto en breve.

Cada instancia se asignará a uno de los cinco grupos. En el contexto de la agrupación, la etiqueta de una instancia es el índice del clúster al que el algoritmo asigna esta instancia; Esto no debe confundirse con las etiquetas de clase en la clasificación, que se utilizan como objetivos.

(recuerde que la agrupación en clústeres es una tarea de aprendizaje no supervisada). La instancia de KMeans conserva las etiquetas previstas de las instancias en las que se entrenó, disponibles a través de la variable de instancia `etiquetas_`:

```
>>> y_pred
array([4, 0, 1, ..., 2, 1, 0], dtype=int32) >>> y_pred es
kmeans.labels_ True
```

También podemos echar un vistazo a los cinco centroides que encontró el algoritmo:

```
>>> kmeans.cluster_centers_
array([-2.80389616, 1.80117999], [0.20876306,
    2.25551336], [-2.79290307,
    2.79641063], [-1.46679593,
    2.28585348], [-2 .80037642,
    1.30082566]))
```

Puede asignar fácilmente nuevas instancias al clúster cuyo centroide esté más cercano:

```
>>> importar numpy como np
>>> X_new = np.array([[0, 2], [3, 2], [-3, 3], [-3, 2.5]]) >>> kmeans.predict (X_new)
matriz([1, 1, 2, 2], dtype=int32)
```

Si traza los límites de decisión del grupo, obtendrá una teselación de Voronoi: consulte [la Figura 9-3](#), donde cada centroide se representa con una X.

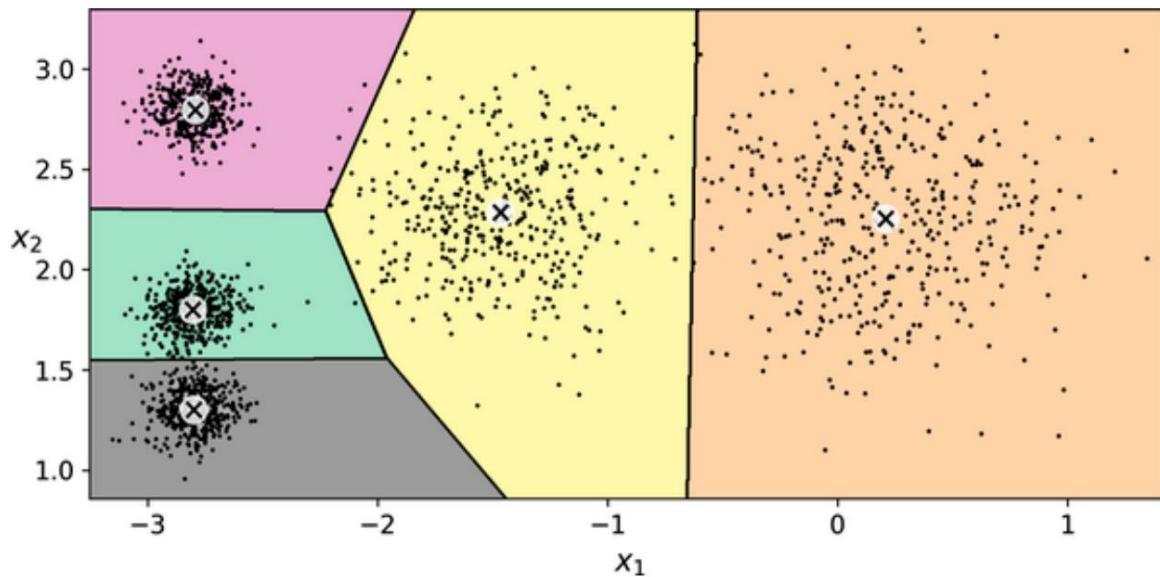


Figura 9-3. k-medias límites de decisión (teselación de Voronoi)

La gran mayoría de las instancias estaban claramente asignadas al grupo apropiado, pero algunas probablemente estaban mal etiquetadas, especialmente cerca del límite entre el grupo superior izquierdo y el grupo central. De hecho, el algoritmo k-means no se comporta muy bien cuando las manchas tienen diámetros muy diferentes porque lo único que le importa al asignar una instancia a un grupo es la distancia al centroide.

En lugar de asignar cada instancia a un único clúster, lo que se denomina agrupación estricta, puede resultar útil asignar a cada instancia una puntuación por clúster, lo que se denomina agrupación suave. La puntuación puede ser la distancia entre la instancia y el centroide o una puntuación de similitud (o afinidad), como la función de base radial gaussiana que utilizamos en el [Capítulo 2](#). En la clase KMeans, el método transform() mide la distancia desde cada instancia a cada centroide:

```
>>> kmeans.transform(X_new).round(2)
matrix([[2.81, 0.33, 2.9, 1.49, 2.89], [5.81, 2.8, 5.85,
    4.48, 5.84], [1.21, 3.29, 0.29, 1.69, 1.71],
    [0.73, 3.22, 0.36, 1.55, 1.22]])
```

En este ejemplo, la primera instancia en  $X_{\text{new}}$  está ubicada a una distancia de aproximadamente 2,81 del primer centroide, 0,33 del segundo centroide, 2,90 del tercer centroide, 1,49 del cuarto centroide y 2,89 del quinto centroide. Si tiene un conjunto de datos de alta dimensión y lo transforma de esta manera, terminará con un conjunto de datos de k dimensiones: esta transformación puede ser una técnica de reducción de dimensionalidad no lineal muy eficiente. Alternativamente, puedes usar estas distancias como características adicionales para entrenar otro modelo, como en [el Capítulo 2](#).

### El algoritmo k-means

Entonces, ¿cómo funciona el algoritmo? Bueno, supongamos que te dieron los centroides. Podrías etiquetar fácilmente todas las instancias del conjunto de datos asignando cada una de ellas al grupo cuyo centroide esté más cercano. Por el contrario, si recibiera todas las etiquetas de las instancias, podría ubicar fácilmente el centroide de cada grupo calculando la media de las instancias en ese grupo. Pero no se te dan ni las etiquetas ni los centroides, entonces, ¿cómo puedes proceder? Comience colocando los centroides al azar (por ejemplo, seleccionando k instancias al azar del conjunto de datos y usando sus ubicaciones como centroides). Luego etiquete las instancias, actualice los centroides, etiquete las instancias, actualice los centroides, y así sucesivamente hasta que los centroides dejen de moverse. Se garantiza que el algoritmo convergerá en un número finito de pasos (normalmente bastante pequeños). Esto se debe a que la distancia media cuadrática entre las instancias y sus centroides más cercanos solo puede disminuir en cada paso y, como no puede ser negativa, se garantiza que convergerá.

Puede ver el algoritmo en acción en [la Figura 9-4](#): los centroides se inicializan aleatoriamente (arriba a la izquierda), luego las instancias se etiquetan (arriba a la derecha), luego los centroides se actualizan (centro izquierda), las instancias se vuelven a etiquetar (centro derecha). ), etcétera. Como puede ver, en sólo tres iteraciones el algoritmo ha alcanzado un agrupamiento que parece cercano al óptimo.

## NOTA

La complejidad computacional del algoritmo es generalmente lineal con respecto al número de instancias m, el número de grupos k y el número de dimensiones n. Sin embargo, esto sólo es cierto cuando los datos tienen una estructura de agrupamiento. Si no es así, en el peor de los casos la complejidad puede aumentar exponencialmente con el número de instancias. En la práctica, esto rara vez sucede y k-means suele ser uno de los algoritmos de agrupación en clústeres más rápidos.

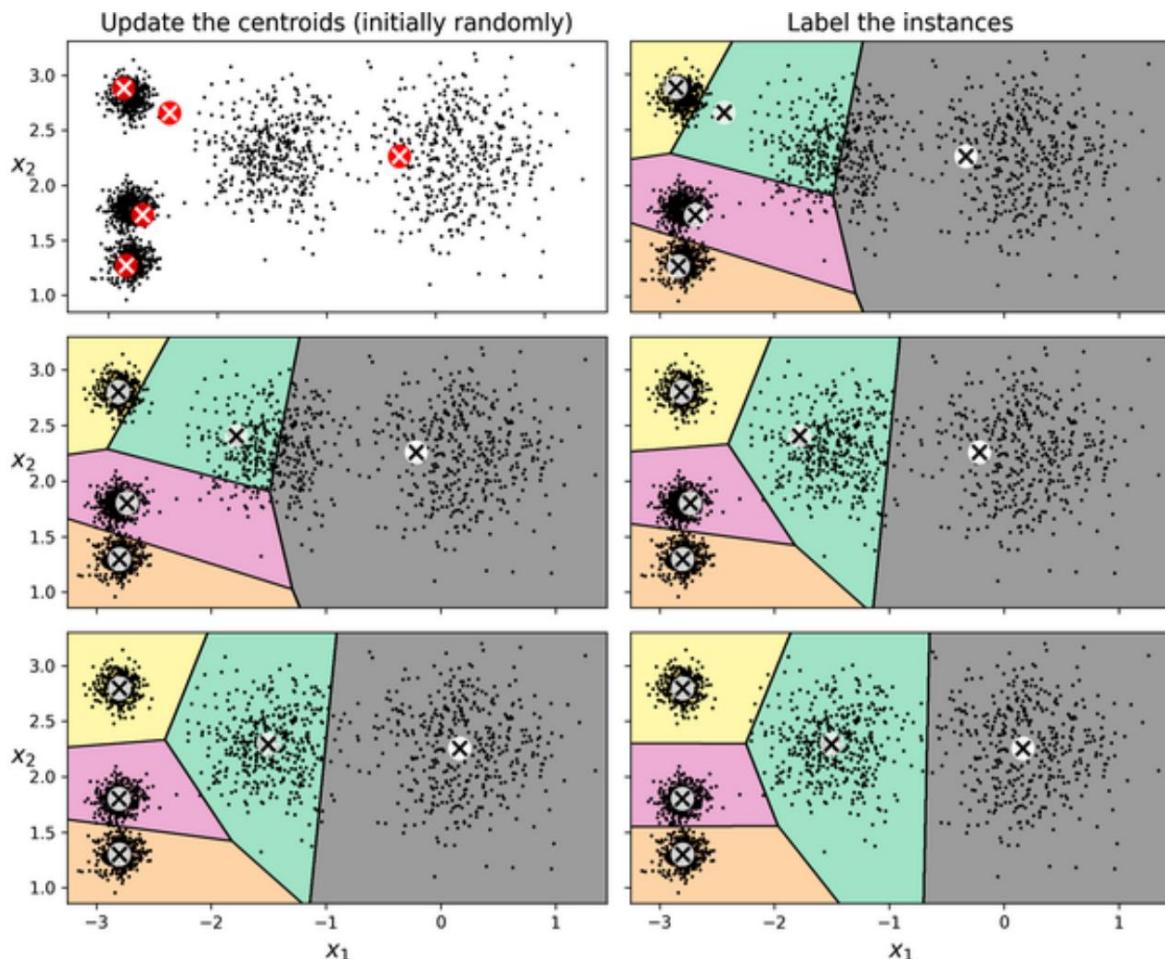


Figura 9-4. El algoritmo k-medias

Aunque se garantiza que el algoritmo convergerá, es posible que no converja a la solución correcta (es decir, puede que converja a un óptimo local): si lo hace o no depende de la inicialización del centroide. **La Figura 9-5** muestra dos soluciones subóptimas que el

El algoritmo puede converger si no tiene suerte con el paso de inicialización aleatoria.

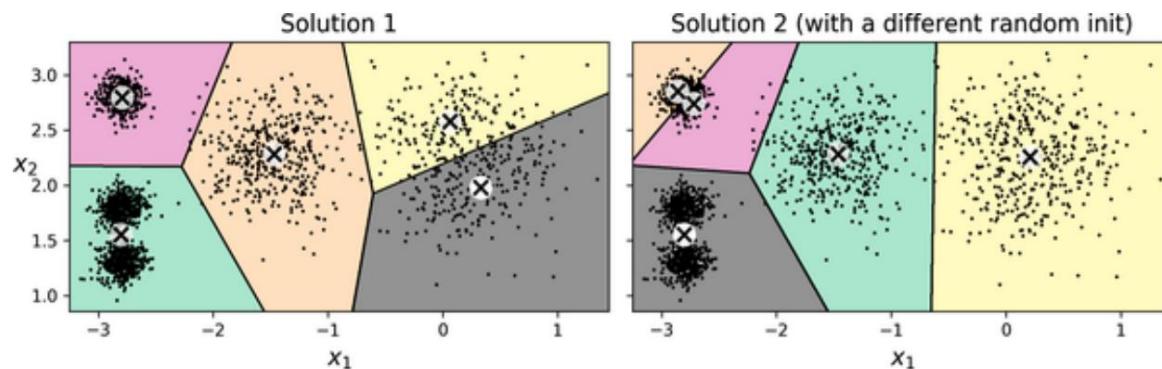


Figura 9-5. Soluciones subóptimas debido a inicializaciones de centroides desafortunadas

Echemos un vistazo a algunas formas en que puede mitigar este riesgo mejorando la inicialización del centroide.

#### Métodos de inicialización de centroide

Si sabe aproximadamente dónde deberían estar los centroides (por ejemplo, si ejecutó otro algoritmo de agrupamiento anteriormente), puede configurar el hiperparámetro `init` en una matriz NumPy que contiene la lista de centroides y establecer `n_init` en 1:

```
good_init = np.array([[-3, 3], [-3, 2], [-3, 1], [-1, 2], [0, 2]]) kmeans =
KMeans(n_clusters=5, init=good_init, n_init=1, random_state=42)
kmeans.fit(X)
```

Otra solución es ejecutar el algoritmo varias veces con diferentes inicializaciones aleatorias y conservar la mejor solución. El número de inicializaciones aleatorias está controlado por el hiperparámetro `n_init`: de forma predeterminada es igual a 10, lo que significa que todo el algoritmo descrito anteriormente se ejecuta 10 veces cuando llamas a `fit()`, y Scikit-Learn mantiene la mejor solución. Pero ¿cómo sabe exactamente cuál es la mejor solución? ¡Utiliza una métrica de rendimiento! Esa métrica se llama **inercia del modelo**, que es la suma de las distancias al cuadrado.

entre las instancias y sus centroides más cercanos. Es aproximadamente igual a 219,4 para el modelo de la izquierda en [la Figura 9-5](#), 258,6 para el modelo de la derecha en [la Figura 9-5](#) y sólo 211,6 para el modelo de [la Figura 9-3](#).

La clase KMeans ejecuta el algoritmo n\_init veces y mantiene el modelo con la inercia más baja. En este ejemplo, se seleccionará el modelo de [la Figura 9-3](#) (a menos que tengamos mucha mala suerte con n\_init inicializaciones aleatorias consecutivas). Si tiene curiosidad, se puede acceder a la inercia de un modelo a través de la variable de instancia inertia\_:

```
>>> kmeans.inertia_
211.59853725816836
```

El método Score() devuelve la inercia negativa (es negativa porque el método Score() de un predictor siempre debe respetar Scikit-Learn la regla de “cuanto más grande, mejor”: si un predictor es mejor que otro, su método score() debería devolver una puntuación mayor):

```
>>> kmeans.puntuación(X)
-211.5985372581684
```

En un [artículo de 2006](#) se propuso una mejora importante del algoritmo k-means, k-means++. Por David Arthur y Sergei Vassilvitskii. Introdujeron un paso de <sup>2</sup> inicialización más inteligente que tiende a seleccionar centroides que están distantes entre sí, y esta mejora hace que sea mucho menos probable que el algoritmo k-means converja a una solución subóptima. El artículo demostró que el cálculo adicional requerido para el paso de inicialización más inteligente vale la pena porque permite reducir drásticamente la cantidad de veces que es necesario ejecutar el algoritmo para encontrar la solución óptima. El algoritmo de inicialización k-means++ funciona así:

1. Tome un centroide  $c^{(1)}$ , elegidos uniformemente al azar de entre conjunto de datos.

2. Tome un nuevo centroide  $c_i^{(i)}$ , eligiendo una instancia  $x$  con  $\frac{1}{2} \sum_{j=1}^n D(x_j^{(i)})^2$ , (i) donde  $D(x_j)$  es el  
probabilidad  $D(x_j^{(i)}) = \frac{1}{\sum_{j=1}^n D(x_j^{(i)})^2}$ , (i) donde  $D(x_j)$  es el  
distancia entre la instancia  $x_j$  y el centroide  $c_i$  más cercano que ya fue elegido. Esta  
distribución de probabilidad garantiza que las instancias más alejadas de los  
centroides ya elegidos tengan muchas más probabilidades de ser seleccionadas  
como centroides.

3. Repita el paso anterior hasta que se hayan elegido todos los  $k$  centroides.

La clase KMeans utiliza este método de inicialización de forma predeterminada.

#### K-medias aceleradas y k-medias de mini lotes

En un [artículo de 2003](#) se propuso otra mejora del algoritmo k-means. por Charles Elkan.

En algunos conjuntos de datos grandes <sup>3</sup> con muchos grupos, el algoritmo se puede  
acelerar evitando muchos cálculos de distancia innecesarios. Elkan logró esto  
explotando la desigualdad del triángulo (es decir, que una línea recta es siempre la  
distancia más corta entre dos puntos) y manteniendo un registro de los límites superior e  
inferior de las distancias entre instancias y centroides. <sup>4</sup>

Sin embargo, el algoritmo de Elkan no siempre acelera el entrenamiento y, a veces, incluso  
puede ralentizarlo significativamente; Depende del conjunto de datos. Aún así, si quieres  
probarlo, configura algoritmo="elkan".

En un [artículo de 2010](#) se propuso otra variante importante del algoritmo k-  
means. por David Sculley. En lugar de utilizar el conjunto de datos completo en cada  
iteración, el algoritmo es capaz de utilizar minilotes, moviendo los centroides  
ligeramente en cada iteración. Esto acelera el algoritmo (normalmente en un factor  
de tres a cuatro) y permite agrupar enormes conjuntos de datos que no caben en la  
memoria.

Scikit-Learn implementa este algoritmo en la clase MiniBatchKMeans, que puedes usar  
igual que la clase KMeans:

```
desde sklearn.cluster importe MiniBatchKMeans
```

```
minibatch_kmeans = MiniBatchKMeans(n_clusters=5,
random_state=42)
minibatch_kmeans.fit(X)
```

Si el conjunto de datos no cabe en la memoria, la opción más sencilla es utilizar la clase memmap, como hicimos para PCA incremental en el [Capítulo 8](#). Alternativamente, puede pasar un mini lote a la vez al método `parcial_fit()`, pero esto requerirá mucho más trabajo, ya que necesitará realizar múltiples inicializaciones y seleccionar la mejor usted mismo.

Aunque el algoritmo k-means de mini lotes es mucho más rápido que el algoritmo k-means normal , su inercia es generalmente ligeramente peor. Puede ver esto en [la Figura 9-6](#): el gráfico de la izquierda compara las inercias de los modelos de k-medias de mini lotes y de k-medias regulares entrenados en el conjunto de datos de cinco blobs anterior utilizando varios números de grupos k. La diferencia entre las dos curvas es pequeña, pero visible. En el gráfico de la derecha, puede ver que las k-means de mini lotes son aproximadamente 3,5 veces más rápidas que las k-means normales en este conjunto de datos.

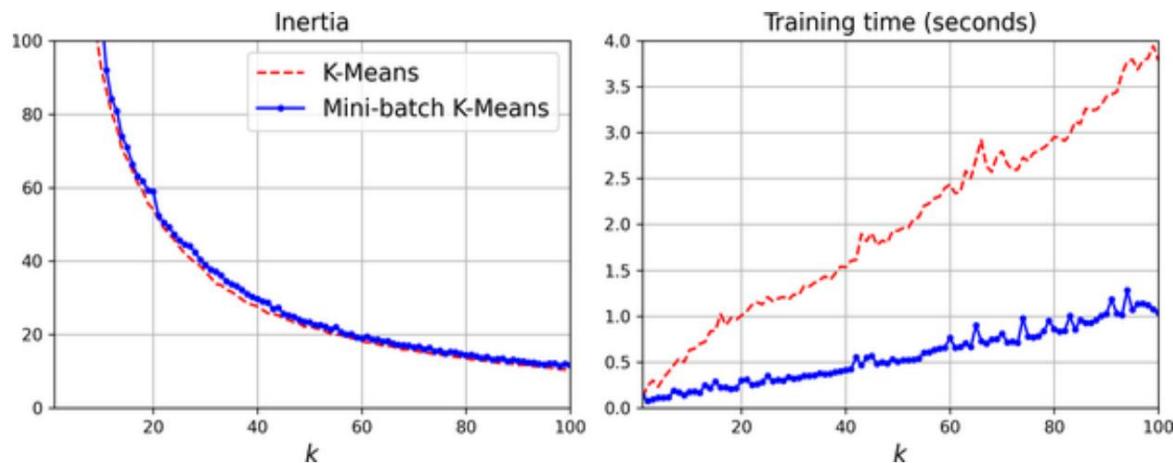


Figura 9-6. Los k-means de mini lotes tienen una inercia mayor que los k-means (izquierda) pero son mucho más rápidos (derecha), especialmente a medida que k aumenta

Encontrar el número óptimo de grupos

Hasta ahora, hemos establecido el número de grupos k en 5 porque, al observar los datos, era obvio que este era el número correcto de

racimos. Pero, en general, no será tan fácil saber cómo establecer  $k$  y el resultado puede ser bastante malo si lo establece en un valor incorrecto. Como puede ver en la Figura 9-7, para este conjunto de datos, establecer  $k$  en 3 u 8 da como resultado modelos bastante malos.

Quizás esté pensando que podría elegir el modelo con la inercia más baja. Lamentablemente, no es tan sencillo. La inercia para  $k=3$  es de aproximadamente 653,2, que es mucho mayor que para  $k=5$  (211,6). Pero con  $k=8$ , la inercia es sólo 119,1. La inercia no es una buena métrica de rendimiento cuando se intenta elegir  $k$  porque sigue disminuyendo a medida que aumentamos  $k$ . De hecho, cuantos más clústeres haya, más cerca estará cada instancia de su centroide más cercano y, por tanto, menor será la inercia. Grafiquemos la inercia en función de  $k$ . Cuando hacemos esto, la curva a menudo contiene un punto de inflexión llamado codo (ver Figura 9-8).

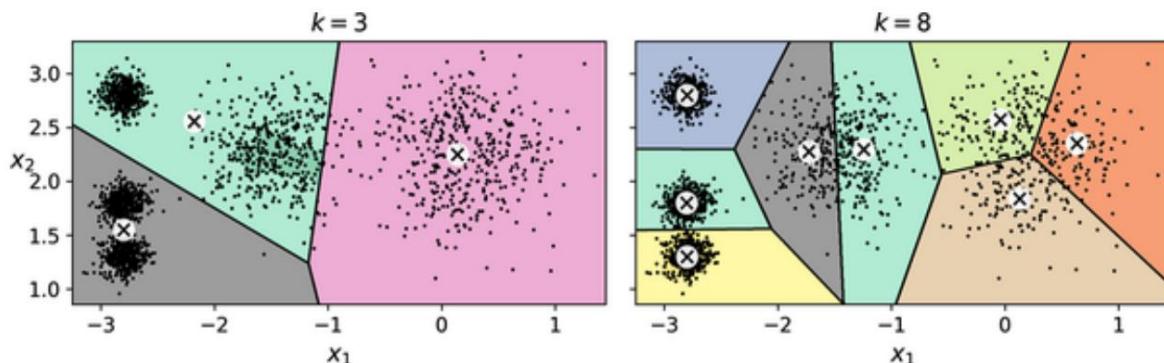


Figura 9-7. Malas elecciones para el número de grupos: cuando  $k$  es demasiado pequeño, los grupos separados se fusionan (izquierda), y cuando  $k$  es demasiado grande, algunos grupos se cortan en varios pedazos (derecha)

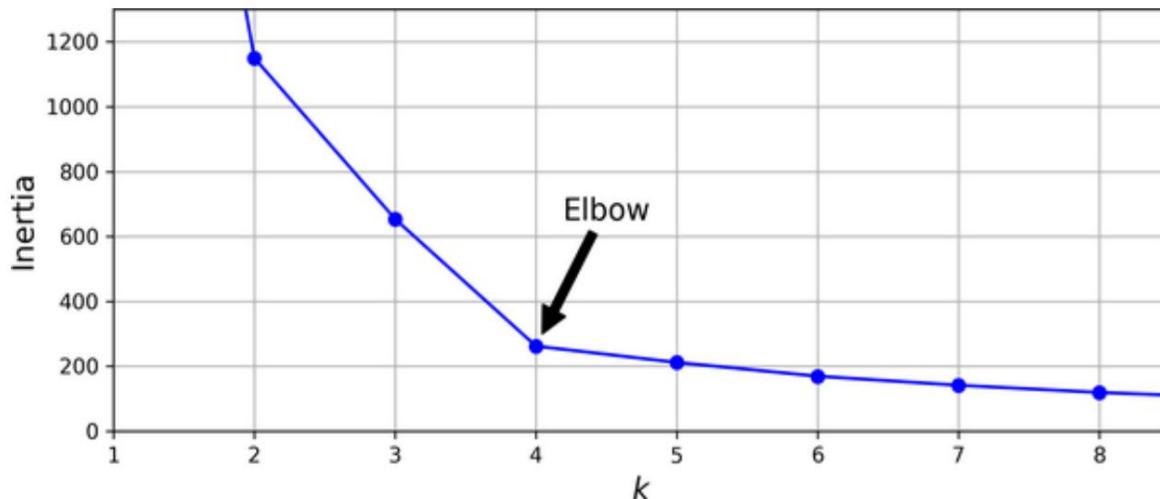


Figura 9-8. Trazar la inercia en función del número de grupos k

Como puede ver, la inercia cae muy rápidamente a medida que aumentamos k hasta 4, pero luego disminuye mucho más lentamente a medida que seguimos aumentando k. Esta curva tiene aproximadamente la forma de un brazo y hay un codo en  $k = 4$ . Entonces, si no supiéramos mejor, podríamos pensar que 4 es una buena elección: cualquier valor más bajo sería dramático, mientras que cualquier valor más alto sería no ayuda mucho, y podríamos estar dividiendo grupos perfectamente buenos por la mitad sin una buena razón.

Esta técnica para elegir el mejor valor para el número de conglomerados es bastante tosca. Un enfoque más preciso (pero también más costoso desde el punto de vista computacional) es utilizar la puntuación de silueta, que es el coeficiente de silueta medio de todas las instancias. El coeficiente de silueta de una instancia es igual a  $(b - a) / \max(a, b)$ , donde a es la distancia media a las otras instancias en el mismo grupo (es decir, la distancia media dentro del grupo) y b es la media más cercana -distancia del clúster (es decir, la distancia media a las instancias del siguiente clúster más cercano, definida como la que minimiza b, excluyendo el propio clúster de la instancia). El coeficiente de silueta puede variar entre  $-1$  y  $+1$ .

Un coeficiente cercano a  $+1$  significa que la instancia está dentro de su propio grupo y lejos de otros grupos, mientras que un coeficiente cercano a  $0$  significa que está cerca del límite de un grupo; finalmente, un coeficiente cercano a  $-1$  significa que es posible que la instancia se haya asignado al clúster incorrecto.

Para calcular la puntuación de la silueta, puede utilizar la función `silueta_score()` de Scikit-Learn, proporcionándole todas las instancias en el conjunto de datos y las etiquetas que se les asignaron:

```
>>> desde sklearn.metrics importar silueta_score >>> silueta_score(X, kmeans.labels_) 0.655517642572828
```

Comparemos las puntuaciones de silueta para diferentes números de conglomerados (consulte la Figura 9-9).

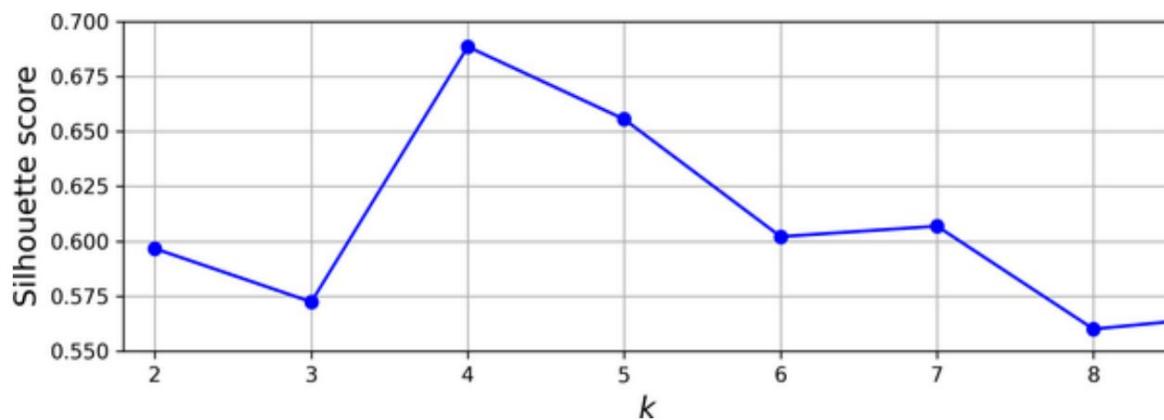


Figura 9-9. Seleccionar el número de grupos k usando la puntuación de silueta

Como puedes ver, esta visualización es mucho más rica que la anterior: aunque confirma que  $k = 4$  es una muy buena elección, también resalta el hecho de que  $k = 5$  también es bastante bueno, y mucho mejor que  $k = 6$  o  $7$ . Esto no fue visible al comparar inercias.

Se obtiene una visualización aún más informativa cuando trazamos el coeficiente de silueta de cada instancia, ordenados por los grupos a los que están asignados y por el valor del coeficiente. Esto se llama diagrama de silueta (ver Figura 9-10). Cada diagrama contiene una forma de cuchillo por grupo. La altura de la forma indica el número de instancias en el clúster y su ancho representa los coeficientes de silueta ordenados de las instancias en el clúster (cuanto más ancho, mejor).

Las líneas discontinuas verticales representan la puntuación media de silueta para cada número de grupos. Cuando la mayoría de las instancias de un clúster

tienen un coeficiente más bajo que esta puntuación (es decir, si muchas de las instancias no llegan a la línea discontinua y termina a la izquierda de ella), entonces el grupo es bastante malo ya que esto significa que sus instancias están demasiado cerca de otros grupos. Aquí podemos ver que cuando  $k = 3$  o  $6$ , obtenemos grupos defectuosos. Pero cuando  $k = 4$  o  $5$ , los grupos se ven bastante bien: la mayoría de los casos se extienden más allá de la línea discontinua, hacia la derecha y más cerca de  $1,0$ . Cuando  $k = 4$ , el grupo en el índice  $1$  (el segundo desde abajo) es bastante grande. Cuando  $k = 5$ , todos los grupos tienen tamaños similares. Entonces, aunque la puntuación general de la silueta de  $k = 4$  es ligeramente mayor que la de  $k = 5$ , parece una buena idea usar  $k = 5$  para obtener grupos de tamaños similares.

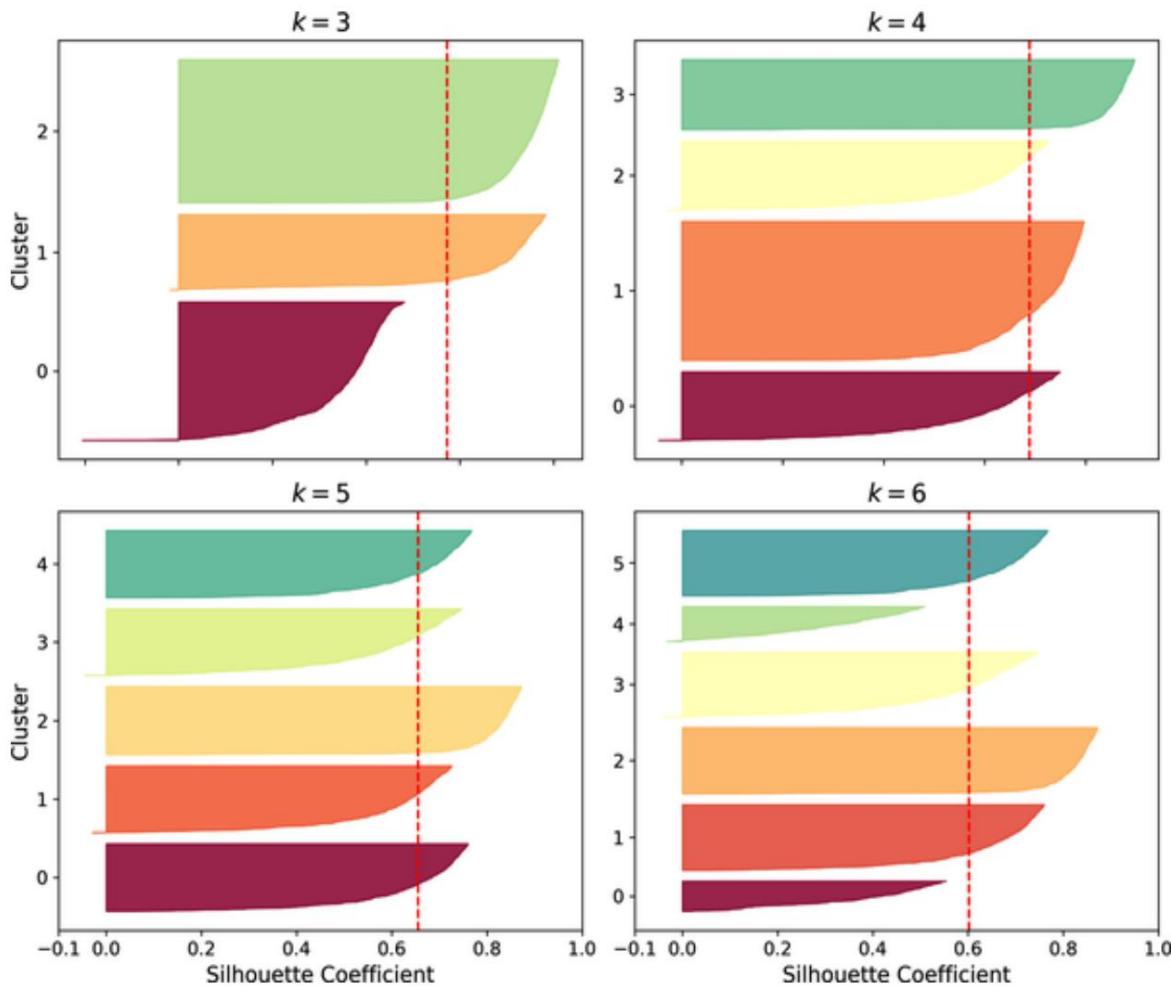


Figura 9-10. Analizando los diagramas de silueta para varios valores de  $k$

## Límites de k-medias

A pesar de sus muchos méritos, entre los que destaca su rapidez y escalabilidad, k-means no es perfecto. Como vimos, es necesario ejecutar el algoritmo varias veces para evitar soluciones subóptimas, además es necesario especificar la cantidad de clústeres, lo que puede ser bastante complicado. Además, k-medias no se comporta muy bien cuando los grupos tienen diferentes tamaños, diferentes densidades o formas no esféricas. Por ejemplo, la Figura 9-11 muestra cómo k-means agrupa un conjunto de datos que contiene tres grupos elipsoidales de diferentes dimensiones, densidades y orientaciones.

Como puede ver, ninguna de estas soluciones es buena. La solución de la izquierda es mejor, pero aún así corta el 25% del grupo del medio y lo asigna al grupo de la derecha. La solución de la derecha es simplemente terrible, aunque su inercia es menor. Entonces, dependiendo de los datos, diferentes algoritmos de agrupación pueden funcionar mejor. En este tipo de grupos elípticos, los modelos de mezcla gaussiana funcionan muy bien.

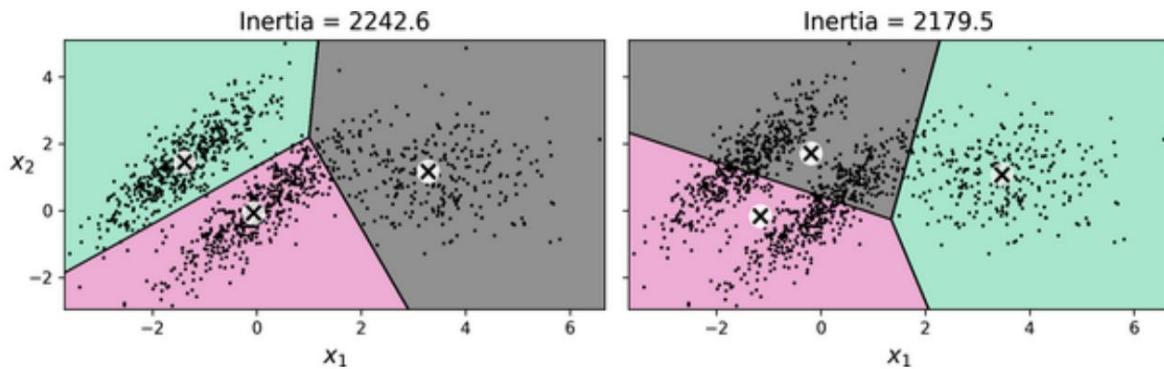


Figura 9-11. k-means no logra agrupar estas manchas elipsoidales correctamente

### CONSEJO

Es importante escalar las características de entrada (consulte [el Capítulo 2](#)) antes de ejecutar k-means, o los grupos pueden estar muy estirados y k-means tendrá un rendimiento deficiente. Escalar las características no garantiza que todos los grupos sean agradables y esféricos, pero generalmente ayuda a k-means.

Ahora veamos algunas formas en las que podemos beneficiarnos de la agrupación. Usaremos k-means, pero siéntete libre de experimentar con otros algoritmos de agrupamiento.

### Uso de agrupaciones en clústeres para la segmentación de imágenes

La segmentación de imágenes es la tarea de dividir una imagen en múltiples segmentos. Hay varias variantes:

- En la segmentación de color, los píxeles con un color similar se asignan al mismo segmento. Esto es suficiente en muchas aplicaciones. Por ejemplo, si desea analizar imágenes de satélite para medir cuánta superficie forestal total hay en una región, la segmentación por colores puede ser adecuada.
- En la segmentación semántica, todos los píxeles que forman parte del mismo tipo de objeto se asignan al mismo segmento. Por ejemplo, en el sistema de visión de un vehículo autónomo, todos los píxeles que forman parte de la imagen de un peatón podrían asignarse al segmento "peatonal" (habría un segmento que contendría a todos los peatones).
- En la segmentación de instancias, todos los píxeles que forman parte del mismo objeto individual se asignan al mismo segmento. En este caso existiría un tramo diferente para cada peatón.

El estado del arte actual en segmentación semántica o de instancias se logra utilizando arquitecturas complejas basadas en redes neuronales convolucionales (ver [Capítulo 14](#)). En este capítulo nos centraremos en la tarea de segmentación de colores (mucho más simple), utilizando k-medias.

Comenzaremos importando el paquete Pillow (sucesor de Python Imaging Library, PIL), que luego usaremos para cargar la imagen ladybug.png (consulte la imagen superior izquierda en [la Figura 9-12](#)), suponiendo que esté ubicada en la ruta del archivo:

```
>>> importar PIL
>>> imagen = np.asarray(PIL.Image.open(ruta de archivo)) >>>
imagen.forma (533,
800, 3)
```

La imagen se representa como una matriz 3D. El tamaño de la primera dimensión es la altura; el segundo es el ancho; y el tercero es el número de canales de color, en este caso rojo, verde y azul (RGB). En otras palabras, para cada píxel hay un vector 3D que contiene las intensidades de rojo, verde y azul como enteros de 8 bits sin signo entre 0 y 255.

Algunas imágenes pueden tener menos canales (como las imágenes en escala de grises, que solo tienen uno) y algunas imágenes pueden tener más canales (como imágenes con un canal alfa adicional para mayor transparencia o imágenes de satélite, que a menudo contienen canales para frecuencias de luz adicionales ( como infrarrojos).

El siguiente código reforma la matriz para obtener una larga lista de colores RGB y luego agrupa estos colores usando k-means con ocho grupos. Crea una matriz segmented\_img que contiene el centro del grupo más cercano para cada píxel (es decir, el color medio del grupo de cada píxel) y, por último, reforma esta matriz a la forma de la imagen original.

La tercera línea utiliza indexación NumPy avanzada; por ejemplo, si las primeras 10 etiquetas en kmeans\_.labels\_ son iguales a 1, entonces los primeros 10 colores en segmented\_img son iguales a kmeans.cluster\_centers\_[1]:

```
X = imagen.reshape(-1, 3)
= KMeans(n_clusters=8, random_state=42).fit(X) segmented_img =
kmeans.cluster_centers_[kmeans.labels_] segmented_img =
segmented_img.reshape(imagen.shape)
```

Esto genera la imagen que se muestra en la parte superior derecha de [la Figura 9-12](#). Puede experimentar con varios números de grupos, como se muestra en la figura. Cuando usas menos de ocho grupos, observa que el llamativo color rojo de la mariquita no logra formar un grupo propio: se fusiona con los colores del entorno. Esto se debe a que k-significa

prefiere grupos de tamaños similares. La mariquita es pequeña, mucho más pequeña que el resto de la imagen, por lo que, aunque su color es llamativo, k- significa no le dedica un grupo.



Figura 9-12. Segmentación de imágenes utilizando k-medias con varios números de grupos de colores

Eso no fue demasiado difícil, ¿verdad? Ahora veamos otra aplicación de la agrupación en clústeres.

### Uso de agrupación en clústeres para el aprendizaje semisupervisado

Otro caso de uso de la agrupación en clústeres es el aprendizaje semisupervisado, cuando tenemos muchas instancias sin etiquetar y muy pocas instancias etiquetadas. En esta sección, usaremos el conjunto de datos de dígitos, que es un conjunto de datos simple similar a MNIST que contiene 1797 imágenes en escala de grises de  $8 \times 8$  que representan los dígitos del 0 al 9. Primero, carguemos y dividamos el conjunto de datos (ya está mezclado):

```
desde sklearn.datasets importar load_digits

X_dígitos, y_dígitos = cargar_dígitos(return_X_y=True)
Tren_X, tren_y = Dígitos_X[:1400], Dígitos_y[:1400]
Prueba_X, prueba_y = X_dígitos[1400:], y_dígitos[1400:]
```

Haremos como que solo tenemos etiquetas para 50 instancias. Para obtener un rendimiento de referencia, entrenemos un modelo de regresión logística en estas 50 instancias etiquetadas:

```
de sklearn.linear_model importar LogisticRegression
```

```
n_labeled = 50
log_reg = LogisticRegression(max_iter=10_000)
log_reg.fit(X_train[:n_labeled], y_train[:n_labeled])
```

Luego podemos medir la precisión de este modelo en el conjunto de prueba (tenga en cuenta que el conjunto de prueba debe estar etiquetado):

```
>>> log_reg.score(prueba_X, prueba_y)
0.7481108312342569
```

La precisión del modelo es sólo del 74,8%. Eso no es genial: de hecho, si intenta entrenar el modelo en el conjunto de entrenamiento completo, encontrará que alcanzará aproximadamente un 90,7% de precisión. Veamos cómo podemos hacerlo mejor. Primero, agrupemos el conjunto de entrenamiento en 50 grupos. Luego, para cada grupo, encontraremos la imagen más cercana al centroide. Llamaremos a estas imágenes las imágenes representativas:

```
k = 50
kmeans = KMeans(n_clusters=k, estado_aleatorio=42)
Dist_dígitos_x = kmeans.fit_transform(tren_x) id_digitRepresentante
= np.argmin(Dist_dígitos_x, eje=0)
X_dígitosRepresentantes = X_tren[idx_dígitosRepresentantes]
```

La Figura 9-13 muestra las 50 imágenes representativas.

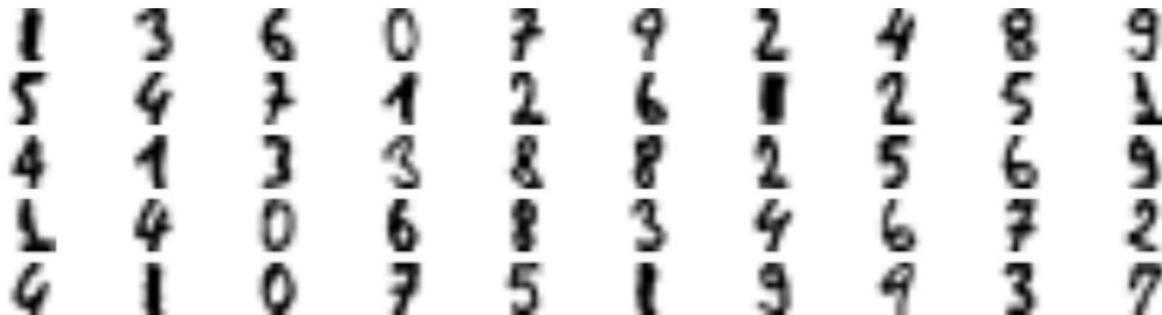


Figura 9-13. Cincuenta imágenes de dígitos representativos (una por grupo)

Miremos cada imagen y etiquetémoslas manualmente:

```
y_dígitos_representantes = np.array([1, 3, 6, 0, 7, 9, 2, ..., 5, 1, 9, 9, 3, 7])
```

Ahora tenemos un conjunto de datos con solo 50 instancias etiquetadas, pero en lugar de ser instancias aleatorias, cada una de ellas es una imagen representativa de su grupo. Veamos si el rendimiento es mejor:

```
>>> log_reg = LogisticRegression(max_iter=10_000) >>>
log_reg.fit(X_digitos_representantes,
y_digitos_representantes) >>>
log_reg.score(X_test, y_test)
0.8488664987405542
```

¡Guau! Saltamos del 74,8% de precisión al 84,9%, aunque todavía solo estamos entrenando el modelo en 50 instancias. Dado que a menudo es costoso y doloroso etiquetar instancias, especialmente cuando los expertos deben hacerlo manualmente, es una buena idea etiquetar instancias representativas en lugar de simplemente instancias aleatorias.

Pero quizás podamos ir un paso más allá: ¿qué pasaría si propagamos las etiquetas a todas las demás instancias del mismo clúster? Esto se llama propagación de etiquetas:

```
y_train_propagated = np.empty(len(X_train), dtype=np.int64) para i en el rango(k):
```

```
y_train_propagated[kmeans.labels_ == i] =
y_dígitos_representantes[i]
```

Ahora entrenemos el modelo nuevamente y observemos su rendimiento:

```
>>> log_reg = LogisticRegression() >>>
log_reg.fit(X_train, y_train_propagated) >>>
log_reg.score(X_test, y_test)
0.8942065491183879
```

¡Obtuvimos otro aumento significativo de precisión! Veamos si podemos hacerlo aún mejor ignorando el 1% de las instancias que están más alejadas del centro del clúster: esto debería eliminar algunos valores atípicos. El siguiente código primero calcula la distancia desde cada instancia hasta el centro de su grupo más cercano, luego, para cada grupo, establece el 1% de las distancias más grandes en -1. Por último, crea un conjunto sin estas instancias marcadas con una distancia - 1:

percentil\_más cercano = 99

```
X_cluster_dist = X_digits_dist[np.arange(len(X_train)), kmeans.labels_] para
i en rango(k):
in_cluster =
    (kmeans.labels_ == i) cluster_dist =
        X_cluster_dist[in_cluster] cutoff_distance =
            np.percentile(cluster_dist, percentil_más cercano) arriba_corte
= (X_cluster_dist >
    cutoff_distance)
    X_cluster_dist[en_cluster y superior_cutoff] = -1

parcialmente propagado = (X_cluster_dist != -1)
X_train_partially_propagated =
X_train[parcialmente_propagado]
y_train_partially_propagated =
y_train_propagated[parcialmente_propagado]
```

Ahora entrenemos el modelo nuevamente en este conjunto de datos parcialmente propagado y veamos qué precisión obtenemos:

```
>>> log_reg = LogisticRegression(max_iter=10_000) >>>
log_reg.fit(X_train_partially_propagated,
y_train_partially_propagated)
```

```
>>> log_reg.score(prueba_X, prueba_y)  
0.9093198992443325
```

¡Lindo! Con solo 50 instancias etiquetadas (¡solo 5 ejemplos por clase en promedio!) obtuvimos una precisión del 90,9 %, que en realidad es ligeramente superior al rendimiento que obtuvimos en el conjunto de datos de dígitos completamente etiquetados (90,7 %). Esto se debe en parte al hecho de que eliminamos algunos valores atípicos y en parte a que las etiquetas propagadas son bastante buenas: su precisión es de aproximadamente el 97,5 %, como muestra el siguiente código:

```
>>> (y_train_partially_propagated ==  
y_train[parcialmente_propagado]).mean()  
0.9755555555555555
```

#### CONSEJO

Scikit-Learn también ofrece dos clases que pueden propagar etiquetas automáticamente: LabelSpreading y LabelPropagation en el paquete `sklearn.semi_supervised`. Ambas clases construyen una matriz de similitud entre todas las instancias y propagan iterativamente etiquetas desde instancias etiquetadas a instancias similares sin etiquetar. También hay una clase muy diferente llamada `SelfTrainingClassifier` en el mismo paquete: le asignas un clasificador base (como `RandomForestClassifier`) y lo entrena en las instancias etiquetadas, luego lo usa para predecir etiquetas para las muestras sin etiquetar. Luego actualiza el conjunto de entrenamiento con las etiquetas en las que tiene más confianza y repite este proceso de entrenamiento y etiquetado hasta que ya no puede agregar etiquetas. Estas técnicas no son soluciones mágicas, pero ocasionalmente pueden darle un pequeño impulso a tu modelo.

## APRENDIZAJE ACTIVO

Para continuar mejorando su modelo y su conjunto de entrenamiento, el siguiente paso podría ser realizar algunas rondas de aprendizaje activo, que es cuando un experto humano interactúa con el algoritmo de aprendizaje, proporcionando etiquetas para instancias específicas cuando el algoritmo las solicita. Existen muchas estrategias diferentes para el aprendizaje activo, pero una de las más comunes se llama muestreo de incertidumbre. Así es como funciona:

1. El modelo se entrena en las instancias etiquetadas recopiladas hasta el momento y este modelo se utiliza para hacer predicciones en todas las instancias sin etiquetar.
2. Los casos en los que el modelo es más incierto (es decir, en los que su probabilidad estimada es más baja) se entregan al experto para que los etiquete.
3. Repita este proceso hasta que la mejora del rendimiento deje de valer el esfuerzo de etiquetado.

Otras estrategias de aprendizaje activo incluyen etiquetar las instancias que darían como resultado el mayor cambio de modelo o la mayor caída en el error de validación del modelo, o las instancias en las que diferentes modelos no están de acuerdo (por ejemplo, una SVM y un bosque aleatorio).

Antes de pasar a los modelos de mezcla gaussiana, echemos un vistazo a DBSCAN, otro algoritmo de agrupamiento popular que ilustra un enfoque muy diferente basado en la estimación de densidad local. Este enfoque permite que el algoritmo identifique grupos de formas arbitrarias.

## DBSCAN

El algoritmo de agrupamiento espacial de aplicaciones con ruido basado en densidad (DBSCAN) define los grupos como regiones continuas de alta densidad. Así es como funciona:

- Para cada instancia, el algoritmo cuenta cuántas instancias se encuentran dentro de una pequeña distancia  $\epsilon$  (épsilon) de ella. Esta región se llama  $\epsilon$ -vecindad de la instancia.
- Si una instancia tiene al menos instancias `min_samples` en su vecindario  $\epsilon$  (incluida ella misma), entonces se considera una instancia central. En otras palabras, las instancias centrales son aquellas que están ubicadas en regiones densas.
- Todas las instancias cercanas a una instancia central pertenecen al mismo clúster. Este vecindario puede incluir otras instancias centrales; por lo tanto, una larga secuencia de instancias centrales vecinas forma un único grupo.
- Cualquier instancia que no sea una instancia central y no tenga ninguna en su vecindario se considera una anomalía.

Este algoritmo funciona bien si todos los grupos están bien separados por regiones de baja densidad. La clase DBSCAN en Scikit-Learn es tan sencilla de usar como cabría esperar. Probémoslo en el conjunto de datos de lunas, presentado en el Capítulo 5:

```
desde sklearn.cluster importar DBSCAN desde
sklearn.datasets importar make_moons
```

```
X, y = make_moons(n_samples=1000, ruido=0.05) dbscan =
DBSCAN(eps=0.05, min_samples=5) dbscan.fit(X)
```

Las etiquetas de todas las instancias ahora están disponibles en la variable de instancia `etiquetas_`:

```
>>> dbscan.labels_
matriz([ 0, 2, -1, -1, 1, 0, 0, 0, 2, 5, [...], 3, 3, 4, 2, 6, 3])
```

Observe que algunas instancias tienen un índice de clúster igual a  $-1$ , lo que significa que el algoritmo las considera anomalías. El

Los índices de las instancias principales están disponibles en la variable de instancia `core_sample_indices_`, y las instancias principales mismas están disponibles en la variable de instancia `components_`:

```
>>> dbSCAN.core_sample_indices_.array([ 0, 4, 5,
7, 995, 997, 998, 999]) >>> dbSCAN.components_
array([-0.02137124, 8, 10, 11, [...], 993,
0.40618608], [-0.84192557,
0.53058695], [...], [0.79419406, 0.60777171]))
```

Esta agrupación se representa en el gráfico de la izquierda de [la Figura 9-14](#). Como puede ver, identificó bastantes anomalías, además de siete grupos diferentes. ¡Que decepcionante! Afortunadamente, si ampliamos la vecindad de cada instancia aumentando los `eps` a 0,2, obtenemos la agrupación de la derecha, que parece perfecta. Sigamos con este modelo.

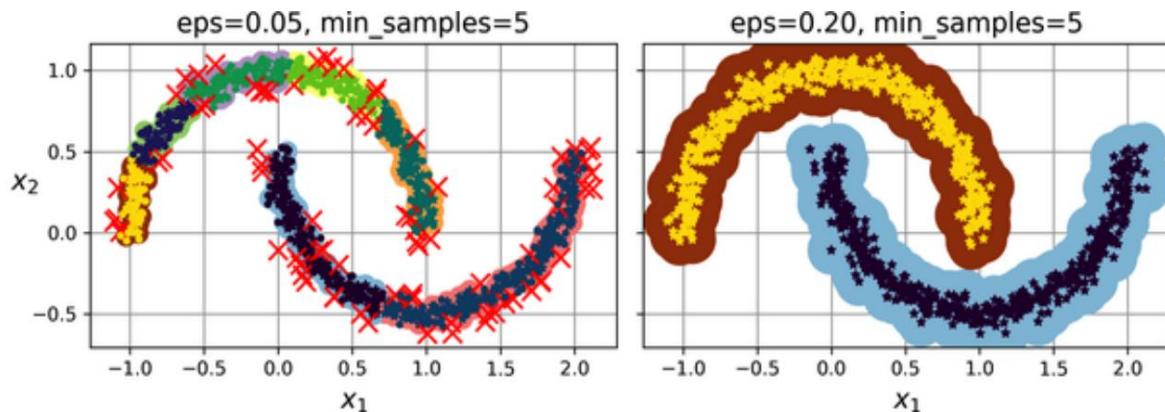


Figura 9-14. Agrupación de DBSCAN utilizando dos radios de vecindad diferentes

Sorprendentemente, la clase DBSCAN no tiene un método `predict()`, aunque tiene un método `fit_predict()`. En otras palabras, no puede predecir a qué clúster pertenece una nueva instancia. Esta decisión se tomó porque diferentes algoritmos de clasificación pueden ser mejores para diferentes tareas, por lo que los autores decidieron dejar que el usuario elija cuál usar. Además, no es difícil de implementar. Por ejemplo, entrenemos un `KNeighborsClassifier`:

```
de sklearn.neighbors importar KNeighborsClassifier

knn = KNeighborsClassifier(n_vecinos=50)
knn.fit(dbSCAN.components_,
        dbSCAN.labels_[dbSCAN.core_sample_indices_])
```

Ahora, dados algunos casos nuevos, podemos predecir a qué grupos es más probable que pertenezcan e incluso estimar una probabilidad para cada grupo:

```
>>> X_nuevo = np.array([-0.5, 0], [0, 0.5], [1, -0.1], [2, 1])) >>> knn.predict(X_nuevo)

array([1, 0, 1, 0]) >>>
knn.predict_proba(X_new)
matriz([[0.18, 0.82], [1.0, 0.12, 0.88],
[1.0, ]])
,
,
,
```

Tenga en cuenta que solo entrenamos al clasificador en las instancias principales, pero también podríamos haber elegido entrenarlo en todas las instancias, o en todas menos en las anomalías: esta elección depende de la tarea final.

El límite de decisión se representa en [la Figura 9-15](#) (las cruces representan las cuatro instancias en X\_new). Tenga en cuenta que, dado que no hay ninguna anomalía en el conjunto de entrenamiento, el clasificador siempre elige un grupo, incluso cuando ese grupo está lejos. Es bastante sencillo introducir una distancia máxima, en cuyo caso las dos instancias que están lejos de ambos grupos se clasifican como anomalías. Para hacer esto, use el método kneighbors() del KNeighborsClassifier. Dado un conjunto de instancias, devuelve las distancias y los índices de los k vecinos más cercanos en el conjunto de entrenamiento (dos matrices, cada una con k columnas):

```
>>> y_dist, y_pred_idx = knn.kneighbors(X_new, n_neighbors=1)
>>> y_pred =
dbSCAN.labels_[dbSCAN.core_sample_indices_]
```

```
[y_pred_idx]
>>> y_pred[y_dist > 0.2] = -1 >>>
y_pred.ravel() matriz([-1,
0, 1, -1])
```

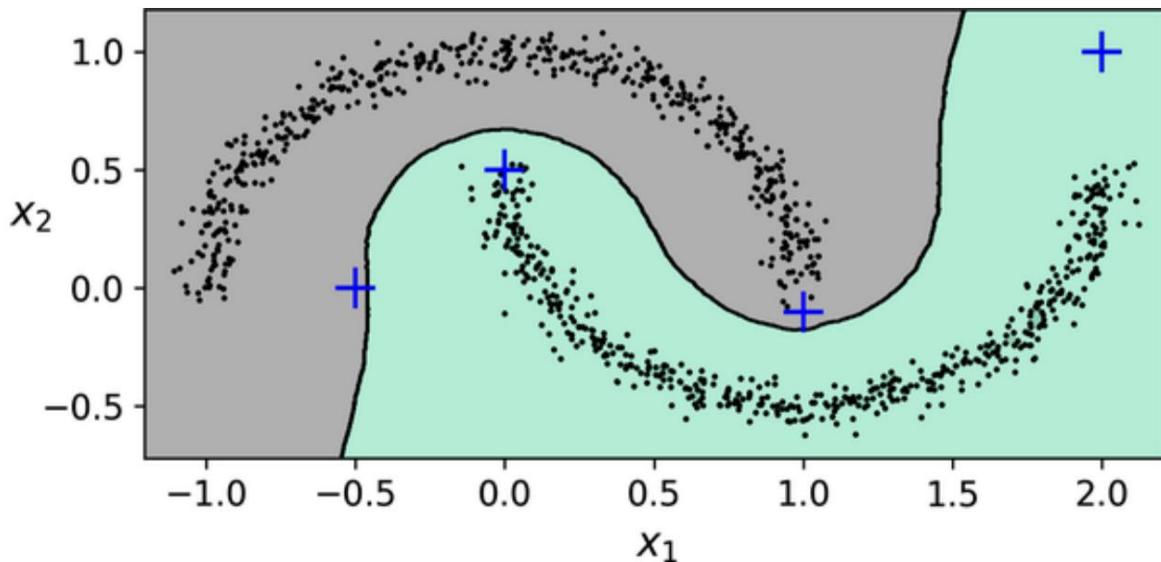


Figura 9-15. Límite de decisión entre dos grupos

En resumen, DBSCAN es un algoritmo muy simple pero poderoso capaz de identificar cualquier número de grupos de cualquier forma. Es resistente a los valores atípicos y tiene solo dos hiperparámetros (`eps` y `min_samples`). Sin embargo, si la densidad varía significativamente entre los grupos, o si no hay una región de densidad suficientemente baja alrededor de algunos grupos, DBSCAN puede tener dificultades para capturar todos los grupos correctamente. Además, su complejidad computacional es aproximadamente  $O(m n)^2$ , por lo que no se adapta bien a grandes conjuntos de datos.

#### CONSEJO

Quizás también quieras probar el DBSCAN jerárquico (HDBSCAN), que se implementa en el [proyecto scikit-learn-contrib](#), ya que suele ser mejor que DBSCAN para encontrar grupos de diferentes densidades.

## Otros algoritmos de agrupación

Scikit-Learn implementa varios algoritmos de agrupación más que deberías analizar. No puedo cubrirlos todos en detalle aquí, pero aquí hay una breve descripción:

### Agrupación aglomerativa

Se construye una jerarquía de grupos de abajo hacia arriba. Piense en muchas burbujas diminutas flotando en el agua y que gradualmente se unen entre sí hasta formar un gran grupo de burbujas. De manera similar, en cada iteración, el clustering aglomerativo conecta el par de clusters más cercano (comenzando con instancias individuales). Si dibujaras un árbol con una rama por cada par de grupos que se fusionaron, obtendrías un árbol binario de grupos, donde las hojas son las instancias individuales. Este enfoque puede capturar grupos de diversas formas; también produce un árbol de conglomerados flexible e informativo en lugar de obligarle a elegir una escala de conglomerado particular, y se puede utilizar con cualquier distancia por pares. Puede escalar bien a una gran cantidad de instancias si proporciona una matriz de conectividad, que es una matriz escasa de  $m \times m$  que indica qué pares de instancias son vecinas (por ejemplo, devuelta por `sklearn.neighbors.kneighbors_graph()`). Sin una matriz de conectividad, el algoritmo no se adapta bien a grandes conjuntos de datos.

### ABEDUL

El algoritmo de agrupación y reducción iterativa equilibrada mediante jerarquías (BIRCH) se diseñó específicamente para conjuntos de datos muy grandes y puede ser más rápido que las k-medias por lotes, con resultados similares, siempre que la cantidad de características no sea demasiado grande (<20). Durante el entrenamiento, crea una estructura de árbol que contiene suficiente información para asignar rápidamente cada nueva instancia a un clúster, sin tener que almacenar todas las instancias en el árbol: este enfoque le permite utilizar memoria limitada mientras maneja enormes conjuntos de datos.

### cambio medio

Este algoritmo comienza colocando un círculo centrado en cada instancia; luego, para cada círculo calcula la media de todos los casos ubicados dentro de él y desplaza el círculo para que quede centrado en la media. A continuación, repite este paso de cambio de media hasta que todos los círculos dejan de moverse (es decir, hasta que cada uno de ellos se centra en la media de las instancias que contiene). El desplazamiento medio desplaza los círculos en la dirección de mayor densidad, hasta que cada uno de ellos haya encontrado un máximo de densidad local. Finalmente, todas las instancias cuyos círculos se han asentado en el mismo lugar (o lo suficientemente cerca) se asignan al mismo grupo. El desplazamiento medio tiene algunas de las mismas características que DBSCAN, como que puede encontrar cualquier número de grupos de cualquier forma, tiene muy pocos hiperparámetros (solo uno: el radio de los círculos, llamado ancho de banda) y se basa en datos locales . estimación de densidad. Pero a diferencia de DBSCAN, el cambio medio tiende a cortar los grupos en pedazos cuando tienen variaciones de densidad interna. Desafortunadamente, su complejidad computacional es  $O(m^2n)$ , por lo que no es adecuado para grandes conjuntos de datos.

### Propagación por afinidad

En este algoritmo, las instancias intercambian mensajes repetidamente entre sí hasta que cada instancia ha elegido otra instancia (o ella misma) para representarla. Estas instancias electas se denominan ejemplares. Cada ejemplar y todas las instancias que lo eligieron forman un grupo. En la política de la vida real, normalmente quieres votar por un candidato cuyas opiniones sean similares a las tuyas, pero también quieres que gane las elecciones, por lo que puedes elegir un candidato con el que no estás completamente de acuerdo, pero que es más popular. .

Normalmente se evalúa la popularidad a través de encuestas. La propagación por afinidad funciona de manera similar y tiende a elegir ejemplares ubicados cerca del centro de los grupos, similar a k-means. Pero a diferencia de k-means, no es necesario seleccionar una cantidad de grupos con anticipación: se determina durante el entrenamiento. Además, la propagación por afinidad puede funcionar bien con grupos de diferentes tamaños.

Lamentablemente, este algoritmo tiene una complejidad computacional de  $O(m^2)$ , por lo que no es adecuado para conjuntos de datos grandes.

### Agrupación espectral

Este algoritmo toma una matriz de similitud entre las instancias y crea una incrustación de baja dimensión a partir de ella (es decir, reduce la dimensionalidad de la matriz), luego usa otro algoritmo de agrupamiento en este espacio de baja dimensión (la implementación de Scikit-Learn usa k-medias). La agrupación espectral puede capturar estructuras de agrupaciones complejas y también se puede utilizar para cortar gráficos (por ejemplo, para identificar grupos de amigos en una red social). No se escala bien a una gran cantidad de instancias y no se comporta bien cuando los clústeres tienen tamaños muy diferentes.

Ahora profundicemos en los modelos de mezcla gaussiana, que se pueden utilizar para la estimación de densidad, agrupación y detección de anomalías.

## Mezclas gaussianas

Un modelo de mezcla gaussiana (GMM) es un modelo probabilístico que supone que las instancias se generaron a partir de una mezcla de varias distribuciones gaussianas cuyos parámetros se desconocen. Todas las instancias generadas a partir de una única distribución gaussiana forman un grupo que normalmente parece un elipsoide. Cada grupo puede tener una forma, tamaño, densidad y orientación elipsoidales diferentes, como en [la Figura 9-11](#). Cuando observa una instancia, sabe que se generó a partir de una de las distribuciones gaussianas, pero no se le dice cuál y no sabe cuáles son los parámetros de estas distribuciones.

Hay varias variantes de GMM. En la variante más simple, implementada en la clase GaussianMixture, es necesario conocer de antemano el número  $k$  de distribuciones gaussianas. El conjunto de datos  $X$  es

Se supone que se generó a través de la siguiente ecuación probabilística.  
proceso:

- Para cada instancia, se elige un grupo al azar entre k  
racimos. La probabilidad de elegir el grupo j es  $\frac{1}{k}$   
Peso de 6 grupos  $\pi_j$ . El índice del cluster elegido para el i  
el ejemplo se observa  $z^{(i)}$
- Si la instancia  $i$  fue asignada al clúster  $j$  (es decir,  $z = j$ ),  
entonces la ubicación  $x^{(i)}$  de esta instancia se muestrea aleatoriamente de  
la distribución gaussiana con media  $\mu_j$  y matriz de covarianza  
 $\Sigma_j$ . Esto se observa  $x^{(i)} \sim \mathcal{N}(\mu_j, \Sigma_j)$ .

Entonces, ¿qué se puede hacer con un modelo así? Bueno, dado el conjunto de datos X,  
normalmente querrás comenzar estimando los pesos  $\pi_j$  y todos los  
parámetros de distribución  $\mu_j$  y  $\Sigma_j$ . Scikit-Learn  
La clase GaussianMixture hace que esto sea súper fácil:

de `sklearn.mixture importar GaussianMixture`

```
gm = Mezcla Gaussiana(n_componentes=3, n_init=10)
gm.fit(X)
```

Veamos los parámetros que estimó el algoritmo:

```
>>> gm.pesos_
array([0.39025715, 0.40007391, 0.20966893])
>>> gm.significas_
array([[ 0.05131611,  0.07521837],
       [-1.40763156,  1.42708225],
       [ 3.39893794,  1.05928897]])
>>> gm.covarianzas_
array([[[ 0.68799922,  0.79606357],
       [ 0.79606357,  1.21236106]],
       [[ 0.63479409,  0.72970799],
       [ 0.72970799,  1.1610351]]])
```

[[ 1.14833585, -0.03256179],  
[-0,03256179, 0,95490931]]])

¡Genial, funcionó bien! De hecho, dos de los tres grupos se generaron con 500 instancias cada uno, mientras que el tercer grupo sólo contiene 250 instancias. Entonces, los verdaderos pesos de los grupos son 0,4, 0,4 y 0,2, respectivamente, y eso es aproximadamente lo que encontró el algoritmo. De manera similar, las medias verdaderas y las matrices de covarianza son bastante cercanas a las encontradas por el algoritmo. ¿Pero cómo? Esta clase se basa en el algoritmo de maximización de expectativas (EM), que tiene muchas similitudes con el algoritmo k-means: también inicializa los parámetros del clúster de forma aleatoria, luego repite dos pasos hasta la convergencia, asignando primero instancias a los clústeres (esto se llama paso de expectativa) y luego actualizar los clústeres (esto se llama paso de maximización). Suena familiar, ¿verdad? En el contexto de la agrupación, se puede pensar en EM como una generalización de k-medias que no solo encuentra los centros  $(\mu_1, \mu_2, \dots, \mu_k)$  de los grupos, sino también su tamaño, forma y orientación ( $\Sigma_1, \Sigma_2, \dots, \Sigma_k$ ), así como sus pesos relativos ( $a_1, a_2, \dots, a_k$ ). Sin embargo, a diferencia de k-means, EM utiliza asignaciones de grupos suaves, no asignaciones duras. Para cada instancia, durante el paso de expectativa, el algoritmo estima la probabilidad de que pertenezca a cada grupo (según los parámetros del grupo actual). Luego, durante el paso de maximización, cada grupo se actualiza utilizando todas las instancias del conjunto de datos, y cada instancia se pondera según la probabilidad estimada de que pertenezca a ese grupo. Estas probabilidades se denominan responsabilidades de los clústeres para las instancias. Durante el paso de maximización, la actualización de cada clúster se verá afectada principalmente por las instancias de las que es mayor responsable.

## ADVERTENCIA

Desafortunadamente, al igual que las k-medias, los mercados emergentes pueden terminar convergiendo hacia niveles pobres. soluciones, por lo que es necesario ejecutarlo varias veces, conservando sólo las mejores solución. Es por eso que configuramos n\_init en 10. Tenga cuidado: de forma predeterminada n\_init se establece en 1.

Puede comprobar si el algoritmo convergió o no y cómo

Se necesitaron muchas iteraciones:

```
>>> gm.convergente_
```

Verdadero

```
>>> gm.n_iter_
```

4

Ahora que tiene una estimación de la ubicación, tamaño, forma, orientación y peso relativo de cada grupo, el modelo puede fácilmente asignar cada instancia al clúster más probable (agrupación dura) o estimar la probabilidad de que pertenezca a un grupo particular (suave agrupamiento). Simplemente use el método predict() para agrupación estricta, o el método predict\_proba() para agrupación suave:

```
>>> gm.predecir(X)
matriz([0, 0, 1, ..., 2, 2, 2])
>>> gm.predict_proba(X).ronda(3)
matriz([[0.977, 0., 0.023],
       [0.983, 0.001, 0.016],
       [0.0, , 1.0, ],
       ...,
       [0.0, 0.0, 1.0, ],
       [0.0, 0.0, 1.0, ],
       [0.0, 0.0, 1.0, ]])
```

Un modelo de mezcla gaussiana es un modelo generativo, lo que significa que puedes muestrear nuevas instancias (tenga en cuenta que están ordenadas por clúster índice):

```
>>> X_nuevo, y_nuevo = gm.sample(6)
>>> X_new
array([-0.86944074, -0.32767626], [ 0.29836051,
    0.28297011], [-2.8014927, -0.09047309],
    [ 3.98203732, 1.49951491], [ 3.81677148,
    0.53095244], [2.84104923, -0.73858639]))
>>> y_nueva matriz([0, 0, 1, 2, 2, 2])
```

También es posible estimar la densidad del modelo en cualquier ubicación determinada. Esto se logra utilizando el método `score_samples()`: para cada instancia dada, este método estima el registro de la función de densidad de probabilidad (PDF) en esa ubicación. Cuanto mayor sea la puntuación, mayor será la densidad:

```
>>> gm.score_samples(X).round(2) matriz([-2.61,
    -3.57, -3.33, ..., -3.51, -4.4
    , -3.81])
```

Si calcula el exponencial de estas puntuaciones, obtendrá el valor del PDF en la ubicación de las instancias dadas. Estas no son probabilidades, sino densidades de probabilidad: pueden tomar cualquier valor positivo, no solo un valor entre 0 y 1. Para estimar la probabilidad de que una instancia caiga dentro de una región particular, tendría que integrar el PDF en esa región. (si lo hace en todo el espacio de posibles ubicaciones de instancias, el resultado será 1).

**La Figura 9-16** muestra las medias de los conglomerados, los límites de decisión (líneas discontinuas) y los contornos de densidad de este modelo.

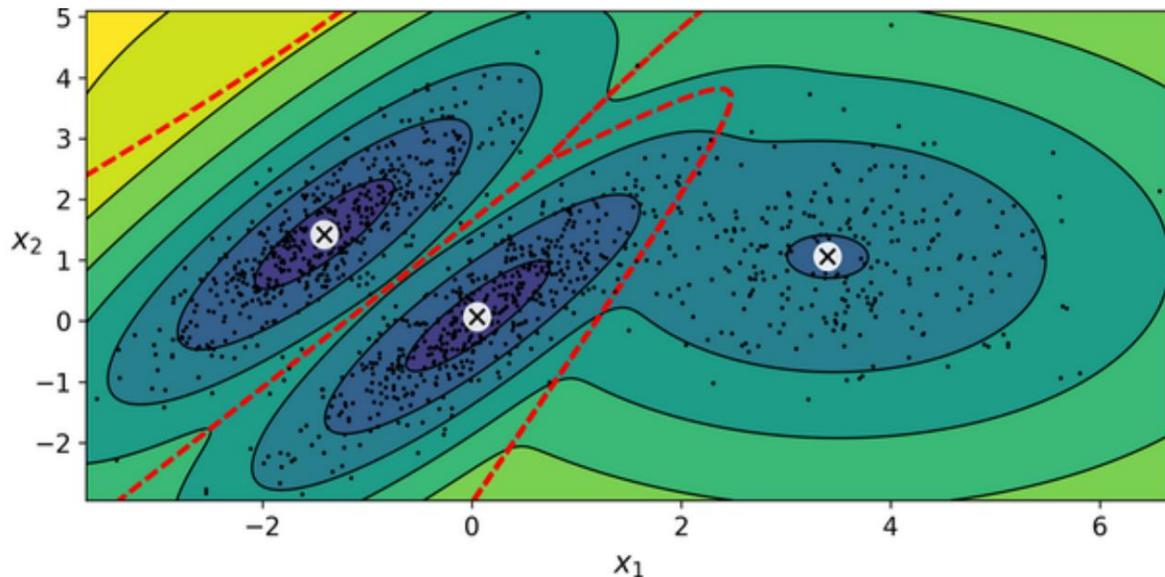


Figura 9-16. Medias de conglomerados, límites de decisión y contornos de densidad de un modelo de mezcla gaussiano entrenado

¡Lindo! El algoritmo claramente encontró una solución excelente. Por supuesto, facilitamos su tarea generando los datos utilizando un conjunto de distribuciones gaussianas 2D (desafortunadamente, los datos de la vida real no siempre son tan gaussianos y de baja dimensión). También le dimos al algoritmo el número correcto de grupos. Cuando hay muchas dimensiones, muchos grupos o pocas instancias, los ME pueden tener dificultades para converger hacia la solución óptima. Es posible que deba reducir la dificultad de la tarea limitando la cantidad de parámetros que el algoritmo debe aprender.

Una forma de hacerlo es limitar la gama de formas y orientaciones que pueden tener los grupos. Esto se puede lograr imponiendo restricciones a las matrices de covarianza. Para hacer esto, establezca el hiperparámetro `covariance_type` en uno de los siguientes valores:

"esférico"

Todos los grupos deben ser esféricos, pero pueden tener diferentes diámetros (es decir, diferentes variaciones).

"diagnóstico"

Los cúmulos pueden adoptar cualquier forma elipsoidal de cualquier tamaño, pero los ejes del elipsoide deben ser paralelos a los ejes de coordenadas (es decir, el

las matrices de covarianza deben ser diagonales).

"atado"

Todos los conglomerados deben tener la misma forma, tamaño y orientación elipsoidales (es decir, todos los conglomerados comparten la misma matriz de covarianza).

De forma predeterminada, covariance\_type es igual a "completo", lo que significa que cada grupo puede adoptar cualquier forma, tamaño y orientación (tiene su propia matriz de covarianza sin restricciones). La Figura 9-17 muestra las soluciones encontradas por el algoritmo EM cuando covariance\_type se establece en "atado" o "esférico".

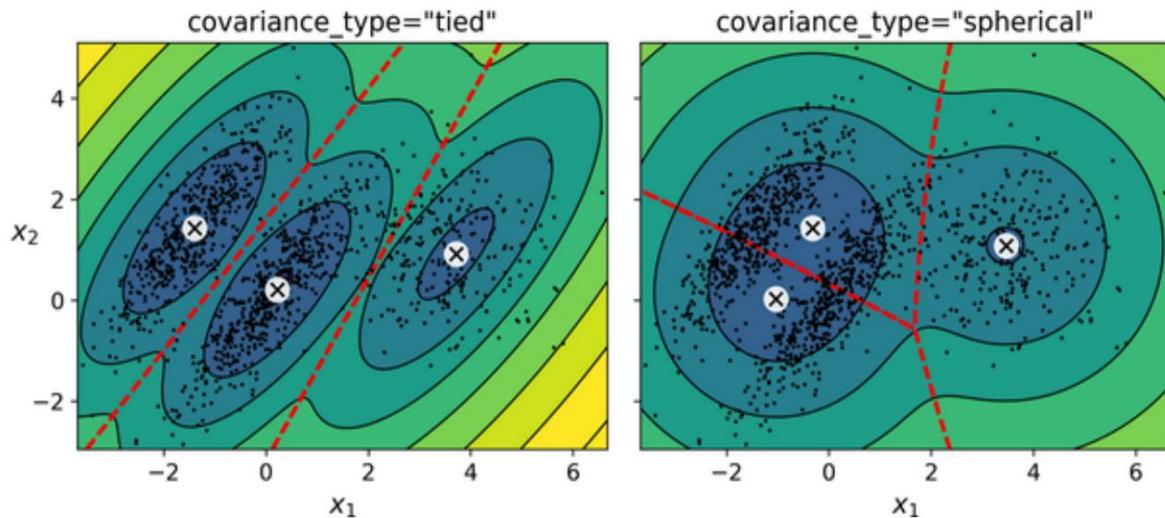


Figura 9-17. Mezclas gaussianas para cúmulos ligados (izquierda) y cúmulos esféricos (derecha)

### NOTA

La complejidad computacional de entrenar un modelo GaussianMixture depende del número de instancias m, el número de dimensiones n, el número de grupos k y las restricciones de las matrices de covarianza. Si covariance\_type es "esférico" o "diag", es  $O(kmn)$ , asumiendo que los datos tienen una estructura de agrupamiento. Si covariance\_type está "vinculado" o "completo", es  $O(kmn + kn^2)$ , por lo que no se escalará a una gran cantidad de funciones.

Los modelos de mezcla gaussiana también se pueden utilizar para la detección de anomalías. Veremos cómo en la siguiente sección.

Uso de mezclas gaussianas para la detección de anomalías Usar un modelo de mezcla gaussiana para la detección de anomalías es bastante simple: cualquier instancia ubicada en una región de baja densidad puede considerarse una anomalía. Debes definir qué umbral de densidad quieras utilizar. Por ejemplo, en una empresa fabricante que intenta detectar productos defectuosos, la proporción de productos defectuosos suele ser bien conocida. Digamos que es igual al 2%. Luego, establece el umbral de densidad para que sea el valor que resulte en tener el 2 % de las instancias ubicadas en áreas por debajo de ese umbral de densidad. Si nota que obtiene demasiados falsos positivos (es decir, productos en perfecto estado que están marcados como defectuosos), puede reducir el umbral. Por el contrario, si tiene demasiados falsos negativos (es decir, productos defectuosos que el sistema no marca como defectuosos), puede aumentar el umbral. Éste es el equilibrio habitual entre precisión y recuperación (consulte [el Capítulo 3](#)). Así es como identificaría los valores atípicos utilizando la densidad más baja del cuarto percentil como umbral (es decir, aproximadamente el 4 % de los casos se marcarán como anomalías):

```
densidades = gm.score_samples(X)
densidad_umbral = np.percentile(densidades, 2) anomalías =
X[densidades < densidad_umbral]
```

[La Figura 9-18](#) representa estas anomalías como estrellas.

Una tarea estrechamente relacionada es la detección de novedades: se diferencia de la detección de anomalías en que se supone que el algoritmo está entrenado en un conjunto de datos "limpio", no contaminado por valores atípicos, mientras que la detección de anomalías no parte de esta suposición. De hecho, la detección de valores atípicos se utiliza a menudo para limpiar un conjunto de datos.

## CONSEJO

Los modelos de mezcla gaussiana intentan ajustar todos los datos, incluidos los valores atípicos; si tiene demasiados, esto sesgará la visión de “normalidad” del modelo, y algunos valores atípicos pueden considerarse erróneamente normales. Si esto sucede, puede intentar ajustar el modelo una vez, usarlo para detectar y eliminar los valores atípicos más extremos y luego ajustar el modelo nuevamente en el conjunto de datos limpio. Otro enfoque es utilizar métodos robustos de estimación de covarianza (consulte la clase `EllipticEnvelope`).

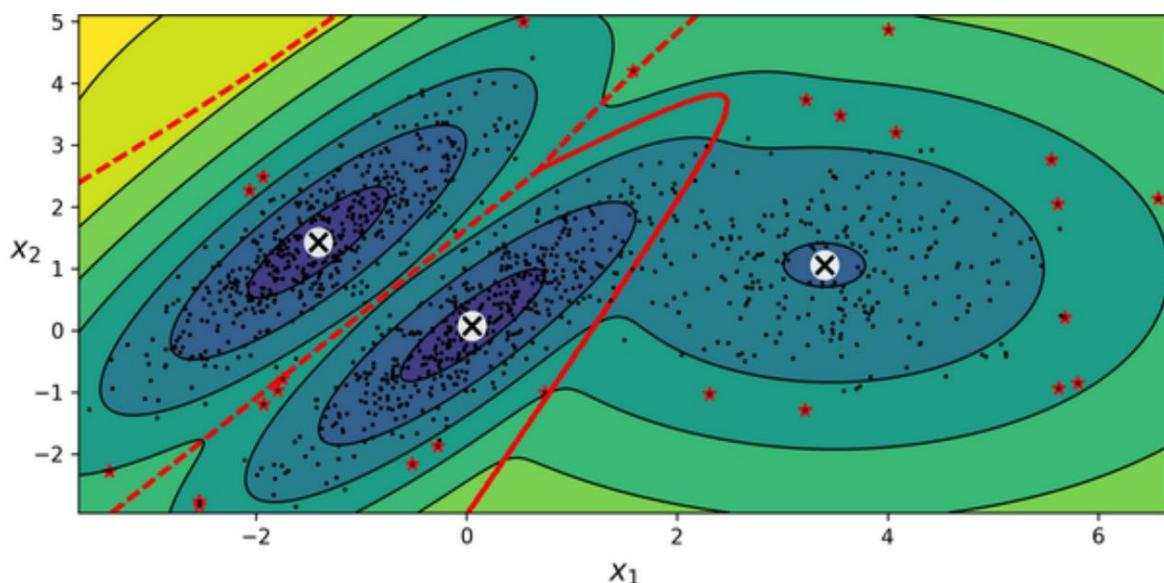


Figura 9-18. Detección de anomalías mediante un modelo de mezcla gaussiana.

Al igual que k-means, el algoritmo GaussianMixture requiere que especifiques el número de grupos. Entonces, ¿cómo puedes encontrar ese número?

## Seleccionar el número de grupos

Con k-medias, puedes utilizar la inercia o la puntuación de silueta para seleccionar el número adecuado de grupos. Pero con las mezclas gaussianas no es posible utilizar estas métricas porque no son confiables cuando los clusters no son esféricos o tienen diferentes tamaños.

En lugar de ello, puedes intentar encontrar el modelo que minimice un valor teórico.

criterio de información, como el criterio de información bayesiano (BIC) o el criterio de información de Akaike (AIC), definido en [la ecuación 9-1](#).

Ecuación 9-1. Criterio de información bayesiano (BIC) y criterio de información de Akaike (AIC)

$$\text{BIC} = \log(m)p - 2 \log(L^*)$$

$$\text{AIC} = 2p - 2 \log(L^*)$$

En estas ecuaciones:

- $m$  es el número de instancias, como siempre.
- $p$  es el número de parámetros aprendidos por el modelo.
- $L^*$  es el valor maximizado de la función de verosimilitud de la modelo.

Tanto el BIC como el AIC penalizan los modelos que tienen más parámetros que aprender (por ejemplo, más grupos) y recompensan los modelos que se ajustan bien a los datos. A menudo acaban seleccionando el mismo modelo. Cuando difieren, el modelo seleccionado por el BIC tiende a ser más simple (menos parámetros) que el seleccionado por el AIC, pero tiende a no ajustarse tan bien a los datos (esto es especialmente cierto para conjuntos de datos más grandes).

## FUNCIÓN DE VEROSIDAD

Los términos “probabilidad” y “verosimilitud” a menudo se usan indistintamente en el lenguaje cotidiano, pero tienen significados muy diferentes en estadística. Dado un modelo estadístico con algunos parámetros  $\theta$ , la palabra "probabilidad" se usa para describir qué tan plausible es un resultado futuro  $x$  (conociendo los valores de los parámetros  $\theta$ ), mientras que la palabra "probabilidad" se usa para describir qué tan plausible es un conjunto particular de parámetros. Los valores  $\theta$  son, después de conocer el resultado  $x$ .

Considere un modelo de mezcla 1D de dos distribuciones gaussianas centradas en  $-4$  y  $+1$ . Para simplificar, este modelo de juguete tiene un único parámetro  $\theta$  que controla las desviaciones estándar de ambas distribuciones. El gráfico de contorno superior izquierdo de [la Figura 9-19](#) muestra el modelo completo  $f(x; \theta)$  en función de  $x$  y  $\theta$ . Para estimar la distribución de probabilidad de un resultado futuro  $x$ , es necesario establecer el parámetro del modelo  $\theta$ . Por ejemplo, si establece  $\theta$  en  $1,3$  (la línea horizontal), obtendrá la función de densidad de probabilidad  $f(x; \theta=1,3)$  que se muestra en el gráfico inferior izquierdo. Supongamos que desea estimar la probabilidad de que  $x$  esté entre  $-2$  y  $+2$ . Debe calcular la integral de la PDF en este rango (es decir, la superficie de la región sombreada). Pero, ¿qué pasa si no sabes  $\theta$  y, en cambio, has observado una única instancia  $x=2,5$  (la línea vertical en el gráfico superior izquierdo)? En este caso, obtienes la función de probabilidad  $(\theta|x=2.5)=f(x=2.5; \theta)$ , representada en el gráfico superior derecho.

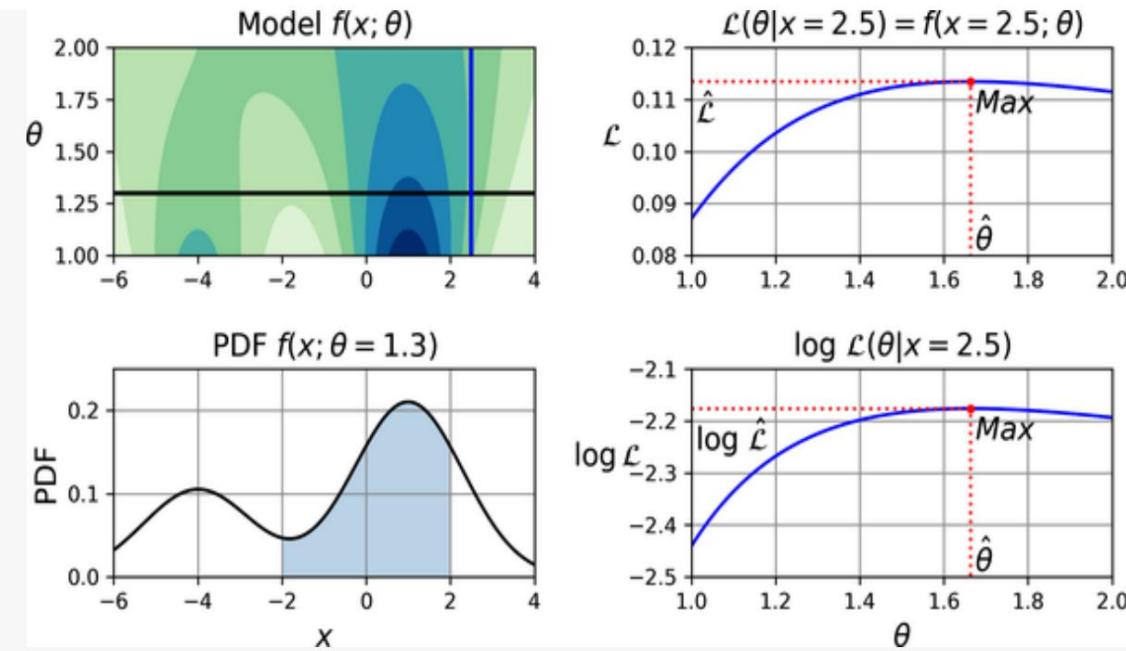


Figura 9-19. La función paramétrica de un modelo (arriba a la izquierda) y algunas funciones derivadas: una PDF (abajo a la izquierda), una función de verosimilitud (arriba a la derecha) y una función logarítmica de verosimilitud (abajo a la derecha)

En resumen, la PDF es una función de  $x$  (con  $\theta$  fijo), mientras que la función de verosimilitud es una función de  $\theta$  (con  $x$  fijo). Es importante entender que la función de probabilidad no es una distribución de probabilidad: si integras una distribución de probabilidad sobre todos los valores posibles de  $x$ , siempre obtienes 1, pero si integras la función de probabilidad sobre todos los valores posibles de  $\theta$  el resultado puede ser cualquier valor positivo.

Dado un conjunto de datos  $X$ , una tarea común es intentar estimar los valores más probables para los parámetros del modelo. Para hacer esto, debe encontrar los valores que maximizan la función de verosimilitud, dado  $X$ . En este ejemplo, si ha observado una sola instancia  $x=2.5$ , la

^  
la estimación de máxima verosimilitud (MLE) de  $\theta$  es  $=1.5$ . Si existe una distribución de probabilidad previa  $g$  sobre  $\theta$ , es posible tenerla en cuenta maximizando  $(\theta|x)g(\theta)$  en lugar de simplemente maximizar  $(\theta|x)$ . Esto se denomina estimación máxima a posteriori (MAP).

Dado que MAP restringe los valores de los parámetros, puede considerarlo como una versión regularizada de MLE.

Observe que maximizar la función de verosimilitud equivale a maximizar su logaritmo (representado en el gráfico inferior derecho de la [Figura 9-19](#)). De hecho, el logaritmo es una función estrictamente creciente, por lo que si  $\theta$  maximiza la probabilidad logarítmica, también maximiza la probabilidad. Resulta que, en general, es más fácil maximizar la probabilidad logarítmica. Por ejemplo, si observara varias (1) ( $m$ ) instancias independientes  $x$  necesitaría encontrar el valor de  $\theta$  que maximiza el producto de las funciones de probabilidad individuales. Pero es equivalente, y mucho más sencillo, maximizar la suma (no el producto) de las funciones logarítmicas de verosimilitud, gracias a la magia del logaritmo que convierte los productos en sumas:  $\log(ab) = \log(a) + \log(b)$ .

Una vez que haya estimado el valor de  $\hat{\theta}$  que maximiza la función de verosimilitud, entonces estará listo para calcular

$\hat{L} = L(\hat{\theta}, X)$ , que es el valor utilizado para calcular el AIC y BIC; Puedes considerarlo como una medida de qué tan bien se ajusta el modelo a los datos.

Para calcular el BIC y el AIC, llame a los métodos `bic()` y `aic()`:

```
>>> gm.bic(X)
8189.747000497186
>>> gm.aic(X)
8102.521720382148
```

[La Figura 9-20](#) muestra el BIC para diferentes números de grupos  $k$ . Como puede ver, tanto el BIC como el AIC son más bajos cuando  $k=3$ , por lo que probablemente sea la mejor opción.

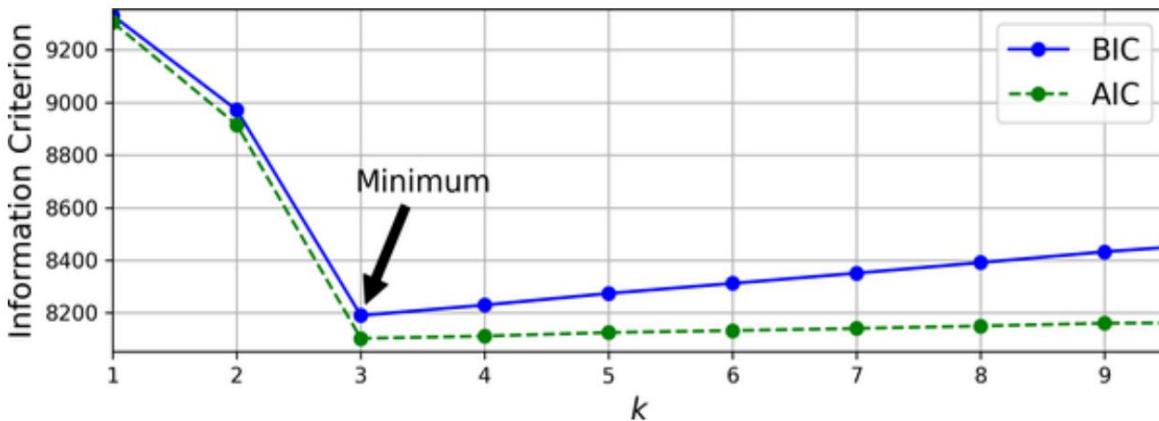


Figura 9-20. AIC y BIC para diferentes números de clusters k

## Modelos de mezcla gaussiana bayesiana

En lugar de buscar manualmente el número óptimo de clústeres, puedes usar la clase BayesianGaussianMixture, que es capaz de dar pesos iguales (o cercanos) a cero a innecesarios racimos. Establezca el número de clústeres `n_components` en un valor que tiene buenas razones para creer que es mayor que el número óptimo de conglomerados (esto supone un conocimiento mínimo sobre la problema en cuestión), y el algoritmo eliminará los innecesarios agrupa automáticamente. Por ejemplo, establezcamos el número de grupos. al 10 y mira que pasa:

```
>>> desde sklearn.mixture importar BayesianGaussianMixture  
>>> bgm = Mezcla Bayesiana-Gaussiana(n_componentes=10,  
n_init=10, estado_aleatorio=42)  
>>> bgm.fit(X)  
>>> bgm.pesos_.redondo(2)  
matrix([0.4, 0.21, 0.4 0.]) , 0. , 0. , 0. , 0. , 0. , 0. , 0.  
,
```

Perfecto: el algoritmo detectó automáticamente que solo tres clusters son necesarios, y los grupos resultantes son casi idénticos a los de la Figura 9-16.

Una nota final sobre los modelos de mezclas gaussianas: aunque funcionan excelentes en grupos con formas elipsoidales, no les va tan bien con

grupos de formas muy diferentes. Por ejemplo, veamos qué sucede si utilizamos un modelo de mezcla bayesiana gaussiana para agrupar el conjunto de datos de las lunas (consulte la Figura 9-21).

¡Ups! El algoritmo buscó desesperadamente elipsoides, por lo que encontró ocho grupos diferentes en lugar de dos. La estimación de la densidad no es tan mala, por lo que este modelo quizás podría usarse para la detección de anomalías, pero no logró identificar las dos lunas. Para concluir este capítulo, echemos un vistazo rápido a algunos algoritmos capaces de tratar con clústeres de formas arbitrarias.

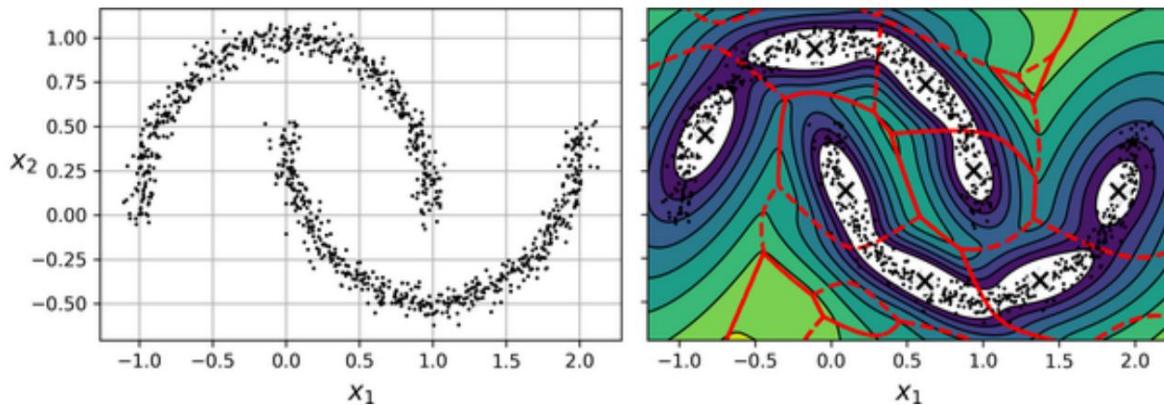


Figura 9-21. Ajuste de una mezcla gaussiana a grupos no elipsoidales

## Otros algoritmos para anomalías y novedades Detección

Scikit-Learn implementa otros algoritmos dedicados a la detección de anomalías o de novedades:

Fast-MCD (determinante de covarianza mínima)

Implementado por la clase `EllipticEnvelope`, este algoritmo es útil para la detección de valores atípicos, en particular para limpiar un conjunto de datos. Se supone que las instancias normales (*inliers*) se generan a partir de una única distribución gaussiana (no una mezcla). También supone que el conjunto de datos está contaminado con valores atípicos que no se generaron a partir de esta distribución gaussiana. Cuando el algoritmo estima los parámetros de la distribución gaussiana (es decir, la forma de la

envolvente elíptica alrededor de los valores interiores), se tiene cuidado de ignorar las instancias que probablemente sean valores atípicos. Esta técnica proporciona una mejor estimación de la envolvente elíptica y, por tanto, hace que el algoritmo identifique mejor los valores atípicos.

#### bosque de aislamiento

Este es un algoritmo eficaz para la detección de valores atípicos, especialmente en conjuntos de datos de alta dimensión. El algoritmo construye un bosque aleatorio en el que cada árbol de decisión crece aleatoriamente: en cada nodo, selecciona una característica aleatoriamente, luego elige un valor umbral aleatorio (entre los valores mínimo y máximo) para dividir el conjunto de datos en dos. El conjunto de datos se va cortando gradualmente en pedazos de esta manera, hasta que todas las instancias terminan aisladas de las demás. Las anomalías suelen estar alejadas de otras instancias, por lo que, en promedio (en todos los árboles de decisión), tienden a aislarse en menos pasos que las instancias normales.

#### Factor de valores atípicos locales (LOF)

Este algoritmo también es bueno para la detección de valores atípicos. Compara la densidad de instancias alrededor de una instancia determinada con la densidad alrededor de sus vecinas. Una anomalía suele estar más aislada que sus k vecinos más cercanos.

#### SVM de una clase

Este algoritmo es más adecuado para la detección de novedades. Recuerde que un clasificador SVM kernelizado separa dos clases asignando primero (implícitamente) todas las instancias a un espacio de alta dimensión y luego separando las dos clases usando un clasificador SVM lineal dentro de este espacio de alta dimensión (consulte el Capítulo 5). Como solo tenemos una clase de instancias, el algoritmo SVM de una clase intenta separar las instancias en un espacio de alta dimensión desde el origen. En el espacio original esto corresponderá a encontrar una pequeña región que englobe todas las instancias. si un

La nueva instancia no cae dentro de esta región, es una anomalía.

Hay algunos hiperparámetros que modificar: los habituales para una SVM kernelizada, más un hiperparámetro de margen que corresponde a la probabilidad de que una nueva instancia se considere erróneamente novedosa cuando en realidad es normal. Funciona muy bien, especialmente con conjuntos de datos de alta dimensión, pero como todas las SVM, no se escala a conjuntos de datos grandes.

PCA y otras técnicas de reducción de dimensionalidad con un método  
`inverse_transform()`

Si se compara el error de reconstrucción de una instancia normal con el error de reconstrucción de una anomalía, este último normalmente será mucho mayor. Este es un enfoque de detección de anomalías simple y, a menudo, bastante eficiente (consulte los ejercicios de este capítulo para ver un ejemplo).

## Ejercicios

1. ¿Cómo definirías la agrupación? ¿Puedes nombrar algunos algoritmos de agrupamiento?
2. ¿Cuáles son algunas de las principales aplicaciones de los algoritmos de agrupamiento?
3. Describe dos técnicas para seleccionar el número correcto de clusters. cuando se utilizan k-medias.
4. ¿Qué es la propagación de etiquetas? ¿Por qué lo implementarías y ¿cómo?
5. ¿Puedes nombrar dos algoritmos de agrupamiento que puedan escalar a grandes conjuntos de datos? ¿Y dos que buscan regiones de alta densidad?
6. ¿Se te ocurre un caso de uso en el que el aprendizaje activo sería útil? ¿Cómo lo implementarías?

7. ¿Cuál es la diferencia entre detección de anomalías y novedad?  
¿detección?
8. ¿Qué es una mezcla gaussiana? ¿Para qué tareas puedes usarlo?
9. ¿Puedes nombrar dos técnicas para encontrar el número correcto de grupos cuando se utiliza un modelo de mezcla gaussiana?
10. El conjunto de datos de caras clásicas de Olivetti contiene 400 escalas de grises de 64 × Imágenes de rostros de 64 píxeles. Cada imagen se aplana a un vector 1D de tamaño 4096. Se fotografiaron cuarenta personas diferentes (diez veces cada una) y la tarea habitual es entrenar un modelo que pueda predecir qué persona está representada en cada imagen. Cargue el conjunto de datos usando la función `sklearn.datasets.fetch_olivetti_faces()`, luego divídalo en un conjunto de entrenamiento, un conjunto de validación y un conjunto de prueba (tenga en cuenta que el conjunto de datos ya está escalado entre 0 y 1). Dado que el conjunto de datos es bastante pequeño, probablemente desee utilizar un muestreo estratificado para asegurarse de que haya la misma cantidad de imágenes por persona en cada conjunto. A continuación, agrupe las imágenes utilizando k-means y asegúrese de tener una buena cantidad de grupos (utilizando una de las técnicas analizadas en este capítulo). Visualiza los grupos: ¿ves caras similares en cada grupo?
11. Continuando con el conjunto de datos de caras de Olivetti, entrene un clasificador para predecir qué persona está representada en cada imagen y evalúelo en el conjunto de validación. A continuación, utilice k-means como herramienta de reducción de dimensionalidad y entrene un clasificador en el conjunto reducido. Busque la cantidad de clusters que permita al clasificador obtener el mejor rendimiento: ¿qué rendimiento puede alcanzar? ¿Qué sucede si agrega las funciones del conjunto reducido a las funciones originales (nuevamente, buscando la mejor cantidad de grupos)?
12. Entrene un modelo de mezcla gaussiana en el conjunto de datos de caras de Olivetti. A acelerar el algoritmo, probablemente deberías reducir el

dimensionalidad del conjunto de datos (por ejemplo, utilizar PCA, preservando el 99% de la varianza). Use el modelo para generar algunas caras nuevas (usando el método `sample()`) y visualícelas (si usó PCA, necesitará usar su método `inverse_transform()`). Intente modificar algunas imágenes (por ejemplo, rotarlas, voltearlas, oscurecerlas) y ver si el modelo puede detectar las anomalías (es decir, comparar la salida del método `score_samples()` para imágenes normales y para anomalías).

13. También se pueden utilizar algunas técnicas de reducción de dimensionalidad para la detección de anomalías. Por ejemplo, tome el conjunto de datos de caras de Olivetti y redúzcalo con PCA, preservando el 99% de la varianza. Luego calcule el error de reconstrucción para cada imagen. A continuación, tome algunas de las imágenes modificadas que creó en el ejercicio anterior y observe su error de reconstrucción: observe cuánto más grandes son. Si trazas una imagen reconstruida, verás por qué: intenta reconstruir una cara normal.

Las soluciones a estos ejercicios están disponibles al final del cuaderno de este capítulo, en <https://homl.info/colab3>.

---

<sup>1</sup> Stuart P. Lloyd, "Cuantización de mínimos cuadrados en PCM", IEEE Transactions on Information Theory 28, no. 2 (1982): 129-137.

<sup>2</sup> David Arthur y Sergei Vassilvitskii, "k-Means++: Las ventajas de Careful Seeding", Actas del 18º Simposio anual ACM-SIAM sobre algoritmos discretos (2007): 1027–1035.

<sup>3</sup> Charles Elkan, "Using the Triangle Inequality to Accelerate k-Means", Actas de la XX Conferencia Internacional sobre Aprendizaje Automático (2003): 147–153.

<sup>4</sup> La desigualdad del triángulo es  $AC \leq AB + BC$ , donde A, B y C son tres puntos y AB, AC y BC son las distancias entre estos puntos.

<sup>5</sup> David Sculley, "Web-Scale K-Means Clustering", Actas de la XIX Conferencia Internacional sobre la World Wide Web (2010): 1177–1178.

<sup>6</sup> Phi (Φ o φ) es la letra número 21 del alfabeto griego.

## Parte II. Redes neuronales y aprendizaje profundo

---

# Capítulo 10. Introducción a las redes neuronales artificiales con Keras

---

Los pájaros nos inspiraron a volar, las plantas de bardana inspiraron el velcro y la naturaleza ha inspirado innumerables inventos más. Parece lógico, entonces, mirar la arquitectura del cerebro en busca de inspiración sobre cómo construir una máquina inteligente. Esta es la lógica que generó las redes neuronales artificiales (RNA), modelos de aprendizaje automático inspirados en las redes de neuronas biológicas que se encuentran en nuestro cerebro. Sin embargo, aunque los aviones se inspiraron en las aves, no necesitan batir las alas para volar.

De manera similar, las RNA se han vuelto gradualmente bastante diferentes de sus primas biológicas. Algunos investigadores incluso sostienen que deberíamos abandonar por completo la analogía biológica (por ejemplo, decir “unidades” en lugar de “neuronas”), para no restringir nuestra creatividad a sistemas biológicamente plausibles.<sup>1</sup>

Las RNA son el núcleo mismo del aprendizaje profundo. Son versátiles, potentes y escalables, lo que los hace ideales para abordar tareas de aprendizaje automático grandes y altamente complejas, como clasificar miles de millones de imágenes (por ejemplo, Google Imágenes), potenciar servicios de reconocimiento de voz (por ejemplo, Siri de Apple), recomendar los mejores videos para mirar a cientos de millones de usuarios todos los días (por ejemplo, YouTube) o aprender a vencer al campeón mundial en el juego de Go (AlphaGo de DeepMind).

La primera parte de este capítulo presenta las redes neuronales artificiales, comenzando con un recorrido rápido por las primeras arquitecturas de ANN y llegando a los perceptrones multicapa, que se utilizan mucho en la actualidad (se explorarán otras arquitecturas en los próximos capítulos). En la segunda parte, veremos cómo implementar redes neuronales usando

API Keras de TensorFlow. Esta es una API de alto nivel simple y bellamente diseñada para construir, entrenar, evaluar y ejecutar redes neuronales. Pero no se deje engañar por su simplicidad: es lo suficientemente expresivo y flexible como para permitirle construir una amplia variedad de arquitecturas de redes neuronales. De hecho, probablemente será suficiente para la mayoría de sus casos de uso. Y si alguna vez necesita flexibilidad adicional, siempre puede escribir componentes Keras personalizados usando su API de nivel inferior, o incluso usar TensorFlow directamente, como verá en el [Capítulo 12](#).

Pero primero, retrocedamos en el tiempo para ver cómo surgieron las redes neuronales artificiales.

**De las neuronas biológicas a las artificiales** Sorprendentemente, las RNA existen desde hace bastante tiempo: fueron introducidas por primera vez en 1943 por el neurofisiólogo Warren McCulloch y el matemático Walter Pitts. En su [artículo histórico](#) "Un cálculo lógico de ideas inmanentes en la actividad nerviosa<sup>2</sup>", McCulloch y Pitts presentaron un modelo computacional simplificado de cómo las neuronas biológicas podrían trabajar juntas en cerebros animales para realizar cálculos complejos utilizando lógica proposicional . Esta fue la primera arquitectura de red neuronal artificial. Desde entonces se han inventado muchas otras arquitecturas, como verás.

Los primeros éxitos de las RNA llevaron a la creencia generalizada de que pronto estaríamos conversando con máquinas verdaderamente inteligentes. Cuando en la década de 1960 quedó claro que esta promesa no se cumpliría (al menos durante bastante tiempo), la financiación se fue a otra parte y las RNA entraron en un largo invierno. A principios de la década de 1980, se inventaron nuevas arquitecturas y se desarrollaron mejores técnicas de entrenamiento, lo que provocó un resurgimiento del interés en el conexiónismo, el estudio de las redes neuronales. Pero el progreso fue lento y en la década de 1990 se habían inventado otras poderosas técnicas de aprendizaje automático, como las máquinas de vectores de soporte (ver [Capítulo 5](#)). Estas técnicas parecieron ofrecer mejores resultados.

y fundamentos teóricos más sólidos que las RNA, por lo que una vez más el estudio de las redes neuronales quedó en suspenso.

Ahora somos testigos de otra ola de interés en las RNA. ¿Esta ola se extinguirá como lo hicieron las anteriores? Bueno, aquí hay algunas buenas razones para creer que esta vez es diferente y que el renovado interés en las RNA tendrá un impacto mucho más profundo en nuestras vidas:

- Actualmente existe una enorme cantidad de datos disponibles para entrenar redes neuronales y las RNA frecuentemente superan a otras técnicas de ML en problemas muy grandes y complejos.
- El tremendo aumento de la potencia informática desde la década de 1990 hace posible ahora entrenar grandes redes neuronales en un período de tiempo razonable. Esto se debe en parte a la ley de Moore (el número de componentes en los circuitos integrados se ha duplicado aproximadamente cada dos años durante los últimos 50 años), pero también gracias a la industria del juego, que ha estimulado la producción de millones de potentes tarjetas GPU. Además, las plataformas en la nube han hecho que este poder sea accesible para todos.
- Se han mejorado los algoritmos de entrenamiento. Para ser justos, son sólo ligeramente diferentes de los utilizados en la década de 1990, pero estos ajustes relativamente pequeños han tenido un enorme impacto positivo.
- Algunas limitaciones teóricas de las RNA han resultado benignas en la práctica. Por ejemplo, mucha gente pensaba que los algoritmos de entrenamiento de RNA estaban condenados porque era probable que se quedaran atrapados en los óptimos locales, pero resulta que esto no es un gran problema en la práctica, especialmente para redes neuronales más grandes: los óptimos locales a menudo funcionan casi como así como el óptimo global.
- Las RNA parecen haber entrado en un círculo virtuoso de financiación y progreso. Productos sorprendentes basados en RNA aparecen regularmente en los titulares de las noticias, lo que atrae cada vez más atención y financiación.

hacia ellos, lo que resulta en más y más progreso y productos aún más sorprendentes.

### Neuronas biológicas Antes de

hablar de las neuronas artificiales, echemos un vistazo rápido a una neurona biológica (representada en [la figura 10-1](#)). Es una célula de aspecto inusual que se encuentra principalmente en cerebros de animales.

Está compuesto por un cuerpo celular que contiene el núcleo y la mayoría de los componentes complejos de la célula, muchas extensiones ramificadas llamadas dendritas, más una extensión muy larga llamada axón. La longitud del axón puede ser sólo unas pocas veces más larga que la del cuerpo celular, o hasta decenas de miles de veces más larga. Cerca de su extremo, el axón se divide en muchas ramas llamadas telodendrias, y en la punta de estas ramas hay estructuras minúsculas llamadas terminales sinápticas (o simplemente sinapsis), que están conectadas a las

dendritas <sup>3</sup> o cuerpos celulares de otras neuronas. Las neuronas biológicas producen impulsos eléctricos cortos llamados potenciales de acción (AP, o simplemente señales), que viajan a lo largo de los axones y hacen que las sinapsis liberen señales químicas llamadas neurotransmisores.

Cuando una neurona recibe una cantidad suficiente de estos neurotransmisores en unos pocos milisegundos, dispara sus propios impulsos eléctricos (en realidad, depende de los neurotransmisores, ya que algunos de ellos inhiben la ac-

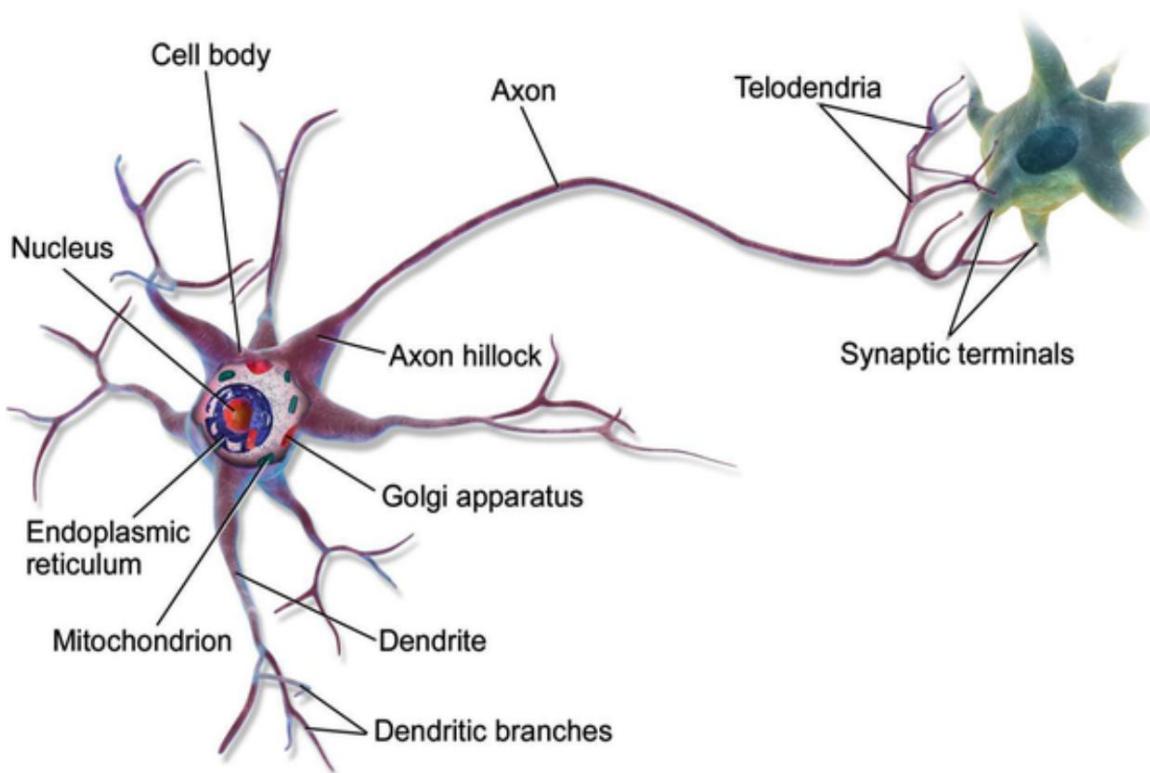


Figura 10-1. Una neurona biológica<sup>4</sup>

Por tanto, las neuronas biológicas individuales parecen comportarse de una manera sencilla, pero están organizadas en una vasta red de miles de millones, y cada neurona suele estar conectada a miles de otras neuronas. Se pueden realizar cálculos muy complejos mediante una red de neuronas bastante simples, de forma muy parecida a como un hormiguero complejo puede surgir de los esfuerzos combinados de hormigas simples. La arquitectura de las redes neuronales<sup>5</sup> biológicas (BNN) es objeto de investigación activa, pero se han mapeado algunas partes del cerebro. Estos esfuerzos muestran que las neuronas suelen estar organizadas en capas consecutivas, especialmente en la corteza cerebral (la capa externa del cerebro), como se muestra en la figura 10-2.

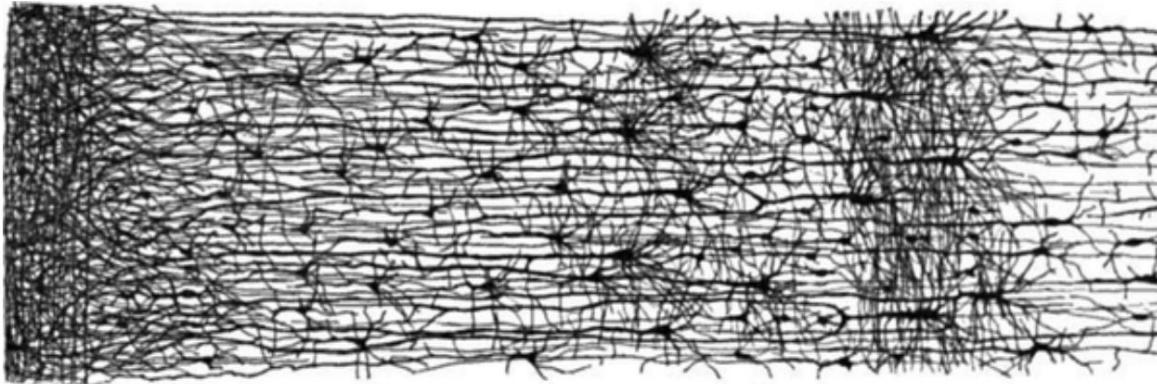


Figura 10-2. Múltiples capas en una red neuronal biológica (corteza humana)<sup>6</sup>

## Cálculos lógicos con neuronas

McCulloch y Pitts propusieron un modelo muy simple de la neurona biológica, que más tarde se conoció como neurona artificial: tiene una o más entradas binarias (encendido/apagado) y una salida binaria. La neurona artificial activa su salida cuando más de un cierto número de sus entradas están activas. En su artículo, McCulloch y Pitts demostraron que incluso con un modelo tan simplificado es posible construir una red de neuronas artificiales que puedan calcular cualquier proposición lógica que se desee.

Para ver cómo funciona una red de este tipo, construyamos algunas RNA que realicen varios cálculos lógicos (consulte la Figura 10-3), suponiendo que una neurona se activa cuando al menos dos de sus conexiones de entrada están activas.

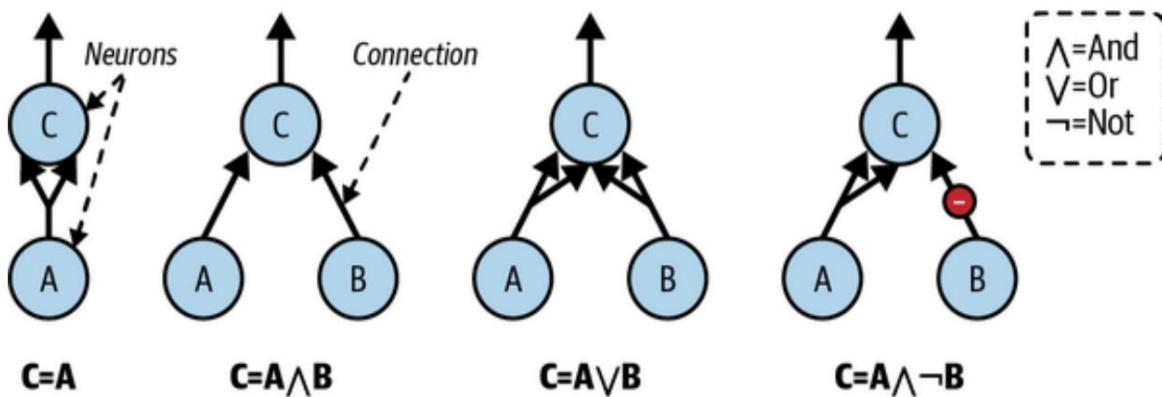


Figura 10-3. ANN que realizan cálculos lógicos simples

Veamos qué hacen estas redes:

- La primera red de la izquierda es la función de identidad: si la neurona A se activa, entonces la neurona C también se activa (ya que recibe dos señales de entrada de la neurona A); pero si la neurona A está apagada, entonces la neurona C también lo estará.
- La segunda red realiza un AND lógico: la neurona C se activa solo cuando las neuronas A y B están activadas (una sola señal de entrada no es suficiente para activar la neurona C).
- La tercera red realiza un OR lógico: la neurona C se activa si se activa la neurona A o la neurona B (o ambas).
- Finalmente, si suponemos que una conexión de entrada puede inhibir la actividad de la neurona (como es el caso de las neuronas biológicas), entonces la cuarta red calcula una proposición lógica un poco más compleja: la neurona C se activa sólo si la neurona A está activa y la neurona B está activa. apagado. Si la neurona A está activa todo el tiempo, entonces obtienes un NO lógico: la neurona C está activa cuando la neurona B está apagada, y viceversa.

Puede imaginar cómo se pueden combinar estas redes para calcular expresiones lógicas complejas (consulte los ejercicios al final del capítulo para ver un ejemplo).

## El perceptrón

El perceptrón es una de las arquitecturas de ANN más simples, inventada en 1957 por Frank Rosenblatt. Se basa en una neurona artificial ligeramente diferente ([consulte la Figura 10-4](#)) llamada unidad lógica de umbral (TLU) o, a veces, unidad de umbral lineal (LTU). Las entradas y salidas son números (en lugar de valores binarios de encendido/apagado) y cada conexión de entrada está asociada con un peso. La TLU primero calcula una función lineal de sus entradas:  $z = w_1x_1 + w_2x_2 + \dots + w_nx_n + b$ . Luego <sub>aplica una función de paso al resultado:  $h(z) = \text{paso}(z)$ .</sub> Entonces es casi como una regresión logística, excepto que usa una función escalonada en lugar de la

función logística ([Capítulo 4](#)). Al igual que en la regresión logística, los parámetros del modelo son los pesos de entrada  $w$  y el término de sesgo  $b$ .

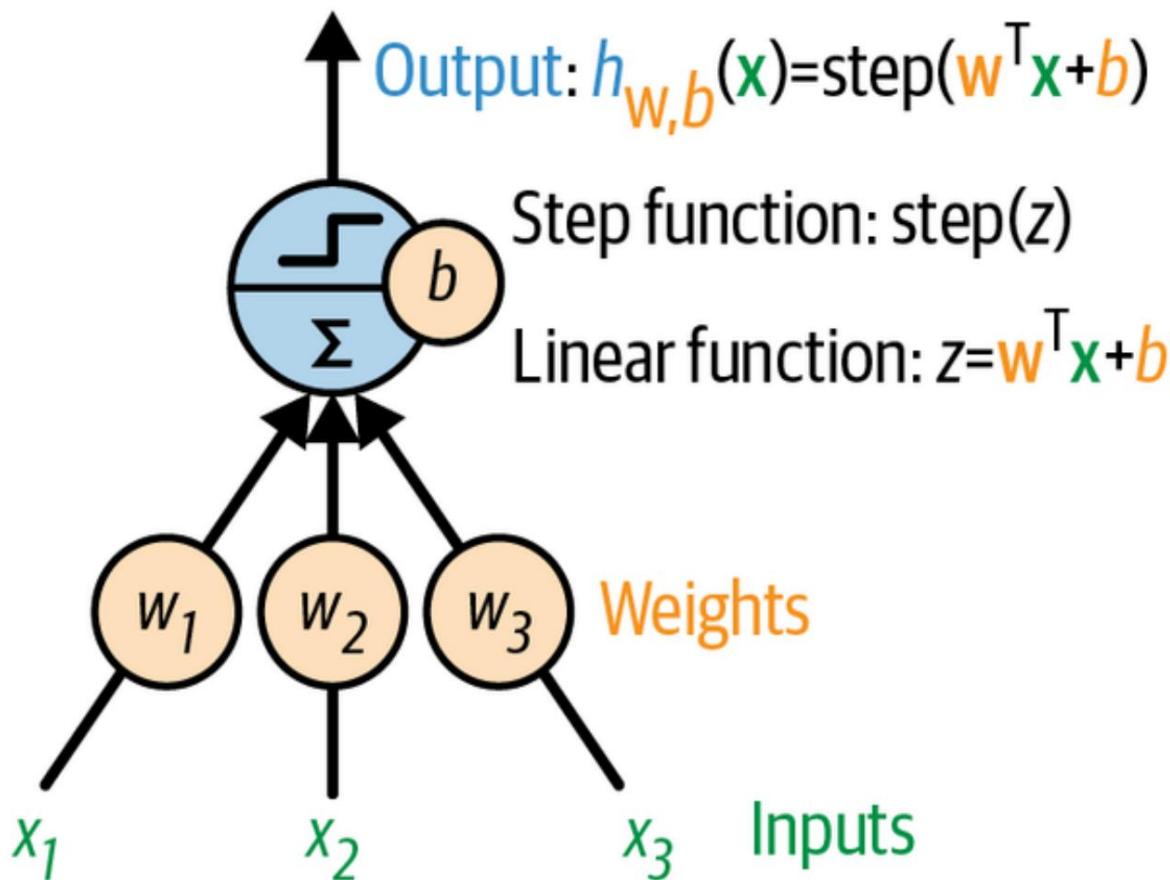


Figura 10-4. TLU: una neurona artificial que calcula una suma ponderada de sus entradas  $wx$ , más un término de sesgo  $b$ , luego aplica una función escalonada

La función escalonada más común utilizada en los perceptrones es la función escalonada de Heaviside (consulte [la ecuación 10-1](#)). A veces se utiliza la función de signo.

Ecuación 10-1. Funciones escalonadas comunes utilizadas en perceptrones (asumiendo umbral = 0)

$$\text{lado pesado (z)} = \begin{cases} 0 & \text{si } z < 0 \\ 1 & \text{si } z \geq 0 \end{cases}$$

$$\text{signo (z)} = \begin{cases} -1 & \text{si } z < 0 \\ 0 & \text{si } z = 0 \\ +1 & \text{si } z > 0 \end{cases}$$

Se puede utilizar una única TLU para una clasificación binaria lineal simple. Él calcula una función lineal de sus entradas, y si el resultado excede un umbral, genera la clase positiva. De lo contrario, genera el clase negativa. Esto puede recordarle a la regresión logística.

(Capítulo 4) o clasificación SVM lineal (Capítulo 5). Podrías, por Por ejemplo, utilice una única TLU para clasificar las flores de iris según el tamaño de sus pétalos. largo y ancho. Entrenar a una TLU de este tipo requeriría encontrar la persona adecuada valores para  $ww_1$ ,  $ww_2$ , y  $b$  (el algoritmo de entrenamiento se analiza en breve).

Un perceptrón está compuesto por una o más TLU organizadas en un solo capa, donde cada TLU está conectada a cada entrada. Tal capa es llamada capa completamente conectada o capa densa. Las entradas constituyen la capa de entrada. Y dado que la capa de TLU produce el final salidas, se llama capa de salida. Por ejemplo, un perceptrón con dos entradas y tres salidas se representan en la Figura 10-5.

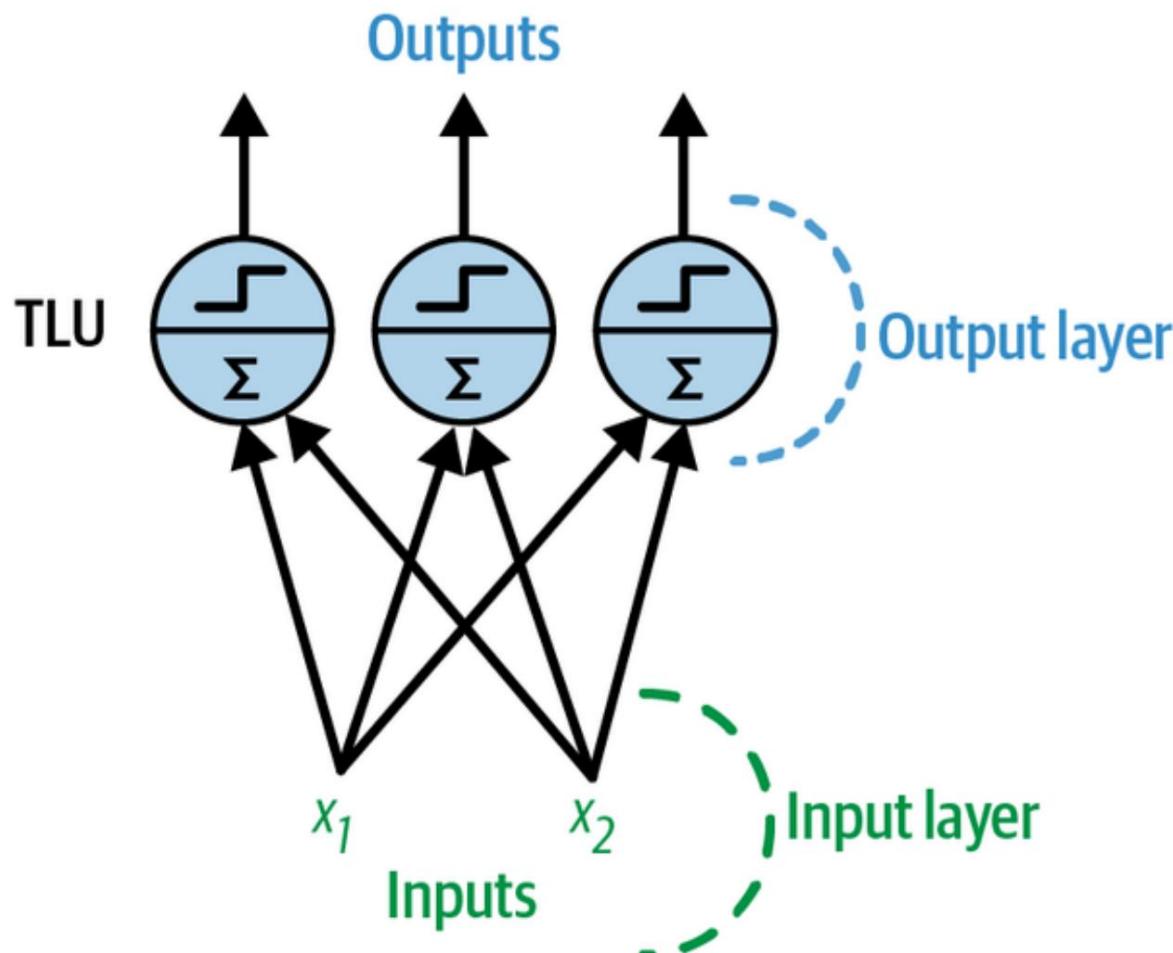


Figura 10-5. Arquitectura de un perceptrón con dos neuronas de entrada y tres de salida.

Este perceptrón puede clasificar instancias simultáneamente en tres clases binarias diferentes, lo que lo convierte en un clasificador de etiquetas múltiples. También se puede utilizar para clasificación multiclas.

Gracias a la magia del álgebra lineal, [la ecuación 10-2](#) se puede utilizar para calcular de manera eficiente las salidas de una capa de neuronas artificiales para varios casos a la vez.

Ecuación 10-2. Calcular las salidas de una capa completamente conectada

$$h_{W,b}(X) = XW + b$$

En esta ecuación:

- Como siempre,  $X$  representa la matriz de características de entrada. Tiene una fila por instancia y una columna por característica.
- La matriz de pesos  $W$  contiene todos los pesos de conexión. Tiene una fila por entrada y una columna por neurona.
- El vector de sesgo  $b$  contiene todos los términos de sesgo: uno por neurona.
- La función  $\sigma$  se llama función de activación: cuando las neuronas artificiales son TLU, es una función escalonada (en breve analizaremos otras funciones de activación).

### NOTA

En matemáticas, la suma de una matriz y un vector no está definida.

Sin embargo, en ciencia de datos, permitimos la "difusión": agregar un vector a una matriz significa agregarlo a cada fila de la matriz. Entonces,  $XW + b$  primero multiplica  $X$  por  $W$ , lo que da como resultado una matriz con una fila por instancia y una columna por salida, luego agrega el vector  $b$  a cada fila de esa matriz, lo que suma cada término de sesgo a la salida correspondiente, por cada instancia. Además, luego se aplica  $\sigma$  por elementos a cada elemento de la matriz resultante.

Entonces, ¿cómo se entrena un perceptrón? El algoritmo de entrenamiento del perceptrón propuesto por Rosenblatt se inspiró en gran medida en la regla de Hebb. En su libro de 1949 *La organización del comportamiento* (Wiley), Donald Hebb sugirió que cuando una neurona biológica activa con frecuencia otra neurona, la conexión entre estas dos neuronas se fortalece.

Siegrid Löwel resumió más tarde la idea de Hebb en la pegadiza frase: "Células que se activan juntas, se conectan juntas"; es decir, el peso de la conexión entre dos neuronas tiende a aumentar cuando se activan simultáneamente.

Esta regla más tarde se conoció como regla de Hebb (o aprendizaje hebbiano).

Los perceptrones se entrenan utilizando una variante de esta regla que tiene en cuenta el error que comete la red cuando realiza una predicción; La regla de aprendizaje del perceptrón refuerza las conexiones que ayudan a reducir el error. Más específicamente, el

El perceptrón recibe una instancia de entrenamiento a la vez y para cada instancia hace sus predicciones. Por cada neurona de salida que produjo una predicción incorrecta, refuerza los pesos de conexión de las entradas que habrían contribuido a la predicción correcta.

La regla se muestra en [la ecuación 10-3](#).

Ecuación 10-3. Regla de aprendizaje del perceptrón (actualización de peso)

$$(siguiente \text{ paso}) w_{i,j} = w_{i,j} + \eta (y_j - \hat{y}_j) x_i$$

En esta ecuación:

- $w_{i,j}$  es el peso de conexión entre la entrada  $i$  y la  $j^{\text{th}}$  neurona.
- $x_i$  es el valor  $i^{\text{th}}$  de entrada  $i$  de la instancia de entrenamiento actual.
- $\hat{y}_j$  es la salida de la neurona  $j^{\text{th}}$  de salida  $j$  para el entrenamiento actual instancia.
- $y_j$  es la salida objetivo de la neurona  $j^{\text{th}}$  de salida  $j$  para la instancia de entrenamiento  $j$  actual .
- $\eta$  es la tasa de aprendizaje (ver [Capítulo 4](#)).

El límite de decisión de cada neurona de salida es lineal, por lo que los perceptrones son incapaces de aprender patrones complejos (al igual que los clasificadores de regresión logística). Sin embargo, si las instancias de entrenamiento son linealmente separables, Rosenblatt demostró que este algoritmo convergería hacia una solución.<sup>7</sup> Esto se llama teorema de convergencia del perceptrón.

Scikit-Learn proporciona una clase Perceptron que se puede utilizar prácticamente como se esperaría, por ejemplo, en el conjunto de datos del iris (presentado en el [Capítulo 4](#)):

```
importar numpy como np
desde sklearn.datasets importar load_iris
```

```

de sklearn.linear_model importar perceptrón

iris = cargar_iris(as_frame=True)
X = iris.data[["largo del pétalo (cm)", "ancho del pétalo (cm)"]].valores y
= (iris.target == 0) #
Iris setosa

per_clf = Perceptrón(estado_aleatorio=42) per_clf.fit(X,
y)

X_new = [[2, 0.5], [3, 1]] y_pred =
per_clf.predict(X_new) # predice Verdadero y Falso para estas 2 flores

```

Es posible que haya notado que el algoritmo de aprendizaje del perceptrón se parece mucho al descenso de gradiente estocástico (presentado en el [Capítulo 4](#)). De hecho, la clase Perceptron de Scikit-Learn equivale a usar un SGDClassifier con los siguientes hiperparámetros:

`loss="perceptron", learning_rate="constant", eta0=1` (la tasa de aprendizaje) y `penalización=None` (sin regularización).

En su monografía Perceptrons de 1969, Marvin Minsky y Seymour Papert destacaron una serie de serias debilidades de los perceptrones, en particular, el hecho de que son incapaces de resolver algunos problemas triviales (por ejemplo, el problema de clasificación OR exclusivo (XOR); ver el lado izquierdo). de la [Figura 10-6](#)). Esto es cierto para cualquier otro modelo de clasificación lineal (como los clasificadores de regresión logística), pero los investigadores esperaban mucho más de los perceptrones, y algunos quedaron tan decepcionados que abandonaron las redes neuronales por completo en favor de problemas de nivel superior, como la lógica y la resolución de problemas. y buscar. La falta de aplicaciones prácticas tampoco ayu

Resulta que algunas de las limitaciones de los perceptrones se pueden eliminar apilando varios perceptrones. La RNA resultante se denomina perceptrón multicapa (MLP). Un MLP puede resolver el problema XOR, como se puede verificar calculando la salida del MLP representado en el lado derecho de la [Figura 10-6](#): con entradas (0, 0) o (1,

1), la red genera 0 y con las entradas (0, 1) o (1, 0) genera 1.  
¡Intente verificar que esta red realmente resuelve el problema XOR!<sup>8</sup>

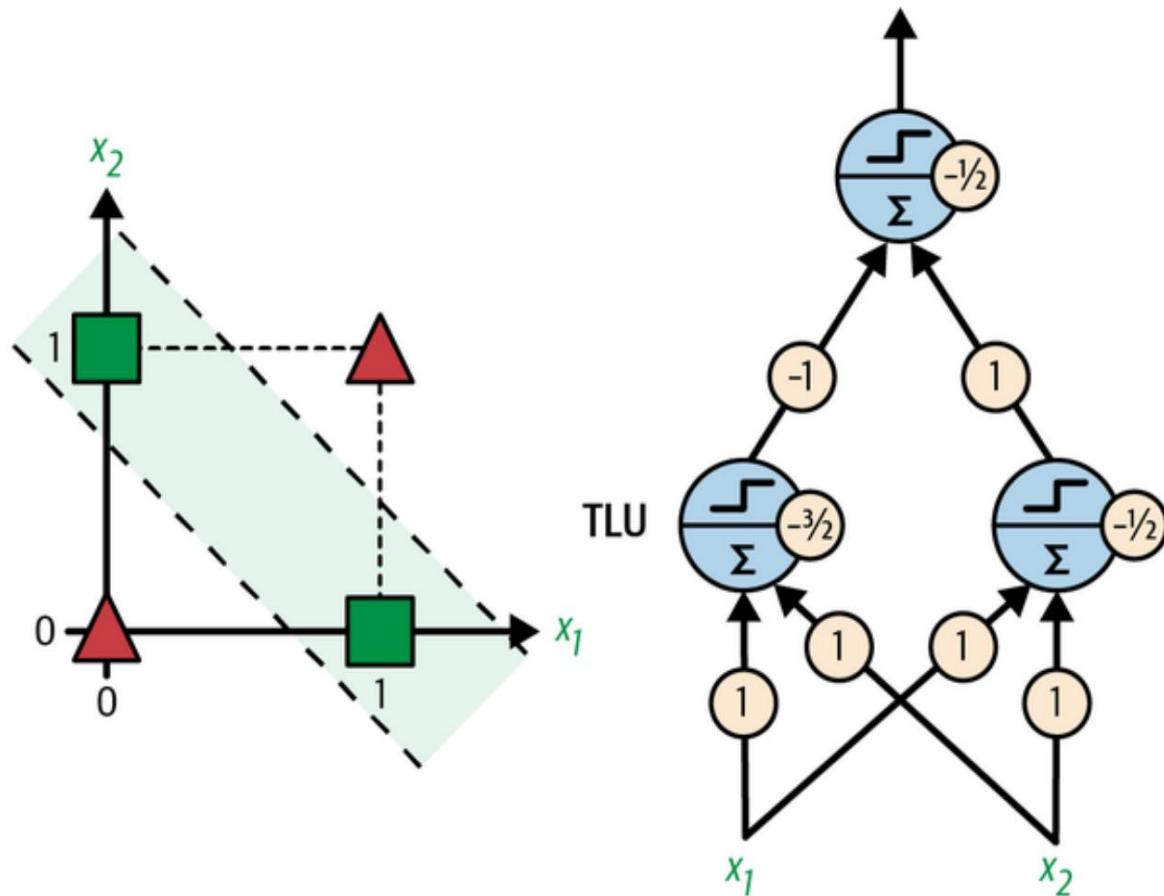


Figura 10-6. Problema de clasificación XOR y un MLP que lo resuelve

### NOTA

Al contrario de los clasificadores de regresión logística, los perceptrones no generan una probabilidad de clase. Ésta es una razón para preferir la regresión logística a los perceptrones. Además, los perceptrones no utilizan ninguna regularización de forma predeterminada y el entrenamiento se detiene tan pronto como no hay más errores de predicción en el conjunto de entrenamiento, por lo que el modelo normalmente no se generaliza tan bien como la regresión logística o un clasificador SVM lineal. Sin embargo, los perceptrones pueden entrenarse un poco más rápido.

## El perceptrón multicapa y la retropropagación

Un MLP se compone de una capa de entrada, una o más capas de TLU llamadas capas ocultas y una capa final de TLU llamada capa de salida (consulte la Figura 10-7). Las capas cercanas a la capa de entrada generalmente se denominan capas inferiores y las cercanas a las salidas generalmente se denominan capas superiores.

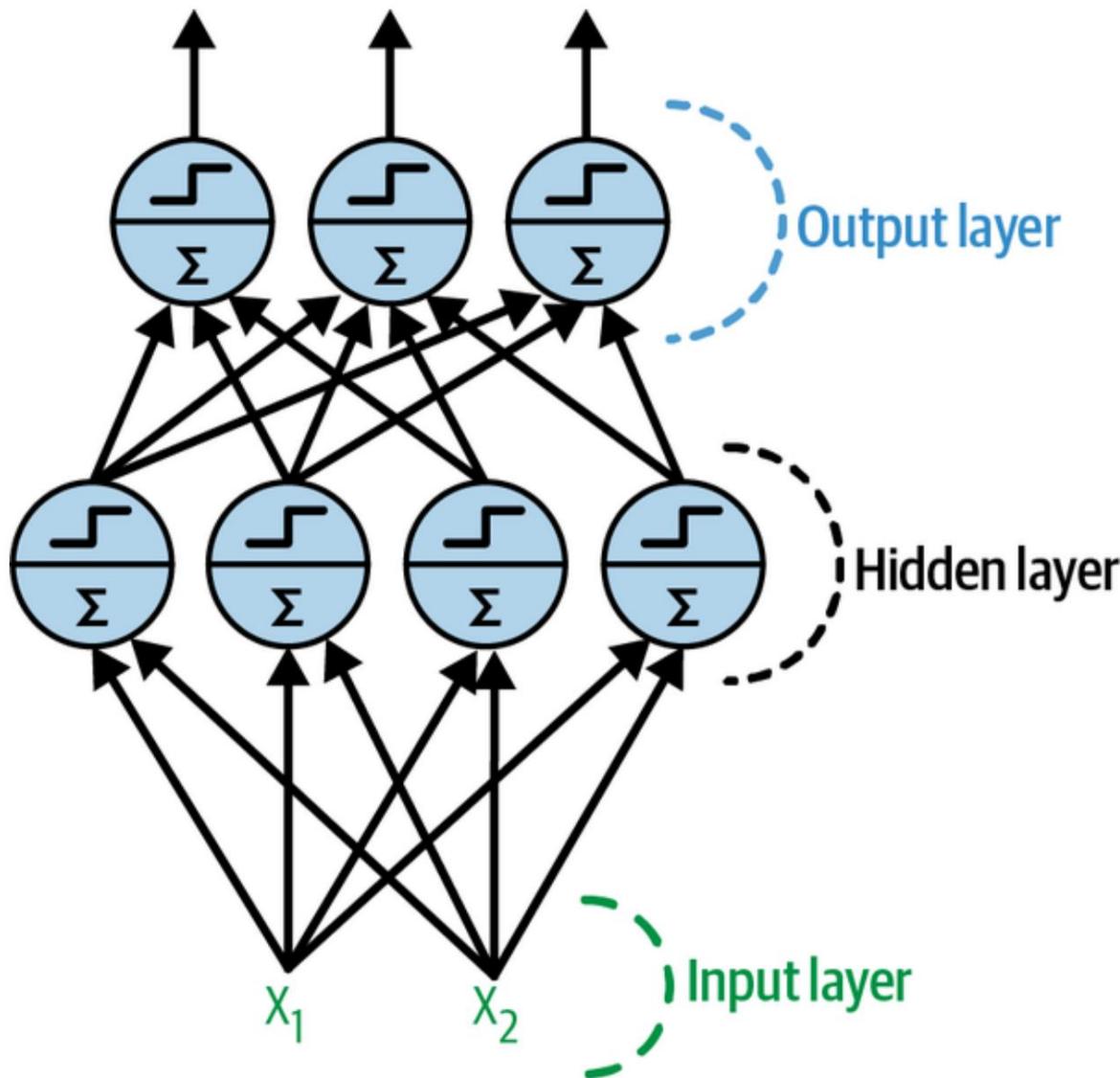


Figura 10-7. Arquitectura de un perceptrón multicapa con dos entradas, una capa oculta de cuatro neuronas y tres neuronas de salida

## NOTA

La señal fluye solo en una dirección (de las entradas a las salidas), por lo que esta arquitectura es un ejemplo de red neuronal feedforward (FNN).

Cuando una ANN contiene una pila profunda de capas ocultas, se <sup>9</sup>denomina red neuronal profunda (DNN). El campo del aprendizaje profundo estudia las DNN y, de manera más general, está interesado en modelos que contienen pilas profundas de cálculos. Aun así, mucha gente habla de aprendizaje profundo cuando se trata de redes neuronales (incluso las superficiales).

Durante muchos años, los investigadores lucharon por encontrar una manera de entrenar MLP, sin éxito. A principios de la década de 1960, varios investigadores discutieron la posibilidad de utilizar el descenso de gradiente para entrenar redes neuronales, pero, como vimos en el [Capítulo 4](#), esto requiere calcular los gradientes del error del modelo con respecto a los parámetros del modelo; En ese momento no estaba claro cómo hacer esto de manera eficiente con un modelo tan complejo que contenía tantos parámetros, especialmente con las computadoras que tenían en ese entonces.

Luego, en 1970, un investigador llamado Seppo Linnainmaa introdujo en su tesis de maestría una técnica para calcular todos los gradientes de forma automática y eficiente. Este algoritmo ahora se llama diferenciación automática en modo inverso (o autodiff en modo inverso para abreviar).

En solo dos pasos a través de la red (uno hacia adelante y otro hacia atrás), es capaz de calcular los gradientes del error de la red neuronal con respecto a cada parámetro del modelo. En otras palabras, puede descubrir cómo se debe modificar el peso de cada conexión y cada sesgo para reducir el error de la red neuronal. Estos gradientes se pueden utilizar luego para realizar un paso de descenso de gradiente. Si repite este proceso de calcular los gradientes automáticamente y realiza un paso de descenso de gradiente, el error de la red neuronal disminuirá gradualmente hasta que finalmente alcance un mínimo. Esta combinación de inversión

El modo de diferenciación automática y descenso de gradiente ahora se llama retropropagación (o backprop para abreviar).

## NOTA

Existen varias técnicas de autodiff, con diferentes pros y contras. La diferenciación automática en modo inverso es muy adecuada cuando la función a diferenciar tiene muchas variables (por ejemplo, pesos y sesgos de conexión) y pocas salidas (por ejemplo, una pérdida). Si desea obtener más información sobre la diferenciación automática, consulte [el Apéndice B](#).

En realidad, la retropropagación se puede aplicar a todo tipo de gráficos computacionales, no sólo a redes neuronales: de hecho, la tesis de maestría de Linnainmaa no trataba sobre redes neuronales, sino que era más general. Pasaron varios años más antes de que el backprop comenzara a usarse para entrenar redes neuronales, pero todavía no era algo común. Luego, en 1985, David Rumelhart, Geoffrey Hinton y Ronald Williams publicaron un [artículo innovador](#). analizar cómo la retropropagación permitió a las redes neuronales aprender representaciones internas útiles. Sus resultados fueron tan impresionantes que la retropropagación se popularizó rápidamente en el campo. Hoy en día, es, con diferencia, la técnica de entrenamiento de redes neuronales más popular.

Repasemos nuevamente cómo funciona la retropropagación con un poco más de detalle:

- Maneja un mini lote a la vez (por ejemplo, que contiene 32 instancias cada uno) y pasa por el conjunto de entrenamiento completo varias veces. Cada paso se llama época.
- Cada mini lote ingresa a la red a través de la capa de entrada. Luego, el algoritmo calcula la salida de todas las neuronas en la primera capa oculta, para cada instancia del minilote. El resultado se pasa a la siguiente capa, su salida se calcula y se pasa a la siguiente capa, y así sucesivamente hasta que obtengamos la salida de la última.

capa, la capa de salida. Este es el pase hacia adelante: es exactamente como hacer predicciones, excepto que todos los resultados intermedios se conservan ya que son necesarios para el pase hacia atrás.

- A continuación, el algoritmo mide el error de salida de la red (es decir, utiliza una función de pérdida que compara la salida deseada y la salida real de la red, y devuelve alguna medida del error).
- Luego calcula cuánto contribuyó al error cada sesgo de salida y cada conexión a la capa de salida. Esto se hace analíticamente aplicando la regla de la cadena (quizás la regla más fundamental en cálculo), lo que hace que este paso sea rápido y preciso.
- Luego, el algoritmo mide cuántas de estas contribuciones de error provienen de cada conexión en la capa inferior, nuevamente usando la regla de la cadena, trabajando hacia atrás hasta llegar a la capa de entrada. Como se explicó anteriormente, este paso inverso mide eficientemente el gradiente de error en todos los pesos y sesgos de conexión en la red propagando el gradiente de error hacia atrás a través de la red (de ahí el nombre del algoritmo).
- Finalmente, el algoritmo realiza un paso de descenso de gradiente para ajustar todos los pesos de conexión en la red, utilizando los gradientes de error que acaba de calcular.

### ADVERTENCIA

Es importante inicializar aleatoriamente los pesos de conexión de todas las capas ocultas; de lo contrario, el entrenamiento fallará. Por ejemplo, si inicializa todos los pesos y sesgos a cero, entonces todas las neuronas en una capa determinada serán perfectamente idénticas y, por lo tanto, la retropropagación las afectará exactamente de la misma manera, por lo que seguirán siendo idénticas. En otras palabras, a pesar de tener cientos de neuronas por capa, tu modelo actuará como si tuviera solo una neurona por capa: no será demasiado inteligente. Si, en cambio, inicializas los pesos al azar, rompes la simetría y permites la retropropagación para entrenar un equipo diverso de neuronas.

En resumen, la propagación hacia atrás hace predicciones para un mini lote (paso hacia adelante), mide el error, luego recorre cada capa en reversa para medir la contribución al error de cada parámetro (paso hacia atrás) y finalmente ajusta los pesos y sesgos de la conexión para reducir el error (paso de descenso de gradiente).

Para que backprop funcione correctamente, Rumelhart y sus colegas hicieron un cambio clave en la arquitectura del MLP: reemplazaron la función escalonada con la función logística,  $\sigma(z) = 1 / (1 + \exp(-z))$ , también llamada la función sigmoidea . Esto era esencial porque la función de paso contiene sólo segmentos planos, por lo que no hay gradiente con el que trabajar (el descenso de gradiente no puede moverse en una superficie plana), mientras que la función sigmoidea tiene una derivada distinta de cero bien definida en todas partes, lo que permite que el descenso de gradiente haga algo progreso en cada paso. De hecho, el algoritmo de retropropagación funciona bien con muchas otras funciones de activación, no solo con la función sigmoidea. Aquí hay otras dos opciones populares:

La función tangente hiperbólica:  $\tanh(z) = 2\sigma(2z) - 1$

Al igual que la función sigmoidea, esta función de activación tiene forma de S, es continua y diferenciable, pero su valor de salida oscila entre  $-1$  y  $1$  (en lugar de  $0$  a  $1$  en el caso de la función sigmoidea). Ese rango tiende a hacer que la salida de cada capa esté más o menos centrada.

alrededor de 0 al comienzo del entrenamiento, lo que a menudo ayuda a acelerar la convergencia.

La función unitaria lineal rectificada:  $\text{ReLU}(z) = \max(0, z)$

La función ReLU es continua pero desafortunadamente no diferenciable en  $z = 0$  (la pendiente cambia abruptamente, lo que puede hacer que el descenso del gradiente rebote), y su derivada es 0 para  $z < 0$ . Sin embargo, en la práctica, funciona muy bien y tiene la ventaja de ser rápido de calcular, por lo que se ha convertido en el valor predeterminado. Es importante<sup>11</sup> destacar que el hecho de que no tenga un valor de salida máximo ayuda a reducir algunos problemas durante el descenso del gradiente (volveremos a esto en el [Capítulo 11](#)).

Estas populares funciones de activación y sus derivadas se representan en [la figura 10-8](#). ¡Pero espera! ¿Por qué necesitamos funciones de activación en primer lugar? Bueno, si encadenas varias transformaciones lineales, todo lo que obtienes es una transformación lineal. Por ejemplo, si  $f(x) = 2x + 3$  y  $g(x) = 5x - 1$ , entonces al encadenar estas dos funciones lineales se obtiene otra función lineal:  $f(g(x)) = 2(5x - 1) + 3 = 10x + 1$ . Entonces, si no hay cierta no linealidad entre las capas, entonces incluso una pila profunda de capas es equivalente a una sola capa, y no se pueden resolver problemas muy complejos con eso. Por el contrario, un DNN lo suficientemente grande con activaciones no lineales puede teóricamente aproximarse a cualquier función continua.

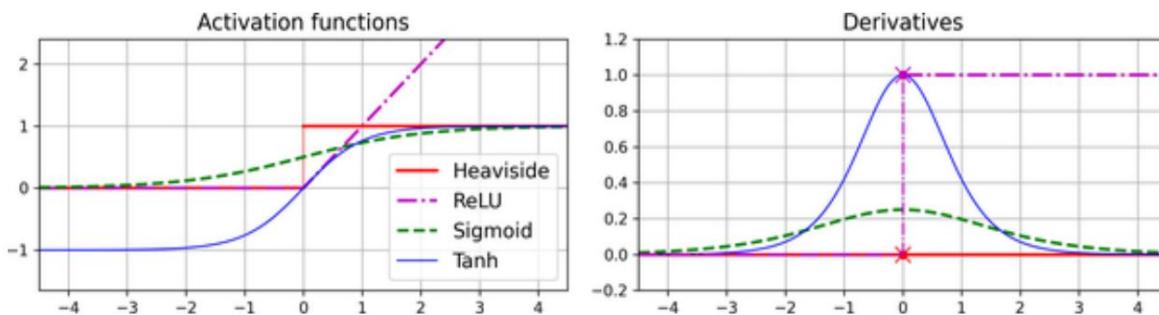


Figura 10-8. Funciones de activación (izquierda) y sus derivadas (derecha)

¡DE ACUERDO! Usted sabe de dónde provienen las redes neuronales, cuál es su arquitectura y cómo calcular sus resultados. También aprendiste sobre el algoritmo de retropropagación. Pero, ¿qué se puede hacer exactamente con las redes neuronales?

## MLP de regresión

En primer lugar, los MLP se pueden utilizar para tareas de regresión. Si desea predecir un valor único (por ejemplo, el precio de una casa, dadas muchas de sus características), entonces sólo necesita una única neurona de salida: su salida es el valor predicho. Para la regresión multivariada (es decir, para predecir múltiples valores a la vez), necesita una neurona de salida por dimensión de salida.

Por ejemplo, para localizar el centro de un objeto en una imagen, es necesario predecir coordenadas 2D, por lo que se necesitan dos neuronas de salida. Si también desea colocar un cuadro delimitador alrededor del objeto, necesitará dos números más: el ancho y el alto del objeto. Entonces, terminas con cuatro neuronas de salida.

Scikit-Learn incluye una clase `MLPRegressor`, así que usémosla para construir un MLP con tres capas ocultas compuestas de 50 neuronas cada una y entrenémoslo en el conjunto de datos de viviendas de California. Para simplificar, usaremos la función `fetch_california_housing()` de Scikit-Learn para cargar los datos. Este conjunto de datos es más simple que el que usamos en el [Capítulo 2](#), ya que contiene solo características numéricas (no hay ninguna característica de proximidad al océano) y no faltan valores. El siguiente código comienza obteniendo y dividiendo el conjunto de datos, luego crea una canalización para estandarizar las características de entrada antes de enviarlas al `MLPRegressor`. Esto es muy importante para las redes neuronales porque se entrena utilizando el descenso de gradiente y, como vimos en el [Capítulo 4](#), el descenso de gradiente no converge muy bien cuando las características tienen escalas muy diferentes. Finalmente, el código entrena el modelo y evalúa su error de validación. El modelo utiliza la función de activación ReLU en las capas ocultas y utiliza una variante de descenso de gradiente llamada Adam (ver [Capítulo 11](#)) para minimizar el

error cuadrático medio, con un poco de regularización  $\ell$  (que puedes controlar mediante el hiperparámetro alfa):

```
de sklearn.datasets importar fetch_california_housing de sklearn.metrics
importar mean_squared_error de sklearn.model_selection
importar train_test_split de sklearn.neural_network importar MLPRegressor
de sklearn.pipeline importar make_pipeline de sklearn.preprocessing
importar StandardScaler

vivienda = fetch_california_vivienda()
X_train_full, X_test, y_train_full, y_test =
    train_test_split(vivienda.datos, vivienda.objetivo, estado_aleatorio=42)
X_train, X_valid, y_train, y_valid = tren_test_split(
    X_train_full, y_train_full, random_state=42)

mlp_reg = MLPRegressor(hidden_layer_sizes=[50, 50, 50], random_state=42)
pipeline =
    make_pipeline(StandardScaler(), mlp_reg) pipeline.fit(X_train, y_train)
y_pred = pipeline.predict(X_valid) rmse =
    mean_squared_error(y_valid, y_pred,
cuadrado=False) # aproximadamente 0,505
```

Obtenemos un RMSE de validación de aproximadamente 0,505, que es comparable a lo que se obtendría con un clasificador de bosque aleatorio. ¡No está mal para un primer intento!

Tenga en cuenta que este MLP no utiliza ninguna función de activación para la capa de salida, por lo que es libre de generar cualquier valor que desee. En general, esto está bien, pero si desea garantizar que la salida siempre será positiva, entonces debe usar la función de activación ReLU en la capa de salida, o la función de activación softplus, que es una variante suave de ReLU:  $\text{softplus}(z) = \text{iniciar sesión}(1 + \exp(z))$ . Softplus está cerca de 0 cuando  $z$  es negativo y cerca de  $z$  cuando  $z$  es positivo. Finalmente, si desea garantizar que las predicciones siempre estarán dentro de un rango de valores determinado, entonces debe usar la función sigmoidea o la tangente hiperbólica y escalar los objetivos al rango apropiado: 0

a 1 para sigmoide y  $-1$  a 1 para tanh. Lamentablemente, la clase MLPRegressor no admite funciones de activación en la capa de salida.

#### ADVERTENCIA

Crear y entrenar un MLP estándar con Scikit-Learn en solo unas pocas líneas de código es muy conveniente, pero las características de la red neuronal son limitadas. Es por eso que cambiaremos a Keras en la segunda parte de este capítulo.

La clase MLPRegressor usa el error cuadrático medio, que generalmente es lo que desea para la regresión, pero si tiene muchos valores atípicos en el conjunto de entrenamiento, es posible que prefiera usar el error absoluto medio. Alternativamente, es posible que desee utilizar la pérdida de Huber, que es una combinación de ambas. Es cuadrático cuando el error es menor que un umbral  $\delta$  (típicamente 1) pero lineal cuando el error es mayor que  $\delta$ . La parte lineal la hace menos sensible a los valores atípicos que el error cuadrático medio, y la parte cuadrática le permite converger más rápido y ser más precisa que el error absoluto medio. Sin embargo, MLPRegressor solo admite MSE.

La tabla 10-1 resume la arquitectura típica de un MLP de regresión.

Tabla 10-1. Arquitectura MLP de regresión típica

Hiperparámetro Valor típico	
# capas ocultas	Depende del problema, pero normalmente de 1 a 5
# neuronas por capa oculta	Depende del problema, pero normalmente entre 10 y 100
# neuronas de salida 1 por dimensión de predicción	
Activación oculta ReLU	
Activación de salida	Ninguno, o ReLU/softplus (si salidas positivas) o sigmoide/tanh (si salidas acotadas)
Función de pérdida	MSE o Huber si hay valores atípicos

## Clasificación MLP

Los MLP también se pueden utilizar para tareas de clasificación. Para un problema de clasificación binaria, solo necesita una única neurona de salida que utilice la función de activación sigmoidea: la salida será un número entre 0 y 1, que puede interpretar como la probabilidad estimada de la clase positiva. La probabilidad estimada de la clase negativa es igual a uno menos ese número.

Los MLP también pueden manejar fácilmente tareas de clasificación binaria de múltiples etiquetas (consulte [el Capítulo 3](#)). Por ejemplo, podría tener un sistema de clasificación de correo electrónico que prediga si cada correo electrónico entrante es spam o spam y, simultáneamente, prediga si es un correo electrónico urgente o no urgente. En este caso, necesitaría dos neuronas de salida, ambas usando la función de activación sigmoidea: la primera generaría la probabilidad de que el correo electrónico sea spam y la segunda generaría la probabilidad de que sea urgente. De manera más general, dedicarías una neurona de salida para cada clase positiva. Tenga en cuenta que las probabilidades de salida no necesariamente suman 1. Esto permite que el modelo genere cualquier combinación de etiquetas: puede tener jamón no urgente, jamón

spam no urgente y quizás incluso spam urgente (aunque eso probablemente sería un error).

Si cada instancia puede pertenecer solo a una sola clase, de tres o más clases posibles (por ejemplo, clases 0 a 9 para clasificación de imágenes de dígitos), entonces necesita tener una neurona de salida por clase y debe usar la función de activación softmax. para toda la capa de salida (ver [Figura 10-9](#)). La función softmax (introducida en [el Capítulo 4](#)) asegurará que todas las probabilidades estimadas estén entre 0 y 1 y que sumen 1, ya que las clases son exclusivas. Como vio en [el Capítulo 3](#), esto se llama clasificación multiclas.

Con respecto a la función de pérdida, dado que estamos prediciendo distribuciones de probabilidad, la pérdida de entropía cruzada (o entropía x o pérdida logarítmica para abreviar, consulte [el Capítulo 4](#)) es generalmente una buena opción.

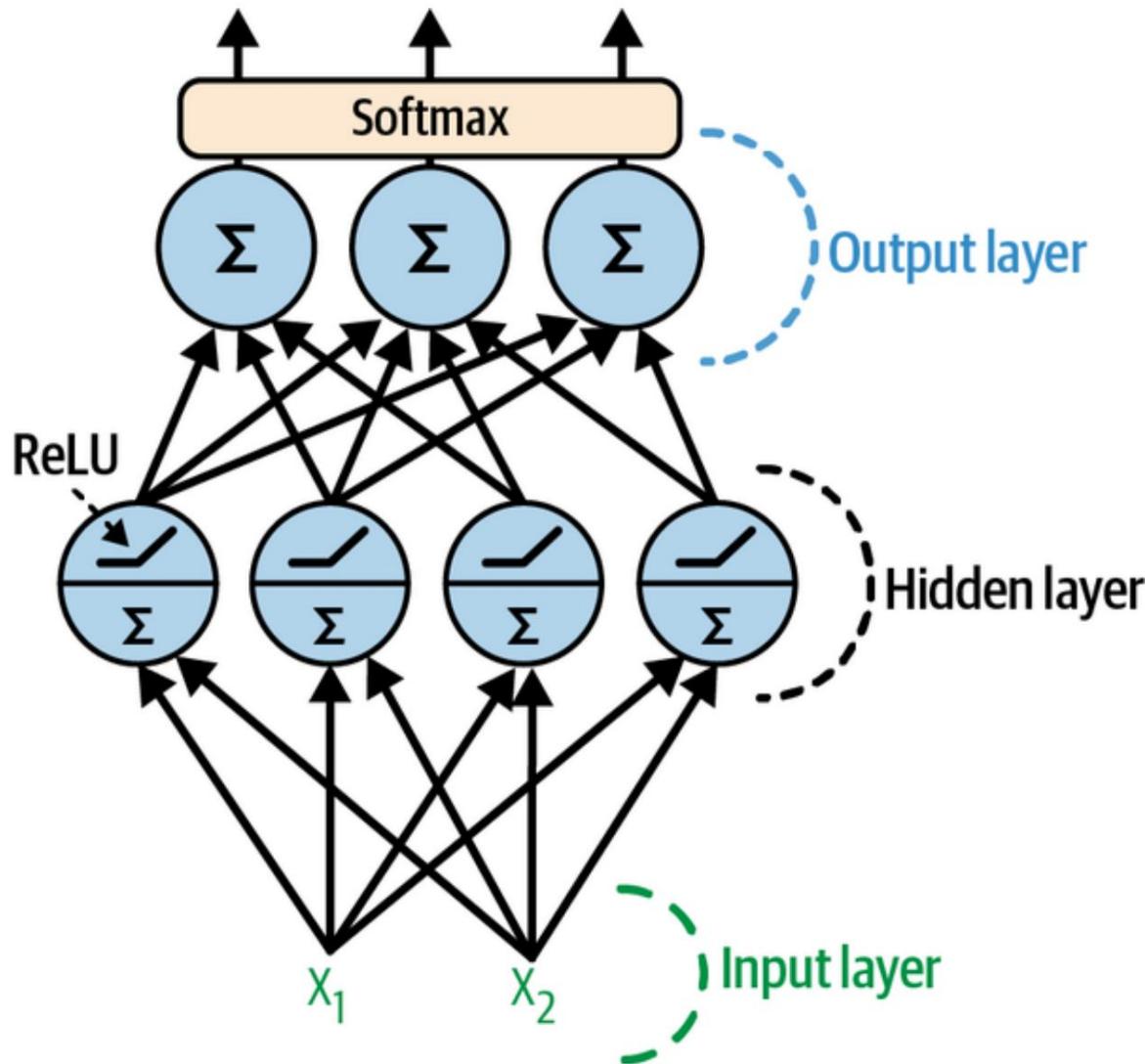


Figura 10-9. Un MLP moderno (incluidos ReLU y softmax) para clasificación

Scikit-Learn tiene una clase `MLPClassifier` en el paquete `sklearn.neural_network`. Es casi idéntica a la clase `MLPRegressor`, excepto que minimiza la entropía cruzada en lugar del MSE. Pruébelo ahora, por ejemplo en el conjunto de datos del iris. Es casi una tarea lineal, por lo que una sola capa con 5 a 10 neuronas debería ser suficiente (asegúrate de escalar las características).

**La Tabla 10-2** resume la arquitectura típica de un MLP de clasificación.

**Tabla 10-2. Arquitectura MLP de clasificación típica**

Clasificación de hiperparámetros	Binario	Clasificación binaria multietiqueta	Clasificación multiclas
# capas ocultas Normalmente de 1 a 5 capas, dependiendo de la tarea			
# neuronas de salida 1		1 por etiqueta binaria	1 por clase
Activación de la capa de salida	Sigmoideo	Sigmoideo	softmax
Función de pérdida	entropía x	entropía x	entropía x

## CONSEJO

Antes de continuar, le recomiendo que realice el ejercicio 1 al final de este capítulo. Jugarás con varias arquitecturas de redes neuronales y visualizarás sus resultados utilizando el área de juegos de TensorFlow. Esto será muy útil para comprender mejor las MLP, incluidos los efectos de todos los hiperparámetros (número de capas y neuronas, funciones de activación y más).

¡Ahora tiene todos los conceptos que necesita para comenzar a implementar MLP con Keras!

### Implementación de MLP con Keras

Keras es la API de aprendizaje profundo de alto nivel de TensorFlow: le permite construir, entrenar, evaluar y ejecutar todo tipo de redes neuronales. La biblioteca Keras original fue desarrollada por François Chollet como parte de un proyecto de investigación<sup>12</sup> y se lanzó como un proyecto independiente de código abierto en marzo de 2015. Rápidamente ganó popularidad debido a su facilidad de uso, flexibilidad y hermoso diseño.

## NOTA

Keras solía admitir múltiples backends, incluidos TensorFlow, PlaidML, Theano y Microsoft Cognitive Toolkit (CNTK) (los dos últimos lamentablemente están en desuso), pero desde la versión 2.4, Keras es solo para TensorFlow. De manera similar, TensorFlow solía incluir múltiples API de alto nivel, pero Keras fue elegida oficialmente como su API de alto nivel preferida cuando salió TensorFlow 2. La instalación de TensorFlow también instalará automáticamente Keras, y Keras no funcionará sin TensorFlow instalado. En resumen, Keras y TensorFlow se enamoraron y se casaron. Otras bibliotecas populares de aprendizaje profundo incluyen [PyTorch de Facebook](#). y [JAX de Google](#). <sup>13</sup>

¡Ahora usemos Keras! Comenzaremos construyendo un MLP para la clasificación de imágenes.

## NOTA

Los tiempos de ejecución de Colab vienen con versiones recientes de TensorFlow y Keras preinstaladas. Sin embargo, si desea instalarlos en su propia máquina, consulte las instrucciones de instalación en <https://homl.info/install>.

## Construyendo un clasificador de imágenes usando el secuencial API

Primero, necesitamos cargar un conjunto de datos. Usaremos Fashion MNIST, que es un reemplazo directo de MNIST (presentado en el [Capítulo 3](#)). Tiene exactamente el mismo formato que MNIST (70.000 imágenes en escala de grises de  $28 \times 28$  píxeles cada una, con 10 clases), pero las imágenes representan artículos de moda en lugar de dígitos escritos a mano, por lo que cada clase es más diversa y el problema resulta significativamente mayor. más desafiante que MNIST. Por ejemplo, un modelo lineal simple alcanza aproximadamente un 92 % de precisión en MNIST, pero solo alrededor de un 83 % en Fashion MNIST.

Usando Keras para cargar el conjunto de datos

Keras proporciona algunas funciones de utilidad para buscar y cargar conjuntos de datos comunes, incluidos MNIST, Fashion MNIST y algunos más. Carguemos Fashion MNIST. Ya está mezclado y dividido en un conjunto de entrenamiento (60 000 imágenes) y un conjunto de prueba (10 000 imágenes), pero guardaremos las últimas 5000 imágenes del conjunto de entrenamiento para su validación:

```
importar tensorflow como tf
```

```
fashion_mnist = tf.keras.datasets.fashion_mnist.load_data()  
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist X_train,  
y_train =  
X_train_full[:-5000], y_train_full[:-5000]  
  
X_valid, y_valid = X_train_full[-5000:], y_train_full[-5000:]
```

CONSEJO

TensorFlow generalmente se importa como tf y la API de Keras está disponible a través de tf.keras.

Al cargar MNIST o Fashion MNIST usando Keras en lugar de Scikit-Learn, una diferencia importante es que cada imagen se representa como una matriz de  $28 \times 28$  en lugar de una matriz 1D de tamaño 784. Además, las intensidades de los píxeles se representan como números enteros (de 0 a 255) en lugar de flotantes (de 0,0 a 255,0). Echemos un vistazo a la forma y el tipo de datos del conjunto de entrenamiento:

```
>>> X_train.forma  
(55000, 28, 28)  
>>> X_train.dtype  
dtype('uint8')
```

Para simplificar, reduciremos las intensidades de los píxeles al rango de 0 a 1 dividiéndolas por 255,0 (esto también las convierte en flotantes):

`X_tren, X_válido, X_prueba = X_tren / 255., X_válido / 255., X_prueba / 255.`

Con MNIST, cuando la etiqueta es igual a 5, significa que la imagen representa el dígito 5 escrito a mano. Fácil. Para Fashion MNIST, sin embargo, necesitamos la lista de nombres de clases para saber con qué estamos tratando:

```
class_names = ["Camiseta/top", "Pantalones", "Jersey",
"Vestido", "Abrigo",
"Sandalia", "Camisa", "Zapatilla", "Bolso", "Tobillo
bota"]
```

Por ejemplo, la primera imagen del conjunto de entrenamiento representa un botín:

```
>>> nombres_clase[y_train[0]]
'Botín'
```

**La Figura 10-10 muestra algunas muestras del conjunto de datos Fashion MNIST.**



Figura 10-10. Muestras de moda MNIST

Creando el modelo usando la API secuencial

¡Ahora construyamos la red neuronal! Aquí hay una clasificación MLP con dos capas ocultas:

```
tf.random.set_seed(42)
modelo = tf.keras.Sequential()
model.add(tf.keras.layers.Input(shape=[28, 28]))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(300, activation="relu"))
model.add(tf.keras.layers.Dense(100, activation="relu")) model.add(tf.
keras.layers.Dense(10, activation="softmax"))
```

Repasemos este código línea por línea:

- Primero, configure la semilla aleatoria de TensorFlow para que los resultados sean reproducibles: los pesos aleatorios de las capas ocultas y la capa de salida serán los mismos cada vez que ejecute el cuaderno. También puede optar por utilizar la función `tf.keras.utils.set_random_seed()`, que establece convenientemente las semillas aleatorias para TensorFlow, Python (`random.seed()`) y NumPy (`np.random.seed()`).
- La siguiente línea crea un modelo secuencial. Este es el tipo más simple de modelo de Keras para redes neuronales que simplemente se componen de una única pila de capas conectadas secuencialmente. Esto se llama API secuencial.
- A continuación, construimos la primera capa (una capa de entrada) y la agregamos al modelo. Especificamos la forma de entrada, que no incluye el tamaño del lote, solo la forma de las instancias. Keras necesita conocer la forma de las entradas para poder determinar la forma de la matriz de peso de conexión de la primera capa oculta.
- Luego agregamos una capa Aplanar. Su función es convertir cada imagen de entrada en una matriz 1D: por ejemplo, si recibe un lote de formas [32, 28, 28], le cambiará la forma a [32, 784]. En otras palabras, si recibe datos de entrada X, calcula `X.reshape(-1, 784)`.  
Esta capa no tiene ningún parámetro; solo está ahí para realizar un preprocesamiento simple.

- A continuación agregamos una capa oculta densa con 300 neuronas. Utilizará la función de activación ReLU. Cada capa Densa gestiona su propia matriz de pesos, que contiene todos los pesos de conexión entre las neuronas y sus entradas. También gestiona un vector de términos de sesgo (uno por neurona). Cuando recibe algunos datos de entrada, calcula **la ecuación 10-2**.
- Luego agregamos una segunda capa oculta densa con 100 neuronas, también usando la función de activación ReLU.
- Finalmente, agregamos una capa de salida Densa con 10 neuronas (una por clase), usando la función de activación softmax porque las clases son exclusivas.

## CONSEJO

Especificar `activación="relu"` equivale a especificar `activación=tf.keras.activations.relu`. Otra activación

Las funciones están disponibles en el paquete `tf.keras.activations`. Usaremos muchos de ellos en este libro; ver <https://keras.io/api/layers/activations> para ver la lista completa. También definiremos nuestras propias funciones de activación personalizadas en el [Capítulo 12](#).

En lugar de agregar las capas una por una como acabamos de hacer, suele ser más conveniente pasar una lista de capas al crear el modelo secuencial. También puede soltar la capa de Entrada y en su lugar especificar `input_shape` en la primera capa:

```
modelo = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]), tf.keras.layers.Dense(300,
    activación="relu"), tf.keras.layers.Dense(100, activación="relu"),
    tf.keras.layers.Dense(10, activación="softmax")
])
```

El método `resumen()` del modelo muestra todas las capas del modelo, incluido<sup>14</sup> el nombre de cada capa (que se genera automáticamente a menos que lo configure al crear la capa), su forma de salida (Ninguna significa que el tamaño del lote puede ser cualquier cosa) y su número de parámetros. . El resumen finaliza con el número total de parámetros, incluidos los parámetros entrenables y no entrenables. Aquí solo tenemos parámetros entrenables (verá algunos parámetros no entrenables más adelante en este capítulo):

```
>>> modelo.resumen()
```

Modelo: "secuencial"

---

Capa (tipo)	Forma de salida	
Parámetro #		
<hr/>		
<hr/>		
aplanar (Aplanar)	(Ninguno, 784)	0
denso (Denso) 235500	(Ninguno, 300)	
denso_1 (denso) 30100	(Ninguno, 100)	
denso_2 (denso)	(Ninguno, 10)	1010
<hr/>		
<hr/>		
Parámetros totales: 266,610		
Parámetros entrenables: 266,610		
Parámetros no entrenables: 0		

---



---

Tenga en cuenta que las capas densas suelen tener muchos parámetros . Por ejemplo, la primera capa oculta tiene pesos de conexión de  $784 \times 300$ , más 300 términos de sesgo, ¡lo que suma 235.500 parámetros! Esto le da al modelo mucha flexibilidad para ajustarse a los datos de entrenamiento, pero también significa

que el modelo corre el riesgo de sobreajustarse, especialmente cuando no se tienen muchos datos de entrenamiento. Vamos a volver a esto más adelante.

Cada capa de un modelo debe tener un nombre único (por ejemplo, "dense\_2"). Puede configurar los nombres de las capas explícitamente usando el argumento de nombre del constructor, pero generalmente es más sencillo dejar que Keras nombre las capas automáticamente, como acabamos de hacer. Keras toma el nombre de la clase de la capa y lo convierte en caso de serpiente (por ejemplo, una capa de la clase MyCoolLayer se llama "my\_cool\_layer" de forma predeterminada). Keras también garantiza que el nombre sea globalmente único, incluso entre modelos, agregando un índice si es necesario, como en "dense\_2". Pero, ¿por qué se molesta en hacer que los nombres sean únicos en todos los modelos? Bueno, esto hace posible fusionar modelos fácilmente sin generar conflictos de nombres.

CONSEJO

Todo el estado global administrado por Keras se almacena en una sesión de Keras, que puede borrar usando `tf.keras.backend.clear_session()`. En particular, esto restablece los contadores de nombres.

Puede obtener fácilmente la lista de capas de un modelo usando el atributo de capas, o usar el método `get_layer()` para acceder a una capa nombre:

```
>>> model.layers
[<keras.layers.core.flatten.Flatten en 0x7fa1dea02250>, <keras.layers.core.dense.Dense en 0x7fa1c8f42520>, <keras.layers.core.dense.Dense en 0x7fa188be7ac0>,
 <keras.layers.core.dense.Dense en 0x7fa188be7fa0>] >>> oculto1 =
model.layers[1] >>> oculto1.nombre

'denso'
>>> model.get_layer('denso') está oculto1 Verdadero
```

Se puede acceder a todos los parámetros de una capa utilizando sus métodos `get_weights()` y `set_weights()`. Para una capa Densa, esto incluye tanto los pesos de conexión como los términos de polarización:

```
>>> pesos, sesgos = oculto1.get_weights() >>> matriz de pesos
([[ 0.02448617,
-0.00877795, -0.02189048, ..., 0.03859074, -0.06889391],
 [0.00476504, -0.03105379, -0.0586676 -0.02763776,      , ...,
-0.04165364],
 ...,
 [ 0.07061854, -0.06960931, 0.07038955, ..., 0.00034875, 0.02878492],
[-0.06022581, 0.01577859,
-0.02585464, ...,,
0.00272203, -0.06793761]],
 dtype=float32) >>>
pesos.forma (784, 300)
>>> sesgos

matriz([0., 0., 0., 0., 0., 0., 0., 0., ..., 0., 0., 0.], dtype=float32) >>> sesgos.forma (300,)
```

Observe que la capa Densa inicializó los pesos de la conexión de forma aleatoria (lo cual es necesario para romper la simetría, como se analizó anteriormente) y los sesgos se inicializaron a ceros, lo cual está bien. Si desea utilizar un método de inicialización diferente, puede configurar `kernel_initializer` (`kernel` es otro nombre para la matriz de pesos de conexión) o `sesgo_initializer` al crear la capa. Hablaremos más sobre los inicializadores en [el Capítulo 11](#) y la lista completa se encuentra en <https://keras.io/api/layers/initializers>.

## NOTA

La forma de la matriz de peso depende de la cantidad de entradas, razón por la cual especificamos `input_shape` al crear el modelo. Si no especifica la forma de entrada, está bien: Keras simplemente esperará hasta conocer la forma de entrada antes de construir los parámetros del modelo. Esto sucederá cuando le proporciones algunos datos (por ejemplo, durante el entrenamiento) o cuando llames a su método `build()`. Hasta que se creen los parámetros del modelo, no podrá hacer ciertas cosas, como mostrar el resumen del modelo o guardar el modelo. Por lo tanto, si conoce la forma de entrada al crear el modelo, es mejor especificarla.

## Compilando el modelo

Después de crear un modelo, debe llamar a su método `compile()` para especificar la función de pérdida y el optimizador a utilizar. Opcionalmente, puede especificar una lista de métricas adicionales para calcular durante la capacitación y la evaluación:

```
model.compile(loss="sparse_categorical_crossentropy", optimizador="sgd",
               metrics=["accuracy"])
```

## NOTA

Usar `loss="sparse_categorical_crossentropy"` es equivalente a usar

`loss=tf.keras.losses.sparse_categorical_crossentropy`.

De manera similar, usar `optimizador="sgd"` es equivalente a usar `optimizador=tf.keras.optimizers.SGD()`, y usar `metrics= ["accuracy"]` es equivalente a usar `metrics=`

`[tf.keras.metrics.sparse_categorical_accuracy]` (cuando se utiliza esta pérdida). Usaremos muchas otras pérdidas, optimizadores y métricas en este libro; para ver las listas completas, consulte <https://keras.io/api/losses>, <https://keras.io/api/optimizers>, y <https://keras.io/api/metrics>.

Este código requiere explicación. Usamos la pérdida "sparse\_categorical\_crossentropy" porque tenemos etiquetas escasas (es decir, para cada instancia, solo hay un índice de clase objetivo, de 0 a 9 en este caso), y las clases son exclusivas. Si, en cambio, tuviéramos una probabilidad objetivo por clase para cada instancia (como vectores one-hot, por ejemplo, [0., 0., 0., 1., 0., 0., 0., 0., 0.] para representar la clase 3), entonces necesitaríamos usar la pérdida "categorical\_crossentropy" en su lugar. Si estuviéramos haciendo una clasificación binaria o una clasificación binaria de etiquetas múltiples, entonces usaríamos la función de activación "sigmoidea" en la capa de salida en lugar de la función de activación "softmax", y usaríamos la pérdida "binary\_crossentropy".

## CONSEJO

Si desea convertir etiquetas dispersas (es decir, índices de clase) en etiquetas vectoriales únicas, utilice la función `tf.keras.utils.to_categorical()`. Para hacerlo al revés, use la función `np.argmax()` con `eje=1`.

Con respecto al optimizador, "sgd" significa que entrenaremos el modelo utilizando un descenso de gradiente estocástico. En otras palabras, Keras realizará el algoritmo de retropropagación descrito anteriormente (es decir, autodiff en modo inverso más descenso de gradiente). Analizaremos optimizadores más eficientes en [el Capítulo 11](#). Mejoran el descenso de gradiente, no la diferenciación automática.

## NOTA

Cuando se utiliza el optimizador SGD, es importante ajustar la tasa de aprendizaje. Por lo tanto, generalmente querrás utilizar `optimizador=tf.keras.optimizadores.SGD(learning_rate=__??__)` para establecer la tasa de aprendizaje, en lugar de `optimizador="sgd"`, que por defecto es una tasa de aprendizaje de 0,01.

Finalmente, dado que se trata de un clasificador, es útil medir su precisión durante el entrenamiento y la evaluación, por lo que establecemos métricas = ["precisión"].

Entrenamiento y evaluación del modelo Ahora

el modelo está listo para ser entrenado. Para esto simplemente necesitamos llamar a su método fit():

```
>>> historia = model.fit(X_train, y_train, épocas=30,
...                         datos_validación=(X_válido, y_válido))
...
```

Época 1/30

```
1719/1719 [=====] - 2s 989us/paso - pérdida: 0,7220 -
sparse_categorical_accuracy : 0,7649 - val_loss: 0,4959 -
val_sparse_categorical_accuracy: 0,8332
```

Época 2/30

```
1719/1719 [=====] - 2s 964us/paso - pérdida: 0,4825 -
sparse_categorical_accuracy : 0,8332 - val_loss: 0,4567 -
val_sparse_categorical_accuracy: 0,8384
```

[...]

Época 30/30

```
1719/1719 [=====] - 2s 963us/paso - pérdida: 0,2235 -
sparse_categorical_accuracy : 0,9200 - val_loss: 0,3056 -
val_sparse_categorical_accuracy: 0,8894
```

Le pasamos las características de entrada (X\_train) y las clases de destino (y\_train), así como el número de épocas para entrenar (de lo contrario, el valor predeterminado sería solo 1, lo que definitivamente no sería suficiente para converger hacia una buena solución). También pasamos un conjunto de validación (esto es opcional). Keras medirá la pérdida y las métricas adicionales en este conjunto al final de cada época, lo cual es muy útil para ver qué tan bien se desempeña realmente el modelo. Si el rendimiento en el conjunto de entrenamiento es mucho mejor que en el conjunto de validación, su modelo probablemente esté sobreajustando el conjunto de entrenamiento o haya un error, como una discrepancia de datos entre el conjunto de entrenamiento y el conjunto de validación.

## CONSEJO

Los errores de forma son bastante comunes, especialmente al comenzar, por lo que debes familiarizarte con los mensajes de error: intenta ajustar un modelo con entradas y/o etiquetas de la forma incorrecta y observa los errores que obtienes. De manera similar, intente compilar el modelo con `loss="categorical_crossentropy"` en lugar de `loss="sparse_categorical_crossentropy"`. O puedes quitar la capa `Aplanar`.

¡Y eso es! La red neuronal está entrenada. En cada época durante el entrenamiento, Keras muestra la cantidad de minibatches procesados hasta el momento en el lado izquierdo de la barra de progreso. El tamaño del lote es 32 de forma predeterminada y, dado que el conjunto de entrenamiento tiene 55 000 imágenes, el modelo pasa por 1719 lotes por época: 1718 de tamaño 32 y 1 de tamaño 24.

Después de la barra de progreso, puede ver el tiempo medio de entrenamiento por muestra y la pérdida y precisión (o cualquier otra métrica adicional que haya solicitado) tanto en el conjunto de entrenamiento como en el conjunto de validación. Observe que la pérdida de entrenamiento disminuyó, lo cual es una buena señal, y la precisión de la validación alcanzó el 88,94 % después de 30 épocas. Esto está ligeramente por debajo de la precisión del entrenamiento, por lo que se está produciendo un poco de sobreajuste, pero no en gran medida.

## CONSEJO

En lugar de pasar un conjunto de validación usando el argumento `validation_data`, puede establecer `validation_split` en la proporción del conjunto de entrenamiento que desea que Keras use para la validación. Por ejemplo, `validation_split=0.1` le dice a Keras que use el último 10% de los datos (antes de barajar) para la validación.

Si el conjunto de entrenamiento estuviera muy sesgado, con algunas clases sobrerepresentadas y otras subrepresentadas, sería útil establecer el argumento `class_weight` al llamar al método `fit()`,

dar una mayor ponderación a las clases subrepresentadas y una menor ponderación a las clases excesivamente representadas. Keras utilizaría estos pesos al calcular la pérdida. Si necesita ponderaciones por instancia, establezca el argumento `sample_weight`. Si se proporcionan `class_weight` y `sample_weight`, Keras los multiplica. Las ponderaciones por instancia podrían ser útiles, por ejemplo, si algunas instancias fueron etiquetadas por expertos mientras que otras fueron etiquetadas mediante una plataforma de crowdsourcing: es posible que desee darle más peso a las primeras. También puede proporcionar ponderaciones de muestra (pero no ponderaciones de clase) para el conjunto de validación agregándolas como un tercer elemento en la tupla `validation_data`.

El método `fit()` devuelve un objeto `History` que contiene los parámetros de entrenamiento (`history.params`), la lista de épocas por las que pasó (`history.epoch`) y, lo más importante, un diccionario (`history.history`) que contiene las pérdidas y las métricas adicionales que midió. al final de cada época en el conjunto de entrenamiento y en el conjunto de validación (si corresponde). Si usa este diccionario para crear un Pandas DataFrame y llama a su método `plot()`, obtendrá las curvas de aprendizaje que se muestran en [la Figura 10-11](#):

```
importar matplotlib.pyplot como plt importar pandas
como pd

pd.DataFrame(history.history).plot( figsize=(8, 5), xlim=[0, 29],
ylim=[0, 1], grid=True, xlabel="Época", estilo=["r --", "r--.", "b-", "b-*"]) plt.show()
```

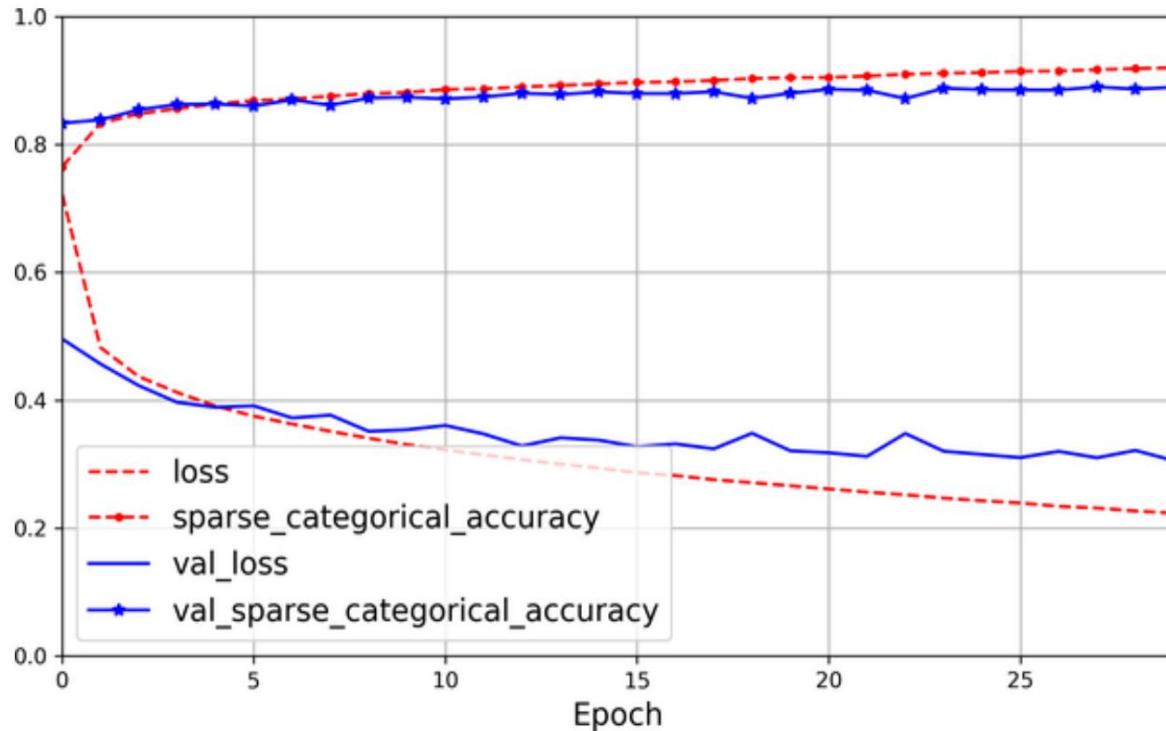


Figura 10-11. Curvas de aprendizaje: la pérdida media de entrenamiento y la precisión medidas en cada época, y la pérdida media de validación y la precisión medidas al final de cada época

Puede ver que tanto la precisión del entrenamiento como la precisión de la validación aumentan constantemente durante el entrenamiento, mientras que la pérdida de entrenamiento y la pérdida de validación disminuyen. Esto es bueno. Las curvas de validación están relativamente cerca entre sí al principio, pero se alejan con el tiempo, lo que muestra que hay un poco de sobreajuste. En este caso particular, parece que el modelo funcionó mejor en el conjunto de validación que en el conjunto de entrenamiento al comienzo del entrenamiento, pero en realidad ese no es el caso. El error de validación se calcula al final de cada época, mientras que el error de entrenamiento se calcula utilizando una media móvil durante cada época, por lo que la curva de entrenamiento debe desplazarse media época hacia la izquierda. Si haces eso, verás que las curvas de entrenamiento y validación se superponen casi perfectamente al inicio del entrenamiento.

El rendimiento del conjunto de entrenamiento termina superando el rendimiento de validación, como suele ocurrir cuando entrenas durante el tiempo suficiente. Se puede decir que el modelo aún no ha convergido del todo, ya que

la pérdida de validación sigue disminuyendo, por lo que probablemente deberías continuar entrenando. Esto es tan simple como volver a llamar al método `fit()`, ya que Keras simplemente continúa entrenando donde lo dejó: debería poder alcanzar aproximadamente el 89,8 % de precisión de validación, mientras que la precisión del entrenamiento seguirá aumentando hasta el 100 % (esto es no siempre es así).

Si no está satisfecho con el rendimiento de su modelo, debe regresar y ajustar los hiperparámetros. El primero a comprobar es la tasa de aprendizaje. Si eso no ayuda, pruebe con otro optimizador (y siempre vuelva a ajustar la tasa de aprendizaje después de cambiar cualquier hiperparámetro).

Si el rendimiento aún no es excelente, intente ajustar los hiperparámetros del modelo, como la cantidad de capas, la cantidad de neuronas por capa y los tipos de funciones de activación que se usarán para cada capa oculta. También puede intentar ajustar otros hiperparámetros, como el tamaño del lote (se puede configurar en el método `fit()` utilizando el argumento `tamaño_batch`, cuyo valor predeterminado es 32). Volveremos al ajuste de hiperparámetros al final de este capítulo. Una vez que esté satisfecho con la precisión de la validación de su modelo, debe evaluarlo en el conjunto de prueba para estimar el error de generalización antes de implementar el modelo en producción. Puedes hacer esto fácilmente usando el método `evaluar()` (también admite varios otros argumentos, como `tamaño_de_lote` y `peso_de_muestra`; consulta la documentación para obtener más detalles):

```
>>> model.evaluate(X_test, y_test) 313/313  
[=====] - 0s 626us/ paso - pérdida: 0,3243  
    - sparse_categorical_accuracy: 0,8864 [0,32431697845458984,  
      0,8863999843597412]
```

Como vio en [el Capítulo 2](#), es común obtener un rendimiento ligeramente inferior en el conjunto de prueba que en el conjunto de validación, porque los hiperparámetros se ajustan en el conjunto de validación, no en el conjunto de prueba (sin embargo, en este ejemplo, no hicimos cualquier ajuste de hiperparámetros, por lo que la menor precisión es simplemente mala suerte). Recuerda resistir el

tentación de modificar los hiperparámetros en el conjunto de prueba, o de lo contrario su estimación del error de generalización será demasiado optimista.

Usando el modelo para hacer predicciones

Ahora usemos el método predict() del modelo para hacer predicciones en nuevas instancias. Como no tenemos instancias nuevas reales, simplemente use las primeras tres instancias del conjunto de prueba:

```
>>> X_nuevo = X_prueba[:3]
>>> y_proba = modelo.predict(X_nuevo)
>>> y_proba.ronda(2)
matriz([[0. , 0. , 0.97],           , 0.      , 0.      , 0.      , 0.01, 0.          , 0.02, 0.
        , 0.      , 0.      , 0.99, 0.          , 0.01, 0.          , 0.      , 0.      , 0.
        , 0.      , 1.      , 0.      , 0.      , 0.      , 0.      , 0.      , 0.      , 0.
        , 0. ]], tipo d=float32)
```

Para cada instancia, el modelo estima una probabilidad por clase, de clase 0 a clase 9. Esto es similar a la salida del Método predict\_proba() en clasificadores Scikit-Learn. Por ejemplo, para la primera imagen estima que la probabilidad de clase 9 (tobillo bota) es del 96%, la probabilidad de la clase 7 (zapatilla de deporte) es del 2%, la probabilidad de la clase 5 (sandalias) es del 1%, y las probabilidades de la otra las clases son insignificantes. En otras palabras, tiene mucha confianza en que el La primera imagen es calzado, probablemente botines pero posiblemente zapatillas de deporte. o sandalias. Si sólo te importa la clase con el nivel más alto probabilidad estimada (incluso si esa probabilidad es bastante baja), entonces Puede usar el método argmax() para obtener la clase de probabilidad más alta. índice para cada instancia:

```
>>> importar números como np
>>> y_pred = y_proba.argmax(eje=-1)
>>> y_pred
matriz([9, 2, 1])
```

```
>>> np.array(class_names)[y_pred] array(['Bota de tobillo', 'Pullover', 'Pantalones'], dtype='<U11')
```

Aquí, el clasificador en realidad clasificó las tres imágenes correctamente (estas imágenes se muestran en la Figura 10-12):

```
>>> y_new = y_test[:3] >>>
y_new
matriz([9, 2, 1], dtype=uint8)
```

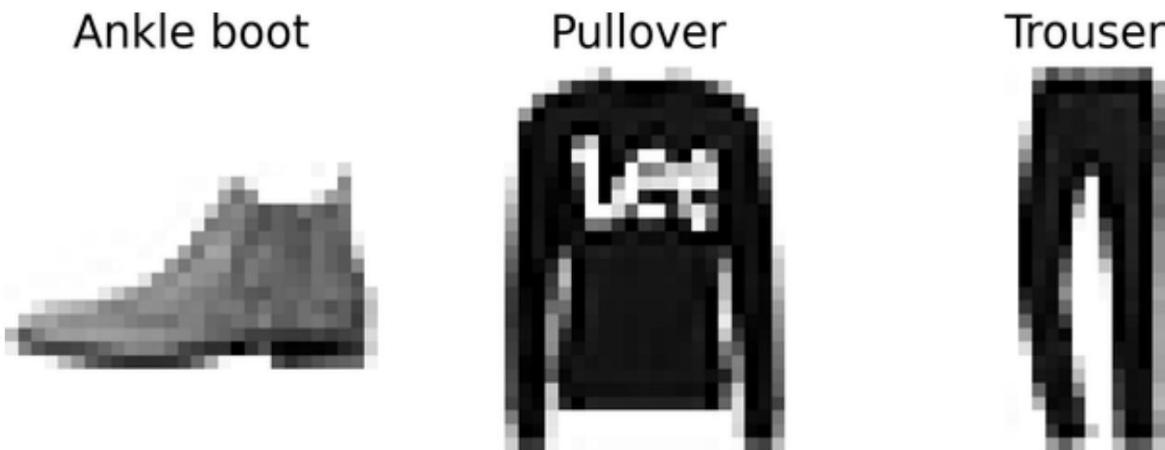


Figura 10-12. Imágenes MNIST de moda correctamente clasificadas

Ahora ya sabe cómo utilizar la API secuencial para crear, entrenar y evaluar un MLP de clasificación. Pero ¿qué pasa con la regresión?

## Construyendo un MLP de regresión usando el secuencial API

Volvamos al problema de la vivienda en California y abordémoslo usando el mismo MLP que antes, con 3 capas ocultas compuestas por 50 neuronas cada una, pero esta vez construyéndolo con Keras.

Usar la API secuencial para construir, entrenar, evaluar y usar un MLP de regresión es bastante similar a lo que hicimos para la clasificación. Las principales diferencias en el siguiente ejemplo de código son el hecho de que la capa de salida tiene una sola neurona (ya que solo queremos predecir un valor único) y no utiliza ninguna función de activación, la función de pérdida es

el error cuadrático medio, la métrica es el RMSE y estamos usando un optimizador Adam como lo hizo MLPRegressor de Scikit-Learn. Además, en este ejemplo no necesitamos una capa Aplanar, y en su lugar estamos usando una capa de Normalización como primera capa: hace lo mismo que StandardScaler de Scikit-Learn, pero debe ajustarse a los datos de entrenamiento usando su método `adapt()` antes de llamar al método `fit()` del modelo. (Keras tiene otras capas de preprocessamiento, que se tratarán en [el Capítulo 13](#)). Vamos a ver:

```
tf.random.set_seed(42)
norm_layer =
tf.keras.layers.Normalization(input_shape=X_train.shape[1:] ) modelo =
tf.keras.Sequential([ norm_layer,
tf.keras.layers.Dense(50, activación ="relu"),
tf.keras.layers.Dense(50, activación="relu"),
tf.keras.layers.Dense(50, activación="relu"), tf.keras.layers.Dense(1)

])
optimizador = tf.keras.optimizers.Adam(learning_rate=1e-3)
model.compile(loss="mse", optimizador=optimizador, métricas=
["RootMeanSquaredError"])
norm_layer.adapt(X_train) historial
= modelo. fit(X_train, y_train, épocas=20,
                datos_validación=(X_válido, y_válido))
mse_test, rmse_test = model.evaluate(X_test, y_test)
X_nuevo = X_prueba[:3]
y_pred = modelo.predict(X_nuevo)
```

### NOTA

La capa de Normalización aprende las medias de las características y las desviaciones estándar en los datos de entrenamiento cuando llama al método `adapt()`. Sin embargo, cuando muestra el resumen del modelo, estas estadísticas aparecen como no entrenables. Esto se debe a que estos parámetros no se ven afectados por el descenso del gradiente.

Como puede ver, la API secuencial es bastante limpia y sencilla. Sin embargo, aunque los modelos secuenciales son extremadamente comunes, a veces resulta útil construir redes neuronales con topologías más complejas o con múltiples entradas o salidas. Para ello, Keras ofrece la API funcional.

## Construyendo modelos complejos usando lo funcional API

Un ejemplo de una red neuronal no secuencial es una red neuronal amplia y profunda . Esta arquitectura de red neuronal fue presentada en un artículo de 2016 por Heng-Tze Cheng et al. entradas <sup>15</sup> Conecta todo o parte del directamente a la capa de salida, como se muestra en [la Figura 10-13](#). Esta arquitectura hace posible que la red neuronal aprenda tanto patrones profundos (usando la ruta profunda) como reglas simples (a través de la ruta corta). Por el contrario, un MLP normal obliga a que todos los datos fluyan a través de la pila completa de capas; por lo tanto, los patrones simples en los datos pueden terminar distorsionados por esta secuencia de transformaciones.

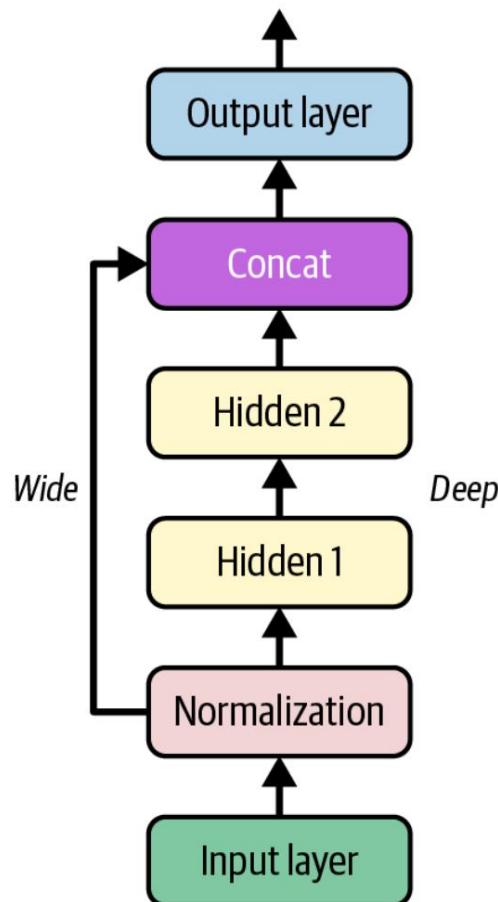


Figura 10-13. Red neuronal amplia y profunda

Construyamos una red neuronal para abordar el problema de la vivienda en California:

```

normalization_layer = tf.keras.layers.Normalization() oculta_layer1 =
tf.keras.layers.Dense(30, activación="relu") oculta_layer2 =
tf.keras.layers.Dense(30,
activación="relu") concat_layer =
tf .keras.layers.Concatenate()
salida_capa = tf.keras.layers.Dense(1)

```

```

input_ = tf.keras.layers.Input(shape=X_train.shape[1:]) normalizado =
normalization_layer(input_) oculto1 =
oculto_layer1(normalizado) oculto2 =
oculto_layer2(oculto1) concat =
concat_layer([normalizado, oculto2]) salida =
capa_salida(concat)

```

```
modelo = tf.keras.Model(entradas=[entrada_], salidas=[salida])
```

En un nivel alto, las primeras cinco líneas crean todas las capas que necesitamos para construir el modelo, las siguientes seis líneas usan estas capas como funciones para ir de la entrada a la salida, y la última línea crea un objeto del modelo Keras señalando a la entrada y a la salida.

Repasemos este código con más detalle:

- Primero, creamos cinco capas: una capa de Normalización para estandarizar las entradas, dos capas Densa con 30 neuronas cada una, usando la función de activación ReLU, una capa Concatenar y una capa Densa más con una sola neurona para la capa de salida, sin ninguna activación. función.
- A continuación, creamos un objeto de entrada (el nombre de la variable input\_ se usa para evitar eclipsar la función input() incorporada de Python). Esta es una especificación del tipo de entrada que recibirá el modelo, incluida su forma y, opcionalmente, su tipo d, que por defecto es flotante de 32 bits. En realidad, un modelo puede tener múltiples entradas, como verá en breve.
- Luego usamos la capa de Normalización como una función, pasándole el objeto de Entrada. Por eso se llama API funcional. Tenga en cuenta que solo le estamos diciendo a Keras cómo debe conectar las capas entre sí; todavía no se están procesando datos reales, ya que el objeto de entrada es solo una especificación de datos. En otras palabras, es una entrada simbólica. El resultado de esta llamada también es simbólico: normalizado no almacena ningún dato real, solo se usa para construir el modelo.
- De la misma manera, luego pasamos normalizado a oculta\_capa1, que genera oculta1, y pasamos oculta1 a oculta\_capa2, que genera oculta2.

- Hasta ahora hemos conectado las capas secuencialmente, pero luego usamos concat\_layer para concatenar la entrada y la salida de la segunda capa oculta. Una vez más, todavía no se han concatenado datos reales: todo es simbólico para construir el modelo.
- Luego pasamos concat a output\_layer, que nos da el resultado final.
- Por último, creamos un modelo Keras, especificando qué entradas y salidas usar.

Una vez que haya creado este modelo de Keras, todo será exactamente como antes, por lo que no es necesario repetirlo aquí: compila el modelo, adapta la capa de Normalización, ajusta el modelo, lo evalúa y lo utiliza para hacer predicciones.

Pero, ¿qué sucede si desea enviar un subconjunto de entidades a través del camino ancho y un subconjunto diferente (posiblemente superpuesto) a través del camino profundo, como se ilustra en la [Figura 10-14](#)? En este caso, una solución es utilizar múltiples entradas. Por ejemplo, supongamos que queremos enviar cinco funciones a través de la ruta ancha (características 0 a 4) y seis funciones a través de la ruta profunda (características 2 a 7). Podemos hacer esto de la siguiente manera:

```
input_wide = tf.keras.layers.Input(shape=[5]) # características 0 a 4
input_deep = tf.keras.layers.Input(shape=[6]) # características 2 a 7
norm_layer_wide = tf.keras.layers.Normalization() norm_layer_deep =
= tf.keras.layers.Normalization() norm_wide =
norm_layer_wide(input_wide) norm_deep =
norm_layer_deep(input_deep) oculto1 =
tf.keras.layers.Dense(30, activación="relu") (norm_deep) oculto2 =
tf.keras.layers.Dense(30, activación="relu") (hidden1) concat =
tf.keras.layers.concatenate([norm_wide, oculto2]) salida = tf.keras.layers. Modelo
denso (1) (concat) = tf.keras.Model (entradas =
[input_wide, input_deep], salidas = [salida])
```

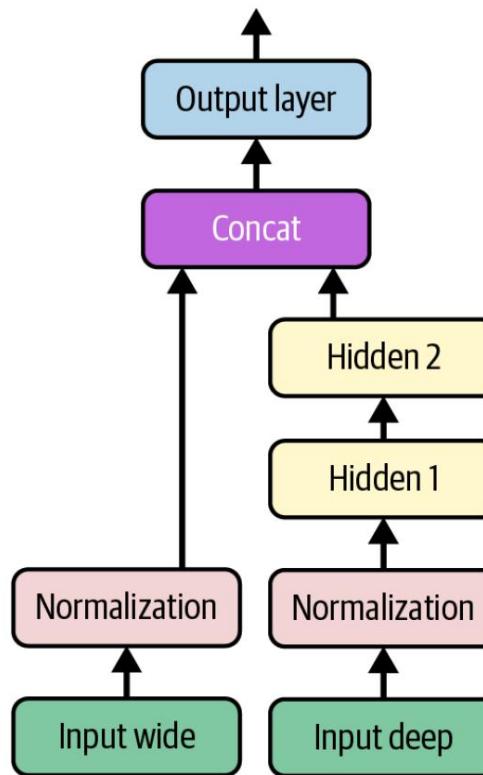


Figura 10-14. Manejo de múltiples entradas

Hay algunas cosas a tener en cuenta en este ejemplo, en comparación con el anterior:

- Cada capa Densa se crea y se llama en la misma línea. Esta es una práctica común, ya que hace que el código sea más conciso sin perder claridad. Sin embargo, no podemos hacer esto con la capa de Normalización ya que necesitamos una referencia a la capa para poder llamar a su método `adapt()` antes de ajustar el modelo.
- Usamos `tf.keras.layers.concatenate()`, que crea una capa `Concatenate` y la llama con las entradas dadas.
- Especificamos `inputs=[input_wide, input_deep]` al crear el modelo, ya que hay dos entradas.

Ahora podemos compilar el modelo como de costumbre, pero cuando llamamos al método `fit()`, en lugar de pasar una única matriz de entrada `X_train`, debemos pasar un par de matrices (`X_train_wide`,

`X_train_deep`), uno por entrada. Lo mismo ocurre con `X_valid`, y también con `X_test` y `X_new` cuando llamas a `evalua()` o `predict()`:

```
optimizador = tf.keras.optimizers.Adam(learning_rate=1e-3)
model.compile(loss="mse", optimizador=optimizador, métricas=
["RootMeanSquaredError"])

X_train_wide, X_train_deep = X_train[:, :5], X_train[:, 2:]
X_valid_wide, X_valid_deep = X_valid[:, :5], X_valid[:, 2:]
X_test_wide, X_test_deep = X_test[:, :5], X_test[:, 2:]
X_new_wide, X_new_deep = X_test_wide[:3], X_test_deep[:3]

norm_layer_wide.adapt(X_train_wide)
norm_layer_deep.adapt(X_train_deep) historial
= model.fit((X_train_wide, X_train_deep), y_train, épocas=20,
            datos_validación=((X_valid_wide,
X_valid_deep), y_valid)) mse_test
= model.evaluate((X_test_wide, X_test_deep), y_test) y_pred =
model.predict((X_new_wide, X_new_deep))
```

## CONSEJO

En lugar de pasar una tupla (`X_train_wide`, `X_train_deep`), puedes pasar un diccionario {"`input_wide`": `X_train_wide`, "`input_deep`": `X_train_deep`}, si configuras `name="input_wide"` y `name="input_deep"` al crear las entradas. Esto es muy recomendable cuando hay muchas entradas, para aclarar el código y evitar errores en el orden.

También hay muchos casos de uso en los que es posible que desee tener múltiples resultados:

- La tarea puede exigirlo. Por ejemplo, es posible que desees localizar y clasificar el objeto principal de una imagen. Esta es tanto una tarea de regresión como una tarea de clasificación.

- De manera similar, es posible que tenga varias tareas independientes basadas en los mismos datos. Claro, podrías entrenar una red neuronal por tarea, pero en muchos casos obtendrás mejores resultados en todas las tareas entrenando una única red neuronal con una salida por tarea. Esto se debe a que la red neuronal puede aprender características de los datos que son útiles en todas las tareas. Por ejemplo, podría realizar una clasificación multitarea en imágenes de rostros, utilizando una salida para clasificar la expresión facial de la persona (sonriente, sorprendida, etc.) y otra salida para identificar si lleva gafas o no.
- Otro caso de uso es como técnica de regularización (es decir, una restricción de entrenamiento cuyo objetivo es reducir el sobreajuste y así mejorar la capacidad del modelo para generalizar). Por ejemplo, es posible que desee agregar una salida auxiliar en una arquitectura de red neuronal (consulte [la Figura 10-15](#)) para garantizar que la parte subyacente de la red aprenda algo útil por sí sola, sin depender del resto de la red.

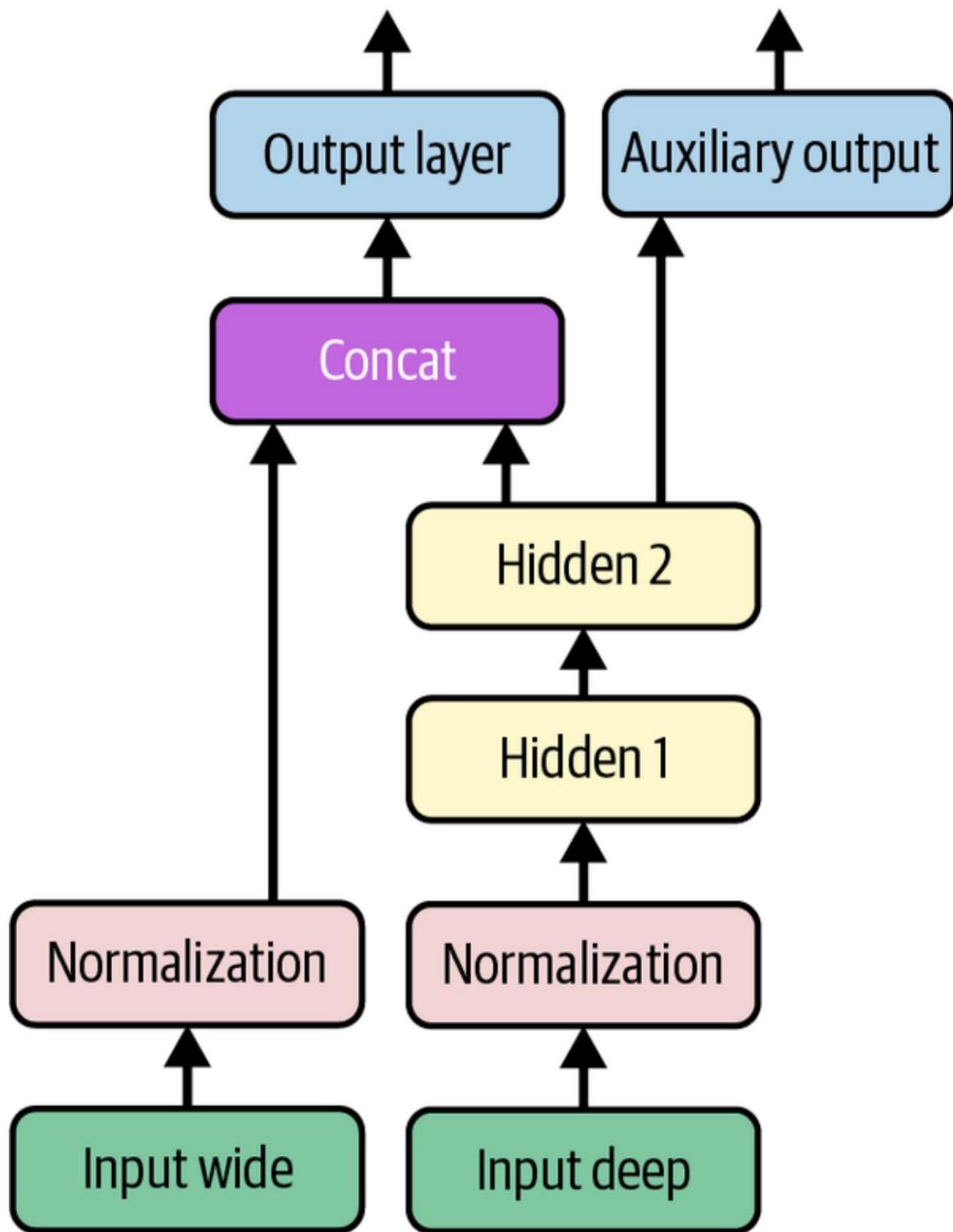


Figura 10-15. Manejo de múltiples salidas, en este ejemplo para agregar una salida auxiliar para la regularización

Agregar una salida adicional es bastante fácil: simplemente la conectamos a la capa apropiada y la agregamos a la lista de salidas del modelo. Para

Por ejemplo, el siguiente código construye la red representada en Figura 10-15:

```
[...] # Igual que arriba, hasta la capa de salida principal salida =
tf.keras.layers.Dense(1)(concat) aux_output =
tf.keras.layers.Dense(1)(hidden2) model = tf.
keras.Model(entradas=[input_wide, input_deep],
            salidas=[salida, salida_auxiliar])
```

Cada salida necesitará su propia función de pérdida. Por lo tanto, cuando compilamos el modelo, debemos pasar una lista de pérdidas. Si pasamos una única pérdida, Keras asumirá que se debe utilizar la misma pérdida para todas las salidas. De forma predeterminada, Keras calculará todas las pérdidas y simplemente las sumará para utilizar la pérdida final para el entrenamiento. Dado que nos preocupamos mucho más por la salida principal que por la salida auxiliar (ya que solo se usa para la regularización), queremos darle un peso mucho mayor a la pérdida de la salida principal. Afortunadamente, es posible establecer todos los pesos de pérdida al compilar el modelo:

```
optimizador = tf.keras.optimizers.Adam(learning_rate=1e-3)
model.compile(loss=("mse", "mse"), loss_weights=(0.9, 0.1), optimizador=optimizador,
               métricas=["RootMeanSquaredError"])
```

#### CONSEJO

En lugar de pasar una tupla `loss=("mse", "mse")`, puede pasar un diccionario `loss={"output": "mse", "aux_output": "mse"}`, asumiendo que creó las capas de salida con nombre `= "salida"` y `nombre="salida_auxiliar"`. Al igual que con las entradas, esto aclara el código y evita errores cuando hay varias salidas. También puedes pasar un diccionario para `loss_weights`.

Ahora, cuando entrenemos el modelo, debemos proporcionar etiquetas para cada salida. En este ejemplo, la salida principal y la salida auxiliar

deberían intentar predecir lo mismo, por lo que deberían utilizar las mismas etiquetas. Entonces, en lugar de pasar `y_train`, debemos pasar `(y_train, y_train)` o un diccionario `{"output": y_train, "aux_output": y_train}` si las salidas se denominaron "output" y "aux\_output". Lo mismo ocurre con `y_valid` y `y_test`:

```
norm_layer_wide.adapt(X_train_wide)
norm_layer_deep.adapt(X_train_deep) historial
= model.fit( (X_train_wide,
               X_train_deep), (y_train, y_train), épocas=20,
               validation_data=((X_valid_wide, X_valid_deep), (y_valid, y_valid)) )
```

Cuando evaluamos el modelo, Keras devuelve la suma ponderada de las pérdidas, así como todas las pérdidas y métricas individuales:

```
eval_results = model.evaluate((X_test_wide, X_test_deep), (y_test, y_test))
suma_pesada_de_pérdida,
main_loss, aux_loss, main_rmse, aux_rmse = eval_results
```

#### CONSEJO

Si configura `return_dict=True`, entonces `evalua()` devolverá un diccionario en lugar de una tupla grande.

De manera similar, el método `predict()` devolverá predicciones para cada salida:

```
y_pred_main, y_pred_aux = model.predict((X_new_wide, X_new_deep))
```

El método `predict()` devuelve una tupla y no tiene un argumento `return_dict` para obtener un diccionario. Sin embargo, tu

Puedes crear uno usando `model.output_names`:

```
y_pred_tuple = model.predict((X_new_wide, X_new_deep)) y_pred =  
dict(zip(model.output_names, y_pred_tuple))
```

Como puede ver, puede crear todo tipo de arquitecturas con la API funcional. A continuación, veremos una última forma de construir modelos Keras.

## Uso de la API de subclases para crear dinámicas Modelos

Tanto la API secuencial como la API funcional son declarativas: comienza declarando qué capas desea usar y cómo deben conectarse, y solo entonces puede comenzar a alimentar el modelo con algunos datos para entrenamiento o inferencia. Esto tiene muchas ventajas: el modelo se puede guardar, clonar y compartir fácilmente; su estructura se puede visualizar y analizar; el marco puede inferir formas y tipos de verificación, por lo que los errores pueden detectarse tempranamente (es decir, antes de que los datos pasen por el modelo). También es bastante sencillo de depurar, ya que todo el modelo es un gráfico estático de capas. Pero la otra cara es justamente esa: es estática. Algunos modelos implican bucles, formas variables, ramificaciones condicionales y otros comportamientos dinámicos. Para tales casos, o simplemente si prefiere un estilo de programación más imperativo, la API de subclases es para usted.

Con este enfoque, subclasificas la clase `Modelo`, creas las capas que necesitas en el constructor y las utilizas para realizar los cálculos que deseas en el método `call()`. Por ejemplo, crear una instancia de la siguiente clase `WideAndDeepModel` nos proporciona un modelo equivalente al que acabamos de crear con la API funcional:

```
clase WideAndDeepModel(tf.keras.Model): def  
    __init__(self, unidades=30, activación="relu",
```

```

**kwargs):
    super().__init__(**kwargs) # necesario para admitir el nombramiento del
    modelo self.norm_layer_wide
        = tf.keras.layers.Normalization()
    self.norm_layer_deep = tf.keras.layers.Normalization()
        self.hidden1 =
    tf.keras.layers.Dense(unidades,
activación=activación)
        self.hidden2 = tf.keras.layers.Dense(unidades,
activación=activación)
        self.main_output = tf.keras.layers.Dense(1) self.aux_output =
    tf.keras.layers.Dense(1)

    def llamada(self, entradas): input_wide,
        input_deep = entradas norm_wide =
    self.norm_layer_wide(input_wide) norm_deep =
    self.norm_layer_deep(input_deep) oculto1 = self.hidden1(norm_deep)
    oculto2 = self.hidden2(oculto1) concat =
    tf.keras.layers.concatenate([norm_wide, oculto2])
    salida = self.main_output(concat) aux_output = self.aux_output(hidden2)
devuelve
    salida, aux_output

```

```
modelo = WideAndDeepModel(30, activación="relu", nombre="my_cool_model")
```

Este ejemplo se parece al anterior, excepto que separamos la creación de las capas en el constructor de su uso en el método call(). Y no necesitamos crear los objetos de entrada: podemos usar el argumento de entrada para el método call().

Ahora que tenemos una instancia de modelo, podemos compilarla, adaptar sus capas de normalización (por ejemplo, usando model.norm\_layer\_wide.adapt(...)) y model.norm\_layer\_deep.adapt(...)), ajustarla y evaluarla. y usarlo para hacer predicciones, exactamente como lo hicimos con la API funcional.

La gran diferencia con esta API es que puedes incluir prácticamente cualquier cosa que quieras en el método call(): bucles for, sentencias if, operaciones de TensorFlow de bajo nivel... ¡tu imaginación es el límite (consulta el Capítulo 12)! Esto la convierte en una excelente API a la hora de experimentar con nuevas ideas, especialmente para los investigadores. Sin embargo, esta flexibilidad adicional tiene un costo: la arquitectura de su modelo está oculta dentro del método call(), por lo que Keras no puede inspeccionarla fácilmente; el modelo no se puede clonar usando `tf.keras.models.clone_model()`; y cuando llamas al método resumen(), solo obtienes una lista de capas, sin ninguna información sobre cómo están conectadas entre sí. Además, Keras no puede comprobar los tipos y formas con antelación, y es más fácil cometer errores. Entonces, a menos que realmente necesite esa flexibilidad adicional, probablemente debería ceñirse a la API secuencial o la API funcional.

## CONSEJO

Los modelos de Keras se pueden utilizar como capas normales, por lo que puedes combinarlos fácilmente para crear arquitecturas complejas.

Ahora que sabes cómo construir y entrenar redes neuronales usando Keras, ¡querrás guardarlas!

## Guardar y restaurar un modelo

Guardar un modelo Keras entrenado es tan simple como parece:

```
model.save("my_keras_model", save_format="tf")
```

Cuando configura `save_format="tf"`, Keras guarda el modelo usando el formato SavedModel de TensorFlow<sup>18</sup>: este es un directorio (con el nombre de pila) que contiene varios archivos y subdirectorios. En particular, el archivo `save_model.pb` contiene la arquitectura y la lógica del modelo en forma de un gráfico de cálculo serializado, por lo que no es necesario

implementar el código fuente del modelo para usarlo en producción; el SavedModel es suficiente (verá cómo funciona esto en el [Capítulo 12](#)). El archivo keras\_metadata.pb contiene información adicional que necesita Keras. El subdirectorio de variables contiene todos los valores de los parámetros (incluidos los pesos de conexión, los sesgos, las estadísticas de normalización y los parámetros del optimizador), posiblemente divididos en varios archivos si el modelo es muy grande. Por último, el directorio de activos puede contener archivos adicionales, como muestras de datos, nombres de características, nombres de clases, etc. De forma predeterminada, el directorio de activos está vacío. Como también se guarda el optimizador, incluidos sus hiperparámetros y cualquier estado que pueda tener, luego de cargar el modelo puedes continuar entrenando si lo deseas.

#### NOTA

Si configura save\_format="h5" o usa un nombre de archivo que termina en .h5, .hdf5 o .keras, Keras guardará el modelo en un solo archivo usando un formato específico de Keras basado en el formato HDF5. Sin embargo, la mayoría de las herramientas de implementación de TensorFlow requieren el formato SavedModel.

Normalmente tendrá un script que entrena un modelo y lo guarda, y uno o más scripts (o servicios web) que cargan el modelo y lo utilizan para evaluarlo o hacer predicciones. Cargar el modelo es tan fácil como guardarlo:

```
modelo = tf.keras.models.load_model("my_keras_model")  
y_pred_main, y_pred_aux  
= modelo.predict((X_new_wide, X_new_deep))
```

También puede utilizar save\_weights() y load\_weights() para guardar y cargar solo los valores de los parámetros. Esto incluye los pesos de conexión, los sesgos, las estadísticas de preprocessamiento, el estado del optimizador, etc. Los valores de los parámetros se guardan en uno o más archivos, como

my\_weights.data-00004-of-00052, además de un archivo de índice como my\_weights.index.

Guardar solo los pesos es más rápido y utiliza menos espacio en disco que guardar todo el modelo, por lo que es perfecto para guardar puntos de control rápidos durante el entrenamiento. Si está entrenando un modelo grande y le lleva horas o días, debe guardar puntos de control con regularidad en caso de que la computadora falle. Pero, ¿cómo se puede indicar al método fit() que guarde los puntos de control? Utilice devoluciones de llamada.

#### Usando devoluciones de llamada

El método fit() acepta un argumento de devolución de llamada que le permite especificar una lista de objetos que Keras llamará antes y después del entrenamiento, antes y después de cada época, e incluso antes y después de procesar cada lote. Por ejemplo, la devolución de llamada ModelCheckpoint guarda los puntos de control de su modelo a intervalos regulares durante el entrenamiento, de forma predeterminada al final de cada época:

```
checkpoint_cb =  
    tf.keras.callbacks.ModelCheckpoint("mis_puntos_de_control",  
  
        save_weights_only=True)  
historial = model.fit(..., devoluciones_de_llamada=[checkpoint_cb])
```

Además, si utiliza un conjunto de validación durante el entrenamiento, puede configurar save\_best\_only=True al crear ModelCheckpoint. En este caso, sólo guardará su modelo cuando su rendimiento en el conjunto de validación sea el mejor hasta el momento. De esta manera, no necesita preocuparse por entrenar durante demasiado tiempo y sobreajustar el conjunto de entrenamiento: simplemente restaure el último modelo guardado después del entrenamiento, y este será el mejor modelo en el conjunto de validación. Esta es una forma de implementar la parada temprana (introducida en [el Capítulo 4](#)), pero en realidad no detendrá el entrenamiento.

Otra forma es utilizar la devolución de llamada EarlyStopping. Interrumpirá el entrenamiento cuando no mida ningún progreso en el conjunto de validación para un

número de épocas (definidas por el argumento de la paciencia), y si configura restaurar\_best\_weights=True, volverá al mejor modelo al final del entrenamiento. Puede combinar ambas devoluciones de llamada para guardar puntos de control de su modelo en caso de que su computadora falle e interrumpir el entrenamiento temprano cuando ya no haya progreso, para evitar perder tiempo y recursos y reducir el sobreajuste:

```
early_stopping_cb =  
tf.keras.callbacks.EarlyStopping(paciencia=10,  
  
restaurar_best_weights=True) historial =  
model.fit(..., devoluciones de llamada=[checkpoint_cb, early_stopping_cb])
```

El número de épocas se puede establecer en un valor grande, ya que el entrenamiento se detendrá automáticamente cuando no haya más progreso (solo asegúrese de que la tasa de aprendizaje no sea demasiado pequeña, de lo contrario, podría seguir progresando lentamente hasta el final). La devolución de llamada EarlyStopping almacenará los pesos del mejor modelo en la RAM y los restaurará al final del entrenamiento.

CONSEJO

Muchas otras devoluciones de llamada están disponibles en el paquete `tf.keras.callbacks`.

Si necesita control adicional, puede escribir fácilmente sus propias devoluciones de llamada personalizadas. Por ejemplo, la siguiente devolución de llamada personalizada mostrará la relación entre la pérdida de validación y la pérdida de entrenamiento durante el entrenamiento (p. ej., para detectar sobreajuste):

```
clase  
PrintValTrainRatioCallback(tf.keras.callbacks.Callback):  
    def on_epoch_end(self, época, registros):
```

```
ratio = logs["val_loss"] / logs["loss"] print(f"Epoch={epoch},  
val/train={ratio:.2f}")
```

Como era de esperar, puede implementar `on_train_begin()`, `on_train_end()`, `on_epoch_begin()`, `on_epoch_end()`, `on_batch_begin()` y `on_batch_end()`. Las devoluciones de llamada también se pueden utilizar durante la evaluación y las predicciones, en caso de que alguna vez las necesite (por ejemplo, para depurar). Para la evaluación, debe implementar `on_test_begin()`, `on_test_end()`, `on_test_batch_begin()` o `on_test_batch_end()`, que son llamados por `evalua()`. Para la predicción, debe implementar `on_predict_begin()`, `on_predict_end()`, `on_predict_batch_begin()` o `on_predict_batch_end()`, que son llamados por `predict()`.

Ahora echemos un vistazo a una herramienta más que definitivamente deberías tener en tu caja de herramientas cuando uses Keras: TensorBoard.

## Usando TensorBoard para visualización

TensorBoard es una excelente herramienta de visualización interactiva que puede utilizar para ver las curvas de aprendizaje durante el entrenamiento, comparar curvas y métricas entre múltiples ejecuciones, visualizar el gráfico de cálculo, analizar estadísticas de entrenamiento, ver imágenes generadas por su modelo, visualizar datos multidimensionales complejos proyectados hasta 3D y agrupado automáticamente para usted, profile su red (es decir, mida su velocidad para identificar cuellos de botella) ¡y más!

TensorBoard se instala automáticamente cuando instalas TensorFlow. Sin embargo, necesitará un complemento de TensorBoard para visualizar los datos de creación de perfiles. Si siguió las instrucciones de instalación en <https://homl.info/install> para ejecutar todo localmente, entonces ya tienes el complemento instalado, pero si estás usando Colab, debes ejecutar el siguiente comando:

```
%pip install -q -U tensorboard-plugin-perfil
```

Para usar TensorBoard, debe modificar su programa para que envíe los datos que desea visualizar a archivos de registro binarios especiales llamados archivos de eventos. Cada registro de datos binarios se denomina resumen. El servidor TensorBoard monitoreará el directorio de registros, recogerá automáticamente los cambios y actualizará las visualizaciones: esto le permite visualizar datos en vivo (con un breve retraso), como las curvas de aprendizaje durante el entrenamiento. En general, desea apuntar el servidor TensorBoard a un directorio de registro raíz y configurar su programa para que escriba en un subdirectorio diferente cada vez. De esta manera, la misma instancia del servidor TensorBoard le permitirá visualizar y comparar datos de múltiples ejecuciones de su programa, sin mezclar todo.

Llaremos al directorio de registro raíz `my_logs` y definamos una pequeña función que genera la ruta del subdirectorio de registro en función de la fecha y hora actuales, para que sea diferente en cada ejecución:

```
desde pathlib import Ruta desde
time import strftime

def get_run_logdir(root_logdir="my_logs"): ruta de retorno
    (root_logdir) /
    strftime("run_%Y_%m_%d_%H_%M_%S")

run_logdir = get_run_logdir() # por ejemplo, my_logs/
run_2022_08_01_17_25_59
```

La buena noticia es que Keras proporciona una conveniente devolución de llamada de `TensorBoard()` que se encargará de crear el directorio de registro por usted (junto con sus directorios principales si es necesario), creará archivos de eventos y les escribirá resúmenes durante el entrenamiento. Medirá la pérdida y las métricas de entrenamiento y validación de su modelo (en este caso, MSE y RMSE), y también perfilará su red neuronal. Es sencillo de utilizar:

```
tensorboard_cb = tf.keras.callbacks.TensorBoard(run_logdir,
```

```
perfil_batch=(100, 200))
historial = model.fit(..., devoluciones de llamada=[tensorboard_cb])
```

¡Eso es todo al respecto! En este ejemplo, perfilará la red entre los lotes 100 y 200 durante la primera época. ¿Por qué 100 y 200? Bueno, a menudo se necesitan algunos lotes para que la red neuronal "calentamiento", por lo que no desea crear perfiles demasiado pronto, y la creación de perfiles utiliza recursos, por lo que es mejor no hacerlo para cada lote.

A continuación, intente cambiar la tasa de aprendizaje de 0,001 a 0,002 y ejecute el código nuevamente, con un nuevo subdirectorio de registro. Terminarás con una estructura de directorio similar a esta:

```
mis_registros
└── run_2022_08_01_17_25_59
    ├── tren
    └── ...
    └── events.out.tfevents.1659331561.my_host_name.42042.0.v2
        ├── ...
        ├── events.out.tfevents.1659331562.my_host_name.profile-empty
        │   ├── complementos
        │   └── perfil
        │       └── 2022_08_01_17_26_02
        │           ├── mi_nombre_host.input_pipeline.pb
        │           └── [...]
        │               ├── ...
        │               └── └── validación
        └── ...
            └── events.out.tfevents.1659331562.my_host_name.42042.1.v2
                ├── run_2022_08_01_17_31_12
                └── [...]
```

Hay un directorio por ejecución, cada uno de los cuales contiene un subdirectorio para registros de entrenamiento y uno para registros de validación. Ambos contienen archivos de eventos y Los registros de entrenamiento también incluyen seguimientos de perfiles.

Ahora que tiene los archivos de eventos listos, es hora de iniciar el Servidor TensorBoard. Esto se puede hacer directamente dentro de Jupyter o Colab usa la extensión Jupyter para TensorBoard, que obtiene

instalado junto con la biblioteca TensorBoard. Esta extensión está preinstalada en Colab. El siguiente código carga la extensión Jupyter para TensorBoard y la segunda línea inicia un servidor TensorBoard para el directorio `my_logs`, se conecta a este servidor y muestra la interfaz de usuario directamente dentro de Jupyter. El servidor escucha en el primer puerto TCP disponible mayor o igual a 6006 (o puede configurar el puerto que desee usando la opción `--port`).

```
%load_ext tensorboard  
%tensorboard --logdir=./my_logs
```

CONSEJO

Si está ejecutando todo en su propia máquina, es posible iniciar TensorBoard ejecutando `tensorboard --logdir=./my_logs` en una terminal. Primero debe activar el entorno Conda en el que instaló TensorBoard e ir al directorio `hands-on-ml3`. Una vez iniciado el servidor, visite <http://localhost:6006>.

Ahora deberías ver la interfaz de usuario de TensorBoard. Haga clic en la pestaña ESCALARES para ver las curvas de aprendizaje (consulte [la Figura 10-16](#)). En la parte inferior izquierda, seleccione los registros que desea visualizar (por ejemplo, los registros de entrenamiento de la primera y segunda ejecución) y haga clic en el escalar `epoch_loss`. Observe que la pérdida de entrenamiento disminuyó bastante durante ambas carreras, pero en la segunda carrera disminuyó un poco más rápido gracias a la mayor tasa de aprendizaje.

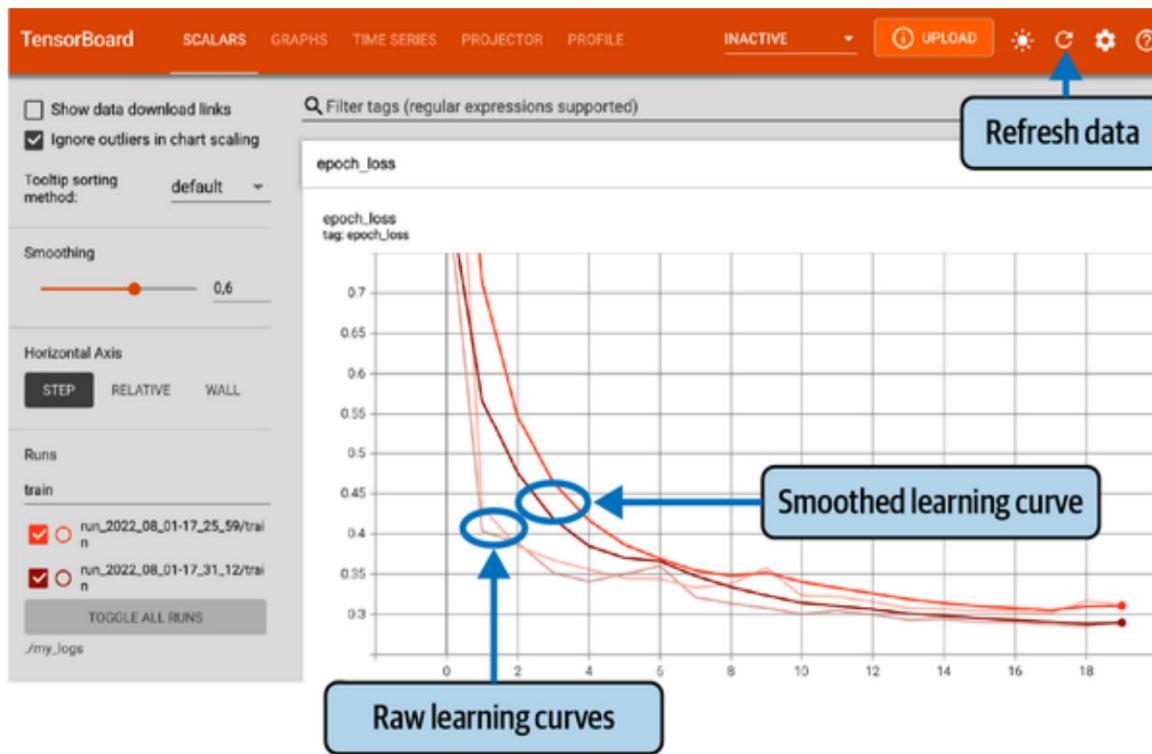


Figura 10-16. Visualizando curvas de aprendizaje con TensorBoard

También puede visualizar el gráfico de cálculo completo en la pestaña GRÁFICOS, los pesos aprendidos proyectados en 3D en la pestaña PROYECTOR y las trazas de perfilado en la pestaña PERFIL. La devolución de llamada de `TensorBoard()` también tiene opciones para registrar datos adicionales (consulte la documentación para obtener más detalles). Puede hacer clic en el botón de actualización ( ) en la parte superior derecha para que `TensorBoard` actualice los datos, y puede hacer clic en el botón de configuración ( ) para activar la actualización automática y especificar el intervalo de actualización.

Además, TensorFlow ofrece una API de nivel inferior en el paquete `tf.summary`. El siguiente código crea un `SummaryWriter` usando la función `create_file_writer()` y usa este escritor como un contexto de Python para registrar escalares, histogramas, imágenes, audio y texto, los cuales luego se pueden visualizar usando `TensorBoard`:

```
test_logdir = get_run_logdir() escritor =
tf.summary.create_file_writer(str(test_logdir)) con escritor.as_default(): para
el paso en el rango(1, 1000 + 1):
```

```
    tf.summary.scalar("my_scalar", np.sin(paso / 10),
```

```

paso = paso)

    data = (np.random.randn(100) + 2) * se hace más      paso / 100 #
grande
    tf.summary.histogram("my_hist", data, buckets=50,
paso = paso)

    imágenes = np.random.rand(2, 32, 32, 3) * # se vuelve más      paso / 1000
brillante
    tf.summary.image("mis_imagenes", imágenes, paso=paso)

    textos = ["El paso es " + str(paso ** 2) + str(paso), "Su cuadrado es
" 2)] tf.summary.text("mi_texto",
textos, paso=paso)

    onda_seno = tf.math.sin(tf.range(12000) / 48000 * 2
* np.pi * paso) audio =
        tf.reshape(tf.cast(sine_wave, tf.float32), [1, -1, 1]) tf.summary.audio("my_audio",
audio,
        sample_rate=48000, paso =paso)

```

Si ejecuta este código y hace clic en el botón Actualizar en TensorBoard, verá aparecer varias pestañas: IMÁGENES, AUDIO, DISTRIBUCIONES, HISTOGRAMAS y TEXTO. Intente hacer clic en la pestaña IMÁGENES y use el control deslizante sobre cada imagen para ver las imágenes en diferentes pasos de tiempo.

De manera similar, vaya a la pestaña AUDIO e intente escuchar el audio en diferentes pasos de tiempo. Como puede ver, TensorBoard es una herramienta útil incluso más allá de TensorFlow o el aprendizaje profundo.

## CONSEJO

Puede compartir sus resultados en línea publicándolos en <https://tensorboard.dev>. Para esto, simplemente ejecute ! tensorboard dev upload --logdir ./my\_logs. La primera vez te pedirá que aceptes los términos y condiciones y te autentiques. Luego se cargarán sus registros y obtendrá un enlace permanente para ver sus resultados en una interfaz de TensorBoard.

Resumamos lo que ha aprendido hasta ahora en este capítulo: ahora sabe de dónde provienen las redes neuronales, qué es un MLP y cómo puede usarlo para clasificación y regresión, cómo usar la API secuencial de Keras para crear MLP y cómo utilice la API funcional o la API de subclase para crear arquitecturas de modelos más complejas (incluidos modelos amplios y profundos, así como modelos con múltiples entradas y salidas). También aprendió cómo guardar y restaurar un modelo y cómo usar devoluciones de llamada para puntos de control, detención anticipada y más. Finalmente, aprendiste a usar TensorBoard para visualización. ¡Ya puedes seguir adelante y utilizar redes neuronales para abordar muchos problemas! Sin embargo, quizás se pregunte cómo elegir la cantidad de capas ocultas, la cantidad de neuronas en la red y todos los demás hiperparámetros. Veamos esto ahora.

## Ajuste fino de los hiperparámetros de la red neuronal

La flexibilidad de las redes neuronales es también uno de sus principales inconvenientes: hay muchos hiperparámetros que modificar. No sólo puedes usar cualquier arquitectura de red imaginable, sino que incluso en un MLP básico puedes cambiar el número de capas, el número de neuronas y el tipo de función de activación a usar en cada capa, la lógica de inicialización del peso, el tipo de optimizador a uso, su tasa de aprendizaje, el tamaño del lote y más. ¿Cómo saber qué combinación de hiperparámetros es la mejor para su tarea?

Una opción es convertir su modelo Keras en un estimador Scikit-Learn y luego usar GridSearchCV o RandomizedSearchCV para ajustar los hiperparámetros, como lo hizo en el Capítulo 2. Para esto, puede **usar** las clases contenedoras KerasRegressor y KerasClassifier de SciKeras. biblioteca (ver <https://github.com/adriangb/scikeras> para más detalles). Sin embargo, hay una manera mejor: puede usar la biblioteca Keras Tuner , que es una biblioteca de ajuste de hiperparámetros para los modelos Keras. Ofrece varios

estrategias de ajuste, es altamente personalizable y tiene una excelente integración con TensorBoard. Veamos cómo usarlo.

Si siguió las instrucciones de instalación en <https://homl.info/install> para ejecutar todo localmente, entonces ya tienes Keras Tuner instalado, pero si estás usando Colab, necesitarás ejecutar %pip install -q -U keras-tuner. A continuación, importe keras\_tuner, generalmente como kt, luego escriba una función que construya, compile y devuelva un modelo de Keras. La función debe tomar un objeto kt.HyperParameters como argumento, que puede usar para definir hiperparámetros (enteros, flotantes, cadenas, etc.) junto con su rango de valores posibles, y estos hiperparámetros pueden usarse para construir y compilar el modelo. . Por ejemplo, la siguiente función crea y compila un MLP para clasificar imágenes MNIST de moda, utilizando hiperparámetros como el número de capas ocultas (n\_hidden), el número de neuronas por capa (n\_neurons), la tasa de aprendizaje (learning\_rate) y el tipo. del optimizador a utilizar (optimizador):

```
importar keras_tuner como kt

def build_model(hp): n_hidden
    = hp.Int("n_hidden", min_value=0, max_value=8, default=2) n_neurons =
        hp.Int("n_neurons", min_value=16, max_value=256) tasa_de_aprendizaje
    = hp.Flotador("tasa_de_aprendizaje", valor_min=1e- 4, valor_max=1e-2,
                    muestreo="log")
    optimizador = hp.Choice("optimizador", valores=["sgd", "adam"]) si
    optimizador
    == "sgd":
        optimizador =
            tf.keras.optimizers.SGD(tasa_de_aprendizaje=tasa_de_aprendizaje)
    demás:
        optimizador =
            tf.keras.optimizers.Adam(tasa_de_aprendizaje=tasa_de_aprendizaje)

    modelo = tf.keras.Sequential()
    model.add(tf.keras.layers.Flatten())
```

```
para _ en rango(n_hidden):
    model.add(tf.keras.layers.Dense(n_neurons, activación="relu"))

model.add(tf.keras.layers.Dense(10, activación="softmax"))

model.compile(loss="sparse_categorical_crossentropy", optimizador=optimizador,
               métricas=["precisión"])
modelo de devolución
```

La primera parte de la función define los hiperparámetros. Por ejemplo, `hp.Int("n_hidden", min_value=0, max_value=8, default=2)` comprueba si un hiperparámetro llamado "n\_hidden" ya está presente en el objeto `HyperParameters` `hp` y, de ser así, devuelve su valor. De lo contrario, registra un nuevo hiperparámetro entero llamado "n\_hidden", cuyos valores posibles oscilan entre 0 y 8 (inclusive), y devuelve el valor predeterminado, que es 2 en este caso (cuando el valor predeterminado no está establecido, entonces `min_value` es devuelto). El hiperparámetro "n\_neurons" se registra de forma similar. El hiperparámetro "learning\_rate" está registrado y desde `sampling="log"`, como un flotador que oscila entre  $10^{-4}$  y  $10^{-2}$ , y 10, se muestrean por igual tasas de aprendizaje de todas las escalas. Por último, el hiperparámetro del optimizador se registra con dos valores posibles: "sgd" o "adam" (el valor predeterminado es el primero, que en este caso es "sgd"). Dependiendo del valor del optimizador, creamos un optimizador SGD o un optimizador Adam con la tasa de aprendizaje dada.

La segunda parte de la función simplemente construye el modelo utilizando los valores de los hiperparámetros. Crea un modelo secuencial comenzando con una capa Aplanar, seguida del número solicitado de capas ocultas (según lo determinado por el hiperparámetro `n_hidden`) usando la función de activación ReLU y una capa de salida con 10 neuronas (una por clase) usando la función de activación softmax. . Por último, la función compila el modelo y lo devuelve.

Ahora, si desea realizar una búsqueda aleatoria básica, puede crear un sintonizador `kt.RandomSearch`, pasando la función `build_model` al constructor y llamar al método `search()` del sintonizador:

```
random_search_tuner = kt.RandomSearch(
    build_model, Objective="val_accuracy", max_trials=5, overwrite=True,
    directorio="my_fashion_mnist",
    project_name="my_rnd_search", seed=42)
random_search_tuner.search(X_train, y_train, epochs=10,
                            validation_data=(X_valid,
                            y_valid))
```

El sintonizador `RandomSearch` primero llama a `build_model()` una vez con un objeto `Hyperparameters` vacío, solo para recopilar todas las especificaciones de hiperparámetros. Luego, en este ejemplo, realiza 5 pruebas; para cada prueba, crea un modelo utilizando hiperparámetros muestreados aleatoriamente dentro de sus respectivos rangos, luego entrena ese modelo durante 10 épocas y lo guarda en un subdirectorio del directorio `my_fashion_mnist/my_rnd_search`. Como `overwrite=True`, el directorio `my_rnd_search` se elimina antes de que comience el entrenamiento. Si ejecuta este código por segunda vez pero con `overwrite=False` y `max_trials=10`, el sintonizador continuará afinando donde lo dejó, ejecutando 5 pruebas más: esto significa que no tiene que ejecutar todas las pruebas de una sola vez. Por último, dado que el objetivo está establecido en `"val_accuracy"`, el sintonizador prefiere modelos con una mayor precisión de validación, por lo que una vez que el sintonizador haya terminado de buscar, podrá obtener los mejores modelos como este:

```
top3_models =
random_search_tuner.get_best_models(num_models=3) best_model
= top3_models[0]
```

También puedes llamar a `get_best_hyperparameters()` para obtener los `kt.HyperParameters` de los mejores modelos:

```
>>> top3_params =
random_search_tuner.get_best_hyperparameters(num_trials=3) >>>
top3_params[0].values # mejores valores de hiperparámetros {'n_hidden': 5,
'n_neurons': 70,
'tasa_de_aprendizaje': 0.00041268008323824807,
'optimizador': 'adam'}
```

Cada sintonizador es guiado por un llamado oráculo: antes de cada prueba, el sintonizador le pide al oráculo que le diga cuál debería ser la próxima prueba. El sintonizador RandomSearch utiliza RandomSearchOracle, que es bastante básico: simplemente elige la siguiente prueba al azar, como vimos anteriormente. Dado que el oráculo realiza un seguimiento de todas las pruebas, puede pedirle que le proporcione la mejor y puede mostrar un resumen de esa prueba:

```
>>> mejor_prueba =
random_search_tuner.oracle.get_best_trials(num_trials=1)[0] >>>
mejor_prueba.summary()
Resumen de la
prueba
Hiperparámetros:
n_hidden: 5
n_neurons: 70 learning_rate:
0.00041268008323824807 optimizador: adam Puntuación: 0.8736000061035156
```

Esto muestra los mejores hiperparámetros (como antes), así como la precisión de la validación. También puedes acceder a todas las métricas directamente:

```
>>> best_trial.metrics.get_last_value("val_accuracy") 0.8736000061035156
```

Si está satisfecho con el rendimiento del mejor modelo, puede continuar entrenándolo durante algunas épocas en el conjunto de entrenamiento completo. (`X_train_full` e `y_train_full`), luego evaluarlo en el conjunto de prueba e implementarlo en producción (consulte [el Capítulo 19](#)):

```
best_model.fit(X_train_full, y_train_full, epochs=10) test_loss, test_accuracy
= best_model.evaluate(X_test, y_test)
```

En algunos casos, es posible que desee ajustar los hiperparámetros de preprocessamiento de datos o los argumentos model.fit(), como el tamaño del lote. Para esto, debes usar una técnica ligeramente diferente: en lugar de escribir una función build\_model(), debes subclásificar la clase kt.HyperModel y definir dos métodos, build() y fit(). El método build() hace exactamente lo mismo que la función build\_model(). El método fit() toma un objeto HyperParameters y un modelo compilado como argumento, así como todos los argumentos model.fit(), ajusta el modelo y devuelve el objeto History. Fundamentalmente, el método fit() puede utilizar hiperparámetros para decidir cómo preprocessar los datos, ajustar el tamaño del lote y más. Por ejemplo, la siguiente clase construye el mismo modelo que antes, con los mismos hiperparámetros, pero también utiliza un hiperparámetro booleano de "normalización" para controlar si se estandarizan o no los datos de entrenamiento al

```
clase MyClassificationHyperModel (kt.HyperModel):
    def build (self, hp):
        devuelve build_model
            (hp)

    def ajuste(self, hp, modelo, X, y, **kwargs):
        if hp.Boolean("normalize"):
            norm_layer
                = tf.keras.layers.Normalization()
            X = norma_capa(X)
        devolver modelo.fit(X, y, **kwargs)
```

Luego puede pasar una instancia de esta clase al sintonizador de su elección, en lugar de pasar la función build\_model. Por ejemplo, creemos un sintonizador kt.Hyperband basado en una instancia de MyClassificationHyperModel:

```

hyperband_tuner =
    kt.Hyperband( MyClassificationHyperModel(), Objective="val_accuracy",
semilla=42,
    max_epochs=10, factor=3, hyperband_iterations=2, overwrite=True,
    directorio="my_fashion_mnist",
nombre_proyecto="hiperbanda")

```

Este sintonizador es similar a la clase HalvingRandomSearchCV que analizamos en el [Capítulo 2](#): comienza entrenando muchos modelos diferentes durante algunas épocas, luego elimina los peores modelos y mantiene solo los modelos de 1/factor superior (es decir, el tercio superior en este caso). , repitiendo este proceso de selección hasta quedar un solo modelo. El argumento max\_epochs controla el número máximo de épocas para las que se entrenará el mejor modelo. En este caso, todo el proceso se repite dos veces (hyperband\_iterations=2). El número total de épocas de entrenamiento en todos los modelos para cada iteración de hiperbanda es de aproximadamente  $\text{max\_epochs} * (\log(\text{max\_epochs}) / \log(\text{factor})) ^ 2$ , por lo que en este ejemplo son aproximadamente 44 épocas. Los demás argumentos son los mismos que para kt.RandomSearch.

Ejecutemos el sintonizador Hyperband ahora. Usaremos la devolución de llamada de TensorBoard, esta vez apuntando al directorio de registro raíz (el sintonizador se encargará de usar un subdirectorio diferente para cada prueba), así como una devolución de llamada EarlyStopping:

```

root_logdir = Ruta(hyperband_tuner.project_dir) / "tensorboard"

tensorboard_cb =
tf.keras.callbacks.TensorBoard(root_logdir) early_stopping_cb
=
tf.keras.callbacks.EarlyStopping(patience=2)
hyperband_tuner.search(X_train, y_train, epochs=10,
validation_data=(X_valid, y_valid), devoluciones
de llamada=[early_stopping_cb,
tensorboard_cb])

```

Ahora, si abre TensorBoard y apunta --logdir al directorio my\_fashion\_mnist/hyperband/tensorboard , verá todos los resultados de la prueba a medida que se desarrollan. Asegúrese de visitar la pestaña HPARAMS: contiene un resumen de todas las combinaciones de hiperparámetros que se probaron, junto con las métricas correspondientes. Observe que hay tres pestañas dentro de la pestaña HPARAMS: una vista de tabla, una vista de coordenadas paralelas y una vista de matriz de diagrama de dispersión. En la parte inferior del panel izquierdo, desmarque todas las métricas excepto validation.epoch\_accuracy: esto hará que los gráficos sean más claros. En la vista de coordenadas paralelas, intente seleccionar un rango de valores altos en la columna validation.epoch\_accuracy: esto filtrará solo las combinaciones de hiperparámetros que alcanzaron un buen rendimiento.

Haga clic en una de las combinaciones de hiperparámetros y las curvas de aprendizaje correspondientes aparecerán en la parte inferior de la página. Tómate un tiempo para revisar cada pestaña; esto le ayudará a comprender el efecto de cada hiperparámetro en el rendimiento, así como las interacciones entre los hiperparámetros.

Hyperband es más inteligente que la búsqueda aleatoria pura en la forma en que asigna recursos, pero en esencia aún explora el espacio de hiperparámetros de forma aleatoria; es rápido, pero tosco. Sin embargo, Keras Tuner también incluye un sintonizador kt.BayesianOptimization: este algoritmo aprende gradualmente qué regiones del espacio de hiperparámetros son más prometedoras ajustando un modelo probabilístico llamado proceso gaussiano. Esto le permite acercarse gradualmente a los mejores hiperparámetros. La desventaja es que el algoritmo tiene sus propios hiperparámetros: alfa representa el nivel de ruido que espera en las medidas de rendimiento en todas las pruebas (el valor predeterminado es 10), y beta<sup>4</sup> especifica cuánto desea que explore el algoritmo, en lugar de simplemente explotar lo conocido. buenas regiones del espacio de hiperparámetros (el valor predeterminado es 2.6). Aparte de eso, este sintonizador se puede utilizar igual que los anteriores:

```
bayesian_opt_tuner =
    kt.BayesianOptimization( MyClassificationHyperModel(), objetivo="val_accuracy",
```

```

semilla=42,
    max_trials=10, alfa=1e-4, beta=2.6,
    sobreescritura=True, directorio="my_fashion_mnist",
    nombre_proyecto="bayesian_opt")
bayesian_opt_tuner.search([...])

```

El ajuste de hiperparámetros sigue siendo un área activa de investigación y se están explorando muchos otros enfoques. Por ejemplo, consulte el excelente [artículo de DeepMind de 2017](#)<sup>20</sup>, donde los autores utilizaron un algoritmo evolutivo para optimizar conjuntamente una población de modelos y sus hiperparámetros. Google también ha utilizado un enfoque evolutivo, no sólo para buscar hiperparámetros sino también para explorar todo tipo de arquitecturas de modelos: potencia su servicio AutoML en Google Vertex AI (ver Capítulo 19). El término AutoML se refiere a cualquier sistema que se encarga de gran parte del flujo de trabajo de ML.

¡Los algoritmos evolutivos incluso se han utilizado con éxito para entrenar redes neuronales individuales, reemplazando el omnipresente descenso de gradiente! Para ver un ejemplo, consulte la [publicación de 2017](#) por Uber donde los autores presentan su técnica de Neuroevolución Profunda .

Pero a pesar de todo este emocionante progreso y todas estas herramientas y servicios, todavía ayuda tener una idea de qué valores son razonables para cada hiperparámetro para poder construir un prototipo rápido y restringir el espacio de búsqueda. Las siguientes secciones proporcionan pautas para elegir el número de capas y neuronas ocultas en un MLP y para seleccionar buenos valores para algunos de los principales hiperparámetros.

## Número de capas ocultas

Para muchos problemas, puede comenzar con una única capa oculta y obtener resultados razonables. En teoría, un MLP con una sola capa oculta puede modelar incluso las funciones más complejas, siempre que tenga suficientes neuronas. Pero para problemas complejos, las redes profundas tienen una eficiencia de parámetros mucho mayor que las superficiales: pueden modelar funciones complejas utilizando exponencialmente menos neuronas que las superficiales.

nets, lo que les permite alcanzar un rendimiento mucho mejor con la misma cantidad de datos de entrenamiento.

Para entender por qué, supongamos que le piden que dibuje un bosque usando algún software de dibujo, pero tiene prohibido copiar y pegar nada. Requeriría una enorme cantidad de tiempo: habría que dibujar cada árbol individualmente, rama por rama, hoja por hoja. Si, en cambio, pudieras dibujar una hoja, copiarla y pegarla para dibujar una rama, luego copiar y pegar esa rama para crear un árbol y, finalmente, copiar y pegar este árbol para hacer un bosque, terminarías en poco tiempo.

Los datos del mundo real a menudo están estructurados de una manera jerárquica, y las redes neuronales profundas aprovechan automáticamente este hecho: las capas ocultas inferiores modelan estructuras de bajo nivel (por ejemplo, segmentos de línea de diversas formas y orientaciones), las capas ocultas intermedias combinan estas estructuras de bajo nivel. -estructuras de nivel para modelar estructuras de nivel intermedio (por ejemplo, cuadrados, círculos), y las capas ocultas más altas y la capa de salida combinan estas estructuras intermedias para modelar estructuras de alto nivel (por ejemplo, caras).

Esta arquitectura jerárquica no solo ayuda a las DNN a converger más rápido hacia una buena solución, sino que también mejora su capacidad de generalizar a nuevos conjuntos de datos. Por ejemplo, si ya entrenó un modelo para reconocer rostros en imágenes y ahora desea entrenar una nueva red neuronal para reconocer peinados, puede iniciar el entrenamiento reutilizando las capas inferiores de la primera red. En lugar de inicializar aleatoriamente los pesos y sesgos de las primeras capas de la nueva red neuronal, puede inicializarlos con los valores de los pesos y sesgos de las capas inferiores de la primera red. De esta manera la red no tendrá que aprender desde cero todas las estructuras de bajo nivel que ocurren en la mayoría de las imágenes; sólo tendrá que aprender las estructuras de nivel superior (por ejemplo, peinados). Esto se llama aprendizaje por transferencia.

En resumen, para muchos problemas puedes empezar con sólo una o dos capas ocultas y la red neuronal funcionará bien. Por ejemplo, puede alcanzar fácilmente una precisión superior al 97 % en el conjunto de datos MNIST

usando solo una capa oculta con unos pocos cientos de neuronas, y con una precisión superior al 98% usando dos capas ocultas con la misma cantidad total de neuronas, en aproximadamente la misma cantidad de tiempo de entrenamiento. Para problemas más complejos, puede aumentar el número de capas ocultas hasta que comience a sobreajustar el conjunto de entrenamiento. Las tareas muy complejas, como la clasificación de imágenes grandes o el reconocimiento de voz, normalmente requieren redes con docenas de capas (o incluso cientos, pero no completamente conectadas, como verá en el Capítulo 14), **y** necesitan una enorme cantidad de datos de entrenamiento. Rara vez será necesario entrenar dichas redes desde cero: es mucho más común reutilizar partes de una red de última generación previamente entrenada que realiza una tarea similar. Entonces el entrenamiento será mucho más rápido y requerirá muchos menos datos (lo discutiremos en el [Capítulo 11](#)).

## Número de neuronas por capa oculta

La cantidad de neuronas en las capas de entrada y salida está determinada por el tipo de entrada y salida que requiere su tarea. Por ejemplo, la tarea MNIST requiere  $28 \times 28 = 784$  entradas y 10 neuronas de salida.

En cuanto a las capas ocultas, solía ser común dimensionarlas para formar una pirámide, con cada vez menos neuronas en cada capa; la razón es que muchas características de bajo nivel pueden fusionarse en muchas menos características de alto nivel. Una red neuronal típica para MNIST podría tener 3 capas ocultas, la primera con 300 neuronas, la segunda con 200 y la tercera con 100. Sin embargo, esta práctica se ha abandonado en gran medida porque parece que usar el mismo número de neuronas en todas las capas ocultas. las capas funcionan igual de bien en la mayoría de los casos, o incluso mejor; Además, solo hay un hiperparámetro para ajustar, en lugar de uno por capa. Dicho esto, dependiendo del conjunto de datos, a veces puede resultar útil hacer que la primera capa oculta sea más grande que las demás.

Al igual que con la cantidad de capas, puede intentar aumentar la cantidad de neuronas gradualmente hasta que la red comience a sobreajustarse. Alternativamente, puedes intentar construir un modelo con un poco más de capas y neuronas.

de lo que realmente necesita, luego utilice la detención anticipada y otras técnicas de regularización para evitar que se sobreajuste demasiado.

Vincent Vanhoucke, científico de Google, lo ha denominado el enfoque de los “pantalones elásticos”: en lugar de perder el tiempo buscando pantalones que combinen perfectamente con su talla, simplemente use pantalones elásticos grandes que se encojan hasta el tamaño correcto. Con este enfoque, evita capas de cuello de botella que podrían arruinar su modelo. De hecho, si una capa tiene muy pocas neuronas, no tendrá suficiente poder de representación para preservar toda la información útil de las entradas (por ejemplo, una capa con dos neuronas sólo puede generar datos 2D, por lo que si recibe datos 3D como entrada, algunas se perderá la información). No importa cuán grande y poderosa sea el resto de la red, esa información nunca será recuperada.

CONSEJO

En general, obtendrá más beneficios si aumenta el número de capas en lugar del número de neuronas por capa.

## Tasa de aprendizaje, tamaño del lote y otros Hiperparámetros

La cantidad de capas ocultas y neuronas no son los únicos hiperparámetros que puede modificar en un MLP. Éstos son algunos de los más importantes, así como consejos sobre cómo configurarlos:

### Tasa de aprendizaje

La tasa de aprendizaje es posiblemente el hiperparámetro más importante.

En general, la tasa de aprendizaje óptima es aproximadamente la mitad de la tasa de aprendizaje máxima (es decir, la tasa de aprendizaje por encima de la cual el algoritmo de entrenamiento diverge, como vimos en el [Capítulo 4](#)). Una forma de encontrar una buena tasa de aprendizaje es entrenar el modelo durante unos cientos de iteraciones, comenzando con una tasa de aprendizaje muy baja (p. ej., 10) y aumentándola gradualmente hasta un valor muy grande (p. ej., 10). Esto se hace multiplicando la tasa de aprendizaje por un factor constante en cada

iteración (por ejemplo, por  $(10/10)^{5,1/500}$  para pasar de 10 a 10 en 500 iteraciones). Si traza la pérdida en función de la tasa de aprendizaje (usando una escala logarítmica para la tasa de aprendizaje), debería ver que disminuye al principio. Pero después de un tiempo, la tasa de aprendizaje será demasiado grande, por lo que la pérdida se disparará nuevamente: la tasa de aprendizaje óptima será un poco más baja que el punto en el que la pérdida comienza a aumentar (generalmente alrededor de 10 veces menor que el punto de inflexión). ). Luego puedes reinicializar tu modelo y entrenarlo normalmente usando esta buena tasa de aprendizaje. Veremos más técnicas de optimización de la tasa de aprendizaje en el [Capítulo 11](#).

## Optimizador

También es bastante importante elegir un optimizador mejor que el antiguo descenso de gradiente de mini lotes (y ajustar sus hiperparámetros). Examinaremos varios optimizadores avanzados en el [Capítulo 11](#).

## Tamaño del lote

El tamaño del lote puede tener un impacto significativo en el rendimiento y el tiempo de entrenamiento de su modelo. El principal beneficio de utilizar lotes de gran tamaño es que los aceleradores de hardware como las GPU pueden procesarlos de manera eficiente (consulte [el Capítulo 19](#)), por lo que el algoritmo de entrenamiento verá más instancias por segundo. Por lo tanto, muchos investigadores y profesionales recomiendan utilizar el tamaño de lote más grande que pueda caber en la RAM de la GPU. Sin embargo, hay un problema: en la práctica, los tamaños de lote grandes a menudo conducen a inestabilidades en el entrenamiento, especialmente al comienzo del mismo, y es posible que el modelo resultante no se generalice tan bien como un modelo entrenado con un tamaño de lote pequeño. En abril de 2018, Yann LeCun incluso tuiteó: “Los amigos no permiten que sus amigos usen mini-[21](#) lotes de más de 32”, citando un [artículo de 2018](#) por Dominic Masters y Carlo Luschi, que concluyó que era preferible utilizar lotes pequeños (de 2 a 32) porque los lotes pequeños conducían a mejores modelos en menos tiempo de entrenamiento. Sin embargo, otras investigaciones apuntan en la dirección opuesta. Por ejemplo, en 2017, artículos de [Elad Hoffer et al.](#) y [Priya Goyal et al.](#) demostró que era

Es posible utilizar tamaños de lotes muy grandes (hasta 8192) junto con varias técnicas, como calentar la tasa de aprendizaje (es decir, comenzar el entrenamiento con una tasa de aprendizaje pequeña y luego aumentarla, como se analiza en el Capítulo 11) y obtener resultados muy cortos . tiempos de entrenamiento, sin ningún vacío de generalización. Por lo tanto, una estrategia es intentar utilizar un tamaño de lote grande, con un calentamiento de la tasa de aprendizaje, y si el entrenamiento es inestable o el rendimiento final es decepcionante, intente utilizar un tamaño de lote pequeño.

### Función de activación

Discutimos cómo elegir la función de activación anteriormente en este capítulo: en general, la función de activación ReLU será una buena opción predeterminada para todas las capas ocultas, pero para la capa de salida realmente depende de su tarea.

### Número de iteraciones

En la mayoría de los casos, no es necesario modificar el número de iteraciones de entrenamiento: en su lugar, simplemente utilice la parada anticipada.

#### CONSEJO

La tasa de aprendizaje óptima depende de los otros hiperparámetros (especialmente el tamaño del lote), por lo que si modifica cualquier hiperparámetro, asegúrese de actualizar también la tasa de aprendizaje.

Para conocer más prácticas recomendadas sobre el ajuste de los hiperparámetros de la red neuronal, consulte el excelente artículo de 24 por Leslie Smith.

Con esto concluye nuestra introducción a las redes neuronales artificiales y su implementación con Keras. En los próximos capítulos, discutiremos técnicas para entrenar redes muy profundas. También exploraremos cómo personalizar modelos usando la API de nivel inferior de TensorFlow y cómo cargar y preprocesar datos de manera eficiente usando la API tf.data. Y nosotros

se sumergirá en otras arquitecturas de redes neuronales populares: redes neuronales convolucionales para procesamiento de imágenes, redes neuronales recurrentes y transformadores para datos y texto secuenciales, codificadores automáticos para aprendizaje de representación y redes generativas adversarias para modelar y generar datos.

25

## Ejercicios

1. El [patio de juegos de TensorFlow](#) es un práctico simulador de redes neuronales creado por el equipo de TensorFlow. En este ejercicio, entrenará varios clasificadores binarios con solo unos pocos clics y modificará la arquitectura del modelo y sus hiperparámetros para obtener cierta intuición sobre cómo funcionan las redes neuronales y qué hacen sus hiperparámetros. Tómate un tiempo para explorar lo siguiente:
  - a. Los patrones aprendidos por una red neuronal. Intenta entrenar el red neuronal predeterminada haciendo clic en el botón Ejecutar (arriba a la izquierda). Observe cómo encuentra rápidamente una buena solución para la tarea de clasificación. Las neuronas de la primera capa oculta han aprendido patrones simples, mientras que las neuronas de la segunda capa oculta han aprendido a combinar los patrones simples de la primera capa oculta en patrones más complejos. En general, cuantas más capas haya, más complejos pueden ser los patrones.
  - b. Funciones de activación. Intente reemplazar la función de activación tanh con una función de activación ReLU y entrene la red nuevamente. Observe que encuentra una solución aún más rápido, pero esta vez los límites son lineales. Esto se debe a la forma de la función ReLU.
- C. El riesgo de mínimos locales. Modifique la arquitectura de la red para que tenga solo una capa oculta con tres neuronas. Entrénelo varias veces (para restablecer los pesos de la red, haga clic en el botón Restablecer al lado del botón Reproducir). Observe que el tiempo de entrenamiento

varía mucho y, a veces, incluso se queda estancado en un mínimo local.

d. ¿Qué sucede cuando las redes neuronales son demasiado pequeñas? Eliminar una neurona para conservar sólo dos. Observe que la red neuronal ahora es incapaz de encontrar una buena solución, incluso si lo intenta varias veces. El modelo tiene muy pocos parámetros y sistemáticamente no se ajusta al conjunto de entrenamiento.

mi. ¿Qué sucede cuando las redes neuronales son lo suficientemente grandes? Establezca el número de neuronas en ocho y entrene la red varias veces. Observe que ahora es consistentemente rápido y nunca se atasca. Esto pone de relieve un hallazgo importante en la teoría de las redes neuronales: las redes neuronales grandes rara vez se quedan atrapadas en mínimos locales, e incluso cuando lo hacen, estos óptimos locales suelen ser casi tan buenos como el óptimo global. Sin embargo, todavía pueden quedarse estancados durante mucho tiempo.

F. El riesgo de que los gradientes desaparezcan en las redes profundas. Seleccione el conjunto de datos en espiral (el conjunto de datos de la parte inferior derecha en "DATOS") y cambie la arquitectura de la red para que tenga cuatro capas ocultas con ocho neuronas cada una. Tenga en cuenta que el entrenamiento lleva mucho más tiempo y, a menudo, se estanca durante largos períodos de tiempo. Observe también que las neuronas de las capas más altas (a la derecha) tienden a evolucionar más rápido que las neuronas de las capas más bajas (a la izquierda). Este problema, llamado problema de gradientes de fuga , se puede aliviar con una mejor inicialización de peso y otras técnicas, mejores optimizadores (como AdaGrad o Adam) o normalización por lotes (que se analiza en el Capítulo 11 ).

gramo. Ve más allá. Tómese aproximadamente una hora para jugar con otros parámetros y tener una idea de lo que hacen, para desarrollar una comprensión intuitiva sobre las redes neuronales.

2. Dibuja una ANN utilizando las neuronas artificiales originales (como las de la Figura 10-3) que calcule  $A \oplus B$  (donde  $\oplus$  representa la operación XOR).  
Sugerencia:  $A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$ .
3. ¿Por qué generalmente es preferible utilizar una regresión logística? clasificador en lugar de un perceptrón clásico (es decir, una sola capa de unidades lógicas de umbral entrenadas utilizando el algoritmo de entrenamiento del perceptrón)? ¿Cómo se puede modificar un perceptrón para que sea equivalente a un clasificador de regresión logística?
4. ¿Por qué la función de activación sigmoidea fue un ingrediente clave en el entrenamiento de los primeros MLP?
5. Nombra tres funciones de activación populares. ¿Puedes dibujarlos?
6. Suponga que tiene un MLP compuesto por una capa de entrada con 10 neuronas de paso, seguida de una capa oculta con 50 neuronas artificiales y, finalmente, una capa de salida con 3 neuronas artificiales. Todas las neuronas artificiales utilizan la función de activación ReLU.
- ¿Cuál es la forma de la matriz de entrada  $X$ ?
  - ¿Cuáles son las formas de la matriz de pesos  $W$  de la capa oculta?  $h$  y el vector de sesgo  $b_h$ ?
  - ¿Cuáles son las formas de la matriz de pesos  $W$  de la capa de salida?  $oh$  y el vector de sesgo  $b_{oh}$ ?
  - ¿Cuál es la forma de la matriz de salida  $Y$  de la red?
- mi. Escribe la ecuación que calcula la salida de la red y  $b$ . matriz  $Y$  en función de  $X$ ,  $W$   $quién$ ,  $h$ ,  $oh$
7. ¿Cuántas neuronas necesitas en la capa de salida si quieres clasificar el correo electrónico como spam o jamón? ¿Qué función de activación debería utilizar en la capa de salida? Si, en cambio, desea abordar MNIST, ¿cuántas neuronas necesita en la capa de salida y qué función de activación debería utilizar? ¿Qué tal hacer que su red prediga los precios de la vivienda, como en el Capítulo 2?

8. ¿Qué es la retropropagación y cómo funciona? ¿Cuál es la diferencia entre retropropagación y autodiff en modo inverso?
9. ¿Puedes enumerar todos los hiperparámetros que puedes modificar de forma básica? ¿MLP? Si el MLP se ajusta demasiado a los datos de entrenamiento, ¿cómo se podrían modificar estos hiperparámetros para intentar resolver el problema?
10. Entrene un MLP profundo en el conjunto de datos MNIST (puede cargarlo usando `tf.keras.datasets.mnist.load_data()`). Vea si puede obtener una precisión superior al 98 % ajustando manualmente los hiperparámetros. Intente buscar la tasa de aprendizaje óptima utilizando el enfoque presentado en este capítulo (es decir, aumentando la tasa de aprendizaje exponencialmente, trazando la pérdida y encontrando el punto donde la pérdida se dispara). A continuación, intente ajustar los hiperparámetros usando Keras Tuner con todas las comodidades: guarde los puntos de control, use la parada anticipada y trace curvas de aprendizaje usando TensorBoard.

Las soluciones a estos ejercicios están disponibles al final del cuaderno de este capítulo, en <https://homl.info/colab3>.

---

<sup>1</sup> Puedes obtener lo mejor de ambos mundos estando abierto a las inspiraciones biológicas sin tener miedo de crear modelos biológicamente poco realistas, siempre que funcionen bien.

<sup>2</sup> Warren S. McCulloch y Walter Pitts, “Un cálculo lógico de las ideas inmanente en la actividad nerviosa”, *The Bulletin of Mathematical Biology* 5, no. 4 (1943): 115-113.

<sup>3</sup> En realidad no están unidos, sólo que están tan cerca que pueden rápidamente intercambiar señales químicas.

<sup>4</sup> Imagen de Bruce Blaus ([Creative Commons 3.0](#)). Reproducido de <https://en.wikipedia.org/wiki/Neuron>.

<sup>5</sup> En el contexto del aprendizaje automático, la frase “redes neuronales” generalmente se refiere a ANN, no a BNN.

**6** Dibujo de una laminación cortical de S. Ramon y Cajal (dominio público).

Reproducido de [https://en.wikipedia.org/wiki/Cerebral\\_cortex](https://en.wikipedia.org/wiki/Cerebral_cortex).

**7** Tenga en cuenta que esta solución no es única: cuando los puntos de datos son linealmente separables, hay una infinidad de hiperplanos que pueden separarlos.

**8** Por ejemplo, cuando las entradas son  $(0, 1)$ , la neurona inferior izquierda calcula  $0 \times 1 + 1 \times 1 - 3/2 = -1/2$ , que es negativo, por lo que genera 0. La neurona inferior derecha calcula  $0 \times 1 + 1 \times 1 - 1/2 = 1/2$ , que es positivo, por lo que genera 1. La neurona de salida recibe las salidas de las dos primeras neuronas como entradas, por lo que calcula  $0 \times (-1) + 1 \times 1 - 1/2 = 1/2$ . Esto es positivo, por lo que genera 1.

**9** En la década de 1990, una RNA con más de dos capas ocultas se consideraba profundo. Hoy en día, es común ver RNA con docenas de capas, o incluso cientos, por lo que la definición de "profunda" es bastante confusa.

**10** David Rumelhart et al., "Learning Internal Representations by Error Propagation" (informe técnico del Centro de Información Técnica de Defensa, septiembre de 1985).

**11** Las neuronas biológicas parecen implementar una forma aproximadamente sigmoidea (en forma de S) función de activación, por lo que los investigadores se apoyaron en las funciones sigmoideas durante mucho tiempo. Pero resulta que ReLU generalmente funciona mejor en ANN. Este es uno de los casos en los que la analogía biológica quizás induciría a error.

**12** Proyecto ONEIROS (Robot inteligente neuroelectrónico de composición abierta) Sistema operativo). Chollet se unió a Google en 2015, donde continúa liderando el proyecto Keras.

**13** La API de PyTorch es bastante similar a la de Keras, por lo que una vez que conoces Keras, no es difícil cambiar a PyTorch, si alguna vez lo deseas. La popularidad de PyTorch creció exponencialmente en 2018, en gran parte gracias a su simplicidad y excelente documentación, que no eran los principales puntos fuertes de TensorFlow 1.x en aquel entonces. Sin embargo, TensorFlow 2 es tan simple como PyTorch, en parte porque adoptó Keras como su API oficial de alto nivel y también porque los desarrolladores han simplificado y limpiado enormemente el resto de la API. La documentación también se ha reorganizado por completo y ahora es mucho más fácil encontrar lo que necesita. De manera similar, las principales debilidades de PyTorch (por ejemplo, portabilidad limitada y falta de análisis de gráficos computacionales) se han abordado en gran medida en PyTorch 1.0. La competencia sana parece beneficiosa para todos.

**14** También puedes usar `tf.keras.utils.plot_model()` para generar una imagen de tu modelo.

**15** Heng-Tze Cheng et al., "Wide & Deep Learning for Recommender Systems", Actas del primer taller sobre aprendizaje profundo para

Sistemas de recomendación (2016): 7–10.

- 16 La ruta corta también se puede utilizar para proporcionar funciones diseñadas manualmente para la red neuronal.
- 17 Los modelos Keras tienen un atributo de salida, por lo que no podemos usar ese nombre para la capa de salida principal, por eso le cambiamos el nombre a `main_output`.
- 18 Actualmente, este es el formato predeterminado, pero el equipo de Keras está trabajando en un nuevo formato que puede convertirse en el predeterminado en las próximas versiones, por lo que prefiero configurar el formato explícitamente para que esté preparado para el futuro.
- 19 La hiperbanda es en realidad un poco más sofisticada que la reducción sucesiva a la mitad; ver [el papel](#) por Lisha Li et al., “Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization”, Journal of Machine Learning Research 18 (abril de 2018): 1–52.
- 20 Max Jaderberg et al., “Entrenamiento de redes neuronales basado en la población”, pre impresión de arXiv arXiv:1711.09846 (2017).
- 21 Dominic Masters y Carlo Luschi, “Revisiting Small Batch Training for Deep Neural Networks”, pre impresión de arXiv arXiv:1804.07612 (2018).
- 22 Elad Hoffer et al., “Train Longer, Generalize Better: Closing the Generalization Gap in Large Batch Training of Neural Networks”, Actas de la 31<sup>a</sup> Conferencia Internacional sobre Sistemas de Procesamiento de Información Neural (2017): 1729–1739.
- 23 Priya Goyal et al., “SGD de minibatch grande y preciso: entrenamiento de ImageNet en 1 hora”, pre impresión de arXiv arXiv:1706.02677 (2017).
- 24 Leslie N. Smith, “Un enfoque disciplinado para la hiperactividad de redes neuronales”. Parámetros: Parte 1: Tasa de aprendizaje, tamaño de lote, impulso y caída de peso”, pre impresión de arXiv arXiv:1803.09820 (2018).
- 25 Algunas arquitecturas ANN adicionales se presentan en el cuaderno en línea en <https://homl.info/extranns>.

# Capítulo 11. Entrenamiento de redes neuronales profundas

---

En el Capítulo 10 construyó, entrenó y ajustó sus primeras redes neuronales artificiales. Pero eran redes poco profundas, con sólo unas pocas capas ocultas. ¿Qué sucede si necesita abordar un problema complejo, como detectar cientos de tipos de objetos en imágenes de alta resolución? Es posible que necesites entrenar una RNA mucho más profunda, quizás con 10 capas o muchas más, cada una de las cuales contenga cientos de neuronas, unidas por cientos de miles de conexiones.

Entrenar una red neuronal profunda no es un paseo por el parque. Estos son algunos de los problemas con los que podría encontrarse:

- Es posible que se enfrente al problema de que los gradientes se vuelven cada vez más pequeños o más grandes cuando fluyen hacia atrás a través del DNN durante el entrenamiento. Ambos problemas hacen que sea muy difícil entrenar las capas inferiores.
- Es posible que no tenga suficientes datos de entrenamiento para una red tan grande o que etiquetarlos sea demasiado costoso.
- El entrenamiento puede ser extremadamente lento.
- Un modelo con millones de parámetros correría un grave riesgo de sobreajustar el conjunto de entrenamiento, especialmente si no hay suficientes instancias de entrenamiento o si son demasiado ruidosas.

En este capítulo analizaremos cada uno de estos problemas y presentaremos técnicas para resolverlos. Comenzaremos explorando los problemas de gradientes que desaparecen y explotan y algunas de sus soluciones más populares.

A continuación, veremos el aprendizaje por transferencia y el entrenamiento previo no supervisado, que pueden ayudarle a abordar tareas complejas incluso cuando tenga pocos datos etiquetados. Luego, analizaremos varios optimizadores que pueden acelerar enormemente el entrenamiento de modelos grandes. Finalmente, cubriremos algunas técnicas de regularización populares para redes neuronales grandes.

Con estas herramientas podrás entrenar redes muy profundas. ¡Bienvenido al aprendizaje profundo!

## Los problemas de los gradientes que desaparecen o explotan

Como se analizó en [el Capítulo 10](#), la segunda fase del algoritmo de retropropagación funciona yendo de la capa de salida a la capa de entrada, propagando el gradiente de error a lo largo del camino. Una vez que el algoritmo ha calculado el gradiente de la función de costo con respecto a cada parámetro de la red, utiliza estos gradientes para actualizar cada parámetro con un paso de descenso de gradiente.

Desafortunadamente, los gradientes a menudo se vuelven cada vez más pequeños a medida que el algoritmo avanza hacia las capas inferiores. Como resultado, la actualización del descenso de gradiente deja los pesos de conexión de las capas inferiores prácticamente sin cambios y el entrenamiento nunca converge hacia una buena solución. A esto se le llama el problema de los gradientes fugaces . En algunos casos, puede suceder lo contrario: los gradientes pueden crecer cada vez más hasta que las capas obtienen actualizaciones de peso increíblemente grandes y el algoritmo diverge. Este es el problema de los gradientes explosivos , que surge con mayor frecuencia en las redes neuronales recurrentes (consulte [el Capítulo 15](#)). De manera más general, las redes neuronales profundas sufren de gradientes inestables; diferentes capas pueden aprender a velocidades muy diferentes.

Este desafortunado comportamiento se observó empíricamente hace mucho tiempo y fue una de las razones por las que las redes neuronales profundas fueron abandonadas en su mayoría a principios de la década de 2000. No estaba claro qué causaba que los gradientes fueran tan inestables al entrenar un DNN, pero un [artículo de 2010](#) arrojó algo de luz. de Xavier Glorot y Yoshua Bengio. Los<sup>1</sup> autores encontraron algunos sospechosos, incluida la combinación de la popular función de activación sigmoidea (logística) y la técnica de inicialización de peso que era más popular en ese momento (es decir, una distribución normal con una media de 0 y una desviación estándar de 1). En resumen, demostraron que con esta función de activación y este esquema de inicialización, la varianza de las salidas de cada capa es mucho mayor que la varianza de sus entradas. En el futuro de la red, la varianza sigue aumentando después de cada capa hasta que la función de activación se satura en las capas superiores. En realidad, esta saturación empeora por el hecho de que la función sigmoidea tiene una media de 0,5, no 0 (la función tangente hiperbólica tiene una media de 0 y se comporta ligeramente mejor que la función sigmoidea en redes profundas).

Si observa la función de activación sigmoidea (consulte [la Figura 11-1](#)), puede ver que cuando las entradas se vuelven grandes (negativas o positivas), la función

se satura en 0 o 1, con una derivada extremadamente cercana a 0 (es decir, la curva es plana en ambos extremos). Por lo tanto, cuando se activa la retropropagación, prácticamente no tiene gradiente para propagarse a través de la red, y el poco gradiente que existe se sigue diluyendo a medida que la retropropagación avanza hacia las capas superiores, por lo que realmente no queda nada para las capas inferiores.

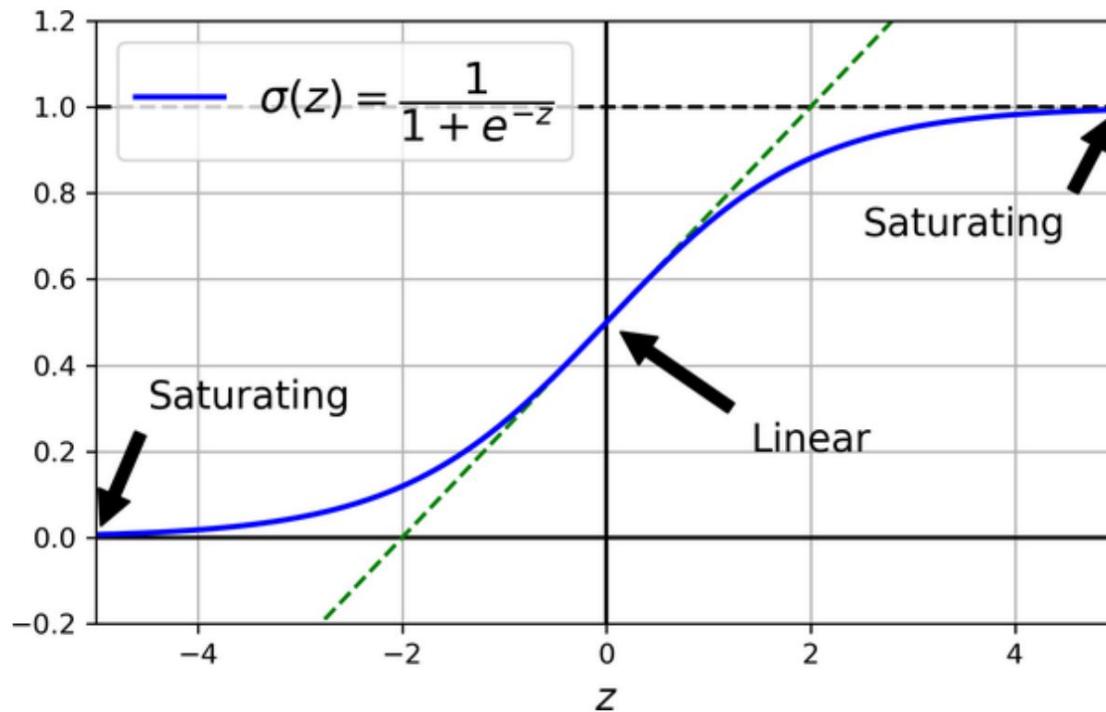


Figura 11-1. Saturación de la función de activación sigmoidea

## Inicialización de Glorot y He

En su artículo, Glorot y Bengio proponen una forma de aliviar significativamente el problema de los gradientes inestables. Señalan que necesitamos que la señal fluya correctamente en ambas direcciones: hacia adelante cuando se hacen predicciones y en dirección inversa cuando se propagan gradientes hacia atrás. No queremos que la señal se apague, ni queremos que explote y se sature. Para que la señal fluya correctamente, los autores argumentan que necesitamos que la varianza de las salidas de cada capa sea igual a la varianza de sus entradas, y necesitamos<sup>2</sup> que los gradientes tengan la misma varianza antes y después de fluir a través de una capa en sentido inverso. dirección (consulte el documento si está interesado en los detalles matemáticos). En realidad, no es posible garantizar ambas cosas a menos que la capa tenga el mismo número de entradas y salidas (estos números se denominan entrada y salida de la capa), pero

Glorot y Bengio propusieron un buen compromiso que ha demostrado funcionar muy bien en la práctica: los pesos de conexión de cada capa deben inicializarse aleatoriamente como se describe en la Ecuación 11-1, donde  $\text{fan} = (\text{fan}_{\text{en}} + \text{fan}_{\text{afuera}}) / 2$ . Esta estrategia de inicialización es llamada inicialización de Xavier o inicialización de Glorot, en honor al primer autor del artículo.

Ecuación 11-1. Inicialización de Glorot (cuando se utiliza la función de activación sigmoidea)

$$\text{Distribución normal con media } 0 \text{ y varianza } \sigma^2 = \frac{1}{\text{fan}_{\text{avg}}}$$

O una distribución uniforme entre  $-\sqrt{3}\sigma$  y  $\sqrt{3}\sigma$ , con  $\sqrt{3}\sigma = \sqrt{\frac{6}{\text{fan}_{\text{avg}}}}$

Si reemplaza un ventilador con un ventilador en la Ecuación 11-1, obtiene una inicialización estratégica que Yann LeCun propuso en los años 1990. Lo llamó inicialización de LeCun. Genevieve Orr y Klaus-Robert Müller incluso lo recomendaron en su libro de 1998 Neural Networks: Tricks of the Trade (Springer). La inicialización de LeCun es equivalente a la inicialización de Glorot cuando  $\text{fan} = \text{fan}_{\text{en}}$ . Los investigadores tardaron más de una década en darse cuenta de la importancia de este truco. El uso de la inicialización de Glorot puede acelerar considerablemente el entrenamiento y es una de las prácticas que condujo al éxito del aprendizaje profundo.

Algunos artículos<sup>3</sup> han proporcionado estrategias similares para diferentes funciones de activación. Estas estrategias difieren sólo por la escala de la varianza y si utilizan ventilador o ventilador como se muestra en la Tabla 11-1 (para el modelo uniforme).

distribución, solo use  $r = \sqrt{3}\sigma / 2$ ). La estrategia de inicialización propuesta para La función de activación de ReLU y sus variantes se denomina inicialización He o inicialización Kaiming, en honor al primer autor del artículo. Para SELU, utilice el método de inicialización de Yann LeCun, preferiblemente con una distribución normal. Cubriremos todas estas funciones de activación en breve.

Tabla 11-1. Parámetros de inicialización para cada tipo de función de activación.

Inicialización	Funciones de activación	$\sigma$ (Normal)
Glorot	Ninguno, tanh, sigmoide, softmax	$1 / \text{ventilador}_{\text{promedio}}$
Él	ReLU, Leaky ReLU, ELU, GELU, Swish, Mish	$2 / \text{ventilador}_{\text{en}}$
lecun	SELU	$1 / \text{ventilador}_{\text{en}}$

De forma predeterminada, Keras utiliza la inicialización de Glorot con una distribución uniforme. Cuando creas una capa, puedes cambiar a la inicialización He configurando `kernel_initializer="he_uniform"` o `kernel_initializer="he_normal"` de esta manera:

```
importar tensorflow como tf

denso = tf.keras.layers.Dense(50, activación="relu",
                             kernel_initializer="él_normal")
```

Alternativamente, puede obtener cualquiera de las inicializaciones enumeradas en [la Tabla 11-1](#) y más usando el inicializador VarianceScaling. Por ejemplo, si desea la inicialización de He con una distribución uniforme y basada en fan (en lugar de fan), puede usar el siguiente código:

en

```
he_avg_init = tf.keras.initializers.VarianceScaling(escala=2., modo="fan_avg",
                                                    distribución="uniforme") densa =
tf.keras.layers.Dense(50, activación="sigmoide",
                      kernel_initializer=he_avg_init)
```

## Mejores funciones de activación

Una de las ideas del artículo de 2010 de Glorot y Bengio fue que los problemas con los gradientes inestables se debían en parte a una mala elección de la función de activación. Hasta entonces, la mayoría de la gente había asumido que si la Madre Naturaleza había elegido utilizar funciones de activación aproximadamente sigmoides en las neuronas biológicas, debía ser una excelente elección. Pero resulta que otros

Las funciones de activación se comportan mucho mejor en redes neuronales profundas, en particular, la función de activación ReLU, principalmente porque no se satura con valores positivos y también porque es muy rápida de calcular.

Desafortunadamente, la función de activación de ReLU no es perfecta. Sufre un problema conocido como ReLU moribundos: durante el entrenamiento, algunas neuronas efectivamente "mueren", lo que significa que dejan de generar cualquier valor que no sea 0. En algunos casos, es posible que descubras que la mitad de las neuronas de tu red están muertas, especialmente si usaste una gran tasa de aprendizaje. Una neurona muere cuando sus pesos se modifican de tal manera que la entrada de la función ReLU (es decir, la suma ponderada de las entradas de la neurona más su término de sesgo) es negativa para todas las instancias del conjunto de entrenamiento. Cuando esto sucede, sigue generando ceros y el descenso del gradiente ya no lo afecta porque el gradiente de la función ReLU es cero cuando su entrada es negativa. <sup>4</sup>

Para resolver este problema, es posible que desee utilizar una variante de la función ReLU, como la ReLU con fugas.

#### ReLU con fugas

La función de activación de ReLU con fugas se define como LeakyReLU ( $z \geq 0$   $\max(\alpha z, z)$  (consulte [la Figura 11-2](#)). El hiperparámetro  $\alpha$  define cuánto se "fuga" la función: es la pendiente de la función para  $z < 0$ . Tener una pendiente para  $z < 0$  garantiza que las ReLU con fugas nunca mueran; pueden entrar en un coma prolongado, pero tienen la posibilidad de despertarse eventualmente. Un [artículo de 2015](#) por Bing Xu et al. comparó varias variantes de la función de activación de ReLU y una de sus conclusiones fue que las variantes con fugas siempre superaron a la estricta función de activación de ReLU. De hecho, establecer  $\alpha = 0,2$  (una fuga grande) pareció dar como resultado un mejor rendimiento que  $\alpha = 0,01$  (una fuga pequeña). El artículo también evaluó el ReLU con fugas aleatorias (RReLU), donde  $\alpha$  se selecciona aleatoriamente en un rango determinado durante el entrenamiento y se fija en un valor promedio durante las pruebas. RReLU también funcionó bastante bien y pareció actuar como un regularizador, reduciendo el riesgo de sobreajustar el conjunto de entrenamiento. Finalmente, el artículo evaluó el ReLU paramétrico con fugas (PReLU), donde  $\alpha$  está autorizado a aprenderse durante el entrenamiento: en lugar de ser un hiperparámetro, se convierte en un parámetro que puede modificarse mediante retropropagación como cualquier otro parámetro. Se informó que PReLU supera con creces a ReLU en conjuntos de datos de imágenes grandes, pero en conjuntos de datos más pequeños corre el riesgo de sobreajustar el conjunto de entrenamiento. <sup>5</sup>

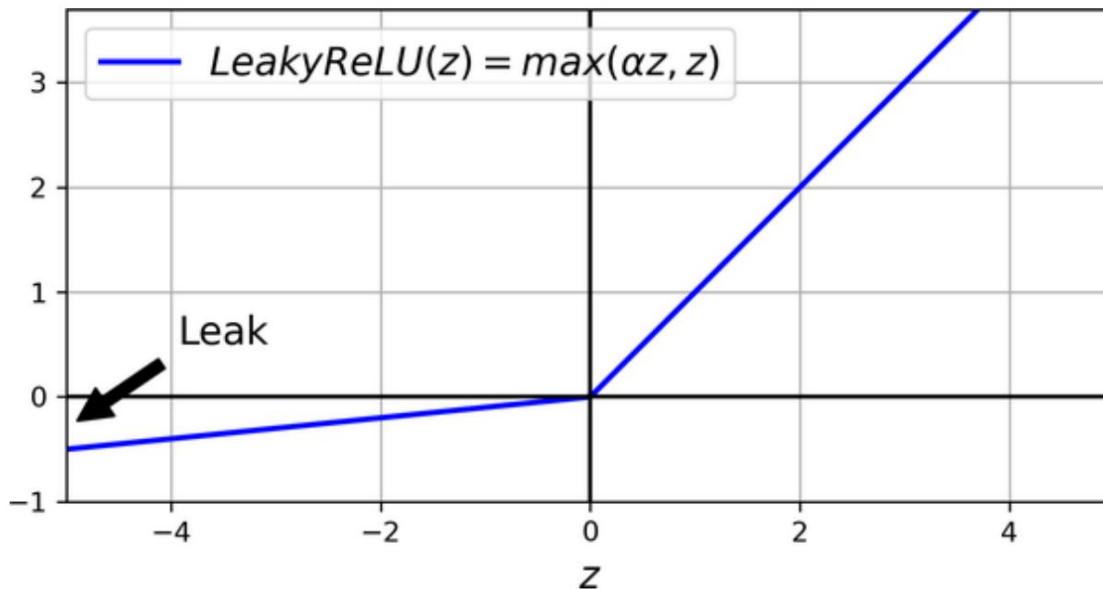


Figura 11-2. Leaky ReLU: como ReLU, pero con una pequeña pendiente para valores negativos

Keras incluye las clases LeakyReLU y PReLU en el paquete `tf.keras.layers`. Al igual que con otras variantes de ReLU, debes usar la inicialización He con estas. Por ejemplo:

```
Leaky_relu = tf.keras.layers.LeakyReLU(alpha=0.2) # por defecto alpha=0.3
dense = tf.keras.layers.Dense(50,
    activation=leaky_relu,
    kernel_initializer="he_normal")
```

Si lo prefieres, también puedes usar LeakyReLU como una capa separada en tu modelo; no hace ninguna diferencia para el entrenamiento y las predicciones:

```
model = tf.keras.models.Sequential([
    ... # más capas
    tf.keras.layers.Dense(50,
        kernel_initializer="he_normal"), # sin activación
    tf.keras.layers.LeakyReLU(alpha=0.2), # activación
    ... # capas más
])
```

Para PReLU, reemplace LeakyReLU con PReLU. Actualmente no existe una implementación oficial de RReLU en Keras, pero puedes implementar la tuya propia con bastante facilidad (para aprender cómo hacerlo, consulta los ejercicios al final del Capítulo 12 ).

ReLU, ReLU con fugas y PReLU adolecen del hecho de que no son funciones fluidas: sus derivadas cambian abruptamente (en  $z = 0$ ). Como vimos en [el Capítulo 4](#) cuando analizamos el lazo, este tipo de discontinuidad puede hacer que el descenso del gradiente rebote alrededor del óptimo y ralentice la convergencia. Ahora veremos algunas variantes suaves de la función de activación ReLU, comenzando con ELU y SELU.

### ELU y SELU

En 2015, un [artículo](#) por Djork-Arné Clevert et al. propuso<sup>6</sup> una nueva función de activación, llamada unidad lineal exponencial (ELU), que superó a todas las variantes de ReLU en los experimentos de los autores: se redujo el tiempo de entrenamiento y la red neuronal funcionó mejor en el conjunto de prueba. [La ecuación 11-2](#) muestra la definición de esta función de activación.

#### Ecuación 11-2. Función de activación ELU

$$\text{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{si } z < 0 \\ z & \text{si } z \geq 0 \end{cases}$$

La función de activación ELU se parece mucho a la función ReLU (consulte [Figura 11-3](#)), con algunas diferencias importantes:

- Toma valores negativos cuando  $z < 0$ , lo que permite que la unidad tenga una salida promedio más cercana a 0 y ayuda a aliviar el problema de los gradientes que desaparecen. El hiperparámetro  $\alpha$  define lo opuesto al valor al que se aproxima la función ELU cuando  $z$  es un número negativo grande. Generalmente se establece en 1, pero puedes modificarlo como cualquier otro hiperparámetro.
- Tiene un gradiente distinto de cero para  $z < 0$ , lo que evita el problema de las neuronas muertas.
- Si  $\alpha$  es igual a 1, entonces la función es suave en todas partes, incluso alrededor de  $z = 0$ , lo que ayuda a acelerar el descenso del gradiente ya que no rebota tanto hacia la izquierda y la derecha de  $z = 0$ .

Usar ELU con Keras es tan fácil como configurar activación="elu" y, al igual que con otras variantes de ReLU, debes usar la inicialización He. El principal inconveniente de la función de activación ELU es que es más lento de calcular que

la función ReLU y sus variantes (debido al uso de la función exponencial). Su tasa de convergencia más rápida durante el entrenamiento puede compensar ese cálculo lento, pero aún así, en el momento de la prueba una red ELU será un poco más lenta que una red ReLU.

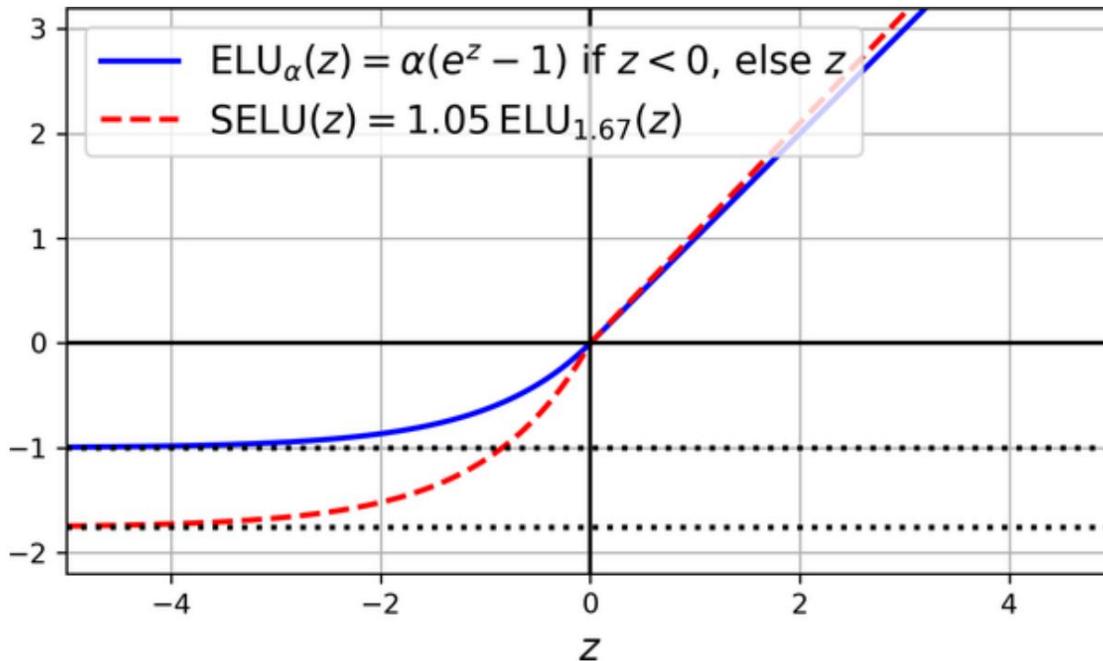


Figura 11-3. Funciones de activación ELU y SELU

Poco después, un [artículo de 2017](#) por Günter Klambauer et al. introdujo la función de activación ELU escalada (SELU): como su nombre indica, es una variante escalada de la función de activación ELU (aproximadamente 1,05 veces ELU, usando  $\alpha \approx 1,67$ ). Los autores demostraron que si se construye una red neuronal compuesta exclusivamente por una pila de capas densas (es decir, un MLP), y si todas las capas ocultas usan la función de activación SELU, entonces la red se autonormalizará: la salida de cada capa tienden a preservar una media de 0 y una desviación estándar de 1 durante el entrenamiento, lo que resuelve el problema de los gradientes que desaparecen/explotan. Como resultado, la función de activación SELU puede superar a otras funciones de activación de MLP, especialmente las profundas. Para usarlo con Keras, simplemente configure activación="selu". Sin embargo, existen algunas condiciones para que se produzca la autonormalización (consulte el artículo para conocer la justificación matemática):

- Las características de entrada deben estar estandarizadas: media 0 y desviación estándar 1.

- Los pesos de cada capa oculta deben inicializarse utilizando la inicialización normal de LeCun. En Keras, esto significa configurar `kernel_initializer="lecun_normal"`.
- La propiedad de autonormalización solo está garantizada con MLP simples. Si intenta utilizar SELU en otras arquitecturas, como redes recurrentes (consulte el Capítulo 15) o redes con conexiones omitidas (es decir, conexiones que omiten capas, como en redes anchas y profundas), probablemente no superará a ELU.
- No puede utilizar técnicas de regularización como  $\ell$  o  $\ell_1$  regularización, norma máxima, norma por lotes o abandono regular (estos se analizan más adelante en este capítulo).

Estas son limitaciones importantes, por lo que, a pesar de sus promesas, SELU no ganó mucha tracción. Además, tres funciones de activación más parecen superarlo de manera bastante consistente en la mayoría de las tareas: GELU, Swish y Mish.

### GELU, Swish y Mish

GELU se presentó en un artículo de 2016.<sup>8</sup> por Dan Hendrycks y Kevin Gimpel. Una vez más, puedes considerarlo como una variante suave de la función de activación ReLU. Su definición se da en la Ecuación 11-3, donde  $\Phi$  es la función de distribución acumulativa (CDF) gaussiana estándar:  $\Phi(z)$  corresponde a la probabilidad de que un valor muestrado aleatoriamente de una distribución normal de media 0 y varianza 1 sea menor que  $z$ .

Ecuación 11-3. Función de activación GELU

$$\text{GELU}(z) = z \Phi(z)$$

Como puede ver en la Figura 11-4, GELU se parece a ReLU: se acerca a 0 cuando su entrada  $z$  es muy negativa y se acerca a  $z$  cuando  $z$  es muy positiva. Sin embargo, mientras que todas las funciones de activación que hemos analizado hasta ahora eran convexas y monótonas, la función<sup>9</sup> de activación GELU no lo es: de izquierda a derecha, comienza yendo recto, luego se mueve hacia abajo y alcanza un punto bajo alrededor de  $-0,17$  (cerca de  $z \approx -0,75$ ), y finalmente rebota y termina yendo directamente hacia la parte superior derecha. Esta forma bastante compleja y el hecho de que tiene una curvatura en cada punto pueden explicar por qué funciona tan bien, especialmente para tareas complejas: puede resultar más fácil descender en gradiente.

adaptarse a patrones complejos. En la práctica, a menudo supera a todas las demás funciones de activación analizadas hasta ahora. Sin embargo, requiere un poco más de computación y el aumento de rendimiento que proporciona no siempre es suficiente para justificar el costo adicional. Dicho esto, es posible demostrar que es aproximadamente igual a  $z\phi(1,702 z)$ , donde  $\phi$  es la función sigmoidea: utilizar esta aproximación también funciona muy bien y tiene la ventaja de ser mucho más rápida de calcular.

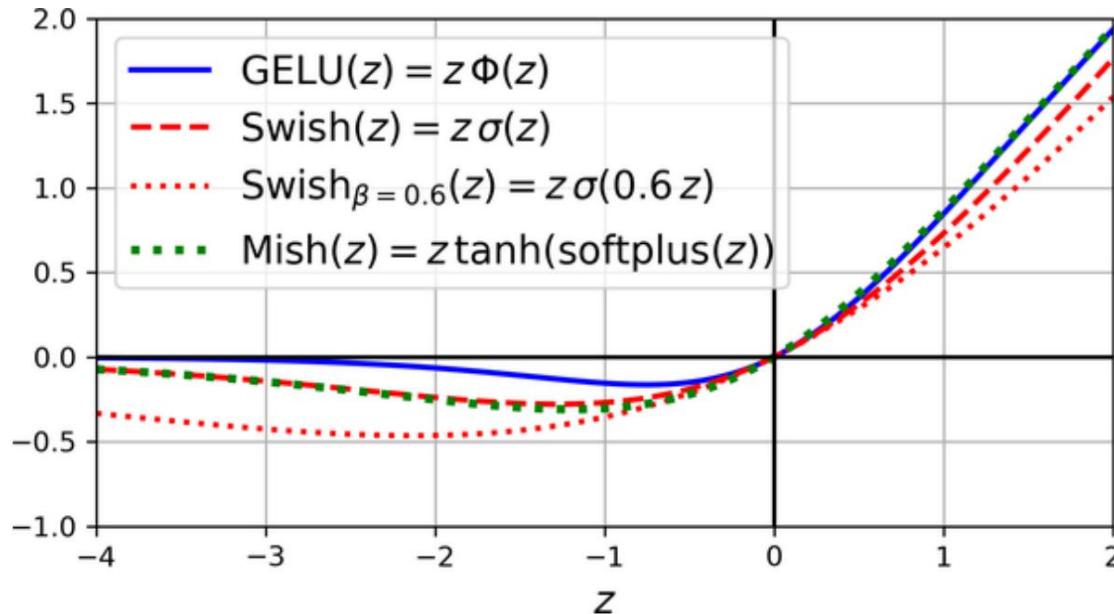


Figura 11-4. Funciones de activación GELU, Swish, Swish parametrizado y Mish

El artículo de GELU también introdujo la función de activación de la unidad lineal sigmoidea (SiLU), que es igual a  $z\phi(z)$ , pero fue superada por GELU en el pruebas de los autores. Curiosamente, un [artículo de 2017](#) por Prajit Ramachandran <sup>10</sup> et al. redescubrió la función SiLU buscando automáticamente buenas funciones de activación. Los autores lo llamaron Swish y el nombre se hizo popular. En su artículo, Swish superó a todas las demás funciones, incluida GELU. Ramachandran et al. Más tarde generalizó Swish agregando un hiperparámetro adicional  $\beta$  para escalar la entrada de la función sigmoidea. La función Swish generalizada es  $\text{Swish}(z) = z\phi(\beta z)$ , por lo que GELU es aproximadamente igual a la función Swish generalizada usando  $\beta = 1,702$ . Puedes ajustar  $\beta$  como cualquier otro hiperparámetro. Alternativamente, también es posible hacer que  $\beta$  sea entrenable y dejar que el descenso de gradiente lo optimice: al igual que PReLU, esto puede hacer que su modelo sea más poderoso, pero también corre el riesgo de sobreajustar los datos.

Otra función de activación bastante similar es Mish, que se presentó en un [artículo de 2019](#), por Diganta Misra. Se define como  $mish(z) = ztanh(\text{softplus}(z))$ , donde  $\text{softplus}(z) = \log(1 + \exp(z))$ . Al igual que GELU y Swish, es una variante suave, no convexa y no monótona de ReLU, y una vez más el autor realizó muchos experimentos y descubrió que Mish generalmente superaba a otras funciones de activación, incluso Swish y GELU, por un pequeño margen. [La Figura 11-4](#) muestra GELU, Swish (ambos con el valor predeterminado  $\beta = 1$  y con  $\beta = 0,6$ ) y, por último, Mish. Como puede ver, Mish se superpone casi perfectamente con Swish cuando  $z$  es negativo y casi perfectamente con GELU cuando  $z$  es positivo.

## CONSEJO

Entonces, ¿qué función de activación debería utilizar para las capas ocultas de sus redes neuronales profundas? ReLU sigue siendo un buen valor predeterminado para tareas simples: a menudo es tan bueno como las funciones de activación más sofisticadas, además es muy rápido de calcular y muchas bibliotecas y aceleradores de hardware proporcionan optimizaciones específicas de ReLU. Sin embargo, Swish es probablemente un mejor valor predeterminado para tareas más complejas, e incluso puedes probar Swish parametrizado con un parámetro  $\beta$  que se puede aprender para las tareas más complejas. Mish puede brindarle resultados ligeramente mejores, pero requiere un poco más de cálculo. Si le importa mucho la latencia del tiempo de ejecución, entonces puede preferir ReLU con fugas o ReLU con fugas parametrizado para tareas más complejas. Para MLP profundos, pruebe SELU, pero asegúrese de respetar las restricciones enumeradas anteriormente. Si tiene tiempo libre y capacidad informática, puede utilizar la validación cruzada para evaluar también otras funciones de activación.

Keras admite GELU y Swish desde el primer momento; simplemente use `activación="gelu"` o `activación="swish"`. Sin embargo, todavía no es compatible con Mish o la función de activación generalizada Swish (pero consulte [el Capítulo 12](#) para ver cómo implementar sus propias funciones y capas de activación).

¡Eso es todo por las funciones de activación! Ahora, veamos una forma completamente diferente de resolver el problema de los gradientes inestables: la normalización por lotes.

## Normalización por lotes

Aunque el uso de la inicialización de He junto con ReLU (o cualquiera de sus variantes) puede reducir significativamente el peligro de que los gradientes desaparezcan o exploten.

problemas al inicio del entrenamiento, no garantiza que no volverán a aparecer durante el entrenamiento.

En un [artículo de 2015](#), Sergey Ioffe y Christian Szegedy propusieron una técnica llamada normalización por lotes (BN) que aborda estos problemas. La técnica consiste en añadir una operación en el modelo justo antes o después de la función de activación de cada capa oculta. Esta operación simplemente centra en cero y normaliza cada entrada, luego escala y desplaza el resultado utilizando dos nuevos vectores de parámetros por capa: uno para escalar y el otro para desplazar. En otras palabras, la operación permite que el modelo aprenda la escala óptima y la media de cada una de las entradas de la capa. En muchos casos, si agrega una capa BN como primera capa de su red neuronal, no necesita estandarizar su conjunto de entrenamiento. Es decir, no hay necesidad de StandardScaler o Normalization; la capa BN lo hará por usted (bueno, aproximadamente, ya que solo analiza un lote a la vez y también puede cambiar la escala y cambiar cada característica de entrada).

Para centrar en cero y normalizar las entradas, el algoritmo necesita estimar la media y la desviación estándar de cada entrada. Lo hace evaluando la media y la desviación estándar de la entrada sobre el mini lote actual (de ahí el nombre "normalización de lotes"). Toda la operación se resume paso a paso en la [Ecuación 11-4](#).

Ecuación 11-4. Algoritmo de normalización por lotes

$$\begin{aligned}
 1. \quad B &= \frac{1}{\text{megabyte}} \sum_{y_0=1}^{\text{megabyte}} x(i) \\
 2. \quad B^2 &= \frac{1}{\text{megabyte}} \sum_{y_0=1}^{\text{megabyte}} (x(i) - B)^2 \\
 3. \quad \hat{x}(i) &= \frac{x(y_0) - B}{\sqrt{B} \sqrt{2 + \epsilon}} \\
 4. \quad z^{(i)} &= \hat{x}(i) + 
 \end{aligned}$$

En este algoritmo:

- $\mu_B$  es el vector de medias de entrada, evaluado sobre todo el mini lote B (contiene una media por entrada).
- $m_B$  es el número de instancias en el mini lote.
- $\sigma_B$  es el vector de desviaciones estándar de entrada, también evaluado en todo el mini lote (contiene una desviación estándar por entrada).
- $x^{(i)}$  es el vector de entradas normalizadas y centradas en cero, por ejemplo i.
- $\epsilon$  es un número pequeño que evita la división por cero y garantiza que los gradientes no crezcan demasiado (normalmente  $10^{-5}$ ).<sup>5</sup> A esto se le llama término de suavizado.
- $\gamma$  es el vector de parámetros de escala de salida para la capa (contiene un parámetro de escala por entrada).
- $\alpha$  representa la multiplicación por elementos (cada entrada se multiplica por su correspondiente parámetro de escala de salida).
- $\beta$  es el vector de parámetros de desplazamiento (desplazamiento) de salida para la capa (contiene un parámetro de desplazamiento por entrada). Cada entrada está compensada por su correspondiente parámetro de desplazamiento.
- (i)  $z$  es la salida de la operación BN. Es una versión reescalada y desplazada de las entradas.

Entonces, durante el entrenamiento, BN estandariza sus entradas, luego las reescala y las compensa. ¡Bien! ¿Qué pasa en el momento del examen? Bueno, no es tan simple. De hecho, es posible que necesitemos hacer predicciones para instancias individuales en lugar de para lotes de instancias: en este caso, no tendremos forma de calcular la media y la desviación estándar de cada entrada. Además, incluso si tenemos un lote de instancias, puede ser demasiado pequeño o las instancias pueden no ser independientes ni estar distribuidas de manera idéntica, por lo que calcular estadísticas sobre las instancias del lote no sería confiable. Una solución podría ser esperar hasta el final del entrenamiento, luego ejecutar todo el conjunto de entrenamiento a través de la red neuronal y calcular la media y la desviación estándar de cada entrada de la capa BN. Estas medias de entrada y desviaciones estándar "finales" podrían usarse en lugar de las medias de entrada por lotes y las desviaciones estándar al hacer predicciones. Sin embargo, la mayoría de las implementaciones de normalización por lotes estiman estas estadísticas finales durante el entrenamiento u

las medias de entrada de la capa y las desviaciones estándar. Esto es lo que Keras hace automáticamente cuando usas la capa BatchNormalization. En resumen, se aprenden cuatro vectores de parámetros en cada capa normalizada por lotes:  $\gamma$  (el vector de escala de salida) y  $\beta$  (el vector de compensación de salida) se aprenden mediante retropropagación regular,  $\mu$  (el vector medio de entrada final) y  $\sigma$  (el vector de desviación estándar de entrada final) se estiman utilizando una media móvil exponencial. Tenga en cuenta que  $\mu$  y  $\sigma$  se estiman durante el entrenamiento, pero se utilizan sólo después del entrenamiento (para reemplazar las medias de entrada por lotes y las desviaciones estándar en la [Ecuación 11-4](#)).

Ioffe y Szegedy demostraron que la normalización por lotes mejoró considerablemente todas las redes neuronales profundas con las que experimentaron, lo que llevó a una enorme mejora en la tarea de clasificación de ImageNet (ImageNet es una gran base de datos de imágenes clasificadas en muchas clases, comúnmente utilizada para evaluar sistemas de visión por computadora). El problema de los gradientes de fuga se redujo fuertemente, hasta el punto de que pudieron utilizar funciones de activación saturantes como la función tanh e incluso la función de activación sigmoidea. Las redes también eran mucho menos sensibles a la inicialización del peso. Los autores pudieron utilizar tasas de aprendizaje mucho mayores, lo que aceleró significativamente el proceso de aprendizaje. En concreto, señalan que:

Aplicada a un modelo de clasificación de imágenes de última generación, la normalización por lotes logra la misma precisión con 14 veces menos pasos de entrenamiento y supera al modelo original por un margen significativo. [...] Utilizando un conjunto de redes normalizadas por lotes, mejoramos el mejor resultado publicado en la clasificación de ImageNet: alcanzando un 4,9% de error de validación entre los cinco primeros (y un 4,8% de error de prueba), superando la precisión de los evaluadores humanos.

Finalmente, como un regalo que se sigue dando, la normalización por lotes actúa como un regularizador, reduciendo la necesidad de otras técnicas de regularización (como la deserción, que se describe más adelante en este capítulo).

Sin embargo, la normalización por lotes agrega cierta complejidad al modelo (aunque puede eliminar la necesidad de normalizar los datos de entrada, como se analizó anteriormente). Además, existe una penalización en el tiempo de ejecución: la red neuronal hace predicciones más lentas debido a los cálculos adicionales necesarios en cada capa. Afortunadamente, a menudo es posible fusionar la capa BN con la capa anterior después del entrenamiento, evitando así la penalización del tiempo de ejecución. Esto se hace por

actualizar los pesos y sesgos de la capa anterior para que produzca directamente resultados de la escala y el desplazamiento apropiados. Por ejemplo, si la capa anterior calcula  $XW + b$ , entonces la capa BN calculará  $\gamma(XW + b - \mu) / \sigma + \beta$  (ignorando el término de suavizado  $\epsilon$  en el denominador). Si definimos  $W' = \gamma W / \sigma$  y  $b' = \gamma(b - \mu) / \sigma + \beta$ , la ecuación se simplifica a  $XW' + b'$ . Entonces, si reemplazamos los pesos y sesgos de la capa anterior ( $W$  y  $b$ ) con los pesos y sesgos actualizados ( $W'$  y  $b'$ ), podemos deshacernos de la capa BN (el convertidor de TFLite hace esto automáticamente; consulte [el Capítulo 19](#)).

### NOTA

Es posible que descubra que el entrenamiento es bastante lento, porque cada época lleva mucho más tiempo cuando utiliza la normalización por lotes. Esto suele verse contrarrestado por el hecho de que la convergencia es mucho más rápida con BN, por lo que se necesitarán menos épocas para alcanzar el mismo rendimiento. Con todo, el tiempo en la pared suele ser más corto (este es el tiempo que mide el reloj de la pared).

### Implementar la normalización por lotes con Keras

Como ocurre con la mayoría de las cosas con Keras, implementar la normalización por lotes es sencillo e intuitivo. Simplemente agregue una capa BatchNormalization antes o después de la función de activación de cada capa oculta. También puede agregar una capa BN como primera capa en su modelo, pero una capa de Normalización simple generalmente funciona igual de bien en esta ubicación (su único inconveniente es que primero debe llamar a su método `adapt()`). Por ejemplo, este modelo aplica BN después de cada capa oculta y como primera capa del modelo (después de aplanar las imágenes de entrada):

```
modelo = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]), tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(300, activation="relu",
        kernel_initializer="he_normal"),

    tf.keras.layers.BatchNormalization(), tf.keras.layers.Dense(100,
        activation="relu", kernel_initializer="he_normal"),

    tf.keras.layers.BatchNormalization(), tf.keras.layers.Dense(10,
        activation="softmax")
])
```

¡Eso es todo! En este pequeño ejemplo con solo dos capas ocultas por lotes  
 Es poco probable que la normalización tenga un gran impacto, pero para redes más profundas sí  
 puede hacer una tremenda diferencia.

Mostremos el resumen del modelo:

```
>>> modelo.resumen()
Modelo: "secuencial"

-
Capa (tipo)           Forma de salida          Parámetro #
=====
-
=                     

aplanar (Aplanar)     (Ninguno, 784)          0

-
normalización_por_lotes (Nº de lote (Ninguno, 784)) 3136

-
denso (denso)         (Ninguno, 300)          235500

-
lote_normalización_1 (Lote (Ninguno, 300))        1200

-
denso_1 (denso)       (Ninguno, 100)          30100

-
lote_normalización_2 (Lote (Ninguno, 100))        400

-
denso_2 (denso)       (Ninguno, 10)           1010
=====

=
Parámetros totales: 271,346
Parámetros entrenables: 268,978
Parámetros no entrenables: 2368
```

Como puede ver, cada capa BN agrega cuatro parámetros por entrada:  $\gamma$ ,  $\beta$ ,  $\mu$ , y  $\sigma$ (por ejemplo, la primera capa BN agrega 3136 parámetros, que es  $4 \times 784$ ). Los dos últimos parámetros,  $\mu$  y  $\sigma$  son las medias móviles; ellos son no se ven afectados por la retropropagación, por lo que Keras los llama "no entrenables" (si<sup>13</sup>

cuenta el número total de parámetros BN,  $3136 + 1200 + 400$ , y lo divide por 2, obtiene 2368, que es el número total de parámetros no entrenables en este modelo).

Veamos los parámetros de la primera capa BN. Dos se pueden entrenar (mediante retropropagación) y dos no:

```
>>> [(var.name, var.trainable) para var en model.layers[1].variables]
[('batch_normalization/gamma:0', True),
 ('batch_normalization/beta:0', True), ('normalización_batch/
 media_movimiento:0', Falso), ('normalización_batch/
 varianza_movimiento:0', Falso)]
```

Los autores del artículo de BN argumentaron a favor de agregar las capas de BN antes de las funciones de activación, en lugar de después (como acabamos de hacer). Existe cierto debate sobre esto, ya que cuál es preferible parece depender de la tarea; también puedes experimentar con esto para ver qué opción funciona mejor en tu conjunto de datos. Para agregar las capas BN antes de la función de activación, debe eliminar las funciones de activación de las capas ocultas y agregarlas como capas separadas después de las capas BN. Además, dado que una capa de normalización por lotes incluye un parámetro de compensación por entrada, puede eliminar el término de sesgo de la capa anterior pasando `use_bias=False` al crearlo. Por último, normalmente puede soltar la primera capa de BN para evitar intercalar la primera capa oculta entre dos capas de BN. El código actualizado se ve así:

```
modelo = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]), tf.keras.layers.Dense(300,
        kernel_initializer="he_normal", use_bias=False), tf.keras.layers.BatchNormalization(), tf.
    keras.layers.Activation("relu"),
    tf.keras.layers.Dense(100, kernel_initializer="he_normal",
        use_bias=False), tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Activation("relu"), tf.keras.layers.Dense(10, activation="softmax")
])
```

La clase `BatchNormalization` tiene bastantes hiperparámetros que puedes modificar. Los valores predeterminados normalmente estarán bien, pero ocasionalmente es posible que necesites

para ajustar el impulso. Este hiperparámetro lo utiliza la capa BatchNormalization cuando actualiza los promedios móviles exponenciales; dado un nuevo valor  $v$  (es decir, un nuevo vector de medias de entrada o desviaciones estándar calculadas sobre el lote actual), la capa actualiza el promedio móvil  $v^*$  usando la siguiente ecuación:

$$v^* \leftarrow v^* \times \text{impulso} + v \times (1 - \text{impulso})$$

Un buen valor de impulso suele estar cerca de 1; por ejemplo, 0,9, 0,99 o 0,999. Quiere más 9 para conjuntos de datos más grandes y para minibatches más pequeños.

Otro hiperparámetro importante es el eje: determina qué eje debe normalizarse. Su valor predeterminado es  $-1$ , lo que significa que, de forma predeterminada, normalizará el último eje (utilizando las medias y desviaciones estándar calculadas en los otros ejes). Cuando el lote de entrada es 2D (es decir, la forma del lote es [tamaño del lote, características]), esto significa que cada característica de entrada se normalizará en función de la media y la desviación estándar calculadas en todas las instancias del lote. Por ejemplo, la primera capa BN en el ejemplo de código anterior normalizará (y reescalará y cambiará) de forma independiente cada una de las 784 características de entrada. Si movemos la primera capa BN antes de la capa Aplanar, entonces los lotes de entrada serán 3D, con forma [tamaño de lote, alto, ancho]; por lo tanto, la capa BN calculará 28 medias y 28 desviaciones estándar (1 por columna de píxeles, calculada en todas las instancias del lote y en todas las filas de la columna) y normalizará todos los píxeles de una columna determinada utilizando la misma media. y desviación estándar.

También habrá sólo 28 parámetros de escala y 28 parámetros de cambio. Si, en cambio, aún desea tratar cada uno de los 784 píxeles de forma independiente, debe establecer `axis=[1, 2]`.

La normalización por lotes se ha convertido en una de las capas más utilizadas en las redes neuronales profundas, especialmente en las redes neuronales convolucionales profundas analizadas en el ([Capítulo 14](#)), hasta el punto de que a menudo se omite en los diagramas de arquitectura: se supone que BN se agrega después de cada capa. Ahora veamos una última técnica para estabilizar los gradientes durante el entrenamiento: el recorte de gradientes.

## Recorte de degradado

Otra técnica para mitigar el problema de la explosión de gradientes es recortar los gradientes durante la retropropagación para que nunca superen algún umbral. Esto se llama **recorte de degradado**.<sup>14</sup> Esta técnica se usa generalmente en redes neuronales recurrentes, donde el uso de la normalización por lotes es complicado (como verá en [el Capítulo 15](#)).

En Keras, implementar el recorte de gradiente es solo una cuestión de configurar el argumento `clipvalue` o `clipnorm` al crear un optimizador, como este:

```
optimizador = tf.keras.optimizers.SGD(clipvalue=1.0) model.compile(..., optimizador=optimizador)
```

Este optimizador recortará cada componente del vector de gradiente a un valor entre -1,0 y 1,0. Esto significa que todas las derivadas parciales de la pérdida (con respecto a todos y cada uno de los parámetros entrenables) se recortarán entre -1,0 y 1,0. El umbral es un hiperparámetro que puede ajustar.

Tenga en cuenta que puede cambiar la orientación del vector de gradiente. Por ejemplo, si el vector de gradiente original es [0,9, 100,0], apunta principalmente en la dirección del segundo eje; pero una vez que lo recortas por valor, obtienes [0.9, 1.0], que apunta aproximadamente a la diagonal entre los dos ejes. En la práctica, este enfoque funciona bien. Si desea asegurarse de que el recorte de degradado no cambie la dirección del vector de degradado, debe recortar según la norma configurando `clipnorm` en lugar de `clipvalue`. Esto recortará todo el gradiente si su norma  $\ell$  es mayor que el umbral que eligió. Por ejemplo, si establece `clipnorm=1.0`, entonces el vector [0.9, 100.0] se recortará a [0.00899964, 0.9999595], conservando su orientación pero casi eliminando el primer componente. Si observa que los gradientes explotan durante el entrenamiento (puede realizar un seguimiento del tamaño de los gradientes usando TensorBoard), puede intentar recortar por valor o recortar por norma, con diferentes umbrales, y ver qué opción funciona mejor en el conjunto de validación.

### **Reutilización de capas previamente entrenadas**

Generalmente no es una buena idea entrenar un DNN muy grande desde cero sin antes intentar encontrar una red neuronal existente que realice una tarea similar a la que estás tratando de abordar (explicaré cómo encontrarlas en el Capítulo 14). Si encuentra una red neuronal, generalmente puede reutilizar la mayoría de sus capas, excepto las superiores. Esta técnica

Se llama aprendizaje por transferencia. No sólo acelerará considerablemente el entrenamiento, sino que también requerirá muchos menos datos de entrenamiento.

Suponga que tiene acceso a un DNN que fue entrenado para clasificar imágenes en 100 categorías diferentes, incluidos animales, plantas, vehículos y objetos cotidianos, y ahora desea entrenar un DNN para clasificar tipos específicos de vehículos. Estas tareas son muy similares, incluso parcialmente superpuestas, por lo que debería intentar reutilizar partes de la primera red (consulte [la Figura 11-5](#)).

### NOTA

Si las imágenes de entrada para su nueva tarea no tienen el mismo tamaño que las utilizadas en la tarea original, generalmente tendrá que agregar un paso de preprocesamiento para cambiar su tamaño al tamaño esperado por el modelo original. En términos más generales, el aprendizaje por transferencia funcionará mejor cuando las entradas tengan características similares de bajo nivel.

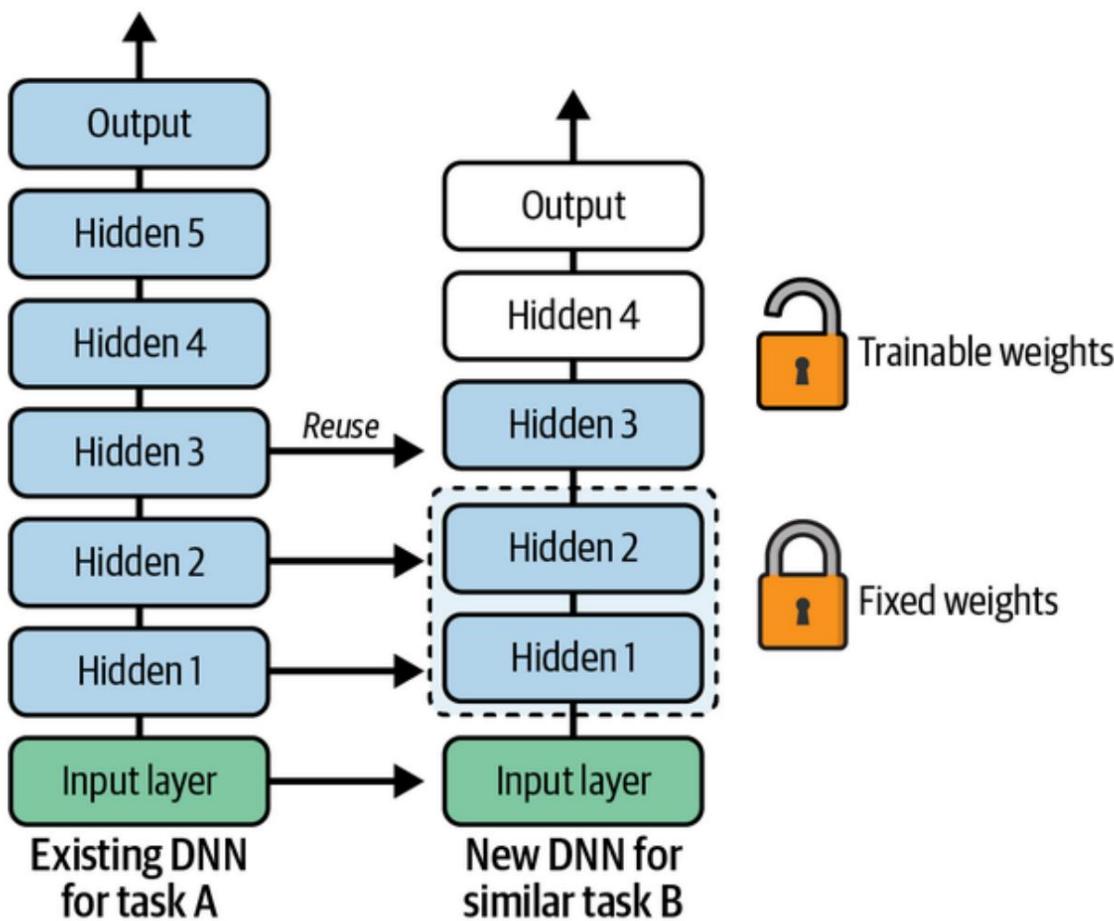


Figura 11-5. Reutilizar capas previamente entrenadas

La capa de salida del modelo original generalmente debe reemplazarse porque lo más probable es que no sea útil en absoluto para la nueva tarea y probablemente no tendrá la cantidad correcta de salidas.

De manera similar, es menos probable que las capas ocultas superiores del modelo original sean tan útiles como las capas inferiores, ya que las características de alto nivel que son más útiles para la nueva tarea pueden diferir significativamente de las que fueron más útiles para la tarea original. . Quiere encontrar la cantidad correcta de capas para reutilizar.

#### CONSEJO

Cuento más similares sean las tareas, más capas querrás reutilizar (comenzando por las capas inferiores). Para tareas muy similares, intente mantener todas las capas ocultas y simplemente reemplace la capa de salida.

Primero intente congelar todas las capas reutilizadas (es decir, haga que sus pesos no se puedan entrenar para que el descenso del gradiente no los modifique y permanezcan fijos), luego entrene su modelo y vea cómo funciona. Luego intente descongelar una o dos de las capas ocultas superiores para permitir que la retropropagación las modifique y vea si el rendimiento mejora. Cuantos más datos de entrenamiento tengas, más capas podrás descongelar. También es útil reducir la tasa de aprendizaje al descongelar capas reutilizadas: esto evitará arruinar sus pesos ajustados.

Si aún no puede obtener un buen rendimiento y tiene pocos datos de entrenamiento, intente eliminar las capas ocultas superiores y congelar todas las capas ocultas restantes nuevamente. Puede iterar hasta encontrar la cantidad correcta de capas para reutilizar. Si tiene muchos datos de entrenamiento, puede intentar reemplazar las capas ocultas superiores en lugar de eliminarlas, e incluso agregar más capas ocultas.

## Transferir aprendizaje con Keras

Veamos un ejemplo. Supongamos que el conjunto de datos Fashion MNIST solo contiene ocho clases; por ejemplo, todas las clases excepto sandalias y camisas. Alguien construyó y entrenó un modelo de Keras en ese conjunto y obtuvo un rendimiento razonablemente bueno (>90% de precisión). Llámemos a este modelo A.

Ahora desea abordar una tarea diferente: tiene imágenes de camisetas y jerseys y desea entrenar un clasificador binario: positivo para camisetas (y blusas), negativo para sandalias. Su conjunto de datos es bastante pequeño; solo tienes 200 imágenes etiquetadas. Cuando entrenas un nuevo modelo para esta tarea (llamémoslo modelo B) con la misma arquitectura que el modelo A, obtienes una precisión de prueba del 91,85 %. Mientras toma su café de la mañana, se da cuenta de que su tarea es bastante similar a la tarea A, entonces, ¿tal vez transferir el aprendizaje pueda ayudar? ¡Vamos a averiguar!

Primero, debe cargar el modelo A y crear un nuevo modelo basado en las capas de ese modelo. Decide reutilizar todas las capas excepto la capa de salida:

```
[...] # Suponiendo que el modelo A ya fue entrenado y guardado en "my_model_A"
model_A =
tf.keras.models.load_model("my_model_A") model_B_on_A =
tf.keras.Sequential(model_A.layers[:-1])
model_B_on_A .add(tf.keras.layers.Dense(1, activación="sigmoide"))
```

Tenga en cuenta que model\_A y model\_B\_on\_A ahora comparten algunas capas. Cuando entrenes model\_B\_on\_A, también afectará a model\_A. Si quieras evitar eso, necesitas clonar model\_A antes de reutilizar sus capas. Para hacer esto, clonas la arquitectura del modelo A con clone\_model() y luego copias sus pesos:

```
model_A_clone = tf.keras.models.clone_model(model_A)
model_A_clone.set_weights(model_A.get_weights())
```

#### ADVERTENCIA

`tf.keras.models.clone_model()` solo clona la arquitectura, no los pesos. Si no los copia manualmente usando `set_weights()`, se inicializarán aleatoriamente cuando se use el modelo clonado por primera vez.

Ahora podría entrenar model\_B\_on\_A para la tarea B, pero dado que la nueva capa de salida se inicializó aleatoriamente, cometerá grandes errores (al menos durante las primeras épocas), por lo que habrá grandes gradientes de error que pueden arruinar los pesos reutilizados. Para evitar esto, una solución es congelar las capas reutilizadas.

durante las primeras épocas, lo que le da a la nueva capa algo de tiempo para aprender pesos razonables. Para hacer esto, establezca el atributo entrenable de cada capa en Falso y compile el modelo:

```
para capa en model_B_on_A.layers[:-1]: capa.trainable =
    False

optimizador = tf.keras.optimizers.SGD(learning_rate=0.001)
model_B_on_A.compile(loss="binary_crossentropy", optimizador=optimizador,
                      métricas=["precisión"])
```

### NOTA

Siempre debes compilar tu modelo después de congelar o descongelar capas.

Ahora puede entrenar el modelo durante algunas épocas, luego descongelar las capas reutilizadas (lo que requiere compilar el modelo nuevamente) y continuar entrenando para ajustar las capas reutilizadas para la tarea B. Después de descongelar las capas reutilizadas, generalmente es una buena idea. para reducir la tasa de aprendizaje, una vez más para evitar dañar los pesos reutilizados.

```
historia = model_B_on_A.fit(X_train_B, y_train_B, épocas=4,
                           datos_validación=(X_valid_B,
                           y_valid_B))

para capa en model_B_on_A.layers[:-1]: capa.trainable =
    True

optimizador = tf.keras.optimizers.SGD(learning_rate=0.001)
model_B_on_A.compile(loss="binary_crossentropy", optimizador=optimizador,
                      métricas=["precisión"])
historia = model_B_on_A.fit(X_train_B, y_train_B, épocas=16,
                           datos_validación=(X_valid_B,
                           y_valid_B))
```

Entonces, ¿cuál es el veredicto final? Bueno, la precisión de la prueba de este modelo es del 93,85%, ¡exactamente dos puntos porcentuales más que el 91,85%! Esto significa que el aprendizaje por transferencia redujo la tasa de error en casi un 25%:

```
>>> model_B_on_A.evaluate(X_test_B, y_test_B)
[0.2546142041683197, 0.9384999871253967]
```

¿Estás convencido? No deberías estarlo: ¡hice trampa! Probé muchas configuraciones hasta que encontré una que demostró una gran mejora. Si intentas cambiar las clases o la semilla aleatoria, verás que la mejora generalmente disminuye, o incluso desaparece o se revierte. Lo que hice se llama “torturar al dato hasta que confiese”. Cuando un artículo parece demasiado positivo, hay que sospechar: tal vez la nueva y llamativa técnica en realidad no ayuda mucho (de hecho, puede incluso degradar el rendimiento), pero los autores probaron muchas variantes y solo informaron los mejores resultados (que pueden ser por pura suerte), sin mencionar cuántos fracasos encontraron en el camino. La mayoría de las veces, esto no es nada malicioso, pero es parte de la razón por la que muchos resultados científicos nunca pueden reproducirse.

¿Por qué hice trampa? Resulta que el aprendizaje por transferencia no funciona muy bien con redes pequeñas y densas, presumiblemente porque las redes pequeñas aprenden pocos patrones y las redes densas aprenden patrones muy específicos, que es poco probable que sean útiles en otras tareas. El aprendizaje por transferencia funciona mejor con redes neuronales convolucionales profundas, que tienden a aprender detectores de características que son mucho más generales (especialmente en las capas inferiores). Revisaremos el aprendizaje por transferencia en [el Capítulo 14](#), utilizando las técnicas que acabamos de analizar (y esta vez no habrá trampas, ¡lo prometo!).

## Preentrenamiento no supervisado

Suponga que desea abordar una tarea compleja para la cual no tiene muchos datos de entrenamiento etiquetados, pero desafortunadamente no puede encontrar un modelo entrenado en una tarea similar. ¡No pierdas la esperanza! En primer lugar, debería intentar recopilar más datos de entrenamiento etiquetados, pero si no puede, aún podrá realizar un entrenamiento previo sin supervisión ([consulte la Figura 11-6](#)). De hecho, a menudo resulta barato recopilar ejemplos de capacitación sin etiquetar, pero es costoso etiquetarlos. Si puede recopilar muchos datos de entrenamiento sin etiquetar, puede intentar usarlos para entrenar un modelo no supervisado, como un codificador automático o una red generativa adversaria (GAN; consulte el Capítulo 17). Luego, puede reutilizar las capas inferiores del codificador automático o las capas inferiores del discriminador de GAN, agregar la capa de salida para su tarea en la parte superior y ajustar la red final utilizando el aprendizaje supervisado (es decir, con los ejemplos de entrenamiento etiquetados).

Es esta técnica la que Geoffrey Hinton y su equipo utilizaron en 2006 y la que condujo al resurgimiento de las redes neuronales y al éxito del aprendizaje profundo. Hasta 2010, el preentrenamiento no supervisado (normalmente con máquinas Boltzmann restringidas (RBM; consulte el cuaderno en <https://homl.info/extranns> )) era la norma para las redes profundas, y sólo después de que se alivió el problema de los gradientes de fuga se volvió común. Es mucho más común entrenar DNN utilizando únicamente el aprendizaje supervisado. El preentrenamiento no supervisado (hoy en día generalmente se usan codificadores automáticos o GAN en lugar de RBM) sigue siendo una buena opción cuando tienes que resolver una tarea compleja, no hay un modelo similar que puedas reutilizar y hay pocos datos de entrenamiento etiquetados pero muchos datos de entrenamiento sin etiquetar.

Tenga en cuenta que en los primeros días del aprendizaje profundo era difícil entrenar modelos profundos, por lo que la gente usaba una técnica llamada preentrenamiento codicioso por capas (que se muestra en [la Figura 11-6](#)). Primero entrenarían un modelo no supervisado con una sola capa, generalmente un RBM, luego congelarían esa capa y agregarían otra encima, luego entrenarían el modelo nuevamente (en realidad, solo entrenarían la nueva capa), luego congelarían la nueva capa. y agregue otra capa encima, entrene el modelo nuevamente, y así sucesivamente. Hoy en día, las cosas son mucho más simples: la gente generalmente entrena el modelo completo no supervisado de una sola vez y usa codificadores automáticos o GAN en lugar de RBM.

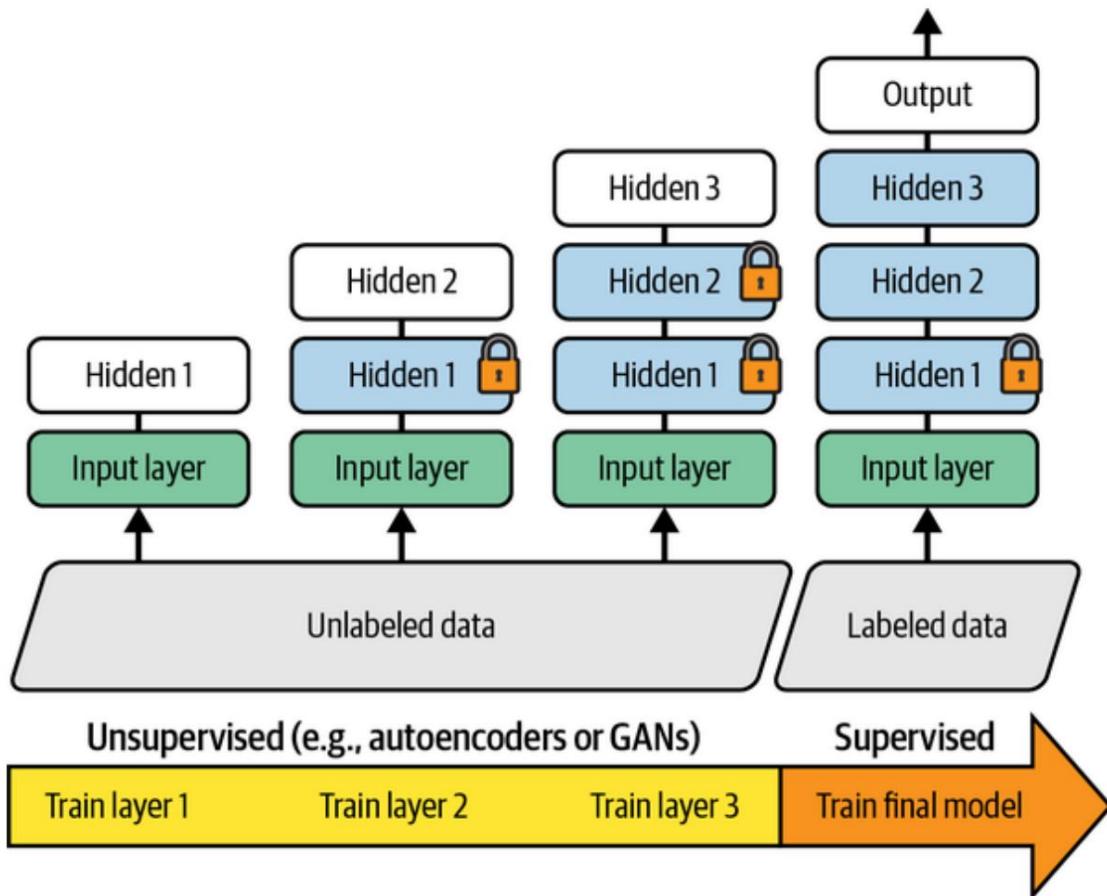


Figura 11-6. En el entrenamiento no supervisado, se entrena un modelo con todos los datos, incluidos los datos sin etiquetar, utilizando una técnica de aprendizaje no supervisado, luego se ajusta para la tarea final solo con los datos etiquetados utilizando una técnica de aprendizaje supervisado; la parte no supervisada puede entrenar una capa a la vez como se muestra aquí, o puede entrenar el modelo completo directamente

## Entrenamiento previo en una tarea auxiliar

Si no tiene muchos datos de entrenamiento etiquetados, una última opción es entrenar una primera red neuronal en una tarea auxiliar para la cual pueda obtener o generar fácilmente datos de entrenamiento etiquetados y luego reutilizar las capas inferiores de esa red para su tarea real. Las capas inferiores de la primera red neuronal aprenderán detectores de características que probablemente serán reutilizables por la segunda red neuronal.

Por ejemplo, si desea crear un sistema para reconocer rostros, es posible que solo tenga unas pocas fotografías de cada individuo, lo que claramente no es suficiente para entrenar a un buen clasificador. Reunir cientos de fotografías de cada persona no sería práctico. Sin embargo, se podrían recopilar muchas imágenes de personas aleatorias en la web y entrenar una primera red neuronal para detectar si dos imágenes diferentes muestran a la misma persona o no. Una red así aprendería

buenos detectores de características para caras, por lo que reutilizar sus capas inferiores le permitiría entrenar un buen clasificador de caras que utilice pocos datos de entrenamiento.

Para aplicaciones de procesamiento de lenguaje natural (NLP), puede descargar un corpus de millones de documentos de texto y generar automáticamente datos etiquetados a partir de ellos. Por ejemplo, podría enmascarar aleatoriamente algunas palabras y entrenar un modelo para predecir cuáles son las palabras que faltan (por ejemplo, debería predecir que la palabra que falta en la oración "¿Qué \_\_\_ estás diciendo?" probablemente sea "son" o "fueron" ). Si puede entrenar un modelo para lograr un buen rendimiento en esta tarea, entonces ya sabrá bastante sobre el lenguaje y ciertamente podrá reutilizarlo para su tarea real y ajustarlo en sus datos etiquetados (analizaremos más entrenamiento previo). tareas del [Capítulo 15](#)).

## NOTA

El aprendizaje autosupervisado se produce cuando se generan automáticamente las etiquetas a partir de los datos mismos, como en el ejemplo del enmascaramiento de texto, y luego se entrena un modelo en el conjunto de datos "etiquetado" resultante utilizando técnicas de aprendizaje supervisado.

## Optimizadores más rápidos

Entrenar una red neuronal profunda muy grande puede ser tremadamente lento. Hasta ahora hemos visto cuatro formas de acelerar el entrenamiento (y alcanzar una mejor solución): aplicar una buena estrategia de inicialización para los pesos de conexión, usar una buena función de activación, usar normalización por lotes y reutilizar partes de una red previamente entrenada (posiblemente construida para una tarea auxiliar o utilizando el aprendizaje no supervisado). Otro gran aumento de velocidad proviene del uso de un optimizador más rápido que el optimizador de descenso de gradiente normal. En esta sección presentaremos los algoritmos de optimización más populares: impulso, gradiente acelerado de Nesterov, AdaGrad, RMSProp y finalmente Adam y sus variantes.

## Impulso

Imagine una bola de boliche rodando por una pendiente suave sobre una superficie lisa: comenzará lentamente, pero rápidamente cobrará impulso hasta que finalmente alcance la velocidad terminal (si hay algo de fricción o resistencia del aire). Esto es

la idea central detrás de la optimización del impulso, propuesta por Boris Polyak en 1964.<sup>15</sup> Por el contrario, el descenso regular en gradiente dará pequeños pasos cuando la pendiente es suave y grandes pasos cuando la pendiente es pronunciada, pero nunca aumentará la velocidad. Como resultado, el descenso de gradiente regular es generalmente mucho más lento para alcanzar el mínimo que la optimización del impulso.

Recuerde que el descenso de gradiente actualiza los pesos  $\theta$  restando directamente el gradiente de la función de costo  $J(\theta)$  con respecto a los pesos ( $-\nabla J(\theta)$ ) multiplicado por la tasa de aprendizaje  $\eta$ . La ecuación es  $\theta \leftarrow \theta - \eta \nabla J(\theta)$ . No le importa cuáles eran los gradientes anteriores. Si el gradiente local es pequeño, avanza muy lentamente.

A la optimización del momento le importa mucho cuáles eran los gradientes anteriores: en cada iteración, resta el gradiente local del vector de momento  $m$  (multiplicado por la tasa de aprendizaje  $\eta$ ) y actualiza los pesos sumando este vector de momento (consulte la Ecuación 11-5). En otras palabras, el gradiente se utiliza como aceleración, no como velocidad. Para simular algún tipo de mecanismo de fricción y evitar que el impulso crezca demasiado, el algoritmo introduce un nuevo hiperparámetro  $\beta$ , llamado impulso, que debe establecerse entre 0 (alta fricción) y 1 (sin fricción). Un valor de impulso típico es 0,9.

#### Ecuación 11-5. Algoritmo de impulso

$$\begin{aligned} 1. \quad m &\leftarrow \beta m - \eta \nabla J(\theta) \\ 2. \quad \theta &\leftarrow \theta + m \end{aligned}$$

Puede verificar que si el gradiente permanece constante, la velocidad terminal (es decir, el tamaño máximo de las actualizaciones de peso) es igual a ese gradiente multiplicado por la tasa de aprendizaje  $\eta$  multiplicada por  $1/(1-\beta)$  (ignorando el signo).

Por ejemplo, si  $\beta = 0,9$ , entonces la velocidad terminal es igual a 10 veces el gradiente multiplicado por la tasa de aprendizaje, por lo que la optimización del impulso termina siendo 10 veces más rápida que el descenso del gradiente. Esto permite que la optimización del impulso salga de los estancamientos mucho más rápido que el descenso del gradiente. Vimos en el capítulo 4 que cuando los insumos tienen escalas muy diferentes, la función de costos se verá como un cuenco alargado (consulte la figura 4-7). El descenso en pendiente baja bastante rápido por la pendiente empinada, pero luego lleva mucho tiempo bajar el valle. Por el contrario, la optimización del impulso se reducirá

valle cada vez más rápido hasta llegar al fondo (el óptimo). En las redes neuronales profundas que no utilizan la normalización por lotes, las capas superiores a menudo terminarán teniendo entradas con escalas muy diferentes, por lo que usar la optimización del impulso ayuda mucho. También puede ayudar a superar los óptimos locales.

### NOTA

Debido al impulso, el optimizador puede sobrepasarse un poco, luego regresar, sobrepasarse nuevamente y oscilar así muchas veces antes de estabilizarse al mínimo. Ésta es una de las razones por las que es bueno tener un poco de fricción en el sistema: elimina estas oscilaciones y, por tanto, acelera la convergencia.

Implementar la optimización del impulso en Keras es una obviedad: simplemente use el optimizador SGD y establezca su hiperparámetro de impulso, luego recuéstese y obtenga ganancias.

```
optimizador = tf.keras.optimizers.SGD (tasa_de_aprendizaje = 0,001, impulso = 0,9)
```

El único inconveniente de la optimización del impulso es que agrega otro hiperparámetro más para ajustar. Sin embargo, el valor de impulso de 0,9 suele funcionar bien en la práctica y casi siempre es más rápido que el descenso de gradiente normal.

### gradiente acelerado de Nesterov

Una pequeña variante de la optimización del impulso, propuesta por [Yuriii Nesterov en 1983](#), es casi siempre más rápida que la optimización del impulso normal.

El método del gradiente acelerado de Nesterov (NAG), también conocido como optimización del impulso de Nesterov, mide el gradiente de la función de costos no en la posición local  $\theta$  sino ligeramente por delante en la dirección del impulso, en  $\theta + \beta m$  (consulte [la ecuación 11-6](#)).

Ecuación 11-6. Algoritmo de gradiente acelerado de Nesterov

$$\begin{aligned} 1. \quad m &\leftarrow \beta m - \eta J'(\theta + \beta m) \\ 2. \quad \theta &\leftarrow \theta + m \end{aligned}$$

Este pequeño ajuste funciona porque, en general, el vector de impulso apuntará en la dirección correcta (es decir, hacia el óptimo), por lo que será un poco más preciso usar el gradiente medido un poco más lejos en esa dirección en lugar del gradiente en la dirección original. posición, como se puede ver en [la Figura 11-7](#) (donde  $\nabla$  representa el gradiente de la función de costos medido en el punto inicial  $\theta$ , y  $\tilde{\nabla}$  representa el gradiente en el punto ubicado en  $\theta + \beta m$ ).

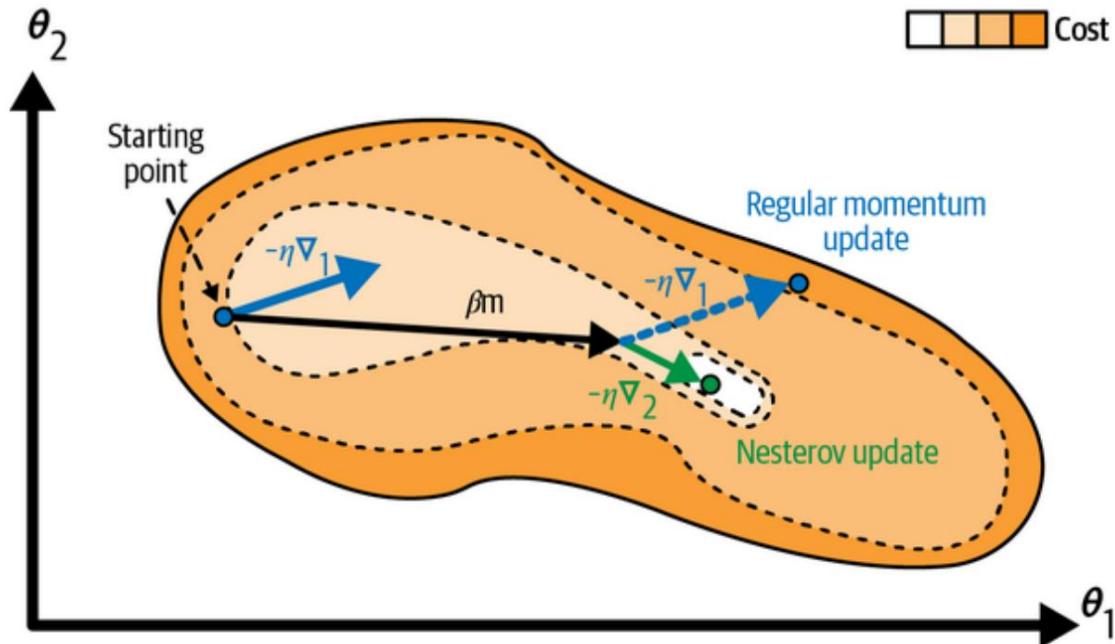


Figura 11-7. Optimización del impulso regular versus Nesterov: el primero aplica los gradientes calculados antes del paso de impulso, mientras que el segundo aplica los gradientes calculados después

Como puede ver, la actualización de Nesterov termina más cerca del óptimo. Después de un tiempo, estas pequeñas mejoras se acumulan y NAG termina siendo significativamente más rápido que la optimización de impulso normal. Además, observe que cuando el impulso empuja los pesos a través de un valle,  $\nabla$  continúa empujando más a través del valle, mientras que  $\tilde{\nabla}$  empuja hacia el fondo del valle. Esto ayuda a reducir las oscilaciones y, por tanto, NAG converge más rápido.

Para usar NAG, simplemente configure `nesterov=True` al crear el optimizador SGD:

```
optimizador = tf.keras.optimizers.SGD(tasa_de_aprendizaje = 0,001, impulso = 0,9,
```

nesterov=Verdadero)

## adagrad

Consideremos nuevamente el problema del cuenco alargado: el descenso del gradiente comienza bajando rápidamente por la pendiente más pronunciada, que no apunta directamente hacia el óptimo global, y luego desciende muy lentamente hasta el fondo del valle. Sería bueno si el algoritmo pudiera corregir su dirección antes para apuntar un poco más hacia el óptimo global. El algoritmo AdaGrad logra esta corrección reduciendo el vector de gradiente a lo largo de las dimensiones más pronunciadas (ver [Ecuación 11-7](#)).<sup>17</sup>

Ecuación 11-7. Algoritmo AdaGrad

$$\begin{aligned} 1. \quad s &\leftarrow s + J(\theta) \quad J(\theta) \\ \eta J(\theta) & \quad \sqrt{s + \epsilon} \end{aligned}$$

El primer paso acumula el cuadrado de los gradientes en el vector  $s$  (recuerde que el símbolo  $\cdot$  representa la multiplicación de elementos). Esta forma vectorizada es equivalente a calcular  $s \leftarrow s + (\partial J(\theta) / \partial \theta)^T (\partial J(\theta) / \partial \theta)$  para cada elemento  $s_i$  del vector  $s$ ; es decir, cada  $s_i$  acumula los cuadrados de la derivada parcial de la función de costos con respecto al parámetro  $\theta_i$ . Si

Si la función de costo es pronunciada a lo largo de la dimensión  $i$ , entonces  $s_i$  se hará cada vez más grande en cada iteración.

El segundo paso es casi idéntico al descenso del gradiente, pero con una gran diferencia: el vector gradiente se reduce en un factor de  $\sqrt{s + \epsilon}$  (el símbolo  $\cdot$  representa la división por elementos y  $\epsilon$  es un término de suavizado para evitar la división por cero, normalmente establecido en 10). Esta forma vectorizada es equivalente a calcular simultáneamente  $\theta_i \leftarrow \theta_i - \eta \frac{\partial J(\theta)}{\partial \theta_i} / \sqrt{s_i + \epsilon}$  para todos los parámetros  $\theta_i$ .

En resumen, este algoritmo reduce la tasa de aprendizaje, pero lo hace más rápido para dimensiones pronunciadas que para dimensiones con pendientes más suaves. Esto se llama tasa de aprendizaje adaptativo. Ayuda a orientar las actualizaciones resultantes más directamente hacia el óptimo global (consulte [la Figura 11-8](#)). Un beneficio adicional es que requiere mucho menos ajuste del hiperparámetro de tasa de aprendizaje  $\eta$ .

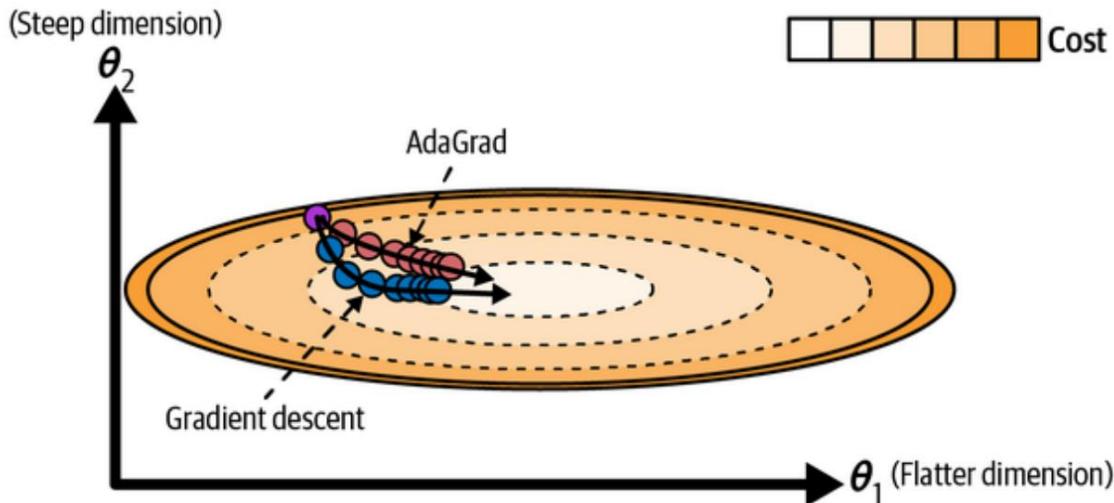


Figura 11-8. AdaGrad versus descenso de gradiente: el primero puede corregir su dirección antes para apuntar al óptimo

AdaGrad frecuentemente funciona bien para problemas cuadráticos simples, pero a menudo se detiene demasiado pronto al entrenar redes neuronales: la tasa de aprendizaje se reduce tanto que el algoritmo termina deteniéndose por completo antes de alcanzar el óptimo global. Entonces, aunque Keras tiene un optimizador Adagrad, no debe usarlo para entrenar redes neuronales profundas (aunque puede ser eficiente para tareas más simples como la regresión lineal). Aún así, comprender AdaGrad es útil para comprender los otros optimizadores de tasa de aprendizaje adaptativo.

## RMSProp

Como hemos visto, AdaGrad corre el riesgo de desacelerarse demasiado rápido y nunca converger al óptimo global. El algoritmo RMSProp soluciona este problema acumulando solo los gradientes de las iteraciones más recientes, en lugar de todos los gradientes desde el comienzo del entrenamiento. Lo hace utilizando una caída exponencial en el primer paso (consulte la ecuación 11-8).

Ecuación 11-8. Algoritmo RMSProp

$$\begin{aligned}
 1. \quad s &\leftarrow \rho s + (1 - \rho) J() \quad \underline{J()} \\
 2. \quad \leftarrow \eta J() \quad \sqrt{s} + \epsilon
 \end{aligned}$$

La tasa de caída  $\rho$  normalmente se establece en 0,9.<sup>19</sup> Sí, es nuevamente un hiperparámetro nuevo, pero este valor predeterminado a menudo funciona bien, por lo que es posible que no necesite ajustarlo en absoluto.

Como era de esperar, Keras tiene un optimizador RMSprop:

```
optimizador = tf.keras.optimizers.RMSprop(tasa_de_aprendizaje=0.001, rho=0.9)
```

Excepto en problemas muy simples, este optimizador casi siempre funciona mucho mejor que AdaGrad. De hecho, era el algoritmo de optimización preferido de muchos investigadores hasta que apareció la optimización de Adam.

## Adán

**Adán**, que significa estimación de momento adaptativo, combina las ideas de optimización de impulso y RMSProp: al igual que la optimización de momento, realiza un seguimiento de un promedio que decresce exponencialmente de gradientes pasados; y al igual que RMSProp, realiza un seguimiento de un promedio que decresce exponencialmente de gradientes cuadrados pasados (consulte la ecuación 11-9). Estas son estimaciones de la media y la varianza (no centrada) de los gradientes. La media suele denominarse primer momento, mientras que la varianza suele denominarse segundo momento, de ahí el nombre del algoritmo.

Ecuación 11-9. algoritmo de adán

$$\begin{aligned}
 1. \quad & m \leftarrow \beta_1 m - (1 - \beta_1) J() \\
 2. \quad & s \leftarrow \beta_2 s + (1 - \beta_2) J() \quad J() \\
 3. \quad & m \leftarrow 1 - \frac{\text{metro}}{\beta_1} \\
 4. \quad & \hat{s} \leftarrow \frac{s}{1 - \beta_2} \\
 5. \quad & \leftarrow + \eta \hat{m} - \sqrt{\hat{s}} + \epsilon
 \end{aligned}$$

En esta ecuación,  $t$  representa el número de iteración (comenzando en 1).

Si simplemente observa los pasos 1, 2 y 5, notará la gran similitud de Adam tanto con la optimización del impulso como con la RMSProp:  $\beta$  corresponde a  $\beta$  en la optimización del impulso y  $\beta$  corresponde a  $\rho$  en RMSProp. La única diferencia es que el paso 1 calcula un promedio que decrece exponencialmente en lugar de una suma que decrece exponencialmente, pero en realidad son equivalentes excepto por un factor constante (el promedio decreciente es solo  $1 - \beta$  veces la suma decreciente). Los pasos 3 y 4 son un detalle técnico: dado que  $mys$  se inicializan en 0, estarán sesgados hacia 0 al comienzo del entrenamiento, por lo que estos dos pasos ayudarán a impulsar  $mys$  al comienzo del entrenamiento.

1

El hiperparámetro  $\beta$  de caída de impulso normalmente se inicializa en 0,9, mientras que el hiperparámetro  $\beta$  de caída de escala a menudo se inicializa en 0,999. Como antes, el término de suavizado  $\epsilon$  generalmente se inicializa en un número pequeño como 10. Estos son los valores predeterminados para la clase Adam. Aquí se explica cómo cree un optimizador Adam usando Keras:

```
optimizador = tf.keras.optimizers.Adam(tasa_de_aprendizaje=0.001, beta_1=0.9,
                                       beta_2=0.999)
```

Dado que Adam es un algoritmo de tasa de aprendizaje adaptativo, como AdaGrad y RMSProp, requiere menos ajuste del hiperparámetro de tasa de aprendizaje  $\eta$ . A menudo puedes usar el valor predeterminado  $\eta = 0,001$ , lo que hace que Adam sea aún más fácil de usar que el descenso de gradiente.

## CONSEJO

Si está comenzando a sentirse abrumado por todas estas diferentes técnicas y se pregunta cómo elegir las adecuadas para su tarea, no se preocupe: al final de este capítulo se proporcionan algunas pautas prácticas.

Finalmente, vale la pena mencionar tres variantes de Adam: AdamMax, Nadam y AdamW.

## adamax

El artículo de Adam también presentó Adamax. Observe que en el paso 2 de la ecuación 11-9, Adam acumula los cuadrados de los gradientes en  $s$  (con un peso mayor para los gradientes más recientes). En el paso 5, si ignoramos  $\epsilon$  y los pasos 3 y 4 (que de todos modos son detalles técnicos), Adam reduce las actualizaciones de parámetros en la raíz cuadrada de  $s$ . En resumen, Adam reduce las actualizaciones de parámetros según la norma  $\ell$  de los gradientes decrecientes en el tiempo (recuerde que la norma  $\ell$  es la raíz cuadrada de la suma de cuadrados).

AdaMax reemplaza la norma  $\ell$  con la norma  $\ell$  (una forma elegante de decir el máximo). Específicamente, reemplaza el paso 2 en la ecuación

11-9 con  $\max(\beta_2 s, \text{abs}(J))$  reduce , baja del paso 4 y en el paso 5 escala las actualizaciones de gradiente en un factor de  $s$ , que es el máximo del valor absoluto de los gradientes decaídos en el tiempo.

En la práctica, esto puede hacer que AdaMax sea más estable que Adam, pero realmente depende del conjunto de datos y, en general, Adam funciona mejor. Entonces, este es solo un optimizador más que puedes probar si tienes problemas con Adam en alguna tarea.

## nadam

La optimización de Nadam es la optimización de Adam más el truco de Nesterov, por lo que a menudo convergerá un poco más rápido que Adam. En su informe que presenta esta técnica,<sup>21</sup> el investigador Timothy Dozat compara muchos optimizadores diferentes en diversas tareas y descubre que Nadam generalmente supera a Adam, pero a veces RMSProp lo supera.

## AdamW

AdamW<sup>22</sup> es una variante de Adam que integra una técnica de regularización llamada caída de peso. La caída de peso reduce el tamaño de los pesos del modelo en cada iteración de entrenamiento multiplicándolos por un factor de caída como 0,99. Esto puede recordarle a la regularización  $\ell_2$  (introducida en el Capítulo 4), que también apunta a mantener los pesos pequeños y, de hecho, se puede demostrar matemáticamente que la regularización  $\ell_2$  es equivalente a la disminución del peso cuando se usa SGD. Sin embargo, cuando se utiliza Adam o sus variantes, la regularización  $\ell_2$  y la caída de peso no son equivalentes: en la práctica, combinar Adam con la regularización  $\ell_2$  da como resultado modelos que a menudo no se generalizan tan bien como

producido por SGD. AdamW soluciona este problema combinando adecuadamente a Adam con la pérdida de peso.

#### ADVERTENCIA

Los métodos de optimización adaptativa (incluida la optimización RMSProp, Adam, AdamMax, Nadam y AdamW) suelen ser excelentes y convergen rápidamente hacia una buena solución. Sin embargo, un [artículo de 2017](#) por Ashia C. Wilson et al. demostró que pueden conducir a soluciones que se generalizan mal en algunos conjuntos de datos. Entonces, cuando esté decepcionado por el rendimiento de su modelo, intente usar NAG en su lugar: es posible que su conjunto de datos sea alérgico a los gradientes adaptativos. Consulte también las últimas investigaciones, porque avanzan rápidamente.

Para usar Nadam, AdaMax o AdamW en Keras, reemplace `tf.keras.optimizers.Adam` con `tf.keras.optimizers.Nadam`, `tf.keras.optimizers.Adamax` o `tf.keras.optimizers.experimental.AdamW`. Para AdamW, probablemente desee ajustar el hiperparámetro `Weight_decay`.

Todas las técnicas de optimización analizadas hasta ahora sólo se basan en las derivadas parciales de primer orden (jacobianas). La literatura sobre optimización también contiene algoritmos sorprendentes basados en las derivadas parciales de segundo orden (las hessianas, que son las derivadas parciales de las jacobianas).

Desafortunadamente, estos algoritmos son muy difíciles de aplicar a redes neuronales profundas porque hay  $n$  hessianos por salida (donde  $n$  es el número de parámetros), a diferencia de solo  $n$  jacobianos por salida. Dado que los DNN suelen tener decenas de miles de parámetros o más, los algoritmos de optimización de segundo orden a menudo ni siquiera caben en la memoria, e incluso cuando lo hacen, calcular los hessianos es demasiado lento.

## ENTRENAMIENTO DE MODELOS DISPAROS

Todos los algoritmos de optimización que acabamos de analizar producen modelos densos, lo que significa que la mayoría de los parámetros serán distintos de cero. Si necesita un modelo increíblemente rápido en tiempo de ejecución, o si necesita que ocupe menos memoria, es posible que prefiera terminar con un modelo disperso.

Una forma de lograr esto es entrenar el modelo como de costumbre y luego deshacerse de los pesos pequeños (ponerlos a cero). Sin embargo, esto normalmente no dará como resultado un modelo muy escaso y puede degradar el rendimiento del modelo.

Una mejor opción es aplicar una fuerte regularización  $\ell_1$  durante el entrenamiento (verá cómo más adelante en este capítulo), ya que empuja al optimizador a poner a cero tantos pesos como pueda (como se analiza en “Regresión de lazo”).

Si estas técnicas siguen siendo insuficientes, consulte el kit de herramientas de optimización de modelos [de TensorFlow \(TF-MOT\)](#), que proporciona una API de poda capaz de eliminar conexiones de forma iterativa durante el entrenamiento en función de su magnitud.

La Tabla 11-2 compara todos los optimizadores que hemos analizado hasta ahora (\* es malo, es promedio y \*\* es bueno).

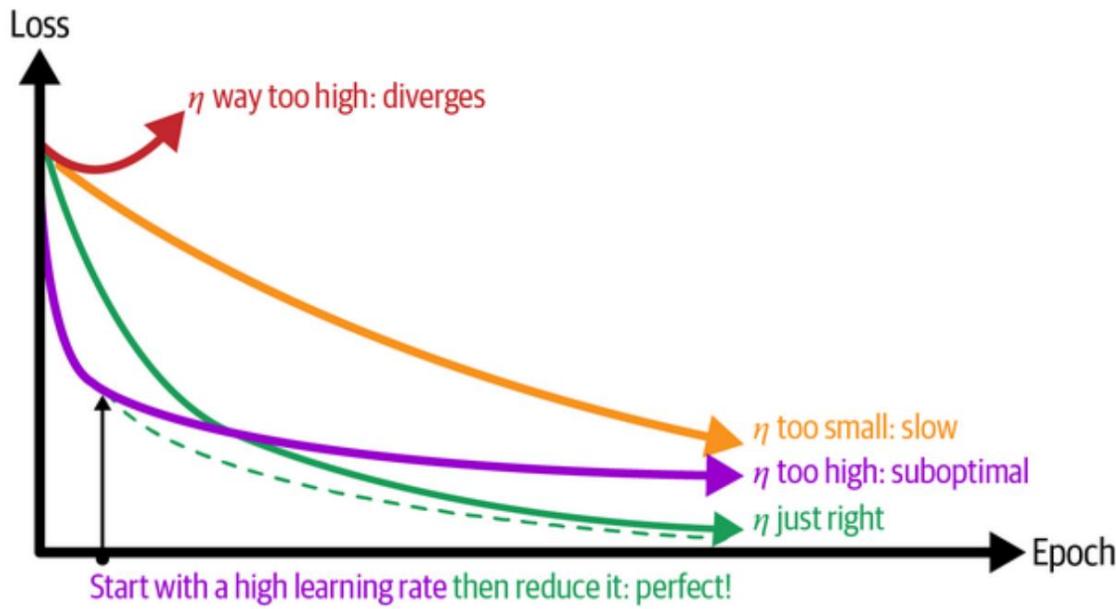
\*\*

Tabla 11-2. Comparación de optimizadores

Clase	Velocidad de convergencia	Calidad de convergencia
SGD	*	***
SGD(impulso=...)	**	***
SGD(impulso=..., nesterov=True)	**	***
Adagrad	***	* (se detiene demasiado pronto)
RMSprop	***	** O ***
Adán	***	** O ***
adamax	***	** O ***
nadam	***	** O ***
AdamW	***	** O ***

## Programación de la tasa de aprendizaje

Encontrar una buena tasa de aprendizaje es muy importante. Si lo pones demasiado alto, el entrenamiento puede divergir (como se analiza en “[Descenso de gradiente](#)”). Si lo configuras también bajo, el entrenamiento eventualmente convergerá al óptimo, pero tomará un tiempo muy largo tiempo. Si lo configura un poco demasiado alto, progresará muy rápidamente en primer lugar, pero terminará bailando alrededor del óptimo y nunca realmente estableciéndose abajo. Si tiene un presupuesto informático limitado, es posible que deba interrumpir el entrenamiento antes de que haya convergido adecuadamente, produciendo una solución subóptima (ver [Figura 11-9](#)).

Figura 11-9. Curvas de aprendizaje para varias tasas de aprendizaje  $\eta$ 

Como se analizó en el Capítulo 10, puede encontrar una buena tasa de aprendizaje entrenando el modelo durante unos cientos de iteraciones, aumentando exponencialmente la tasa de aprendizaje desde un valor muy pequeño a un valor muy grande, y luego observando la curva de aprendizaje y eligiendo una tasa de aprendizaje ligeramente inferior a aquella en la que la curva de aprendizaje comienza a dispararse. Luego puedes reinicializar tu modelo y entrenarlo con esa tasa de aprendizaje.

Pero puede hacerlo mejor que una tasa de aprendizaje constante: si comienza con una tasa de aprendizaje alta y luego la reduce una vez que el entrenamiento deja de progresar rápidamente, puede alcanzar una buena solución más rápido que con la tasa de aprendizaje constante óptima.

Existen muchas estrategias diferentes para reducir la tasa de aprendizaje durante el entrenamiento. También puede resultar beneficioso comenzar con una tasa de aprendizaje baja, aumentarla y luego bajarla nuevamente. Estas estrategias se denominan programas de aprendizaje (introduce brevemente este concepto en el Capítulo 4). Estos son los horarios de aprendizaje más utilizados:

#### Programación de energía

Establezca la tasa de aprendizaje en función del número de iteración  $t$ :  $\eta(t) = \eta_0 / (1 + t/s)$ . La tasa de aprendizaje inicial  $\eta_0$ , la potencia  $c$  (normalmente establecida en 1) y los pasos  $s$  son hiperparámetros. La tasa de aprendizaje cae en cada paso.

Después de  $s$  pasos, la tasa de aprendizaje baja a  $\eta_0/2$ . Después de  $s$  pasos más, baja a  $\eta_0/3$ , luego baja a  $\eta_0/4$ , luego  $\eta_0/5$ , y así sucesivamente. Como puede ver, este horario primero disminuye rápidamente, luego cada vez más

despacio. Por supuesto, la programación de energía requiere ajustar  $\eta$  y  $\gamma$ s (y posiblemente  $c$ ).

### Programación exponencial

Establezca la tasa de aprendizaje en  $\eta(t) \propto \eta(0) e^{-t/s}$ . La tasa de aprendizaje disminuirá gradualmente en un factor de 10 en cada paso. Mientras que la programación energética reduce la tasa de aprendizaje cada vez más lentamente, la programación exponencial sigue reduciéndola en un factor de 10 en cada s pasos.

### Programación constante por partes

Utilice una tasa de aprendizaje constante para varias épocas (p. ej.,  $\eta = 0,1$  para 5 épocas), luego una tasa de aprendizaje menor para otra cantidad de épocas (p. ej.,  $\eta = 0,001$  para 50 épocas), y así sucesivamente. Aunque esta solución puede funcionar muy bien, requiere experimentar para determinar la secuencia correcta de tasas de aprendizaje y cuánto tiempo usar cada una de ellas.

### Programación de desempeño

Mida el error de validación cada N pasos (al igual que para la parada anticipada) y reduzca la tasa de aprendizaje en un factor de  $\lambda$  cuando el error deje de disminuir.

### programación de 1 ciclo

1 ciclo se presentó en un [artículo de 2018](#), por Leslie Smith. A diferencia de los otros enfoques, comienza aumentando la tasa de aprendizaje inicial  $\eta$ , creciendo linealmente hasta  $\eta$  a mitad del entrenamiento. Luego disminuye la tasa de aprendizaje linealmente hasta  $\eta$  nuevamente durante la segunda mitad del entrenamiento, terminando las últimas épocas reduciendo la tasa en varios órdenes de magnitud (aún linealmente). La tasa de aprendizaje máxima  $\eta$  se elige utilizando el mismo enfoque que utilizamos para encontrar la tasa de aprendizaje óptima, y la tasa de aprendizaje inicial  $\eta$  suele ser 10 veces menor.

0

Cuando usamos un impulso, primero comenzamos con un impulso alto (p. ej., 0,95), luego lo bajamos a un impulso más bajo durante la primera mitad del entrenamiento (p. ej., hasta 0,85, linealmente) y luego lo volvemos a subir al nivel inicial. valor máximo (por ejemplo, 0,95) durante la segunda mitad del entrenamiento, terminando las últimas épocas con ese valor máximo. Smith hizo muchos experimentos que demostraron que este enfoque a menudo podía acelerar

entrenar considerablemente y alcanzar un mejor rendimiento. Por ejemplo, en el popular conjunto de datos de imágenes CIFAR10, este enfoque alcanzó una precisión de validación del 91,9 % en solo 100 épocas, en comparación con una precisión del 90,3 % en 800 épocas mediante un enfoque estándar (con la misma arquitectura de red neuronal). Esta hazaña se denominó superconvergencia.

Un [artículo de 2013](#) por Andrew Senior et al. <sup>25</sup> comparó el rendimiento de algunos de los programas de aprendizaje más populares cuando se utiliza la optimización del impulso para entrenar redes neuronales profundas para el reconocimiento de voz. Los autores concluyeron que, en este contexto, tanto la programación de rendimiento como la programación exponencial funcionaron bien. Favorecían la programación exponencial porque era fácil de ajustar y convergía ligeramente más rápido hacia la solución óptima. También mencionaron que era más fácil de implementar que la programación del rendimiento, pero en Keras ambas opciones son fáciles. Dicho esto, el enfoque de 1 ciclo parece funcionar aún mejor.

Implementar la programación de energía en Keras es la opción más sencilla: simplemente configure el hiperparámetro de caída al crear un optimizador:

```
optimizador = tf.keras.optimizers.SGD(tasa_de_aprendizaje=0.01,
decamiento=1e-4)
```

La caída es la inversa de s (el número de pasos que se necesitan para dividir la tasa de aprendizaje por una unidad más), y Keras supone que c es igual a 1.

La programación exponencial y la programación por partes también son bastante simples. Primero debe definir una función que tome la época actual y devuelva la tasa de aprendizaje. Por ejemplo, implementemos la programación exponencial:

```
def exponencial_decay_fn(época):
    devolver 0.01 * 0.1 ** (época / 20)
```

Si no desea codificar  $\eta$  y s, puede crear una función que devuelva una función configurada:

```
def decadencia_exponencial(lr0, s):
    def exponencial_decay_fn(época):
        devolver lr0 * 0.1 ** (época / s) devolver
    exponencial_decay_fn
```

```
decadencia_exponencial_fn = decadencia_exponencial(lr0=0.01, s=20)
```

A continuación, cree una devolución de llamada de LearningRateScheduler, dándole la función de programación y pase esta devolución de llamada al método fit():

```
lr_scheduler =
tf.keras.callbacks.LearningRateScheduler(exponential_decay_fn) historial =
model.fit(X_train, y_train, [...], devoluciones de llamada= [lr_scheduler])
```

LearningRateScheduler actualizará el atributo learning\_rate del optimizador al comienzo de cada época. Actualizar la tasa de aprendizaje una vez por época suele ser suficiente, pero si desea que se actualice con más frecuencia, por ejemplo en cada paso, siempre puede escribir su propia devolución de llamada (consulte la sección "Programación exponencial" del cuaderno de este capítulo para ver un ejemplo). ). Actualizar la tasa de aprendizaje en cada paso puede ser útil si hay muchos pasos por época. Alternativamente, puede utilizar el enfoque tf.keras.optimizers.schedules, que se describe en breve.

#### CONSEJO

Después del entrenamiento, History.history["lr"] le brinda acceso a la lista de tasas de aprendizaje utilizadas durante el entrenamiento.

Opcionalmente, la función de programación puede tomar la tasa de aprendizaje actual como segundo argumento. Por ejemplo, la siguiente función de programación multiplica la tasa de aprendizaje anterior por 0,1, lo que da como resultado la misma caída exponencial (excepto que la caída ahora comienza al comienzo de la época 0 en lugar de 1):

```
def exponential_decay_fn(epoca, lr): devolver lr *
    0.1 ** (1/20)
```

Esta implementación se basa en la tasa de aprendizaje inicial del optimizador (a diferencia de la implementación anterior), así que asegúrese de configurarla adecuadamente.

Cuando guarda un modelo, el optimizador y su tasa de aprendizaje se guardan junto con él. Esto significa que con esta nueva función de programación, puede simplemente cargar un modelo entrenado y continuar entrenando donde lo dejó, sin problema. Sin embargo, las cosas no son tan simples si su función de programación usa el argumento epoch: la época no se guarda y se restablece a 0 cada vez que llama al método fit(). Si continuara entrenando un modelo donde lo dejó, esto podría generar una tasa de aprendizaje muy alta, lo que probablemente dañaría los pesos de su modelo. Una solución es configurar manualmente el argumento de la época\_inicial del método fit() para que la época comience en el valor correcto.

Para una programación constante por partes, puede usar una función de programación como la siguiente (como antes, puede definir una función más general si lo desea; consulte la sección "Programación constante por partes" del cuaderno para ver un ejemplo), luego cree un LearningRateScheduler Devuelve la llamada con esta función y pásala al método fit(), al igual que para la programación exponencial:

```
def piecewise_constant_fn(época): si época < 5:  
    devuelve 0,01  
  
    época elif < 15: devuelve  
        0,005  
    demás:  
        devolver 0.001
```

Para la programación del rendimiento, utilice la devolución de llamada ReduceLROnPlateau. Por ejemplo, si pasa la siguiente devolución de llamada al método fit(), multiplicará la tasa de aprendizaje por 0,5 siempre que la mejor pérdida de validación no mejore durante cinco épocas consecutivas (hay otras opciones disponibles; consulte la documentación para obtener más detalles). :

```
lr_scheduler = tf.keras.callbacks.ReduceLROnPlateau(factor=0.5, paciencia=5) historial =  
model.fit(X_train,  
y_train, [...], devoluciones de llamada= [lr_scheduler])
```

Por último, Keras ofrece una forma alternativa de implementar la programación de la tasa de aprendizaje: puede definir una tasa de aprendizaje programada utilizando uno de los

clases disponibles en `tf.keras.optimizers.schedules`, luego páselo a cualquier optimizador. Este enfoque actualiza la tasa de aprendizaje en cada paso en lugar de en cada época. Por ejemplo, aquí se explica cómo implementar el mismo programa exponencial que la función `exponential_decay_fn()` que definimos anteriormente:

```
importar matematicas

tamaño_por_lotes =
32 n_épocas = 25
n_pasos = n_épocas * math.ceil(len(X_train) / tamaño_por_lotes)
tasa_de_aprendizaje_programada
= tf.keras.optimizers.schedules.ExponentialDecay(
    tasa_de_aprendizaje_inicial=0.01, pasos_decaimiento=n_pasos,
    tasa_de_decaimiento=0.1) optimizador = tf.keras.optimizers.SGD(tasa_de_aprendizaje=tasa_de_aprendizaje_programada)
```

Esto es agradable y simple, además, cuando guardas el modelo, la tasa de aprendizaje y su cronograma (incluido su estado) también se guardan.

En cuanto a 1cycle, Keras no lo admite, pero es posible implementarlo en menos de 30 líneas de código creando una devolución de llamada personalizada que modifica la tasa de aprendizaje en cada iteración. Para actualizar la tasa de aprendizaje del optimizador desde el método `on_batch_begin()` de la devolución de llamada, debe llamar `tf.keras`.

`backend.set_value(self.model.optimizer.learning_rate, new_learning_rate)`. Consulte la sección "Programación de 1 ciclo" del cuaderno para ver un ejemplo.

En resumen, la caída exponencial, la programación del rendimiento y 1 ciclo pueden acelerar considerablemente la convergencia, ¡así que pruébalos!

## Evitar el sobreajuste mediante la regularización

Con cuatro parámetros puedo encajar un elefante y con cinco puedo hacerle mover la trompa.

—John von Neumann, citado por Enrico Fermi en Nature 427

Con miles de parámetros, puedes adaptar todo el zoológico. Las redes neuronales profundas suelen tener decenas de miles de parámetros, a veces incluso millones. Esto les brinda una increíble libertad y significa que pueden adaptarse a una gran variedad de conjuntos de datos complejos. Pero esta gran flexibilidad también hace que la red sea propensa a sobreajustar el conjunto de entrenamiento. A menudo es necesaria la regularización para evitar esto.

Ya implementamos una de las mejores técnicas de regularización en el [Capítulo 10](#): la parada anticipada. Además, aunque la normalización por lotes se diseñó para resolver los problemas de gradientes inestables, también actúa como un regularizador bastante bueno. En esta sección examinaremos otras técnicas de regularización populares para redes neuronales: regularización  $\ell_1$  y  $\ell_2$ , abandono y regularización de norma máxima.

2

## $\ell_1$ y $\ell_2$ Regularización

Tal como lo hizo en [el Capítulo 4](#) para modelos lineales simples, puede usar la regularización  $\ell_2$  para restringir los pesos de conexión de una red neuronal y/o la regularización  $\ell_1$  si desea un modelo disperso (con muchos pesos iguales a 0).

1

A continuación se explica cómo aplicar la regularización  $\ell_2$  a los pesos de conexión de una capa de Keras, utilizando un factor de regularización de 0,01:

```
capa = tf.keras.layers.Dense(100, activación="relu",
                             kernel_initializer="he_normal",
                             kernel_regularizer=tf.keras.regularizers.l2(0.01))
```

La función `l2()` devuelve un regularizador que se llamará en cada paso durante el entrenamiento para calcular la pérdida de regularización. Esto luego se suma a la pérdida final. Como era de esperar, puede usar `tf.keras.regularizers.l1()` si desea una regularización  $\ell_1$ ; si desea regularización tanto  $\ell_1$  como  $\ell_2$ , use `tf.keras.regularizers.l1_l2()` (especificando ambos factores de regularización).

Dado que normalmente querrás aplicar el mismo regularizador a todas las capas de tu red, además de utilizar la misma función de activación y la misma estrategia de inicialización en todas las capas ocultas, es posible que te encuentres repitiendo los mismos argumentos. Esto hace que el código sea feo y propenso a errores. Para evitar

esto, puedes intentar refactorizar tu código para usar bucles. Otra opción es usar la función `functools.partial()` de Python, que le permite crear un contenedor delgado para cualquier invocable, con algunos valores de argumento predeterminados:

```
desde functools importar parcial

RegularizedDense = parcial(tf.keras.layers.Dense, activación="relu",
                           kernel_initializer="he_normal",

kernel_regularizer=tf.keras.regularizers.l2(0.01))

modelo = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]), RegularizedDense(100),
    RegularizedDense(100),
    RegularizedDense(10,
                     activación="softmax")
])
```

### ADVERTENCIA

Como vimos anteriormente, la regularización  $\ell_1$  está bien cuando se usa SGD, optimización de impulso y optimización de impulso de Nesterov, pero no con Adam y sus variantes. Si desea utilizar Adam con disminución de peso, no utilice la regularización  $\ell_1$ : utilice AdamW en su lugar.

## Abandonar

El abandono es una de las técnicas de regularización más populares para redes neuronales profundas. Fue [propuesto en un documento](#)<sup>26</sup> por Geoffrey Hinton et al. en 2012 y más detallado en un [documento de 2014](#)<sup>27</sup> por Nitish Srivastava et al., y ha demostrado ser muy exitoso: muchas redes neuronales de última generación utilizan la deserción, ya que les da un aumento de precisión del 1% al 2%. Puede que esto no parezca mucho, pero cuando un modelo ya tiene un 95% de precisión, obtener un aumento de precisión del 2% significa reducir la tasa de error en casi un 40% (pasando del 5% de error a aproximadamente el 3%).

Es un algoritmo bastante simple: en cada paso de entrenamiento, cada neurona (incluidas las de entrada, pero siempre excluyendo las de salida) tiene una probabilidad  $p$  de ser "abandonada" temporalmente, lo que significa que será completamente eliminada.

ignorado durante este paso de entrenamiento, pero puede estar activo durante el siguiente paso (consulte [la Figura 11-10](#)). El hiperparámetro  $p$  se denomina tasa de abandono y normalmente se establece entre el 10 % y el 50 %: más cerca del 20 %-30 % en las redes neuronales recurrentes (consulte el [Capítulo 15](#)) y más cerca del 40 %-50 % en las redes neuronales convolucionales. (ver [Capítulo 14](#)). Después del entrenamiento, las neuronas ya no se caen. Y eso es todo (excepto un detalle técnico que comentaremos en un momento).

Al principio sorprende que esta técnica destructiva funcione. ¿Se desempeñaría mejor una empresa si a sus empleados se les dijera que lanzaran una moneda al aire todas las mañanas para decidir si van o no a trabajar? Bueno, quién sabe; ¡tal vez lo haría! La empresa se vería obligada a adaptar su organización; no podía depender de una sola persona para manejar la máquina de café o realizar otras tareas críticas, por lo que esta experiencia tendría que repartirse entre varias personas. Los empleados tendrían que aprender a cooperar con muchos de sus compañeros de trabajo, no sólo con un puñado de ellos. La empresa se volvería mucho más resistente. Si una persona renunciara, no haría mucha diferencia. No está claro si esta idea realmente funcionaría para las empresas, pero ciertamente funciona para las redes neuronales. Las neuronas entrenadas con abandono no pueden coadaptarse con sus neuronas vecinas; tienen que ser lo más útiles posible por sí solos. Tampoco pueden depender excesivamente de unas pocas neuronas de entrada; deben prestar atención a cada una de sus neuronas de entrada. Terminan siendo menos sensibles a ligeros cambios en las entradas.

Al final, se obtiene una red más sólida que se generaliza mejor.

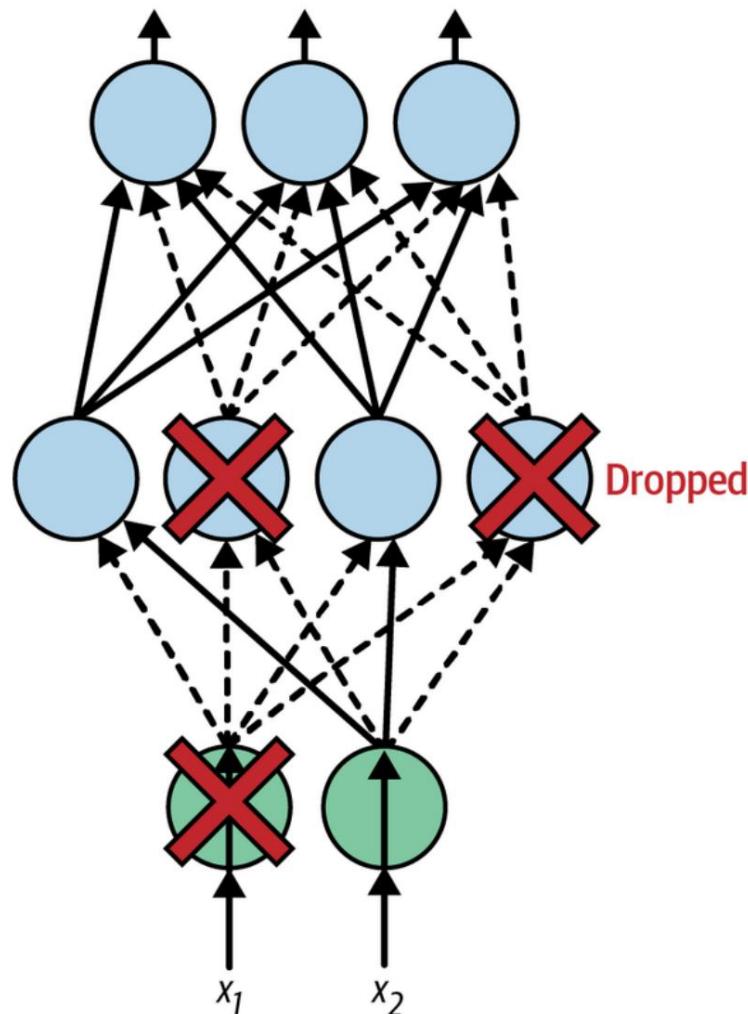


Figura 11-10. Con la regularización de abandono, en cada iteración de entrenamiento, se "abandona" un subconjunto aleatorio de todas las neuronas en una o más capas, excepto la capa de salida; estas neuronas generan 0 en esta iteración (representada por las flechas discontinuas)

Otra forma de comprender el poder del abandono es darse cuenta de que en cada paso del entrenamiento se genera una red neuronal única. Dado que cada neurona puede estar presente o ausente, hay un total de  $2^N$  redes posibles (donde  $N$  es el número total de neuronas que se pueden descartar). Se trata de un número tan grande que es prácticamente imposible muestrear dos veces la misma red neuronal. Una vez que haya ejecutado 10.000 pasos de entrenamiento, esencialmente habrá entrenado 10.000 redes neuronales diferentes, cada una con una sola instancia de entrenamiento. Obviamente, estas redes neuronales no son independientes porque comparten muchos de sus pesos, pero, aun así, todas son diferentes. La red neuronal resultante puede verse como un conjunto promedio de todas estas redes neuronales más pequeñas.

## CONSEJO

En la práctica, normalmente se puede aplicar la exclusión solo a las neuronas de una a tres capas superiores (excluyendo la capa de salida).

Hay un pequeño pero importante detalle técnico. Supongamos  $p = 75\%$ : en promedio, solo el 25% de todas las neuronas están activas en cada paso durante el entrenamiento. Esto significa que después del entrenamiento, una neurona estaría conectada a cuatro veces más neuronas de entrada que durante el entrenamiento. Para compensar este hecho, necesitamos multiplicar los pesos de las conexiones de entrada de cada neurona por cuatro durante el entrenamiento. Si no lo hacemos, la red neuronal no funcionará bien ya que verá datos diferentes durante y después del entrenamiento. De manera más general, necesitamos dividir los pesos de las conexiones por la probabilidad de mantenimiento ( $1 - p$ ) durante el entrenamiento.

Para implementar el abandono usando Keras, puede usar la capa `tf.keras.layers.Dropout`. Durante el entrenamiento, elimina aleatoriamente algunas entradas (estableciéndolas en 0) y divide las entradas restantes por la probabilidad de mantener. Después del entrenamiento, no hace nada en absoluto; simplemente pasa las entradas a la siguiente capa. El siguiente código aplica la regularización de abandono antes de cada capa densa, utilizando una tasa de abandono de 0,2:

```
modelo = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dropout(rate=0.2),
    tf.keras.layers.Dense(100, activación="relu", kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(rate=0.2),
    tf.keras.layers.Dense(100, activación="relu",
                         kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(tasa=0.2),
    tf.keras.layers.Dense(10, activación="softmax")
])
[...] # compilar y entrenar el modelo
```

## ADVERTENCIA

Dado que el abandono solo está activo durante el entrenamiento, comparar la pérdida de entrenamiento y la pérdida de validación puede resultar engañoso. En particular, un modelo puede estar sobreajustando el conjunto de entrenamiento y aun así tener pérdidas de entrenamiento y validación similares. Por lo tanto, asegúrese de evaluar la pérdida de entrenamiento sin abandono (por ejemplo, después del entrenamiento).

Si observa que el modelo se está sobreajustando, puede aumentar la tasa de abandono. Por el contrario, deberías intentar reducir la tasa de abandono si el modelo no se ajusta al conjunto de entrenamiento. También puede ayudar a aumentar la tasa de abandono para ponedoras grandes y reducirla para las pequeñas. Además, muchas arquitecturas de última generación solo utilizan la eliminación después de la última capa oculta, por lo que es posible que desee probar esto si la eliminación total es demasiado fuerte.

La deserción tiende a ralentizar significativamente la convergencia, pero a menudo da como resultado un mejor modelo cuando se ajusta correctamente. Por lo tanto, generalmente vale la pena invertir tiempo y esfuerzo extra, especialmente en modelos grandes.

## CONSEJO

Si desea regularizar una red autonormalizada basada en la función de activación SELU (como se discutió anteriormente), debe usar la deserción alfa: esta es una variante de la deserción que preserva la media y la desviación estándar de sus entradas. Se introdujo en el mismo artículo que SELU, ya que la deserción escolar regular rompería la autonormalización.

## Montecarlo (MC) Abandono

En 2016, un [artículo](#)<sup>28</sup> por Yarin Gal y Zoubin Ghahramani agregaron algunas buenas razones más para utilizar la deserción escolar:

- En primer lugar, el artículo estableció una conexión profunda entre las redes de abandono (es decir, redes neuronales que contienen capas de abandono) y la inferencia bayesiana aproximada, dando al abandono [una justificación matemática sólida](#).
- En segundo lugar, los autores introdujeron una poderosa técnica llamada abandono de MC, que puede mejorar el rendimiento de cualquier desertor entrenado.

modelo sin tener que volver a entrenarlo o incluso modificarlo en absoluto. También proporciona una medida mucho mejor de la incertidumbre del modelo y puede implementarse en tan sólo unas pocas líneas de código.

Si todo esto suena como un clickbait de “un truco extraño”, entonces eche un vistazo al siguiente código. Es la plena implementación del abandono de MC, impulsando la modelo de abandono que entrenamos anteriormente sin volver a entrenarlo:

importar numpy como np

```
y_probas = np.stack([modelo(X_test, entrenamiento=True)  
                     para muestra en el rango(100)])  
y_proba = y_probas.media(eje=0)
```

Tenga en cuenta que `model(X)` es similar a `model.predict(X)` excepto que devuelve un tensor en lugar de una matriz NumPy, y admite el entrenamiento argumento. En este ejemplo de código, establecer `Training=True` garantiza que el La capa de abandono permanece activa, por lo que todas las predicciones serán un poco diferentes. Nosotros simplemente haga 100 predicciones sobre el conjunto de prueba y calcularemos su promedio. Más específicamente, cada llamada al modelo devuelve una matriz con una fila por instancia y una columna por clase. Porque hay 10.000 casos en el conjunto de prueba y 10 clases, esta es una matriz de forma [10000, 10]. apilamos 100 de esas matrices, por lo que `y_probas` es una matriz 3D de forma [100, 10000, 10]. Una vez que promediamos la primera dimensión (eje = 0), obtenemos `y_proba`, un matriz de forma [10000, 10], como la que obtendríamos con una sola predicción. ¡Eso es todo! Promediar múltiples predicciones con la deserción activada da una estimación de Monte Carlo que generalmente es más confiable que el resultado de una única predicción con el abandono desactivado. Por ejemplo, veamos el predicción del modelo para la primera instancia en el conjunto de pruebas Fashion MNIST, con abandono desactivado:

```
>>> modelo.predict(X_test[:1]).ronda(3)
matriz([[0. , 0. , 0. , 0. , 0. , 0.024, 0. , 0.132,
        ,
        0.844]], dtype=float32)
```

El modelo está bastante seguro (84,4%) de que esta imagen pertenece a la clase 9. (botín). Compare esto con la predicción de abandono de MC:

```
>>> y_proba[0].ronda(3)
matriz([0. , 0. 0.001, 0. , 0. , 0. , 0. , 0.067, 0. , 0.209,
       0.723], dtype=float32)
```

El modelo todavía parece preferir la clase 9, pero su confianza cayó a 72,3%, y las probabilidades estimadas para las clases 5 (sandalias) y 7 (zapatillas de deporte) han aumentado, lo cual tiene sentido dado que también son calzado.

La abandono de MC tiende a mejorar la confiabilidad de la probabilidad del modelo estimados. Esto significa que es menos probable que tenga confianza pero se equivoque, lo que puede ser peligroso: imagínese un automóvil autónomo ignorando con confianza señal de stop. También es útil saber exactamente qué otras clases son más probable. Además, puedes echar un vistazo a la **desviación estándar de la estimaciones de probabilidad**:

```
>>> y_std = y_probas.std(eje=0)
>>> y_std[0].ronda(3)
matriz([0. , 0. 0.001, 0. , 0.001, 0. , 0.096, 0. , 0.162,
       0.183], dtype=float32)
```

Aparentemente hay mucha variación en las estimaciones de probabilidad para clase 9: la desviación estándar es 0,183, que debe compararse con la probabilidad estimada de 0,723: si estuviera construyendo un sistema sensible al riesgo (por ejemplo, un sistema médico o financiero), probablemente trataría tal situación predicción incierta con extrema precaución. Definitivamente no lo tratarías como una predicción segura del 84,4%. La precisión del modelo también obtuvo un (muy) pequeño aumento del 87,0% al 87,2%:

```
>>> y_pred = y_proba.argmax(eje=1)
>>> precisión = (y_pred == y_test).sum() / len(y_test)
>>> precisión
0.8717
```

## NOTA

La cantidad de muestras de Monte Carlo que utiliza (100 en este ejemplo) es un hiperparámetro que puede modificar. Cuanto mayor sea, más precisas serán las predicciones y sus estimaciones de incertidumbre. Sin embargo, si lo duplica, el tiempo de inferencia también se duplicará. Además, a partir de un cierto número de muestras, notará pocas mejoras. Su trabajo es encontrar el equilibrio adecuado entre latencia y precisión, según su aplicación.

Si su modelo contiene otras capas que se comportan de una manera especial durante el entrenamiento (como las capas BatchNormalization), entonces no debe forzar el modo de entrenamiento como acabamos de hacer. En su lugar, deberías reemplazar las capas de abandono con la siguiente clase MCDropout:

30

```
clase MCDropout(tf.keras.layers.Dropout):
    llamada def (auto, entradas, entrenamiento = Falso):
        devolver super().call(entradas, entrenamiento=Verdadero)
```

Aquí, simplemente subclasificamos la capa Dropout y anulamos el método call() para forzar su argumento de entrenamiento a Verdadero (consulte el [Capítulo 12](#)).

De manera similar, podría definir una clase MCAlphaDropout subclasicando AlphaDropout. Si está creando un modelo desde cero, solo es cuestión de usar MCDropout en lugar de Dropout. Pero si tiene un modelo que ya fue entrenado usando Dropout, necesita crear un nuevo modelo que sea idéntico al modelo existente excepto con Dropout en lugar de MCDropout, luego copie los pesos del modelo existente a su nuevo modelo.

En resumen, la deserción de MC es una gran técnica que impulsa los modelos de deserción y proporciona mejores estimaciones de incertidumbre. Y por supuesto, al tratarse de un abandono habitual durante el entrenamiento, también actúa como regularizador.

## Regularización de norma máxima

Otra técnica de regularización popular para redes neuronales se llama regularización de norma máxima: para cada neurona, restringe los pesos  $w$  de las conexiones entrantes de modo que  $\|w\| \leq r$ , donde  $r$  es el hiperparámetro de norma máxima y  $\|\cdot\|$  es la  $\ell_p$  norma.

La regularización de norma máxima no agrega un término de pérdida de regularización a la función de pérdida general. En cambio, normalmente se implementa calculando  $w$  después de cada paso de entrenamiento y reescalando  $w$  si es necesario ( $w \leftarrow w / \|w\|_2$ ).

Reducir  $r$  aumenta la cantidad de regularización y ayuda a reducir el sobreajuste. La regularización de norma máxima también puede ayudar a aliviar los problemas de gradientes inestables (si no está utilizando la normalización por lotes).

Para implementar la regularización de norma máxima en Keras, establezca el argumento `kernel_constraint` de cada capa oculta en una restricción `max_norm()` con el valor máximo apropiado, así:

```
denso = tf.keras.layers.Dense(
    100, activation="relu", kernel_initializer="he_normal",
    kernel_constraint=tf.keras.constraints.max_norm(1.))
```

Después de cada iteración de entrenamiento, el método `fit()` del modelo llamará al objeto devuelto por `max_norm()`, le pasará los pesos de la capa y obtendrá a cambio pesos reescalados, que luego reemplazan los pesos de la capa. Como verá en [el Capítulo 12](#), puede definir su propia función de restricción personalizada si es necesario y utilizarla como `kernel_constraint`. También puede restringir los términos de sesgo configurando el argumento `sesgo_constraint`.

La función `max_norm()` tiene un argumento de eje cuyo valor predeterminado es 0. Una capa Densa generalmente tiene pesos de forma [número de entradas, número de neuronas], por lo que usar `eje=0` significa que la restricción de norma máxima se aplicará de forma independiente a cada neurona. vector de peso. Si desea utilizar max-norm con capas convolucionales (consulte [el Capítulo 14](#)), asegúrese de establecer apropiadamente el argumento del eje de la restricción `max_norm()` (generalmente `eje= [0, 1, 2]`).

## Resumen y directrices prácticas

En este capítulo hemos cubierto una amplia gama de técnicas y quizás se pregunte cuáles debería utilizar. Esto depende de la tarea y todavía no hay un consenso claro, pero he encontrado que la configuración en [la Tabla 11-3](#) funciona bien en la mayoría de los casos, sin requerir mucho

ajuste de hiperparámetros. Dicho esto, no considere estos valores predeterminados.  
¡Como reglas duras!

Tabla 11-3. Configuración DNN predeterminada

Hiperparámetro	Valor por defecto
Inicializador del kernel	Él inicialización
Función de activación	ReLU si es poco profundo; Swish si es profundo
Normalización	Ninguno si es poco profundo; norma de lote si es profundo
Regularización	Parada anticipada; caída de peso si es necesario
Optimizador	Nesterov aceleró gradientes o AdamW
Programación de tasa de aprendizaje	Programación de desempeño o 1 ciclo

Si la red es una simple pila de capas densas, entonces puede autonormalizarse, y debería utilizar la configuración de la [Tabla 11-4](#) en su lugar.

Tabla 11-4. Configuración DNN para una red autonormalizada

Hiperparámetro	Valor por defecto
Inicializador del kernel	Inicialización de LeCun
Función de activación	SELU
Normalización	Ninguno (autonormalización)
Regularización	Abandono alfa si es necesario
Optimizador	Nesterov aceleró gradientes
Programación de tasa de aprendizaje	Programación de desempeño o 1 ciclo

¡No olvides normalizar las funciones de entrada! También deberías intentar reutilizar partes de una red neuronal previamente entrenada si puede encontrar una que resuelva un problema similar, o utilice un entrenamiento previo no supervisado si tiene mucha

datos sin etiquetar o utilice el entrenamiento previo en una tarea auxiliar si tiene muchos datos etiquetados para una tarea similar.

Si bien las pautas anteriores deberían cubrir la mayoría de los casos, aquí hay algunas excepciones:

- Si necesita un modelo disperso, puede usar la regularización  $\ell$  (y opcionalmente poner a cero los pesos pequeños después del entrenamiento). Si necesita un modelo aún más reducido, puede utilizar el kit de herramientas de optimización de modelos de TensorFlow. Esto romperá la autonormalización, por lo que deberías usar la configuración predeterminada en este caso.
- Si necesita un modelo de baja latencia (uno que realice predicciones ultrarrápidas), es posible que necesite usar menos capas, usar una función de activación rápida como ReLU o ReLU con fugas y doblar las capas de normalización por lotes en las capas anteriores después del entrenamiento. . Tener un modelo escaso también ayudará. Finalmente, es posible que desee reducir la precisión flotante de 32 bits a 16 o incluso 8 bits (consulte “[Implementación de un modelo en un dispositivo móvil o integrado](#)”). Nuevamente, consulte TF-MOT.
- Si está creando una aplicación sensible al riesgo o la latencia de inferencia no es muy importante en su aplicación, puede utilizar la deserción de MC para mejorar el rendimiento y obtener estimaciones de probabilidad más confiables, junto con estimaciones de incertidumbre.

¡Con estas pautas ya estás preparado para entrenar redes muy profundas! Espero que ahora esté convencido de que puede recorrer un largo camino utilizando sólo la conveniente API de Keras. Sin embargo, puede llegar un momento en el que necesites tener aún más control; por ejemplo, para escribir una función de pérdida personalizada o modificar el algoritmo de entrenamiento. Para tales casos, necesitará utilizar la API de nivel inferior de TensorFlow, como verá en el siguiente capítulo.

## Ejercicios

1. ¿Cuál es el problema al que apuntan la inicialización de Glorot y la inicialización de He? ¿Arreglar?
2. ¿Está bien inicializar todos los pesos con el mismo valor siempre que ese valor se seleccione aleatoriamente mediante la inicialización He?

3. ¿Está bien inicializar los términos de sesgo en 0?
4. ¿En qué casos le gustaría utilizar cada una de las funciones de activación que analizamos en este capítulo?
5. ¿Qué puede suceder si establece el hiperparámetro de impulso demasiado cerca de 1 (por ejemplo, 0,99999) cuando utiliza un optimizador SGD?
6. Nombra tres formas en las que puedes producir un modelo disperso.
7. ¿El abandono frena el entrenamiento? ¿Ralentiza la inferencia (es decir, hacer predicciones sobre nuevos casos)? ¿Qué pasa con el abandono de MC?
8. Practique el entrenamiento de una red neuronal profunda en la imagen CIFAR10.

conjunto de datos:

- a. Construya un DNN con 20 capas ocultas de 100 neuronas cada una (son demasiadas, pero es el objetivo de este ejercicio). Utilice la inicialización He y la función de activación Swish.
- b. Utilice la optimización de Nadam y la parada anticipada para entrenar la red en el conjunto de datos CIFAR10. Puedes cargarlo con  
`tf.keras.datasets.cifar10.load_data()`. El conjunto de datos se compone de 60 000 imágenes en color de  $32 \times 32$  píxeles (50 000 para entrenamiento, 10 000 para pruebas) con 10 clases, por lo que necesitará una capa de salida softmax con 10 neuronas. Recuerde buscar la tasa de aprendizaje adecuada cada vez que cambie la arquitectura o los hiperparámetros del modelo.
- C. Ahora intente agregar normalización por lotes y compare las curvas de aprendizaje: ¿está convergiendo más rápido que antes? ¿Produce un modelo mejor? ¿Cómo afecta la velocidad de entrenamiento?
- d. Intente reemplazar la normalización por lotes con SELU y haga el ajustes necesarios para garantizar que la red se normalice automáticamente (es decir, estandarizar las características de entrada, usar la inicialización normal de LeCun, asegurarse de que el DNN contenga solo una secuencia de capas densas, etc.).
- mi. Intente regularizar el modelo con abandono alfa. Luego, sin volver a entrenar su modelo, vea si puede lograr una mayor precisión utilizando la deserción de MC.

F. Vuelva a entrenar su modelo usando la programación de 1 ciclo y vea si mejora  
Velocidad de entrenamiento y precisión del modelo.

Las soluciones a estos ejercicios están disponibles al final del cuaderno de este capítulo, en <https://homl.info/colab3>.

---

<sup>1</sup> Xavier Glorot y Yoshua Bengio, “Understanding the Difficulty of Training Deep Feedforward Neural Networks”, Actas de la 13<sup>a</sup> Conferencia Internacional sobre Inteligencia Artificial y Estadística (2010): 249–256.

<sup>2</sup> He aquí una analogía: si ajustas el mando de un amplificador de micrófono demasiado cerca de cero, la gente no escuchará tu voz, pero si lo configuras demasiado cerca del máximo, tu voz se saturará y la gente no entenderá lo que estás diciendo. Ahora imagine una cadena de amplificadores de este tipo: es necesario configurarlos correctamente para que su voz suene alta y clara al final de la cadena. Tu voz tiene que salir de cada amplificador con la misma amplitud con la que entró.

<sup>3</sup> Por ejemplo, Kaiming He et al., “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”, Actas de la Conferencia Internacional IEEE 2015 sobre Visión por Computadora (2015): 1026–1034.

<sup>4</sup> Una neurona muerta puede volver a la vida si sus entradas evolucionan con el tiempo y finalmente regresan dentro de un rango en el que la función de activación ReLU vuelve a recibir una entrada positiva. Por ejemplo, esto puede suceder si el descenso de gradiente modifica las neuronas en las capas debajo de la neurona muerta.

<sup>5</sup> Bing Xu et al., “Evaluación empírica de activaciones rectificadas en redes convolucionales”, preimpresión de arXiv arXiv:1505.00853 (2015).

<sup>6</sup> Djork-Arné Clevert et al., “Aprendizaje en red profundo rápido y preciso mediante Unidades lineales exponenciales (ELU)”, Actas de la Conferencia Internacional sobre Representaciones del Aprendizaje, preimpresión de arXiv (2015).

<sup>7</sup> Günter Klambauer et al., “Self-Normalizing Neural Networks”, Actas de la 31.<sup>a</sup> Conferencia Internacional sobre Sistemas de Procesamiento de Información Neural (2017): 972–981.

<sup>8</sup> Dan Hendrycks y Kevin Gimpel, “Unidades lineales de error gaussiano (GELU)”, arXiv preimpresión arXiv:1606.08415 (2016).

<sup>9</sup> Una función es convexa si el segmento de recta entre dos puntos cualesquiera de la curva nunca se encuentra debajo de la curva. Una función monótona sólo aumenta o sólo disminuye.

<sup>10</sup> Prajit Ramachandran et al., “Searching for Activation Functions”, preimpresión de arXiv arXiv:1710.05941 (2017).

<sup>11</sup> Diganta Misra, “Mish: una función de activación no monotónica autoregulada”, preimpresión de arXiv arXiv:1908.08681 (2019).

- <sup>12</sup> Sergey Ioffe y Christian Szegedy, "Normalización por lotes: aceleración profunda Network Training by Reduction Internal Covariate Shift", Actas de la 32<sup>a</sup> Conferencia Internacional sobre Aprendizaje Automático (2015): 448–456.
- <sup>13</sup> Sin embargo, se estiman durante el entrenamiento en función de los datos de entrenamiento, por lo que podría decirse que son entrenables . En Keras, "no entrenable" en realidad significa "no afectado por la retropropagación".
- <sup>14</sup> Razvan Pascanu et al., "Sobre la dificultad de entrenar redes neuronales recurrentes", Actas de la 30<sup>a</sup> Conferencia Internacional sobre Aprendizaje Automático (2013): 1310–1318.
- <sup>15</sup> Boris T. Polyak, "Algunos métodos para acelerar la convergencia de la iteración Métodos", Matemática Computacional y Física Matemática de la URSS 4, no. 5 (1964): 1–17.
- <sup>16</sup> Yurii Nesterov, "Un método para el problema de minimización convexa sin restricciones con la tasa de convergencia  $O(1/k^2)$ ", Doklady AN URSS 269 (1983): 543–547.
- <sup>17</sup> John Duchi et al., "Métodos adaptativos de subgradiente para el aprendizaje en línea y Optimización estocástica", Journal of Machine Learning Research 12 (2011): 2121–2159.
- <sup>18</sup> Este algoritmo fue creado por Geoffrey Hinton y Tijmen Tieleman en 2012 y presentado por Geoffrey Hinton en su clase de Coursera sobre redes neuronales (diapositivas: <https://homl.info/57>; vídeo: <https://homl.info/58>). Curiosamente, dado que los autores no escribieron un artículo para describir el algoritmo, los investigadores suelen citar la "diapositiva 29 de la conferencia 6e" en sus artículos.
- <sup>19</sup>  $\rho$  es la letra griega rho.
- <sup>20</sup> Diederik P. Kingma y Jimmy Ba, "Adam: A Method for Stochastic Optimization", preimpresión de arXiv arXiv:1412.6980 (2014).
- <sup>21</sup> Timothy Dozat, "Incorporando el impulso de Nesterov a Adán" (2016).
- <sup>22</sup> Ilya Loshchilov y Frank Hutter, "Regularización de la caída de peso desacoplada", arXiv preimpresión arXiv:1711.05101 (2017).
- <sup>23</sup> Ashia C. Wilson et al., "El valor marginal de los métodos de gradiente adaptativo en Aprendizaje automático", Avances en sistemas de procesamiento de información neuronal 30 (2017): 4148–4158.
- <sup>24</sup> Leslie N. Smith, "Un enfoque disciplinado de los hiperparámetros de las redes neuronales: Parte 1: tasa de aprendizaje, tamaño de lote, impulso y caída de peso", preimpresión de arXiv arXiv:1803.09820 (2018).
- <sup>25</sup> Andrew Senior et al., "Un estudio empírico de tasas de aprendizaje en redes neuronales profundas para el reconocimiento de voz", Actas de la Conferencia Internacional IEEE sobre Acústica, Habla y Procesamiento de Señales (2013): 6724–6728.
- <sup>26</sup> Geoffrey E. Hinton et al., "Mejora de las redes neuronales mediante la prevención de la coadaptación de detectores de características", preimpresión de arXiv arXiv:1207.0580 (2012).

**27** Nitish Srivastava et al., "Abandono: una forma sencilla de evitar el sobreajuste de las redes neuronales", *Journal of Machine Learning Research* 15 (2014): 1929–1958.

**28** Yarin Gal y Zoubin Ghahramani, "La deserción como aproximación bayesiana: representación de la incertidumbre del modelo en el aprendizaje profundo", *Actas de la 33<sup>a</sup> Conferencia Internacional sobre Aprendizaje Automático* (2016): 1050–1059.

**29** Específicamente, muestran que entrenar una red de deserción es matemáticamente equivalente a aproximar la inferencia bayesiana en un tipo específico de modelo probabilístico llamado proceso gaussiano profundo.

**30** Esta clase MCDropout funcionará con todas las API de Keras, incluida la API secuencial. Si solo le importa la API funcional o la API de subclases, no es necesario que cree una clase MCDropout; puedes crear una capa de abandono normal y llamarla con `entrenamiento=True`.

# Capítulo 12. Modelos personalizados y entrenamiento con TensorFlow

---

Hasta ahora, hemos usado solo la API de alto nivel de TensorFlow, Keras, pero ya nos llevó bastante lejos: construimos varias arquitecturas de redes neuronales, incluidas redes de regresión y clasificación, redes anchas y profundas y redes autonormalizadas, usando todo tipo de técnicas, como normalización por lotes, abandono y programas de tasa de aprendizaje. De hecho, el 95% de los casos de uso que encontrará no requerirán nada más que Keras (y `tf.data`; consulte [el Capítulo 13](#)). Pero ahora es el momento de profundizar en TensorFlow y echar un vistazo a su [API Python de nivel inferior](#). Esto será útil cuando necesite control adicional para escribir funciones de pérdida personalizadas, métricas personalizadas, capas, modelos, inicializadores, regularizadores, restricciones de peso y más. Es posible que incluso necesites controlar completamente el ciclo de entrenamiento; por ejemplo, para aplicar transformaciones o restricciones especiales a los gradientes (más allá de simplemente recortarlos) o utilizar múltiples optimizadores para diferentes partes de la red. Cubriremos todos estos casos en este capítulo y también veremos cómo puede mejorar sus modelos personalizados y algoritmos de entrenamiento utilizando la función de generación automática de gráficos de TensorFlow. Pero primero, hagamos un recorrido rápido por TensorFlow.

## Un recorrido rápido por TensorFlow

Como sabe, TensorFlow es una poderosa biblioteca para cálculo numérico, particularmente adecuada y optimizada para el aprendizaje automático a gran escala (pero puede usarla para cualquier otra cosa que requiera cálculos pesados). Fue desarrollado por el equipo de Google Brain e impulsa muchos de los servicios a gran escala de Google, como Google Cloud Speech, Google Photos y Google Search. Fue de código abierto en noviembre de 2015 y ahora es el más utilizado.

Biblioteca de aprendizaje profundo en la industria:<sup>1</sup> innumerables proyectos utilizan TensorFlow para todo tipo de tareas de aprendizaje automático, como clasificación de imágenes, procesamiento de lenguaje natural, sistemas de recomendación y pronóstico de series de tiempo.

Entonces, ¿qué ofrece TensorFlow? Aquí hay un resumen:

- Su núcleo es muy similar a NumPy, pero con soporte para GPU.
- Admite informática distribuida (en múltiples dispositivos y servidores).
- Incluye una especie de compilador justo a tiempo (JIT) que le permite optimizar los cálculos en cuanto a velocidad y uso de memoria. Funciona extrayendo el gráfico de cálculo de una función de Python, optimizándolo (por ejemplo, podando nodos no utilizados) y ejecutándolo de manera eficiente (por ejemplo, ejecutando automáticamente operaciones independientes en paralelo).
- Los gráficos de cálculo se pueden exportar a un formato portátil, por lo que puede entrenar un modelo de TensorFlow en un entorno (por ejemplo, usando Python en Linux) y ejecutarlo en otro (por ejemplo, usando Java en un dispositivo Android).
- Implementa autodiff en modo inverso (consulte [el Capítulo 10](#) y [el Apéndice B](#)) y proporciona algunos optimizadores excelentes, como RMSProp y Nadam (consulte [el Capítulo 11](#)), para que pueda minimizar fácilmente todo tipo de funciones de pérdida.

TensorFlow ofrece muchas más funciones basadas en estas funciones principales: la más importante es, por supuesto, Keras, pero también tiene operaciones de preprocesamiento y carga de datos (tf.data, tf.io, etc.), operaciones de procesamiento de imágenes (tf.image ), operaciones de procesamiento de señales (tf.signal) y más (consulte [la Figura 12-1](#) para obtener una descripción general de la API de Python de TensorFlow).

## CONSEJO

Cubriremos muchos de los paquetes y funciones de la API de TensorFlow, pero es imposible cubrirlos todos, por lo que deberías tomarte un tiempo para explorar la API; Verás que es bastante rico y está bien documentado.

En el nivel más bajo, cada operación de TensorFlow (op para abreviar) se implementa utilizando código C++ altamente eficiente.<sup>3</sup> Muchas operaciones tienen múltiples implementaciones llamadas kernels: cada kernel está dedicado a un tipo de dispositivo específico, como CPU, GPU o incluso TPU (unidades de procesamiento tensorial). Como sabrá, las GPU pueden acelerar drásticamente los cálculos dividiéndolos en muchos fragmentos más pequeños y ejecutándolos en paralelo en muchos subprocessos de GPU. Los TPU son aún más rápidos: son chips ASIC personalizados creados específicamente<sup>4</sup> para operaciones de aprendizaje profundo (analizaremos cómo usar TensorFlow con GPU o TPU en el [Capítulo 19](#)).

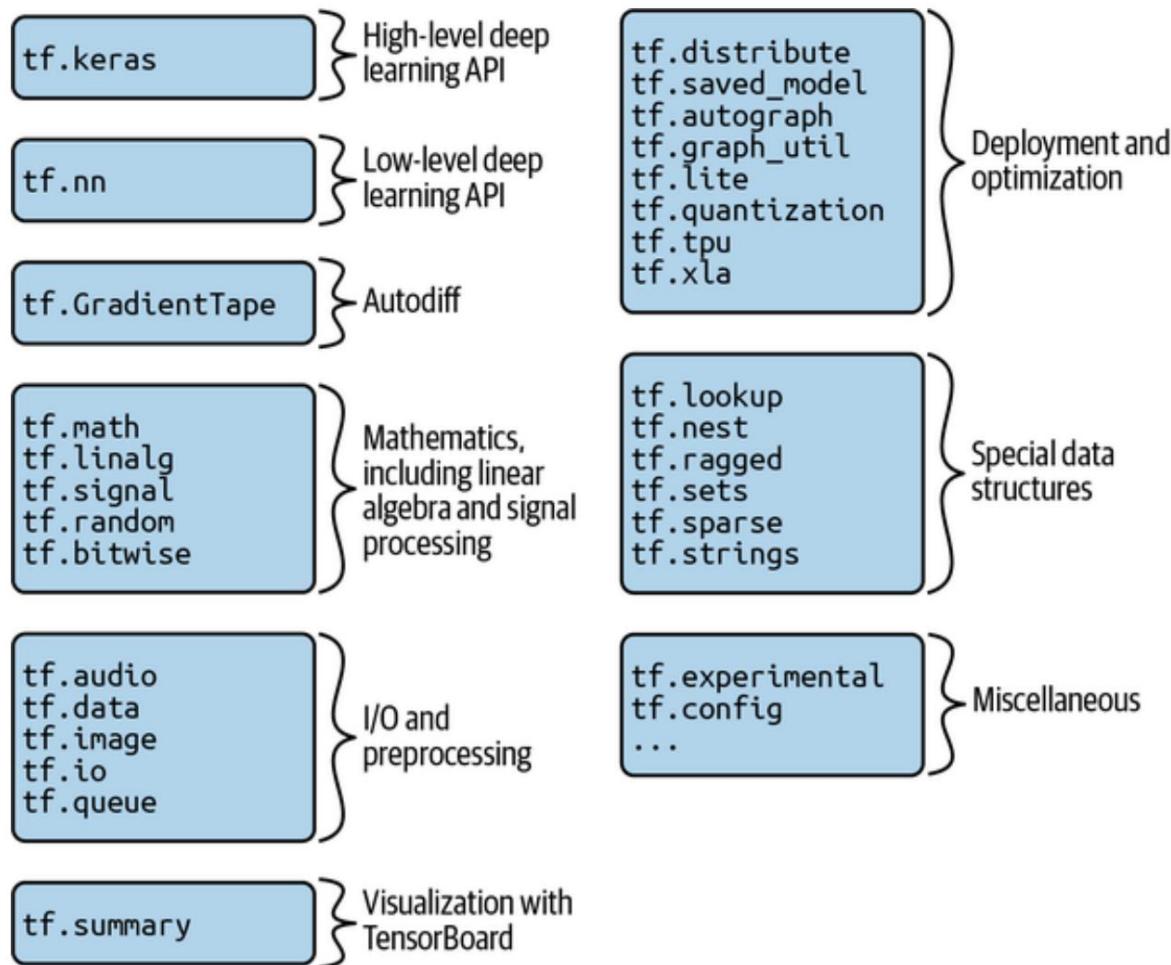


Figura 12-1. API de Python de TensorFlow

La arquitectura de TensorFlow se muestra en la Figura 12-2. La mayoría de las veces su código utilizará las API de alto nivel (especialmente Keras y tf.data), pero cuando necesite más flexibilidad utilizará la API de Python de nivel inferior, manejando tensores directamente. En cualquier caso, el motor de ejecución de TensorFlow se encargará de ejecutar las operaciones de manera eficiente, incluso en múltiples dispositivos y máquinas si así lo indica.

TensorFlow se ejecuta no solo en Windows, Linux y macOS, sino también en dispositivos móviles (usando TensorFlow Lite), incluidos iOS y Android (consulte el Capítulo 19). Tenga en cuenta que las API para otros lenguajes también están disponibles, si no desea utilizar la API de Python: hay API de C++, Java y Swift. Incluso existe una implementación de JavaScript llamada TensorFlow.js que permite ejecutar sus modelos directamente en su navegador.

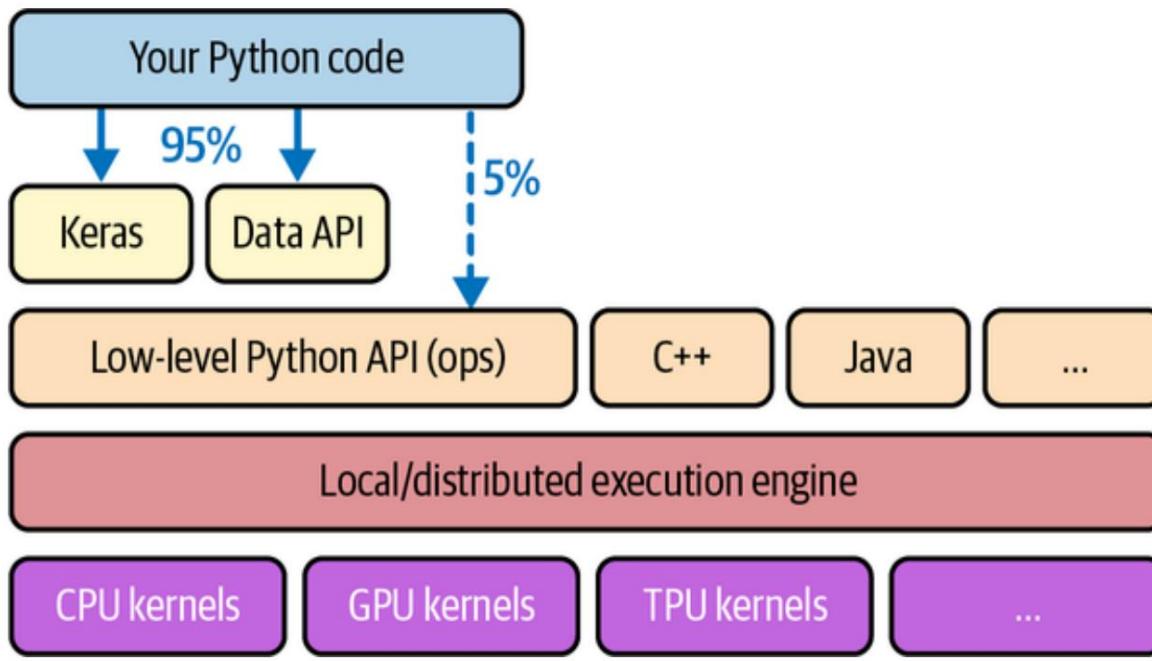


Figura 12-2. Arquitectura de TensorFlow

TensorFlow es más que una biblioteca. TensorFlow está en el centro de un extenso ecosistema de bibliotecas. Primero, está TensorBoard para visualización (consulte [el Capítulo 10](#)). A continuación, está [TensorFlow Extended \(TFX\)](#), que es un conjunto de bibliotecas creadas por Google para producir proyectos de TensorFlow: incluye herramientas para validación de datos, preprocesamiento, análisis de modelos y servicio (con TF Serving; consulte el [Capítulo 19](#)). TensorFlow Hub de Google proporciona una manera de descargar y reutilizar fácilmente redes neuronales previamente entrenadas. También puede obtener muchas arquitecturas de redes neuronales, algunas de ellas previamente entrenadas, en el [jardín modelo de TensorFlow](#). Consulte los [recursos de TensorFlow](#) y <https://github.com/jtoy/awesome-tensorflow> para más proyectos basados en TensorFlow. Encontrarás cientos de proyectos de TensorFlow en GitHub, por lo que suele ser fácil encontrar el código existente para cualquier cosa que estés intentando hacer.

## CONSEJO

Cada vez se publican más artículos sobre ML junto con sus implementaciones y, a veces, incluso con modelos previamente entrenados. Consulte <https://paperswithcode.com> para encontrarlos fácilmente.

Por último, pero no menos importante, TensorFlow cuenta con un equipo dedicado de desarrolladores apasionados y serviciales, así como una gran comunidad que contribuye a mejorarlo. Para hacer preguntas técnicas, debe utilizar <https://stackoverflow.com> y etiquete su pregunta con tensorflow y python. Puede presentar errores y solicitudes de funciones a través de [GitHub](#). Para debates generales, únase al [foro de TensorFlow](#).

Bien, ¡es hora de empezar a codificar!

## Usando TensorFlow como NumPy

La API de TensorFlow gira en torno a tensores, que fluyen de una operación a otra, de ahí el nombre TensorFlow. Un tensor es muy similar a un ndarray NumPy: suele ser una matriz multidimensional, pero también puede contener un escalar (un valor simple, como 42).

Estos tensores serán importantes cuando creemos funciones de costos personalizadas, métricas personalizadas, capas personalizadas y más, así que veamos cómo crearlos y manipularlos.

## Tensores y operaciones

Puedes crear un tensor con `tf.constant()`. Por ejemplo, aquí hay un tensor que representa una matriz con dos filas y tres columnas de flotantes:

```
>>> importar tensorflow como tf >>> t =  
tf.constant([[1., 2., 3.], [4., 5., 6.]]) # matriz  
>>> t  
<tf.Tensor: forma=(2, 3), dtype=float32, numpy=
```

```
matriz([[1., 2., 3.], [4., 5., 6.]],
      dtype=float32)>
```

Al igual que un ndarray, un tf.Tensor tiene una forma y un tipo de datos (dtype):

```
>>> t.shape
TensorShape([2, 3]) >>>
t.dtype tf.float32
```

La indexación funciona de manera muy similar a NumPy:

```
>>> t[:, 1:]
<tf.Tensor: forma=(2, 2), dtype=float32, numpy= array([[2., 3.], [5., 6.]],

dtype =floatante32)>

>>> t[..., 1, tf.newaxis] <tf.Tensor:
forma=(2, 1), dtype=float32, numpy= matriz([[2.,

[5.]], dtype=float32)>
```

Lo más importante es que se encuentran disponibles todo tipo de operaciones tensoriales:

```
>>> t + 10
<tf.Tensor: forma=(2, 3), dtype=float32, numpy= matriz([[11., 12., 13.],

[14., 15., 16.]], dtype=float32)> >>> tf.square(t)
<tf.Tensor: forma=(2,
3), dtype=float32, numpy= array([[ 1., 4., 9.,

[16., 25., 36.]], dtype=float32)>
>>> t @ tf.transpose(t) <tf.Tensor:
forma=(2, 2), dtype=float32, numpy= matriz([[14., 32.],

[32., 77.]], dtype=float32)>
```

Tenga en cuenta que escribir `t + 10` equivale a llamar a `tf.add(t, 10)` (de hecho, Python llama al método mágico `t.__add__(10)`, que

simplemente llama a `tf.add(t, 10)`). También se admiten otros operadores, como `-` y `*`. El operador `@` se agregó en Python 3.5, para la multiplicación de matrices: equivale a llamar a la función `tf.matmul()`.

### NOTA

Muchas funciones y clases tienen alias. Por ejemplo, `tf.add()` y `tf.math.add()` son la misma función. Esto permite que TensorFlow tenga nombres concisos para las operaciones más comunes y al mismo tiempo preserva<sup>5</sup> los paquetes bien organizados.

Un tensor también puede contener un valor escalar. En este caso, la forma está vacía:

```
>>> tf.constant(42)
<tf.Tensor: forma=(), dtype=int32, numpy=42>
```

### NOTA

La API de Keras tiene su propia API de bajo nivel, ubicada en `tf.keras.backend`. Este paquete normalmente se importa como `K`, por razones de concisión. Solía incluir funciones como `K.square()`, `K.exp()` y `K.sqrt()`, que puede encontrar en el código existente: esto era útil para escribir código portátil cuando Keras admitía múltiples backends, pero ahora Para saber que Keras es solo para TensorFlow, debes llamar directamente a la API de bajo nivel de TensorFlow (por ejemplo, `tf.square()` en lugar de `K.square()`). Técnicamente, `K.square()` y sus amigos todavía están ahí por compatibilidad con versiones anteriores, pero la documentación del paquete `tf.keras.backend` solo enumera un puñado de funciones de utilidad, como `clear_session()` (mencionada en el Capítulo 10).

Encontrará todas las operaciones matemáticas básicas que necesita (`tf.add()`, `tf.multiply()`, `tf.square()`, `tf.exp()`, `tf.sqrt()`, etc.) y la mayoría de las operaciones que pueda buscar en NumPy (por ejemplo,

`tf.reshape()`, `tf.squeeze()`, `tf.tile()`). Algunas funciones tienen un nombre diferente al de NumPy; por ejemplo, `tf.reduce_mean()`, `tf.reduce_sum()`, `tf.reduce_max()` y `tf.math.log()` son el equivalente de `np.mean()`, `np.sum()`, `np.max()` y `np.log()`. Cuando el nombre difiere, suele haber una buena razón para ello. Por ejemplo, en TensorFlow debes escribir `tf.transpose(t)`; no puedes simplemente escribir `tT` como en NumPy. La razón es que la función `tf.transpose()` no hace exactamente lo mismo que el atributo `T` de NumPy: en TensorFlow, se crea un nuevo tensor con su propia copia de los datos transpuestos, mientras que en NumPy, `tT` es solo una vista transpuesta sobre los mismos datos. De manera similar, la operación `tf.reduce_sum()` se denomina de esta manera porque su núcleo de GPU (es decir, la implementación de GPU) utiliza un algoritmo de reducción que no garantiza el orden en el que se agregan los elementos: debido a que los flotantes de 32 bits tienen una precisión limitada, el resultado puede cambiar ligeramente cada vez que llame a esta operación. Lo mismo ocurre con `tf.reduce_mean()` (pero, por supuesto, `tf.reduce_max()` es determinista).

## Tensores y NumPy

Los tensores funcionan bien con NumPy: puedes crear un tensor a partir de una matriz NumPy y viceversa. Incluso puedes aplicar operaciones de TensorFlow a matrices NumPy y operaciones NumPy a tensores:

```
>>> importar numpy como np
>>> a = np.array([2., 4., 5.]) >>> tf.constant(a)
<tf.Tensor: id=111,
forma=(3, ), dtype=float64, numpy=array([2., 4., 5.])> >>> t.numpy()
# o np.array(t) array([[1., 2., 3.], [4., 5.,
6.]], dtype=float32) >>> tf.square(a) <tf.Tensor:
id=116, shape=(3,),
dtype=float64, numpy=array ([4., 16., 25.])>
```

```
>>> np.square(t)
matriz([[ 1., 4., 9.], [16., 25., 36.]],
       dtype=float32)
```

### ADVERTENCIA

Tenga en cuenta que NumPy usa una precisión de 64 bits de forma predeterminada, mientras que TensorFlow usa 32 bits. Esto se debe a que la precisión de 32 bits es generalmente más que suficiente para las redes neuronales, además se ejecuta más rápido y utiliza menos RAM. Entonces, cuando crea un tensor a partir de una matriz NumPy, asegúrese de configurar `dtype=tf.float32`.

## Conversiones de tipos

Las conversiones de tipos pueden perjudicar significativamente el rendimiento y pueden pasar desapercibidas fácilmente cuando se realizan automáticamente. Para evitar esto, TensorFlow no realiza ninguna conversión de tipos automáticamente: simplemente genera una excepción si intenta ejecutar una operación en tensores con tipos incompatibles. Por ejemplo, no puedes agregar un tensor flotante y un tensor entero, y ni siquiera puedes agregar un flotante de 32 bits y un flotante de 64 bits:

```
>>> tf.constant(2.) + tf.constant(40)
[...] InvalidArgumentError: [...] se espera que sea un tensor flotante [...]
>>> tf.constant(2.) +
tf.constant(40.,
           dtype=tf.float64)
[...] InvalidArgumentError: [...] se espera que sea un tensor flotante [...]
```

Esto puede resultar un poco molesto al principio, ¡pero recuerda que es por una buena causa! Y, por supuesto, puedes usar `tf.cast()` cuando realmente necesites convertir tipos:

```
>>> t2 = tf.constant(40., dtype=tf.float64) >>> tf.constant(2.0) + tf.cast(t2,
tf.float32) <tf.Tensor: id=136, forma=( ), dtype=float32, numpy=42.0>
```

## variables

Los valores de `tf.Tensor` que hemos visto hasta ahora son inmutables: no podemos modificarlos. Esto significa que no podemos usar tensores regulares para implementar pesos en una red neuronal, ya que es necesario modificarlos mediante retropropagación. Además, es posible que también sea necesario cambiar otros parámetros con el tiempo (por ejemplo, un optimizador de impulso realiza un seguimiento de los gradientes pasados). Lo que necesitamos es una `tf.Variable`:

```
>>> v = tf.Variable([[1., 2., 3.], [4., 5., 6.]])
>>> v
<tf.Variable 'Variable:0' forma=(2, 3) dtype=float32,
numpy=
 matriz([[1., 2., 3.], [4., 5., 6.]],
 dtype=float32)>
```

Un `tf.Variable` actúa de manera muy similar a un `tf.Tensor`: puedes realizar las mismas operaciones con él, también funciona muy bien con NumPy y es igual de exigente con los tipos. Pero también se puede modificar en el lugar usando el método `asignar()` (o `asignar_add()` o `asignar_sub()`, que incrementan o disminuyen la variable en el valor dado). También puede modificar celdas (o sectores) individuales, utilizando el método `asignar()` de la celda (o sector) o utilizando los métodos `scatter_update()` o `scatter_nd_update()`:

```
v.assign(2 * v) [8., 10.,
12.]] v[0, 1].assign(42)
[8., 10., 12.]] v[:, 2].assign( [0.,
1.]) # v ahora es igual
a [[2., 42., 0.], [8., 10., 1.]] v.scatter_nd_update( [8., 10., 200.]] índices=[[0, 0], [1, 2]],
actualizaciones=[100.,
200.]) # v ahora es igual a [[100., 42., 0.],
```

La asignación directa no funcionará:

```
>>> v[1] = [7., 8., 9.]
```

```
[...] TypeError: el objeto 'ResourceVariable' no admite la asignación de elementos
```

## NOTA

En la práctica, rara vez tendrás que crear variables manualmente; Keras proporciona un método `add_weight()` que se encargará de ello por usted, como verá. Además, los optimizadores generalmente actualizarán los parámetros del modelo directamente, por lo que rara vez necesitará actualizar las variables manualmente.

## Otras estructuras de datos

TensorFlow admite varias otras estructuras de datos, incluidas las siguientes (consulte la sección "Otras estructuras de datos" en el cuaderno de este capítulo o el [Apéndice C](#) para obtener más detalles):

### Tensores dispersos (`tf.SparseTensor`)

Representa eficientemente tensores que contienen principalmente ceros.

El paquete `tf.sparse` contiene operaciones para tensores dispersos.

### Matrices tensoriales (`tf.TensorArray`)

Son listas de tensores. Tienen una longitud fija de forma predeterminada, pero opcionalmente pueden hacerse extensibles. Todos los tensores que contienen deben tener la misma forma y tipo de datos.

### Tensores irregulares (`tf.RaggedTensor`)

Representa listas de tensores, todos del mismo rango y tipo de datos, pero con diferentes tamaños. Las dimensiones a lo largo de las cuales varían los tamaños de los tensores se denominan dimensiones irregulares. El paquete `tf.ragged` contiene operaciones para tensores irregulares.

### Tensores de cuerdas

Son tensores regulares de tipo `tf.string`. Estos representan cadenas de bytes, no cadenas Unicode, por lo que si crea un tensor de cadena usando una cadena Unicode (por ejemplo, una cadena Python 3 normal como "café"), se codificará en UTF-8 automáticamente (por ejemplo, `b"caf \xc3\xaa9"`). Alternativamente, puede representar cadenas Unicode usando tensores de tipo `tf.int32`, donde cada elemento representa un punto de código Unicode (por ejemplo, [99, 97, 102, 233]). El paquete `tf.strings` (con una s) contiene operaciones para cadenas de bytes y cadenas Unicode (y para convertir una en otra). Es importante tener en cuenta que `tf.string` es atómico, lo que significa que su longitud no aparece en la forma del tensor. Una vez que lo convierte a un tensor Unicode (es decir, un tensor de tipo `tf.int32` que contiene puntos de código Unicode), la longitud aparece en la forma.

#### Conjuntos

Se representan como tensores regulares (o tensores dispersos). Por ejemplo, `tf.constant([[1, 2], [3, 4]])` representa los dos conjuntos  $\{1, 2\}$  y  $\{3, 4\}$ . De manera más general, cada conjunto está representado por un vector en el último eje del tensor. Puede manipular conjuntos utilizando operaciones del paquete `tf.sets`.

#### Colas

Almacene tensores en varios pasos. TensorFlow ofrece varios tipos de colas: colas básicas de primero en entrar, primero en salir (FIFO) (`FIFOQueue`), además de colas que pueden priorizar algunos elementos (`PriorityQueue`), mezclar sus elementos (`RandomShuffleQueue`) y agrupar elementos de diferentes formas mediante relleno (`RellenoFIFOQueue`). Todas estas clases están en el paquete `tf.queue`.

Con tensores, operaciones, variables y diversas estructuras de datos a su disposición, ¡ahora está listo para personalizar sus modelos y algoritmos de entrenamiento!

Personalización de modelos y algoritmos de entrenamiento Comenzará creando una función de pérdida personalizada, que es un caso de uso sencillo y común.

## Funciones de pérdida personalizadas

Suponga que desea entrenar un modelo de regresión, pero su conjunto de entrenamiento es un poco ruidoso. Por supuesto, se empieza intentando limpiar el conjunto de datos eliminando o corrigiendo los valores atípicos, pero eso resulta insuficiente; el conjunto de datos sigue siendo ruidoso. ¿Qué función de pérdida debería utilizar? El error cuadrático medio puede penalizar demasiado los errores grandes y hacer que su modelo sea impreciso. El error absoluto medio no penalizaría tanto a los valores atípicos, pero el entrenamiento podría tardar un poco en converger y el modelo entrenado podría no ser muy preciso. Probablemente este sea un buen momento para utilizar la pérdida de Huber (introducida en [el capítulo 10](#)) en lugar del viejo MSE. La pérdida de Huber está disponible en Keras (solo use una instancia de la clase `tf.keras.losses.Huber`), pero supongamos que no está allí. Para implementarlo, simplemente cree una función que tome las etiquetas y las predicciones del modelo como argumentos, y use operaciones de TensorFlow para calcular un tensor que contenga todas las pérdidas (una por muestra):

```
def huber_fn(y_true, y_pred): error =  
    y_true - y_pred is_small_error  
    = tf.abs(error) < 1 squared_loss =  
    tf.square(error) / 2 linear_loss = tf.abs(error) -  
    0.5 return tf.where(is_small_error,  
    pérdida_cuadrada, pérdida_lineal)
```

### ADVERTENCIA

Para un mejor rendimiento, debes utilizar una implementación vectorizada, como en este ejemplo. Además, si desea beneficiarse de las funciones de optimización de gráficos de TensorFlow, debe utilizar solo operaciones de TensorFlow.

También es posible devolver la pérdida media en lugar de las pérdidas de muestra individuales, pero esto no se recomienda ya que hace imposible utilizar pesos de clase o pesos de muestra cuando los necesita (consulte el Capítulo 10).

Ahora puedes usar esta función de pérdida de Huber cuando compilas el Modelo Keras, luego entrena tu modelo como de costumbre:

```
model.compile(loss=huber_fn, optimizador="nadam") model.fit(X_train,  
y_train, [...])
```

¡Y eso es! Para cada lote durante el entrenamiento, Keras llamará a la función huber\_fn() para calcular la pérdida, luego utilizará autodiff en modo inverso para calcular los gradientes de la pérdida con respecto a todos los parámetros del modelo y, finalmente, realizará un descenso de gradiente. paso (en este ejemplo usando un optimizador Nadam). Además, realizará un seguimiento de la pérdida total desde el comienzo de la época y mostrará la pérdida media.

Pero, ¿qué sucede con esta pérdida personalizada cuando guardas el modelo?

## Guardar y cargar modelos que contienen elementos personalizados Componentes

Guardar un modelo que contiene una función de pérdida personalizada funciona bien, pero cuando lo cargue, deberá proporcionar un diccionario que asigne el nombre de la función a la función real. De manera más general, cuando cargas

Para un modelo que contiene objetos personalizados, es necesario asignar los nombres a los objetos:

```
modelo =
tf.keras.models.load_model("mi_modelo_con_una_pérdida_personalizada",
                             objetos_personalizados=
{"huber_fn": huber_fn})
```

## CONSEJO

Si decoras la función `huber_fn()` con `@keras.utils.Register_keras_serializable()`, estará automáticamente disponible para la función `load_model()`: no es necesario incluirlo en el diccionario `custom_objects`.

Con la implementación actual, cualquier error entre  $-1$  y  $1$  se considera “pequeño”. Pero ¿qué pasa si quieres un umbral diferente? Una solución es crear una función que cree una función de pérdida configurada:

```
def create_huber(umbral=1.0):
    def huber_fn(y_true,
                 y_pred):
        error = y_true - y_pred
        is_small_error = tf.abs(error) < umbral
        squared_loss = tf.square(error)
        linear_loss = umbral * tf.abs(error) - umbral
        return tf.where(is_small_error, squared_loss, linear_loss)

    return huber_fn
```

```
model.compile(loss=create_huber(2.0), optimizador="nadam")
```

Lamentablemente, cuando guarde el modelo, el umbral no se guardará. Esto significa que tendrá que especificar el valor umbral al cargar el modelo (tenga en cuenta que el nombre a utilizar es

"huber\_fn", que es el nombre de la función que le diste a Keras, no el nombre de la función que la creó):

```
modelo =
    tf.keras.models.load_model( "my_model_with_a_custom_loss_threshold_2",
    custom_objects={"huber_fn": create_huber(2.0)}
)
```

Puedes resolver esto creando una subclase de la clase `tf.keras.losses.Loss` y luego implementando su método `get_config()`:

```
clase HuberLoss(tf.keras.losses.Loss):
    def __init__(self, umbral=1.0, **kwargs): self.threshold = umbral

        super().__init__(**kwargs)

    def call(self, y_true, y_pred): error = y_true - y_pred
        is_small_error = tf.abs(error) <
            self.threshold squared_loss = tf.square(error) / 2 linear_loss = self.threshold *
            tf.abs(error) -

        umbral.auto**2 / 2
            devolver tf.where(is_small_error, squared_loss, linear_loss)

    def get_config(self): base_config
        = super().get_config() return {"base_config, "umbral":
            self.threshold}
```

Repasemos este código:

- El constructor acepta `**kwargs` y los pasa al constructor principal, que maneja los hiperparámetros estándar: el nombre de la pérdida y el algoritmo de reducción que se utilizará para agregar las pérdidas de instancias individuales. Por defecto es "AUTO", que equivale a "SUM\_OVER\_BATCH\_SIZE": la pérdida será la suma de las pérdidas de las instancias, ponderadas por el

pesos de la muestra, si los hay, y divididos por el tamaño del lote (no por la suma de los pesos, por lo que esta no es la media ponderada). Otros valores<sup>6</sup> posibles son "SUM" y "NONE".

- El método call() toma las etiquetas y las predicciones, calcula todas las pérdidas de instancia y las devuelve.
- El método get\_config() devuelve un diccionario que asigna cada nombre de hiperparámetro a su valor. Primero llama al método get\_config() de la clase principal y luego agrega los nuevos hiperparámetros a este diccionario.

<sup>7</sup>

Luego puedes usar cualquier instancia de esta clase cuando compilas el modelo:

```
model.compile(pérdida=HuberLoss(2.), optimizador="nadam")
```

Cuando guarde el modelo, el umbral se guardará junto con él; y cuando cargas el modelo, solo necesitas asignar el nombre de la clase a la clase misma:

```
modelo =
tf.keras.models.load_model("my_model_with_a_custom_loss_cla ss",
                             objetos_personalizados=
{"HuberLoss": HuberLoss})
```

Cuando guarda un modelo, Keras llama al método get\_config() de la instancia de pérdida y guarda la configuración en el formato SavedModel. Cuando carga el modelo, llama al método de clase from\_config() en la clase HuberLoss: este método es implementado por la clase base (Loss) y crea una instancia de la clase, pasando \*\*config al constructor.

¡Eso es todo por las pérdidas! Como verá ahora, las funciones de activación personalizadas, los inicializadores, los regularizadores y las restricciones no son muy diferentes.

## Funciones de activación personalizadas, inicializadores, Regularizadores y restricciones

La mayoría de las funcionalidades de Keras, como pérdidas, regularizadores, restricciones, inicializadores, métricas, funciones de activación, capas e incluso modelos completos, se pueden personalizar de la misma manera. La mayoría de las veces, sólo necesitarás escribir una función simple con las entradas y salidas apropiadas. A continuación se muestran ejemplos de una función de activación personalizada (equivalente a `tf.keras.activations.softplus()` o `tf.nn.softplus()`), un inicializador Glorot personalizado (equivalente a `tf.keras.initializers.glorot_normal()`), un  $\ell_1$  regularizador (equivalente a `tf.keras.regularizers.l1(0.01)`) y una restricción personalizada que garantiza que todos los pesos sean positivos (equivalente a `tf.keras.constraints.nonneg()` o `tf.nn.relu()`):

```
def my_softplus(z): devuelve
    tf.math.log(1.0 + tf.exp(z))

def my_glorot_initializer(forma, dtype=tf.float32): stddev = tf.sqrt(2. / (forma[0] +
    forma[1])) return tf.random.normal(forma, stddev=stddev,
    tipod=tipod)

def my_l1_regularizer(pesos): devuelve
    tf.reduce_sum(tf.abs(0.01 * pesos))

def my_positive_weights(pesos): # el valor de retorno es solo tf.nn.relu(pesos) return
    tf.where(pesos < 0.,
        tf.zeros_like(pesos), pesos)
```

Como puede ver, los argumentos dependen del tipo de función personalizada. Estas funciones personalizadas se pueden utilizar normalmente, como se muestra aquí:

```
capa = tf.keras.layers.Dense(1, activación=my_softplus,
                            kernel_initializer=my_glorot_initializer,
                            kernel_regularizer=my_l1_regularizer,
                            kernel_constraint=mi_pesos_positivos)
```

La función de activación se aplicará a la salida de esta capa Densa y su resultado se pasará a la siguiente capa. Los pesos de la capa se inicializarán utilizando el valor devuelto por el inicializador. En cada paso de entrenamiento, los pesos se pasarán a la función de regularización para calcular la pérdida de regularización, que se agregará a la pérdida principal para obtener la pérdida final utilizada para el entrenamiento.

Finalmente, se llamará a la función de restricción después de cada paso de entrenamiento y los pesos de la capa serán reemplazados por los pesos restringidos.

Si una función tiene hiperparámetros que deben guardarse junto con el modelo, entonces querrá crear una subclase de la clase apropiada, como `tf.keras.regularizers.Regularizer`, `tf.keras.constraints.Constraint`, `tf.keras.initializers.Initializer`, o `tf.keras.layers.Layer` (para cualquier capa, incluidas las funciones de activación). Al igual que lo hizo con la pérdida personalizada, aquí hay una clase simple para la regularización  $\ell$  que guarda su hiperparámetro de factor (esta vez no necesita llamar al constructor principal ni al método `get_config()`, ya que no están definidos por la clase principal). ):

```
clase MyL1Regularizer (tf.keras.regularizers.Regularizer):
    def __init__(yo, factor):
        self.factor = factor

    def __call__(self, pesos):
        devolver tf.reduce_sum(tf.abs(self.factor * pesos))

    def get_config(yo):
        devolver {"factor": self.factor}
```

Tenga en cuenta que debe implementar el método `call()` para pérdidas, capas (incluidas funciones de activación) y modelos, o el método `__call__()` para regularizadores, inicializadores y restricciones. Para las métricas, las cosas son un poco diferentes, como verás ahora.

## Métricas personalizadas

Las pérdidas y las métricas no son conceptualmente lo mismo: las pérdidas (por ejemplo, entropía cruzada) se utilizan mediante el descenso de gradientes para entrenar un modelo, por lo que deben ser diferenciables (al menos en los puntos donde se evalúan) y sus gradientes no deben ser diferenciables. cero en todas partes. Además, está bien si los humanos no los interpretan fácilmente. Por el contrario, las métricas (por ejemplo, la precisión) se utilizan para evaluar un modelo: deben ser más fácilmente interpretables y pueden ser no diferenciables o tener gradientes cero en todas partes.

Dicho esto, en la mayoría de los casos, definir una función métrica personalizada es exactamente lo mismo que definir una función de pérdida personalizada. De hecho, incluso podríamos utilizar la función de pérdida de Huber que creamos anteriormente como métrica; funcionaría bien (y la persistencia también funcionaría de la misma manera, en este caso solo guardando el nombre de la función, "huber\_fn", no el umbral):

```
model.compile(loss="mse", optimizador="nadam", métricas=
[create_huber(2.0)])
```

Para cada lote durante el entrenamiento, Keras calculará esta métrica y realizará un seguimiento de su media desde el comienzo de la época. La mayoría de las veces, esto es exactamente lo que quieres. ¡Pero no siempre! Considere, por ejemplo, la precisión de un clasificador binario. Como vio en [el Capítulo 3](#), la precisión es el número de verdaderos positivos dividido por el número de predicciones positivas (incluidos tanto los verdaderos positivos como los falsos positivos). Supongamos que el modelo hizo cinco predicciones positivas en el primer lote, cuatro de las cuales fueron correctas: eso es un 80% de precisión. Entonces supongamos que el modelo hizo tres predicciones positivas en el segundo

lote, pero todos eran incorrectos: eso es 0% de precisión para el segundo lote. Si simplemente calculas la media de estas dos precisiones, obtienes 40%. Pero espere un segundo: ¡esa no es la precisión del modelo en estos dos lotes! De hecho, hubo un total de cuatro verdaderos positivos ( $4 + 0$ ) de ocho predicciones positivas ( $5 + 3$ ), por lo que la precisión general es del 50%, no del 40%. Lo que necesitamos es un objeto que pueda realizar un seguimiento del número de verdaderos positivos y el número de falsos positivos y que pueda calcular la precisión en función de estos números cuando se solicite. Esto es precisamente lo que hace la clase `tf.keras.metrics.Precision`:

```
>>> precisión = tf.keras.metrics.Precision() >>> precisión([0, 1, 1 ,  
1 , 0, 1, 0, 1], [1, 1, 0, 1, 0, 1, 0, 1]) <tf.Tensor: forma=(), dtype=float32, numpy=0.8> >>>  
  
precisión([0, 1, 0, 0, 1, 0, 1, 1], [1, 0 , 1, 1, 0, 0, 0, 0]) <tf.Tensor: forma=(),  
dtype=float32, numpy=0.5>
```

En este ejemplo, creamos un objeto de Precisión, luego lo usamos como una función, pasándole las etiquetas y predicciones para el primer lote, luego para el segundo lote (opcionalmente, también puede pasar pesos de muestra, si lo desea). Usamos la misma cantidad de verdaderos y falsos positivos que en el ejemplo que acabamos de discutir. Después del primer lote, devuelve una precisión del 80%; luego, después del segundo lote, devuelve el 50% (que es la precisión general hasta el momento, no la precisión del segundo lote). Esto se denomina métrica de transmisión (o métrica con estado), ya que se actualiza gradualmente, lote tras lote.

En cualquier momento, podemos llamar al método `result()` para obtener el valor actual de la métrica. También podemos ver sus variables (seguimiento del número de verdaderos y falsos positivos) usando el atributo de variables, y podemos restablecer estas variables usando el método `reset_states()`:

```
>>> precision.result() <tf.Tensor:
forma=(), dtype=float32, numpy=0.5> >>> precision.variables [<tf.Variable
'true_positives:0' [...], numpy=
array([4.], dtype=float32)>, <tf.Variable 'false_positives:0' [...], numpy=array([4.], dtype=float32)>]
>>> precision.reset_states
() # ambas variables se restablecen a 0.0
```

Si necesita definir su propia métrica de transmisión personalizada, cree una subclase de la clase `tf.keras.metrics.Metric`. Aquí tienes un básico.

ejemplo que realiza un seguimiento de la pérdida total de Huber y el número de casos vistos hasta ahora. Cuando se le solicita el resultado, devuelve la proporción, que es solo la pérdida media de Huber:

```
clase HuberMetric(tf.keras.metrics.Metric):
    def __init__(self, umbral=1.0, **kwargs):
        super().__init__(**kwargs) # maneja argumentos base
        (por ejemplo, tipo d)
        self.umbral = umbral
        self.huber_fn = create_huber(umbral) self.total =
        self.add_weight("total",
        inicializador="ceros")
        self.count = self.add_weight("cuenta",
        inicializador="ceros")

    def update_state(self, y_true, y_pred, sample_weight=Ninguno):
        sample_metrics =
            self.huber_fn(y_true, y_pred)

        self.total.assign_add(tf.reduce_sum(sample_metrics))
        self.count.assign_add(tf.cast(tf.size(y_true), tf.float32))

    resultado def (auto):
        devuelve self.total / self.count

    def get_config(self): base_config
        = super().get_config() return {**base_config, "umbral":
        self.threshold}
```

Repasemos este código:

- El constructor utiliza el método `add_weight()` para crear las variables necesarias para realizar un seguimiento del estado de la métrica en varios lotes; en este caso, la suma de todas las pérdidas de Huber (total) y el número de instancias vistas hasta el momento (recuento). Podrías crear variables manualmente si lo prefieres. Keras rastrea cualquier `tf.Variable` que se establece como atributo (y, más generalmente, cualquier objeto "rastreable", como capas o modelos).
- El método `update_state()` se llama cuando usas una instancia de esta clase como función (como hicimos con el objeto `Precision`). Actualiza las variables, dadas las etiquetas y predicciones para un lote (y los pesos de la muestra, pero en este caso los ignoramos).
- El método `result()` calcula y devuelve el resultado final, en este caso la métrica de Huber media en todas las instancias. Cuando se utiliza la métrica como función, primero se llama al método `update_state()`, luego se llama al método `result()` y se devuelve su salida.
- También implementamos el método `get_config()` para garantizar que el umbral se guarde junto con el modelo.
- La implementación predeterminada del método `reset_states()` restablece todas las variables a 0.0 (pero puedes anularla si es necesario).

#### NOTA

Keras se encargará de la persistencia variable sin problemas; no se requiere ninguna acción.

Cuando define una métrica usando una función simple, Keras la llama automáticamente para cada lote y realiza un seguimiento de la media.

durante cada época, tal como lo hicimos manualmente. Entonces, el único beneficio de nuestra clase HuberMetric es que se guardará el umbral. Pero, por supuesto, algunas métricas, como la precisión, no pueden simplemente promediarse en lotes: en esos casos, no hay otra opción que implementar una métrica de transmisión.

Ahora que ha creado una métrica de transmisión, crear una capa personalizada le parecerá un paseo por el parque.

## Capas personalizadas

En ocasiones, es posible que desees crear una arquitectura que contenga una capa exótica para la cual TensorFlow no proporciona una implementación predeterminada. O simplemente puede querer construir una arquitectura muy repetitiva, en la que un bloque particular de capas se repita muchas veces, y sería conveniente tratar cada bloque como una sola capa. En tales casos, querrás crear una capa personalizada.

Hay algunas capas que no tienen pesos, como tf.keras.layers.Flatten o tf.keras.layers.ReLU. Si desea crear una capa personalizada sin pesos, la opción más sencilla es escribir una función y envolverla en una capa tf.keras.layers.Lambda. Por ejemplo, la siguiente capa aplicará la función exponencial a sus entradas:

```
capa_exponencial = tf.keras.layers.Lambda(lambda x: tf.exp(x))
```

Luego, esta capa personalizada se puede utilizar como cualquier otra capa, utilizando la API secuencial, la API funcional o la API de subclases. También puedes usarlo como función de activación, o puedes usar activación=tf.exp. La capa exponencial se utiliza a veces en la capa de salida de un modelo de regresión cuando los valores a predecir tienen escalas muy diferentes (p. ej., 0,001, 10, 1000). De hecho, la función exponencial es una de las funciones de activación estándar en Keras, por lo que puedes usar activación="exponencial".

Como puede imaginar, para crear una capa con estado personalizada (es decir, una capa con pesos), debe crear una subclase de la clase `tf.keras.layers.Layer`. Por ejemplo, la siguiente clase implementa una versión simplificada de la capa Densa:

```
clase MyDense(tf.keras.layers.Layer):
    def __init__(self, unidades, activación=Ninguno, **kwargs):
        super().__init__(**kwargs)
        self.units = unidades
        self.activation =
            tf.keras.activations.get(activación)

    def build(self, lote_input_shape):
        self.kernel =
            self.add_weight( nombre="kernel",
                forma=[batch_input_shape[-1], self.units], inicializador="glorot_normal")

        self.bias = self.add_weight( nombre="sesgo",
            forma=[self.units], inicializador="ceros")

    def llamada(self, X):
        devuelve
            self.activation(X @ self.kernel + self.bias)

    def get_config(self):
        base_config =
            super().get_config()
        return {**base_config, "units": self.units, "activación":
            tf.keras.activations.serialize(self.activation)}
```

Repasemos este código:

- El constructor toma todos los hiperparámetros como argumentos (en este ejemplo, `unidades` y `activación`) y, lo que es más importante, también toma un argumento `**kwargs`. Llama al constructor principal y le pasa los `kwargs`: esto se encarga de los argumentos estándar como `input_shape`, `trainable` y `name`. Luego guarda los hiperparámetros como atributos, convirtiendo el argumento de `activación` en la función de `activación` apropiada usando el

Función `tf.keras.activations.get()` (acepta funciones, cadenas estándar como "relu" o "swish", o simplemente Ninguna).

- La función del método `build()` es crear las variables de la capa llamando al método `add_weight()` para cada peso. El método `build()` se llama la primera vez que se utiliza la capa. En ese punto, Keras conocerá la forma de las entradas de esta capa y la pasará al método `build()`, que suele ser necesario para<sup>10</sup> crear algunos de los pesos. Por ejemplo, necesitamos saber el número de neuronas en la capa anterior para crear la matriz de pesos de conexión (es decir, el "núcleo"): esto corresponde al tamaño de la última dimensión de las entradas. Al final del método `build()` (y solo al final), debes llamar al método `build()` del padre: esto le dice a Keras que la capa está construida (simplemente establece `self.built = True`).
- El método `call()` realiza las operaciones deseadas. En este caso, calculamos la multiplicación matricial de las entradas `X` y el núcleo de la capa, sumamos el vector de polarización y aplicamos la función de activación al resultado, y esto nos da la salida de la capa.
- El método `get_config()` es como en las clases personalizadas anteriores. Tenga en cuenta que guardamos la configuración completa de la función de activación llamando a `tf.keras.activaciones.serialize()`.

¡Ahora puedes usar una capa MyDense como cualquier otra capa!

## NOTA

Keras infiere automáticamente la forma de salida, excepto cuando la capa es dinámica (como verá en breve). En este (poco común) caso, debe implementar el método `Compute_output_shape()`, que debe devolver un objeto `TensorShape`.

Para crear una capa con múltiples entradas (por ejemplo, `Concatenar`), el argumento del método `call()` debe ser una tupla que contenga todas las entradas. Para crear una capa con múltiples salidas, el método `call()` debe devolver la lista de salidas. Por ejemplo, la siguiente capa de juguete toma dos entradas y devuelve tres salidas:

```
clase MyMultiLayer(tf.keras.layers.Layer): def llamada(self, X):
    X1, X2 = X return X1 + X2,
                           X1 * X2, X1 /
                           X2
```

Esta capa ahora se puede usar como cualquier otra capa, pero, por supuesto, solo usando las API funcionales y de subclases, no la API secuencial (que solo acepta capas con una entrada y una salida).

Si su capa necesita tener un comportamiento diferente durante el entrenamiento y durante las pruebas (por ejemplo, si usa capas `Dropout` o `BatchNormalization`), entonces debe agregar un argumento de entrenamiento al método `call()` y usar este argumento para decidir qué hacer. Por ejemplo, creemos una capa que agregue ruido gaussiano durante el entrenamiento (para regularización) pero no haga nada durante las pruebas (Keras tiene una capa que hace lo mismo, `tf.keras.layers.GaussianN`

```
clase MyGaussianNoise(tf.keras.layers.Layer): def __init__(self,
    stddev, **kwargs):
    super().__init__(**kwargs) self.stddev
    = stddev
```

llamada `def (self, X, entrenamiento=False):`

```
si entrena:  
    ruido = tf.random.normal(tf.shape(X),  
    stddev=self.stddev)  
    volver X + ruido  
demás:  
    volver X
```

¡Con eso, ahora puedes crear cualquier capa personalizada que necesites! Ahora veamos cómo crear modelos personalizados.

## Modelos personalizados

Ya vimos la creación de clases de modelo personalizadas en el [Capítulo 10](#), cuando analizamos la API de subclases. Es sencillo.<sup>11</sup> subclase la clase `tf.keras.Model`, crea capas y variables en el constructor e implementa el método `call()` para hacer lo que quieras que haga el modelo. Por ejemplo, supongamos que queremos construir el modelo representado en [la Figura 12-3](#).

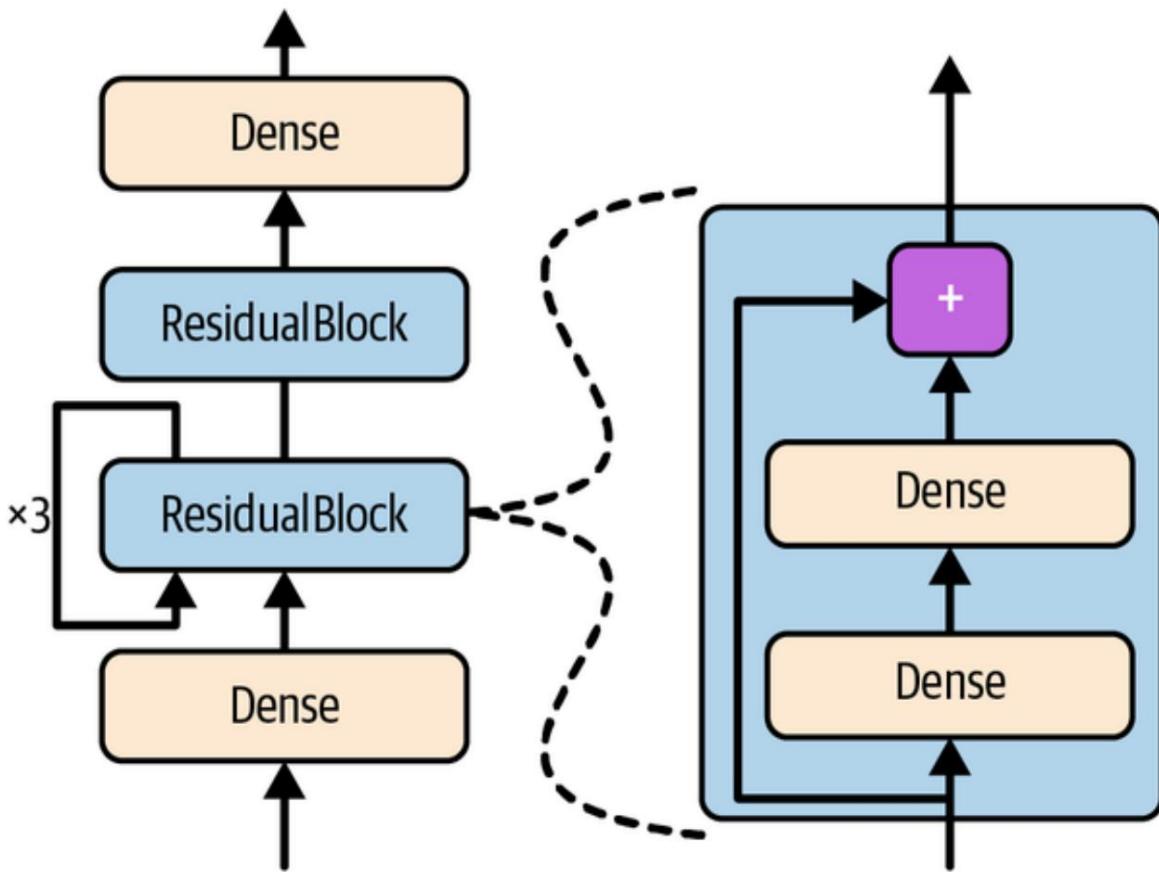


Figura 12-3. Ejemplo de modelo personalizado: un modelo arbitrario con una capa ResidualBlock personalizada que contiene una conexión de omisión

Las entradas pasan por una primera capa densa, luego por un bloque residual compuesto por dos capas densas y una operación de suma (como verá en [el Capítulo 14](#), un bloque residual suma sus entradas a sus salidas), luego por este mismo bloque residual tres más veces, luego a través de un segundo bloque residual, y el resultado final pasa a través de una capa de salida densa. No te preocupes si este modelo no tiene mucho sentido; es sólo un ejemplo para ilustrar el hecho de que puedes construir fácilmente cualquier tipo de modelo que deseas, incluso uno que contenga bucles y conexiones de salto. Para implementar este modelo, lo mejor es crear primero una capa ResidualBlock, ya que vamos a crear un par de bloques idénticos (y es posible que queramos reutilizarlos en otro modelo):

```
clase Bloque Residual (tf.keras.layers.Layer):
    def __init__(self, n_layers, n_neurons, **kwargs):
```

```

super().__init__(**kwargs) self.hidden =
[tf.keras.layers.Dense(n_neurons, activation="relu",
kernel_initializer="he_normal") para
    - en rango (n_capas)]

def llamada(self, entradas): Z =
    entradas para
    capa en self.hidden: Z = capa(Z) entradas
        de retorno + Z

```

Esta capa es un poco especial ya que contiene otras capas. Keras maneja esto de forma transparente: detecta automáticamente que el atributo oculto contiene objetos rastreables (capas en este caso), por lo que sus variables se agregan automáticamente a la lista de variables de esta capa. El resto de esta clase se explica por sí mismo. A continuación, usemos la API de subclases para definir el modelo en sí:

```

clase ResidualRegressor (tf.keras.Model):
    def __init__(self, salida_dim, **kwargs): super().__init__(**kwargs)
        self.hidden1 = tf.keras.layers.Dense(30,
            activation="relu",
kernel_initializer="he_normal") self.block1 =
            ResidualBlock(2, 30) self.block2 = ResidualBlock(2, 30)
            self.out = tf.keras.layers.Dense(output_dim)

    llamada def (auto, entradas):
        Z = self.hidden1(entradas) para dentro
        del rango(1 + 3): Z = self.block1(Z)

        Z = self.block2(Z) devuelve
        self.out(Z)

```

Creamos las capas en el constructor y las usamos en el método call(). Este modelo se puede utilizar como cualquier otro modelo (compilar

ajustarlo, evaluarlo y utilizarlo para hacer predicciones). Si también desea poder guardar el modelo usando el método `save()` y cargarlo usando la función `tf.keras.models.load_model()`, debe implementar el método `get_config()` (como hicimos antes) tanto en la Clase `ResidualBlock` y clase `ResidualRegressor`.

Alternativamente, puede guardar y cargar los pesos usando los métodos `save_weights()` y `load_weights()`.

La clase `Modelo` es una subclase de la clase `Capa`, por lo que los modelos se pueden definir y utilizar exactamente como capas. Pero un modelo tiene algunas funcionalidades adicionales, incluidos, por supuesto, sus métodos `compilar()`, `ajustar()`, `evaluar()` y `predecir()` (y algunas variantes), además del método `get_layer()` (que puede devolver cualquiera de los datos del modelo). capas por nombre o por índice) y el método `save()` (y soporte para `tf.keras.models.load_model()` y `tf.keras.models.clone_model()`).

CONSEJO

Si los modelos proporcionan más funcionalidad que las capas, ¿por qué no definir cada capa como un modelo? Bueno, técnicamente podrías hacerlo, pero generalmente es más claro distinguir los componentes internos de tu modelo (es decir, capas o bloques de capas reutilizables) del modelo mismo (es decir, el objeto que entrenarás). El primero debería subclásificarse la clase `Capa`, mientras que el segundo debería subclásificarse la clase `Modelo`.

Con eso, puedes construir de forma natural y concisa casi cualquier modelo que encuentres en un artículo, usando la API secuencial, la API funcional, la API de subclásificación o incluso una combinación de estas. ¿“Casi” cualquier modelo? Sí, todavía hay algunas cosas que debemos considerar: primero, cómo definir pérdidas o métricas basadas en los aspectos internos del modelo y, segundo, cómo crear un ciclo de entrenamiento personalizado.

## Pérdidas y métricas basadas en elementos internos del modelo

Todas las pérdidas y métricas personalizadas que definimos anteriormente se basaron en las etiquetas y las predicciones (y, opcionalmente, en las ponderaciones de las muestras). Habrá ocasiones en las que querrás definir pérdidas en función de otras partes de tu modelo, como los pesos o activaciones de sus capas ocultas. Esto puede resultar útil para fines de regularización o para monitorear algún aspecto interno de su modelo.

Para definir una pérdida personalizada basada en los aspectos internos del modelo, calculela en función de cualquier parte del modelo que desee y luego pase el resultado al método `add_loss()`. Por ejemplo, creemos un modelo MLP de regresión personalizado compuesto por una pila de cinco capas ocultas más una capa de salida. Este modelo personalizado también tendrá una salida auxiliar encima de la capa oculta superior. La pérdida asociada con esta salida auxiliar se llamará pérdida de reconstrucción (ver [Capítulo 17](#)): es la diferencia media cuadrática entre la reconstrucción y las entradas. Al agregar esta pérdida de reconstrucción a la pérdida principal, alentaremos al modelo a preservar la mayor cantidad de información posible a través de las capas ocultas, incluso información que no es directamente útil para la tarea de regresión en sí. En la práctica, esta pérdida a veces mejora la generalización (es una pérdida de regularización). También es posible agregar una métrica personalizada utilizando el método `add_metric()` del modelo. Aquí está el código para este modelo personalizado con una pérdida de reconstrucción personalizada y una métrica correspondiente:

```
clase ReconstructingRegressor(tf.keras.Model): def __init__(self,
    output_dim, **kwargs): super().__init__(**kwargs) self.hidden
        = [tf.keras.layers.Dense(30, activación="relu",
        kernel_initializer="he_normal") para
            - en rango(5)]
        self.out = tf.keras.layers.Dense(output_dim) self.reconstruction_mean
        = tf.keras.metrics.Mean( name="reconstruction_error")
```

```

def build(self, forma_entrada_por_lotes):
    n_entradas = forma_entrada_por_lotes[-1]
    self.reconstruct = tf.keras.layers.Dense(n_entradas)

def call(self, entradas, entrenamiento=False): Z =
    entradas para
    la capa en self.hidden: Z = capa(Z)
        reconstrucción
    = self.reconstruct(Z) recon_loss =
        tf.reduce_mean(tf.square(reconstrucción - entradas)) self.add_loss(0.05
        * recon_loss) si se está entrenando:
            resultado = self.reconstruction_mean(recon_loss)
            self.add_metric(resultado) return
            self.out(Z)

```

Repasemos este código:

- El constructor crea el DNN con cinco capas densas ocultas y una capa de salida densa. También creamos una métrica de transmisión media para realizar un seguimiento del error de reconstrucción durante el entrenamiento.
- El método build() crea una capa extra densa que se utilizará para reconstruir las entradas del modelo. Debe crearse aquí porque su número de unidades debe ser igual al número de entradas, y este número se desconoce antes de llamar al método build() [12](#).
- El método call() procesa las entradas a través de las cinco capas ocultas y luego pasa el resultado a través de la capa de reconstrucción, que produce la reconstrucción.
- Luego, el método call() calcula la pérdida de reconstrucción (la diferencia cuadrática media entre la reconstrucción y las entradas) y la agrega a la lista de pérdidas del modelo usando el método add\_loss(). Observe que reducimos la pérdida de reconstrucción multiplicándola por 0,05 (este es un [13](#)

hiperparámetro que puedes ajustar). Esto garantiza que las pérdidas por reconstrucción no dominen a las pérdidas principales.

- A continuación, solo durante el entrenamiento, el método call() actualiza la métrica de reconstrucción y la agrega al modelo para que pueda mostrarse. En realidad, este ejemplo de código se puede simplificar llamando a self.add\_metric(recon\_loss): Keras rastrearía automáticamente la media por usted.
- Finalmente, el método call() pasa la salida de las capas ocultas a la capa de salida y devuelve su salida.

Tanto la pérdida total como la pérdida por reconstrucción disminuirán durante el entrenamiento:

```
Época 1/5
363/363 [=====] - 1s 820us/paso - pérdida: 0,7640 - error_reconstrucción:
1,2728 Época 2/5 363/363 [=====] - 0s
809us /paso -
pérdida: 0,4584 - error_reconstrucción: 0,6340 [...]
```

En la mayoría de los casos, todo lo que hemos discutido hasta ahora será suficiente para implementar cualquier modelo que desee construir, incluso con arquitecturas, pérdidas y métricas complejas. Sin embargo, para algunas arquitecturas, como las GAN (consulte el [Capítulo 17](#)), deberá personalizar el ciclo de entrenamiento. Antes de llegar allí, debemos ver cómo calcular gradientes automáticamente en TensorFlow.

## Calcular gradientes usando Autodiff

Para entender cómo usar autodiff (consulte el [Capítulo 10](#) y el [Apéndice B](#)) para calcular gradientes automáticamente, consideraremos una función de juguete simple:

```
definición f(w1, w2):
    devolver 3 * w1 ** 2 + 2 * w1 * w2
```

Si sabes cálculo, puedes encontrar analíticamente que la derivada parcial de esta función con respecto a  $w_1$  es  $6 * w_1 + 2 * w_2$ .

También puedes encontrar que su derivada parcial con respecto a  $w_2$  es  $2 * w_1$ .

Por ejemplo, en el punto  $(w_1, w_2) = (5, 3)$ , estas derivadas parciales son iguales a 36 y 10, respectivamente, por lo que el vector gradiente en este punto es (36, 10). Pero si se tratara de una red neuronal, la función sería mucho más compleja, normalmente con decenas de miles de parámetros, y encontrar las derivadas parciales analíticamente a mano sería una tarea prácticamente imposible. Una solución podría ser calcular una aproximación de cada derivada parcial midiendo cuánto cambia la salida de la función cuando modificas el parámetro correspondiente en una pequeña cantidad:

```
>>> w1, w2 = 5, 3 >>>
eps = 1e-6 >>> (f(w1
+ eps, w2) - f(w1, w2)) / eps 36.000003007075065 >>>
(f(w1, w2 + eps) - f(w1,
w2)) / eps 10.000000003174137
```

¡Parece correcto! Esto funciona bastante bien y es fácil de implementar, pero es sólo una aproximación y, lo que es más importante, es necesario llamar a  $f()$  al menos una vez por parámetro (no dos veces, ya que podríamos calcular  $f(w_1, w_2)$  solo una vez). Tener que llamar a  $f()$  al menos una vez por parámetro hace que este enfoque sea intratable para redes neuronales grandes. Entonces, en su lugar, deberíamos usar autodiff en modo inverso.

TensorFlow hace esto bastante simple:

```
w1, w2 = tf.Variable(5.), tf.Variable(3.) con tf.GradientTape()
como cinta: z = f(w1, w2)
```

```
gradientes = cinta.gradiente(z, [w1, w2])
```

Primero definimos dos variables `w1` y `w2`, luego creamos un contexto `tf.GradientTape` que registrará automáticamente cada operación que involucre una variable, y finalmente le pedimos a esta cinta que calcule los gradientes del resultado `z` con respecto a ambas variables [`w1`, `w2`]. Echemos un vistazo a los gradientes que calculó TensorFlow:

```
>>> gradientes
```

```
[<tf.Tensor: forma=(), dtype=float32, numpy=36.0>, <tf.Tensor: forma=(),
  dtype=float32, numpy=10.0>]
```

¡Perfecto! El resultado no solo es preciso (la precisión solo está limitada por los errores de punto flotante), sino que el método `gradient()` solo realiza los cálculos registrados una vez (en orden inverso), sin importar cuántas variables haya, por lo que es increíblemente eficiente. ¡Es como magia!

#### CONSEJO

Para ahorrar memoria, coloque solo el mínimo estricto dentro del bloque `tf.GradientTape()`. Alternativamente, pausa la grabación creando un bloque `with tape.stop_recording()` dentro del bloque `tf.GradientTape()`.

La cinta se borra automáticamente inmediatamente después de llamar a su método `gradient()`, por lo que obtendrá una excepción si intenta llamar a `gradient()` dos veces:

```
con tf.GradientTape() como cinta: z = f(w1, w2)
```

```
dz_dw1 = tape.gradient(z, w1) # devuelve el tensor 36.0 dz_dw2 = tape.gradient(z,
  w2) # genera un RuntimeError!
```

Si necesita llamar a `gradient()` más de una vez, debe hacer que la cinta sea persistente y eliminarla cada vez que termine con ella para liberarla.

[14 recursos:](#)

```
con tf.GradientTape(persistent=True) como cinta:
```

```
    z = f(w1, w2)
```

```
dz_dw1 = tape.gradient(z, w1) # devuelve el tensor 36.0 dz_dw2 = tape.gradient(z,
w2) # devuelve el tensor 10.0, ¡funciona bien ahora!
```

[la cinta](#)

De forma predeterminada, la cinta sólo rastreará operaciones que involucren variables, por lo que si intenta calcular el gradiente de `z` con respecto a algo que no sea una variable, el resultado será Ninguno:

```
c1, c2 = tf.constant(5.), tf.constant(3.) con tf.GradientTape() como
cinta: z = f(c1, c2)
```

```
gradientes = tape.gradient(z, [c1, c2]) # devuelve [Ninguno, Ninguno]
```

Sin embargo, puede forzar la cinta para que observe los tensores que desee y registre cada operación que los involucre. Luego puedes calcular gradientes con respecto a estos tensores, como si fueran variables:

```
con tf.GradientTape() como cinta: tape.watch(c1)
    tape.watch(c2) z =
        f(c1, c2)
```

```
gradientes = tape.gradient(z, [c1, c2]) # devuelve [tensor 36., tensor 10.]
```

Esto puede ser útil en algunos casos, como si desea implementar una pérdida de regularización que penalice las activaciones que varían mucho cuando las entradas varían poco: la pérdida se basará en el gradiente del

activaciones con respecto a las entradas. Dado que las entradas no son variables, deberá indicarle a la cinta que las mire.

La mayoría de las veces se utiliza una cinta de gradiente para calcular los gradientes de un único valor (normalmente la pérdida) con respecto a un conjunto de valores (normalmente los parámetros del modelo). Aquí es donde brilla el autodiff en modo inverso, ya que sólo necesita hacer una pasada hacia adelante y una pasada hacia atrás para obtener todos los gradientes a la vez. Si intenta calcular los gradientes de un vector, por ejemplo, un vector que contiene múltiples pérdidas, TensorFlow calculará los gradientes de la suma del vector. Entonces, si alguna vez necesita obtener los gradientes individuales (por ejemplo, los gradientes de cada pérdida con respecto a los parámetros del modelo), debe llamar al método `jacobian()` de la cinta: realizará una diferenciación automática en modo inverso una vez por cada pérdida en el vector. (todo en paralelo por defecto). Incluso es posible calcular derivadas parciales de segundo orden (las hessianas, es decir, las derivadas parciales de las derivadas parciales), pero esto rara vez es necesario en la práctica (consulte la sección “Cálculo de gradientes usando Autodiff” del cuaderno de este capítulo para ver un ejemplo). ).

En algunos casos, es posible que desee evitar que los gradientes se propaguen hacia atrás a través de alguna parte de su red neuronal. Para hacer esto, debes usar la función `tf.stop_gradient()`. La función devuelve sus entradas durante el paso hacia adelante (como `tf.identity()`), pero no deja pasar los gradientes durante la propagación hacia atrás (actúa como una constante):

```
def f(w1, w2):
    devuelve 3 * w1 ** 2 + tf.stop_gradient(2 * w1 * w2)

con tf.GradientTape() como cinta: z = f(w1,
    w2) # el pase hacia adelante no se ve afectado por stop_gradient()

gradientes = tape.gradient(z, [w1, w2]) # devuelve [tensor 30., Ninguno]
```

Finalmente, es posible que ocasionalmente te encuentres con algunos problemas numéricos al calcular gradientes. Por ejemplo, si calcula los gradientes de

la función de raíz cuadrada en  $x = 10$  en  $-50$ , el resultado será infinito. En realidad, la pendiente en ese punto no es infinita, pero es más de lo que los flotantes de 32 bits pueden manejar:

```
>>> x = tf.Variable(1e-50) >>> con
tf.GradientTape() como cinta: z = tf.sqrt(x)
...
...
>>> tape.gradient(z, [x]) [<tf.Tensor:
 forma=(), dtype=float32, numpy=inf>]
```

Para resolver esto, suele ser una buena idea agregar un valor pequeño a  $x$  (como  $10^{-6}$ ) al calcular su raíz cuadrada.

La función exponencial también es una fuente frecuente de dolores de cabeza, ya que crece extremadamente rápido. Por ejemplo, la forma en que se definió `my_softplus()` anteriormente no es numéricamente estable. Si calcula `my_softplus(100.0)`, obtendrá infinito en lugar del resultado correcto (aproximadamente 100). Pero es posible reescribir la función para hacerla numéricamente estable: la función `softplus` se define como  $\log(1 + \exp(z))$ , que también es igual a  $\log(1 + \exp(-|z|)) + \max(z, 0)$  (ver el cuaderno para la prueba matemática) y la ventaja de esta segunda forma es que el término exponencial no puede explotar. Entonces, aquí hay una mejor implementación de la función `my_softplus()`:

```
def my_softplus(z): devuelve
    tf.math.log(1 + tf.exp(-tf.abs(z))) + tf.maximum(0., z)
```

En algunos casos raros, una función numéricamente estable aún puede tener gradientes numéricamente inestables. En tales casos, tendrás que decirle a TensorFlow qué ecuación usar para los gradientes, en lugar de permitirle usar `autodiff`. Para esto, debes usar el `@tf. decorador custom_gradient` al definir la función y devolver tanto el resultado habitual de la función como una función que calcula el

gradientes. Por ejemplo, actualicemos la función `my_softplus()` para que también devuelva una función de gradientes numéricamente estable:

```
@tf.custom_gradient def
my_softplus(z): def
    my_softplus_gradients(grads): # grads = retroprop'ed desde
    capas superiores return grads * (1 - 1 /
        (1 + tf.exp(z))) # grads estables de softplus

    resultado = tf.math.log(1 + tf.exp(-tf.abs(z))) + tf.maximum(0., z)
devuelve resultado,
my_softplus_gradients
```

Si conoces cálculo diferencial (consulta el cuaderno tutorial sobre este tema), puedes encontrar que la derivada de  $\log(1 + \exp(z))$  es  $\exp(z) / (1 + \exp(z))$ . Pero esta forma no es estable: para valores grandes de  $z$ , termina calculando infinito dividido por infinito, lo que devuelve NaN. Sin embargo, con un poco de manipulación algebraica, puedes demostrar que también es igual a  $1 - 1 / (1 + \exp(z))$ , que es estable. La función

`my_softplus_gradients()` utiliza esta ecuación para calcular los gradientes. Tenga en cuenta que esta función recibirá como entrada los gradientes que se propagaron hacia atrás hasta el momento, hasta la función `my_softplus()`, y de acuerdo con la regla de la cadena debemos multiplicarlos con los gradientes de esta función.

Ahora, cuando calculamos los gradientes de la función `my_softplus()`, obtenemos el resultado adecuado, incluso para valores de entrada grandes.

¡Felicidades! Ahora puedes calcular los gradientes de cualquier función (siempre que sea diferenciable en el punto donde la calculas), incluso bloquear la retropropagación cuando sea necesario, ¡y escribir tus propias funciones de gradiente! Probablemente esto represente más flexibilidad de la que necesitará, incluso si crea sus propios circuitos de entrenamiento personalizados. Verás cómo hacerlo a continuación.

Bucles de entrenamiento personalizados

En algunos casos, es posible que el método `fit()` no sea lo suficientemente flexible para lo que necesita hacer. Por ejemplo, el [documento Wide & Deep](#) que analizamos en [el Capítulo 10](#) utiliza dos optimizadores diferentes: uno para la ruta amplia y el otro para la ruta profunda. Dado que el método `fit()` solo utiliza un optimizador (el que especificamos al compilar el modelo), implementar este documento requiere escribir su propio bucle personalizado.

También es posible que desee escribir ciclos de entrenamiento personalizados simplemente para sentirse más seguro de que hacen precisamente lo que usted pretende que hagan (tal vez no esté seguro de algunos detalles del método `fit()`). A veces puede parecer más seguro hacer todo explícito. Sin embargo, recuerde que escribir un ciclo de entrenamiento personalizado hará que su código sea más largo, más propenso a errores y más difícil de mantener.

CONSEJO

A menos que esté aprendiendo o realmente necesite flexibilidad adicional, debería preferir usar el método `fit()` en lugar de implementar su propio ciclo de entrenamiento, especialmente si trabaja en equipo.

Primero, construyamos un modelo simple. No es necesario compilarlo, ya que manejaremos el ciclo de entrenamiento manualmente:

```
I2_reg = tf.keras.regularizers.l2(0.05)
model =
tf.keras.models.Sequential([
    tf.keras.layers.Dense(30,
        activation="relu", kernel_initializer="he_normal",
        kernel_regularizer=I2_reg),
    tf.keras.layers.Dense(1, kernel_regularizer=I2_reg)
])
```

A continuación, creemos una pequeña función que muestreará aleatoriamente un lote de instancias del conjunto de entrenamiento (en el [Capítulo 13](#) analizaremos la API `tf.data`, que ofrece una alternativa mucho mejor):

```
def lote_aleatorio(X, y, tamaño_lote=32): idx =
    np.random.randint(len(X), tamaño=tamaño_lote) return X[idx], y[idx]
```

También definamos una función que mostrará el estado del entrenamiento, incluido el número de pasos, el número total de pasos, la pérdida media desde el inicio de la época (usaremos la métrica Media para calcularla) y otras métricas:

```
def print_status_bar(paso, total, pérdida, métricas=Ninguno): métricas = - ".join([f'{m.name}':
    {m.result():.4f}" para m en [pérdida] + (métricas o
[]]) fin
=         """ si paso < total else "\n" print(f"\r{paso}/{total}"
- + métricas, fin=fin)
```

Este código se explica por sí mismo, a menos que no esté familiarizado con el formato de cadenas de Python: `{m.result():.4f}` formateará el resultado de la métrica como un flotante con cuatro dígitos después del punto decimal y utilizará `\r` (retorno de carro). junto con `end=""` garantiza que la barra de estado siempre se imprima en la misma línea.

Dicho esto, ¡pongámonos manos a la obra! Primero, necesitamos definir algunos hiperparámetros y elegir el optimizador, la función de pérdida y las métricas (solo el MAE en este ejemplo):

```
n_epochs = 5
tamaño_lote = 32 n_steps
= len(X_train) // optimizador de tamaño_lote =
tf.keras.optimizers.SGD(learning_rate=0.01) loss_fn = tf.keras.losses.mean_squared_error
mean_loss = tf.keras.metrics.Mean(nombre= "mean_loss") métricas =
[tf.keras.metrics.MeanAbsoluteError()]
```

¡Y ahora estamos listos para construir el bucle personalizado!

```
para época en rango (1, n_epochs + 1): print("Epoch {}"
    {}.format(epoch, n_epochs))
```

```

para paso en rango (1, n_pasos + 1):
    X_batch, y_batch = random_batch(X_train_scaled, y_train) con

    tf.GradientTape() como cinta: y_pred = model(X_batch,
                                                Training=True) main_loss = tf.reduce_mean(loss_fn(y_batch,
                                                y_pred))

    pérdida = tf.add_n([pérdida_principal] + modelo.pérdidas)

    gradientes = cinta.gradiente (pérdida,
                                modelo.trainable_variables)
    optimizador.apply_gradients(zip(gradientes,
                                modelo.trainable_variables))
    mean_loss(loss) para
    métrica en métricas:
    métrica(y_batch, y_pred)

    print_status_bar(paso, n_pasos, pérdida_media, métricas)

    para métrica en [mean_loss] + métricas:
        metric.reset_states()

```

Están sucediendo muchas cosas en este código, así que repasémoslas:

- Creamos dos bucles anidados: uno para las épocas y el otro para los lotes dentro de una época.
- Luego tomamos muestras de un lote aleatorio del conjunto de entrenamiento.
- Dentro del bloque `tf.GradientTape()`, hacemos una predicción para un lote, usando el modelo como función, y calculamos la pérdida: es igual a la pérdida principal más las otras pérdidas (en este modelo, hay una regularización pérdida por capa). Dado que la función `mean_squared_error()` devuelve una pérdida por instancia, calculamos la media sobre el lote usando `tf.reduce_mean()` (si quisieras aplicar pesos diferentes a cada instancia, aquí es donde lo harías). Las pérdidas de regularización ya se reducen a un único escalar

cada uno, por lo que solo necesitamos sumarlos (usando `tf.add_n()`, que suma múltiples tensores de la misma forma y tipo de datos).

- A continuación, le pedimos a la cinta que calcule los gradientes de pérdida con respecto a cada variable entrenable (*¡no todas las variables!*) y los aplicamos al optimizador para realizar un paso de descenso de gradiente.
- Luego actualizamos la pérdida media y las métricas (durante la época actual) y mostramos la barra de estado.
- Al final de cada época, restablecemos los estados de pérdida media y las métricas.

Si desea aplicar recorte de degradado (consulte [el Capítulo 11](#)), establezca el hiperparámetro `clipnorm` o `clipvalue` del optimizador. Si desea aplicar cualquier otra transformación a los gradientes, simplemente hágalo antes de llamar al método `apply_gradients()`. Y si desea agregar restricciones de peso a su modelo (por ejemplo, configurando `kernel_constraint` o `sesgo_constraint` al crear una capa), debe actualizar el bucle de entrenamiento para aplicar estas restricciones justo después de `apply_gradients()`, así:

```
para variable en model.variables:
    si variable.constraint no es Ninguna:
        variable.asignar(variable.restricción(variable))
```

#### ADVERTENCIA

No olvide establecer `train=True` cuando llame al modelo en el ciclo de entrenamiento, especialmente si su modelo se comporta de manera diferente durante el entrenamiento y las pruebas (por ejemplo, si usa `BatchNormalization` o `Dropout`). Si es un modelo personalizado, asegúrese de propagar el argumento de entrenamiento a las capas a las que llama su modelo.

Como puedes ver, hay muchas cosas que debes hacer bien y es fácil cometer un error. Pero lo bueno es que te llenas

control.

Ahora que sabes cómo personalizar cualquier parte de tus modelos y algoritmos de entrenamiento, veamos cómo puedes usar la función de generación automática de gráficos de TensorFlow: puede acelerar considerablemente tu código personalizado y también lo hará portátil a cualquier plataforma compatible con TensorFlow. (ver [Capítulo 19](#)). 15

## Funciones y gráficos de TensorFlow

En TensorFlow 1, los gráficos eran inevitables (al igual que las complejidades que los acompañaban) porque eran una parte central de la API de TensorFlow. Desde TensorFlow 2 (lanzado en 2019), los gráficos todavía están ahí, pero no son tan centrales y son mucho (¡mucho!) más simples de usar. Para mostrar cuán simple es, comencemos con una función trivial que calcula el cubo de su entrada:

```
def cubo(x):
    devolver x ** 3
```

Obviamente podemos llamar a esta función con un valor de Python, como un int o un float, o podemos llamarla con un tensor:

```
>>> cubo(2) 8

>>> cubo(tf.constant(2.0)) <tf.Tensor:
forma=(), dtype=float32, numpy=8.0>
```

Ahora, usemos `tf.function()` para convertir esta función de Python en una función de TensorFlow:

```
>>> tf_cube = tf.function(cubo) >>> tf_cube

<tensorflow.python.eager.def_function.Funcióen en 0x7fbfe0c54d50>
```

Esta función TF se puede usar exactamente como la función Python original y devolverá el mismo resultado (pero siempre como tensores):

```
>>> tf_cube(2)
<tf.Tensor: forma=(), dtype=int32, numpy=8> >>>
tf_cube(tf.constant(2.0)) <tf.Tensor:
forma=(), dtype=float32 , numeroso=8.0>
```

¡Debajo del capó, `tf.function()` analizó los cálculos realizados por la función `cube()` y generó un gráfico de cálculo equivalente! Como puede ver, fue bastante indoloro (veremos cómo funciona esto en breve). Alternativamente, podríamos haber usado `tf.function` como decorador; En realidad, esto es más común:

```
@tf.function def
tf_cube(x): devuelve
x ** 3
```

La función Python original todavía está disponible a través del atributo `python_function` de la función TF, en caso de que alguna vez la necesite:

```
>>> tf_cube.python_function(2) 8
```

TensorFlow optimiza el gráfico de cálculo, elimina los nodos no utilizados, simplifica las expresiones (por ejemplo,  $1 + 2$  se reemplazaría por  $3$ ) y más. Una vez que el gráfico optimizado está listo, la función TF ejecuta eficientemente las operaciones en el gráfico, en el orden apropiado (y en paralelo cuando puede). Como resultado, una función TF normalmente se ejecutará mucho más rápido que la función Python original, especialmente si realiza cálculos complejos. La mayoría de las veces no necesitarás saber más que eso: cuando quieras potenciar una función de Python, simplemente transfórmala en una función TF. ¡Eso es todo!

Además, si configura `jit_compile=True` al llamar a `tf.function()`, entonces TensorFlow utilizará la función lineal acelerada.

álgebra (XLA) para compilar núcleos dedicados para su gráfico, a menudo fusionando múltiples operaciones. Por ejemplo, si su función TF llama a `tf.reduce_sum(a * b + c)`, entonces, sin XLA, la función primero necesitaría calcular `a * b` y almacenar el resultado en una variable temporal, luego agregar `c` a esa variable y, por último, llame a `tf.reduce_sum()` en el resultado. Con XLA, todo el cálculo se compila en un único núcleo, que calculará `tf.reduce_sum(a * b + c)` de una sola vez, sin utilizar ninguna variable temporal grande. Esto no sólo será mucho más rápido, sino que también utilizará muchísimo menos RAM.

Cuando escribe una función de pérdida personalizada, una métrica personalizada, una capa personalizada o cualquier otra función personalizada y la usa en un modelo de Keras (como lo hemos hecho a lo largo de este capítulo), Keras convierte automáticamente su función en una función TF. No es necesario utilizar `tf.function()`. Así que la mayoría de las veces, la magia es 100% transparente. Y si desea que Keras use XLA, solo necesita configurar `jit_compile=True` al llamar al método `compile()`. ¡Fácil!

CONSEJO

Puede indicarle a Keras que no convierta sus funciones de Python en funciones TF configurando `dinámico=True` al crear una capa personalizada o un modelo personalizado. Alternativamente, puede configurar `run_eagerly=True` al llamar al método `compile()` del modelo.

De forma predeterminada, una función TF genera un nuevo gráfico para cada conjunto único de formas de entrada y tipos de datos y lo almacena en caché para llamadas posteriores. Por ejemplo, si llama a `tf_cube(tf.constant(10))`, se generará un gráfico para tensores `int32` de forma `[]`. Luego, si llama a `tf_cube(tf.constant(20))`, se reutilizará el mismo gráfico. Pero si luego llamas a `tf_cube(tf.constant([10, 20]))`, se generará un nuevo gráfico para tensores de forma `int32 [2]`. Así es como las funciones TF manejan el polimorfismo (es decir, diferentes tipos de argumentos y

formas). Sin embargo, esto solo es cierto para los argumentos tensoriales: si pasa valores numéricos de Python a una función TF, se generará un nuevo gráfico para cada valor distinto: por ejemplo, llamar a `tf_cube(10)` y `tf_cube(20)` generará dos gráficos.

#### ADVERTENCIA

Si llama a una función TF muchas veces con diferentes valores numéricos de Python, se generarán muchos gráficos, lo que ralentizará su programa y consumirá mucha RAM (debe eliminar la función TF para liberarla). Los valores de Python deben reservarse para argumentos que tendrán pocos valores únicos, como hiperparámetros como el número de neuronas por capa. Esto permite a TensorFlow optimizar mejor cada variante de su modelo.

## Autógrafo y calco

Entonces, ¿cómo genera gráficos TensorFlow? Comienza analizando el código fuente de la función Python para capturar todas las declaraciones de flujo de control, como bucles `for`, bucles `while` y declaraciones `if`, así como declaraciones `break`, `continue` y `return`. Este primer paso se llama AutoGraph. La razón por la que TensorFlow tiene que analizar el código fuente es que Python no proporciona ninguna otra forma de capturar declaraciones de flujo de control: ofrece métodos mágicos como `__add__()` y `__mul__()` para capturar operadores como `+` y `*`, pero no hay `__while__()` o `__if__()` métodos mágicos. Después de analizar el código de la función, AutoGraph genera una versión mejorada de esa función en la que todas las declaraciones de flujo de control se reemplazan por las operaciones apropiadas de TensorFlow, como `tf.while_loop()` para bucles y `tf.cond()` para declaraciones `if`. Por ejemplo, en [la Figura 12-4](#), AutoGraph analiza el código fuente Python y genera la función `tf_sum_squares()`. En esta función, el bucle `for` se reemplaza por la definición de la función `loop_body()` (que contiene el cuerpo del bucle `for` original).

bucle), seguido de una llamada a la función `for_stmt()`. Esta llamada creará la operación `tf.while_loop()` apropiada en el gráfico de cálculo.

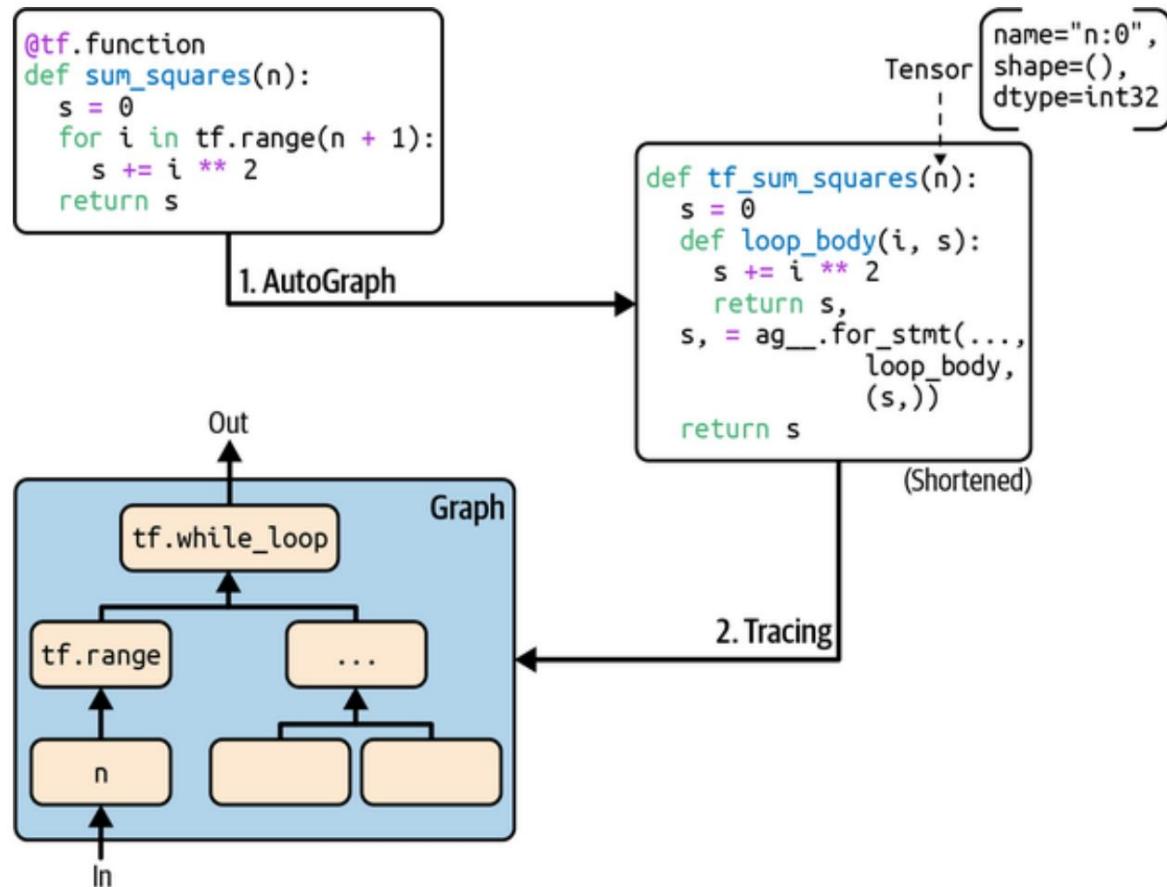


Figura 12-4. Cómo TensorFlow genera gráficos usando AutoGraph y rastreo

A continuación, TensorFlow llama a esta función "actualizada", pero en lugar de pasar el argumento, pasa un tensor simbólico: un tensor sin ningún valor real, solo un nombre, un tipo de datos y una forma. Por ejemplo, si llama a `sum_squares(tf.constant(10))`, entonces la función `tf_sum_squares()` se llamará con un tensor simbólico de tipo `int32` y forma `[]`. La función se ejecutará en modo gráfico, lo que significa que cada operación de TensorFlow agregará un nodo en el gráfico para representarse a sí misma y a sus tensores de salida (a diferencia del modo normal, llamado ejecución ansiosa o modo ansioso). En el modo gráfico, las operaciones TF no realizan ningún cálculo. El modo gráfico era el modo predeterminado en TensorFlow 1. En [la Figura 12-](#)

la función `tf.__sum_squares()` se llama con un tensor simbólico como argumento (en este caso, un tensor `int32` de forma `[]`) y el gráfico final se genera durante el rastreo. Los nodos representan operaciones y las flechas representan tensores (tanto la función generada como el gráfico están simplificados).

## CONSEJO

Para ver el código fuente de la función generada, puede llamar a `tf.autograph.to_code(sum_squares.python_function)`. El código no pretende ser bonito, pero a veces puede ayudar a depurar.

## Reglas de función TF

La mayoría de las veces, convertir una función de Python que realiza operaciones de TensorFlow en una función de TF es trivial: decórala con `@tf.function` o deja que Keras se encargue de ello por ti. Sin embargo, hay algunas reglas a respetar:

- Si llama a cualquier biblioteca externa, incluida NumPy o incluso la biblioteca estándar, esta llamada se ejecutará solo durante el seguimiento; no será parte del gráfico. De hecho, un gráfico de TensorFlow solo puede incluir construcciones de TensorFlow (tensores, operaciones, variables, conjuntos de datos, etc.). Por lo tanto, asegúrese de usar `tf.reduce_sum()` en lugar de `np.sum()`, `tf.sort()` en lugar de la función incorporada `sorted()`, etc. (a menos que realmente desee que el código se ejecute solo durante el seguimiento). Esto tiene algunas implicaciones adicionales:
  - Si define una función TF `f(x)` que simplemente devuelve `np.random.rand()`, solo se generará un número aleatorio cuando se rastree la función, por lo que `f(tf.constant(2.))` y `f(tf.constant(3.))` devolverá el mismo número aleatorio, pero `f(tf.constant([2., 3.]))` devolverá uno diferente.

Si reemplaza `np.random.rand()` con `tf.random.uniform([])`, se generará un nuevo número aleatorio en cada llamada, ya que la operación será parte del gráfico.

- Si su código que no es de TensorFlow tiene efectos secundarios (como registrar algo o actualizar un contador de Python), entonces no debe esperar que esos efectos secundarios ocurran cada vez que llame a la función TF, ya que solo ocurrirán cuando se rastree la función.
- Puede empaquetar código Python arbitrario en una operación `tf.py_function()`, pero hacerlo obstaculizará el rendimiento, ya que TensorFlow no podrá realizar ninguna optimización gráfica en este código. También reducirá la portabilidad, ya que el gráfico solo se ejecutará en plataformas donde Python esté disponible (y donde estén instaladas las bibliotecas adecuadas).
- Puedes llamar a otras funciones de Python o funciones TF, pero deben seguir las mismas reglas, ya que TensorFlow capturará sus operaciones en el gráfico de cálculo. Tenga en cuenta que estas otras funciones no necesitan estar decoradas con `@tf.function`.
- Si la función crea una variable de TensorFlow (o cualquier otro objeto de TensorFlow con estado, como un conjunto de datos o una cola), debe hacerlo en la primera llamada, y solo entonces, o de lo contrario obtendrá una excepción. Generalmente es preferible crear variables fuera de la función TF (por ejemplo, en el método `build()` de una capa personalizada). Si desea asignar un nuevo valor a la variable, asegúrese de llamar a su método `asignar()` en lugar de utilizar el operador `=`.
- El código fuente de su función Python debería estar disponible para TensorFlow. Si el código fuente no está disponible (por ejemplo, si define su función en el shell de Python, que no proporciona

acceso al código fuente, o si implementa solo los archivos Python \*.pyc compilados en producción), entonces el proceso de generación de gráficos fallará o tendrá una funcionalidad limitada.

- TensorFlow solo capturará bucles for que iteren sobre un tensor o un tf.data.Dataset (consulte [el Capítulo 13](#)). Por lo tanto, asegúrese de utilizar for i en tf.range(x) en lugar de for i en range(x), de lo contrario el bucle no se capturará en el gráfico.  
En cambio, se ejecutará durante el seguimiento. (Esto puede ser lo que desea si el bucle for está destinado a construir el gráfico; por ejemplo, para crear cada capa en una red neuronal).
- Como siempre, por razones de rendimiento, siempre que puedas deberías preferir una implementación vectorizada, en lugar de utilizar bucles.

¡Es hora de resumir! En este capítulo comenzamos con una breve descripción general de TensorFlow, luego analizamos la API de bajo nivel de TensorFlow, incluidos tensores, operaciones, variables y estructuras de datos especiales. Luego utilizamos estas herramientas para personalizar casi todos los componentes de la API de Keras. Finalmente, vimos cómo las funciones TF pueden mejorar el rendimiento, cómo se generan gráficos usando AutoGraph y rastreo, y qué reglas seguir al escribir funciones TF (si desea abrir el cuadro negro un poco más y explorar los gráficos generados, Encontrará detalles técnicos en el [Apéndice D](#)).

En el próximo capítulo, veremos cómo cargar y preprocesar datos de manera eficiente con TensorFlow.

## Ejercicios

1. ¿Cómo describirías TensorFlow en una oración corta? ¿Cuáles son sus principales características? ¿Puedes nombrar otras bibliotecas populares de aprendizaje profundo?

2. ¿Es TensorFlow un reemplazo directo de NumPy? ¿Cuáles son las principales diferencias entre los dos?
3. ¿Obtienes el mismo resultado con `tf.range(10)` y `tf.constant(np.arange(10))`?
4. ¿Puedes nombrar otras seis estructuras de datos disponibles en TensorFlow?  
¿Más allá de los tensores regulares?
5. Puede definir una función de pérdida personalizada escribiendo una función o creando subclases de la clase `tf.keras.losses.Loss`. ¿Cuándo usarías cada opción?
6. De manera similar, puede definir una métrica personalizada en una función o como una subclase de `tf.keras.metrics.Metric`. ¿Cuándo usarías cada opción?
7. ¿Cuándo deberías crear una capa personalizada versus una personalizada?  
¿modelo?
8. ¿Cuáles son algunos casos de uso que requieren escribir su propio ciclo de entrenamiento personalizado?
9. ¿Pueden los componentes personalizados de Keras contener código Python arbitrario?  
¿O deben ser convertibles a funciones TF?
10. ¿Cuáles son las principales reglas a respetar si desea que una función sea convertible en una función TF?
11. ¿Cuándo necesitarías crear un modelo Keras dinámico? ¿Cómo haces eso? ¿Por qué no hacer que todos tus modelos sean dinámicos?
12. Implemente una capa personalizada que realice la normalización de capas (usaremos este tipo de capa en [el Capítulo 15](#)):
  - a. El método `build()` debe definir dos pesos entrenables  $\alpha$  y  $\beta$ , ambos de forma `input_shape[-1:]` y tipo de datos `tf.float32`.  $\alpha$  debe inicializarse con 1 y  $\beta$  con 0.

b. El método `call()` debe calcular la media  $\mu$  y desviación estándar  $\sigma$  de las características de cada instancia. Para esto, puede usar `tf.nn.moments(inputs, axes=-1, keepdims=True)`, que devuelve la media  $\mu$  y la varianza  $\sigma^2$  de todas las instancias (calcule la raíz cuadrada<sup>2</sup> de la varianza para obtener la desviación estándar). Entonces la función debe calcular y devolver  $\alpha \cdot (X - \mu) / (\sigma + \epsilon) + \beta$ , donde  $\alpha$  representa la multiplicación por elementos (\*) y  $\epsilon$  es un término de suavizado (una pequeña constante para evitar la división por cero, por ejemplo, 0,001 ).

C. Asegúrese de que su capa personalizada produzca el mismo (o muy casi lo mismo) salida que la capa `tf.keras.layers.LayerNormalization`.

13. Entrene a un modelo usando un ciclo de entrenamiento personalizado para abordar la Moda.

Conjunto de datos MNIST (ver [Capítulo 10](#)):

- a. Muestra la época, la iteración, la pérdida media de entrenamiento y la precisión media de cada época (actualizada en cada iteración), así como la pérdida de validación y la precisión al final de cada época.
- b. Intente utilizar un optimizador diferente con una tasa de aprendizaje diferente para las capas superiores y las capas inferiores.

Las soluciones a estos ejercicios están disponibles al final del cuaderno de este capítulo, en <https://homl.info/colab3>.

<sup>1</sup> Sin embargo, la biblioteca PyTorch de Facebook es actualmente más popular en el mundo académico: más artículos citan PyTorch que TensorFlow o Keras. Además, la biblioteca JAX de Google está ganando impulso, especialmente en el mundo académico.

<sup>2</sup> TensorFlow incluye otra API de aprendizaje profundo llamada API de estimadores, pero ahora está en desuso.

<sup>3</sup> Si alguna vez lo necesita (pero probablemente no lo hará), puede escribir sus propias operaciones utilizando la API de C++.

- 4** Para obtener más información sobre las TPU y cómo funcionan, consulte <https://homl.info/tpus>.
- 5** Una excepción notable es `tf.math.log()`, que se usa comúnmente pero no tiene un alias `tf.log()`, ya que podría confundirse con el registro.
- 6** No sería una buena idea utilizar una media ponderada: si lo hiciera, entonces dos instancias con el mismo peso pero en diferentes lotes tendrían un impacto diferente en el entrenamiento, dependiendo del peso total de cada lote.
- 7** La sintaxis `{**x, [...]}` se agregó en Python 3.5, para fusionar todos los pares clave/valor del diccionario `x` a otro diccionario. Desde Python 3.9, puedes usar el mejor `x | y` en su lugar, la sintaxis `y` (donde `x` y `y` son dos diccionarios).
- 8** Sin embargo, la pérdida de Huber rara vez se utiliza como métrica: el MAE o MSE es generalmente preferido.
- 9** Esta clase es sólo para fines ilustrativos. Una más simple y mejor la implementación simplemente subclasicaría la clase `tf.keras.metrics.Mean`; consulte la sección "Métricas de transmisión" del cuaderno de este capítulo para ver un ejemplo.
- 10** La API de Keras llama a este argumento `forma_entrada`, pero como también incluye la dimensión del lote, prefiero llamarlo `forma_entrada_lote`.
- 11** El nombre "API de subclases" en Keras normalmente se refiere sólo a la creación de modelos personalizados mediante subclases, aunque se pueden crear muchas otras cosas mediante subclases, como has visto en este capítulo.
- 12** Debido al problema #46858 de TensorFlow, la llamada a `super().build()` puede fallar en este caso, a menos que el problema se haya solucionado en el momento de leer esto. De lo contrario, debe reemplazar esta línea con `self.built = True`.
- 13** También puede llamar a `add_loss()` en cualquier capa dentro del modelo, ya que el modelo recopila de forma recursiva las pérdidas de todas sus capas.
- 14** Si la cinta sale del alcance, por ejemplo cuando regresa la función que la usó, el recolector de basura de Python la eliminará por usted.
- 15** Con la excepción de los optimizadores, ya que muy pocas personas los personalizan; consulte la sección "Optimizadores personalizados" en el cuaderno para ver un ejemplo.
- 16** Sin embargo, en este ejemplo trivial, el gráfico de cálculo es tan pequeño que no hay nada que optimizar, por lo que `tf_cube()` en realidad se ejecuta mucho más lento que `cube()`.

# Capítulo 13. Carga y preprocesamiento de datos con TensorFlow

---

En [el Capítulo 2](#), vio que cargar y preprocesar datos es una parte importante de cualquier proyecto de aprendizaje automático. Utilizó Pandas para cargar y explorar el conjunto de datos de vivienda de California (modificado), que estaba almacenado en un archivo CSV, y aplicó los transformadores de Scikit-Learn para el preprocesamiento. Estas herramientas son bastante convenientes y probablemente las utilizará con frecuencia, especialmente cuando explore y experimente con datos.

Sin embargo, al entrenar modelos de TensorFlow en grandes conjuntos de datos, es posible que prefiera utilizar la API de preprocesamiento y carga de datos propia de TensorFlow, llamada `tf.data`. Es capaz de cargar y preprocesar datos de manera extremadamente eficiente, leer desde múltiples archivos en paralelo usando subprocessos múltiples y colas, barajar y agrupar muestras, y más. Además, puede hacer todo esto sobre la marcha: carga y preprocesa el siguiente lote de datos en múltiples núcleos de CPU, mientras sus GPU o TPU están ocupadas entrenando el lote de datos actual.

La API `tf.data` le permite manejar conjuntos de datos que no caben en la memoria y le permite aprovechar al máximo sus recursos de hardware, acelerando así el entrenamiento. Disponible, la API `tf.data` puede leer desde archivos de texto (como archivos CSV), archivos binarios con registros de tamaño fijo y archivos binarios que usan el formato TFRecord de TensorFlow, que admite registros de diferentes tamaños.

TFRecord es un formato binario flexible y eficiente que generalmente contiene buffers de protocolo (un formato binario de código abierto). La API `tf.data` también admite la lectura de bases de datos SQL. Además, hay muchas extensiones de código abierto disponibles para leer todo tipo de datos.

fuentes, como el servicio BigQuery de Google (consulte <https://tensorflow.org/io>).

Keras también viene con capas de preprocessamiento potentes pero fáciles de usar que pueden integrarse en sus modelos: de esta manera, cuando implemente un modelo en producción, podrá ingerir datos sin procesar directamente, sin que usted tenga que agregar ningún preprocessamiento adicional. código. Esto elimina el riesgo de que no coincidan entre el código de preprocessamiento utilizado durante el entrenamiento y el código de preprocessamiento usado en producción, lo que probablemente causaría un sesgo en el entrenamiento/servicio. Y si implementa su modelo en varias aplicaciones codificadas en diferentes lenguajes de programación, no tendrá que volver a implementar el mismo código de preprocessamiento varias veces, lo que también reduce el riesgo de discrepancias.

Como verá, ambas API se pueden utilizar conjuntamente, por ejemplo, para beneficiarse de la carga de datos eficiente que ofrece tf.data y la conveniencia de las capas de preprocessamiento de Keras.

En este capítulo, primero cubriremos la API tf.data y el formato TFRecord. Luego exploraremos las capas de preprocessamiento de Keras y cómo usarlas con la API tf.data. Por último, echaremos un vistazo rápido a algunas bibliotecas relacionadas que pueden resultarle útiles para cargar y preprocessar datos, como TensorFlow Datasets y TensorFlow Hub. ¡Entonces empecemos!

## La API tf.data

Toda la API tf.data gira en torno al concepto de tf.data.Dataset: esto representa una secuencia de elementos de datos.

Por lo general, usará conjuntos de datos que leen datos gradualmente del disco, pero para simplificar, creemos un conjunto de datos a partir de un tensor de datos simple usando tf.data.Dataset.from\_tensor\_slices():

```
>>> import tensorflow como tf >>> X  
= tf.range(10) # cualquier tensor de datos
```

```
>>> conjunto de datos = tf.data.Dataset.from_tensor_slices(X) >>>
conjunto de datos
<Formas TensorSliceDataset: (), tipos: tf.int32>
```

La función `from_tensor_slices()` toma un tensor y crea un `tf.data.Dataset` cuyos elementos son todos los sectores de X a lo largo de la primera dimensión, por lo que este conjunto de datos contiene 10 elementos: tensores 0, 1, 2, 9. En este ..., caso, hemos obtenido el mismo conjunto de datos si había usado `tf.data.Dataset.range(10)` (excepto que los elementos serían enteros de 64 bits en lugar de enteros de 32 bits).

Simplemente puedes iterar sobre los elementos de un conjunto de datos de esta manera:

```
>>> para el elemento del conjunto de datos:
...
      imprimir (artículo)
...
tf.Tensor(0, forma=(), dtype=int32) tf.Tensor(1,
forma=(), dtype=int32) [...] tf.Tensor(9, forma=(),
dtype=int32)
```

### NOTA

La API `tf.data` es una API de transmisión: puede iterar de manera muy eficiente a través de los elementos de un conjunto de datos, pero la API no está diseñada para indexar o dividir.

Un conjunto de datos también puede contener tuplas de tensores, o diccionarios de pares nombre/tensor, o incluso tuplas y diccionarios de tensores anidados. Al dividir una tupla, un diccionario o una estructura anidada, el conjunto de datos solo dividirá los tensores que contiene, preservando al mismo tiempo la estructura de tupla/diccionario. Por ejemplo:

```
>>> X_nested = {"a": ([1, 2, 3], [4, 5, 6]), "b": [7, 8, 9]} >>> conjunto de datos =
tf.data.Dataset .from_tensor_slices(X_nested) >>> para el elemento del conjunto de
datos:
```

```

...
    imprimir (artículo)
...
{'a': (<tf.Tensor: [...]=1>, <tf.Tensor: [...]=4>), 'b': <tf.Tensor: [...]= 7>} {'a': (<tf.Tensor:
[...]=2>, <tf.Tensor: [...]=5>), 'b':
<tf.Tensor: [...]=8>} {'a': (<tf.Tensor: [...]=3>, <tf.Tensor: [...]=6>), 'b': <tf.Tensor: [...]=9>}

```

## Encadenamiento de transformaciones

Una vez que tenga un conjunto de datos, puede aplicarle todo tipo de transformaciones llamando a sus métodos de transformación. Cada método devuelve un nuevo conjunto de datos, por lo que puede encadenar transformaciones como esta (esta cadena se ilustra en [la Figura 13-1](#)):

```

>>> conjunto de datos =
tf.data.Dataset.from_tensor_slices(tf.range(10)) >>> conjunto de
datos = dataset.repeat(3).batch(7) >>> para el elemento
del conjunto de datos:
...
    imprimir (artículo)
...
tf.Tensor([0 1 2 3 4 5 6], forma=(7,), dtype=int32) tf.Tensor([7 8 9 0 1 2 3],
forma=(7,), dtype=int32) tf.Tensor([4 5 6 7 8 9 0], forma=(7,), dtype=int32)
tf.Tensor([1 2 3 4 5 6 7], forma=(7,), dtype=int32) tf.Tensor([8 9], forma=(2,),
dtype=int32)

```

En este ejemplo, primero llamamos al método `repetir()` en el conjunto de datos original y devuelve un nuevo conjunto de datos que repite los elementos del conjunto de datos original tres veces. ¡Por supuesto, esto no copiará todos los datos en la memoria tres veces! Si llama a este método sin argumentos, el nuevo conjunto de datos repetirá el conjunto de datos de origen para siempre, por lo que el código que itera sobre el conjunto de datos tendrá que decidir cuándo detenerse.

Luego llamamos al método `lote()` en este nuevo conjunto de datos y nuevamente esto crea un nuevo conjunto de datos. Éste agrupará los elementos del conjunto de datos anterior en lotes de siete elementos.

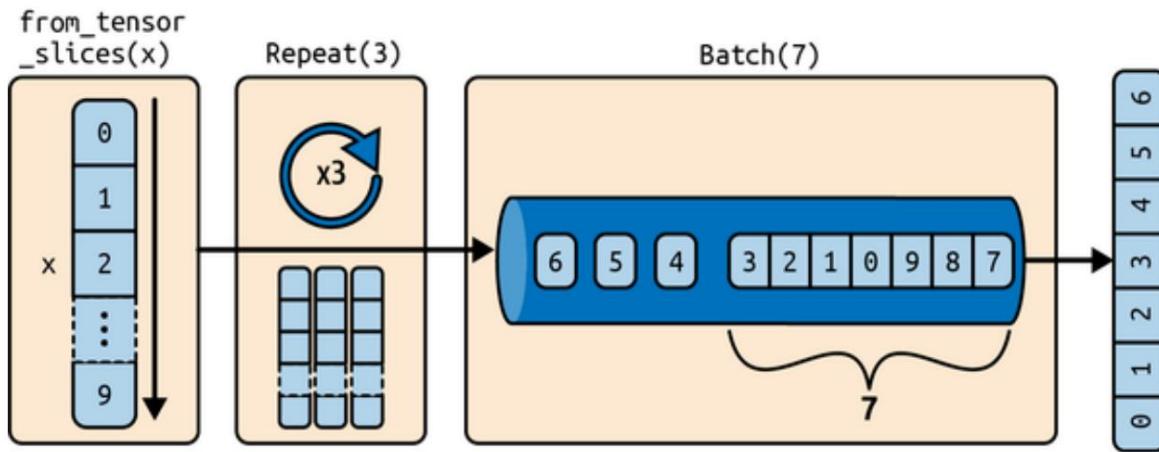


Figura 13-1. Encadenamiento de transformaciones de conjuntos de datos

Finalmente, iteramos sobre los elementos de este conjunto de datos final. El método `lote()` tenía que generar un lote final de tamaño dos en lugar de siete, pero puede llamar a `lote()` con `drop_remainder=True` si desea que elimine este lote final, de modo que todos los lotes tengan exactamente el mismo tamaño.

#### ADVERTENCIA

Los métodos de conjuntos de datos no modifican los conjuntos de datos: crean otros nuevos. Así que asegúrese de mantener una referencia a estos nuevos conjuntos de datos (por ejemplo, con `dataset = ...`), o de lo contrario no sucederá nada.

También puedes transformar los elementos llamando al método `map()`. Por ejemplo, esto crea un nuevo conjunto de datos con todos los lotes multiplicados por dos:

```
>>> conjunto de datos = conjunto de datos.map(lambda x: x * 2) # x es un lote
>>> para el elemento del conjunto de datos:
...     imprimir (artículo)
...
tf.Tensor([ 0  2  4  6  8 10 12], forma=(7,), dtype=int32) tf.Tensor([14 16 18  0  2  4  6], forma=(7,), dtype=int32) [...]
```

Este método `map()` es al que llamará para aplicar cualquier preprocessamiento a sus datos. A veces, esto incluirá cálculos que pueden ser bastante intensivos, como remodelar o rotar una imagen, por lo que normalmente querrás generar varios subprocessos para acelerar las cosas. Esto se puede hacer estableciendo el argumento `num_parallel_calls` en la cantidad de subprocessos a ejecutar o en `tf.data.AUTOTUNE`. Tenga en cuenta que la función que pase al método `map()` debe poder convertirse en una función TF (consulte [el Capítulo 12](#)).

También es posible simplemente filtrar el conjunto de datos utilizando el método `filter()`. Por ejemplo, este código crea un conjunto de datos que solo contiene los lotes cuya suma es mayor que 50:

```
>>> conjunto de datos = conjunto de datos.filter(lambda x: tf.reduce_sum(x) > 50) >>>
para el elemento del conjunto de datos:
...
    imprimir (artículo)
...
tf.Tensor([14 16 18 0 2 4 6], forma=(7,), dtype=int32) tf.Tensor([ 8 10 12 14 16 18 0],
forma=(7,), dtype=int32) tf.Tensor([ 2 4 6 8 10 12 14], forma=(7,), dtype=int32)
```

A menudo querrás ver sólo unos pocos elementos de un conjunto de datos. Puedes usar el método `take()` para eso:

```
>>> para el elemento en dataset.take(2):
...
    imprimir(elemento)
...
tf.Tensor([14 16 18 0 2 4 6], forma=(7,), dtype=int32) tf.Tensor([ 8 10 12 14 16 18 0],
forma=(7,), dtype=int32)
```

## Mezclando los datos

Como analizamos en [el Capítulo 4](#), el descenso de gradiente funciona mejor cuando las instancias del conjunto de entrenamiento son independientes y están distribuidas de manera idéntica (IID). Una forma sencilla de garantizar esto es barajar el

instancias, utilizando el método `shuffle()`. Creará un nuevo conjunto de datos que comenzará llenando un búfer con los primeros elementos del conjunto de datos de origen. Luego, cada vez que se le solicita un elemento, extraerá uno aleatoriamente del búfer y lo reemplazará con uno nuevo del conjunto de datos de origen, hasta que haya iterado por completo a través del conjunto de datos de origen. En este punto, continuará extrayendo elementos aleatoriamente del búfer hasta que esté vacío. Debe especificar el tamaño del búfer y es importante que sea lo suficientemente grande; de lo contrario, la mezcla no <sup>1</sup> será muy efectiva. Simplemente no excedas la cantidad de RAM que tienes, aunque incluso si tienes mucha, no es necesario ir más allá del tamaño del conjunto de datos. Puede proporcionar una semilla aleatoria si desea el mismo orden aleatorio cada vez que ejecute su programa. Por ejemplo, el siguiente código crea y muestra un conjunto de datos que contiene los números enteros del 0 al 9, repetido dos veces, mezclado usando un búfer de tamaño 4 y una semilla aleatoria de 42, y agrupado con un tamaño de lote de 7:

```
>>> conjunto de datos = tf.data.Dataset.range(10).repeat(2) >>>
conjunto de datos = dataset.shuffle(buffer_size=4,
semilla=42).batch(7)
>>> para el elemento del conjunto de datos:
...     imprimir (artículo)
...
tf.Tensor([3 0 1 6 2 5 7], forma=(7,), dtype=int64) tf.Tensor([8 4 1 9 4 2 3],
forma=(7,), dtype=int64) tf.Tensor([7 5 0 8 9 6], forma=(6,), dtype=int64)
```

#### CONSEJO

Si llama a `repetir()` en un conjunto de datos mezclado, de forma predeterminada generará un nuevo orden en cada iteración. En general, esto es una buena idea, pero si prefiere reutilizar el mismo orden en cada iteración (por ejemplo, para pruebas o depuración), puede configurar `reshuffle_each_iteration=False` al llamar a `shuffle()`.

Para un conjunto de datos grande que no cabe en la memoria, este método simple de mezcla de búfer puede no ser suficiente, ya que el búfer será pequeño en comparación con el conjunto de datos. Una solución es mezclar los datos de origen (por ejemplo, en Linux puede mezclar archivos de texto usando el comando `shuf`). ¡Esto definitivamente mejorará mucho la mezcla! Incluso si los datos de origen están mezclados, normalmente querrás hacerlo un poco más, o de lo contrario se repetirá el mismo orden en cada época, y el modelo puede terminar siendo sesgado (por ejemplo, debido a algunos patrones espurios presentes por casualidad en el orden de los datos de origen). Para mezclar un poco más las instancias, un enfoque común es dividir los datos de origen en varios archivos y luego leerlos en orden aleatorio durante el entrenamiento.

Sin embargo, las instancias ubicadas en el mismo archivo seguirán estando cerca unas de otras. Para evitar esto puedes seleccionar varios archivos al azar y leerlos simultáneamente, intercalando sus registros. Luego, además de eso, puede agregar un búfer de reproducción aleatoria utilizando el método `shuffle()`. Si esto le parece mucho trabajo, no se preocupe: la API `tf.data` hace que todo esto sea posible en tan solo unas pocas líneas de código. Repasemos cómo puedes hacer esto.

## Intercalando líneas de múltiples archivos

Primero, supongamos que cargó el conjunto de datos de vivienda de California, lo barajó (a menos que ya estuviera barajado) y lo dividió en un conjunto de entrenamiento, un conjunto de validación y un conjunto de prueba. Luego, divide cada conjunto en muchos archivos CSV que se ven así (cada fila contiene ocho características de entrada más el valor medio objetivo de la casa):

```
MedInc,HouseAge,AveRooms,AveBedrms,Popul...,AveOccup,Lat...,Lon  
g...,MedianHouseValue  
3.5214,15.0,3.050,1.107,1447.0,1.606,37.63,-122.43,1.442 5.3275,5.0,6.490,0.  
991,3464.0,3.443 ,33.69,-117.39,1.687 3.1,29.0,7.542,1.592,1328.0,2.251,38.44,-122.98,1.621  
[...]
```

Supongamos también que `train_filepaths` contiene la lista de rutas de archivos de entrenamiento (y también tiene `valid_filepaths` y `test_filepaths`):

```
>>> train_filepaths
['conjuntos de datos/vivienda/mi_tren_00.csv',
 'conjuntos de datos/vivienda/mi_tren_01.csv', ...]
```

Alternativamente, puedes usar patrones de archivos; por ejemplo, `train_filepaths = "conjuntos de datos/vivienda/mi_tren_*.csv"`. Ahora creamos un conjunto de datos que contenga solo estas rutas de archivo:

```
filepath_dataset =
tf.data.Dataset.list_files(train_filepaths, seed=42)
```

De forma predeterminada, la función `list_files()` devuelve un conjunto de datos que mezcla las rutas de los archivos. En general, esto es algo bueno, pero puedes configurar `shuffle=False` si no lo deseas por algún motivo.

A continuación, puede llamar al método `interleave()` para leer cinco archivos a la vez e intercalar sus líneas. También puedes omitir la primera línea de cada archivo (que es la fila del encabezado) usando el método `skip()`:

```
n_readers = 5
conjunto de datos = filepath_dataset.interleave( ruta
    de archivo lambda :
    tf.data.TextLineDataset(ruta de archivo).skip(1),
    longitud_ciclo=n_lecadores)
```

El método `interleave()` creará un conjunto de datos que extraerá cinco rutas de archivo del `filepath_dataset`, y para cada una llamará a la función que le asignó (una lambda en este ejemplo) para crear un nuevo conjunto de datos (en este caso, un `TextLineDataset`). Para ser claros, en esta etapa habrá siete conjuntos de datos en total: el conjunto de datos de ruta de archivo, el conjunto de datos intercalado y los cinco conjuntos de datos de línea de texto creados.

internamente por el conjunto de datos entrelazados. Cuando itera sobre el conjunto de datos entrelazados, recorrerá estos cinco TextLineDatasets, leyendo una línea a la vez de cada uno hasta que todos los conjuntos de datos se queden sin elementos. Luego, buscará las siguientes cinco rutas de archivo del filepath\_dataset y las intercalará de la misma manera, y así sucesivamente hasta que se quede sin rutas de archivo. Para que el intercalado funcione mejor, es preferible tener archivos de idéntica longitud; de lo contrario, el final del archivo más largo no se intercalará.

De forma predeterminada, `interleave()` no utiliza paralelismo; simplemente lee una línea a la vez de cada archivo, de forma secuencial. Si desea que realmente lea archivos en paralelo, puede configurar el argumento `num_parallel_calls` del método `interleave()` en el número de subprocessos que desee (recuerde que el método `map()` también tiene este argumento). Incluso puede configurarlo en `tf.data.AUTOTUNE` para que TensorFlow elija dinámicamente la cantidad correcta de subprocessos según la CPU disponible. Veamos ahora lo que contiene el conjunto de datos:

```
>>> para línea en dataset.take(5): imprimir(línea)
...
...
tf.Tensor(b'4.5909,16.0,[...],33.63,-117.71,2.418', forma=(), dtype=cadena)
tf.Tensor(b'2.4792,24.0,
[...],34.18 ,-118.38,2.0', forma=(), dtype=cadena) tf.Tensor(b'4.2708,45.0,
[...],37.48,-122.19,2.67', forma=(), dtype=cadena) tf .Tensor(b'2.1856,41.0,
[...],32.76,-117.12,1.205',
forma=(), dtype=cadena) tf.Tensor(b'4.1812,52.0[...],33.73, -118.31,3.215', forma=(),
tipod=cadena)
```

Estas son las primeras filas (ignorando la fila del encabezado) de cinco archivos CSV, elegidos al azar. ¡Se ve bien!

## NOTA

Es posible pasar una lista de rutas de archivos al constructor `TextLineDataset`: revisará cada archivo en orden, línea por línea. Si también establece el argumento `num_parallel_reads` en un número mayor que uno, entonces el conjunto de datos leerá esa cantidad de archivos en paralelo e intercalarán sus líneas (sin tener que llamar al método `interleave()`). Sin embargo, no mezclará los archivos ni se saltará las líneas del encabezado.

## Preprocesamiento de los datos

Ahora que tenemos un conjunto de datos de alojamiento que devuelve cada instancia como un tensor que contiene una cadena de bytes, necesitamos realizar un poco de preprocesamiento, incluido el análisis de las cadenas y el escalado de los datos.

Implementemos un par de funciones personalizadas que realizarán este preprocesamiento:

```
X_mean, X_std = [...] # media y escala de cada característica en el conjunto de
entrenamiento
n_inputs = 8

def parse_csv_line(línea): defs =
    [0.] * n_inputs + [tf.constant([], dtype=tf.float32)] campos
= tf.io.decode_csv(línea,
    record_defaults=defs) return tf.stack(campos[:-1]), tf.stack(campos[-1:])

def proceso(línea): x, y =
    parse_csv_line(línea) return (x -
    X_mean) / X_std, y
```

Repasemos este código:

- Primero, el código supone que hemos calculado previamente la media y la desviación estándar de cada característica en el conjunto de entrenamiento. `X_mean` y `X_std` son solo tensores 1D (o matrices NumPy) que contienen ocho flotantes, uno por característica de entrada. Esto puede hacerse

utilizando un Scikit-Learn StandardScaler en una muestra aleatoria suficientemente grande del conjunto de datos. Más adelante en este capítulo, usaremos una capa de preprocesamiento de Keras.

- La función `parse_csv_line()` toma una línea CSV y la analiza. Para ayudar con eso, utiliza la función `tf.io.decode_csv()`, que toma dos argumentos: el primero es la línea a analizar y el segundo es una matriz que contiene el valor predeterminado para cada columna en el archivo CSV. Esta matriz (`defs`) le dice a TensorFlow no solo el valor predeterminado para cada columna, sino también el número de columnas y sus tipos. En este ejemplo, le decimos que todas las columnas de características son flotantes y que los valores faltantes deben ser cero por defecto, pero proporcionamos una matriz vacía de tipo `tf.float32` como valor predeterminado para la última columna (el destino): la matriz le dice a TensorFlow que esta columna contiene flotantes, pero que no hay un valor predeterminado, por lo que generará una excepción si encuentra un valor faltante.
- La función `tf.io.decode_csv()` devuelve una lista de tensores escalares (uno por columna), pero necesitamos devolver una matriz de tensores 1D. Entonces llamamos a `tf.stack()` en todos los tensores excepto en el último (el objetivo): esto apilará estos tensores en una matriz 1D. Luego hacemos lo mismo con el valor objetivo: esto lo convierte en una matriz tensorial 1D con un valor único, en lugar de un tensor escalar. La función `tf.io.decode_csv()` está completa, por lo que devuelve las características de entrada y el objetivo.
- Finalmente, la función `preprocess()` personalizada simplemente llama a la función `parse_csv_line()`, escala las características de entrada restando las medias de las características y luego dividiéndolas por las desviaciones estándar de las características, y devuelve una tupla que contiene las características escaladas y el objetivo.

Probemos esta función de preprocesamiento:

```
>>>
preproceso(b'4.2083,44.0,5.3232,0.9171,846.0,2.3370,37.47,- 122.2,2.782') (<tf.Tensor:
forma=(8,),
dtype=float32, numpy= array([ 0.16579159, 1.216324, -0.05204564,
-0.39215982,
-0.5277444 ,
-0.2633488 , 0.8543046 , -1.3072058],
dtype=float32>,
<tf.Tensor: forma=(1,), dtype=float32,
numpy=matriz([2.782], dtype=float32)>)
```

¡Se ve bien! La función preprocess() puede convertir una instancia de una cadena de bytes a un tensor escalado agradable, con su etiqueta correspondiente. Ahora podemos usar el método map() del conjunto de datos para aplicar la función preprocess() a cada muestra del conjunto de datos.

## Poniendo todo junto

Para que el código sea más reutilizable, juntemos todo lo que hemos discutido hasta ahora en otra función auxiliar; creará y devolverá un conjunto de datos que cargará eficientemente datos de vivienda de California desde múltiples archivos CSV, los procesará previamente, los mezclará y los agrupará (consulte Figura 13-2):

```
def csv_reader_dataset(rutas de archivos, n_readers=5,
n_read_threads=Ninguno,
n_parse_threads=5,
shuffle_buffer_size=10_000, semilla=42, lote_size=32):
    conjunto de datos =
        tf.data.Dataset.list_files(rutas de archivos,
semilla=semilla)
    conjunto de datos = conjunto de datos.interleave(
        ruta de archivo lambda :
            tf.data.TextLineDataset(filépath).skip(1), Cycle_length=n_readers,
            num_parallel_calls=n_read_threads)

    conjunto de datos = conjunto de datos.map(preproceso,
num_parallel_calls=n_parse_threads) conjunto de
    datos = conjunto de datos.shuffle(shuffle_buffer_size,
```

```
semilla=semilla)
    devuelve conjunto de datos.batch(batch_size).prefetch(1)
```

Tenga en cuenta que utilizamos el método prefetch() en la última línea. Esto es importante para el rendimiento, como verá ahora.

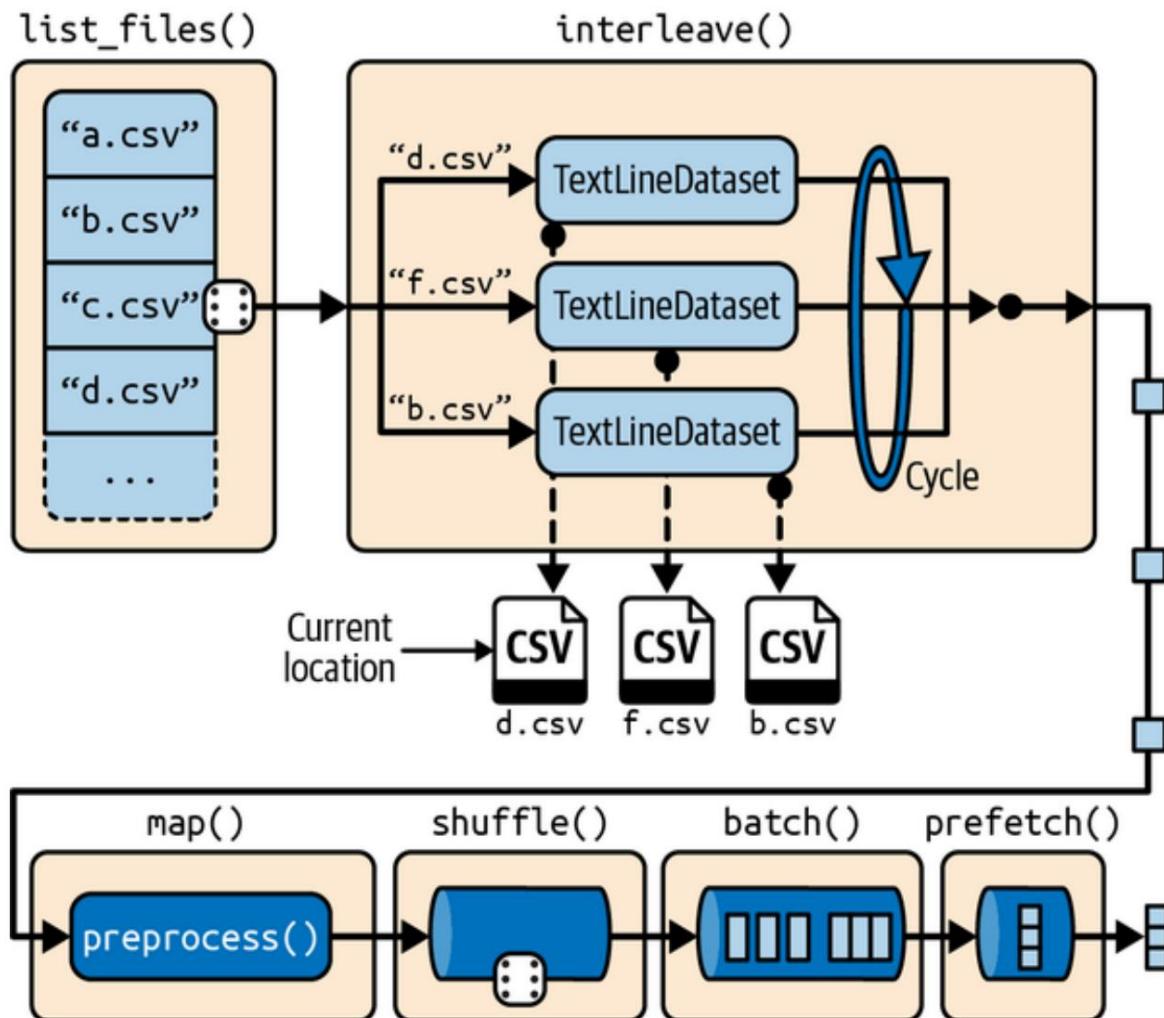


Figura 13-2. Carga y preprocesamiento de datos de múltiples archivos CSV

## Precarga

Al llamar a prefetch(1) al final de la función personalizada csv\_reader\_dataset(), estamos creando un conjunto de datos que hará todo lo posible para estar siempre un lote por delante. En otras palabras, mientras nuestro algoritmo de entrenamiento funciona en un lote, el conjunto de datos

Ya estamos trabajando en paralelo para preparar el siguiente lote (por ejemplo, leyendo los datos del disco y preprocesándolos). Esto puede mejorar drásticamente el rendimiento, como se ilustra en [la Figura 13-3](#).

Si también nos aseguramos de que la carga y el preprocesamiento sean multiproceso (al configurar `num_parallel_calls` al llamar a `interleave()` y `map()`), podemos explotar múltiples núcleos de CPU y, con suerte, hacer que la preparación de un lote de datos sea más corta que ejecutar un paso de entrenamiento en la GPU: esto de esta manera, la GPU se utilizará casi al 100% (excepto el tiempo de transferencia de datos de la CPU <sup>3</sup> a la GPU) y el entrenamiento se ejecutará mucho más rápido.

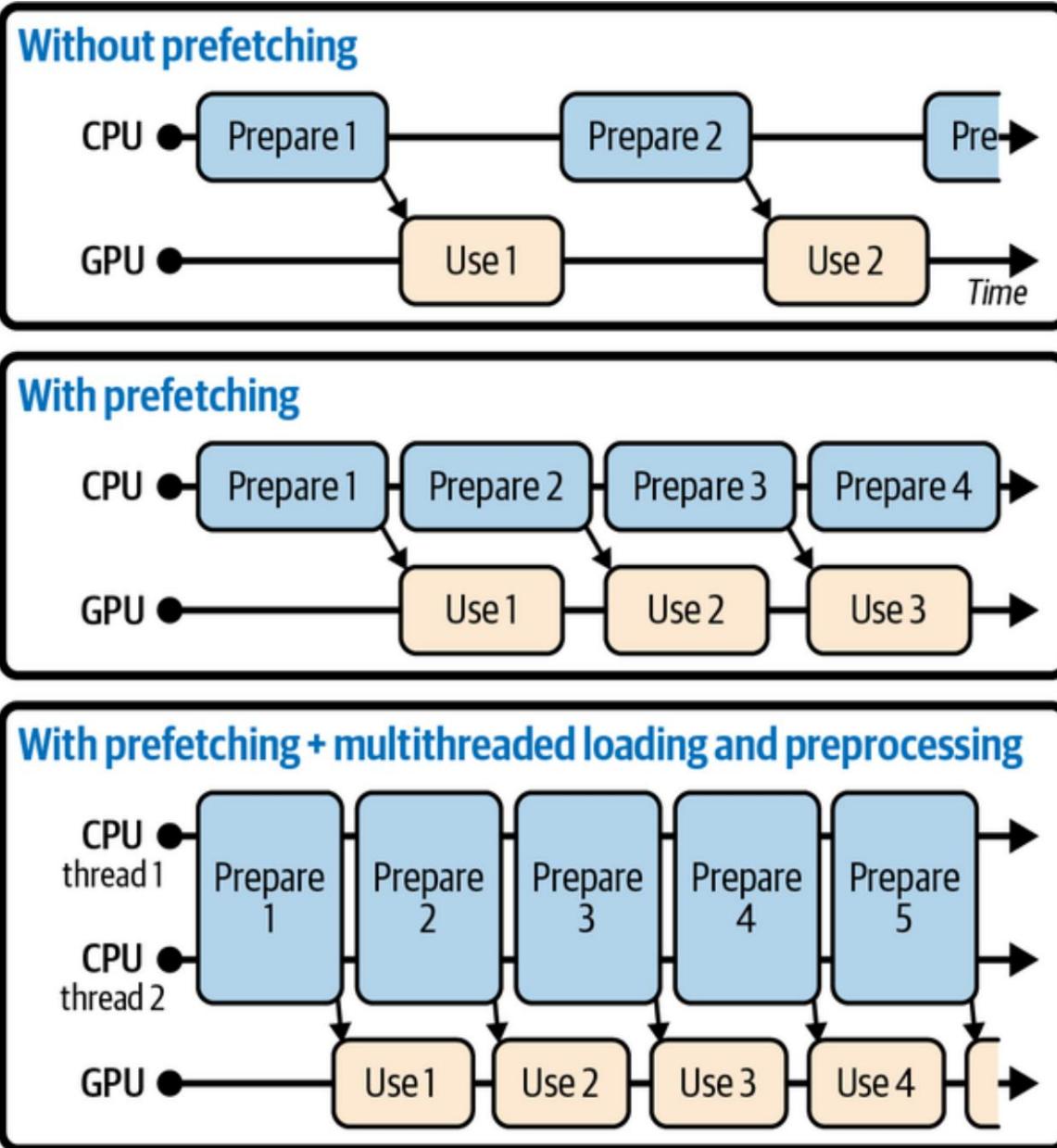


Figura 13-3. Con la captación previa, la CPU y la GPU funcionan en paralelo: mientras la GPU trabaja en un lote, la CPU trabaja en el siguiente.

## CONSEJO

Si planea comprar una tarjeta GPU, su potencia de procesamiento y su tamaño de memoria son, por supuesto, muy importantes (en particular, una gran cantidad de RAM es crucial para grandes modelos de procesamiento de lenguaje natural o visión por computadora). Igual de importante para un buen rendimiento es el ancho de banda de la memoria de la GPU; esta es la cantidad de gigabytes de datos que puede entrar o salir de su RAM por segundo.

Si el conjunto de datos es lo suficientemente pequeño como para caber en la memoria, puede acelerar significativamente el entrenamiento utilizando el método `cache()` del conjunto de datos para almacenar en caché su contenido en la RAM. Por lo general, debe hacer esto después de cargar y preprocesar los datos, pero antes de mezclarlos, repetirlos, agruparlos y buscarlos previamente. De esta manera, cada instancia solo se leerá y preprocesará una vez (en lugar de una vez por época), pero los datos se seguirán mezclando de forma diferente en cada época y el siguiente lote se seguirá preparando con antelación.

Ahora ha aprendido cómo crear canales de entrada eficientes para cargar y preprocesar datos de múltiples archivos de texto. Hemos analizado los métodos de conjuntos de datos más comunes, pero hay algunos más que quizás quieras ver, como `concatenate()`, `zip()`, `window()`, `reduce()`, `shard()`, `flat_map()`, `apply( )`, `desbatch()` y `acolchado_batch()`. También hay algunos métodos de clase más, como `from_generator()` y `from_tensors()`, que crean un nuevo conjunto de datos a partir de un generador de Python o una lista de tensores, respectivamente. Consulte la documentación de la API para obtener más detalles.

También tenga en cuenta que hay funciones experimentales disponibles en `tf.data.experimental`, muchas de las cuales probablemente llegarán a la API principal en futuras versiones (por ejemplo, consulte la clase `CsvDataset`, así como el método `make_csv_dataset()`, que se encarga de inferir el tipo de cada columna).

## Usando el conjunto de datos con Keras

Ahora podemos usar la función personalizada `csv_reader_dataset()` que escribimos anteriormente para crear un conjunto de datos para el conjunto de entrenamiento, y para el conjunto de validación y el conjunto de prueba. El conjunto de entrenamiento se barajará en cada época (tenga en cuenta que el conjunto de validación y el conjunto de prueba también se barajarán, aunque realmente no los necesitamos):

```
train_set = csv_reader_dataset(train_filepaths)
valid_set = csv_reader_dataset(valid_filepaths)
test_set = csv_reader_dataset(test_filepaths)
```

Ahora puedes simplemente construir y entrenar un modelo de Keras usando estos conjuntos de datos. Cuando llamas al método `fit()` del modelo, pasas `train_set` en lugar de `X_train`, `y_train`, y pasas `validation_data=valid_set` en lugar de `validation_data=(X_valid, y_valid)`. El método `fit()` se encargará de repetir el conjunto de datos de entrenamiento una vez por época, utilizando un orden aleatorio diferente en cada época:

```
modelo = tf.keras.Sequential([...])
model.compile(loss="mse", optimizer="sgd")
model.fit(train_set, validation_data=valid_set, epochs=5)
```

De manera similar, puedes pasar un conjunto de datos a los métodos `evaluar()` y `predecir()`:

```
test_mse = model.evaluate(test_set)
new_set = test_set.take(3) # finge que tenemos 3 muestras nuevas
y_pred = model.predict(new_set) # o simplemente podrías pasar una matriz NumPy
```

A diferencia de los otros conjuntos, el `new_set` normalmente no contendrá etiquetas. Si es así, como es el caso aquí, Keras los ignorará. Tenga en cuenta que en todos estos casos, aún puede usar matrices NumPy en lugar de conjuntos de datos si lo prefiere (pero, por supuesto, primero deben haberse cargado y preprocessado).

Si desea crear su propio circuito de entrenamiento personalizado (como se explica en [Capítulo 12](#)), puedes simplemente iterar sobre el conjunto de entrenamiento, de forma muy natural:

```
n_epochs = 5
para época en rango (n_epochs): para
    X_batch, y_batch en train_set: [...] # realiza un
        paso de descenso de gradiente
```

De hecho, incluso es posible crear una función TF (ver [Capítulo 12](#)) que entrene el modelo para una época completa. Esto realmente puede acelerar el entrenamiento:

```
@tf.function def
train_one_epoch(modelo, optimizador, loss_fn, train_set): para X_batch, y_batch
    en train_set:
        con tf.GradientTape() como cinta: y_pred =
            modelo(X_batch) main_loss =
                tf.reduce_mean(loss_fn(y_batch,
y_pred))
            pérdida = tf.add_n([main_loss] + modelo.losses) gradientes =
                tape.gradient(pérdida,
                modelo.trainable_variables)
            optimizador.apply_gradients(zip(gradientes,
                modelo.trainable_variables))

        optimizador = tf.keras.optimizers.SGD(learning_rate=0.01) loss_fn =
            tf.keras.losses.mean_squared_error para la época en el rango
            (n_epochs): print("\rEpoch {}/
            {}".format(epoch + 1, n_epochs ), end="")
            train_one_epoch(modelo,
            optimizador, loss_fn, train_set)
```

En Keras, el argumento `steps_per_execution` del método `compile()` le permite definir el número de lotes que procesará el método `fit()` durante cada llamada a la función `tf.que` utiliza para el entrenamiento. El valor predeterminado es sólo 1, por lo que si lo configura en 50, a menudo verá una mejora significativa en el rendimiento. Sin embargo, los métodos `on_batch_*`() de las devoluciones de llamada de Keras solo se llamarán cada 50 lotes.

¡Felicitaciones, ahora sabe cómo crear potentes canales de entrada utilizando la API tf.data! Sin embargo, hasta ahora hemos estado usando archivos CSV, que son comunes, simples y convenientes, pero no realmente eficientes y no admiten muy bien estructuras de datos grandes o complejas (como imágenes o audio). Entonces, veamos cómo usar TFRecords en su lugar.

## CONSEJO

Si está satisfecho con los archivos CSV (o cualquier otro formato que esté utilizando), no es necesario que utilice TFRecords. Como dice el refrán, si no está roto, ¡no lo arregles! Los TFRecords son útiles cuando el cuello de botella durante el entrenamiento es cargar y analizar los datos.

## El formato TFRecord

El formato TFRecord es el formato preferido de TensorFlow para almacenar grandes cantidades de datos y leerlos de manera eficiente. Es un formato binario muy simple que solo contiene una secuencia de registros binarios de diferentes tamaños (cada registro se compone de una longitud, una suma de verificación CRC para verificar que la longitud no esté dañada, luego los datos reales y, finalmente, una suma de verificación CRC para los datos). Puede crear fácilmente un archivo TFRecord usando la clase `tf.io.TFRecordWriter`:

```
con tf.io.TFRecordWriter("my_data.tfrecord") como f: f.write(b"Este es el primer  
registro") f.write(b"Y este es el segundo registro")
```

Y luego puede usar `tf.data.TFRecordDataset` para leer uno o más archivos TFRecord:

```
filepaths = ["my_data.tfrecord"] conjunto de datos =  
tf.data.TFRecordDataset(filepaths) para el elemento del conjunto de  
datos:  
imprimir (artículo)
```

Esto generará:

```
tf.Tensor(b'Este es el primer registro', forma=(), dtype=cadena)
tf.Tensor(b'Y este
es el segundo registro', forma=(), dtype=cadena)
```

#### CONSEJO

De forma predeterminada, un TFRecordDataset leerá los archivos uno por uno, pero puede hacer que lea varios archivos en paralelo e intercalar sus registros pasando al constructor una lista de rutas de archivos y configurando num\_parallel\_reads en un número mayor que uno. Alternativamente, puede obtener el mismo resultado usando list\_files() e interleave() como hicimos antes para leer múltiples archivos CSV.

## Archivos TFRecord comprimidos

A veces puede resultar útil comprimir sus archivos TFRecord, especialmente si deben cargarse a través de una conexión de red. Puede crear un archivo TFRecord comprimido configurando el argumento de opciones:

```
opciones = tf.io.TFRecordOptions(compression_type="GZIP") con
tf.io.TFRecordWriter("my_compressed.tfrecord", opciones) como f:
    f.write(b"Comprimir,
            comprimir, comprimir!")
```

Al leer un archivo TFRecord comprimido, debe especificar el tipo de compresión:

```
conjunto de datos =
tf.data.TFRecordDataset(["my_compressed.tfrecord"],
                        tipo_compresión="GZIP")
```

## Una breve introducción a los buffers de protocolo

Aunque cada registro puede usar cualquier formato binario que desee, los archivos TFRecord generalmente contienen búferes de protocolo serializados (también llamados protobufs). Este es un formato binario portátil, extensible y eficiente desarrollado en Google en 2001 y hecho de código abierto en 2008; Los protobufs ahora se usan ampliamente, en particular en [gRPC](#), El sistema de llamadas a procedimientos remotos de Google. Se definen usando un lenguaje simple que se ve así:

```
sintaxis = "proto3"; mensaje
Persona { nombre de
    cadena = 1; int32
    identificación = 2;
    correo electrónico de cadena repetida = 3;
}
```

Esta definición de protobuf dice que estamos usando la versión 3 del formato protobuf y especifica que cada objeto Persona puede (opcionalmente) tener un nombre de tipo cadena, una identificación de tipo int32 y cero o más campos de correo electrónico, cada uno de tipo cadena. Los números 1, 2 y 3 son los identificadores de campo: se utilizarán en la representación binaria de cada registro. Una vez que tenga una definición en un archivo .proto , puede compilarlo. Esto requiere protoc, el compilador de protobuf, para generar clases de acceso en Python (o algún otro lenguaje). Tenga en cuenta que las definiciones de protobuf que generalmente usará en TensorFlow ya han sido compiladas para usted y sus clases de Python son parte de la biblioteca de TensorFlow, por lo que no necesitará usar protoc. Todo lo que necesitas saber es cómo usar las clases de acceso de protobuf en Python. Para ilustrar los conceptos básicos, veamos un ejemplo simple que utiliza las clases de acceso generadas para el protobuf Person (el código se explica en los comentarios):

```
>>> de person_pb2 importar Persona # importar la clase de acceso generada

>>> persona = Persona(nombre="Al", id=123, correo electrónico=["a@b.com"]) # crear
una Persona >>>
imprimir(persona) # mostrar la Persona
```

```

nombre: "Al"
identificación: 123
correo electrónico:
"a@b.com" >>> persona.nombre # leer un
campo 'Al'
>>> person.name = "Alice" # modificar un campo >>> person.email[0]
# se puede acceder a campos repetidos como
arrays
'a@b.com'
>>> persona.email.append("c@d.com") # agregar una dirección de correo electrónico >>>
serializado = persona.SerializeToString() # serializar persona en una cadena de bytes
>>> serializado

b'\n\x05Alice\x10{\x1a\x07a@b.com\x1a\x07c@d.com' >>> persona2 =
Persona() # crear una nueva Persona >>>
persona2.ParseFromString(serializado) # analizar la cadena de bytes (27 bytes de
longitud) 27

>>> persona == persona2 # ahora son iguales
Verdadero

```

En resumen, importamos la clase Person generada por protoc, creamos una instancia y jugamos con ella, visualizándola y leyendo y escribiendo algunos campos, luego la serializamos usando el método SerializeToString(). Estos son los datos binarios que están listos para ser guardados o transmitidos a través de la red. Al leer o recibir estos datos binarios, podemos analizarlos usando el método ParseFromString() y obtenemos una copia del objeto que 5 fue serializado.

Puede guardar el objeto Persona serializado en un archivo TFRecord, luego cargarlo y analizarlo: todo funcionaría bien. Sin embargo, ParseFromString() no es una operación de TensorFlow, por lo que no podría usarlo en una función de preprocessamiento en una canalización tf.data (excepto envolviéndolo en una operación tf.py\_function(), lo que haría que el código fuera más lento y menos portátil, como vio en el Capítulo 12). Sin embargo, puede utilizar la función tf.io.decode\_proto(), que puede analizar cualquier protobuf que desee, siempre que propo

Es la definición de protobuf (consulte el cuaderno para ver un ejemplo). Dicho esto, en la práctica generalmente querrás utilizar los protobufs predefinidos para los cuales TensorFlow proporciona operaciones de análisis dedicadas.

Veamos ahora estos protobufs predefinidos.

## Protobufs de TensorFlow

El protobuf principal que se utiliza normalmente en un archivo TFRecord es el protobuf de ejemplo, que representa una instancia en un conjunto de datos. Contiene una lista de funciones con nombre, donde cada función puede ser una lista de cadenas de bytes, una lista de flotantes o una lista de números enteros. Aquí está la definición de protobuf (del código fuente de TensorFlow):

```
sintaxis = "proto3"; mensaje
BytesList { valor de bytes repetidos = 1; } mensaje FloatList { valor
flotante repetido = 1 [empaquetado = verdadero]; } mensaje Int64List { valor int64
repetido =
1 [empaquetado = verdadero]; } mensaje Característica { única en su tipo {
    Lista de bytes lista_bytes = 1;
    Lista flotante lista_flotante = 2;
    Int64List int64_list = 3;
}
};

mensaje Características { mapa<cadena, Característica> característica = 1; };
mensaje Ejemplo { Características características = 1; };
```

Las definiciones de BytesList, FloatList e Int64List son bastante sencillas. Tenga en cuenta que [packed = true] se utiliza para campos numéricos repetidos, para una codificación más eficiente. Una característica contiene una BytesList, una FloatList o una Int64List. Una característica (con una s) contiene un diccionario que asigna un nombre de característica al valor de característica correspondiente. Y finalmente, un Ejemplo contiene solo un objeto Características.

## NOTA

¿Por qué se definió Ejemplo, ya que no contiene más que un objeto Características? Bueno, es posible que algún día los desarrolladores de TensorFlow decidan agregarle más campos. Siempre que la nueva definición de ejemplo todavía contenga el campo de características, con el mismo ID, será compatible con versiones anteriores. Esta extensibilidad es una de las grandes características de los protobufs.

Así es como puedes crear un tf.train.Example que represente a la misma persona que antes:

```
desde tensorflow.train importar BytesList, FloatList, Int64List

de tensorflow.train importar Característica, Características, Ejemplo

person_example =


Ejemplo( características=Características( característica={ "nombre":  

    Característica(bytes_list=BytesList(valor= [b"Alice"])), "id": Característica(int64_list=Int64List(valor=  

    [123])),  

    "correos electrónicos": Característica(bytes_list=BytesList(valor=  

    [b"a@b.com",  

    b"c@d.com"])) }))
```

El código es un poco detallado y repetitivo, pero puedes incluirlo fácilmente dentro de una pequeña función auxiliar. Ahora que tenemos un protobuf de ejemplo, podemos serializarlo llamando a su método SerializeToString() y luego escribir los datos resultantes en un archivo TFRecord. Escribámoslo cinco veces para fingir que tenemos varios contactos:

```
con tf.io.TFRecordWriter("my_contacts.tfrecord") como f: en rango(5):
    para _  

        f.write(person_example.SerializeToString())
```

¡Normalmente escribirías mucho más de cinco ejemplos! Por lo general, crearía un script de conversión que lea su formato actual (por ejemplo, archivos CSV), cree un protobuf de ejemplo para cada instancia, los serialice y los guarde en varios archivos TFRecord, idealmente mezclándolos en el proceso. Esto requiere un poco de trabajo, así que una vez más asegúrese de que sea realmente necesario (quizás su canalización funcione bien con archivos CSV).

Ahora que tenemos un buen archivo TFRecord que contiene varios ejemplos serializados, intentemos cargarlo.

Cargando y analizando ejemplos Para cargar los protobufs de ejemplo serializados, usaremos un tf.data.TFRecordDataset una vez más y analizaremos cada ejemplo usando tf.io.parse\_single\_example(). Requiere al menos dos argumentos: un tensor escalar de cadena que contenga los datos serializados y una descripción de cada característica. La descripción es un diccionario que asigna cada nombre de característica a un descriptor tf.io.FixedLenFeature que indica la forma, el tipo y el valor predeterminado de la característica, o a un descriptor tf.io.VarLenFeature que indica solo el tipo si la longitud de la lista de características puede varían (como para la función "correos electrónicos").

El siguiente código define un diccionario de descripción y luego crea un TFRecordDataset y le aplica una función de preprocessamiento personalizada para analizar cada protobuf de ejemplo serializado que contiene este conjunto de datos:

```
descripción_característica = {
    "nombre": tf.io.FixedLenFeature([], tf.string, default_value=""),
    "id": tf.io.FixedLenFeature([], tf.int64, default_value=0), "correos electrónicos":

        tf.io.VarLenFeature(tf.cadena),
}
```

```

def parse(ejemplo_serializado): devuelve
    tf.io.parse_single_example(ejemplo_serializado, descripción_característica)

conjunto de datos =
tf.data.TFRecordDataset(["my_contacts.tfrecord"]).map(parse ) para parsed_example en

el conjunto de datos: print(parsed_example)

```

Las características de longitud fija se analizan como tensores regulares, pero las características de longitud variable se analizan como tensores dispersos. Puedes convertir un tensor disperso en un tensor denso usando `tf.sparse.to_dense()`, pero en este caso es más sencillo simplemente acceder a sus valores:

```

>>> tf.sparse.to_dense(parsed_example["correos electrónicos"],
default_value=b'')
<tf.Tensor: [...] dtype=string, numpy=array([b'a@b.com', b'c@d.com'], [...])> >>>
parsed_example["correos
electrónicos"].values <tf.Tensor: [...] dtype=string,
numpy=array([b'a @b.com', b'c@d.com'], [...])>

```

En lugar de analizar ejemplos uno por uno usando `tf.io.parse_single_example()`, es posible que desees analizarlos lote por lote usando `tf.io.parse_example()`:

```

def parse(ejemplos_serializados): devuelve
    tf.io.parse_example(ejemplos_serializados, descripción_característica)

conjunto de datos =
tf.data.TFRecordDataset(["mis_contactos.tfrecord"]).batch(2). map(parse) para

parsed_examples en el conjunto de datos:
    print(parsed_examples) # dos ejemplos a la vez

```

Por último, una BytesList puede contener cualquier dato binario que desee, incluido cualquier objeto serializado. Por ejemplo, puede usar `tf.io.encode_jpeg()` para codificar una imagen usando el formato JPEG y colocar estos datos binarios en una BytesList. Más tarde, cuando su código lea TFRecord, comenzará analizando el ejemplo, luego deberá llamar a `tf.io.decode_jpeg()` para analizar los datos y obtener la imagen original (o puede usar `tf.io.decode_image()`, que puede decodificar cualquier imagen BMP, GIF, JPEG o PNG). También puede almacenar cualquier tensor que desee en BytesList serializando el tensor usando `tf.io.serialize_tensor()` y luego colocando la cadena de bytes resultante en una función BytesList. Más adelante, cuando analice TFRecord, podrá analizar estos datos usando `tf.io.parse_tensor()`. Consulte el cuaderno de este capítulo en <https://homl.info/colab3> para ver ejemplos de almacenamiento de imágenes y tensores en un archivo TFRecord.

Como puede ver, el protobuf de ejemplo es bastante flexible, por lo que probablemente será suficiente para la mayoría de los casos de uso. Sin embargo, puede resultar un poco engorroso de utilizar cuando se trata de listas de listas. Por ejemplo, supongamos que desea clasificar documentos de texto. Cada documento se puede representar como una lista de oraciones, donde cada oración se representa como una lista de palabras. Y quizás cada documento también tenga una lista de comentarios, donde cada comentario se representa como una lista de palabras. También puede haber algunos datos contextuales, como el autor, el título y la fecha de publicación del documento. El protobuf SequenceExample de TensorFlow está diseñado para dicho uso. Los casos.

## Manejo de listas de listas utilizando el SecuenciaEjemplo Protobuf

Aquí está la definición del protobuf SequenceExample:

```
mensaje FeatureList { característica característica repetida = 1; }; mensaje Listas de
funciones { mapa<cadena, Lista de funciones> lista_características = 1; };
mensaje SecuenciaEjemplo {

    Contexto de características = 1;
    Listas de funciones feature_lists = 2;
};
```

Un SequenceExample contiene un objeto Features para los datos contextuales y un objeto FeatureLists que contiene uno o más objetos FeatureList con nombre (por ejemplo, una FeatureList denominada "contenido" y otra denominada "comentarios"). Cada FeatureList contiene una lista de objetos Feature, cada uno de los cuales puede ser una lista de cadenas de bytes, una lista de enteros de 64 bits o una lista de flotantes (en este ejemplo, cada Feature representaría una oración o un comentario, tal vez en forma de lista de identificadores de palabras). Crear un SequenceExample, serializarlo y analizarlo es similar a crear, serializar y analizar un ejemplo, pero debe usar `tf.io.parse_single_sequence_example()` para analizar un único SequenceExample o `tf.io.parse_sequence_example()` para analizar un lote . Ambas funciones devuelven una tupla que contiene las características del contexto (como un diccionario) y las listas de características (también como un diccionario).

Si las listas de características contienen secuencias de diferentes tamaños (como en el ejemplo anterior), es posible que desee convertirlas en tensores irregulares usando `tf.RaggedTensor.from_sparse()` (consulte el cuaderno para ver el código completo):

```
parsed_context, parsed_feature_lists =
tf.io.parse_single_sequence_example( serialized_sequence_example,
descripciones_de_características_de_contexto,
secuencia_feature_descriptions) parsed_content
=
tf.RaggedTensor.from_sparse(parsed_feature_lists["contenido"] )
```

Ahora que sabe cómo almacenar, cargar, analizar y preprocesar datos de manera eficiente utilizando la API tf.data, TFRecords y protobufs, es hora de centrar nuestra atención en las capas de preprocesamiento de Keras.

## Capas de preprocesamiento de Keras La

preparación de sus datos para una red neuronal generalmente requiere normalizar las características numéricas, codificar las características categóricas y el texto, recortar y cambiar el tamaño de las imágenes, y más. Hay varias opciones para esto:

- El preprocesamiento se puede realizar con anticipación al preparar sus archivos de datos de entrenamiento, utilizando cualquier herramienta que desee, como NumPy, Pandas o Scikit-Learn. Deberá aplicar exactamente los mismos pasos de preprocesamiento en producción para garantizar que su modelo de producción reciba entradas preprocesadas similares a aquellas en las que fue entrenado.
- Alternativamente, puede preprocesar sus datos sobre la marcha mientras los carga con tf.data, aplicando una función de preprocesamiento a cada elemento de un conjunto de datos usando el método map() de ese conjunto de datos, como hicimos anteriormente en este capítulo. Nuevamente, deberá aplicar los mismos pasos de preprocesamiento en producción.
- Un último enfoque es incluir capas de preprocesamiento directamente dentro de su modelo para que pueda preprocesar todos los datos de entrada sobre la marcha durante el entrenamiento y luego usar las mismas capas de preprocesamiento en producción. El resto de este capítulo analizará este último enfoque.

Keras ofrece muchas capas de preprocesamiento que puede incluir en sus modelos: se pueden aplicar a características numéricas, características categóricas, imágenes y texto. Repasaremos las características numéricas y categóricas en las siguientes secciones, así como el texto básico.

preprocesamiento, y cubriremos el preprocesamiento de imágenes en [el Capítulo 14](#) y el preprocesamiento de texto más avanzado en [el Capítulo 16](#).

La capa de normalización Como vimos

en [el Capítulo 10](#), Keras proporciona una capa de normalización que podemos usar para estandarizar las características de entrada. Podemos especificar la media y la varianza de cada característica al crear la capa o, más simplemente, pasar el conjunto de entrenamiento al método `adapt()` de la capa antes de ajustar el modelo, para que la capa pueda medir las medias y las varianzas de las características por sí sola antes. capacitación:

```
norm_layer = tf.keras.layers.Normalization() modelo =  
tf.keras.models.Sequential([ norm_layer,  
  
    tf.keras.layers.Dense(1)  
])  
model.compile(loss="mse",  
optimizer=tf.keras.optimizers.SGD(learning_rate=2e-3))  
norm_layer.adapt(X_train) # calcula la media y la varianza de cada característica  
model.fit(X_train,  
y_train, validation_data=(X_valid, y_valid), épocas=5)
```

#### CONSEJO

La muestra de datos pasada al método `adapt()` debe ser lo suficientemente grande como para ser representativa de su conjunto de datos, pero no tiene que ser el conjunto de entrenamiento completo: para la capa de Normalización, generalmente se necesitarán unos cientos de instancias muestreadas aleatoriamente del conjunto de entrenamiento. ser suficiente para obtener una buena estimación de las medias y variaciones de las características.

Dado que incluimos la capa de Normalización dentro del modelo, ahora podemos implementar este modelo en producción sin tener que preocuparnos por la normalización nuevamente: el modelo simplemente se encargará de ello (consulte la [Figura 13-4](#)). ¡Fantástico! Este enfoque elimina completamente el riesgo.

de discrepancia en el preprocessamiento, que ocurre cuando las personas intentan mantener un código de preprocessamiento diferente para capacitación y producción, pero actualizan uno y se olvidan de actualizar el otro. Luego, el modelo de producción termina recibiendo datos preprocessados de una manera que no esperaba.

Si tienen suerte, consiguen un error claro. De lo contrario, la precisión del modelo simplemente se degrada silenciosamente.

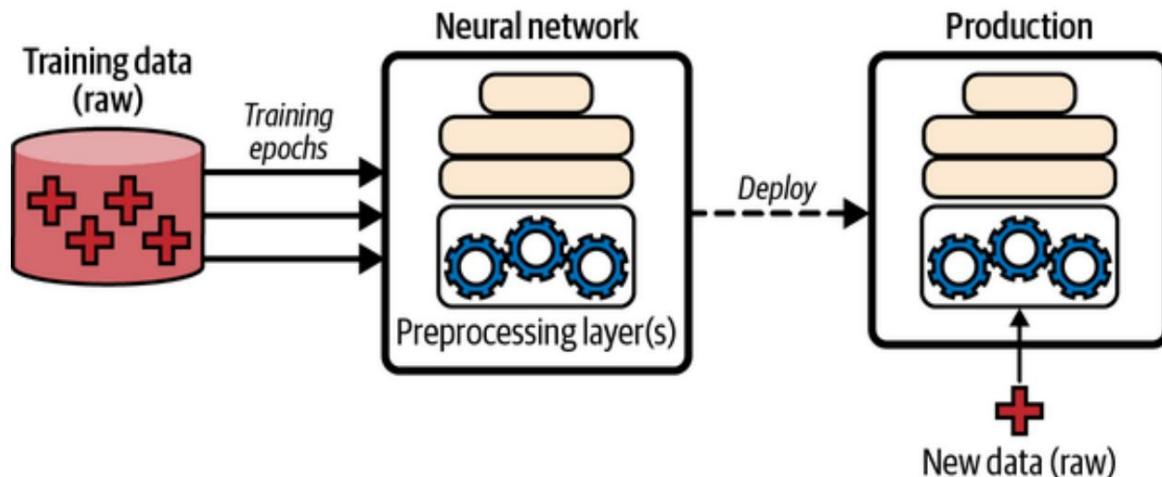


Figura 13-4. Incluir capas de preprocessamiento dentro de un modelo

Incluir la capa de preprocessamiento directamente en el modelo es agradable y sencillo, pero ralentizará el entrenamiento (sólo muy ligeramente en el caso de la capa de Normalización): de hecho, dado que el preprocessamiento se realiza sobre la marcha durante el entrenamiento, ocurre una vez por época. Podemos hacerlo mejor normalizando todo el conjunto de entrenamiento solo una vez antes del entrenamiento. Para hacer esto, podemos usar la capa de Normalización de forma independiente (muy parecida a un Scikit-Learn StandardScaler):

```
norma_capa = tf.keras.layers.Normalización()
norma_capa.adapt(X_train)
X_train_scaled = norma_layer(X_train)
X_valid_scaled = norma_layer(X_valid)
```

Ahora podemos entrenar un modelo con los datos escalados, esta vez sin Capa de normalización:

```

modelo =
tf.keras.models.Sequential([tf.keras.layers.Dense(1)]) model.compile(loss="mse",

optimizador=tf.keras.optimizers.SGD(learning_rate=2e-3)) modelo.
ajuste(X_train_scaled, y_train, épocas=5,
validation_data=(X_valid_scaled, y_valid))

```

¡Bien! Esto debería acelerar un poco el entrenamiento. Pero ahora el modelo no preprocesará sus entradas cuando lo implementemos en producción. Para solucionar este problema, sólo necesitamos crear un nuevo modelo que envuelva tanto la capa de Normalización adaptada como el modelo que acabamos de entrenar. Luego podemos implementar este modelo final en producción, y este se encargará tanto de preprocesar sus entradas como de hacer predicciones (consulte la [Figura 13-5](#)):

```

final_model = tf.keras.Sequential([capa_norma, modelo])
X_new = X_test[:3] # finge que tenemos algunas instancias nuevas (sin escalar) y_pred
=
final_model(X_new) # preprocesa los datos y hace predicciones

```

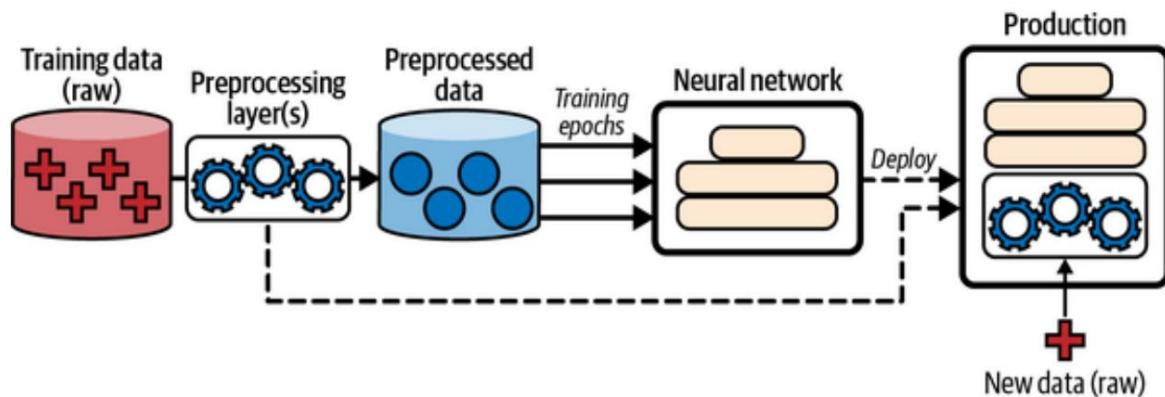


Figura 13-5. Preprocesar los datos solo una vez antes del entrenamiento usando capas de preprocesamiento y luego implementar estas capas dentro del modelo final.

Ahora tenemos lo mejor de ambos mundos: el entrenamiento es rápido porque solo preprocesamos los datos una vez antes de que comience el entrenamiento, y el modelo final puede preprocesar sus entradas sobre la marcha sin ningún riesgo de que el preprocesamiento no coincida.

Además, las capas de preprocessamiento de Keras funcionan muy bien con la API tf.data. Por ejemplo, es posible pasar un tf.data.Dataset al método adapt() de una capa de preprocessamiento. También es posible aplicar una capa de preprocessamiento de Keras a un tf.data.Dataset utilizando el método map() del conjunto de datos. Por ejemplo, así es como podría aplicar una capa de Normalización adaptada a las entidades de entrada de cada lote en un conjunto de datos:

```
conjunto de datos = conjunto de datos.map(lambda X, y: (norm_layer(X), y))
```

Por último, si alguna vez necesita más funciones de las que proporcionan las capas de preprocessamiento de Keras, siempre puede escribir su propia capa de Keras, tal como lo comentamos en el [Capítulo 12](#). Por ejemplo, si la capa de Normalización no existiera, podría obtener un resultado similar. usando la siguiente capa personalizada:

```
importar numpy como np
```

```
clase MiNormalización(tf.keras.layers.Layer):
    def adaptar(self, X):
        self.mean_ = np.mean(X,
                             eje=0, keepdims=True)
        self.std_ = np.std(X, eje= 0,
                           mantenerdims=True)

    llamada def (auto, entradas):
        eps = tf.keras.backend.epsilon() # un pequeño
        término de suavizado
        retorno (entradas - self.mean_) / (self.std_ + eps)
```

A continuación, veamos otra capa de preprocessamiento de Keras para características numéricas: la capa de Discretización.

## La capa de discretización

El objetivo de la capa de discretización es transformar una característica numérica en una característica categórica asignando rangos de valores (llamados contenedores) a categorías. Esto a veces es útil para funciones con

distribuciones multimodales, o con características que tienen una relación altamente no lineal con el objetivo. Por ejemplo, el siguiente código asigna una característica de edad numérica a tres categorías: menos de 18, 18 a 50 (no incluido) y 50 o más:

```
>>> edad = tf.constant([[10.], [93.], [57.], [18.], [37.], [5.]]) >>> discretize_layer
=
tf.keras.
capas.Discretización(bin_boundaries=[18., 50.]) >>> categorías_edad =
discretize_layer(edad) >>> categorías_edad <tf.Tensor:
forma=(6, 1), dtype=int64,
numpy=array([[ 0], [2],[2],[1],[1],[0]])>
```

En este ejemplo, proporcionamos los límites de contenedor deseados. Si lo prefiere, puede proporcionar la cantidad de contenedores que desea y luego llamar al método adapt() de la capa para permitirle encontrar los límites de contenedor apropiados según los percentiles de valor. Por ejemplo, si configuramos num\_bins=3, entonces los límites del contenedor se ubicarán en los valores justo debajo de los percentiles 33 y 66 (en este ejemplo, en los valores 10 y 37):

```
>>> discretize_layer =
tf.keras.layers.Discretization(num_bins=3) >>>
discretize_layer.adapt(edad) >>>
categorías_edad = discretize_layer(edad) >>>
categorías_edad
<tf.Tensor: forma=(6, 1 ), dtype=int64, numpy=array([[1], [2],[2],[1],[2],[0]])>
```

Por lo general, los identificadores de categorías como estos no deben pasarse directamente a una red neuronal, ya que sus valores no se pueden comparar de manera significativa. En su lugar, deberían codificarse, por ejemplo utilizando codificación one-hot. Veamos cómo hacer esto ahora.

## La capa de codificación de categorías

Cuando sólo hay unas pocas categorías (p. ej., menos de una docena o dos), la codificación one-hot suele ser una buena opción (como se analiza en el [Capítulo 2](#)). Para hacer esto, Keras proporciona la capa CategoryEncoding. Por ejemplo, codifiquemos en caliente la función age\_categories que acabamos de crear:

```
>>> onehot_layer =
tf.keras.layers.CategoryEncoding(num_tokens=3) >>>
onehot_layer(age_categories) <tf.Tensor:
forma=(6, 3), dtype=float32, numpy= array([[0., 1., 0.], [0., 0., 1.],
[0., 0., 1.], [0., 1., 0.], [0.,
0., 1.] , [1., 0.,
0.]], dtype=float32)>
```

Si intenta codificar más de una característica categórica a la vez (lo que solo tiene sentido si todas usan las mismas categorías), la clase CategoryEncoding realizará codificación multi-caliente de forma predeterminada: el tensor de salida contendrá un 1 para cada categoría presente en cualquier característica de entrada. Por ejemplo:

```
>>> dos_categorías_edad = np.array([[1, 0], [2, 2], [2, 0]]) >>>
onehot_layer(dos_categorías_edad) <tf.Tensor:
forma=(3, 3), dtype =float32, numpy= matriz([[1., 1., 0.], [0., 0., 1.],
[1., 0., 1.]], dtype=float32)>
```

Si cree que es útil saber cuántas veces ocurrió cada categoría, puede configurar output\_mode="count" al crear la capa CategoryEncoding, en cuyo caso el tensor de salida contendrá el número de ocurrencias de cada categoría. En el ejemplo anterior, el resultado sería el mismo excepto por la segunda fila, que sería [0., 0., 2.].

Tenga en cuenta que tanto la codificación multi-hot como la codificación de recuento pierden información, ya que no es posible saber de qué característica proviene cada categoría activa. Por ejemplo, tanto [0, 1] como [1, 0] están codificados como [1., 1., 0.]. Si desea evitar esto, debe codificar en caliente cada función por separado y concatenar las salidas. De esta manera, [0, 1] se codificaría como [1., 0., 0., 1., 0.] y [1, 0] se codificaría como [0., 1., 0., 1., 0., 0.]. Puede obtener el mismo resultado modificando los identificadores de categorías para que no se superpongan. Por ejemplo:

```
>>> onehot_layer =
tf.keras.layers.CategoryEncoding(num_tokens=3 + 3) >>>
onehot_layer(two_age_categories + [0, 3]) # agrega 3 a la segunda característica

<tf.Tensor: forma=(3, 6), dtype=float32, numpy= matriz([[0., 1., 0., 1., 0.,
0.], [0., 0., 1., 0., 0., 1.], [0., 0., 1., 1., 0., 0.]],
dtype=float32)>
```

En este resultado, las tres primeras columnas corresponden a la primera característica y las tres últimas corresponden a la segunda característica. Esto permite que el modelo distinga las dos características. Sin embargo, también aumenta la cantidad de características alimentadas al modelo y, por lo tanto, requiere más parámetros del modelo. Es difícil saber de antemano si una única codificación multi-hot o una codificación one-hot por función funcionará mejor: depende de la tarea y es posible que deba probar ambas opciones.

Ahora puede codificar características de enteros categóricos mediante codificación one-hot o multi-hot. Pero ¿qué pasa con las características del texto categórico? Para esto, puedes usar la capa StringLookup.

## La capa StringLookup

Usemos la capa Keras StringLookup para codificar en caliente una característica de ciudades:

```
>>> ciudades = ["Auckland", "París", "París", "San Francisco"] >>>
str_lookup_layer
= tf.keras.layers.StringLookup() >>> str_lookup_layer.adapt(ciudades)
>>> str_lookup_layer([["París"], ["Auckland"],
["Auckland"], ["Montreal"]]) <tf.Tensor: forma=(4, 1), dtype=int64,
numpy=array([[ 1], [3], [3], [0]])>
```

Primero creamos una capa StringLookup, luego la adaptamos a los datos: encuentra que hay tres categorías distintas. Luego usamos la capa para codificar algunas ciudades. Están codificados como números enteros de forma predeterminada. Las categorías desconocidas se asignan a 0, como es el caso de "Montreal" en este ejemplo. Las categorías conocidas están numeradas empezando por 1, desde la categoría más frecuente hasta la menos frecuente.

Convenientemente, si configura `output_mode="one_hot"` al crear la capa StringLookup, generará un vector one-hot para cada categoría, en lugar de un número entero:

```
>>> str_lookup_layer =
tf.keras.layers.StringLookup(output_mode="one_hot") >>>
str_lookup_layer.adapt(ciudades) >>>
str_lookup_layer([["París"], ["Auckland"], ["Auckland" ],
["Montreal"]]) <tf.Tensor: forma=(4,
4), dtype=float32, numpy= matriz([[0., 1., 0., 0.], [0., 0., 0., 1.], [0.,
0., 0., 1.], [1., 0., 0., 0.]]),
dtype=float32)>
```

## CONSEJO

Keras también incluye una capa IntegerLookup que actúa de manera muy similar a la capa StringLookup pero toma números enteros como entrada, en lugar de cadenas.

Si el conjunto de entrenamiento es muy grande, puede ser conveniente adaptar la capa solo a un subconjunto aleatorio del conjunto de entrenamiento. En este caso, el método `adapt()` de la capa puede omitir algunas de las categorías más raras. De forma predeterminada, los asignaría a todos a la categoría 0, haciéndolos indistinguibles para el modelo. Para reducir este riesgo (sin dejar de adaptar la capa solo en un subconjunto del conjunto de entrenamiento), puede establecer `num_oov_indices` en un número entero mayor que 1. Esta es la cantidad de depósitos fuera de vocabulario (OOV) que se usarán: cada uno desconocido La categoría se asignará pseudoaleatoriamente a uno de los depósitos OOV, utilizando una función hash que mide el número de depósitos OOV. Esto permitirá que el modelo distinga al menos algunas de las categorías raras. Por ejemplo:

```
>>> str_lookup_layer =
tf.keras.layers.StringLookup(num_oov_indices=5) >>>
str_lookup_layer.adapt(ciudades) >>>
str_lookup_layer([["París"], ["Auckland"], ["Foo"], ["Bar"], ["Baz"]]) <tf.Tensor:
  forma=(4, 1), dtype=int64,
  numpy=array([[5], [7], [4], [3], [4]])>
```

Dado que hay cinco depósitos OOV, el ID de la primera categoría conocida ahora es 5 ("París"). Pero "Foo", "Bar" y "Baz" son desconocidos, por lo que cada uno de ellos se asigna a uno de los depósitos OOV. "Bar" tiene su propio depósito dedicado (con ID 3), pero lamentablemente "Foo" y "Baz" están asignados al mismo depósito (con ID 4), por lo que no se pueden distinguir según el modelo. Esto se llama colisión de hash. La única forma de reducir el riesgo de colisión es aumentar la cantidad de depósitos OOV. Sin embargo, esto también aumentará el número total de categorías, lo que requerirá más RAM y parámetros de modelo adicionales una vez que las categorías estén codificadas en caliente. Por lo tanto, no aumente demasiado ese número.

Esta idea de asignar categorías de forma pseudoaleatoria a depósitos se denomina truco de hash. Keras proporciona una capa dedicada que hace precisamente eso: la capa Hashing.

## La capa de hash

Para cada categoría, la capa Keras Hashing calcula un hash, módulo el número de depósitos (o "contenedores"). El mapeo es completamente pseudoaleatorio, pero estable entre ejecuciones y plataformas (es decir, la misma categoría siempre se asignará al mismo número entero, siempre que el número de contenedores no cambie). Por ejemplo, usemos la capa Hashing para codificar algunas ciudades:

```
>>> hashing_layer = tf.keras.layers.Hashing(num_bins=10) >>>
hashing_layer([["París"], ["Tokio"], ["Auckland"], ["Montreal"]]) <tf.Tensor:
  forma=(4, 1),
  dtype=int64, numpy=array([[0], [1], [9], [1]])>
```

La ventaja de esta capa es que no es necesario adaptarla en absoluto, lo que a veces puede resultar útil, especialmente en una configuración fuera del núcleo (cuando el conjunto de datos es demasiado grande para caber en la memoria). Sin embargo, una vez más tenemos una colisión de hash: "Tokio" y "Montreal" están asignados al mismo ID, lo que los hace indistinguibles según el modelo. Por lo tanto, normalmente es preferible ceñirse a la capa StringLookup.

Veamos ahora otra forma de codificar categorías: incrustaciones entrenables.

## Codificación de características categóricas mediante Incrustaciones

Una incrustación es una representación densa de algunos datos de dimensiones superiores, como una categoría o una palabra en un vocabulario. Si hay 50.000 categorías posibles, entonces la codificación one-hot produciría un vector disperso de 50.000 dimensiones (es decir, que contendría principalmente ceros). Por el contrario, una incrustación sería un vector denso comparativamente pequeño; por ejemplo, con sólo 100 dimensiones.

En el aprendizaje profundo, las incorporaciones generalmente se inicializan de forma aleatoria y luego se entrena mediante descenso de gradiente, junto con los demás parámetros del modelo. Por ejemplo, la categoría "NEAR BAY" en el conjunto de datos de vivienda de California podría representarse inicialmente mediante un vector aleatorio como [0,131, 0,890], mientras que la categoría "NEAR OCEAN" podría representarse mediante otro vector aleatorio como [0,631, 0,791 ]. En este ejemplo, utilizamos incrustaciones 2D, pero el número de dimensiones es un hiperparámetro que puedes modificar dado que estas incorporaciones se pueden entrenar, mejorarán gradualmente durante el entrenamiento; y como en este caso representan categorías bastante similares, el descenso del gradiente ciertamente terminará acercándolos, mientras que tenderá a alejarlos de la incorporación de la categoría "INLAND" (ver Figura 13-6) . De hecho, cuanto mejor sea la representación, más fácil será para la red neuronal hacer predicciones precisas, por lo que el entrenamiento tiende a hacer que las incorporaciones sean representaciones útiles de las categorías. Esto se llama aprendizaje de representación (verá otros tipos de aprendizaje de representación en el [Capítulo 17](#)).

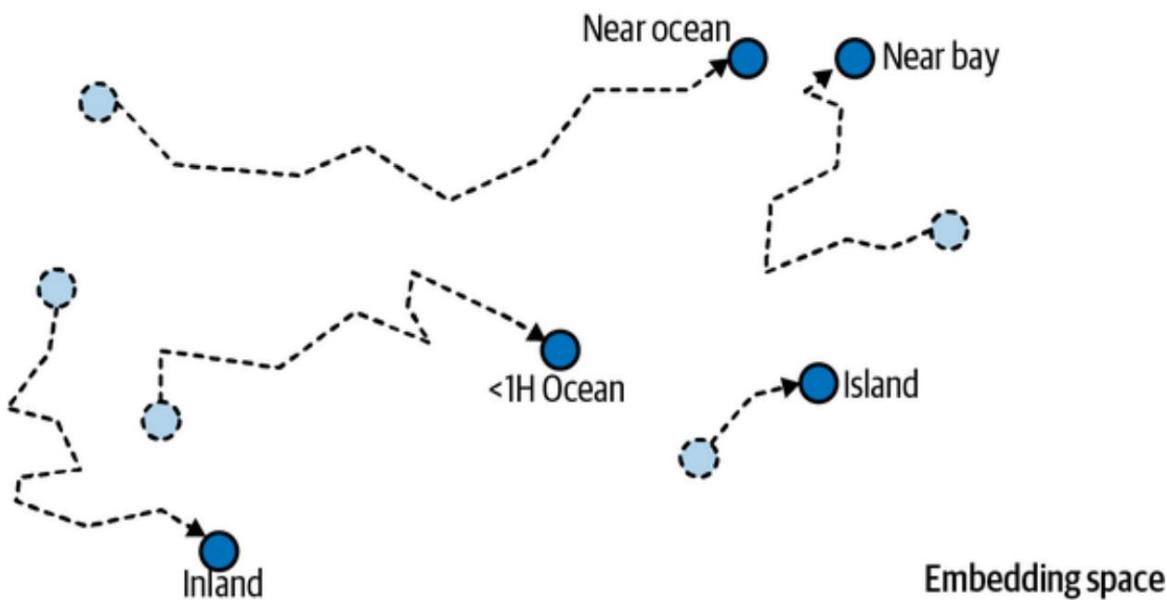


Figura 13-6. Las incrustaciones mejorarán gradualmente durante el entrenamiento.

## INTEGRACIONES DE PALABRAS

Las incrustaciones no sólo serán generalmente representaciones útiles para la tarea en cuestión, sino que muy a menudo estas mismas incrustaciones se pueden reutilizar con éxito para otras tareas. El ejemplo más común de esto son las incrustaciones de palabras (es decir, incrustaciones de palabras individuales): cuando trabaja en una tarea de procesamiento de lenguaje natural, a menudo es mejor reutilizar las incrustaciones de palabras previamente entrenadas que entrenar las suyas propias.

La idea de utilizar vectores para representar palabras se remonta a la década de 1960 y se han utilizado muchas técnicas sofisticadas para generar vectores útiles, incluido el uso de redes neuronales. Pero las cosas realmente despegaron en 2013, cuando Tomáš Mikolov y otros investigadores de Google publicaron un [artículo](#)<sup>6</sup> que describe una técnica eficiente para aprender incrustaciones de palabras utilizando redes neuronales, superando significativamente los intentos anteriores. Esto les permitió aprender incrustaciones en un corpus de texto muy grande: entrenaron una red neuronal para predecir las palabras cercanas a cualquier palabra determinada y obtuvieron incrustaciones de palabras asombrosas. Por ejemplo, los sinónimos tenían incrustaciones muy cercanas y palabras semánticamente relacionadas como Francia, España e Italia terminaron agrupadas.

Sin embargo, no se trata sólo de proximidad: las incrustaciones de palabras también se organizaron a lo largo de ejes significativos en el espacio de incrustación. Aquí hay un ejemplo famoso: si calcula Rey – Hombre + Mujer (sumando y restando los vectores de incrustación de estas palabras), entonces el resultado será muy cercano a la incrustación de la palabra Reina (ver [Figura 13-7](#)). En otras palabras, ¡la palabra incrustaciones codifica el concepto de género! De manera similar, se puede calcular Madrid – España + Francia, y el resultado es cercano al de París, lo que parece mostrar que la noción de ciudad capital también estaba codificada en las incrustaciones.

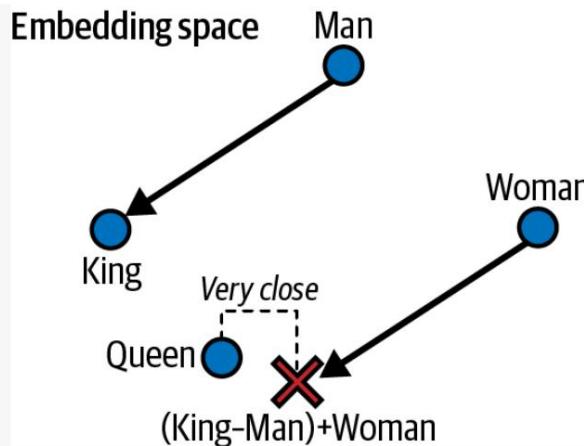


Figura 13-7. Las incrustaciones de palabras similares tienden a ser cercanas y algunos ejes parecen codificar conceptos significativos.

Desafortunadamente, las incrustaciones de palabras a veces capturan nuestros peores prejuicios. Por ejemplo, aunque aprenden correctamente que el hombre es para el rey lo que la mujer es para la reina, también parecen aprender que el hombre es para el médico como la mujer es para la enfermera: ¡un sesgo bastante sexista! Para ser justos, este ejemplo en particular probablemente sea exagerado, como se señaló en un <sup>7</sup> artículo de 2019. por Malvina Nissim et al. Sin embargo, garantizar la equidad en los algoritmos de aprendizaje profundo es un tema de investigación importante y activo.

Keras proporciona una capa de incrustación, que envuelve una matriz de incrustación: esta matriz tiene una fila por categoría y una columna por dimensión de incrustación. De forma predeterminada, se inicializa aleatoriamente. Para convertir un ID de categoría en una incrustación, la capa Incrustación simplemente busca y devuelve la fila que corresponde a esa categoría.

¡Eso es todo al respecto! Por ejemplo, inicialicemos una capa de incrustación con cinco filas e incrustaciones 2D, y usémosla para codificar algunas categorías:

```
>>> tf.random.set_seed(42) >>>
embedding_layer =
tf.keras.layers.Embedding(input_dim=5, output_dim=2) >>>
embedding_layer(np.array([2, 4, 2])) <tf.Tensor: forma=(3,
2), dtype=float32, numpy= matriz([-0.04663396, 0.01846724],
```

```
[ -0.02736737, -0.02768031], [-0.04663396,  
0.01846724]], dtype=float32)>
```

Como puede ver, la categoría 2 se codifica (dos veces) como el vector 2D [-0.04663396, 0.01846724], mientras que la categoría 4 se codifica como [-0.02736737, -0.02768031]. Dado que la capa aún no está entrenada, estas codificaciones son simplemente aleatorias.

#### ADVERTENCIA

Una capa de incrustación se inicializa aleatoriamente, por lo que no tiene sentido usarla fuera de un modelo como una capa de preprocesamiento independiente a menos que la inicialice con pesos previamente entrenados.

Si desea incrustar un atributo de texto categórico, simplemente puede encadenar una capa StringLookup y una capa Incrustar, así:

```
>>> tf.random.set_seed(42) >>>  
ocean_prox = ["<1H OCÉANO", "INTERRESTRE", "CERCA DEL OCÉANO", "CERCA  
DE LA BAHÍA", "ISLA"] >>>  
str_lookup_layer = tf.keras.layers.StringLookup() >>> str_lookup_layer.adapt(ocean_prox)  
>>> lookup_and_embed = tf.keras.Sequential([  
  
    ...         str_lookup_layer,  
    ...  
    tf.keras.layers.Embedding(input_dim=str_lookup_layer.vocabulary_size(),  
  
    ...                     salida_dim=2)  
    ...])  
    ...  
>>> lookup_and_embed(np.array([["<1H OCEAN"], ["ISLAND"], ["<1H OCEAN"]]))  
<tf.Tensor: forma=(3,  
2), dtype=float32 , numpy= matriz([-0.01896119, 0.02223358],  
[ 0.02401174, 0.03724445], [-0.01896119,  
0.02223358]], dtype=float32)>
```

Tenga en cuenta que el número de filas en la matriz de incrustación debe ser igual al tamaño del vocabulario: ese es el número total de categorías, incluidas las categorías conocidas más los depósitos OOV (solo uno de forma predeterminada). El método `vocabulario_size()` de la clase `StringLookup` devuelve convenientemente este número.

## CONSEJO

En este ejemplo utilizamos incrustaciones 2D, pero como regla general, las incrustaciones suelen tener de 10 a 300 dimensiones, según la tarea, el tamaño del vocabulario y el tamaño de su conjunto de entrenamiento. Tendrá que ajustar este hiperparámetro.

Juntando todo, ahora podemos crear un modelo de Keras que puede procesar una característica de texto categórico junto con características numéricas regulares y aprender una incrustación para cada categoría (así como para cada depósito OOV):

```
X_train_num, X_train_cat, y_train = [...] # carga el conjunto de
entrenamiento
X_valid_num, X_valid_cat, y_valid = [...] # y el conjunto de validación

num_input = tf.keras.layers.Input(shape=[8], nombre="num") cat_input =
tf.keras.layers.Input(shape=[], dtype=tf.string, nombre="cat")
cat_embeddings =
lookup_and_embed(cat_input) encoded_inputs =
tf.keras.layers.concatenate([num_input, cat_embeddings]) salidas =
tf.keras.layers.Dense(1)
(encoded_inputs) modelo = tf.keras.models.Model(inputs=[num_input,
cat_input], salidas=[salidas]) model.compile(loss="mse",
optimizador="sgd") historial =
model.fit((X_train_num, X_train_cat), y_train, epochs=5,
datos_validación=((X_valid_num,
X_valid_cat), y_valid))
```

Este modelo toma dos entradas: num\_input, que contiene ocho características numéricas por instancia, más cat\_input, que contiene una única entrada de texto categórico por instancia. El modelo utiliza el modelo lookup\_and\_embed que creamos anteriormente para codificar cada categoría de proximidad al océano como la incrustación entrenable correspondiente. A continuación, concatena las entradas numéricas y las incrustaciones utilizando la función concatenate() para producir las entradas codificadas completas, que están listas para enviarse a una red neuronal. Podríamos agregar cualquier tipo de red neuronal en este punto, pero por simplicidad simplemente agregamos una única capa de salida densa y luego creamos el modelo Keras con las entradas y salidas que acabamos de definir. A continuación compilamos el modelo y lo entrenamos, pasando las entradas numéricas y categóricas.

Como vio en [el Capítulo 10](#), dado que las capas de entrada se denominan "num" y "cat", también podríamos haber pasado los datos de entrenamiento al método fit() usando un diccionario en lugar de una tupla: {"num": X\_train\_num, "gato": X\_train\_cat}. Alternativamente, podríamos haber pasado un tf.data.Dataset que contenga lotes, cada uno representado como ((X\_batch\_num, X\_batch\_cat), y\_batch) o como {"num": X\_batch\_num, "cat": X\_batch\_cat}, y\_batch). Y por supuesto lo mismo ocurre con los datos de validación.

## NOTA

La codificación one-hot seguida de una capa densa (sin función de activación ni sesgos) es equivalente a una capa de incrustación. Sin embargo, la capa de incrustación utiliza muchos menos cálculos, ya que evita muchas multiplicaciones por cero; la diferencia de rendimiento se vuelve clara cuando crece el tamaño de la matriz de incrustación. La matriz de peso de la capa Densa desempeña el papel de matriz de incrustación. Por ejemplo, usar vectores one-hot de tamaño 20 y una capa Densa con 10 unidades es equivalente a usar una capa de Incrustación con `input_dim=20` y `output_dim=10`. Como resultado, sería un desperdicio utilizar más dimensiones de incrustación que el número de unidades en la capa que sigue a la capa de incrustación.

Bien, ahora que ha aprendido a codificar características categóricas, es hora de centrar nuestra atención en el preprocesamiento de texto.

## Preprocesamiento de texto

Keras proporciona una capa `TextVectorization` para el preprocesamiento básico de texto. Al igual que la capa `StringLookup`, debes pasarle un vocabulario al crearla o dejar que aprenda el vocabulario a partir de algunos datos de entrenamiento usando el método `adapt()`. Veamos un ejemplo:

```
>>> train_data = ["Ser", "!(ser)", "Esa es la pregunta", "Ser, ser, ser."]
>>>
text_vec_layer =
tf.keras.layers.TextVectorization() >>> text_vec_layer.adapt(train_data) >>>
text_vec_layer(["¡Sé bueno!", "Pregunta: ¿ser o
ser?"])
<tf.Tensor: forma=(2, 4), dtype=int64, numpy= array( [[2, 1, 0, 0], [6, 2, 1,
2]])>
```

Las dos frases "¡Sé bueno!" y "Pregunta: ¿ser o ser?" se codificaron como `[2, 1, 0, 0]` y `[6, 2, 1, 2]`, respectivamente. El

El vocabulario se aprendió de las cuatro oraciones en los datos de entrenamiento: “be” = 2, “to” = 3, etc. Para construir el vocabulario, el método `adapt()` primero convirtió las oraciones de entrenamiento a minúsculas y eliminó la puntuación, razón por la cual “¿Ser”, “ser” y “ser?” están todos codificados como “be” = 2. A continuación, las oraciones se dividieron en espacios en blanco y las palabras resultantes se ordenaron por frecuencia descendente, produciendo el vocabulario final. Al codificar oraciones, las palabras desconocidas se codifican como 1. Por último, dado que la primera oración es más corta que la segunda, se completó con ceros.

## CONSEJO

La capa `TextVectorization` tiene muchas opciones. Por ejemplo, puede conservar el caso y la puntuación si lo desea, estableciendo `estandarizar = Ninguno`, o puede pasar cualquier función de estandarización que desee como argumento de estandarización. Puede evitar la división configurando `split=None`, o puede pasar su propia función de división. Puede configurar el argumento `output_sequence_length` para garantizar que todas las secuencias de salida se recorten o rellenen a la longitud deseada, o puede configurar `ragged=True` para obtener un tensor irregular en lugar de un tensor normal. Consulte la documentación para conocer más opciones.

Los ID de palabras deben codificarse, normalmente usando una capa de incrustación: lo haremos en [el Capítulo 16](#). Alternativamente, puede configurar el argumento `modo_salida` de la capa `TextVectorization` en “multi\_hot” o “count” para obtener las codificaciones correspondientes. Sin embargo, simplemente contar palabras no suele ser lo ideal: palabras como “para” y “el” son tan frecuentes que apenas importan, mientras que palabras más raras como “baloncesto” son mucho más informativas. Por lo tanto, en lugar de configurar `Output_mode` en “multi\_hot” o “count”, generalmente es preferible configurarlo en “tf\_idf”, que significa frecuencia de término × frecuencia de documento inversa (TF-IDF). Esto es similar a la codificación de conteo, pero las palabras que ocurren con frecuencia en el entrenamiento

los datos se reducen y, a la inversa, las palabras raras se aumentan.

Por ejemplo:

```
>>> text_vec_layer =
tf.keras.layers.TextVectorization(output_mode="tf_idf")
>>> text_vec_layer.adapt(tren_datos)
>>> text_vec_layer(["¡Sé bueno!", "Pregunta: ¿ser o ser?"])
<tf.Tensor: forma=(2, 6), dtype=float32, numpy=
array([[0.96725637, 0.6931472, 0. [0.96725637, 1.3862944], 0, dtype=float32>,
       , 0., 0., 0., 1.0986123
      ],
```

Hay muchas variantes de TF-IDF, pero la forma en que

La capa TextVectorization lo implementa multiplicando cada uno.

recuento de palabras por un peso igual a  $\log(1 + d / (f + 1))$ , donde  $d$  es el número total de oraciones (también conocidas como documentos) en los datos de entrenamiento y  $f$  cuenta cuántas de estas oraciones de entrenamiento contienen la dado

palabra. Por ejemplo, en este caso hay  $d = 4$  oraciones en el

datos de entrenamiento, y la palabra "be" aparece en  $f = 3$  de estos. desde el

La palabra "ser" aparece dos veces en la oración "Pregunta: ¿ser o ser?", se vuelve codificado como  $2 \times \log(1 + 4 / (1 + 3)) \approx 1.3862944$ . La palabra "pregunta" sólo aparece una vez, pero como es una palabra menos común, su codificación es casi tan alto:  $1 \times \log(1 + 4 / (1 + 1)) \approx 1.0986123$ . Tenga en cuenta que el

El peso promedio se utiliza para palabras desconocidas.

Este enfoque para la codificación de texto es sencillo de usar y puede dar resultados bastante buenos para tareas básicas de procesamiento del lenguaje natural, pero tiene varias limitaciones importantes: sólo funciona con idiomas que separa palabras con espacios, no distingue entre

homónimos (por ejemplo, "llevar" versus "oso de peluche"), no da ninguna pista para su modelo de que palabras como "evolución" y "evolutivo" están relacionadas, etc. Y si utiliza codificación multi-hot, count o TF-IDF, entonces el

Se pierde el orden de las palabras. ¿Entonces cuáles son las otras opciones?

Una opción es utilizar la [biblioteca de texto TensorFlow](#), que proporciona funciones de preprocessamiento de texto más avanzadas que las Capa de vectorización de texto. Por ejemplo, incluye varios

tokenizadores de subpalabras capaces de dividir el texto en tokens más pequeños que palabras, lo que hace posible que el modelo detecte más fácilmente que “evolución” y “evolutivo” tienen algo en común (más sobre tokenización de subpalabras en el Capítulo 16 ).

Otra opción más es utilizar componentes de modelo de lenguaje previamente entrenados. Veamos esto ahora.

## Uso de componentes del modelo de lenguaje previamente entrenado

La [biblioteca de TensorFlow Hub](#) facilita la reutilización de componentes de modelos previamente entrenados en sus propios modelos, para texto, imágenes, audio y más. Estos componentes del modelo se denominan módulos. Simplemente explore el [repositorio de TF Hub](#), encuentre el que necesita y copie el ejemplo de código en su proyecto, y el módulo se descargará automáticamente y se incluirá en una capa de Keras que puede incluir directamente en su modelo. Los módulos normalmente contienen código de preprocesamiento y pesos previamente entrenados y, por lo general, no requieren capacitación adicional (pero, por supuesto, el resto de su modelo ciertamente requerirá capacitación).

Por ejemplo, se encuentran disponibles algunos potentes modelos de lenguaje previamente entrenados. Los más potentes son bastante grandes (varios gigabytes), así que para un ejemplo rápido usemos el módulo nnlm-en-dim50, versión 2, que es un módulo bastante básico que toma texto sin formato como entrada y genera incrustaciones de oraciones de 50 dimensiones. Importaremos TensorFlow Hub y lo usaremos para cargar el módulo, luego usaremos ese módulo [8](#) para codificar dos oraciones en vectores:

```
>>> import tensorflow_hub as hub
hub_layer = hub.KerasLayer("https://tfhub.dev/google/nnlm-en-dim50/2") >>>
frase_embeddings = hub_layer(tf.constant(["Para ser", "No ser"])) >>>

frase_embeddings.numpy().round(2) array([-0.25,
 0.28, 0.01, 0.1 [...], -0.2, 0.2, -0.08, 0.02, [...], 0.15], , 0.05, 0.31], , -0.04,
 dtype=float32)
```

La capa hub.KerasLayer descarga el módulo desde la URL proporcionada. Este módulo en particular es un codificador de oraciones: toma cadenas como entrada y codifica cada una como un único vector (en este caso, un vector de 50 dimensiones). Internamente, analiza la cadena (dividiendo palabras en espacios) e incrusta cada palabra usando una matriz de incrustación que fue previamente entrenada en un corpus enorme: el corpus Google News 7B (¡siete mil millones de palabras!). Luego calcula la media de todas las incrustaciones de palabras y el resultado es la incrustación de la oración.

9

Solo necesita incluir este hub\_layer en su modelo y estará listo para comenzar. Tenga en cuenta que este modelo de idioma en particular fue entrenado en el idioma inglés, pero hay muchos otros idiomas disponibles, así como modelos multilingües.

Por último, pero no menos importante, la excelente [biblioteca Transformers de código abierto de Hugging Face](#) también facilita la inclusión de potentes componentes de modelos de lenguaje dentro de sus propios modelos. Puedes navegar por [Hugging Face Hub](#), Elija el modelo que desee y utilice los ejemplos de código proporcionados para comenzar. Solía contener sólo modelos de lenguaje, pero ahora se ha ampliado para incluir modelos de imágenes y más.

Volveremos al procesamiento del lenguaje natural con más profundidad en el Capítulo 16. Veamos ahora las capas de preprocessamiento de imágenes de Keras.

## Capas de preprocessamiento de imágenes

La API de preprocessamiento de Keras incluye tres capas de preprocessamiento de imágenes:

- tf.keras.layers.Resizing cambia el tamaño de las imágenes de entrada al tamaño deseado. Por ejemplo, Cambiar tamaño (alto = 100, ancho = 200) cambia el tamaño de cada imagen a  $100 \times 200$ , posiblemente distorsionando la imagen. Si configura crop\_to\_aspect\_ratio=True, la imagen se recortará según la proporción de imagen de destino para evitar distorsiones.

- `tf.keras.layers.Rescaling` cambia la escala de los valores de píxeles. Por ejemplo, `Rescaling(scale=2/255, offset=-1)` escala los valores de  $0 \rightarrow 255$  a  $-1 \rightarrow 1$ .
- `tf.keras.layers.CenterCrop` recorta la imagen, manteniendo solo un parche central de la altura y el ancho deseados.

Por ejemplo, carguemos un par de imágenes de muestra y las recortemos al centro. Para esto, usaremos la función `load_sample_images()` de Scikit-Learn; esto carga dos imágenes en color, una de un templo chino y la otra de una flor (esto requiere la biblioteca Pillow, que ya debería estar instalada si está utilizando Colab o si siguió las instrucciones de instalación):

```
desde sklearn.datasets importar load_sample_images

imágenes = load_sample_images()["imágenes"]
crop_image_layer = tf.keras.layers.CenterCrop(alto=100, ancho=100)

cropped_images = crop_image_layer(imágenes)
```

Keras también incluye varias capas para el aumento de datos, como `RandomCrop`, `RandomFlip`, `RandomTranslation`, `RandomRotation`, `RandomZoom`, `RandomHeight`, `RandomWidth` y `RandomContrast`. Estas capas solo están activas durante el entrenamiento y aplican aleatoriamente alguna transformación a las imágenes de entrada (sus nombres se explican por sí solos). El aumento de datos aumentará artificialmente el tamaño del conjunto de entrenamiento, lo que a menudo conduce a un mejor rendimiento, siempre que las imágenes transformadas parezcan imágenes realistas (no aumentadas). Cubriremos el procesamiento de imágenes más de cerca en el próximo capítulo.

## NOTA

En esencia, las capas de preprocessamiento de Keras se basan en la API de bajo nivel de TensorFlow. Por ejemplo, la capa de Normalización usa `tf.nn.moments()` para calcular tanto la media como la varianza, la capa de Discretización usa `tf.raw_ops.Bucketize()`, CategoricalEncoding usa `tf.math.bincount()`, IntegerLookup y StringLookup usan `tf.lookup`, Hashing y TextVectorization usan varias operaciones del paquete `tf.strings`, Embedding usa `tf.nn.embedding_lookup()` y las capas de preprocessamiento de imágenes usan las operaciones del paquete `tf.image`. Si la API de preprocessamiento de Keras no es suficiente para sus necesidades, es posible que ocasionalmente necesite utilizar la API de bajo nivel de TensorFlow directamente.

Ahora veamos otra forma de cargar datos de manera fácil y eficiente en TensorFlow.

## El proyecto de conjuntos de datos de TensorFlow

Los [conjuntos de datos de TensorFlow \(TFDS\)](#) El proyecto hace que sea muy fácil cargar conjuntos de datos comunes, desde pequeños como MNIST o Fashion MNIST hasta conjuntos de datos enormes como ImageNet (¡necesitará bastante espacio en disco!). La lista incluye conjuntos de datos de imágenes, conjuntos de datos de texto (incluidos conjuntos de datos de traducción), conjuntos de datos de audio y vídeo, series temporales y mucho más. Puedes visitar <https://homl.info/tfds> para ver la lista completa, junto con una descripción de cada conjunto de datos. También puede consultar [Conozca sus datos](#), que es una herramienta para explorar y comprender muchos de los conjuntos de datos proporcionados por TFDS.

TFDS no está incluido con TensorFlow, pero si está ejecutando Colab o si siguió las instrucciones de instalación en <https://homl.info/install>, entonces ya está instalado. Luego puede importar `tensorflow_datasets`, generalmente como `tfds`, luego llamar a la función `tfds.load()`, que descargará los datos que desea.

(a menos que ya se haya descargado anteriormente) y devolver los datos como un diccionario de conjuntos de datos (normalmente uno para entrenamiento y otro para prueba, pero esto depende del conjunto de datos que elija). Por ejemplo, descarguemos MNIST:

```
importar tensorflow_datasets como tfds
conjuntos de datos = tfds.load(nombre="mnist")
mnist_train, mnist_test = conjuntos de datos["tren"], conjuntos
de datos["prueba"]
```

Luego puede aplicar cualquier transformación que desee (por lo general, barajar, agrupar y captar previamente) y estará listo para entrenar su modelo.

Aquí hay un ejemplo simple:

```
para lote en mnist_train.shuffle(10_000,
seed=42).batch(32).prefetch(1): imágenes
    = lote["imagen"] etiquetas =
    lote["etiqueta"] # [...] hacer algo
    con el imágenes y etiquetas
```

#### CONSEJO

La función load() puede mezclar los archivos que descarga: simplemente configure shuffle\_files=True. Sin embargo, esto puede ser insuficiente, por lo que es mejor mezclar un poco más los datos de entrenamiento.

Tenga en cuenta que cada elemento del conjunto de datos es un diccionario que contiene tanto las características como las etiquetas. Pero Keras espera que cada elemento sea una tupla que contenga dos elementos (nuevamente, las características y las etiquetas). Podrías transformar el conjunto de datos usando el método map(), así:

```
mnist_train = mnist_train.shuffle(buffer_size=10_000, semilla=42).batch(32)
mnist_train =
mnist_train.map( elementos lambda :
```

```
(elementos["imagen"], elementos["etiqueta"]))
mnist_train = mnist_train.prefetch(1)
```

Pero es más sencillo pedirle a la función load() que haga esto por usted configurando as\_supervised=True (obviamente, esto funciona solo para conjuntos de datos etiquetados).

Por último, TFDS proporciona una forma conveniente de dividir los datos utilizando el argumento de división. Por ejemplo, si desea utilizar el primer 90% del conjunto de entrenamiento para entrenamiento, el 10% restante para validación y todo el conjunto de prueba para pruebas, entonces puede configurar

split= ["train[:90%]", "entrenar[90%:]", "prueba"]. La función load() devolverá los tres conjuntos. Aquí hay un ejemplo completo, cargando y dividiendo el conjunto de datos MNIST usando TFDS, luego usando estos conjuntos para entrenar y evaluar un modelo Keras simple:

```
train_set, valid_set, test_set = tfds.load( nombre="mnist",
    split=["tren[:90%]",
        "tren[90%:]", "prueba"], as_supervised=True

) train_set = train_set.shuffle(buffer_size=10_000,
    seed=42).batch(32).prefetch(1) valid_set
= valid_set.batch(32).cache() test_set =
test_set.batch(32).cache() tf .random.set_seed(42)
modelo = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(10, activation="softmax")
])
model.compile(loss="sparse_categorical_crossentropy",
    optimizador="nadam",
    métricas=["exactitud"])
historial = model.fit(train_set, validation_data=valid_set, epochs=5) test_loss,
test_accuracy
= model.evaluate(test_set)
```

¡Felicitaciones, ha llegado al final de este capítulo bastante técnico! Quizás sientas que está un poco lejos de la belleza abstracta de

redes neuronales, pero el hecho es que el aprendizaje profundo a menudo implica grandes cantidades de datos, y saber cómo cargarlos, analizarlos y preprocesarlos de manera eficiente es una habilidad crucial. En el próximo capítulo, analizaremos las redes neuronales convolucionales, que se encuentran entre las arquitecturas de redes neuronales más exitosas para el procesamiento de imágenes y muchas otras aplicaciones.

## Ejercicios

1. ¿Por qué querrías utilizar la API `tf.data`?
2. ¿Cuáles son los beneficios de dividir un gran conjunto de datos en varios archivos?
3. Durante la capacitación, ¿cómo puede saber si su canal de entrada es el cuello de botella? ¿Qué puedes hacer para solucionarlo?
4. ¿Puede guardar datos binarios en un archivo `TFRecord` o solo en búferes de protocolo serializados?
5. ¿Por qué pasarías por la molestia de convertir todos tus datos al formato protobuf de ejemplo? ¿Por qué no utilizar su propia definición de protobuf?
6. Al usar `TFRecords`, ¿cuándo desearías activar la compresión? ¿Por qué no hacerlo sistemáticamente?
7. Los datos se pueden preprocesar directamente al escribir los archivos de datos, o dentro del proceso `tf.data`, o en capas de preprocesamiento dentro de su modelo. ¿Puedes enumerar algunos pros y contras de cada opción?
8. Mencione algunas formas comunes en las que puede codificar características de enteros categóricos. ¿Qué pasa con el texto?
9. Cargue el conjunto de datos Fashion MNIST (presentado en [el Capítulo 10](#)); dividirlo en un conjunto de entrenamiento, un conjunto de validación y un conjunto de prueba; barajar el conjunto de entrenamiento; y guarde cada conjunto de datos en múltiples `TFRecord`

archivos. Cada registro debe ser un protobuf de ejemplo serializado con dos características: la imagen serializada (use `tf.io.serialize_tensor()` para serializar cada imagen) y la etiqueta. Luego use `tf.data` para ~~10~~ crear un conjunto de datos eficiente para cada conjunto.

Finalmente, utilice un modelo Keras para entrenar estos conjuntos de datos, incluida una capa de preprocessamiento para estandarizar cada característica de entrada. Intente hacer que la canalización de entrada sea lo más eficiente posible utilizando TensorBoard para visualizar los datos de generación de perfiles.

10. En este ejercicio descargará un conjunto de datos, lo dividirá, creará un `tf.data.Dataset` para cargarlo y preprocessarlo de manera eficiente, luego construir y entrenar un modelo de clasificación binaria que contenga una capa de incrustación:
  - a. Descargue el [conjunto de datos de reseñas de películas grandes](#), que contiene 50.000 reseñas de películas de [Internet Movie Database \(IMDb\)](#). Los datos están organizados en dos directorios, train y test, cada uno de los cuales contiene un subdirectorio pos con 12.500 reseñas positivas y un subdirectorio neg con 12.500 reseñas negativas. Cada reseña se almacena en un archivo de texto separado. Hay otros archivos y carpetas (incluidas versiones preprocessadas de bolsas de palabras), pero los ignoraremos en este ejercicio.
  - b. Divida el conjunto de prueba en un conjunto de validación (15 000) y un conjunto de prueba (10.000).

C. Utilice `tf.data` para crear un conjunto de datos eficiente para cada conjunto.

  - d. Cree un modelo de clasificación binaria, utilizando una Capa `TextVectorization` para preprocessar cada revisión.
  - mi. Agregue una capa de incrustación y calcule la media incrustación para cada reseña, multiplicada por la raíz cuadrada del número de palabras (consulte [el Capítulo 16](#)). Esta incrustación media reescalada se puede pasar al resto de su modelo.

F. Entrene el modelo y vea qué precisión obtiene. Intentar optimizar sus canales para que la capacitación sea lo más rápida posible.

gramo. Utilice TFDS para cargar el mismo conjunto de datos más fácilmente: `tfds.load("imdb_reviews")`.

Las soluciones a estos ejercicios están disponibles al final del cuaderno de este capítulo, en <https://homl.info/colab3>.

---

**1** Imagine una baraja de cartas ordenada a su izquierda: suponga que simplemente toma la parte superior Tres cartas y barájala, luego elige una al azar y colócala a tu derecha, manteniendo las otras dos en tus manos. Toma otra carta a tu izquierda, baraja las tres cartas que tienes en la mano, elige una de ellas al azar y colócala a tu derecha. Cuando hayas terminado de revisar todas las cartas de esta manera, tendrás una baraja de cartas a tu derecha: ¿crees que quedará perfectamente barajada?

**2** En general, está bien capturar previamente un lote, pero en algunos casos es posible que necesites capturar previamente algunos más. Alternativamente, puedes dejar que TensorFlow decida automáticamente pasando `tf.data.AUTOTUNE` a `prefetch()`.

**3** Pero mira el experimento

Función `tf.data.experimental.prefetch_to_device()`, que puede precargar datos directamente a la GPU. Cualquier función o clase de TensorFlow con `experimental` en su nombre puede cambiar sin previo aviso en versiones futuras.

Si una función experimental falla, intente eliminar la palabra `experimental`: es posible que se haya movido a la API principal. De lo contrario, revise el cuaderno, ya que me aseguraré de que contenga el código actualizado.

**4** Dado que los objetos protobuf están destinados a ser serializados y transmitidos, se denominan mensajes.

**5** Este capítulo contiene lo mínimo que necesita saber sobre protobuffs para utilizar TFRecords. Para obtener más información sobre protobuffs, visite <https://homl.info/protobuf>.

**6** Tomáš Mikolov et al., “Representaciones distribuidas de palabras y frases y su composición”, Actas de la 26<sup>a</sup> Conferencia Internacional sobre Sistemas de Procesamiento de Información Neural 2 (2013): 3111–3119.

**7** Malvina Nissim et al., “Lo justo es mejor que lo sensacional: el hombre es médico como La mujer es doctora”, preimpresión de arXiv arXiv:1905.09866 (2019).

**8** TensorFlow Hub no está incluido con TensorFlow, pero si está ejecutando Colab o si siguió las instrucciones de instalación en <https://homl.info/install>,

entonces ya está instalado.

- 9 Para ser precisos, la incrustación de la oración es igual a la palabra media incrustación multiplicada por la raíz cuadrada del número de palabras en la oración. Esto compensa el hecho de que la media de  $n$  vectores aleatorios se acorta a medida que  $n$  crece.
- 10 Para imágenes grandes, puedes usar `tf.io.encode_jpeg()` en su lugar. Esto ahorraría mucho espacio, pero perdería un poco de calidad de imagen.

# Capítulo 14. Visión informática profunda utilizando redes neuronales convolucionales

---

Aunque la supercomputadora Deep Blue de IBM venció al campeón mundial de ajedrez Garry Kasparov en 1996, no fue hasta hace poco que las computadoras pudieron realizar de manera confiable tareas aparentemente triviales, como detectar un cachorro en una imagen o reconocer palabras habladas. ¿Por qué estas tareas nos resultan tan sencillas a los humanos? La respuesta está en el hecho de que la percepción tiene lugar en gran medida fuera del ámbito de nuestra conciencia, dentro de módulos visuales, auditivos y otros módulos sensoriales especializados en nuestro cerebro. Cuando la información sensorial llega a nuestra conciencia, ya está adornada con características de alto nivel; por ejemplo, cuando miras una foto de un lindo cachorro, no puedes elegir no verlo, no notar su ternura. Tampoco puedes explicar cómo se reconoce a un lindo cachorro; es simplemente obvio para ti. Así, no podemos confiar en nuestra experiencia subjetiva: la percepción no es nada trivial, y para comprenderla debemos fijarnos en cómo funcionan nuestros módulos sensoriales.

Las redes neuronales convolucionales (CNN) surgieron del estudio de la corteza visual del cerebro y se han utilizado en el reconocimiento de imágenes por computadora desde la década de 1980. Durante los últimos 10 años, gracias al aumento de la potencia computacional, la cantidad de datos de entrenamiento disponibles y los trucos presentados en el [Capítulo 11](#) para entrenar redes profundas, las CNN han logrado lograr un rendimiento sobrehumano en algunas tareas visuales complejas. Impulsan servicios de búsqueda de imágenes, vehículos autónomos, sistemas automáticos de clasificación de videos y más. Además, las CNN no se limitan a la percepción visual: también tienen éxito en muchas otras tareas, como el reconocimiento de voz y el procesamiento del lenguaje natural. Sin embargo, por ahora nos centraremos en las aplicaciones visuales.

En este capítulo exploraremos de dónde provienen las CNN, cómo son sus componentes básicos y cómo implementarlos utilizando Keras. Luego discutiremos algunas de las mejores arquitecturas de CNN, así como otras tareas visuales, incluida la detección de objetos (clasificar múltiples objetos en una imagen y

colocar cuadros delimitadores a su alrededor) y segmentación semántica (clasificar cada píxel según la clase del objeto al que pertenece).

## La arquitectura de la corteza visual

David H. Hubel y Torsten Wiesel realizaron una serie de experimentos con gatos en 1958. y 1959 (y unos años más tarde, los monos), que aporta información crucial sobre la estructura de la corteza visual (los autores recibieron el Premio Nobel de Fisiología o Medicina en 1981 por su trabajo). En particular, demostraron que muchas neuronas de la corteza visual tienen un pequeño campo receptivo local, lo que significa que reaccionan sólo a estímulos visuales ubicados en una región limitada del campo visual (ver Figura 14-1, en la que los campos receptivos locales de cinco neuronas) las neuronas están representadas por círculos discontinuos). Los campos receptivos de diferentes neuronas pueden superponerse y juntos forman todo el campo visual.

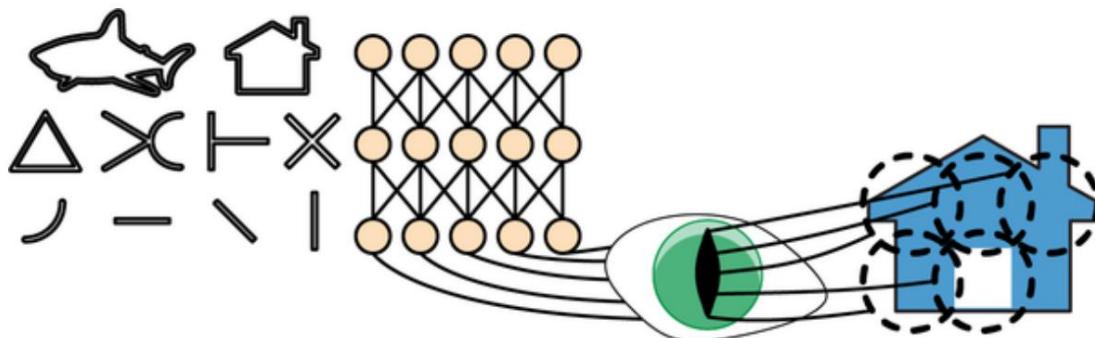


Figura 14-1. Las neuronas biológicas de la corteza visual responden a patrones específicos en pequeñas regiones del campo visual llamadas campos receptivos; A medida que la señal visual recorre módulos cerebrales consecutivos, las neuronas responden a patrones más complejos en campos receptivos más grandes.

Además, los autores demostraron que algunas neuronas reaccionan sólo a imágenes de líneas horizontales, mientras que otras reaccionan sólo a líneas con diferentes orientaciones (dos neuronas pueden tener el mismo campo receptivo pero reaccionar a diferentes orientaciones de líneas). También notaron que algunas neuronas tienen campos receptivos más grandes y reaccionan a patrones más complejos que son combinaciones de patrones de nivel inferior. Estas observaciones llevaron a la idea de que las neuronas de nivel superior se basan en las salidas de las neuronas vecinas de nivel inferior (en la figura 14-1, observe que cada neurona está conectada sólo a neuronas cercanas de la capa anterior). Esta poderosa arquitectura es capaz de detectar todo tipo de patrones complejos en cualquier área del campo visual.

Estos estudios de la corteza visual inspiraron el [neocognitrón](#), introducido en 1980, que gradualmente evolucionó hasta convertirse en lo que ahora llamamos redes neuronales convolucionales. Un hito importante fue un [artículo de 1998](#), por Yann LeCun et al. que introdujo la famosa arquitectura LeNet-5<sup>4</sup>, que fue ampliamente utilizada por los bancos para reconocer dígitos escritos a mano en cheques. Esta arquitectura tiene algunos componentes básicos que ya conoce, como capas completamente conectadas y funciones de activación sigmoidea, pero también introduce dos nuevos componentes básicos: capas convolucionales y capas de agrupación. Miremoslos ahora.

### NOTA

¿Por qué no utilizar simplemente una red neuronal profunda con capas completamente conectadas para tareas de reconocimiento de imágenes? Desafortunadamente, aunque esto funciona bien para imágenes pequeñas (por ejemplo, MNIST), falla para imágenes más grandes debido a la gran cantidad de parámetros que requiere. Por ejemplo, una imagen de  $100 \times 100$  píxeles tiene 10.000 píxeles, y si la primera capa tiene sólo 1.000 neuronas (lo que ya restringe severamente la cantidad de información transmitida a la siguiente capa), esto significa un total de 10 millones de conexiones. Y esa es sólo la primera capa. Las CNN resuelven este problema utilizando capas parcialmente conectadas y reparto de peso.

## Capas convolucionales

El componente más importante de una CNN es la capa convolucional: las neuronas de la primera capa convolucional no están conectadas a cada píxel de la imagen de entrada (como lo estaban en las capas analizadas en capítulos anteriores), sino sólo a los píxeles de su capa receptiva. (ver [Figura 14-2](#)). A su vez, cada neurona de la segunda capa convolucional está conectada sólo a neuronas ubicadas dentro de un pequeño rectángulo en la primera capa. Esta arquitectura permite que la red se centre en pequeñas características de bajo nivel en la primera capa oculta, luego las ensamble en características más grandes de nivel superior en la siguiente capa oculta, y así sucesivamente. Esta estructura jerárquica es común en imágenes del mundo real, lo cual es una de las razones por las que las CNN funcionan tan bien para el reconocimiento de imágenes.

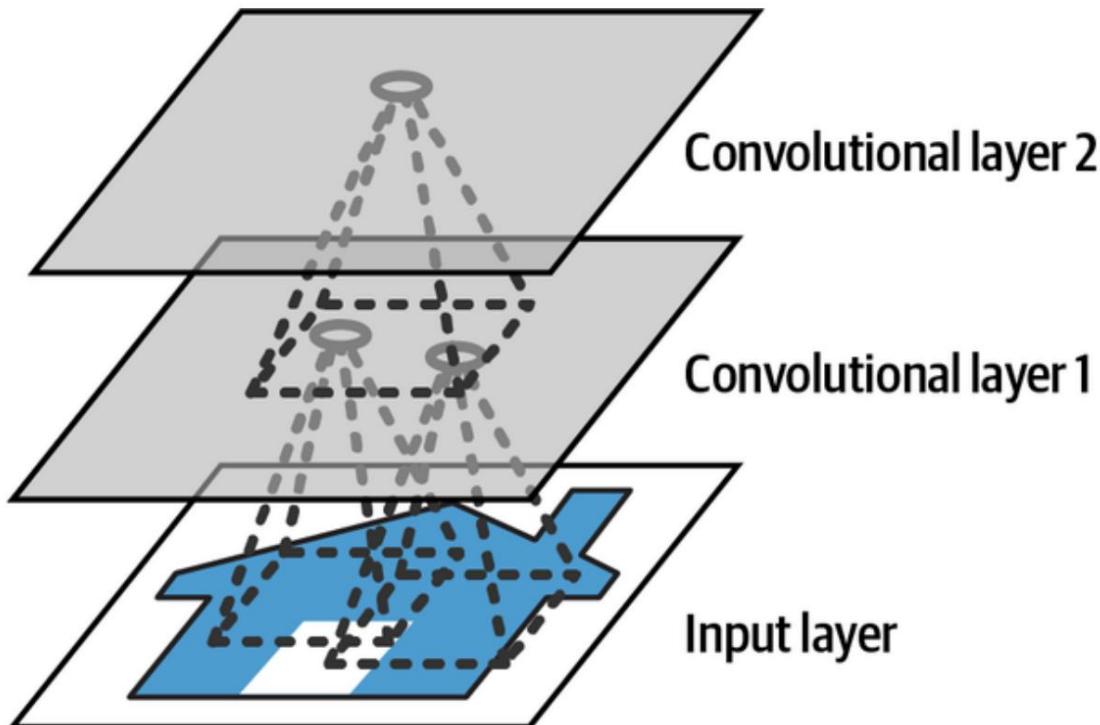


Figura 14-2. Capas CNN con campos receptivos locales rectangulares

### NOTA

Todas las redes neuronales multicapa que hemos analizado hasta ahora tenían capas compuestas por una larga línea de neuronas, y tuvimos que aplanar las imágenes de entrada a 1D antes de alimentarlas a la red neuronal. En una CNN cada capa está representada en 2D, lo que facilita la comparación de neuronas con sus correspondientes entradas.

Una neurona ubicada en la fila  $i$ , columna  $j$  de una capa determinada está conectada a las salidas de las neuronas de la capa anterior ubicadas en las filas  $i$  a  $i + f - 1$ , columnas  $j$  a  $j + f - 1$ , donde  $f$  es la altura y el ancho del campo receptivo (ver [Figura 14-3](#)). Para que una capa tenga la misma altura y ancho que la capa anterior, es común agregar ceros alrededor de las entradas, como se muestra en el diagrama. Esto se llama relleno cero.

También es posible conectar una capa de entrada grande a una capa mucho más pequeña espaciando los campos receptivos, como se muestra en la [Figura 14-4](#). Esto reduce drásticamente la complejidad computacional del modelo. El tamaño del paso horizontal o vertical de un campo receptivo al siguiente se llama zancada. En el diagrama, una capa de entrada de  $5 \times 7$  (más relleno de ceros) está conectada a una capa de  $3 \times 4$ , utilizando campos receptivos de  $3 \times 3$  y una zancada de 2 (en este ejemplo, la

la zancada es la misma en ambos sentidos, pero no tiene por qué ser así). A La neurona ubicada en la fila  $i$ , la columna  $j$  en la capa superior está conectada a las salidas de las neuronas de la capa anterior ubicadas en las filas  $i \times s_w + f_h - 1$ , columnas  $j \times s_w + f_w - 1$ , donde  $s_w$  y  $f_w$  son la vertical y zancadas horizontales.

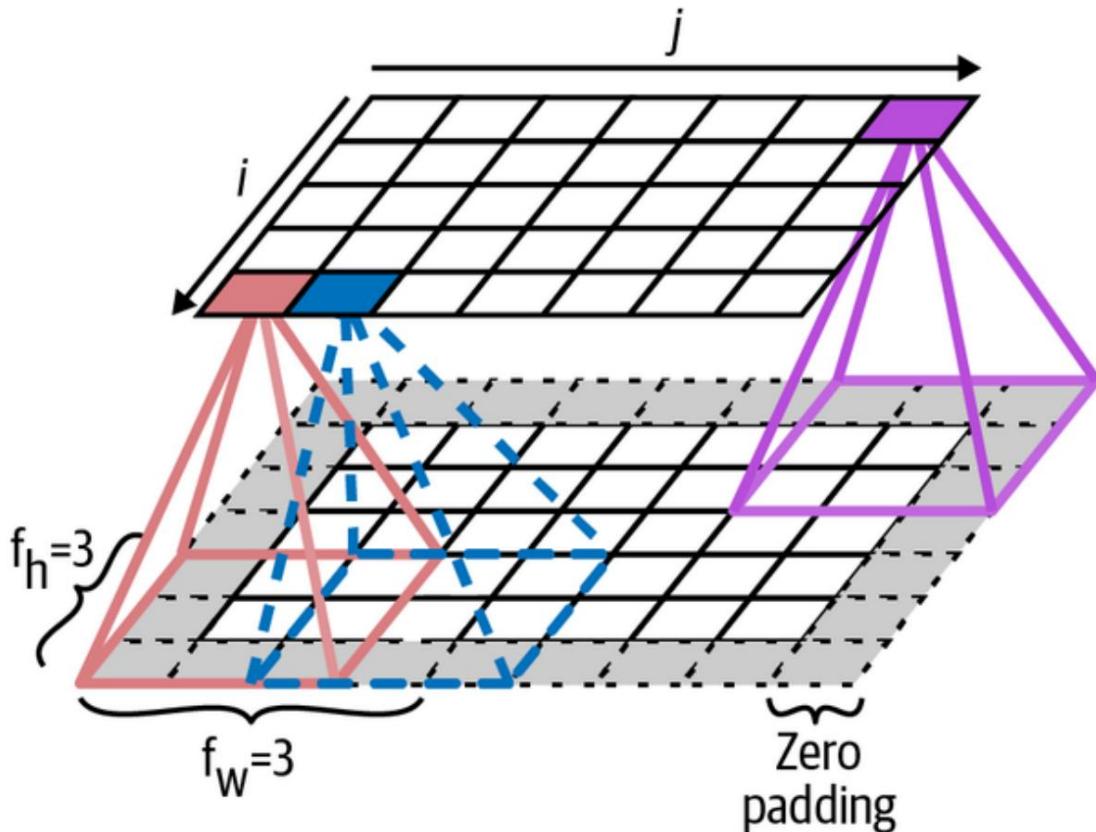


Figura 14-3. Conexiones entre capas y relleno cero.

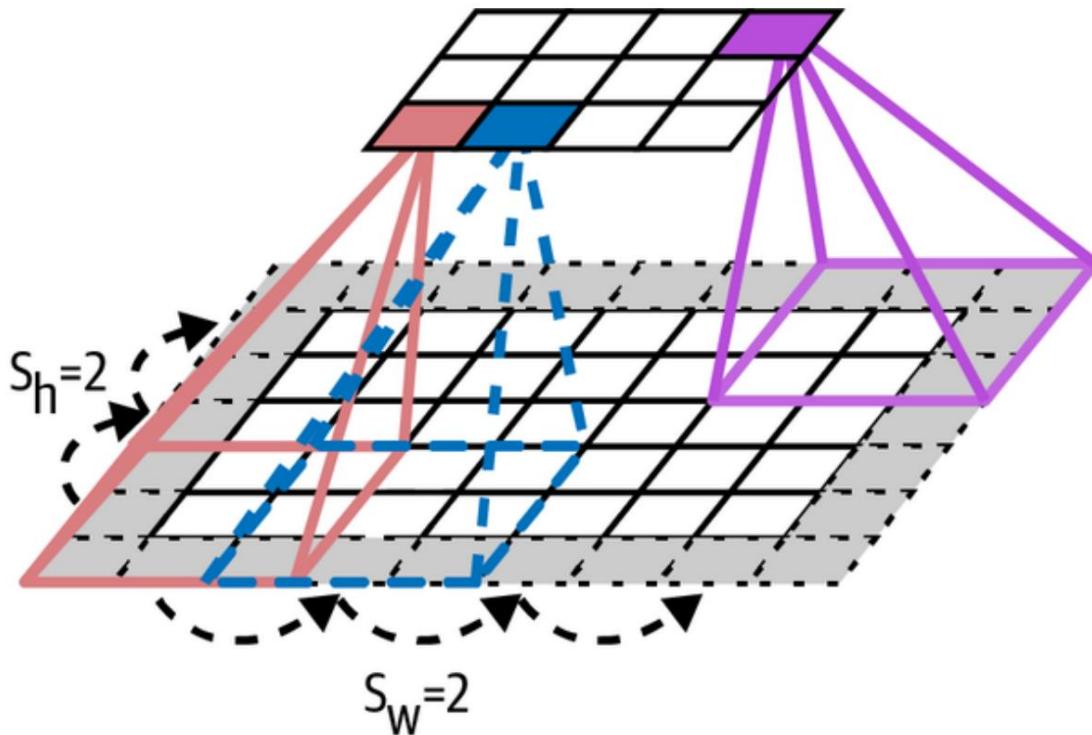


Figura 14-4. Reducir la dimensionalidad usando una zancada de 2

## Filtros

Los pesos de una neurona se pueden representar como una pequeña imagen del tamaño del campo receptivo. Por ejemplo, la Figura 14-5 muestra dos posibles conjuntos de pesos, llamados filtros (o núcleos de convolución, o simplemente núcleos). El primero se representa como un cuadrado negro con una línea blanca vertical en el medio (es una matriz de  $7 \times 7$  llena de ceros excepto la columna central, que está llena de unos); las neuronas que usan estos pesos ignorarán todo lo que esté en su campo receptivo excepto la línea vertical central (ya que todas las entradas se multiplicarán por 0, excepto las de la línea vertical central). El segundo filtro es un cuadrado negro con una línea blanca horizontal en el medio. Las neuronas que utilizan estos pesos ignorarán todo lo que se encuentre en su campo receptivo excepto la línea horizontal central.

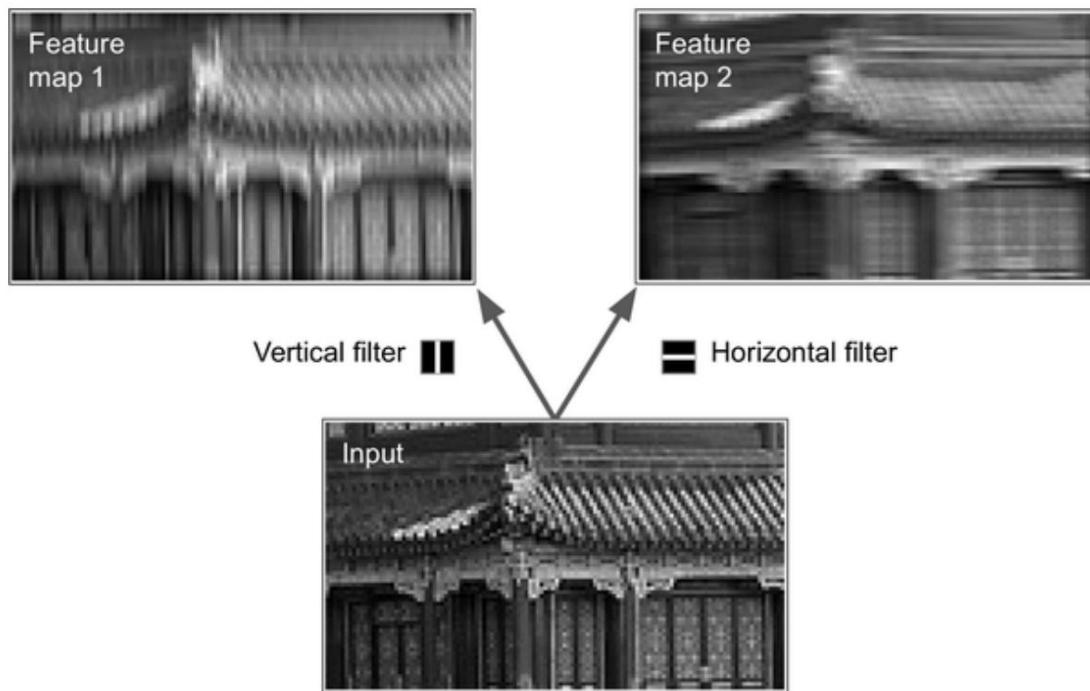


Figura 14-5. Aplicar dos filtros diferentes para obtener dos mapas de características

Ahora, si todas las neuronas en una capa usan el mismo filtro de línea vertical (y el mismo término de sesgo), y usted alimenta a la red con la imagen de entrada que se muestra en la [Figura 14-5](#) (la imagen inferior), la capa generará la imagen superior izquierda. .

Observe que las líneas blancas verticales se realzan mientras que el resto se difumina. De manera similar, la imagen superior derecha es la que se obtiene si todas las neuronas usan el mismo filtro de línea horizontal; observe que las líneas blancas horizontales se realzan mientras que el resto se difumina. Por lo tanto, una capa llena de neuronas que utiliza el mismo filtro genera un mapa de características, que resalta las áreas de una imagen que activan más el filtro. Pero no se preocupe, no tendrá que definir los filtros manualmente: en cambio, durante el entrenamiento, la capa convolucional aprenderá automáticamente los filtros más útiles para su tarea y las capas superiores aprenderán a combinarlos en patrones más complejos.

### Apilamiento de múltiples mapas de características

Hasta ahora, por simplicidad, he representado la salida de cada capa convolucional como una capa 2D, pero en realidad una capa convolucional tiene múltiples filtros (usted decide cuántos) y genera un mapa de características por filtro, por lo que se representa con mayor precisión. en 3D (ver [Figura 14-6](#)). Tiene una neurona por píxel en cada mapa de características, y todas las neuronas dentro de un mapa de características determinado comparten los mismos parámetros (es decir, el mismo núcleo y sesgo)

término). Las neuronas en diferentes mapas de características utilizan diferentes parámetros. El campo receptivo de una neurona es el mismo que se describió anteriormente, pero se extiende a través de todos los mapas de características de la capa anterior. En resumen, una capa convolucional aplica simultáneamente múltiples filtros entrenables a sus entradas, lo que la hace capaz de detectar múltiples características en cualquier lugar de sus entradas.

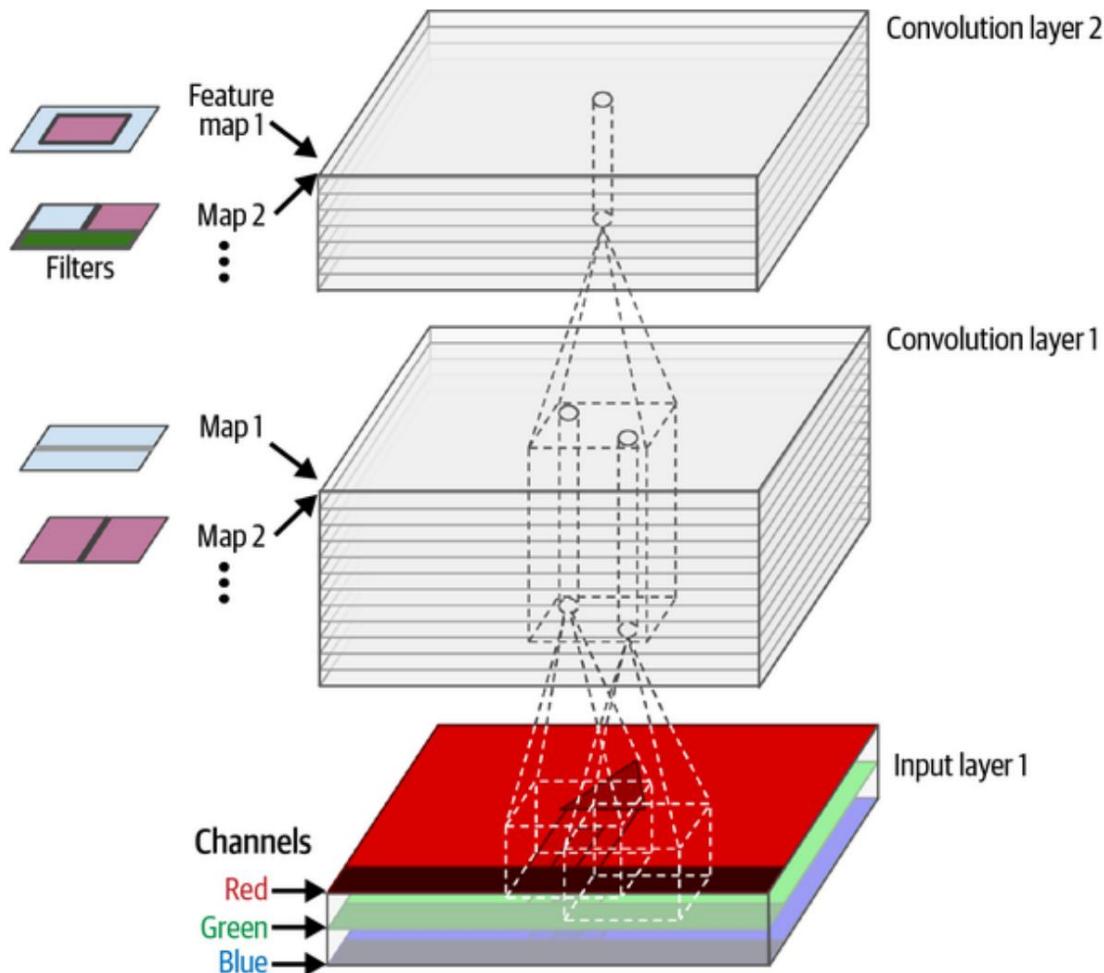


Figura 14-6. Dos capas convolucionales con múltiples filtros cada una (kernels), que procesan una imagen en color con tres canales de color; cada capa convolucional genera un mapa de características por filtro

## NOTA

El hecho de que todas las neuronas en un mapa de características comparten los mismos parámetros reduce drásticamente la cantidad de parámetros en el modelo. Una vez que la CNN ha aprendido a reconocer un patrón en un lugar, puede reconocerlo en cualquier otro lugar. Por el contrario, una vez que una red neuronal completamente conectada ha aprendido a reconocer un patrón en una ubicación, sólo puede reconocerlo en esa ubicación en particular.

Las imágenes de entrada también se componen de varias subcapas: una por canal de color. Como se mencionó en [el Capítulo 9](#), normalmente hay tres: rojo, verde y azul (RGB). Las imágenes en escala de grises tienen un solo canal, pero algunas imágenes pueden tener muchos más; por ejemplo, imágenes de satélite que capturan frecuencias de luz adicionales (como las infrarrojas).

Específicamente, una neurona ubicada en la fila  $i$ , columna  $j$  del mapa de características  $k$  en una capa convolucional dada  $l$  está conectada a las salidas de las neuronas en la capa anterior  $l - 1$ , ubicada en las filas  $i \times s$  a  $i \times s + f - 1$  y columnas  $j \times s$  a  $j \times s + f - 1$ , en todos los mapas de características (en la capa  $l - 1$ ). Tenga en cuenta que, dentro de una capa, todas las neuronas ubicadas en la misma fila  $i$  y columna  $j$  pero en diferentes mapas de características están conectadas a las salidas de exactamente las mismas neuronas en la capa anterior.

[La ecuación 14-1](#) resume las explicaciones anteriores en una gran ecuación matemática: muestra cómo calcular la salida de una neurona determinada en una capa convolucional. Es un poco feo debido a los diferentes índices, pero todo lo que hace es calcular la suma ponderada de todas las entradas, más el término de sesgo.

Ecuación 14-1. Calcular la salida de una neurona en una capa convolucional

$$z_{i,j,k} = b_k + \sum_{u=0}^{fh-1} \sum_{v=0}^{fw-1} \sum_{k'=0}^{fn'-1} x_{i',j',k'} \times w_{u,v,k',k} \quad i' = i \times sh + u \quad j' = j \times sw + v$$

En esta ecuación:

- $z_{i,j,k}$  es la salida de la neurona ubicada en la fila  $i$ , columna  $j$  en la característica  $k$  mapa  $k$  de la capa convolucional (capa  $l$ ).

- Como se explicó anteriormente,  $s$  y  $s'$  son los pasos verticales y horizontales,  $f$  y  $f'$  son la altura y el ancho del campo receptivo y  $f$  es el número de mapas de características  $k'$  en la capa anterior (capa  $l - 1$ ).
- $x_{i', j', k'}$  es la salida de la neurona ubicada en la capa  $l - 1$ , fila  $i'$ , columna  $j'$ , mapa de características  $k'$  (o canal  $k'$  si la capa anterior es la capa de entrada).
- $b_k$  es el término de sesgo para el mapa de características  $k$  (en la capa  $l$ ). Puedes considerarlo como una perilla que ajusta el brillo general del mapa de características  $k$ .
- $w_{u, v, k', k}$  es el peso de conexión entre cualquier neurona en el mapa de características  $k'$  de la capa  $l - 1$  y su entrada ubicada en la fila  $u$ , columna  $v$  (en relación con el campo receptivo de la neurona) y mapa de características  $k$ .

Veamos cómo crear y usar una capa convolucional usando Keras.

## Implementación de capas convolucionales con Keras

Primero, carguemos y preprocesemos un par de imágenes de muestra, usando Scikit-Learn la función `load_sample_image()` de Learn y `CenterCrop` de Keras y Cambio de escala de capas (todas las cuales se introdujeron en el Capítulo 13):

```
desde sklearn.datasets importar load_sample_images importar tensorflow
como tf

imágenes = load_sample_images()["imágenes"] imágenes
= tf.keras.layers.CenterCrop(alto=70, ancho=120)(imágenes) imágenes =
tf.keras.layers.Rescaling(escala=1 / 255)(imágenes)
```

Veamos la forma del tensor de imágenes:

```
>>> imágenes.forma
TensorShape([2, 70, 120, 3])
```

Vaya, es un tensor 4D; ¡No hemos visto esto antes! ¿Qué significan todas estas dimensiones? Bueno, hay dos imágenes de muestra, lo que explica la primera dimensión. Entonces cada imagen es de  $70 \times 120$ , ya que ese es el tamaño que especificamos al crear la capa `CenterCrop` (las imágenes originales eran de  $427 \times 640$ ). Esto explica la segunda y tercera dimensión. Y, por último, cada píxel tiene un valor por canal de color, y hay tres (rojo, verde y azul), lo que explica la última dimensión.

Ahora creemos una capa convolucional 2D y alimentémosla con estas imágenes para ver qué sale. Para ello, Keras proporciona una capa Convolution2D, alias Conv2D. En esencia, esta capa se basa en la operación `tf.nn.conv2d()` de TensorFlow. Creemos una capa convolucional con 32 filtros, cada uno de tamaño  $7 \times 7$  (usando `kernel_size=7`, que es equivalente a usar `kernel_size=(7, 7)`), y apliquemos esta capa a nuestro pequeño lote de dos imágenes:

```
conv_layer = tf.keras.layers.Conv2D(filtros=32, kernel_size=7) fmaps = conv_layer(imágenes)
```

### NOTA

Cuando hablamos de una capa convolucional 2D, “2D” se refiere al número de dimensiones espaciales (alto y ancho), pero como puedes ver, la capa toma entradas 4D: como vimos, las dos dimensiones adicionales son el tamaño del lote ( primera dimensión) y los canales (última dimensión).

Ahora veamos la forma de la salida:

```
>>> fmaps.shape  
TensorShape([2, 64, 114, 32])
```

La forma de salida es similar a la forma de entrada, con dos diferencias principales. Primero, hay 32 canales en lugar de 3. Esto se debe a que configuramos `filtros = 32`, por lo que obtenemos 32 mapas de características de salida: en lugar de la intensidad de rojo, verde y azul en cada ubicación, ahora tenemos la intensidad de cada característica. en cada ubicación. En segundo lugar, la altura y el ancho se han reducido en 6 píxeles. Esto se debe al hecho de que la capa Conv2D no utiliza ningún relleno con ceros de forma predeterminada, lo que significa que perdemos algunos píxeles en los lados de los mapas de características de salida, dependiendo del tamaño de los filtros. En este caso, dado que el tamaño del kernel es 7, perdemos 6 píxeles horizontalmente y 6 píxeles verticalmente (es decir, 3 píxeles en cada lado).

### ADVERTENCIA

Sorprendentemente, la opción predeterminada se llama padding="valid", lo que en realidad significa que no hay ningún relleno con ceros. Este nombre proviene del hecho de que en este caso el campo receptivo de cada neurona se encuentra estrictamente dentro de las posiciones válidas dentro de la entrada (no sale de los límites). No es una peculiaridad de los nombres de Keras: todo el mundo usa esta nomenclatura extraña.

Si en lugar de eso configuramos padding="same", entonces las entradas se llenan con suficientes ceros en todos los lados para garantizar que los mapas de características de salida terminen con el mismo tamaño que las entradas (de ahí el nombre de esta opción):

```
>>> conv_layer = tf.keras.layers.Conv2D(filtros=32, kernel_size=7,  
...                                         relleno="igual")  
...  
>>> fmaps = conv_layer(imágenes) >>>  
fmaps.shape  
TensorShape([2, 70, 120, 32])
```

Estas dos opciones de relleno se ilustran en [la Figura 14-7](#). Para simplificar, aquí solo se muestra la dimensión horizontal, pero, por supuesto, la misma lógica se aplica también a la dimensión vertical.

Si la zancada es mayor que 1 (en cualquier dirección), entonces el tamaño de salida no será igual al tamaño de entrada, incluso si padding="same". Por ejemplo, si establece pasos = 2 (o equivalentemente pasos = (2, 2)), entonces los mapas de características de salida serán  $35 \times 60$ : reducido a la mitad tanto vertical como horizontalmente.

[La Figura 14-8](#) muestra lo que sucede cuando zancadas = 2, con ambas opciones de relleno.

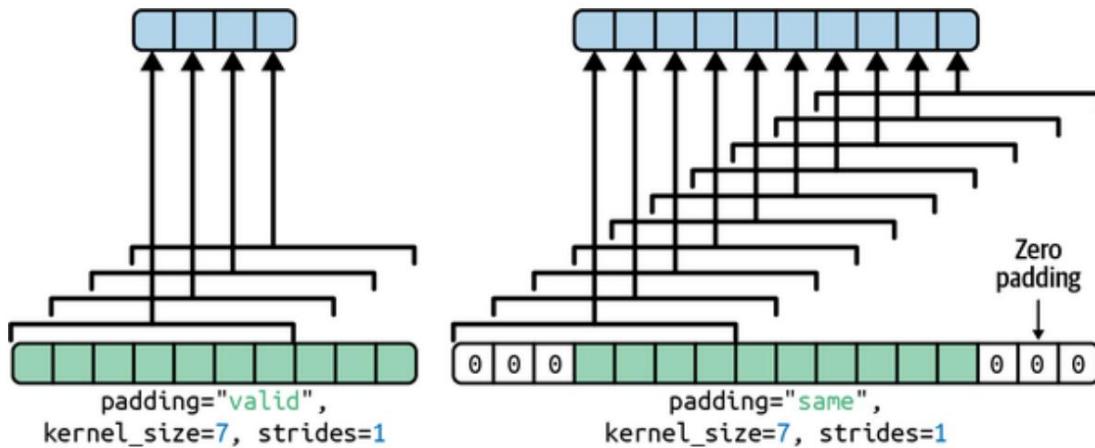


Figura 14-7. Las dos opciones de relleno, cuando zancadas = 1

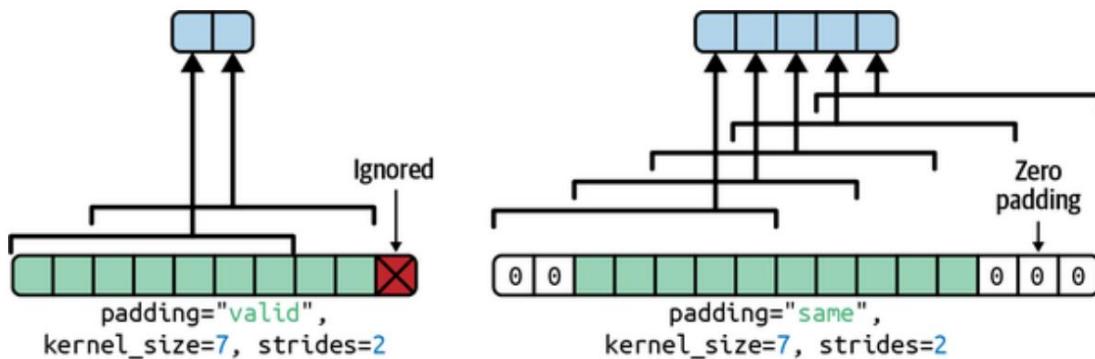


Figura 14-8. Con zancadas mayores que 1, el resultado es mucho menor incluso cuando se usa "mismo" relleno (y el relleno "válido" puede ignorar algunas entradas)

Si tiene curiosidad, así es como se calcula el tamaño de salida:

- Con  $\text{padding}=\text{"valid"}$ , si el ancho de la entrada es  $i$  y el ancho del núcleo es  $f$ , y la zancada horizontal es  $s$ , entonces la salida es igual a  $(i - f + s) / s$  arriba, redondeado hacia abajo. Recuerde que  $i$  es el ancho de la imagen de entrada y  $s$  es la zancada horizontal. Cualquier resto en la división corresponde a las columnas ignoradas en el lado derecho de la imagen. Se puede utilizar la misma lógica para calcular la altura de salida, y cualquier fila ignorada en la parte inferior de la imagen.
- Con  $\text{padding}=\text{"same"}$ , el ancho de salida es igual a  $i / s$  arriba. Para que sea posible, se utiliza el número apropiado de columnas cero. Rellenando a la izquierda y a la derecha de la imagen de entrada (un número igual si es posible, o sólo uno más en el lado derecho). Suponiendo el ancho de salida es  $o_w$ , entonces el número de columnas con ceros llenos es  $(o_w - i) \times s + f - i$ . Nuevamente, se puede usar la misma lógica para calcular la altura de salida y el número de filas acolchadas.

Ahora veamos los pesos de la capa (que se anotaron  $w$  y  $b$  en  $u$ ,  $v$ ,  $k'$ )<sup>14</sup>.  
 Al igual que una capa Densa, una capa Conv2D contiene todos los pesos de la capa, incluidos los núcleos y los sesgos. Los núcleos se inicializan aleatoriamente, mientras que los sesgos se inicializan a cero. Se puede acceder a estos pesos como variables TF a través del atributo de pesos, o como matrices NumPy a través del método `get_weights()`:

```
>>> núcleos, sesgos = conv_layer.get_weights() >>>
núcleos.shape (7, 7, 3,
32) >>>
sesgos.shape (32,)
```

La matriz de kernels es 4D y su forma es [kernel\_height, kernel\_width, input\_channels, output\_channels]. La matriz de sesgos es 1D, con forma [output\_channels]. La cantidad de canales de salida es igual a la cantidad de mapas de características de salida, que también es igual a la cantidad de filtros.

Lo más importante es tener en cuenta que la altura y el ancho de las imágenes de entrada no aparecen en la forma del núcleo: esto se debe a que todas las neuronas en los mapas de características de salida comparten los mismos pesos, como se explicó anteriormente. Esto significa que puede alimentar imágenes de cualquier tamaño a esta capa, siempre que sean al menos tan grandes como los núcleos y tengan el número correcto de canales (tres en este caso).

Por último, generalmente querrás especificar una función de activación (como ReLU) al crear una capa Conv2D, y también especificar el inicializador del kernel correspondiente (como la inicialización He). Esto se debe a la misma razón que para las capas densas: una capa convolucional realiza una operación lineal, por lo que si apilara varias capas convolucionales sin ninguna función de activación, todas serían equivalentes a una sola capa convolucional y no podrían aprender. algo realmente complejo.

Como puede ver, las capas convolucionales tienen bastantes hiperparámetros: filtros, `kernel_size`, `padding`, `strides`, activación, `kernel_initializer`, etc. Como siempre, puede usar la validación cruzada para encontrar los valores de hiperparámetro correctos, pero esto lleva mucho tiempo. . Analizaremos las arquitecturas CNN comunes más adelante en este capítulo, para darle una idea de qué valores de hiperparámetros funcionan mejor en la práctica.

## Requisitos de memoria Otro desafío

de las CNN es que las capas convolucionales requieren una enorme cantidad de RAM. Esto es especialmente cierto durante el entrenamiento, porque el paso inverso de la propagación hacia atrás requiere todos los valores intermedios calculados durante el paso hacia adelante.

Por ejemplo, considere una capa convolucional con 200 filtros de  $5 \times 5$ , con paso 1 y "mismo" relleno. Si la entrada es una imagen RGB de  $150 \times 100$  (tres canales), entonces el número de parámetros es  $(5 \times 5 \times 3 + 1) \times 200 = 15\,200$  (el + 1 corresponde a los términos de sesgo), lo cual es bastante pequeño en comparación. a una capa completamente conectada. Sin embargo, <sup>7</sup> cada uno de los 200 mapas de características contiene  $150 \times 100$  neuronas, y cada una de estas neuronas necesita calcular una suma ponderada de sus  $5 \times 5 \times 3 = 75$  entradas: eso es un total de 225 millones de multiplicaciones flotantes. No es tan malo como una capa totalmente conectada, pero sigue siendo bastante intensivo desde el punto de vista computacional. Además, si los mapas de características se representan utilizando flotantes de 32 bits, entonces la salida de la capa convolucional ocupará  $200 \times 150 \times 100 \times 32 = 96$  millones <sup>8</sup> de bits (12 MB) de RAM. Y eso es solo para una instancia: si un lote de entrenamiento contiene 100 instancias, ¡esta capa consumirá 1,2 GB de RAM!

Durante la inferencia (es decir, al hacer una predicción para una nueva instancia), la RAM ocupada por una capa se puede liberar tan pronto como se haya calculado la siguiente capa, por lo que solo necesita tanta RAM como la que requieren dos capas consecutivas. Pero durante el entrenamiento, todo lo calculado durante el paso directo debe conservarse para el paso inverso, por lo que la cantidad de RAM necesaria es (al menos) la cantidad total de RAM requerida por todas las capas.

### CONSEJO

Si el entrenamiento falla debido a un error de falta de memoria, puede intentar reducir el tamaño del mini lote. Alternativamente, puede intentar reducir la dimensionalidad usando un paso, eliminando algunas capas, usando flotantes de 16 bits en lugar de flotantes de 32 bits, o distribuyendo la CNN entre múltiples dispositivos (verá cómo hacerlo en el Capítulo 19).

Ahora veamos el segundo componente común de las CNN: la capa de agrupación.

## Capas de agrupación

Una vez que comprenda cómo funcionan las capas convolucionales, las capas de agrupación son bastante fáciles de entender. Su objetivo es submuestrear (es decir, reducir) la imagen de entrada para reducir la carga computacional, el uso de memoria y la cantidad de parámetros (limitando así el riesgo de sobreajuste).

Al igual que en las capas convolucionales, cada neurona en una capa de agrupación está conectada a las salidas de un número limitado de neuronas en la capa anterior, ubicadas dentro de un pequeño campo receptivo rectangular. Debes definir su tamaño, la zancada y el tipo de acolchado, como antes. Sin embargo, una neurona de agrupación no tiene pesos; todo lo que hace es agregar las entradas usando una función de agregación como el máximo o la media. [La Figura 14-9](#) muestra una capa de agrupación máxima, que es el tipo más común de capa de agrupación. En este ejemplo, utilizamos un núcleo de agrupación<sup>9</sup> de  $2 \times 2$ , con un paso de 2 y sin relleno. Solo el valor de entrada máximo en cada campo receptivo pasa a la siguiente capa, mientras que las otras entradas se eliminan. Por ejemplo, en el campo receptivo inferior izquierdo de [la Figura 14-9](#), los valores de entrada son 1, 5, 3, 2, por lo que solo el valor máximo, 5, se propaga a la siguiente capa. Debido al paso de 2, la imagen de salida tiene la mitad de la altura y la mitad del ancho de la imagen de entrada (redondeada hacia abajo ya que no usamos relleno).

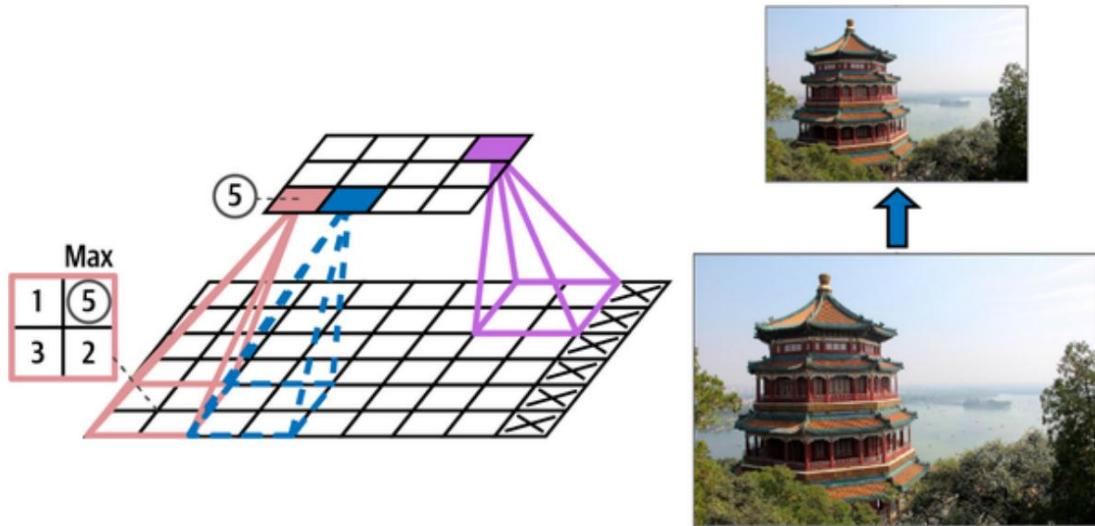


Figura 14-9. Capa de agrupación máxima (núcleo de agrupación  $2 \times 2$ , zancada 2, sin relleno)

## NOTA

Una capa de agrupación normalmente funciona en cada canal de entrada de forma independiente, por lo que la profundidad de salida (es decir, el número de canales) es la misma que la profundidad de entrada.

Además de reducir los cálculos, el uso de memoria y la cantidad de parámetros, una capa de agrupación máxima también introduce cierto nivel de invariancia para traducciones pequeñas, como se muestra en [la Figura 14-10](#). Aquí asumimos que los píxeles brillantes tienen un valor menor que los píxeles oscuros y consideramos tres imágenes (A, B, C) que pasan por una capa de agrupación máxima con un núcleo de  $2 \times 2$  y un paso de 2. Las imágenes B y C son iguales, como la imagen A, pero desplazada uno y dos píxeles hacia la derecha. Como puede ver, las salidas de la capa de agrupación máxima para las imágenes A y B son idénticas. Esto es lo que significa la invariancia de traducción. Para la imagen C, el resultado es diferente: se desplaza un píxel hacia la derecha (pero todavía hay un 50% de invariancia). Al insertar una capa de agrupación máxima cada pocas capas en una CNN, es posible obtener cierto nivel de invariancia de traducción a mayor escala. Además, la agrupación máxima ofrece una pequeña cantidad de invariancia rotacional y una ligera invariancia de escala. Dicha invariancia (incluso si es limitada) puede resultar útil en casos en los que la predicción no debería depender de estos detalles, como en tareas de clasificación.

Sin embargo, la agrupación máxima también tiene algunas desventajas. Obviamente es muy destructivo: incluso con un núcleo diminuto de  $2 \times 2$  y una zancada de 2, la salida será dos veces menor en ambas direcciones (por lo que su área será cuatro veces menor), simplemente eliminando el 75% de los valores de entrada. Y en algunas aplicaciones, la invariancia no es deseable. Tomemos como ejemplo la segmentación semántica (la tarea de clasificar cada píxel de una imagen según el objeto al que pertenece ese píxel, que exploraremos más adelante en este capítulo): obviamente, si la imagen de entrada se traslada un píxel a la derecha, la imagen de salida También debe traducirse un píxel a la derecha. El objetivo en este caso es la equivarianza, no la invariancia: un pequeño cambio en los insumos debería conducir a un pequeño cambio correspondiente en la salida.

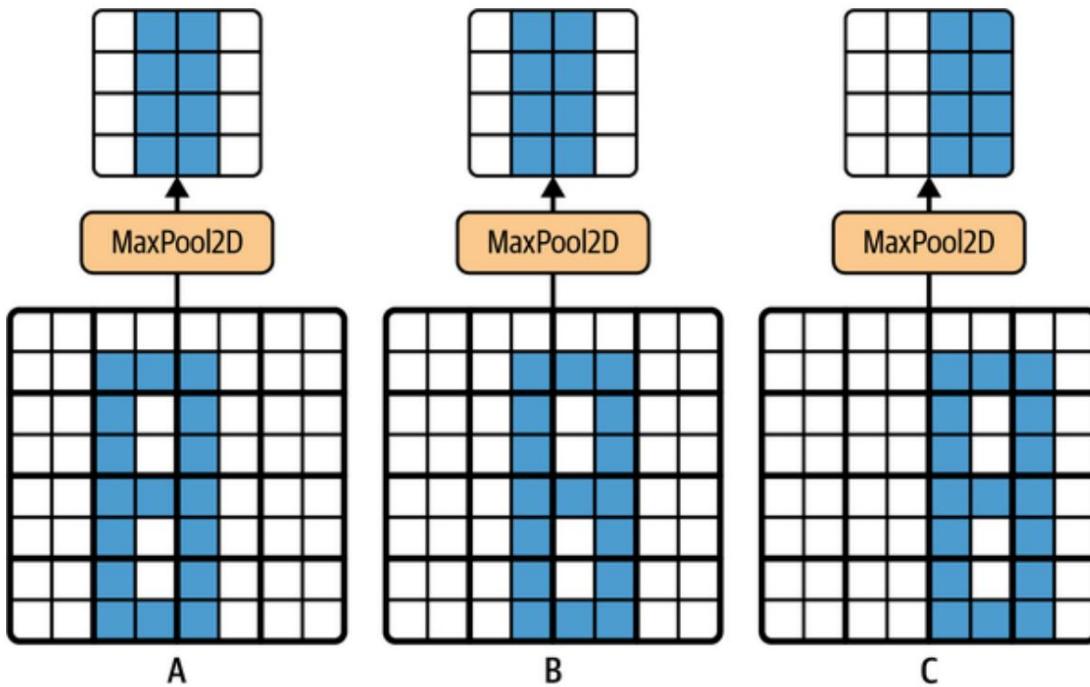


Figura 14-10. Invariancia a pequeñas traducciones

## Implementación de capas de agrupación con Keras

El siguiente código crea una capa MaxPooling2D, alias MaxPool2D, utilizando un kernel 2 × 2. Las zancadas tienen por defecto el tamaño del núcleo, por lo que esta capa usa una zancada de 2 (horizontal y verticalmente). De forma predeterminada, utiliza un relleno "válido" (es decir, sin ningún relleno):

```
max_pool = tf.keras.layers.MaxPool2D(tamaño_grupo=2)
```

Para crear una capa de agrupación promedio, simplemente use AveragePooling2D, alias AvgPool2D, en lugar de MaxPool2D. Como era de esperar, funciona exactamente como una capa de agrupación máxima, excepto que calcula la media en lugar del máximo.

Las capas de agrupación promedio solían ser muy populares, pero ahora la gente usa principalmente capas de agrupación máxima, ya que generalmente funcionan mejor.

Esto puede parecer sorprendente, ya que al calcular la media generalmente se pierde menos información que al calcular el máximo. Pero, por otro lado, la agrupación máxima conserva solo las características más potentes, eliminando todas las que no tienen sentido, de modo que las siguientes capas obtienen una señal más limpia con la que trabajar. Además, la agrupación máxima ofrece una invariancia de traducción más fuerte que la agrupación promedio y requiere un poco menos de procesamiento.

Tenga en cuenta que la agrupación máxima y la agrupación promedio se pueden realizar a lo largo de la dimensión de profundidad en lugar de las dimensiones espaciales, aunque no es tan común. Esto puede permitir que la CNN aprenda a ser invariante ante varias características. Por ejemplo, podría aprender varios filtros, cada uno de los cuales detectaría una rotación diferente del mismo patrón (como dígitos escritos a mano; consulte [la Figura 14-11](#)), y la capa de agrupación máxima en profundidad garantizaría que la salida sea la misma independientemente de la rotación. De manera similar, la CNN podría aprender a ser invariante ante cualquier cosa: grosor, brillo, sesgo, color, etc.

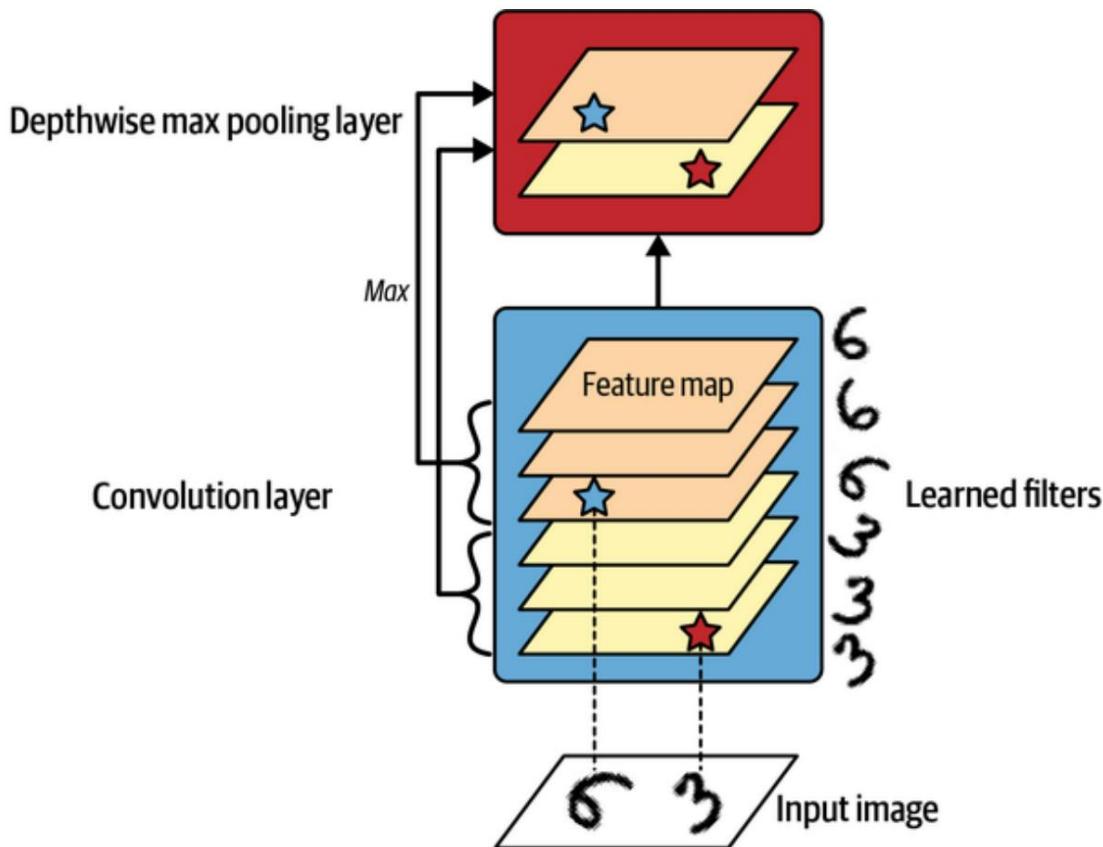


Figura 14-11. La agrupación máxima en profundidad puede ayudar a la CNN a aprender a ser invariante (a la rotación en este caso)

Keras no incluye una capa de agrupación máxima en profundidad, pero no es demasiado difícil implementar una capa personalizada para eso:

```
clase DepthPool (tf.keras.layers.Layer):
    def __init__(self, tamaño_piscina=2, **kwargs):
        super().__init__(**kwargs)
        self.tamaño_piscina = tamaño_piscina
```

llamada **def** (auto, entradas):

```

forma = tf.shape(entradas) # forma[-1] es el número de canales

grupos = forma[-1] // self.pool_size # número de canal
grupos

new_shape = tf.concat([forma[:-1], [grupos, self.pool_size]], eje=0)

devolver tf.reduce_max(tf.reshape(entradas, new_shape), eje=-1)

```

Esta capa reforma sus entradas para dividir los canales en grupos del tamaño deseado (pool\_size), luego usa tf.reduce\_max() para calcular el máximo de cada grupo. Esta implementación supone que la zancada es igual al tamaño de la piscina, que generalmente es lo que desea. Alternativamente, puede usar la operación tf.nn.max\_pool() de TensorFlow y envolverla en una capa Lambda para usarla dentro de un modelo Keras, pero lamentablemente esta operación no implementa la agrupación profunda para la GPU, solo para la CPU.

Un último tipo de capa de agrupación que verá a menudo en las arquitecturas modernas es la capa de agrupación promedio global. Funciona de manera muy diferente: todo lo que hace es calcular la media de cada mapa de características completo (es como una capa de agrupación promedio que utiliza un núcleo de agrupación con las mismas dimensiones espaciales que las entradas). Esto significa que solo genera un único número por mapa de características y por instancia. Aunque, por supuesto, esto es extremadamente destructivo (la mayor parte de la información en el mapa de características se pierde), puede ser útil justo antes de la capa de salida, como verá más adelante en este capítulo. Para crear dicha capa, simplemente use la clase GlobalAveragePooling2D, alias GlobalAvgPool2D:

```
global_avg_pool = tf.keras.layers.GlobalAvgPool2D()
```

Es equivalente a la siguiente capa Lambda, que calcula la media sobre las dimensiones espaciales (alto y ancho):

```

global_avg_pool = tf.keras.layers.Lambda(
    lambda X: tf.reduce_mean(X, eje=[1, 2]))

```

Por ejemplo, si aplicamos esta capa a las imágenes de entrada, obtenemos la intensidad media de rojo, verde y azul para cada imagen:

```

>>> global_avg_pool(imágenes) <tf.Tensor:
forma=(2, 3), dtype=float32, numpy=

```

```
matriz([[0.64338624, 0.5971759 [0.76306933, , 0.5824972],
       0.26011038, 0.10849128]], dtype=float32)>
```

Ahora conoce todos los componentes básicos para crear redes neuronales convolucionales. Veamos cómo montarlos.

## Arquitecturas CNN

Las arquitecturas típicas de CNN apilan algunas capas convolucionales (cada una generalmente seguida por una capa ReLU), luego una capa de agrupación, luego otras pocas capas convolucionales (+ReLU), luego otra capa de agrupación, y así sucesivamente. La imagen se hace cada vez más pequeña a medida que avanza a través de la red, pero también suele volverse cada vez más profunda (es decir, con más mapas de características), gracias a las capas convolucionales (consulte la Figura 14-12). En la parte superior de la pila, se agrega una red neuronal de avance regular, compuesta por algunas capas completamente conectadas (+ReLU), y la capa final genera la predicción (por ejemplo, una capa softmax que genera probabilidades de clase estimadas).

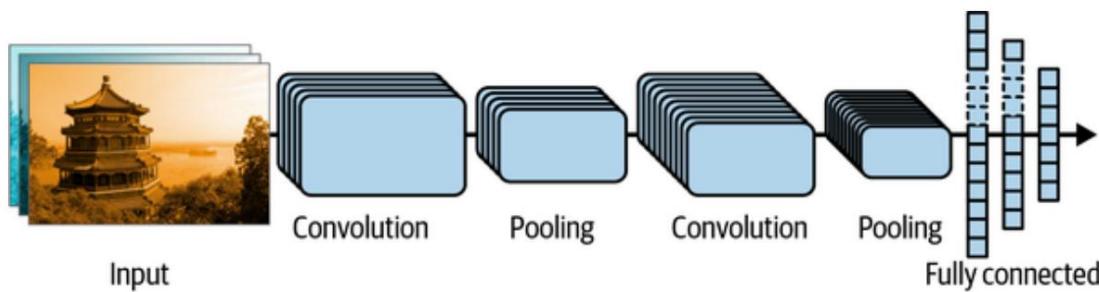


Figura 14-12. Arquitectura típica de CNN

### CONSEJO

Un error común es utilizar núcleos de convolución demasiado grandes. Por ejemplo, en lugar de utilizar una capa convolucional con un núcleo de  $5 \times 5$ , apile dos capas con núcleos de  $3 \times 3$ : utilizará menos parámetros y requerirá menos cálculos y, por lo general, funcionará mejor. Una excepción es la primera capa convolucional: normalmente puede tener un núcleo grande (p. ej.,  $5 \times 5$ ), normalmente con un paso de 2 o más. Esto reducirá la dimensión espacial de la imagen sin perder demasiada información y, dado que la imagen de entrada solo tiene tres canales en general, no será demasiado costoso.

Así es como se puede implementar una CNN básica para abordar el conjunto de datos Fashion MNIST (presentado en [el Capítulo 10](#)):

```
desde functools importar parcial

DefaultConv2D = parcial(tf.keras.layers.Conv2D, kernel_size=3, padding="igual",
                      activación="relu",
                      kernel_initializer="he_normal") modelo =
tf.keras.Sequential([
    DefaultConv2D(filtros=64, kernel_size=7, input_shape=[28, 28, 1]), tf.keras.layers.MaxPool2D(),
    DefaultConv2D(filtros=128), DefaultConv2D(filtros=128),
    tf.keras.layers .MaxPool2D(),
    DefaultConv2D(filtros=256),
    DefaultConv2D(filtros=256),
    tf.keras.layers.MaxPool2D(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(unidades=128 ,
                          activación="relu", kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(0.5), tf.keras.layers.Dense(units=64, activación="relu",
                                                       kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(unidades=10, activación="softmax")
])

])
```

Repasemos este código:

- Usamos la función `functools.partial()` (introducida en [el Capítulo 11](#)) para definir `DefaultConv2D`, que actúa igual que `Conv2D` pero con diferentes argumentos predeterminados: un tamaño de kernel pequeño de 3, "mismo" relleno, la función de activación ReLU y su correspondiente Él inicializador.
- A continuación, creamos el modelo secuencial. Su primera capa es un `DefaultConv2D` con 64 filtros bastante grandes ( $7 \times 7$ ). Utiliza el paso predeterminado de 1 porque las imágenes de entrada no son muy grandes. También establece `input_shape=[28, 28, 1]`, porque las imágenes son de  $28 \times 28$  píxeles, con un solo canal de color (es decir, escala de grises). Cuando cargue el conjunto de datos Fashion MNIST, asegúrese de que cada imagen tenga esta forma:

Es posible que necesite usar `np.reshape()` o `np.expanddims()` para agregar la dimensión de los canales. Alternativamente, puede usar una capa `Reformar` como primera capa del modelo.

- Luego agregamos una capa de agrupación máxima que utiliza el tamaño de agrupación predeterminado de 2, de modo que divide cada dimensión espacial por un factor de 2.
- Luego repetimos la misma estructura dos veces: dos capas convolucionales seguidas de una capa de agrupación máxima. Para imágenes más grandes, podríamos repetir esta estructura varias veces más. El número de repeticiones es un hiperparámetro que puedes ajustar.
- Tenga en cuenta que la cantidad de filtros se duplica a medida que subimos por la CNN hacia la capa de salida (inicialmente es 64, luego 128, luego 256): tiene sentido que crezca, ya que la cantidad de características de bajo nivel suele ser bastante baja. (p. ej., círculos pequeños, líneas horizontales), pero hay muchas formas diferentes de combinarlos en funciones de nivel superior. Es una práctica común duplicar el número de filtros después de cada capa de agrupación: dado que una capa de agrupación divide cada dimensión espacial por un factor de 2, podemos permitirnos duplicar la cantidad de mapas de características en la siguiente capa sin temor a explotar el número de parámetros, uso de memoria o carga computacional.
- La siguiente es la red completamente conectada, compuesta por dos capas densas ocultas y una capa de salida densa. Dado que es una tarea de clasificación con 10 clases, la capa de salida tiene 10 unidades y utiliza la función de activación softmax. Tenga en cuenta que debemos aplanar las entradas justo antes de la primera capa densa, ya que espera una matriz 1D de características para cada instancia. También agregamos dos capas de abandono, con una tasa de abandono del 50% cada una, para reducir el sobreajuste.

Si compila este modelo utilizando la pérdida

`"sparse_categorical_crossentropy"` y ajusta el modelo al conjunto de entrenamiento Fashion MNIST, debería alcanzar más del 92% de precisión en el conjunto de prueba. No es lo último en tecnología, pero es bastante bueno y claramente mucho mejor que lo que logramos con redes densas en el [Capítulo 10](#).

A lo largo de los años, se han desarrollado variantes de esta arquitectura fundamental, lo que ha dado lugar a avances sorprendentes en este campo. Una buena medida de este progreso es la tasa de error en competiciones como el ILSVRC.

**Desafío ImageNet.** En esta competencia, los cinco primeros en tasa de error en imágenes clasificación, es decir, el número de imágenes de prueba para las cuales la parte superior del sistema cinco predicciones no incluían la respuesta correcta: cayó de más del 26% al menos del 2,3% en sólo seis años. Las imágenes son bastante grandes (por ejemplo, 256 píxeles alto) y hay 1.000 clases, algunas de las cuales son realmente sutiles (pruebe distinguiendo 120 razas de perros). Mirando la evolución de la victoria Las entradas son una buena manera de comprender cómo funcionan las CNN y cómo se investiga en ellas. El aprendizaje profundo avanza.

Primero veremos la arquitectura clásica LeNet-5 (1998), luego varios ganadores del desafío ILSVRC: AlexNet (2012), GoogLeNet (2014), ResNet (2015) y SENet (2017). A lo largo del camino, también veremos algunos más arquitecturas, incluidas Xception, ResNeXt, DenseNet, MobileNet, CSPNet y EfficientNet.

## LeNet-5

La arquitectura LeNet-5 es quizás la <sup>10</sup> CNN más conocida arquitectura. Como se mencionó anteriormente, fue creado por Yann LeCun en 1998. y se ha utilizado ampliamente para el reconocimiento de dígitos escritos a mano (MNIST). Es compuesto por las capas que se muestran en la Tabla 14-1.

Tabla 14-1. Arquitectura LeNet-5

Capa	Tipo	Mapas	Tamaño	Tamaño del grano	Calle
Afuera	Totalmente conectado –	10	–	–	–
F6	Totalmente conectado –	84	–	–	–
C5	Circunvolución	120	1×1	5×5	1
T4	agrupación promedio	desde	5×5	2 × 2	2
C3	Circunvolución	desde	10×10	5×5	1
T2	agrupación promedio	6	14 × 14	2 × 2	2
C1	Circunvolución	6	28 × 28	5×5	1
En	Aporte	1	32 × 32	–	–

Como puede ver, esto se parece bastante a nuestro modelo Fashion MNIST: una pila de capas convolucionales y capas de agrupación, seguidas de una red densa. Quizás la principal diferencia con las CNN de clasificación más modernas son las funciones de activación: hoy usaríamos ReLU en lugar de tanh y softmax en lugar de RBF. Hubo varias otras diferencias menores que realmente no importan mucho, pero en caso de que esté interesado, se enumeran en el cuaderno de este capítulo en <https://homl.info/colab3>. Sitio web de Yann LeCun También incluye excelentes demostraciones de los dígitos de clasificación de LeNet-5.

## alexnet

La **arquitectura AlexNet CNN** Ganó<sup>11</sup> el desafío ILSVRC 2012 por un amplio margen: logró una tasa de error entre los cinco primeros del 17 %, mientras que el segundo mejor competidor logró solo el 26 %. AlexaNet fue desarrollada por Alex Krizhevsky (de ahí el nombre), Ilya Sutskever y Geoffrey Hinton. Es similar a LeNet-5, solo que mucho más grande y profundo, y fue el primero en apilar capas convolucionales directamente una encima de otra, en lugar de apilar una capa de agrupación encima de cada capa convolucional. La **Tabla 14-2** presenta esta arquitectura.

Tabla 14-2. Arquitectura AlexNet

Capa	Tipo	Mapas	Tamaño	Tamaño del grano	Calle
Afuera	Totalmente conectado –	1.000	–	–	–
F10	Totalmente conectado –	4.096	–	–	–
F9	Totalmente conectado –	4.096	–	–	–
T8	Agrupación máxima	256	6×6	3 × 3	2
C7	Circunvolución	256	13 × 13	3 × 3	1
C6	Circunvolución	384	13 × 13	3 × 3	1
C5	Circunvolución	384	13 × 13	3 × 3	1
T4	Agrupación máxima	256	13 × 13	3 × 3	2
C3	Circunvolución	256	27 × 27	5×5	1
T2	Agrupación máxima	96	27 × 27	3 × 3	2
C1	Circunvolución	96	55 × 55	11 × 11	4
En	Aporte	3 (RGB)	227 × 227	–	–

Para reducir el sobreajuste, los autores utilizaron dos técnicas de regularización. Primero, aplicaron la deserción (introducida en [el Capítulo 11](#)) con una tasa de deserción del 50% durante el entrenamiento a las salidas de las capas F9 y F10. En segundo lugar, realizaron aumento de datos cambiando aleatoriamente las imágenes de entrenamiento por varios desplazamientos, volteándolos horizontalmente y cambiando las condiciones de iluminación.

## AUMENTO DE DATOS

El aumento de datos aumenta artificialmente el tamaño del conjunto de entrenamiento al generar muchas variantes realistas de cada instancia de entrenamiento.

Esto reduce el sobreajuste, lo que la convierte en una técnica de regularización. Las instancias generadas deben ser lo más realistas posible: idealmente, dada una imagen del conjunto de entrenamiento aumentado, un humano no debería poder decir si fue aumentado o no. Simplemente agregar ruido blanco no ayudará; las modificaciones deben poder aprenderse (el ruido blanco no).

Por ejemplo, puede desplazar, rotar y cambiar ligeramente el tamaño de cada imagen en el conjunto de entrenamiento en varias cantidades y agregar las imágenes resultantes al conjunto de entrenamiento (consulte la [Figura 14-13](#)). Para hacer esto, puede usar las capas de aumento de datos de Keras, introducidas en [el Capítulo 13](#) (por ejemplo, RandomCrop, RandomRotation, etc.). Esto obliga al modelo a ser más tolerante a las variaciones en la posición, orientación y tamaño de los objetos en las imágenes. Para producir un modelo que sea más tolerante con diferentes condiciones de iluminación, también puedes generar muchas imágenes con varios contrastes. En general, también puedes voltear las imágenes horizontalmente (excepto el texto y otros objetos asimétricos). Al combinar estas transformaciones, puedes aumentar considerablemente el tamaño de tu conjunto de entrenamiento.

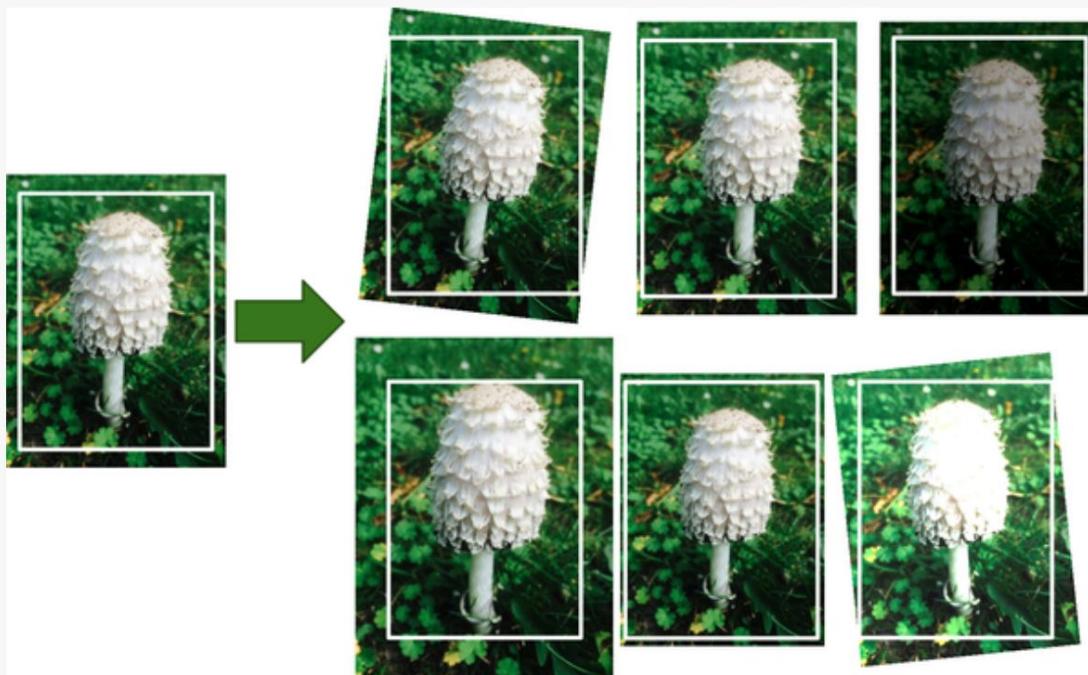


Figura 14-13. Generar nuevas instancias de capacitación a partir de las existentes.

El aumento de datos también es útil cuando tienes un conjunto de datos desequilibrado: puedes usarlo para generar más muestras de las clases menos frecuentes. Esto se denomina técnica de sobremuestreo minoritario sintético, o SMOTE para abreviar.

AlexNet también utiliza un paso de normalización competitiva inmediatamente después del paso ReLU de las capas C1 y C3, llamado normalización de respuesta local (LRN): las neuronas más fuertemente activadas inhiben otras neuronas ubicadas en la misma posición en mapas de características vecinas. Esta activación competitiva se ha observado en neuronas biológicas. Esto fomenta que diferentes mapas de características se especialicen, separándolos y obligándolos a explorar una gama más amplia de características, lo que en última instancia mejora la generalización. [La ecuación 14-2](#) muestra cómo aplicar LRN.

#### Ecuación 14-2. Normalización de respuesta local (LRN)

$$b_i = a_i(k + \alpha \sum_{j=j_{\text{bajo}}}^{j_{\text{alto}}} a_j)^{-\beta} \quad \text{con} \quad j_{\text{alto}} = \min(i + \frac{r}{2}, f_n - \frac{r}{2}) \\ j_{\text{bajo}} = \max(0, i - \frac{r}{2})$$

En esta ecuación:

- $b_i$  es la salida normalizada de la neurona ubicada en el mapa de características  $i$ , en alguna fila  $u$  y columna  $v$  (tenga en cuenta que en esta ecuación consideramos solo las neuronas ubicadas en esta fila y columna, por lo que  $u$  y  $v$  no se muestran).
- $a_i$  es la activación de esa neurona después del paso ReLU, pero antes de la normalización.
- $k$ ,  $\alpha$ ,  $\beta$  y  $r$  son hiperparámetros.  $k$  se llama sesgo y  $r$  se llama radio de profundidad.
- $f$  es el número de mapas de características.

Por ejemplo, si  $r = 2$  y una neurona tiene una fuerte activación, inhibirá la activación de las neuronas ubicadas en los mapas de características inmediatamente encima y debajo de la suya.

En AlexNet, los hiperparámetros se establecen como:  $r = 5$ ,  $\alpha = 0,0001$ ,  $\beta = 0,75$  y  $k = 2$ . Puede implementar este paso utilizando el

Función `tf.nn.local_response_normalization()` (que puede envolver en una capa Lambda si desea usarla en un modelo Keras).

Una variante de AlexNet llamada **ZF Net**<sup>12</sup> fue desarrollado por Matthew Zeiler y Rob Fergus y ganó el desafío ILSVRC 2013. Es esencialmente AlexNet con algunos hiperparámetros modificados (número de mapas de características, tamaño del núcleo, zancada, etc.).

## GoogleLeNet

La **arquitectura de GoogleNet** Fue desarrollado por Christian Szegedy et al. de Google Research, y ganó el desafío **ILSVRC 2014** al llevar la tasa de error de los cinco primeros por debajo del 7%. Este gran desempeño se debió en gran parte al hecho de que la red era mucho más profunda que las CNN anteriores (como verá en **la Figura 14-15**). Esto fue posible gracias a las subredes llamadas módulos iniciales, que permiten a GoogLeNet usar parámetros de manera mucho más eficiente que las arquitecturas anteriores: GoogLeNet en realidad tiene 10 veces menos parámetros que AlexNet (aproximadamente 6 millones en lugar de 60 millones).

**La Figura 14-14** muestra la arquitectura de un módulo inicial. La notación “ $3 \times 3 + 1(S)$ ” significa que la capa usa un núcleo de  $3 \times 3$ , zancada 1 y el “mismo” relleno. La señal de entrada se envía primero a cuatro capas diferentes en paralelo. Todas las capas convolucionales utilizan la función de activación ReLU. Tenga en cuenta que las capas convolucionales superiores utilizan diferentes tamaños de núcleo ( $1 \times 1$ ,  $3 \times 3$  y  $5 \times 5$ ), lo que les permite capturar patrones a diferentes escalas. También tenga en cuenta que cada capa utiliza un paso de 1 y el “mismo” relleno (incluso la capa de agrupación máxima), por lo que todas sus salidas tienen la misma altura y ancho que sus entradas. Esto hace posible concatenar todas las salidas a lo largo de la dimensión de profundidad en la capa de concatenación de profundidad final (es decir, apilar los mapas de características de las cuatro capas convolucionales superiores). Se puede implementar usando la capa `Concatenate` de Keras, usando el eje predeterminado = -1.

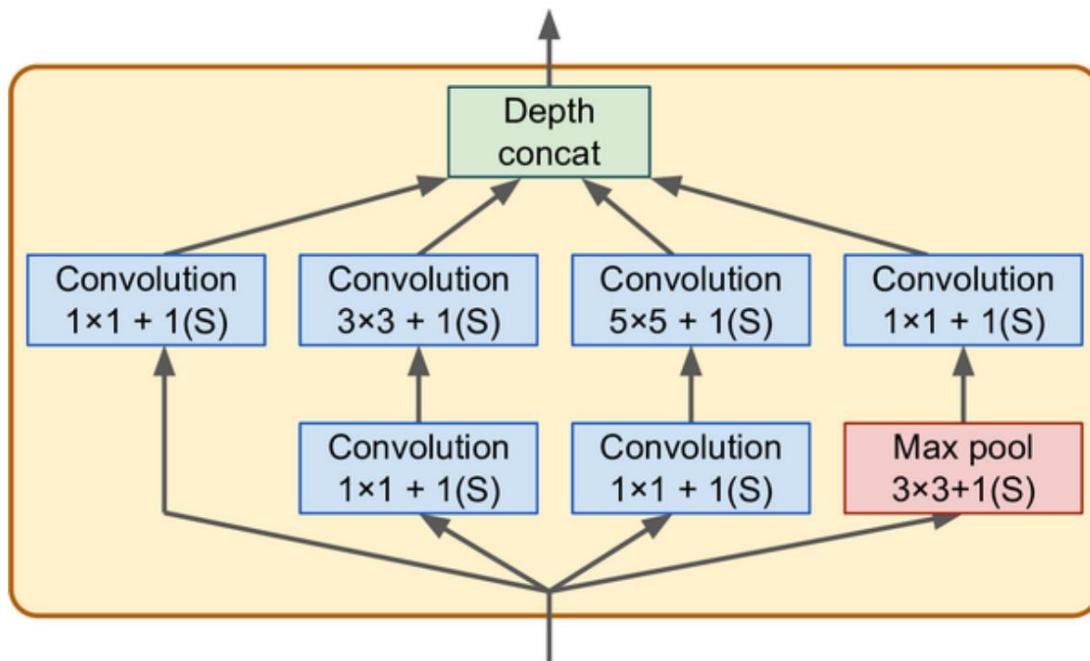


Figura 14-14. Módulo inicial

Quizás se pregunte por qué los módulos iniciales tienen capas convolucionales con núcleos  $1 \times 1$ .

Seguramente estas capas no pueden capturar ninguna característica porque solo miran un píxel a la vez, ¿verdad? De hecho, estas capas tienen tres propósitos:

- Aunque no pueden capturar patrones espaciales, pueden capturar patrones a lo largo de la dimensión de profundidad (es decir, a través de canales).
- Están configurados para generar menos mapas de características que sus entradas, por lo que sirven como capas de cuello de botella, lo que significa que reducen la dimensionalidad. Esto reduce el coste computacional y el número de parámetros, acelerando el entrenamiento y mejorando la generalización.
- Cada par de capas convolucionales ( $[1 \times 1, 3 \times 3]$  y  $[1 \times 1, 5 \times 5]$ ) actúa como una única capa convolucional poderosa, capaz de capturar patrones más complejos. Una capa convolucional equivale a barrer una capa densa a través de la imagen (en cada ubicación, solo mira un pequeño campo receptivo), y estos pares de capas convolucionales equivalen a barrer redes neuronales de dos capas a través de la imagen.

En resumen, puede pensar en todo el módulo inicial como una capa convolucional con esteroides, capaz de generar mapas de características que capturan patrones complejos en varias escalas.

Ahora veamos la arquitectura de GoogLeNet CNN (consulte [la Figura 14-15](#) ). El número de mapas de características generados por cada capa convolucional y cada capa de agrupación se muestra antes del tamaño del núcleo. La arquitectura es tan profunda que tiene que representarse en tres columnas, pero GoogLeNet es en realidad una pila alta, que incluye nueve módulos iniciales (las cajas con las peonzas). Los seis números en los módulos iniciales representan el número de mapas de características generados por cada capa convolucional en el módulo (en el mismo orden que en la [Figura 14-14](#)). Tenga en cuenta que todas las capas convolucionales utilizan la función de activación ReLU.

Pasemos por esta red:

- Las dos primeras capas dividen la altura y el ancho de la imagen por 4 (por lo que su área se divide por 16), para reducir la carga computacional. La primera capa utiliza un tamaño de kernel grande,  $7 \times 7$ , para conservar gran parte de la información.
- Luego, la capa de normalización de respuesta local garantiza que las capas anteriores aprendan una amplia variedad de características (como se analizó anteriormente).
- Siguen dos capas convolucionales, donde la primera actúa como una capa de cuello de botella. Como se mencionó, puede pensar en este par como una única capa convolucional más inteligente.
- Nuevamente, una capa de normalización de respuesta local asegura que las capas anteriores capturen una amplia variedad de patrones.
- A continuación, una capa de agrupación máxima reduce la altura y el ancho de la imagen en 2, nuevamente para acelerar los cálculos.
- Luego viene la columna vertebral de CNN : una pila alta de nueve módulos iniciales, intercalados con un par de capas de agrupación máxima para reducir la dimensionalidad y acelerar la red.
- A continuación, la capa de agrupación promedio global genera la media de cada mapa de características: esto descarta cualquier información espacial restante, lo cual está bien porque no queda mucha información espacial en ese punto. De hecho, normalmente se espera que las imágenes de entrada de GoogLeNet tengan  $224 \times 224$  píxeles, por lo que después de 5 capas de agrupación máximas, cada una de las cuales divide la altura y el ancho por 2, los mapas de características se reducen a  $7 \times 7$ . Además, esta es una tarea de clasificación, no localización, por lo que no importa dónde esté el objeto. Gracias a

Gracias a la reducción de dimensionalidad que aporta esta capa, no es necesario tener varias capas completamente conectadas en la parte superior de la CNN (como en AlexNet), y esto reduce considerablemente la cantidad de parámetros en la red y limita el riesgo de sobreajuste.

- Las últimas capas se explican por sí mismas: abandono para regularización, luego una capa completamente conectada con 1000 unidades (ya que hay 1000 clases) y una función de activación softmax para generar probabilidades de clase estimadas.

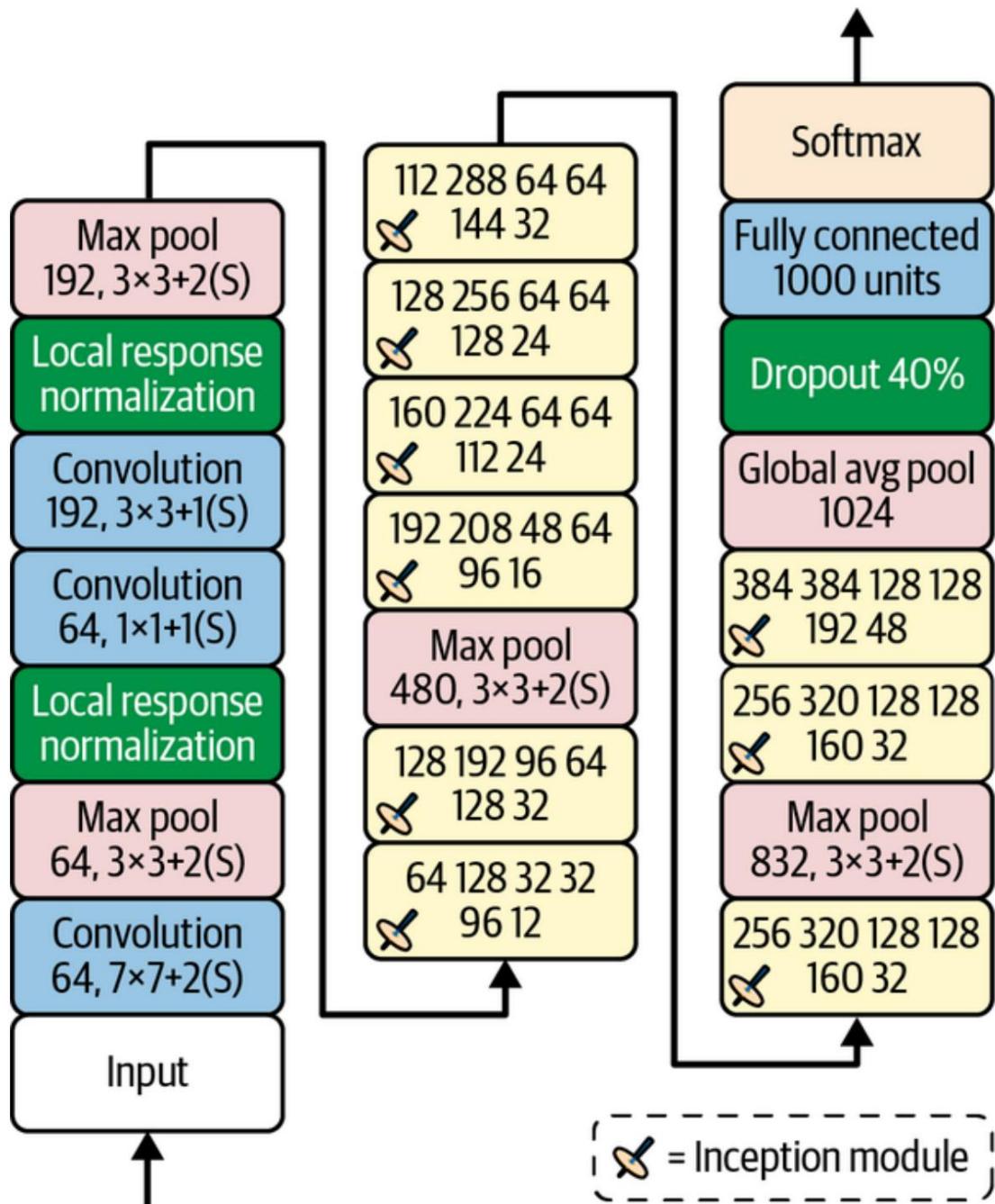


Figura 14-15. Arquitectura de GoogleNet

La arquitectura original de GoogLeNet incluía dos clasificadores auxiliares conectados encima del tercer y sexto módulo inicial. Ambos estaban compuestos por una capa de agrupación promedio, una capa convolucional, dos capas completamente conectadas y una capa de activación softmax. Durante el entrenamiento, su pérdida (reducida en un 70%) se añadió a la pérdida total. El objetivo era luchar.

El problema de los gradientes de fuga y regularizar la red, pero más tarde se demostró que su efecto era relativamente menor.

Posteriormente, los investigadores de Google propusieron varias variantes de la arquitectura GoogLeNet, incluidas Inception-v3 e Inception-v4, utilizando módulos de inicio ligeramente diferentes para alcanzar un rendimiento aún mejor.

## VGGNet

El subcampeón del desafío ILSVRC 2014 fue **VGGNet**, Karen Simonyan y Andrew Zisserman, del laboratorio de investigación Visual Geometry Group (VGG) de la Universidad de Oxford, desarrollaron una arquitectura muy simple y clásica; tenía 2 o 3 capas convolucionales y una capa de agrupación, luego nuevamente 2 o 3 capas convolucionales y una capa de agrupación, y así sucesivamente (llegando a un total de 16 o 19 capas convolucionales, dependiendo de la variante VGG), más una red densa final con 2 capas ocultas y la capa de salida. Utilizaba filtros pequeños de  $3 \times 3$ , pero tenía muchos.

## Resnet

Kaiming He et al. ganó el desafío ILSVRC 2015 utilizando una **red residual** (ResNet) que arrojó ~~una~~ una asombrosa tasa de error entre los cinco primeros por debajo del 3,6 %. La variante ganadora utilizó una CNN extremadamente profunda compuesta por 152 capas (otras variantes tenían 34, 50 y 101 capas). Confirmó la tendencia general: los modelos de visión por computadora eran cada vez más profundos, con cada vez menos parámetros. La clave para poder entrenar una red tan profunda es utilizar conexiones de salto (también llamadas conexiones de acceso directo): la señal que ingresa a una capa también se agrega a la salida de una capa ubicada más arriba en la pila. Veamos por qué esto es útil.

Al entrenar una red neuronal, el objetivo es hacer que modele una función objetivo  $h(x)$ . Si agrega la entrada  $x$  a la salida de la red (es decir, agrega una conexión de salto), entonces la red se verá obligada a modelar  $f(x) = h(x) - x$  en lugar de  $h(x)$ . Esto se llama aprendizaje residual (ver **Figura 14-16**).

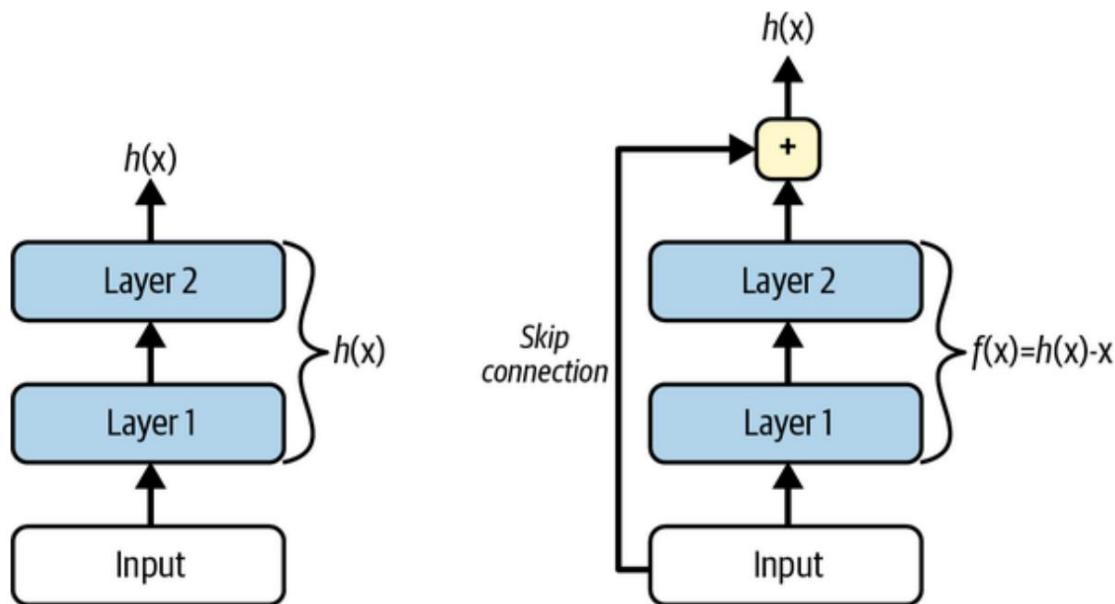


Figura 14-16. Aprendizaje residual

Cuando inicializa una red neuronal normal, sus pesos son cercanos a cero, por lo que la red simplemente genera valores cercanos a cero. Si agrega una conexión de omisión, la red resultante simplemente genera una copia de sus entradas; en otras palabras, inicialmente modela la función de identidad. Si la función objetivo está bastante cerca de la función de identidad (que suele ser el caso), esto acelerará considerablemente el entrenamiento.

Además, si agrega muchas conexiones de salto, la red puede comenzar a progresar incluso si varias capas aún no han comenzado a aprender (consulte [la Figura 14-17](#)). Gracias a las conexiones omitidas, la señal puede llegar fácilmente a través de toda la red. La red residual profunda puede verse como una pila de unidades residuales (RU), donde cada unidad residual es una pequeña red neuronal con una conexión de salto.

Ahora veamos la arquitectura de ResNet (consulte [la Figura 14-18](#)). Es sorprendentemente simple. Comienza y termina exactamente como GoogLeNet (excepto sin una capa de abandono), y en el medio hay solo una pila muy profunda de unidades residuales. Cada unidad residual se compone de dos capas convolucionales (¡y ninguna capa de agrupación!), con normalización por lotes (BN) y activación ReLU, utilizando núcleos de  $3 \times 3$  y preservando las dimensiones espaciales (zancada 1, "mismo" relleno).

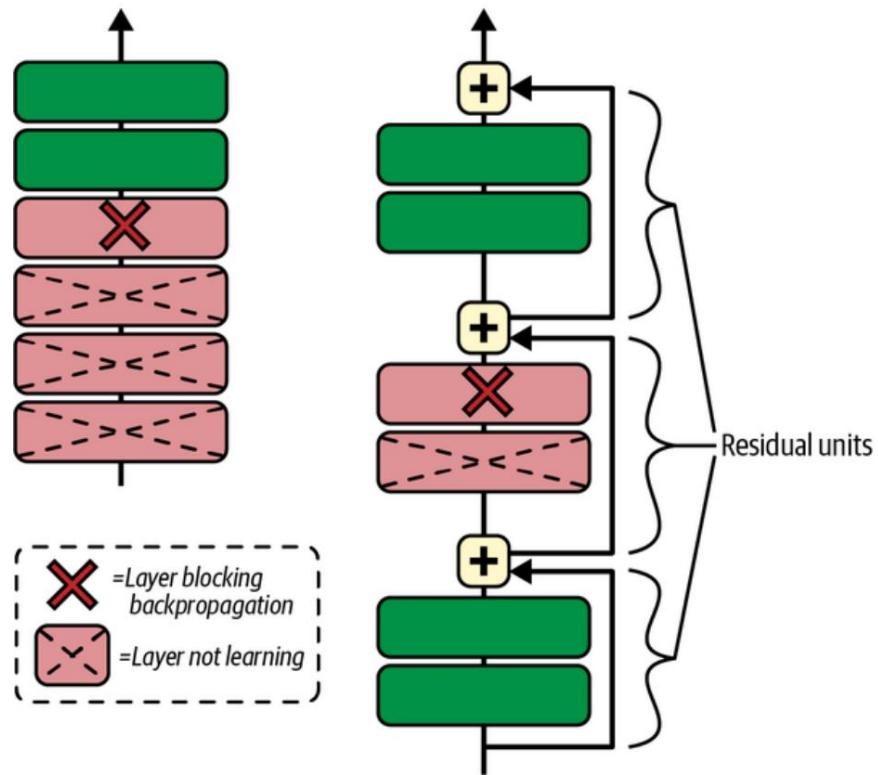


Figura 14-17. Red neuronal profunda regular (izquierda) y red residual profunda (derecha)

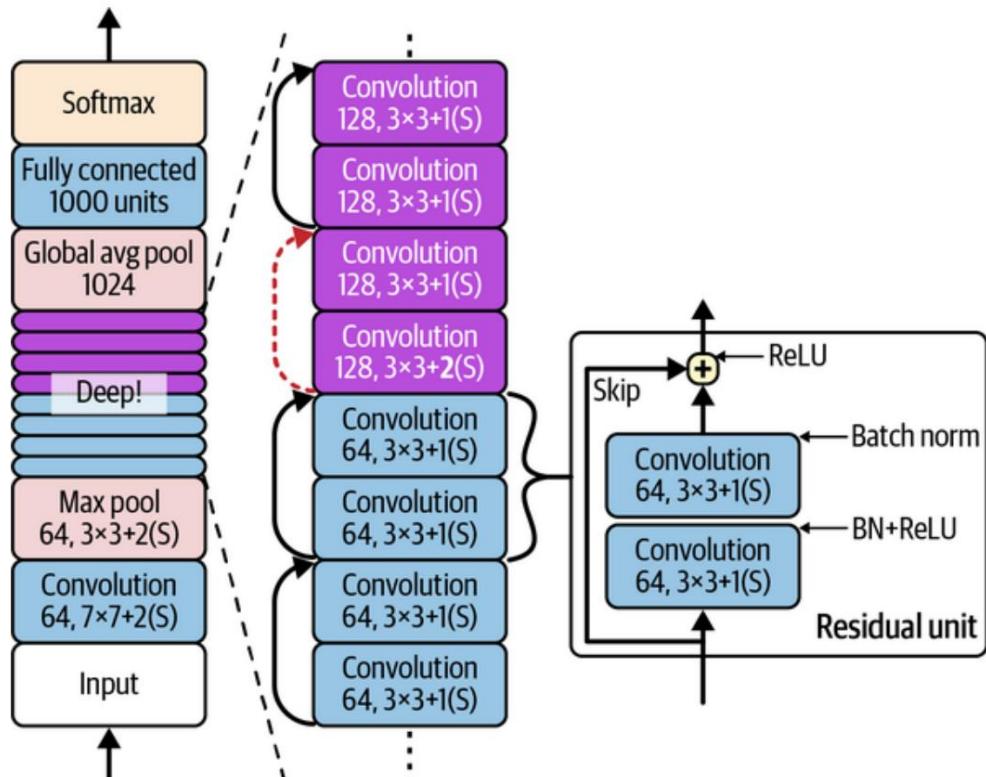


Figura 14-18. Arquitectura ResNet

Tenga en cuenta que el número de mapas de características se duplica cada pocas unidades residuales, al mismo tiempo que su altura y ancho se reducen a la mitad (usando una capa convolucional con paso 2). Cuando esto sucede, las entradas no se pueden agregar directamente a las salidas de la unidad residual porque no tienen la misma forma (por ejemplo, este problema afecta la conexión de salto representada por la flecha discontinua en la Figura 14-18) . Para resolver este problema, las entradas pasan a través de una capa convolucional  $1 \times 1$  con paso 2 y el número correcto de mapas de características de salida (consulte [la Figura 14-19](#)).

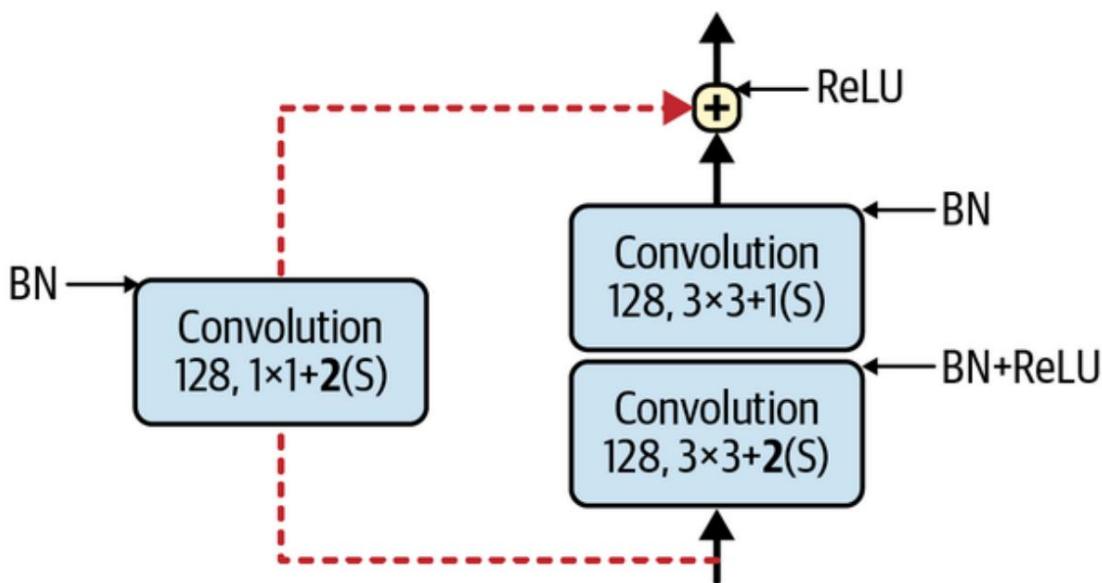


Figura 14-19. Saltar conexión al cambiar el tamaño y la profundidad del mapa de características

Existen diferentes variaciones de la arquitectura, con diferente número de capas.

ResNet-34 es una ResNet con 34 capas (solo contando las capas convolucionales y la capa completamente conectada) que contiene 3 RU que generan<sup>17</sup> 64 mapas de características, 4 RU con 128 mapas, 6 RU con 256 mapas y 3 RU con 512 mapas.

Implementaremos esta arquitectura más adelante en este capítulo.

### NOTA

[Inception-v4](#) de Google La arquitectura fusionó las ideas de GoogLeNet y ResNet y logró una tasa de error entre los cinco primeros de cerca del 3% en la clasificación de ImageNet.<sup>18</sup>

Las ResNet más profundas que eso, como ResNet-152, utilizan unidades residuales ligeramente diferentes. En lugar de dos capas convolucionales de  $3 \times 3$  con, digamos, 256

mapas de características, utilizan tres capas convolucionales: primero una capa convolucional  $1 \times 1$  con solo 64 mapas de características ( $4 \times$  menos), que actúa como una capa de cuello de botella (como ya se discutió), luego una capa de  $3 \times 3$  con 64 mapas de características, y finalmente otra capa convolucional  $1 \times 1$  con 256 mapas de características ( $4 \times 64$ ) que restaura la profundidad original. ResNet-152 contiene 3 RU que generan 256 mapas, luego 8 RU con 512 mapas, la friolera de 36 RU con 1024 mapas y finalmente 3 RU con 2048 mapas.

## Xception

Cabe destacar otra variante de la arquitectura GoogLeNet: **Xception** (que significa <sup>19</sup> Extreme Inception) fue propuesto en 2016 por François Chollet (el autor de Keras), y superó significativamente a Inception-v3 en una enorme tarea de visión (350 millones de imágenes y 17.000 clases). Al igual que Inception-v4, fusiona las ideas de GoogLeNet y ResNet, pero reemplaza los módulos de inicio con un tipo especial de capa llamada capa de convolución separable en profundidad (o capa de convolución separable para abreviar). <sup>20</sup>

Estas capas se habían utilizado antes en algunas arquitecturas CNN, pero no eran tan centrales como en la arquitectura Xception. Mientras que una capa convolucional regular utiliza filtros que intentan capturar simultáneamente patrones espaciales (p. ej., un óvalo) y patrones de canales cruzados (p. ej., boca + nariz + ojos = cara), una capa convolucional separable asume firmemente que los patrones espaciales y los patrones cruzados -Los patrones de canales se pueden modelar por separado (consulte [la Figura 14-20](#)). Por lo tanto, se compone de dos partes: la primera parte aplica un único filtro espacial a cada mapa de características de entrada, luego la segunda parte busca exclusivamente patrones entre canales; es solo una capa convolucional regular con filtros  $1 \times 1$ .

Dado que las capas convolucionales separables solo tienen un filtro espacial por canal de entrada, debe evitar usarlas después de capas que tienen muy pocos canales, como la capa de entrada (claro, eso es lo que representa la Figura 14-20, pero es solo para fines ilustrativos) . . Por esta razón, la arquitectura Xception comienza con 2 capas convolucionales regulares, pero luego el resto de la arquitectura usa solo convoluciones separables (34 en total), además de algunas capas de agrupación máxima y las capas finales habituales (una capa de agrupación promedio global y una capa de salida densa).

Quizás se pregunte por qué Xception se considera una variante de GoogLeNet, ya que no contiene ningún módulo inicial. Bueno, como se discutió anteriormente, un

El módulo de inicio contiene capas convolucionales con filtros  $1 \times 1$ : estos buscan exclusivamente patrones de canales cruzados. Sin embargo, las capas convolucionales que se encuentran encima de ellas son capas convolucionales regulares que buscan patrones espaciales y entre canales. Por lo tanto, puede pensar en un módulo inicial como un intermedio entre una capa convolucional regular (que considera patrones espaciales y patrones entre canales de forma conjunta) y una capa convolucional separable (que los considera por separado). En la práctica, parece que las capas convolucionales separables suelen fu-

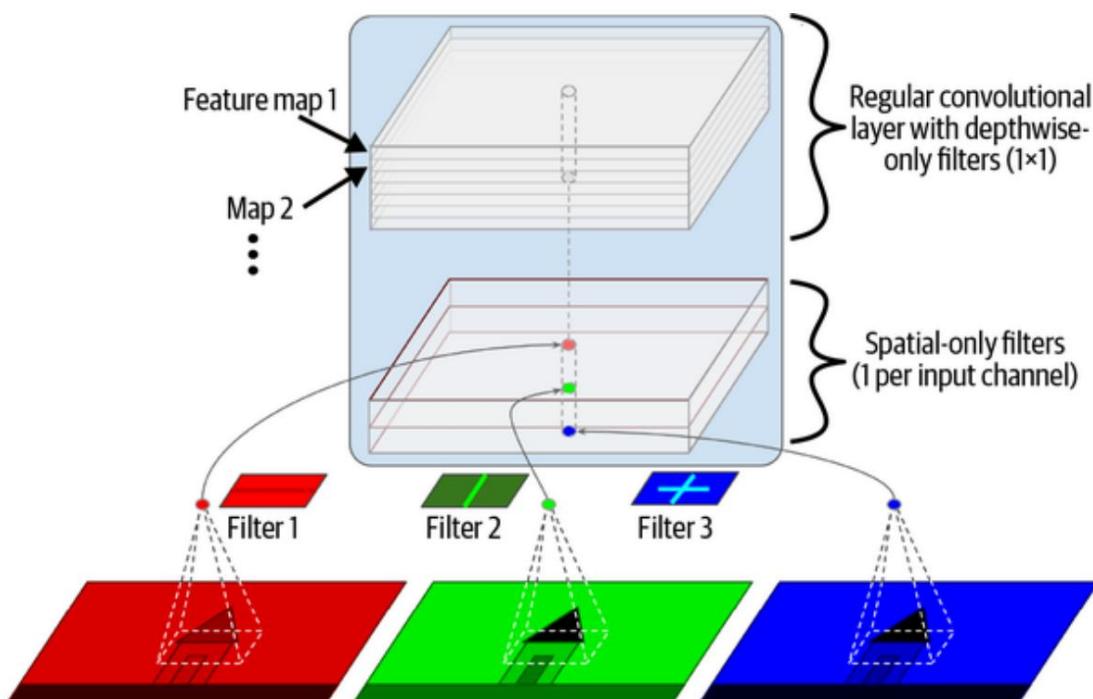


Figura 14-20. Capa convolucional separable en profundidad

#### CONSEJO

Las capas convolucionales separables utilizan menos parámetros, menos memoria y menos cálculos que las capas convolucionales normales y, a menudo, funcionan mejor. Considere usarlos de forma predeterminada, excepto después de capas con pocos canales (como el canal de entrada). En Keras, simplemente use SeparableConv2D en lugar de Conv2D: es un reemplazo directo. Keras también ofrece una capa DepthwiseConv2D que implementa la primera parte de una capa convolucional separable en profundidad (es decir, aplicando un filtro espacial por mapa de características de entrada).

## SENET

La arquitectura ganadora en el desafío ILSVRC 2017 fue [Squeeze-and-Excitation Network \(SENet\)](#). Esta arquitectura amplía las arquitecturas existentes, como las redes iniciales y ResNets, y mejora su rendimiento. Esto permitió a SENet ganar la competencia con una asombrosa tasa de error entre los cinco primeros del 2,25%! Las versiones extendidas de redes iniciales y ResNets se denominan SE-Inception y SE-ResNet, respectivamente. El impulso proviene del hecho de que SENet agrega una pequeña red neuronal, llamada bloque SE, a cada módulo inicial o unidad residual en la arquitectura original, como se muestra en la Figura 14-21.

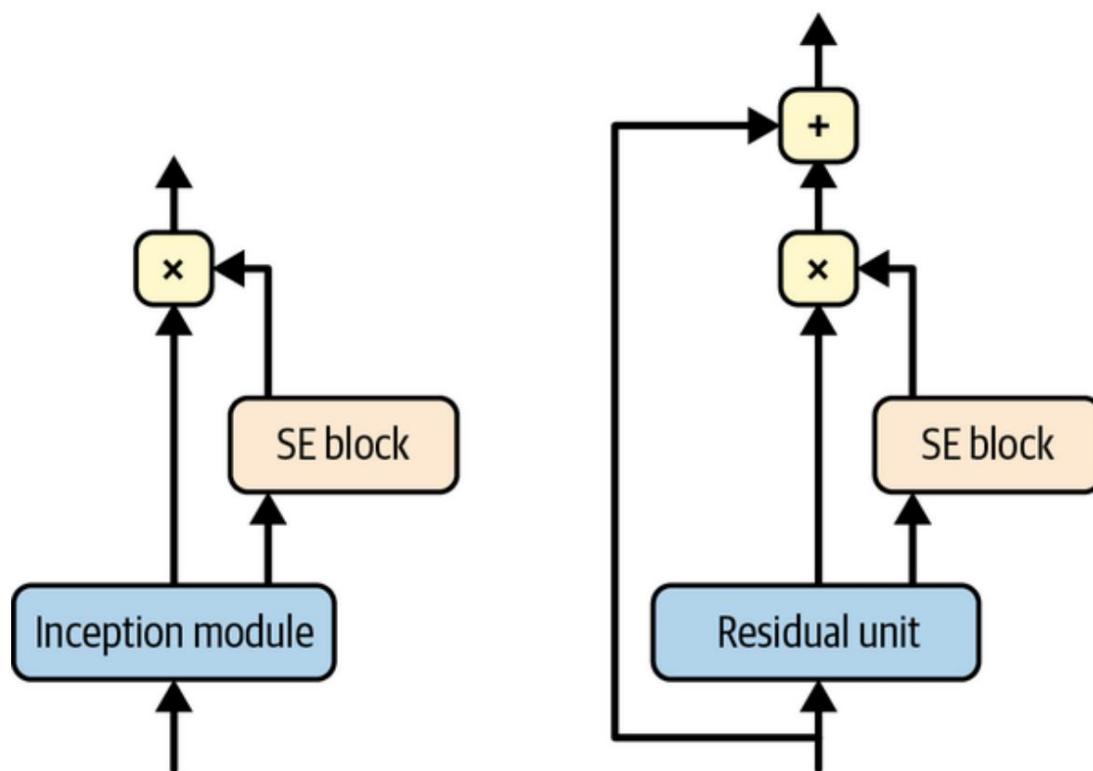


Figura 14-21. Módulo SE-Inception (izquierda) y unidad SE-ResNet (derecha)

Un bloque SE analiza la salida de la unidad a la que está conectado, centrándose exclusivamente en la dimensión de profundidad (no busca ningún patrón espacial) y aprende qué características suelen ser más activas juntas. Luego utiliza esta información para recalibrar los mapas de características, como se muestra en la Figura 14-22. Por ejemplo, un bloque SE puede aprender que las bocas, las narices y los ojos suelen aparecer juntos en las imágenes: si ves una boca y una nariz, deberías esperar ver ojos también. Entonces, si el bloque ve una fuerte activación en los mapas de características de la boca y la nariz, pero solo una activación leve en el mapa de características del ojo, mejorará el mapa de características del ojo (más precisamente, reducirá

mapas de características irrelevantes). Si los ojos estaban algo confundidos con otra cosa, esta recalibración del mapa de características ayudará a resolver la ambigüedad.

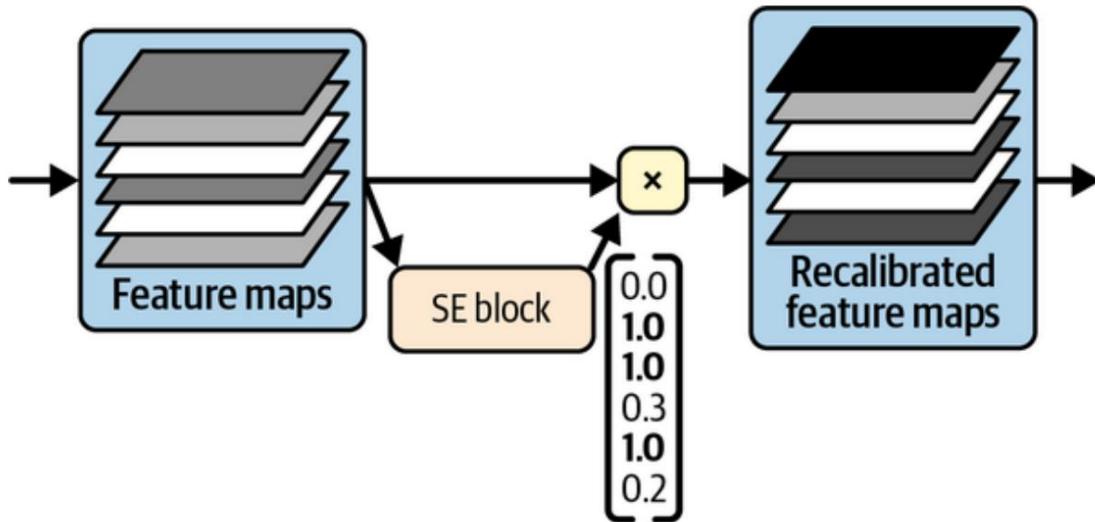


Figura 14-22. Un bloque SE realiza la recalibración del mapa de características

Un bloque SE se compone de solo tres capas: una capa de agrupación promedio global, una capa densa oculta que usa la función de activación ReLU y una capa de salida densa que usa la función de activación sigmoidea (consulte la Figura 14-23).

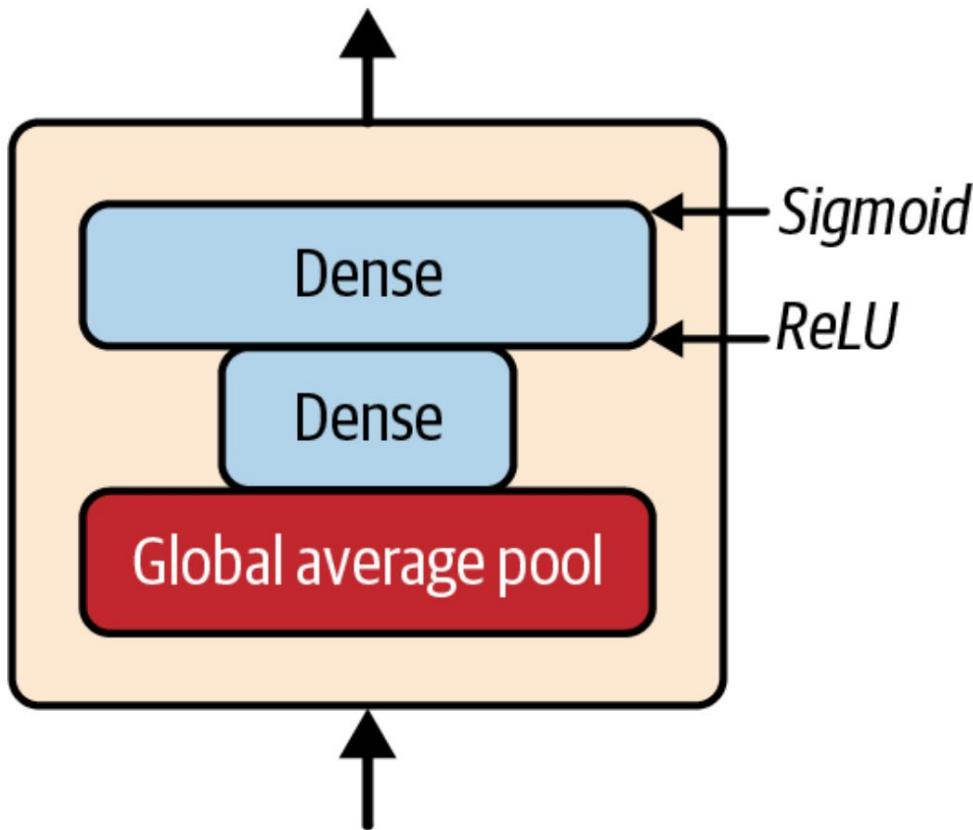


Figura 14-23. Arquitectura de bloque SE

Como antes, la capa de agrupación promedio global calcula la activación media para cada mapa de características: por ejemplo, si su entrada contiene 256 mapas de características, generará 256 números que representan el nivel general de respuesta para cada filtro. La siguiente capa es donde ocurre la "compresión": esta capa tiene significativamente menos de 256 neuronas (normalmente 16 veces menos que el número de mapas de características (p. ej., 16 neuronas), por lo que los 256 números se comprimen en un pequeño vector (p. ej., 16 dimensiones). Esta es una representación vectorial de baja dimensión (es decir, una incrustación) de la distribución de respuestas de características. Este paso de cuello de botella obliga al bloque SE a aprender una representación general de las combinaciones de características (veremos este principio en acción nuevamente cuando analicemos los codificadores automáticos en el [Capítulo 17](#)).

Finalmente, la capa de salida toma la incrustación y genera un vector de recalibración que contiene un número por mapa de características (por ejemplo, 256), cada uno entre 0 y 1. Luego, los mapas de características se multiplican por este vector de recalibración, por lo que las características irrelevantes (con una recalibración baja) puntuación se reducen mientras que las características relevantes (con una puntuación de recalibración cercana a 1) se dejan solas.

Otras arquitecturas destacadas Hay muchas otras arquitecturas de CNN para explorar. He aquí un breve resumen de algunos de los más destacados:

#### Resiguiente<sup>22</sup>

ResNeXt mejora las unidades residuales en ResNet. Mientras que las unidades residuales en los mejores modelos ResNet solo contienen 3 capas convolucionales cada una, las unidades residuales ResNeXt se componen de muchas pilas paralelas (por ejemplo, 32 pilas), con 3 capas convolucionales cada una. Sin embargo, las dos primeras capas de cada pila solo usan unos pocos filtros (por ejemplo, solo cuatro), por lo que la cantidad total de parámetros sigue siendo la misma que en ResNet. Luego, las salidas de todas las pilas se suman y el resultado se pasa a la siguiente unidad residual (junto con la conexión de salto).

#### Red densa <sup>23</sup>

Una DenseNet se compone de varios bloques densos, cada uno de los cuales consta de unas pocas capas convolucionales densamente conectadas. Esta arquitectura logró una precisión excelente utilizando comparativamente pocos parámetros. ¿Qué significa “densamente conectado”? La salida de cada capa se alimenta como entrada a cada capa posterior dentro del mismo bloque. Por ejemplo, la capa 4 de un bloque toma como entrada la concatenación en profundidad de las salidas de las capas 1, 2 y 3 de ese bloque. Los bloques densos están separados por algunas capas de transición.

#### red móvil <sup>24</sup>

Los MobileNets son modelos optimizados diseñados para ser livianos y rápidos, lo que los hace populares en aplicaciones móviles y web. Se basan en capas convolucionales separables en profundidad, como Xception. Los autores propusieron varias variantes, cambiando un poco de precisión por modelos más rápidos y más pequeños.

#### RedCSP <sup>25</sup>

Una red parcial entre etapas (CSPNet) es similar a una DenseNet, pero parte de la entrada de cada bloque denso se concatena directamente a la salida de ese bloque, sin pasar por el bloque.

#### Red eficiente <sup>26</sup>

EfficientNet es posiblemente el modelo más importante de esta lista. Los autores propusieron un método para escalar cualquier CNN de manera eficiente, aumentando conjuntamente la profundidad (número de capas), el ancho (número de filtros por capa) y la resolución (tamaño de la imagen de entrada) de una manera basada en principios. Esto se llama escalamiento compuesto. Utilizaron la búsqueda de arquitectura neuronal para encontrar una buena arquitectura para una versión reducida de ImageNet (con cada vez menos imágenes) y luego utilizaron escalado compuesto para crear versiones cada vez más grandes de esta arquitectura. Cuando aparecieron los modelos EfficientNet, superaron ampliamente a todos los modelos existentes, en todos los presupuestos informáticos, y siguen estando entre los mejores modelos que existen en la actualidad.

Comprender el método de escalado compuesto de EfficientNet es útil para obtener una comprensión más profunda de las CNN, especialmente si alguna vez necesita escalar una arquitectura de CNN. Se basa en una medida logarítmica del presupuesto de cómputo, denominado  $\alpha$ : si su presupuesto de cómputo se duplica, entonces  $\alpha$  aumenta en 1. En otras palabras, el número de operaciones de punto flotante disponibles para entrenamiento es  $\alpha$  proporcional a 2. La profundidad, el ancho y la resolución de su arquitectura CNN  $\alpha$  deben escalarse como  $\alpha^\beta$  y  $\gamma$ , respectivamente. Los factores  $\alpha$ ,  $\beta$  y  $\gamma$  deben ser mayores que 1, y  $\alpha + \beta + \gamma$  deben estar cerca de 2. Los valores óptimos para estos factores dependen de la arquitectura de la CNN. Para encontrar los valores óptimos para la arquitectura EfficientNet, los autores comenzaron con un pequeño modelo de referencia (EfficientNetB0), fijaron  $\alpha = 1$  y simplemente ejecutaron una búsqueda en cuadrícula: encontraron  $\alpha = 1,2$ ,  $\beta = 1,1$  y  $\gamma = 1,1$ . Luego utilizaron estos factores para crear varias arquitecturas más grandes, denominadas EfficientNetB1 a EfficientNetB7, para aumentar los valores de  $\alpha$ .

Elegir la arquitectura CNN adecuada Con tantas arquitecturas CNN,

¿cómo elige cuál es mejor para su proyecto? Bueno, depende de lo que más te importe: ¿Precisión?

¿Tamaño del modelo (por ejemplo, para implementación en un dispositivo móvil)? ¿Velocidad de inferencia en la CPU? ¿En GPU? [La Tabla 14-3](#) enumera los mejores modelos previamente entrenados disponibles actualmente en Keras (verá cómo usarlos más adelante en este capítulo), ordenados por tamaño de modelo. Puede encontrar la lista completa en <https://keras.io/api/applications>. Para cada modelo, la tabla muestra el nombre de la clase Keras que se utilizará (en el paquete `tf.keras.applications`), el tamaño del modelo en MB, la precisión de validación top 1 y top 5 en el conjunto de datos ImageNet, el número de

parámetros (millones) y el tiempo de inferencia en CPU y GPU en ms, utilizando lotes de 32 imágenes en hardware razonablemente potente. Para cada 27 columna, se resalta el mejor valor. Como puede ver, los modelos más grandes son generalmente más preciso, pero no siempre; por ejemplo, EfficientNetB2 supera a InceptionV3 tanto en tamaño como en precisión. Sólo me quedé con InceptionV3 en la lista porque es casi el doble de rápido que EfficientNetB2 en una CPU. De manera similar, InceptionResNetV2 es rápido en una CPU, y ResNet50V2 y ResNet101V2 son increíblemente rápidos en una GPU.

Tabla 14-3. Modelos previamente entrenados disponibles en Keras

Nombre de la clase	Tamaño (MB)	Cuenta top 1	Top-5 de cuentas	parámetros
MóvilNetV2	14	71,3%	90,1%	3,5 millones
red móvil	dicidid	70,4%	89,5%	4,3 millones
NASNetMóvil	23	74,4%	91,9%	5,3 millones
EfficientNetB0	29	77,1%	93,3%	5,3 millones
EfficientNetB1	31	79,1%	94,4%	7,9 millones
EfficientNetB2	36	80,1%	94,9%	9,2 millones
EfficientNetB3	48	81,6%	95,7%	12,3 millones
EfficientNetB4	75	82,9%	96,4%	19,5 millones
InicioV3	92	77,9%	93,7%	23,9 millones
ResNet50V2	98	76,0%	93,0%	25,6 millones
EfficientNetB5	118	83,6%	96,7%	30,6 millones
EfficientNetB6	166	84,0%	96,8%	43,3 millones
ResNet101V2	171	77,2%	93,8%	44,7 millones
InicioResNetV2 215		80,3%	95,3%	55,9 millones
EfficientNetB7	256	84,3%	97,0%	66,7 millones



¡Espero que hayas disfrutado de esta inmersión profunda en las principales arquitecturas de CNN! Ahora veamos cómo implementar uno de ellos usando Keras.

## Implementación de una CNN ResNet-34 usando Keras

La mayoría de las arquitecturas CNN descritas hasta ahora se pueden implementar de forma bastante natural utilizando Keras (aunque generalmente, como verá, cargaría una red previamente entrenada). Para ilustrar el proceso, implementemos un ResNet-34 desde cero con Keras. Primero, crearemos una capa ResidualUnit:

```
DefaultConv2D = parcial(tf.keras.layers.Conv2D, kernel_size=3, strides=1,
                       padding="igual",
                       kernel_initializer="he_normal", use_bias=False)

clase Unidad Residual (tf.keras.layers.Layer):
    def __init__(self, filtros, zancadas=1, activación="relu",
                 **kwargs):
        super().__init__(**kwargs) self.activation
        = tf.keras.activations.get(activación) self.main_layers = [ DefaultConv2D(filtros,
                           zancadas=zancadas),
                           tf.keras.layers.BatchNormalization(), self. activación,
                           DefaultConv2D (filtros), tf.keras.layers.BatchNormalization()

    ] self.skip_layers = [] si zancadas
    > 1:
        self.skip_layers =
            [ DefaultConv2D(filtros, kernel_size=1, zancadas=zancadas),
              tf.keras.layers.BatchNormalization()
            ]

    def call(self, entradas): Z = entradas
        para la capa
        en self.main_layers: Z = capa(Z) skip_Z =
            entradas para la
            capa en self.skip_layers:
```

```

    saltar_Z = capa(saltar_Z)
    devolver auto.activación(Z + saltar_Z)

```

Como puede ver, este código coincide bastante con la Figura 14-19 . En el constructor, creamos todas las capas que necesitaremos: las capas principales son las del lado derecho del diagrama y las capas de salto son las de la izquierda (solo son necesarias si la zancada es mayor que 1). Luego, en el método call(), hacemos que las entradas pasen por las capas principales y las capas de salto (si las hay), agregamos ambas salidas y aplicamos la función de activación.

Ahora podemos construir un ResNet-34 usando un modelo secuencial, ya que en realidad es solo una larga secuencia de capas; podemos tratar cada unidad residual como una sola capa ahora que tenemos la clase ResidualUnit. El código coincide mucho con la Figura 14-18:

```

modelo = tf.keras.Sequential([
    DefaultConv2D(64, kernel_size=7, strides=2, input_shape=[224, 224, 3]),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Activation("relu"),
    Layers.MaxPool2D(pool_size=3, strides=2, padding="same"), ])
prev_filters
= 64 para filtros en [64]
*
3 + [128] * 4 + [256] * 6 +
[512] * 3 : zancadas = 1 si filtros == prev_filters else 2
model.add(ResidualUnit(filters,
    zancadas=zancadas)) prev_filters = filtros

model.add(tf.keras.layers.GlobalAvgPool2D())
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(10, activation="softmax"))

```

La única parte complicada de este código es el bucle que agrega las capas ResidualUnit al modelo: como se explicó anteriormente, las primeras 3 RU tienen 64 filtros, luego las siguientes 4 RU tienen 128 filtros, y así sucesivamente. En cada iteración, debemos establecer la zancada en 1 cuando el número de filtros es el mismo que en el RU anterior, o en caso contrario lo establecemos en 2; luego agregamos ResidualUnit y finalmente actualizamos prev\_filters.

¡Es sorprendente que en aproximadamente 40 líneas de código podamos construir el modelo que ganó el desafío ILSVRC 2015! Esto demuestra tanto la elegancia del

Modelo ResNet y la expresividad de la API de Keras. Implementar las otras arquitecturas de CNN es un poco más largo, pero no mucho más difícil. Sin embargo, Keras viene con varias de estas arquitecturas integradas, así que ¿por qué no utilizarlas en su lugar?

## Uso de modelos previamente entrenados de Keras

En general, no tendrá que implementar modelos estándar como GoogLeNet o ResNet manualmente, ya que las redes previamente entrenadas están disponibles con una sola línea de código en el paquete `tf.keras.applications`.

Por ejemplo, puede cargar el modelo ResNet-50, previamente entrenado en ImageNet, con la siguiente línea de código:

```
modelo = tf.keras.applications.ResNet50(pesos="imagenet")
```

¡Eso es todo! Esto creará un modelo ResNet-50 y descargará pesos previamente entrenados en el conjunto de datos de ImageNet. Para usarlo, primero debes asegurarte de que las imágenes tengan el tamaño correcto. Un modelo ResNet-50 espera imágenes de  $224 \times 224$  píxeles (otros modelos pueden esperar otros tamaños, como  $299 \times 299$ ), así que usemos la capa `Resizing` de Keras (presentada en el Capítulo 13) para cambiar el tamaño de dos imágenes de muestra (después de recortarlas al relación de aspecto objetivo):

```
imágenes = load_sample_images()["imágenes"]
imágenes_resized = tf.keras.layers.Resizing(alto=224, ancho=224,
                                             crop_to_aspect_ratio=True)(imágenes)
```

Los modelos previamente entrenados suponen que las imágenes se preprocesan de una manera específica. En algunos casos, pueden esperar que las entradas se escale de 0 a 1, o de -1 a 1, y así sucesivamente. Cada modelo proporciona una función `preprocess_input()` que puede utilizar para preprocesar sus imágenes. Estas funciones suponen que los valores de píxeles originales oscilan entre 0 y 255, como es el caso aquí:

```
entradas =
tf.keras.applications.resnet50.preprocess_input(imágenes_resized)
```

Ahora podemos usar el modelo previamente entrenado para hacer predicciones:

```
>>> Y_proba = modelo.predecir(entradas)
>>> Y_proba.forma (2, 1000)
```

Como es habitual, la salida Y\_proba es una matriz con una fila por imagen y una columna por clase (en este caso, hay 1000 clases). Si desea mostrar las K predicciones principales, incluido el nombre de la clase y la probabilidad estimada de cada clase predicha, utilice la función decode\_predictions(). Para cada imagen, devuelve una matriz que contiene las K predicciones principales, donde cada predicción se representa como una matriz que contiene el identificador de clase, su nombre y la puntuación de confianza correspondiente:

```
top_K =
tf.keras.applications.resnet50.decode_predictions(Y_proba, top=3) para image_index en el rango(len(images)):
print(f"Imagen #{image_index}") para class_id, nombre, y_proba en
    top_K[image_index]:
        print(f" {class_id} - {nombre:12s} {y_proba:.2%}")
```

La salida se ve así:

Imagen #0	
n03877845 - palacio	54,69%
n03781244	
- monasterio	24,72%
n02825657 - bell_cote	
	18,55%
Imagen #1	
n04522168 - jarrón	32,66%
n11939491 - margarita	17,81%
n03530642 - panal	12,06%

Las clases correctas son palacio y dalia, por lo que el modelo es correcto para la primera imagen pero incorrecto para la segunda. Sin embargo, eso se debe a que dahlia no es una de las 1000 clases de ImageNet. Teniendo esto en cuenta, el jarrón es una suposición razonable (¿quizás la flor está en un jarrón?), y la margarita tampoco es una mala elección, ya que tanto las dalias como las margaritas pertenecen a la misma familia Compositae.

Como puede ver, es muy fácil crear un clasificador de imágenes bastante bueno utilizando un modelo previamente entrenado. Como vio en [la Tabla 14-3](#), hay muchos otros modelos de visión disponibles en tf.keras.applications, desde modelos livianos y rápidos hasta modelos grandes y precisos.

Pero, ¿qué sucede si desea utilizar un clasificador de imágenes para clases de imágenes que no forman parte de ImageNet? En ese caso, aún puede beneficiarse de los modelos previamente entrenados si los utiliza para realizar aprendizaje por transferencia.

### Modelos previamente entrenados para el aprendizaje por transferencia

Si desea crear un clasificador de imágenes pero no tiene datos suficientes para entrenarlo desde cero, suele ser una buena idea reutilizar las capas inferiores de un modelo previamente entrenado, como analizamos en el Capítulo 11. Por ejemplo , entrenemos un modelo para clasificar imágenes de flores, reutilizando un modelo Xception previamente entrenado.

Primero, cargaremos el conjunto de datos de flores usando TensorFlow Datasets (presentado en el [Capítulo 13](#)):

```
importar tensorflow_datasets como tfds

conjunto de datos, info = tfds.load("tf_flowers", as_supervised=True, with_info=True)
dataset_size =
info.splits["train"].num_examples # 3670 class_names =
info.features["label"].names # ["diente de león ", "margarita", ...] n_classes =

info.features["label"].num_classes # 5
```

Tenga en cuenta que puede obtener información sobre el conjunto de datos configurando `with_info=True`. Aquí obtenemos el tamaño del conjunto de datos y los nombres de las clases. Desafortunadamente, solo hay un conjunto de datos de "entrenamiento", no un conjunto de prueba ni un conjunto de validación, por lo que debemos dividir el conjunto de entrenamiento. Llamemos a `tfds.load()` nuevamente, pero esta vez tomando el primer 10% del conjunto de datos para prueba, el siguiente 15% para validación y el 75% restante para entrenamiento:

```
test_set_raw, valid_set_raw, train_set_raw = tfds.load( "tf_flowers",
split=["entrenar[:10%]", "entrenar[10%:25%]", "entrenar[25%:]"],
as_supervised=True )
```

Los tres conjuntos de datos contienen imágenes individuales. Necesitamos agruparlos, pero primero debemos asegurarnos de que todos tengan el mismo tamaño, o el procesamiento por lotes fallará. Podemos usar una capa de cambio de tamaño para esto. También debemos llamar a la función `tf.keras.applications.xception.preprocess_input()`

para preprocesar las imágenes de forma adecuada para el modelo Xception. Por último, también mezclaremos el conjunto de entrenamiento y usaremos la captación previa:

```
tamaño_lote = 32
preproceso = tf.keras.Sequential([
    tf.keras.layers.Resizing(alto=224, ancho=224, crop_to_aspect_ratio=True),

    tf.keras.layers.Lambda(tf.keras.applications.xception.preprocess_input)]) train_set = train_set_raw.map(lambda
X, y:

    preprocess(X, y)) train_set = train_set.shuffle(1000, semilla =42).batch(batch_size).prefetch(1) valid_set =
valid_set_raw.map(lambda X, y: (preprocess(X,
y)).batch(batch_size) test_set = test_set_raw.map(lambda X, y:
( preproceso(X), y)).batch(batch_size)
```

Ahora cada lote contiene 32 imágenes, todas ellas de  $224 \times 224$  píxeles, con valores de píxeles que van de  $-1$  a  $1$ . ¡Perfecto!

Dado que el conjunto de datos no es muy grande, un poco de aumento de datos sin duda será útil. Creemos un modelo de aumento de datos que incorporaremos en nuestro modelo final. Durante el entrenamiento, volteará aleatoriamente las imágenes horizontalmente, las rotará un poco y ajustará el contraste:

```
data_augmentation =
tf.keras.Sequential([
    tf.keras.layers.RandomFlip(mode="horizontal", semilla=42),
    tf.keras.layers.RandomRotation(factor=0.05, semilla=42), tf.keras.layers .Contraste
aleatorio (factor = 0,2, semilla = 42)
])
```

## CONSEJO

La clase `tf.keras.preprocessing.image.ImageDataGenerator` facilita cargar imágenes desde el disco y aumentarlas de varias maneras: puede desplazar cada imagen, rotarla, cambiar su escala, voltearla horizontal o verticalmente, cortarla o aplicar cualquier función de transformación que deseé. Esto es muy conveniente para proyectos simples. Sin embargo, una canalización `tf.data` no es mucho más complicada y, en general, es más rápida. Además, si tiene una GPU e incluye las capas de preprocessamiento o aumento de datos dentro de su modelo, se beneficiarán de la aceleración de la GPU durante el entrenamiento.

A continuación, carguemos un modelo Xception, previamente entrenado en ImageNet. Excluimos la parte superior de la red configurando `include_top=False`. Esto excluye la capa de agrupación promedio global y la capa de salida densa. Luego agregamos nuestra propia capa de agrupación promedio global (alimentándola con la salida del modelo base), seguida de una capa de salida densa con una unidad por clase, usando la función de activación softmax. Finalmente, envolvemos todo esto en un Modelo Keras:

```
base_model =
tf.keras.applications.Xception(pesos="imagenet",

include_top=False)
promedio = tf.keras.layers.GlobalAveragePooling2D()(base_model.output) salida =
tf.keras.layers.Dense(n_classes, activación="softmax") (promedio) modelo =
tf.keras.Model(entradas =modelo_base.entrada, salidas=salida)
```

Como se explica en [el Capítulo 11](#), normalmente es una buena idea congelar los pesos de las capas previamente entrenadas, al menos al comienzo del entrenamiento:

```
para capa en base_model.layers: capa.trainable
    = False
```

## ADVERTENCIA

Dado que nuestro modelo utiliza las capas del modelo base directamente, en lugar del objeto `base_model` en sí, establecer `base_model.trainable=False` no tendría ningún efecto.

Finalmente, podemos compilar el modelo y comenzar a entrenar:

```
optimizador = tf.keras.optimizers.SGD(learning_rate=0.1, impulse=0.9)

model.compile(loss="sparse_categorical_crossentropy",
optimizador=optimizador,
métricas=["exactitud"])
historial = model.fit(train_set, validation_data=conjunto_válido, épocas = 3)
```

#### ADVERTENCIA

Si está ejecutando Colab, asegúrese de que el tiempo de ejecución utilice una GPU:  
seleccione Tiempo de ejecución → "Cambiar tipo de tiempo de ejecución", elija "GPU" en el menú desplegable "Acelerador de hardware" y luego haga clic en Guardar. Es posible entrenar el modelo sin una GPU, pero será terriblemente lento (minutos por época, en lugar de segundos).

Después de entrenar el modelo durante algunas épocas, su precisión de validación debería alcanzar un poco más del 80% y luego dejar de mejorar. Esto significa que las capas superiores ahora están bastante bien entrenadas y estamos listos para descongelar algunas de las capas superiores del modelo base y luego continuar con el entrenamiento. Por ejemplo, descongelemos las capas 56 y superiores (ese es el comienzo de la unidad residual 7 de 14, como puede ver si enumera los nombres de las capas):

```
para capa en base_model.layers[56:]:
    Layer.trainable = True
```

No olvide compilar el modelo cada vez que congele o descongele capas.

También asegúrese de utilizar una tasa de aprendizaje mucho más baja para evitar dañar los pesos previamente entrenados:

```
optimizador = tf.keras.optimizers.SGD(learning_rate=0.01, impulse=0.9)

model.compile(loss="sparse_categorical_crossentropy",
optimizador=optimizador,
métricas=["exactitud"])
historial = model.fit(train_set, validation_data=conjunto_válido, épocas = 10)
```

Este modelo debería alcanzar alrededor del 92% de precisión en el conjunto de prueba, en solo unos minutos de entrenamiento (con una GPU). Si ajusta los hiperparámetros, reduce la tasa de aprendizaje y entrena durante un poco más de tiempo, debería poder alcanzar entre el 95% y el 97%. ¡Con eso, puedes comenzar a entrenar clasificadores de imágenes increíbles en tus propias imágenes y clases! Pero la visión por computadora es mucho más que una simple clasificación. Por ejemplo, ¿qué pasa si también quieras saber dónde está la flor en una imagen? Veamos esto ahora.

## Clasificación y localización

La localización de un objeto en una imagen se puede expresar como una tarea de regresión, como se analiza en el [Capítulo 10](#): para predecir un cuadro delimitador alrededor del objeto, un enfoque común es predecir las coordenadas horizontales y verticales del centro del objeto, así como su altura y ancho. Esto significa que tenemos cuatro números para predecir. No requiere muchos cambios en el modelo; solo necesitamos agregar una segunda capa de salida densa con cuatro unidades (generalmente encima de la capa de agrupación promedio global) y se puede entrenar usando la pérdida de MSE:

```
base_model =
tf.keras.applications.Xception(pesos="imagenet",
                                   include_top=False)
avg = tf.keras.layers.GlobalAveragePooling2D()(base_model.output)
class_output = tf.keras.layers.Dense(n_classes, activation="softmax")(avg)
loc_output = tf.keras.layers.Dense(4)(avg)

modelo = tf.keras.Model(entradas=base_model.input,
                       salidas=[class_output, loc_output])
modelo.compile(loss=["sparse_categorical_crossentropy",
                     "mse"], loss_weights=[0.8, 0.2], # depende de lo que
# preocupe más por
# optimizador=optimizador, métricas=["precisión"])
```

Pero ahora tenemos un problema: el conjunto de datos de flores no tiene cuadros delimitadores alrededor de las flores. Por lo tanto, debemos agregarlos nosotros mismos. Esta suele ser una de las partes más difíciles y costosas de un proyecto de aprendizaje automático: conseguir las etiquetas. Es una buena idea dedicar tiempo a buscar las herramientas adecuadas. Para anotar imágenes con cuadros delimitadores, es posible que desee utilizar una herramienta de etiquetado de imágenes de código abierto como VGG Image Annotator, LabelImg, OpenLabeler o ImgLab, o quizás una herramienta comercial como LabelBox o

Supervisadamente. También es posible que desee considerar plataformas de crowdsourcing como Amazon Mechanical Turk si tiene una gran cantidad de imágenes para anotar. Sin embargo, es mucho trabajo configurar una plataforma de crowdsourcing, preparar el formulario para enviarlo a los trabajadores, supervisarlos y garantizar que la calidad de los cuadros delimitadores que producen sea buena, así que asegúrese de que valga la pena. esfuerzo. Adriana Kovashka et al. escribió un artículo<sup>29</sup> muy práctico sobre crowdsourcing en visión por computadora. Te recomiendo que lo consultes, incluso si no planeas utilizar crowdsourcing. Si sólo hay unos cientos o incluso un par de miles de imágenes para etiquetar, y no planeas hacerlo con frecuencia, puede ser preferible hacerlo tú mismo: con las herramientas adecuadas, solo te llevará unos días. y también obtendrá una mejor comprensión de su conjunto de datos y su tarea.

Ahora supongamos que ha obtenido los cuadros delimitadores para cada imagen en el conjunto de datos de flores (por ahora asumiremos que hay un único cuadro delimitador por imagen). Luego deberá crear un conjunto de datos cuyos elementos serán lotes de imágenes preprocesadas junto con sus etiquetas de clase y sus cuadros delimitadores. Cada elemento debe ser una tupla del formato (imágenes, (class\_labels,bounding\_boxes)). ¡Entonces estás listo para entrenar tu modelo!

## CONSEJO

Los cuadros delimitadores deben normalizarse para que las coordenadas horizontales y verticales, así como la altura y el ancho, oscilen entre 0 y 1. Además, es común predecir la raíz cuadrada de la altura y el ancho en lugar de la altura y el ancho. directamente: de esta manera, un error de 10 píxeles para un cuadro delimitador grande no se penalizará tanto como un error de 10 píxeles para un cuadro delimitador pequeño.

El MSE a menudo funciona bastante bien como función de costos para entrenar el modelo, pero no es una buena métrica para evaluar qué tan bien el modelo puede predecir cuadros delimitadores. La métrica más común para esto es la intersección sobre unión (IoU): el área de superposición entre el cuadro delimitador previsto y el cuadro delimitador objetivo, dividida por el área de su unión (consulte la Figura 14-24). En Keras, lo implementa la clase `tf.keras.metrics.MeanIoU`.

Clasificar y localizar un solo objeto es bueno, pero ¿qué pasa si las imágenes contienen varios objetos (como suele ser el caso en el conjunto de datos de flores)?



Figura 14-24. Métrica IoU para cuadros delimitadores

## Detección de objetos

La tarea de clasificar y localizar múltiples objetos en una imagen se llama detección de objetos. Hasta hace unos años, un enfoque común era tomar una CNN entrenada para clasificar y ubicar un solo objeto aproximadamente centrado en la imagen, luego deslizar esta CNN a través de la imagen y hacer predicciones en cada paso. La CNN generalmente fue entrenada para predecir no solo probabilidades de clase y un cuadro delimitador, sino también una puntuación de objetividad: esta es la probabilidad estimada de que la imagen contenga un objeto centrado cerca del medio. Esta es una salida de clasificación binaria; puede ser producida por una capa de salida densa con una sola unidad, usando la función de activación sigmoidea y entrenado usando la pérdida binaria de entropía cruzada.

### NOTA

En lugar de una puntuación de objetividad, a veces se agregaba una clase "sin objeto", pero en general esto no funcionó tan bien: las preguntas "¿Hay un objeto presente?" y "¿Qué tipo de objeto es?" es mejor responderlas por separado.

Este enfoque de CNN deslizante se ilustra en [la Figura 14-25](#). En este ejemplo, la imagen se cortó en una cuadrícula de  $5 \times 7$  y vemos una CNN, la gruesa

rectángulo negro: deslizándose por todas las regiones de  $3 \times 3$  y haciendo predicciones en cada paso.

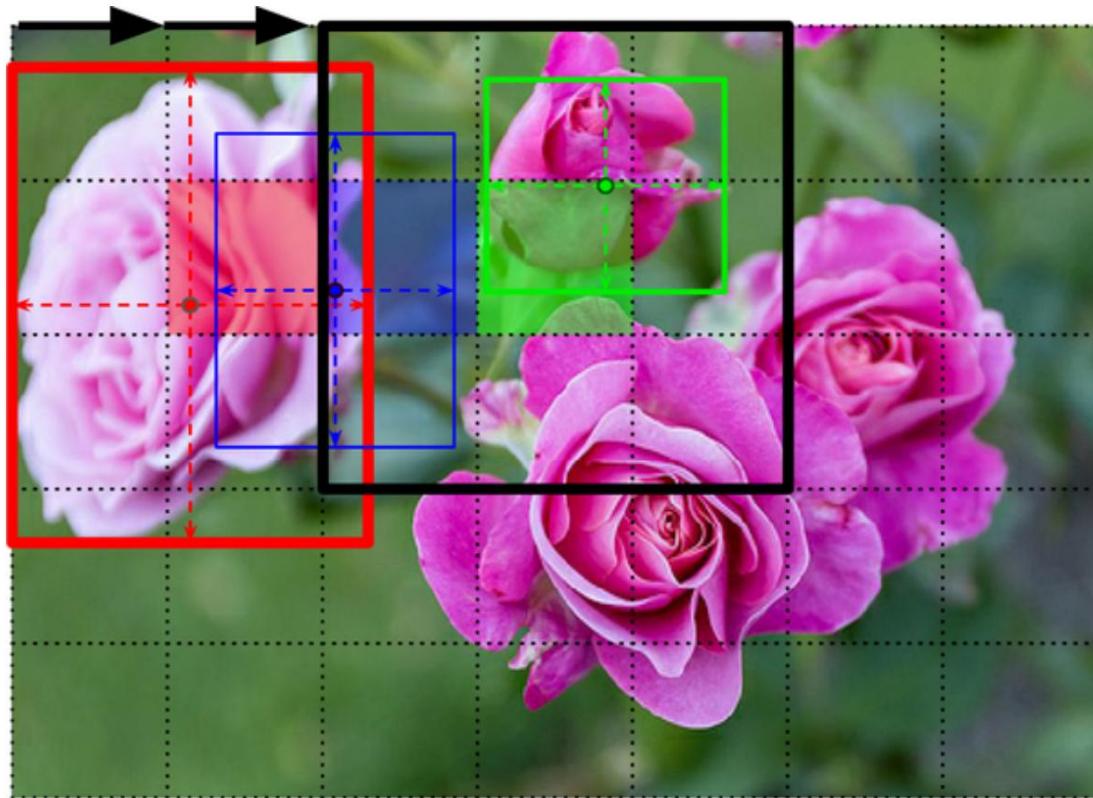


Figura 14-25. Detectar múltiples objetos deslizando una CNN por la imagen

En esta figura, la CNN ya ha hecho predicciones para tres de estas regiones  $3 \times 3$ :

- Al observar la región superior izquierda de  $3 \times 3$  (centrada en la celda de la cuadrícula sombreada en rojo ubicada en la segunda fila y la segunda columna), detectó la rosa más a la izquierda. Observe que el cuadro delimitador previsto excede el límite de esta región de  $3 \times 3$ . Eso está absolutamente bien: aunque la CNN no pudo ver la parte inferior de la rosa, pudo hacer una suposición razonable sobre dónde podría estar. También predijo las probabilidades de clase, dando una alta probabilidad a la clase "rosa". Por último, predijo una puntuación de objetividad bastante alta, ya que el centro del cuadro delimitador se encuentra dentro de la celda de la cuadrícula central (en esta figura, la puntuación de objetividad está representada por el grosor del cuadro delimitador).
- Al observar la siguiente región de  $3 \times 3$ , una celda de la cuadrícula a la derecha (centrada en el cuadrado sombreado en azul), no detectó ninguna flor centrada en esa región, por lo que predijo una puntuación de objetividad muy baja;

por lo tanto, el cuadro delimitador previsto y las probabilidades de clase se pueden ignorar con seguridad. Puede ver que el cuadro delimitador previsto no era bueno de todos modos.

- finalmente, al mirar la siguiente región de  $3 \times 3$ , nuevamente una celda de la cuadrícula a la derecha (centrada en la celda sombreada en verde), detectó la rosa en la parte superior, aunque no perfectamente: esta rosa no está bien centrada dentro de esta región. por lo que la puntuación de objetividad prevista no fue muy alta.

Puede imaginar cómo deslizar la CNN por toda la imagen le daría un total de 15 cuadros delimitadores predichos, organizados en una cuadrícula de  $3 \times 5$ , con cada cuadro delimitador acompañado de sus probabilidades de clase estimadas y su puntuación de objetividad. Dado que los objetos pueden tener diferentes tamaños, es posible que desees deslizar la CNN nuevamente a lo largo de regiones más grandes de  $4 \times 4$  para obtener aún más cuadros delimitadores.

Esta técnica es bastante sencilla, pero como puede ver, a menudo detectará el mismo objeto varias veces, en posiciones ligeramente diferentes. Se necesita algo de posprocesamiento para eliminar todos los cuadros delimitadores innecesarios.

Un enfoque común para esto se llama supresión no máxima. Así es como funciona:

1. Primero, elimine todos los cuadros delimitadores para los cuales la puntuación de objetividad está por debajo de algún umbral: dado que CNN cree que no hay ningún objeto en esa ubicación, el cuadro delimitador es inútil.
2. Encuentre el cuadro delimitador restante con la puntuación de objetividad más alta. y elimine todos los demás cuadros delimitadores restantes que se superponen mucho con él (por ejemplo, con un IoU superior al 60%). Por ejemplo, en [la Figura 14-25](#), el cuadro delimitador con la puntuación máxima de objetividad es el cuadro delimitador grueso sobre la rosa más a la izquierda. El otro cuadro delimitador que toca esta misma rosa se superpone mucho con el cuadro delimitador máximo, por lo que lo eliminaremos (aunque en este ejemplo ya se habría eliminado en el paso anterior).
3. Repita el paso 2 hasta que no queden más cuadros delimitadores de los que deshacerse.

Este enfoque simple para la detección de objetos funciona bastante bien, pero requiere ejecutar CNN muchas veces (15 veces en este ejemplo), por lo que es bastante lento.

Afortunadamente, existe una forma mucho más rápida de deslizar una CNN a través de una imagen: utilizando una red totalmente convolucional (FCN).

## Redes totalmente convolucionales La idea de FCN

se introdujo por primera vez en un [artículo de 2015](#).<sup>30</sup> de Jonathan Long et al., para la segmentación semántica (la tarea de clasificar cada píxel de una imagen según la clase del objeto al que pertenece). Los autores señalaron que se podrían reemplazar las capas densas en la parte superior de una CNN con capas convolucionales. Para entender esto, veamos un ejemplo: supongamos que una capa densa con 200 neuronas se encuentra encima de una capa convolucional que genera 100 mapas de características, cada uno de un tamaño de  $7 \times 7$  (este es el tamaño del mapa de características, no el tamaño del núcleo). Cada neurona calculará una suma ponderada de todas las  $100 \times 7 \times 7$  activaciones de la capa convolucional (más un término de sesgo). Ahora veamos qué sucede si reemplazamos la capa densa con una capa convolucional usando 200 filtros, cada uno de tamaño  $7 \times 7$ , y con un relleno "válido". Esta capa generará 200 mapas de características, cada uno de  $1 \times 1$  (ya que el núcleo tiene exactamente el tamaño de los mapas de características de entrada y estamos usando un relleno "válido"). En otras palabras, generará 200 números, tal como lo hizo la capa densa; y si observa de cerca los cálculos realizados por una capa convolucional, notará que estos números serán exactamente los mismos que los que produjo la capa densa. La única diferencia es que la salida de la capa densa fue un tensor de forma [tamaño de lote, 200], mientras que la capa convolucional generará un tensor de forma [tamaño de lote, 1, 1, 200].

### CONSEJO

Para convertir una capa densa en una capa convolucional, la cantidad de filtros en la capa convolucional debe ser igual a la cantidad de unidades en la capa densa, el tamaño del filtro debe ser igual al tamaño de los mapas de características de entrada y debe usar relleno "válido". La zancada se puede establecer en 1 o más, como verá en breve.

¿Porque es esto importante? Bueno, mientras que una capa densa espera un tamaño de entrada específico (ya que tiene un peso por característica de entrada), una capa convolucional procesa felizmente imágenes de cualquier tamaño<sup>31</sup> (sin embargo, espera que sus entradas tengan un número específico de canales, ya que cada núcleo contiene un conjunto diferente de pesos para cada canal de entrada). Dado que un FCN contiene solo capas convolucionales (y capas de agrupación, que tienen la misma propiedad), se puede entrenar y ejecutar en imágenes de cualquier tamaño.

Por ejemplo, supongamos que ya hemos entrenado una CNN para la clasificación y localización de flores. Fue entrenado en imágenes de  $224 \times 224$  y genera 10 números:

- Las salidas 0 a 4 se envían a través de la función de activación softmax, y esto da las probabilidades de clase (una por clase).
- La salida 5 se envía a través de la función de activación sigmoidea, y esto proporciona la puntuación de objetividad.
- Las salidas 6 y 7 representan las coordenadas centrales del cuadro delimitador; también pasan por una función de activación sigmoidea para garantizar que oscilen entre 0 y 1.
- Por último, las salidas 8 y 9 representan la altura y el ancho del cuadro delimitador; no pasan por ninguna función de activación para permitir que los cuadros delimitadores se extiendan más allá de los bordes de la imagen.

Ahora podemos convertir las capas densas de CNN en capas convolucionales. De hecho, ni siquiera necesitamos volver a entrenarlo; ¡Podemos simplemente copiar los pesos de las capas densas a las capas convolucionales! Alternativamente, podríamos haber convertido la CNN en una FCN antes del entrenamiento.

Ahora supongamos que la última capa convolucional antes de la capa de salida (también llamada capa de cuello de botella) genera mapas de características de  $7 \times 7$  cuando la red recibe una imagen de  $224 \times 224$  (consulte el lado izquierdo de la Figura 14-26). Si alimentamos al FCN con una imagen de  $448 \times 448$  (consulte el lado derecho de la Figura 14-26), la capa de cuello de botella ahora generará mapas de características de  $14 \times 14$ . Dado que la capa de salida densa fue reemplazada por una capa convolucional usando 10 filtros de tamaño  $7 \times 7$ , con relleno "válido" y zancada 1, la salida estará compuesta por 10 mapas de características, cada uno de tamaño  $8 \times 8$  (desde  $14 - 7 + 1 = 8$ ). En otras palabras, el FCN procesará la imagen completa solo una vez y generará una cuadrícula de  $8 \times 8$  donde cada celda contiene 10 números (5 probabilidades de clase, 1 puntuación de objetividad y 4 coordenadas del cuadro delimitador). Es exactamente como tomar la CNN original y deslizarla por la imagen usando 8 pasos por fila y 8 pasos por columna. Para visualizar esto, imagine cortar la imagen original en una cuadrícula de  $14 \times 14$  y luego deslizar una ventana de  $7 \times 7$  a través de esta cuadrícula; Habrá  $8 \times 8 = 64$  ubicaciones posibles para la ventana, por lo tanto, predicciones de  $8 \times 8$ . Sin embargo, el enfoque FCN es mucho más eficiente, ya que la red sólo mira el

imagen una vez. De hecho, You Only Look Once (YOLO) es el nombre de una arquitectura de detección de objetos muy popular, que veremos a continuación.

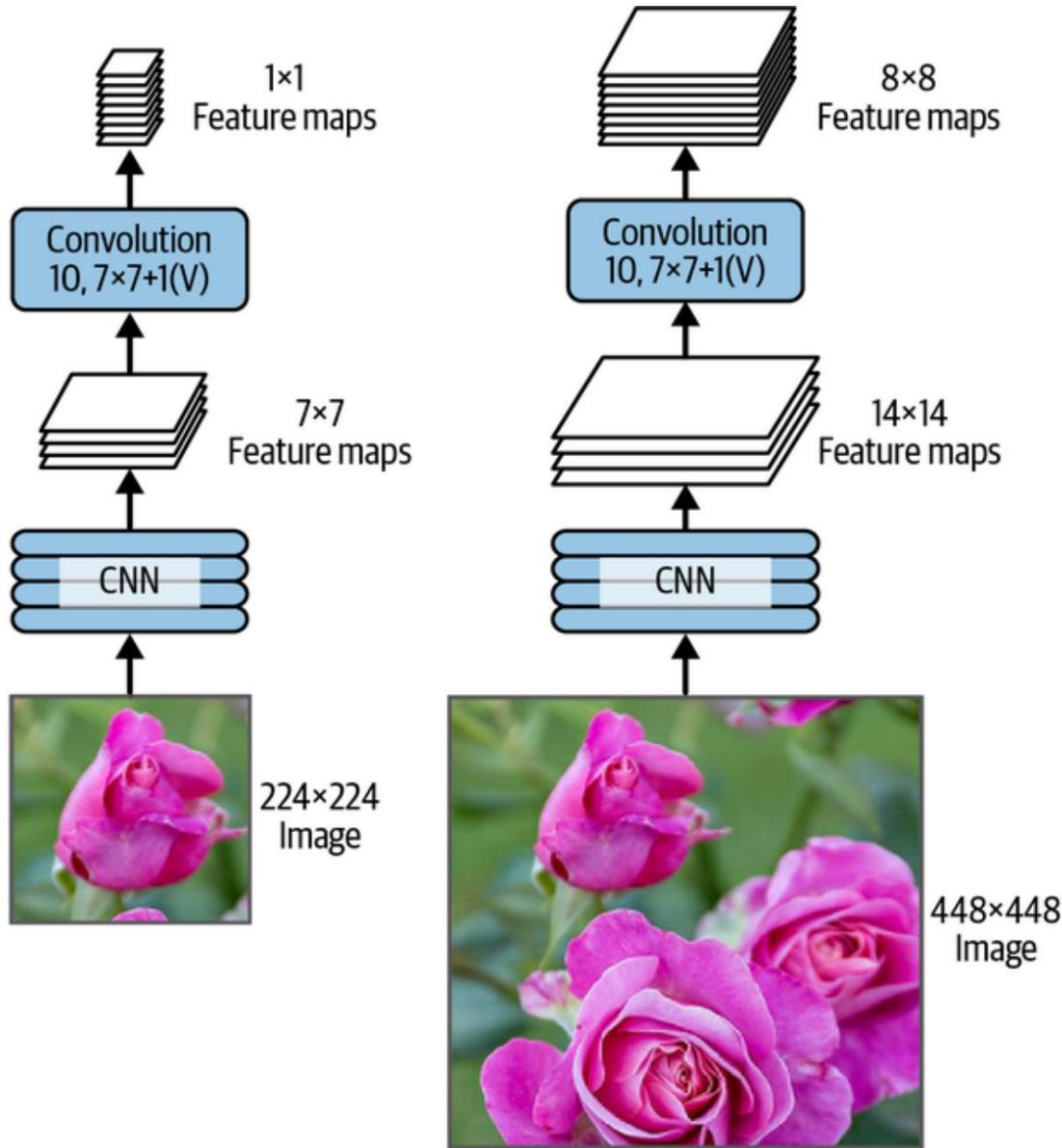


Figura 14-26. La misma red totalmente convolucional que procesa una imagen pequeña (izquierda) y una grande (derecha)

## Sólo miras una vez

YOLO es una arquitectura de detección de objetos rápida y precisa propuesta por Joseph Redmon et al. en un [artículo de 2015](#). Es tan rápido que puede ejecutarse en tiempo real en un vídeo, como se ve en la [demostración de Redmon](#). La arquitectura de YOLO es bastante similar a la que acabamos de comentar, pero con algunas diferencias importantes:

- Para cada celda de la cuadrícula, YOLO solo considera objetos cuyo centro del cuadro delimitador se encuentra dentro de esa celda. Las coordenadas del cuadro delimitador son relativas a esa celda, donde (0, 0) significa la esquina superior izquierda de la celda y (1, 1) significa la esquina inferior derecha. Sin embargo, la altura y el ancho del cuadro delimitador pueden extenderse mucho más allá de la celda.
- Genera dos cuadros delimitadores para cada celda de la cuadrícula (en lugar de solo uno), lo que permite que el modelo maneje casos en los que dos objetos están tan cerca uno del otro que los centros de sus cuadros delimitadores se encuentran dentro de la misma celda. Cada cuadro delimitador también viene con su propia puntuación de objetividad.
- YOLO también genera una distribución de probabilidad de clase para cada celda de la cuadrícula, prediciendo 20 probabilidades de clase por celda de la cuadrícula, ya que YOLO fue entrenado en el conjunto de datos PASCAL VOC, que contiene 20 clases. Esto produce un mapa de probabilidad de clase aproximado. Tenga en cuenta que el modelo predice una distribución de probabilidad de clase por celda de la cuadrícula, no por cuadro delimitador. Sin embargo, es posible estimar las probabilidades de clase para cada cuadro delimitador durante el posprocesamiento, midiendo qué tan bien coincide cada cuadro delimitador con cada clase en el mapa de probabilidad de clase. Por ejemplo, imagine la imagen de una persona parada frente a un automóvil. Habrá dos cuadros delimitadores: uno horizontal grande para el automóvil y otro vertical más pequeño para la persona. Estos cuadros delimitadores pueden tener sus centros dentro de la misma celda de la cuadrícula. Entonces, ¿cómo podemos saber qué clase se debe asignar a cada cuadro delimitador? Bueno, el mapa de probabilidad de clase contendrá una región grande donde la clase "automóvil" es dominante, y dentro de él habrá una región más pequeña donde la clase "persona" es dominante. Con suerte, el cuadro delimitador del automóvil coincidirá aproximadamente con la región del "automóvil", mientras que el cuadro delimitador de la persona coincidirá aproximadamente con la región de la "persona": esto permitirá asignar la clase correcta a cada cuadro delimitador.

YOLO se desarrolló originalmente utilizando Darknet, un marco de aprendizaje profundo de código abierto desarrollado inicialmente en C por Joseph Redmon, pero pronto se transfirió a TensorFlow, Keras, PyTorch y más. Se mejoró continuamente a lo largo de los años, con YOLOv2, YOLOv3 y YOLO9000 (nuevamente por Joseph Redmon et al.), YOLOv4 (por Alexey Bochkovskiy et al.), YOLOv5 (por Glenn Jocher) y PP-YOLO (por Xiang Long). et al.).

Cada versión trajo algunas mejoras impresionantes en velocidad y precisión, utilizando una variedad de técnicas; por ejemplo, YOLOv3 aumentó la precisión en parte gracias a los anclajes anteriores, aprovechando el hecho de que algunas formas de cuadros delimitadores son más probables que otras, dependiendo de la clase (por ejemplo, las personas tienden a tener cuadros delimitadores verticales, mientras que los automóviles generalmente no los tienen). También aumentaron el número de cuadros delimitadores por celda de la cuadrícula, entrenaron en diferentes conjuntos de datos con muchas más clases (hasta 9000 clases organizadas en una jerarquía en el caso de YOLO9000), agregaron conexiones de omisión para recuperar parte de la resolución espacial que es perdido en la CNN (discutiremos esto en breve, cuando analicemos la segmentación semántica) y mucho más. También hay muchas variantes de estos modelos, como YOLOv4-tiny, que está optimizado para entrenarse en máquinas menos potentes y que puede funcionar extremadamente rápido (¡a más de 1000 fotogramas por segundo!), pero con una precisión media ligeramente inferior ( mapa).

## PRECISIÓN PROMEDIO MEDIA

Una métrica muy común utilizada en tareas de detección de objetos es la precisión promedio media. "Promedio medio" suena un poco redundante, ¿no?

Para comprender esta métrica, volvamos a dos métricas de clasificación que analizamos en [el Capítulo 3](#): precisión y recuperación. Recuerde la contrapartida: cuanto mayor sea la recuperación, menor será la precisión. Puede visualizar esto en una curva de precisión/recuperación ([consulte la Figura 3-6](#)). Para resumir esta curva en un solo número, podríamos calcular su área bajo la curva (AUC). Pero tenga en cuenta que la curva de precisión/recuperación puede contener algunas secciones donde la precisión realmente aumenta cuando la recuperación aumenta, especialmente en valores de recuperación bajos (puede ver esto en la parte superior izquierda de la Figura 3-6). Ésta es una de las motivaciones de la métrica mAP.

Supongamos que el clasificador tiene una precisión del 90% con una recuperación del 10%, pero una precisión del 96% con una recuperación del 20%. Realmente no hay ninguna compensación aquí: simplemente tiene más sentido usar el clasificador con una recuperación del 20% en lugar de con una recuperación del 10%, ya que obtendrá una mayor recuperación y mayor precisión. Entonces, en lugar de mirar la precisión con una recuperación del 10%, realmente deberíamos mirar la precisión máxima que el clasificador puede ofrecer con al menos un 10% de recuperación. Sería el 96%, no el 90%. Por lo tanto, una forma de tener una idea clara del rendimiento del modelo es calcular la precisión máxima que se puede obtener con al menos un 0% de recuperación, luego un 10% de recuperación, un 20%, y así sucesivamente hasta un 100%, y luego calcular la media de estas precisiones máximas. Esto se denomina métrica de precisión promedio (AP). Ahora, cuando hay más de dos clases, podemos calcular el AP para cada clase y luego calcular el AP medio (mAP). ¡Eso es todo!

En un sistema de detección de objetos, existe un nivel adicional de complejidad: ¿qué pasa si el sistema detecta la clase correcta, pero en la ubicación incorrecta (es decir, el cuadro delimitador está completamente fuera de lugar)? Seguramente no deberíamos considerar esto como una predicción positiva. Un enfoque consiste en definir un umbral de IoU: por ejemplo, podemos considerar que una predicción es correcta sólo si el IoU es mayor que, digamos, 0,5 y la clase predicha es correcta. El mapa correspondiente generalmente se indica como mAP@0.5 (o mAP@50%, o a veces simplemente AP). En algunas competiciones (como el desafío PASCAL VOC), esto es lo que se hace. En otros (como la competencia COCO), el mAP se calcula para diferentes umbrales de IoU (0,50,

0,55, 0,60,..., 0,95), y la métrica final es la media de todos estos mAP (anotado como mAP@[.50:.95] o mAP@[.50:0.05:.95]). Sí, ese es un promedio medio medio.

Muchos modelos de detección de objetos están disponibles en TensorFlow Hub, a menudo con pesos previamente entrenados, como YOLOv5, <sup>34</sup> SSD, R-CNN<sup>35</sup> más rápido, y <sup>36</sup> EfficientDet.

SSD y EfficientDet son modelos de detección de "mirar una vez", similares a YOLO. EfficientDet se basa en la arquitectura convolucional EfficientNet. R-CNN más rápido es más complejo: la imagen primero pasa por una CNN, luego la salida se pasa a una red de propuesta de región (RPN) que propone cuadros delimitadores que tienen más probabilidades de contener un objeto; Luego se ejecuta un clasificador para cada cuadro delimitador, según la salida recortada de CNN. El mejor lugar para comenzar a utilizar estos modelos es el excelente tutorial de detección [de objetos](#) de TensorFlow Hub .

Hasta ahora, sólo hemos considerado la detección de objetos en imágenes individuales. Pero ¿qué pasa con los vídeos? Los objetos no sólo deben detectarse en cada cuadro, sino que también deben rastrearse a lo largo del tiempo. Echemos ahora un vistazo rápido al seguimiento de objetos.

## Seguimiento de objetos

El seguimiento de objetos es una tarea desafiante: los objetos se mueven, pueden crecer o encogerse a medida que se acercan o se alejan de la cámara, su apariencia puede cambiar a medida que giran o se mueven a diferentes condiciones de iluminación o fondos, pueden quedar temporalmente ocultos por otros objetos, y así en.

Uno de los sistemas de seguimiento de objetos más populares es DeepSORT. Se basa en <sup>38</sup> una combinación de algoritmos clásicos y aprendizaje profundo:

- Utiliza filtros de Kalman para estimar la posición actual más probable de un objeto dadas las detecciones anteriores y suponiendo que los objetos tienden a moverse a una velocidad constante.
- Utiliza un modelo de aprendizaje profundo para medir la semejanza entre las nuevas detecciones y los objetos rastreados existentes.

- Por último, utiliza el algoritmo húngaro para asignar nuevas detecciones a objetos rastreados existentes (o a nuevos objetos rastreados): este algoritmo encuentra eficientemente la combinación de asignaciones que minimiza la distancia entre las detecciones y las posiciones previstas de los objetos rastreados, al mismo tiempo que minimiza la discrepancia de apariencia.

Por ejemplo, imagina una bola roja que acaba de rebotar en una bola azul que viaja en dirección opuesta. Basándose en las posiciones previas de las bolas, el filtro de Kalman predecirá que las bolas se atravesarán entre sí: de hecho, supone que los objetos se mueven a una velocidad constante, por lo que no esperará el rebote. Si el algoritmo húngaro solo considerara las posiciones, felizmente asignaría las nuevas detecciones a las bolas equivocadas, como si acabaran de atravesarse e intercambiar colores. Pero gracias a la medida de semejanza, el algoritmo húngaro se dará cuenta del problema. Suponiendo que las bolas no sean muy similares, el algoritmo asignará las nuevas detecciones a las bolas correctas.

CONSEJO

Hay algunas implementaciones de DeepSORT disponibles en GitHub, incluida una implementación de TensorFlow de YOLOv4 + DeepSORT: <https://github.com/theAIGuyzCode/yolov4-deepsort>.

Hasta ahora hemos localizado objetos usando cuadros delimitadores. Esto suele ser suficiente, pero a veces es necesario localizar objetos con mucha más precisión (por ejemplo, para eliminar el fondo detrás de una persona durante una videoconferencia). Veamos cómo bajar al nivel de píxeles.

## Segmentación semántica

En la segmentación semántica, cada píxel se clasifica según la clase del objeto al que pertenece (por ejemplo, carretera, automóvil, peatón, edificio, etc.), como se muestra en la **Figura 14-27**. Tenga en cuenta que no se distinguen diferentes objetos de la misma clase . Por ejemplo, todas las bicicletas en el lado derecho de la imagen segmentada terminan como un gran conjunto de píxeles. La principal dificultad en esta tarea es que cuando las imágenes pasan por una CNN normal, van perdiendo gradualmente su resolución espacial (debido a las capas con pasos mayores a 1); entonces, un

La CNN normal puede terminar sabiendo que hay una persona en algún lugar en la parte inferior izquierda de la imagen, pero no será mucho más preciso que eso.



Figura 14-27. Segmentación semántica

Al igual que con la detección de objetos, existen muchos enfoques diferentes para abordar este problema, algunos bastante complejos. Sin embargo, en el artículo de 2015 de Jonathan Long et al. se propuso una solución bastante simple. Mencioné anteriormente, en redes totalmente convolucionales. Los autores comienzan tomando una CNN previamente entrenada y convirtiéndola en una FCN. La CNN aplica un paso general de 32 a la imagen de entrada (es decir, si suma todos los pasos mayores que 1), lo que significa que la última capa genera mapas de características que son 32 veces más pequeños que la imagen de entrada. Esto es claramente demasiado burdo, por lo que agregaron una única capa de muestreo que multiplica la resolución por 32.

Hay varias soluciones disponibles para el muestreo superior (aumentar el tamaño de una imagen), como la interpolación bilineal, pero eso solo funciona razonablemente.

[39](#) pozos hasta  $\times 4$  o  $\times 8$ . En su lugar, utilizan una capa convolucional transpuesta: esto equivale a estirar primero la imagen insertando filas y columnas vacías (llenas de ceros) y luego realizar una convolución regular (consulte la [Figura 14-28](#)). Alternativamente, algunas personas prefieren pensar en ella como una capa convolucional regular que utiliza pasos fraccionarios (por ejemplo, el paso es 1/2 en la [Figura 14-28](#)). La capa convolucional transpuesta se puede inicializar para realizar algo cercano a la interpolación lineal, pero como es una capa entrenable, aprenderá a funcionar mejor durante el entrenamiento. En Keras, puedes usar la capa Conv2DTranspose.

**NOTA**

En una capa convolucional transpuesta, el paso define cuánto se estirará la entrada, no el tamaño de los pasos del filtro, por lo que cuanto mayor sea el paso, mayor será la salida (a diferencia de las capas convolucionales o de agrupación).

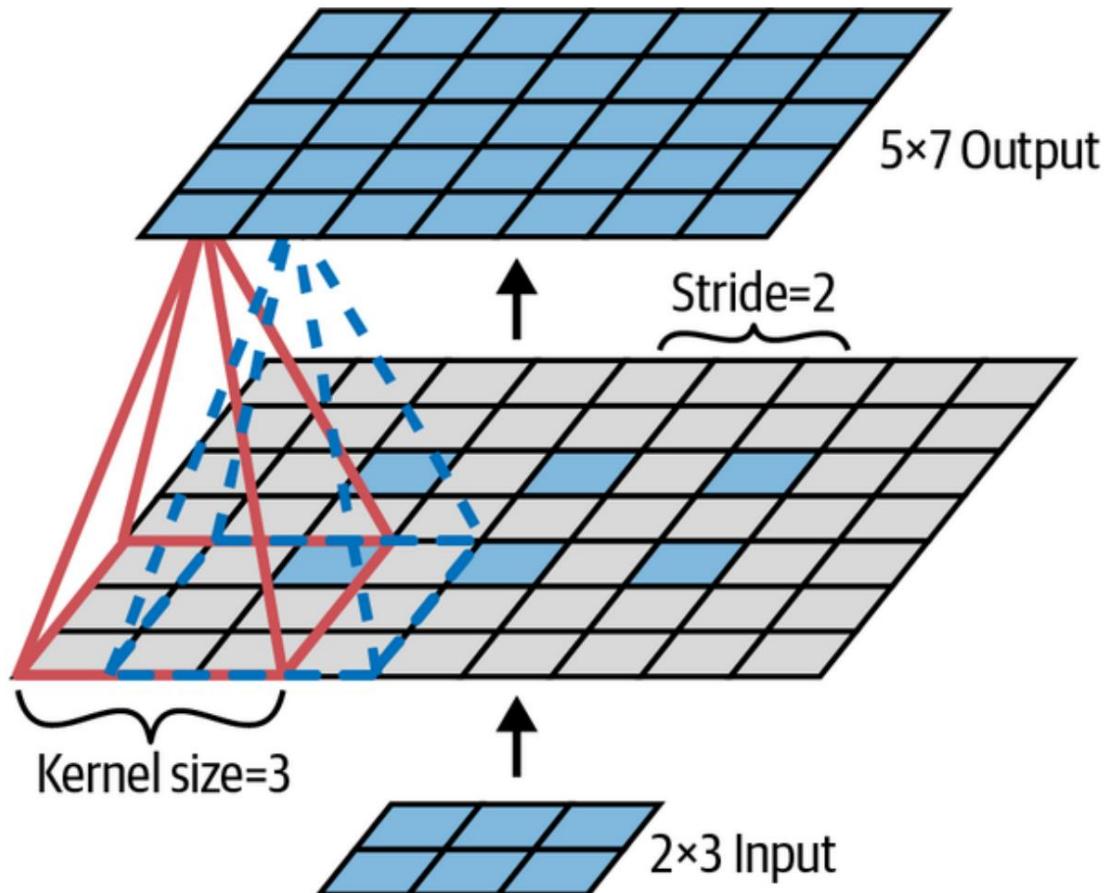


Figura 14-28. Muestreo mejorado utilizando una capa convolucional transpuesta

## OTRAS CAPAS CONVOLUCIONALES DE KERAS

Keras también ofrece algunos otros tipos de capas convolucionales:

`tf.keras.capas.Conv1D`

Una capa convolucional para entradas 1D, como series de tiempo o texto (secuencias de letras o palabras), como verá en el [Capítulo 15](#).

`tf.keras.capas.Conv3D`

Una capa convolucional para entradas 3D, como escaneos PET 3D.

`tasa_dilatación`

Establecer el hiperparámetro `dilation_rate` de cualquier capa convolucional en un valor de 2 o más crea una capa convolucional à-trous (“à trous” en francés significa “con agujeros”). Esto equivale a utilizar una capa convolucional regular con un filtro dilatado insertando filas y columnas de ceros (es decir, huecos). Por ejemplo, un filtro de  $1 \times 3$  igual a `[[1,2,3]]` puede dilatarse con una velocidad de dilatación de 4, lo que da como resultado un filtro dilatado de `[[1, 0, 0, 0, 2, 0, 0 , 0, 3]]`. Esto permite que la capa convolucional tenga un campo receptivo más grande sin costo computacional y sin utilizar parámetros adicionales.

Usar capas convolucionales transpuestas para el muestreo superior está bien, pero sigue siendo demasiado impreciso. Para hacerlo mejor, Long et al. agregaron conexiones de salto de capas inferiores: por ejemplo, aumentaron la muestra de la imagen de salida en un factor de 2 (en lugar de 32) y agregaron la salida de una capa inferior que tenía esta doble resolución. Luego, aumentaron el muestreo del resultado en un factor de 16, lo que llevó a un factor de aumento total de 32 (consulte [la Figura 14-29](#)). Esto recuperó parte de la resolución espacial que se perdió en capas de agrupación anteriores. En su mejor arquitectura, utilizaron una segunda conexión de salto similar para recuperar detalles aún más finos de una capa aún más baja. En resumen, la salida de la CNN original pasa por los siguientes pasos adicionales: aumentar la muestra  $\times 2$ , agregar la salida de una capa inferior (de la escala apropiada), aumentar la muestra  $\times 2$ , agregar la salida de una capa aún más baja y, finalmente, aumentar la muestra  $\times 8$ . Incluso es posible ampliar la escala más allá del tamaño de la imagen original: esto se puede utilizar para aumentar la resolución de una imagen, lo cual es una técnica llamada superresolución.

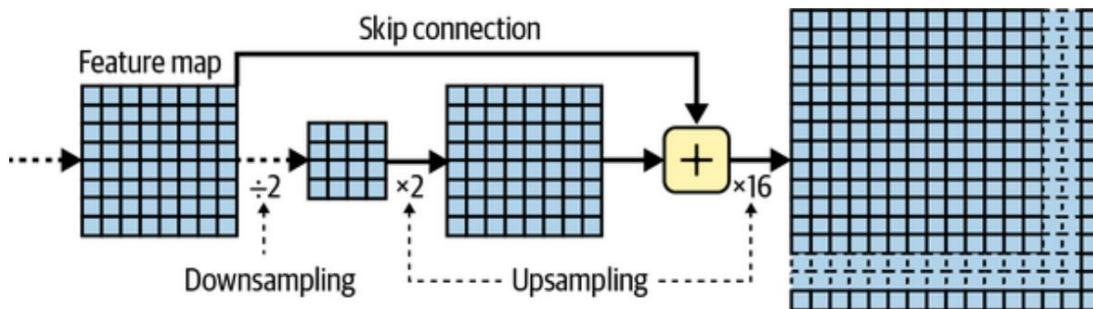


Figura 14-29. Saltar capas recuperan algo de resolución espacial de las capas inferiores

La segmentación de instancias es similar a la segmentación semántica, pero en lugar de fusionar todos los objetos de la misma clase en un gran conjunto, cada objeto se distingue de los demás (por ejemplo, identifica cada bicicleta individual). Por ejemplo, la arquitectura Mask R-CNN , propuesta en un artículo de 2017. de Kaiming He et al.,<sup>40</sup> amplía el modelo Faster R-CNN al producir adicionalmente una máscara de píxeles para cada cuadro delimitador. Entonces, no solo obtienes un cuadro delimitador alrededor de cada objeto, con un conjunto de probabilidades de clase estimadas, sino que también obtienes una máscara de píxeles que ubica los píxeles en el cuadro delimitador que pertenecen al objeto. Este modelo está disponible en TensorFlow Hub, previamente entrenado en el conjunto de datos COCO 2017. Sin embargo, el campo avanza rápidamente, por lo que si desea probar los mejores y más recientes modelos, consulte la sección de última generación de <https://paperswithcode.com>

Como puede ver, el campo de la visión profunda por computadora es vasto y de ritmo rápido, con todo tipo de arquitecturas apareciendo cada año. Casi todos se basan en redes neuronales convolucionales, pero desde 2020 otra arquitectura de red neuronal ha entrado en el espacio de la visión por computadora: los transformadores (que discutiremos en el Capítulo 16). El progreso logrado durante la última década ha sido asombroso, y los investigadores ahora se están centrando en problemas cada vez más difíciles, como el aprendizaje adversario (que intenta hacer que la red sea más resistente a imágenes diseñadas para engañarla), la explicabilidad (comprender por qué la red hace una clasificación específica), generación de imágenes realistas (a la que volveremos en el Capítulo 17), aprendizaje de un solo disparo (un sistema que puede reconocer un objeto después de haberlo visto solo una vez), predecir los siguientes fotogramas de un vídeo, combinar tareas de texto e imagen, y más.

Pasemos al siguiente capítulo, donde veremos cómo procesar datos secuenciales, como series de tiempo, utilizando redes neuronales recurrentes y redes neuronales convolucionales.

## Ejercicios

1. ¿Cuáles son las ventajas de una CNN sobre una DNN completamente conectada para la clasificación de imágenes?
2. Considere una CNN compuesta por tres capas convolucionales, cada una con  $3 \times 3$  núcleos, un paso de 2 y el "mismo" relleno. La capa más baja genera 100 mapas de características, la del medio genera 200 y la superior genera 400. Las imágenes de entrada son imágenes RGB de  $200 \times 300$  píxeles:
  - a. ¿Cuál es el número total de parámetros en la CNN?
  - b. Si usamos flotantes de 32 bits, ¿cuánta RAM al menos tendrá?  
¿La red requiere al hacer una predicción para una sola instancia?
  - c. ¿Qué pasa cuando entrenas con un mini lote de 50 imágenes?
3. Si su GPU se queda sin memoria mientras entrena una CNN, ¿cuáles son cinco cosas que podría intentar para resolver el problema?
4. ¿Por qué querrías agregar una capa de agrupación máxima en lugar de una capa convolucional con el mismo paso?
5. ¿Cuándo le gustaría agregar una capa de normalización de respuesta local?
6. ¿Puedes nombrar las principales innovaciones de AlexNet, en comparación con LeNet-5? ¿Qué pasa con las principales innovaciones en GoogLeNet, ResNet, SENet, Xception y EfficientNet?
7. ¿Qué es una red totalmente convolucional? ¿Cómo se puede convertir una capa densa en una capa convolucional?
8. ¿Cuál es la principal dificultad técnica de la segmentación semántica?
9. Cree su propia CNN desde cero e intente lograr la mayor precisión posible en MNIST.
10. Utilice el aprendizaje por transferencia para la clasificación de imágenes grandes, revisando estos pasos:
  - a. Cree un conjunto de entrenamiento que contenga al menos 100 imágenes por clase. Por ejemplo, puede clasificar sus propias imágenes según la ubicación (playa, montaña, ciudad, etc.) o, alternativamente, puede utilizar un conjunto de datos existente (por ejemplo, de TensorFlow Datasets).

- b. Divídalo en un conjunto de entrenamiento, un conjunto de validación y un conjunto de prueba.
  - C. Cree la canalización de entrada, aplique las operaciones de preprocesamiento adecuadas y, opcionalmente, agregue aumento de datos.
  - d. Ajuste un modelo previamente entrenado en este conjunto de datos.
11. Consulte el tutorial de transferencia de estilo de TensorFlow. Esta es una forma divertida de generar arte mediante el aprendizaje profundo.

Las soluciones a estos ejercicios están disponibles al final del cuaderno de este capítulo, en <https://homl.info/colab3>.

- 
- 1** David H. Hubel, "Actividad de unidad única en la corteza estriada de gatos sin restricciones", The Revista de fisiología 147 (1959): 226–238.
  - 2** David H. Hubel y Torsten N. Wiesel, "Campos receptivos de neuronas individuales en el Corteza estriada de gato", The Journal of Physiology 148 (1959): 574–591.
  - 3** David H. Hubel y Torsten N. Wiesel, "Campos receptivos y funciones Arquitectura de Monkey Striate Cortex", The Journal of Physiology 195 (1968): 215–243.
  - 4** Kunihiko Fukushima, "Neocognitron: un modelo de red neuronal autoorganizada para un mecanismo de reconocimiento de patrones que no se ve afectado por el cambio de posición", Biological Cybernetics 36 (1980): 193–202.
  - 5** Yann LeCun et al., "Aprendizaje basado en gradientes aplicado al reconocimiento de documentos", Actas del IEEE 86, no. 11 (1998): 2278–2324.
  - 6** Una convolución es una operación matemática que desliza una función sobre otra y mide la integral de su multiplicación puntual. Tiene profundas conexiones con la transformada de Fourier y la transformada de Laplace y se utiliza mucho en el procesamiento de señales. Las capas convolucionales en realidad utilizan correlaciones cruzadas, que son muy similares a las convoluciones (consulte <https://homl.info/76> para más detalles).
  - 7** Para producir salidas del mismo tamaño, una capa completamente conectada necesitaría  $200 \times 150 \times 100$  neuronas, cada una conectada a las  $150 \times 100 \times 3$  entradas. ¡Tendría  $200 \times 150 \times 100 \times (150 \times 100 \times 3 + 1) \approx 135$  mil millones de parámetros!
  - 8** En el sistema internacional de unidades (SI),  $1 \text{ MB} = 1.000 \text{ KB} = 1.000 \times 1.000 \text{ bytes} = 1.000 \times 1.000 \times 8 \text{ bits}$ . Y  $1 \text{ MiB} = 1024 \text{ kB} = 1024 \times 1024 \text{ bytes}$ . Entonces  $12 \text{ MB} \approx 11,44 \text{ MiB}$ .
  - 9** Otros núcleos que hemos analizado hasta ahora tenían pesos, pero los núcleos de agrupación no: Son sólo ventanas correderas sin estado.
  - 10** Yann LeCun et al., "Aprendizaje basado en gradientes aplicado al reconocimiento de documentos", Actas del IEEE 86, no. 11 (1998): 2278–2324.

- <sup>11</sup> Alex Krizhevsky et al., "Clasificación de ImageNet con tecnología neuronal convolucional profunda Networks", Actas de la 25<sup>a</sup> Conferencia Internacional sobre Sistemas de Procesamiento de Información Neural 1 (2012): 1097–1105.
- <sup>12</sup> Matthew D. Zeiler y Rob Fergus, "Visualizing and Understanding Convolutional Networks", Actas de la Conferencia europea sobre visión por computadora (2014): 818–833.
- <sup>13</sup> Christian Szegedy et al., "Going Deeper with Convolutions", Actas de la Conferencia IEEE sobre visión por computadora y reconocimiento de patrones (2015): 1–9.
- <sup>14</sup> En la película Inception de 2010, los personajes siguen profundizando cada vez más en múltiples capas de sueños; de ahí el nombre de estos módulos.
- <sup>15</sup> Karen Simonyan y Andrew Zisserman, "Redes convolucionales muy profundas para el reconocimiento de imágenes a gran escala", preimpresión de arXiv arXiv:1409.1556 (2014).
- <sup>16</sup> Kaiming He et al., "Aprendizaje residual profundo para el reconocimiento de imágenes", preimpresión de arXiv arXiv:1512:03385 (2015).
- <sup>17</sup> Es una práctica común al describir una red neuronal contar solo capas con parámetros.
- <sup>18</sup> Christian Szegedy et al., "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning", preimpresión de arXiv arXiv:1602.07261 (2016).
- <sup>19</sup> François Chollet, "Xception: aprendizaje profundo con funciones separables en profundidad Convoluciones", preimpresión de arXiv arXiv:1610.02357 (2016).
- <sup>20</sup> Este nombre puede ser a veces ambiguo, ya que las convoluciones espacialmente separables son A menudo también se les llama "convoluciones separables".
- <sup>21</sup> Jie Hu et al., "Squeeze-and-Excitation Networks", Actas de la Conferencia IEEE sobre visión por computadora y reconocimiento de patrones (2018): 7132–7141.
- <sup>22</sup> Saining Xie et al., "Transformaciones residuales agregadas para la estimulación neuronal profunda Redes", preimpresión de arXiv arXiv:1611.05431 (2016).
- <sup>23</sup> Gao Huang et al., "Densely Connected Convolutional Networks", preimpresión de arXiv arXiv:1608.06993 (2016).
- <sup>24</sup> Andrew G. Howard et al., "MobileNets: redes neuronales convolucionales eficientes para Aplicaciones de visión móvil", preimpresión de arXiv arxiv:1704.04861 (2017).
- <sup>25</sup> Chien-Yao Wang et al., "CSPNet: A New Backbone That Can Enhance Learning Capability of CNN", preimpresión de arXiv arXiv:1911.11929 (2019).
- <sup>26</sup> Mingxing Tan y Quoc V. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks", preimpresión de arXiv arXiv:1905.11946 (2019).
- <sup>27</sup> Una CPU AMD EPYC de 92 núcleos con IBPB, 1,7 TB de RAM y una Nvidia Tesla A100 GPU.
- <sup>28</sup> En el conjunto de datos ImageNet, cada imagen se asigna a una palabra en el conjunto de datos WordNet: el ID de clase es solo un ID de WordNet.

- <sup>29</sup> Adriana Kovashka et al., "Crowdsourcing en visión por computadora", Fundamentos y tendencias en gráficos y visión por computadora 10, no. 3 (2014): 177–243.
- <sup>30</sup> Jonathan Long et al., "Fully Convolutional Networks for Semantic Segmentation", Actas de la Conferencia IEEE sobre visión por computadora y reconocimiento de patrones (2015): 3431–3440.
- <sup>31</sup> Hay una pequeña excepción: una capa convolucional que use un relleno "válido" quejarse si el tamaño de entrada es menor que el tamaño del kernel.
- <sup>32</sup> Esto supone que utilizamos sólo el "mismo" relleno en la red: relleno "válido" reduciría el tamaño de los mapas de características. Además, 448 se puede dividir claramente entre 2 varias veces hasta llegar a 7, sin ningún error de redondeo. Si alguna capa usa una zancada diferente a la 1 o 2, entonces puede haber algún error de redondeo, por lo que nuevamente los mapas de características pueden terminar siendo más pequeños.
- <sup>33</sup> Joseph Redmon et al., "Solo miras una vez: detección unificada de objetos en tiempo real", Actas de la Conferencia IEEE sobre visión por computadora y reconocimiento de patrones (2016): 779–788.
- <sup>34</sup> Puede encontrar YOLOv3, YOLOv4 y sus pequeñas variantes en los modelos de TensorFlow. proyecto en <https://homl.info/yolotf>.
- <sup>35</sup> Wei Liu et al., "SSD: Single Shot Multibox Detector", Actas del 14º Conferencia europea sobre visión por computadora 1 (2016): 21–37.
- <sup>36</sup> Shaoqing Ren et al., "Faster R-CNN: Hacia la detección de objetos en tiempo real con redes de propuestas regionales", Actas de la 28.ª Conferencia internacional sobre sistemas de procesamiento de información neuronal 1 (2015): 91–99.
- <sup>37</sup> Mingxing Tan et al., "EfficientDet: Scalable and Efficient Object Inspection", preimpresión de arXiv arXiv:1911.09070 (2019).
- <sup>38</sup> Nicolai Wojke et al., "Seguimiento simple en línea y en tiempo real con una profunda asociación Métrica", preimpresión de arXiv arXiv:1703.07402 (2017).
- <sup>39</sup> Este tipo de capa a veces se denomina capa de deconvolución, pero no realiza lo que los matemáticos llaman deconvolución, por lo que se debe evitar este nombre.
- <sup>40</sup> Kaiming He et al., "Mask R-CNN", preimpresión de arXiv arXiv:1703.06870 (2017).

# Capítulo 15. Procesamiento de secuencias utilizando RNN y CNN

---

Predecir el futuro es algo que haces todo el tiempo, ya sea que estés terminando la frase de un amigo o anticipando el olor del café en el desayuno. En este capítulo analizaremos las redes neuronales recurrentes (RNN), una clase de redes que pueden predecir el futuro (bueno, hasta cierto punto). Los RNN pueden analizar datos de series temporales, como la cantidad de usuarios activos diarios en su sitio web, la temperatura horaria en su ciudad, el consumo diario de energía de su hogar, las trayectorias de los automóviles cercanos y más. Una vez que un RNN aprende patrones pasados en los datos, puede usar su conocimiento para pronosticar el futuro, asumiendo, por supuesto, que los patrones pasados aún se mantienen en el futuro.

De manera más general, los RNN pueden funcionar en secuencias de longitudes arbitrarias, en lugar de entradas de tamaño fijo. Por ejemplo, pueden tomar oraciones, documentos o muestras de audio como entrada, lo que los hace extremadamente útiles para aplicaciones de procesamiento del lenguaje natural, como la traducción automática o la conversión de voz a texto.

En este capítulo, primero analizaremos los conceptos fundamentales que subyacen a los RNN y cómo entrenarlos mediante la propagación hacia atrás en el tiempo. Luego, los usaremos para pronosticar una serie de tiempo. En el camino, veremos la popular familia de modelos ARMA, que a menudo se usa para pronosticar series de tiempo, y las usaremos como puntos de referencia para comparar con nuestros RNN. Después de eso, exploraremos las dos dificultades principales que enfrentan los RNN:

- gradientes inestables (discutidos en [el capítulo 11](#)), que pueden aliviarse mediante diversas técnicas, incluida la deserción recurrente

y normalización de capas recurrentes.

- Una memoria a corto plazo (muy) limitada, que se puede ampliar utilizando celdas LSTM y GRU.

Las RNN no son los únicos tipos de redes neuronales capaces de manejar datos secuenciales. Para secuencias pequeñas, una red densa normal puede ser la solución, y para secuencias muy largas, como muestras de audio o texto, las redes neuronales convolucionales también pueden funcionar bastante bien. Analizaremos ambas posibilidades y finalizaremos este capítulo implementando una WaveNet, una arquitectura CNN capaz de manejar secuencias de decenas de miles de pasos de tiempo. ¡Empecemos!

## Neuronas y capas recurrentes

Hasta ahora nos hemos centrado en redes neuronales feedforward, donde las activaciones fluyen en una sola dirección, desde la capa de entrada a la capa de salida. Una red neuronal recurrente se parece mucho a una red neuronal de avance, excepto que también tiene conexiones que apuntan hacia atrás.

Veamos el RNN más simple posible, compuesto por una neurona que recibe entradas, produce una salida y envía esa salida a sí misma, como se muestra en la [Figura 15-1](#) (izquierda). En cada paso de tiempo  $t$  (también llamado cuadro), esta neurona recurrente recibe las entradas  $x$  así como su propia salida del paso de tiempo anterior,  $\hat{y}$ . Dado que no hay una salida previa 1) en el primer paso de tiempo, generalmente se establece en 0. Podemos representar esta pequeña red contra el eje de tiempo, como se muestra en la [Figura 15-1](#) (derecha). A esto se le llama desenrollar la red a través del tiempo (es la misma neurona recurrente representada una vez por paso de tiempo)

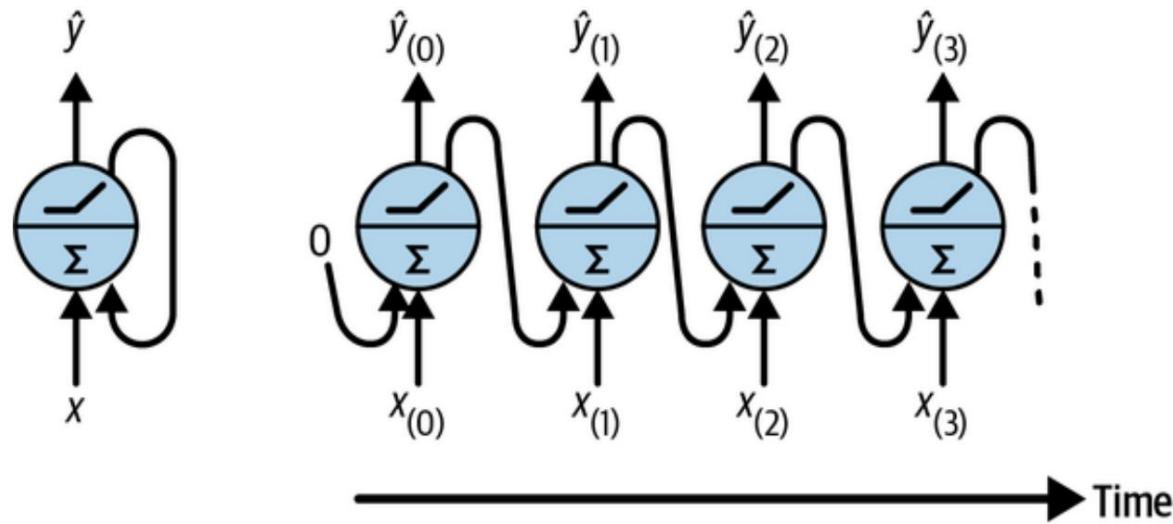


Figura 15-1. Una neurona recurrente (izquierda) desenrollada en el tiempo (derecha)

Puedes crear fácilmente una capa de neuronas recurrentes. En cada paso de tiempo  $t$ , cada neurona recibe tanto el vector de entrada  $x$  como el vector de salida ( $t$ ) del paso de tiempo anterior  $\hat{y}_{(t-1)}$  que se muestra en la Figura. Tenga en cuenta que tanto las entradas como las salidas ahora son vectores (cuando solo había una neurona, la salida era un escalar).

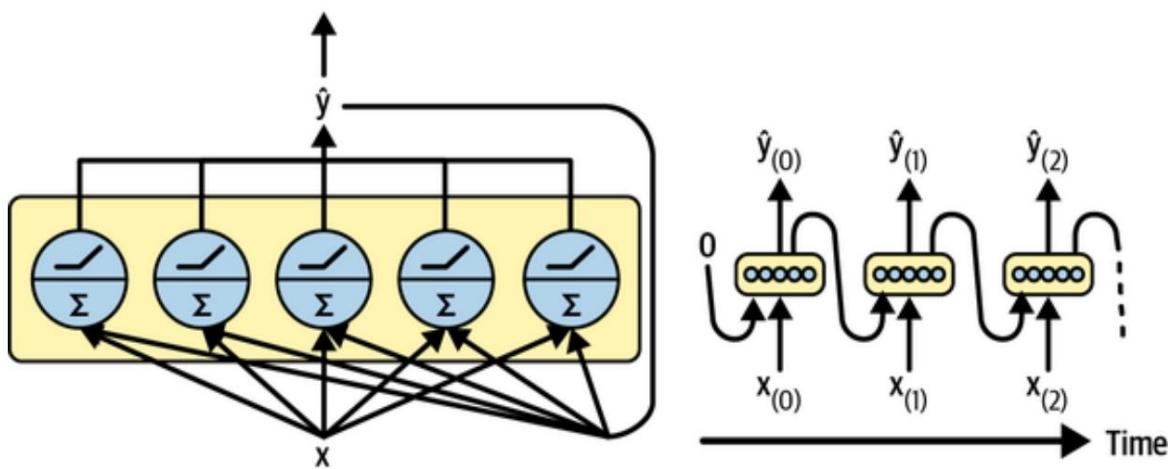


Figura 15-2. Una capa de neuronas recurrentes (izquierda) desenrollada a través del tiempo (derecha)

Cada neurona recurrente tiene dos conjuntos de pesos: uno para las entradas  $x$  ( $t$ ) y el otro para las salidas del paso de tiempo anterior,  $\hat{y}_{(t-1)}$ . Llamaremos  $w_x$  y  $w_y$  estos vectores de peso. Si consideramos toda la capa recurrente  $\hat{y}$  en lugar de solo una neurona recurrente, podemos colocar todos los vectores de peso en dos matrices de peso:  $W_x$  y  $W_y$ .

Luego se puede calcular el vector de salida de toda la capa recurrente.

más o menos como se podría esperar, como se muestra en [la Ecuación 15-1](#), donde <sup>1</sup>  
b es el vector de polarización y  $\cdot(\cdot)$  es la función de activación (por ejemplo, ReLU).

Ecuación 15-1. Salida de una capa recurrente para una sola instancia

$$\hat{y}(t) = (Wx \quad x(t) + W\hat{y} \quad \hat{y}(t-1) + b)$$

Al igual que con las redes neuronales feedforward, podemos calcular una salida de la capa recurrente de una sola vez para un mini lote completo colocar todas las entradas en el paso de tiempo t en una matriz de entrada X ([ver Ecuación 15-2](#)).

Ecuación 15-2. Salidas de una capa de neuronas recurrentes para todas las instancias de un pase:  
[mini-lote]

$$\begin{aligned}\hat{Y}(t) &= (X(t)Wx + \hat{Y}(t-1)W\hat{y} + b) \\ &= ([X(t) \hat{Y}(t-1)]W + b) \text{ con } W = \begin{bmatrix} Wx \\ W\hat{y} \end{bmatrix}\end{aligned}$$

En esta ecuación:

- $\hat{Y}$  es un  $m \times n$  (t) neuronas matriz que contiene las salidas de la capa en paso de tiempo t para cada instancia en el mini-lote (m es el número de instancias en el mini-lote y n neuronas es el numero de neuronas).
- $X$  es una matriz  $m \times n$  que contiene las entradas para todas las instancias (n entradas es el número de características de entrada).
- $W$  es una matriz  $n \times n$  que contiene la conexión ingresa pesos para las entradas del paso de tiempo actual.
- $W$  es un  $n \times n$  neuronas  $\times n$  neuronas matriz que contiene la conexión pesos para las salidas del paso de tiempo anterior.
- $b$  es un vector de tamaño  $n$  neuronas que contiene el término de polarización de cada neurona.

- Las matrices de peso  $W$  y  $W$  a menudo se concatenan verticalmente en una única matriz de peso  $W$  de forma  $(n \text{ entradas} + n \text{ neuronas}) \times n \text{ neuronas}$  (ver la segunda línea de la Ecuación 15-2).
- La notación  $[X \hat{Y}]$  representa la concatenación horizontal de las matrices  $X$  y  $\hat{Y}$ .

Observe que  $\hat{Y}$  es una función de  $X$  y  $\hat{Y}(t) (t-1)$ , que es una función de  $X$  y  $\hat{Y}$  que es función de  $X$  y  $\hat{Y}(t-1) (t-2) (t-3)$ , etcétera.

Esto hace que  $\hat{Y}$  sea una función de todas las entradas desde el tiempo  $t = 0$  (es decir,  $X(0), X(1), \dots, X(t)$ ). En el primer paso de tiempo,  $t = 0$ , no hay cambios previos. salidas, por lo que normalmente se supone que son todas ceros.

## Células de memoria

Dado que la salida de una neurona recurrente en el paso de tiempo  $t$  es función de todas las entradas de pasos de tiempo anteriores, se podría decir que tiene una forma de memoria. Una parte de una red neuronal que conserva algún estado.

a través de pasos de tiempo se llama celda de memoria (o simplemente celda). un solo La neurona recurrente, o una capa de neuronas recurrentes, es una célula muy básica, capaz de aprender sólo patrones cortos (normalmente de unos 10 pasos de largo, pero esto varía dependiendo de la tarea). Más adelante en este capítulo, veremos observe algunos tipos de células más complejas y poderosas capaces de aprender patrones más largos (aproximadamente 10 veces más, pero nuevamente, esto depende de la tarea).

El estado de una celda en el paso de tiempo  $t$ , denominado  $h(t)$  (la “ $h$ ” significa “oculto”), es una función de algunas entradas en ese paso de tiempo y su estado en el paso de tiempo anterior:  $h = f(x(t-1), h(t-1))$ . Su salida en el paso de tiempo  $t$ , denotado  $\hat{y}(t)$ , también es función del estado anterior y del actual entradas. En el caso de las células básicas que hemos comentado hasta ahora, la La salida es igual al estado, pero en celdas más complejas esto no es así. Siempre es así, como se muestra en la Figura 15-3.

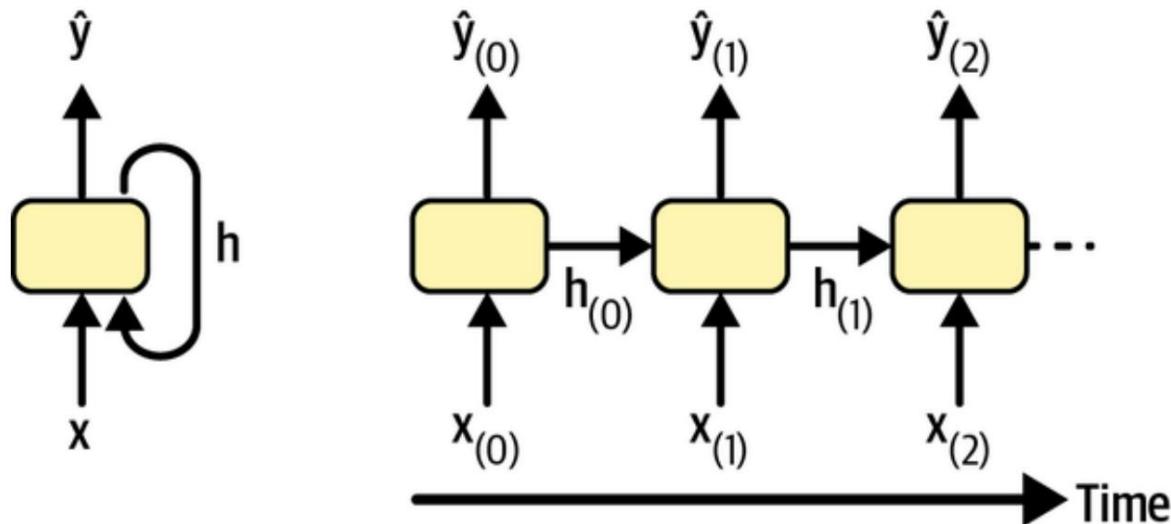


Figura 15-3. El estado oculto de una celda y su salida pueden ser diferentes

## Secuencias de entrada y salida

Un RNN puede tomar simultáneamente una secuencia de entradas y producir una secuencia de salidas (consulte la red superior izquierda en la [Figura 15-4](#)). Este tipo de red de secuencia a secuencia es útil para pronosticar series temporales, como el consumo de energía diario de su hogar: le proporciona los datos de los últimos N días y lo entrena para generar el consumo de energía desplazado un día en el futuro (es decir, desde hace N – 1 días hasta mañana).

Alternativamente, puede alimentar a la red con una secuencia de entradas e ignorar todas las salidas excepto la última (consulte la red superior derecha en la [Figura 15-4](#)). Esta es una red de secuencia a vector. Por ejemplo, podría alimentar a la red con una secuencia de palabras correspondientes a la reseña de una película, y la red generaría una puntuación de sentimiento (por ejemplo, de 0 [odio] a 1 [amor]).

Por el contrario, podría alimentar a la red con el mismo vector de entrada una y otra vez en cada paso de tiempo y dejar que genere una secuencia (consulte la red inferior izquierda de la [Figura 15-4](#)). Esta es una red de vector a secuencia. Por ejemplo, la entrada podría ser una imagen (o la salida de una CNN) y la salida podría ser un título para esa imagen.

Por último, podría tener una red de secuencia a vector, llamada codificador, seguida de una red de vector a secuencia, llamada decodificador (consulte la red inferior derecha de la [Figura 15-4](#)).

Por ejemplo, esto podría usarse para traducir una oración de un idioma a otro. Alimentaría a la red con una oración en un idioma, el codificador convertiría esta oración en una representación

vectorial única y luego el decodificador decodificaría este vector en una oración en otro idioma. Este modelo de dos pasos, llamado

**codificador-decodificador**, funciona mucho mejor que intentar traducir sobre la marcha con un único RNN de secuencia a secuencia

(como el representado en la parte superior izquierda): las últimas palabras de una oración pueden afectar las primeras palabras de la traducción, por lo que debes esperar hasta que hayas visto la frase completa antes de traducirla. Analizaremos la implementación de un codificador-decodificador en [el Capítulo 16](#) (como verá, es un poco más complejo de lo

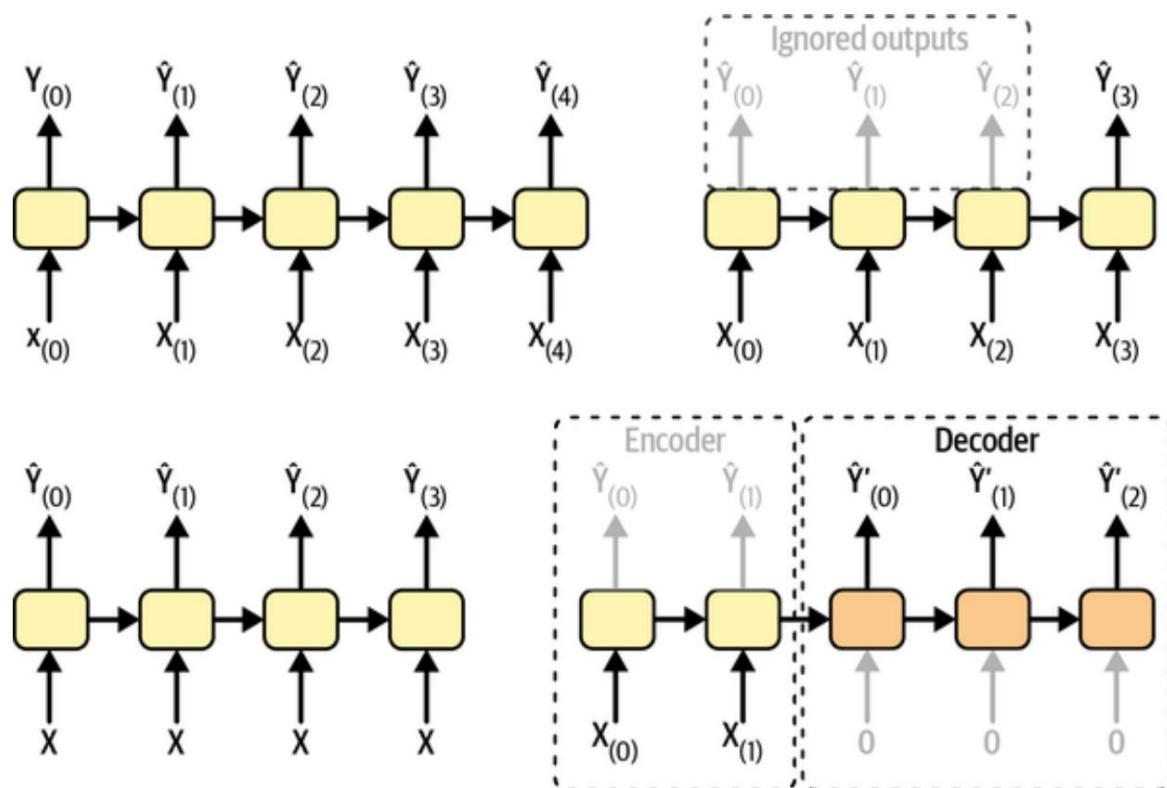


Figura 15-4. Redes de secuencia a secuencia (arriba a la izquierda), secuencia a vector (arriba a la derecha), vector a secuencia (abajo a la izquierda) y codificador-decodificador (abajo a la derecha)

Esta versatilidad suena prometedora, pero ¿cómo se entrena a un recurrente red neuronal?

## Entrenamiento de RNN

Para entrenar un RNN, el truco consiste en desarrollarlo en el tiempo (como acabamos de hacer) y luego utilice la propagación hacia atrás regular ([consulte la Figura 15-5](#)). Este La estrategia se llama retropropagación a través del tiempo (BPTT).

Al igual que en la propagación hacia atrás normal, hay un primer pase hacia adelante. a través de la red desenrollada (representada por las flechas discontinuas).

Luego, la secuencia de salida se evalúa utilizando una función de pérdida  $(Y_0, Y_1, \dots, Y_T)$  donde  $Y_i$  es el objetivo  $i$ ,  $\hat{Y}_i$  es el  $i$  predicción y  $T$  es el paso de tiempo máximo). Tenga en cuenta que esta función de pérdida puede ignorar algunas salidas. Por ejemplo, en una secuencia a vector RNN, todas las salidas se ignoran excepto la última. En

[Figura 15-5](#), la función de pérdida se calcula en función de los últimos tres Solo salidas. Los gradientes de esa función de pérdida luego se propagan hacia atrás a través de la red desenrollada (representada por el sólido flechas). En este ejemplo, dado que las salidas  $\hat{Y}_1$  y  $\hat{Y}_2$  no se utilizan para calcular la pérdida, los gradientes no fluyen hacia atrás a través de ellos; sólo fluyen a través de  $\hat{Y}_0$ . Es más, desde el Se utilizan los mismos parámetros  $W$  y  $b$  en cada paso de tiempo, sus Los gradientes se modificarán varias veces durante la retroprop. Una vez el La fase de retroceso está completa y todos los gradientes han sido calculado, BPTT puede realizar un paso de descenso de gradiente para actualizar el parámetros (esto no es diferente del backprop normal).

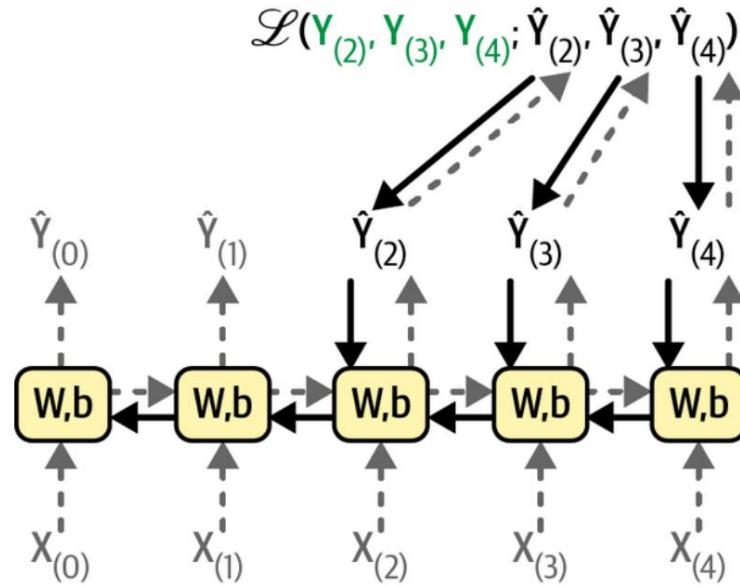


Figura 15-5. Propagación hacia atrás a través del tiempo

Afortunadamente, Keras se encarga de toda esta complejidad por usted, como verá. Pero antes de llegar allí, carguemos una serie de tiempo y comenzemos a analizarla utilizando herramientas clásicas para comprender mejor a qué nos enfrentamos y obtener algunas métricas de referencia.

## Pronosticar una serie temporal

¡Está bien! Supongamos que la Autoridad de Tránsito de Chicago acaba de contratarlo como científico de datos. Su primera tarea es construir un modelo capaz de pronosticar el número de pasajeros que viajarán en autobús y tren al día siguiente. Tiene acceso a datos diarios sobre el número de pasajeros desde 2001. Analicemos juntos cómo manejarías esto. Empezaremos por

3

cargando y limpiando los datos:

```
importar pandas como pd
desde pathlib import ruta
```

```
ruta = Ruta("datasets/ridership/CTA_-_Ridership_-
_Daily_Boarding_Totals.csv") df =
pd.read_csv(ruta, parse_dates=["service_date"]) df.columns = ["fecha",
"tipo_día", "autobús" , "ferrocarril", "total"] # nombres más cortos
```

```

df = df.sort_values("fecha").set_index("fecha")
df = df.drop("total", axis=1) # no es necesario el total, es
solo autobús + tren
df = df.drop_duplicates() # eliminar meses duplicados
(2011-10 y 2014-07)

```

Cargamos el archivo CSV, configuramos nombres cortos de columnas, ordenamos las filas por fecha, elimine la columna total redundante y elimine las filas duplicadas. Ahora veamos cómo se ven las primeras filas:

```

>>> df.cabeza()
      tipo_día    autobús    carril
fecha
2001-01-01      297192 126455
2001-01-02        W 780827 501952
2001-01-03        W 824923 536432
2001-01-04        W 870021 550011
2001-01-05        W 890426 557917

```

El 1 de enero de 2001, 297.192 personas subieron a un autobús en Chicago, y 126.455 subieron a un tren. La columna day\_type contiene W para Días laborables, A para sábados y U para domingos o festivos.

Ahora tracemos las cifras de usuarios de autobuses y trenes durante unos meses en 2019, para ver cómo se ve (ver [Figura 15-6](#)):

```

importar matplotlib.pyplot como plt

df["2019-03":"2019-05"].plot(grid=True, marcador=".",
tamaño de higo=(8, 3.5))
plt.mostrar()

```

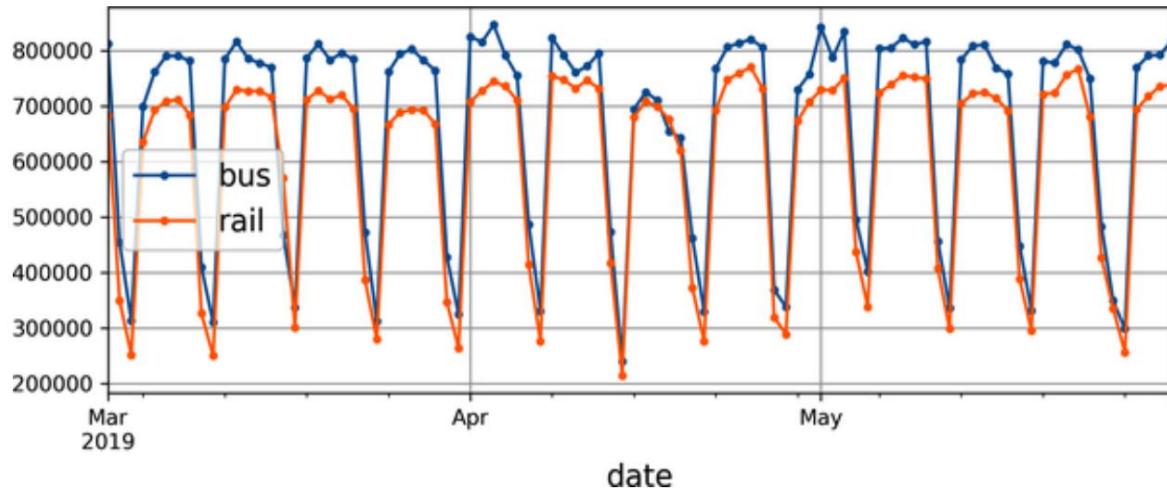


Figura 15-6. Número de pasajeros diarios en Chicago

Tenga en cuenta que Pandas incluye tanto el mes de inicio como el de finalización en el rango, por lo que esto traza los datos desde el 1 de marzo hasta el 31 de mayo. Se trata de una serie temporal: datos con valores en diferentes intervalos de tiempo, normalmente a intervalos regulares. Más específicamente, dado que hay múltiples valores por paso de tiempo, esto se denomina serie de tiempo multivariada. Si solo miráramos la columna del bus, sería una serie temporal univariada, con un único valor por paso de tiempo. Predecir valores futuros (es decir, pronosticar) es la tarea más típica cuando se trata de series temporales, y en esto nos centraremos en este capítulo. Otras tareas incluyen imputación (completar valores anteriores faltantes), clasificación, detección de anomalías y más.

Si observamos [la figura 15-6](#), podemos ver que un patrón similar se repite claramente cada semana. A esto se le llama estacionalidad semanal . De hecho, en este caso es tan fuerte que pronosticar el número de pasajeros de mañana simplemente copiando los valores de una semana antes arrojará resultados razonablemente buenos. A esto se le llama pronóstico ingenuo: simplemente copiar un valor pasado para hacer nuestro pronóstico. Los pronósticos ingenuos suelen ser una excelente base y, en algunos casos, incluso pueden ser difíciles de superar.

## NOTA

En general, hacer pronósticos ingenuos significa copiar el último valor conocido (por ejemplo, pronosticar que mañana será igual que hoy). Sin embargo, en nuestro caso, copiar el valor de la semana anterior funciona mejor debido a la fuerte estacionalidad semanal.

Para visualizar estos pronósticos ingenuos, superpongamos las dos series de tiempo (para autobús y tren), así como la misma serie de tiempo retrasada una semana (es decir, desplazada hacia la derecha) usando líneas de puntos. También trazaremos la diferencia entre los dos (es decir, el valor en el momento  $t$  menos el valor en el momento  $t - 7$ ); esto se llama diferenciación (ver [Figura 15-7](#)):

```
diff_7 = df[["autobús", "ferrocarril"]].diff(7)["2019-03":"2019-05"]

fig, axs = plt.subplots(2, 1, sharex=True, figsize=(8, 5)) df.plot(ax=axs[0], legend=False,
Marker=".") # serie temporal original

df.shift(7).plot(ax=axs[0], grid=True, legend=False, linestyle="--") # retrasado
diff_7.plot(ax=axs[1], grid=True,
Marker=" .") # Serie temporal de diferencia de 7 días

plt.mostrar()
```

¡No está mal! Observe cuán estrechamente las series de tiempo retrasadas siguen la serie de tiempo real. Cuando una serie de tiempo se correlaciona con una versión retrasada de sí misma, decimos que la serie de tiempo está autocorrelacionada. Como puede ver, la mayoría de las diferencias son bastante pequeñas, excepto a finales de mayo. ¿Quizás había un día festivo en ese momento? Comprobemos la columna `day_type`:

```
>>> lista(df.loc["2019-05-25":"2019-05-27"]["tipo_día"])
['A', 'U', 'U']
```

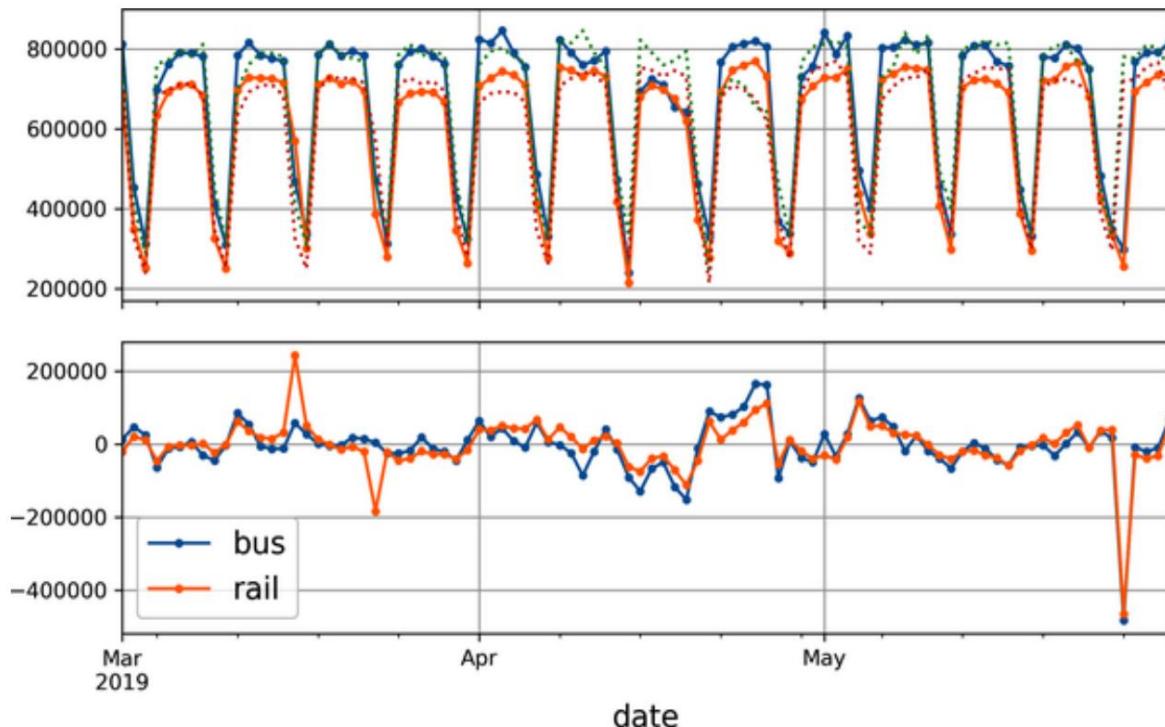


Figura 15-7. Serie temporal superpuesta con serie temporal retrasada de 7 días (arriba) y diferencia entre  $t$  y  $t - 7$  (abajo)

De hecho, en aquel entonces había un fin de semana largo: el lunes era el feriado del Día de los Caídos. Podríamos usar esta columna para mejorar nuestros pronósticos, pero por ahora solo midimos el error absoluto medio durante el período de tres meses en el que nos centramos arbitrariamente (marzo, abril y mayo de 2019) para tener una idea aproximada:

```
>>> diff_7.abs().mean() bus
43915.608696
carril      42143.271739
tipo de letra: float64
```

Nuestros ingenuos pronósticos arrojan un MAE de aproximadamente 43.916 pasajeros de autobús y alrededor de 42.143 pasajeros de tren. Es difícil decir de un vistazo qué tan bueno o malo es esto, así que pongamos los errores de pronóstico en perspectiva dividiéndolos por los valores objetivo:

```
>>> objetivos = df[["bus", "rail"]]["2019-03":"2019-05"] >>> (diff_7 /
objetivos.abs().mean() bus 0.082938
```

```
carril      0.089948
tipo de letra: float64
```

Lo que acabamos de calcular se llama error porcentual absoluto medio (MAPE): parece que nuestros ingenuos pronósticos nos dan un MAPE de aproximadamente el 8,3% para el autobús y el 9,0% para el ferrocarril. Es interesante observar que el MAE para las previsiones ferroviarias parece ligeramente mejor que el MAE para las previsiones de autobuses, mientras que ocurre lo contrario para el MAPE. Esto se debe a que el número de pasajeros en autobús es mayor que el de ferrocarril, por lo que, naturalmente, los errores de pronóstico también son mayores, pero cuando ponemos los errores en perspectiva, resulta que los pronósticos de autobús son en realidad ligeramente mejores que los pronósticos de ferrocarril.

#### CONSEJO

MAE, MAPE y MSE se encuentran entre las métricas más comunes que puede utilizar para evaluar sus pronósticos. Como siempre, elegir la métrica correcta depende de la tarea. Por ejemplo, si su proyecto sufre cuadráticamente más errores grandes que pequeños, entonces el MSE puede ser preferible, ya que penaliza fuertemente los errores grandes.

Si observamos las series temporales, no parece haber ninguna estacionalidad mensual significativa, pero verifiquemos si hay alguna estacionalidad anual. Analizaremos los datos de 2001 a 2019. Para reducir el riesgo de espionaje de datos, ignoraremos los datos más recientes por ahora. Tracemos también un promedio móvil de 12 meses para cada serie para visualizar tendencias a largo plazo (consulte la [Figura 15-8](#)):

```
period = slice("2001", "2019") df_monthly =
df.resample('M').mean() # calcula la media de cada mes

Rolling_average_12_months =
df_monthly[period].rolling(ventana=12).mean()

fig, ax = plt.subplots(figsize=(8, 4))
df_monthly[period].plot(ax=ax, marcador=".")
```

```
Rolling_average_12_months.plot(ax=ax, grid=True,
                                 leyenda=False)
plt.show()
```

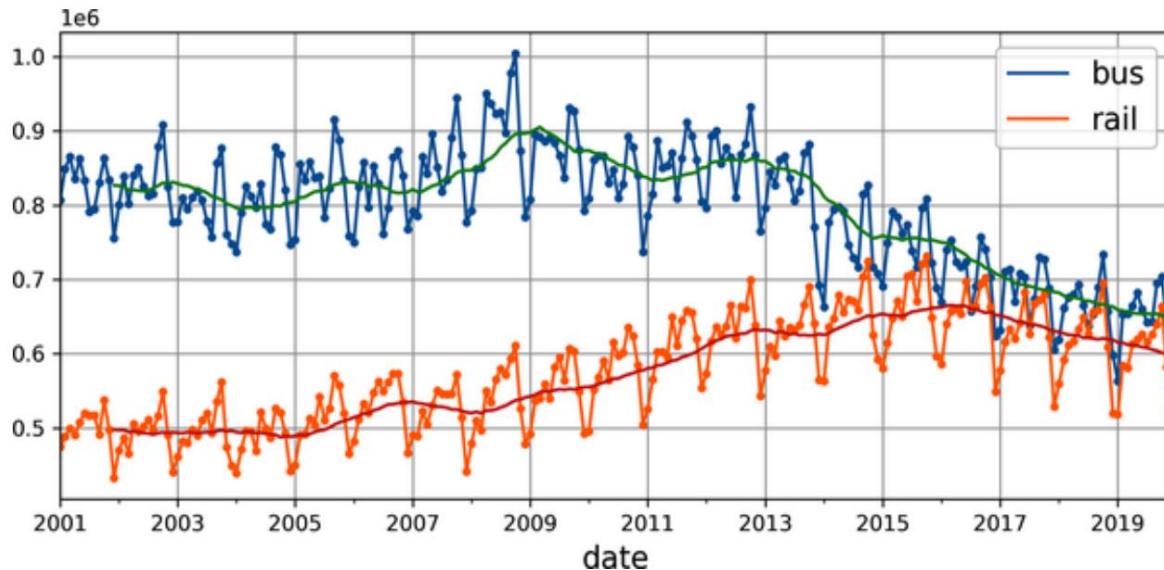


Figura 15-8. Estacionalidad anual y tendencias a largo plazo.

¡Sí! Definitivamente también hay cierta estacionalidad anual, aunque es más ruidosa que la estacionalidad semanal y más visible para la serie ferroviaria que para la serie de autobuses: vemos picos y valles aproximadamente en las mismas fechas cada año. Comprobemos lo que obtenemos si trazamos la diferencia de 12 meses (consulte [la Figura 15-9](#)):

```
df_monthly.diff(12)[periodo].plot(grid=True, marcador=".", figsize=(8, 3))
plt.show()
```

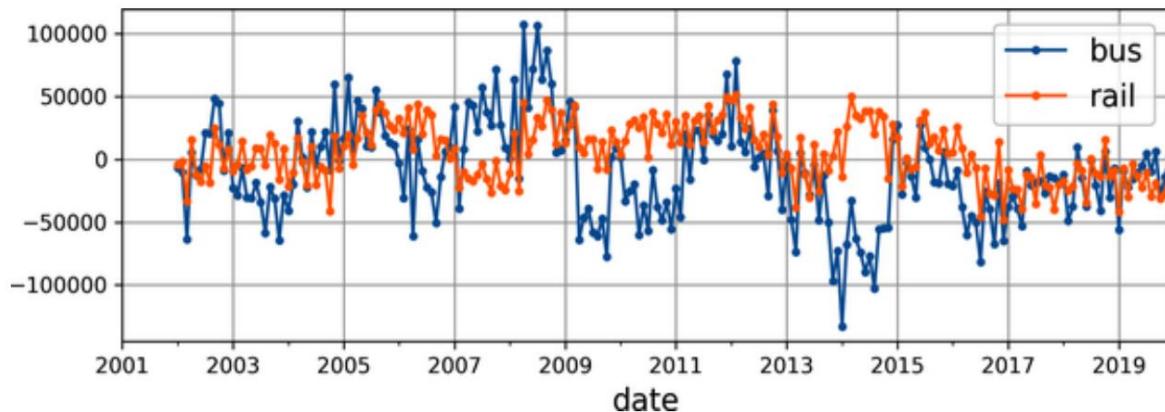


Figura 15-9. La diferencia de 12 meses

Observe cómo la diferenciación no sólo eliminó la estacionalidad anual, sino que también eliminó las tendencias a largo plazo. Por ejemplo, la tendencia lineal a la baja presente en la serie temporal de 2016 a 2019 se convirtió en un valor negativo aproximadamente constante en la serie temporal diferenciada. De hecho, la diferenciación es una técnica común utilizada para eliminar la tendencia y la estacionalidad de una serie temporal: es más fácil estudiar una serie temporal estacionaria , es decir, una cuyas propiedades estadísticas permanecen constantes a lo largo del tiempo, sin estacionalidad ni tendencias. Una vez que pueda hacer pronósticos precisos sobre las series de tiempo diferenciadas, es fácil convertirlos en pronósticos para la serie de tiempo real simplemente sumando los valores anteriores que se restaron previamente.

Quizás esté pensando que sólo estamos tratando de predecir el número de pasajeros del mañana, por lo que los patrones a largo plazo importan mucho menos que los de corto plazo. Tienes razón, pero aun así es posible que podamos mejorar ligeramente el rendimiento si tenemos en cuenta los patrones a largo plazo. Por ejemplo, el número de pasajeros diarios en autobús se redujo en aproximadamente 2500 en octubre de 2017, lo que representa aproximadamente 570 pasajeros menos cada semana, por lo que si estuviéramos a finales de octubre de 2017, tendría sentido pronosticar el número de pasajeros de mañana copiando el valor de la semana pasada. menos 570. Tener en cuenta la tendencia hará que sus pronósticos sean un poco más precisos en promedio.

Ahora que está familiarizado con las series temporales de usuarios, así como con algunos de los conceptos más importantes en el análisis de series temporales,

incluyendo estacionalidad, tendencia, diferenciación y promedios móviles, echemos un vistazo rápido a una familia muy popular de modelos estadísticos que se usan comúnmente para analizar series de tiempo.

## La familia de modelos ARMA

Comenzaremos con el modelo de promedio móvil autorregresivo (ARMA), desarrollado por Herman Wold en la década de 1930: calcula sus pronósticos utilizando una simple suma ponderada de valores rezagados y corrige estos pronósticos agregando un promedio móvil, muy parecido a lo que acabamos de comentar. . Específicamente, el componente de media móvil se calcula utilizando una suma ponderada de los últimos errores de pronóstico. [La ecuación 15-3](#) muestra cómo el modelo hace sus pronósticos.

Ecuación 15-3. Previsión utilizando un modelo ARMA

$$\hat{y}(t) = \sum_{i=1}^p \alpha_i y(t-i) + \sum_{i=1}^q \theta_i \epsilon(t-i)$$

$$(t) = y(t) - \hat{y}(t)$$

En esta ecuación:

- $\hat{y}(t)$  es el pronóstico del modelo para el paso de tiempo  $t$ .
- $y(t)$  es el valor de la serie temporal en el paso de tiempo  $t$ .
- La primera suma es la suma ponderada de los valores  $p$  pasados de la serie temporal, utilizando los pesos aprendidos  $\alpha$ . El número  $p$  es un hiperparámetro y determina qué tan atrás en el pasado debe verse el modelo. Esta suma es el componente autorregresivo del modelo: realiza una regresión basada en valores pasados.
- La segunda suma es la suma ponderada del pronóstico  $q$  pasado , utilizando las ponderaciones aprendidas  $\theta$  . El número  $q$  es un error  $\epsilon(t)$  hiperparámetro. Esta suma es el componente de media móvil del modelo.

Es importante destacar que este modelo supone que la serie temporal es estacionaria. Si no es así, entonces la diferenciación puede ayudar. El uso de la diferenciación en un solo paso de tiempo producirá una aproximación de la derivada de la serie de tiempo: de hecho, dará la pendiente de la serie en cada paso de tiempo. Esto significa que eliminará cualquier tendencia lineal, transformándola en un valor constante. Por ejemplo, si aplica la diferenciación de un paso a la serie [3, 5, 7, 9, 11], obtiene la serie diferenciada [2, 2, 2, 2].

Si la serie temporal original tiene una tendencia cuadrática en lugar de una tendencia lineal, entonces una única ronda de diferenciación no será suficiente. Por ejemplo, la serie [1, 4, 9, 16, 25, 36] se convierte en [3, 5, 7, 9, 11] después de una ronda de diferenciación, pero si ejecutas la diferenciación para una segunda ronda, obtienes [2, 2, 2, 2]. Por lo tanto, ejecutar dos rondas de diferenciación eliminará las tendencias cuadráticas. De manera más general, ejecutar  $d$  rondas consecutivas de diferenciación calcula una aproximación de<sup>th</sup> la derivada de orden  $d$  de la serie temporal, por lo que eliminará las tendencias polinómicas hasta el grado  $d$ . Este hiperparámetro  $d$  se llama orden de integración.

La diferenciación es la contribución central del modelo de media móvil integrada autorregresiva (ARIMA), introducido en 1970 por George Box y Gwilym Jenkins en su libro Time Series Analysis (Wiley): este modelo ejecuta  $d$  rondas de diferenciación para hacer que la serie temporal sea más estacionaria. Luego aplica un modelo ARMA normal. Al realizar pronósticos, utiliza este modelo ARMA y luego suma los términos que se restaron mediante la diferenciación.

Un último miembro de la familia ARMA es el modelo estacional ARIMA (SARIMA): modela la serie temporal de la misma manera que ARIMA, pero además modela un componente estacional para una frecuencia determinada (por ejemplo, semanal), utilizando exactamente el mismo ARIMA. acercarse. Tiene un total de siete hiperparámetros: los mismos hiperparámetros  $p$ ,  $d$  y  $q$  que ARIMA, más hiperparámetros  $P$ ,  $D$  y  $Q$  adicionales para modelar el patrón estacional y, por último, el período del patrón estacional, señaló  $s$ . Los hiperparámetros  $P$ ,  $D$  y  $Q$

son como p, d y q, pero se usan para modelar la serie de tiempo en t – s, t – 2s, t – 3s, etc.

Veamos cómo ajustar un modelo SARIMA a la serie temporal ferroviaria y usarlo para hacer un pronóstico del número de pasajeros de mañana. Supondremos que hoy es el último día de mayo de 2019 y queremos pronosticar el número de pasajeros en tren para "mañana", el 1 de junio de 2019. Para ello, podemos utilizar la biblioteca statsmodels, que contiene muchos modelos estadísticos diferentes. incluyendo el modelo ARMA y sus variantes, implementado por la clase ARIMA:

```
desde statsmodels.tsa.arima.model importar ARIMA
```

```
origen, hoy = "2019-01-01", "2019-05-31" rail_series =
df.loc[origin:today][["rail"]].asfreq("D") modelo = ARIMA(rail_series, orden=( 1, 0,
0), orden_estacional=(0, 1, 1, 7))
            modelo = model.fit()
            y_pred = model.forecast() # devuelve
427.758,6
```

En este ejemplo de código:

- Comenzamos importando la clase ARIMA, luego tomamos los datos de uso de trenes desde principios de 2019 hasta "hoy" y usamos asfreq("D") para establecer la frecuencia de la serie temporal en diaria: esto no cambia los datos en este caso, ya que ya son diarios, pero sin esto la clase ARIMA tendría que adivinar la frecuencia y mostraría una advertencia.
- A continuación, creamos una instancia ARIMA, le pasamos todos los datos hasta "hoy" y configuramos los hiperparámetros del modelo: orden=(1, 0, 0) significa que p = 1, d = 0, q = 0 y orden\_estacional = (0, 1, 1, 7) significa que P = 0, D = 1, Q = 1 y s = 7.  
Tenga en cuenta que la API de statsmodels difiere un poco de la API de Scikit-Learn, ya que pasamos los datos al modelo en el momento de la construcción, en lugar de pasarlos al método fit().

- A continuación, ajustamos el modelo y lo utilizamos para hacer un pronóstico para “mañana”, el 1 de junio de 2019.

La previsión es de 427.759 pasajeros, cuando en realidad fueron 379.044.

Vaya, tenemos un 12,9% de descuento, eso es bastante malo. En realidad, es ligeramente peor que el pronóstico ingenuo, que pronostica 426.932, un 12,6% menos. ¿Pero tal vez simplemente tuvimos mala suerte ese día? Para comprobar esto, podemos ejecutar el mismo código en un bucle para hacer pronósticos para todos los días de marzo, abril y mayo, y calcular el MAE durante ese período:

```
origen, fecha_inicio, fecha_final = "2019-01-01", "2019-03-01", "2019-05-31"

time_period = pd.date_range(start_date, end_date) rail_series =
df.loc[origin:end_date]["rail"].asfreq("D") y_preds = [] para hoy en time_period.shift(-1):
modelo = ARIMA
(rail_series[origen:hoy], # entrenar con datos hasta
el pedido "hoy"=(1, 0, 0), estacional_order=(0, 1, 1, 7))

model = model.fit() # tenga en cuenta que volvemos a entrenar el modelo
¡cada día!
y_pred = model.forecast()[0]
y_preds.append(y_pred)

y_preds = pd.Series(y_preds, index=time_period) mae = (y_preds -
rail_series[time_period]).abs().mean() # devuelve 32.040,7
```

¡Ah, eso es mucho mejor! El MAE es de aproximadamente 32.041, que es significativamente menor que el MAE que obtuvimos con un pronóstico ingenuo (42.143). Entonces, aunque el modelo no es perfecto, en promedio supera a los pronósticos ingenuos por un amplio margen.

En este punto, quizás se pregunte cómo elegir buenos hiperparámetros para el modelo SARIMA. Hay varios métodos, pero el más sencillo de entender y comenzar es el enfoque de fuerza bruta: simplemente ejecute una búsqueda en la cuadrícula. Para cada modelo que desee evaluar (es decir, cada combinación de hiperparámetros), puede

Ejecute el ejemplo de código anterior y cambie solo los valores de los hiperparámetros. Los buenos valores de p, q, P y Q suelen ser bastante pequeños (normalmente de 0 a 2, a veces hasta 5 ó 6), y d y D suelen ser 0 ó 1, a veces 2. En cuanto a s, es sólo el factor estacional principal. Período del patrón: en nuestro caso es 7 ya que hay una fuerte estacionalidad semanal. Gana el modelo con el MAE más bajo. Por supuesto, puedes reemplazar el MAE con otra métrica si se adapta mejor a tu objetivo comercial. ¡Y eso es todo !

## Preparación de los datos para modelos de aprendizaje automático

Ahora que tenemos dos líneas de base, pronóstico ingenuo y SARIMA, intentemos usar los modelos de aprendizaje automático que hemos cubierto hasta ahora para pronosticar esta serie de tiempo, comenzando con un modelo lineal básico. Nuestro objetivo será pronosticar el número de pasajeros de mañana en función de los datos de las últimas 8 semanas (56 días). Por lo tanto, las entradas a nuestro modelo serán secuencias (generalmente una secuencia única por día una vez que el modelo esté en producción), cada una conteniendo 56 valores desde los pasos de tiempo t – 55 hasta t. Para cada secuencia de entrada, el modelo generará un único valor: el pronóstico para el paso de tiempo t + 1.

Pero, ¿qué usaremos como datos de entrenamiento? Bueno, ese es el truco: usaremos cada ventana de 56 días del pasado como datos de entrenamiento, y el objetivo para cada ventana será el valor que le sigue inmediatamente.

Keras en realidad tiene una función de utilidad interesante llamada `tf.keras.utils.timeseries_dataset_from_array()` para ayudarnos a preparar el conjunto de entrenamiento. Toma una serie de tiempo como entrada y crea un `tf.data.Dataset` (presentado en el [Capítulo 13](#)) que contiene todas las ventanas de la longitud deseada, así como sus objetivos correspondientes. A continuación se muestra un ejemplo que toma una serie de tiempo que contiene los números del 0 al 5 y crea un conjunto de datos que contiene todas las ventanas de longitud 3, con sus objetivos correspondientes, agrupados en lotes de tamaño 2:

```

importar tensorflow como tf

my_series = [0, 1, 2, 3, 4, 5] my_dataset =
tf.keras.utils.timeseries_dataset_from_array( my_series, target=my_series[3:], # los objetivos
están en 3
pasos
el futuro
longitud_secuencia=3,
tamaño_lote=2
)

```

Inspeccionemos el contenido de este conjunto de datos:

```

>>> lista(mi_conjunto_de_datos)
[(<tf.Tensor: forma=(2, 3), dtype=int32, numpy=
matriz([[0, 1, 2], [1, 2, 3]],
dtype=int32)>, <tf.Tensor: forma=(2,),
dtype=int32, numpy=array([3, 4 ],
dtype=int32)>),
(<tf.Tensor: forma=(1, 3), dtype=int32, numpy=array([[2, 3, 4]], dtype=int32)>,
<tf.Tensor: forma=(1,), dtype=int32, numpy=array([5], dtype=int32)>)]

```

Cada muestra en el conjunto de datos es una ventana de longitud 3, junto con su objetivo correspondiente (es decir, el valor inmediatamente después de la ventana).

Las ventanas son [0, 1, 2], [1, 2, 3] y [2, 3, 4], y sus respectivos objetivos son 3, 4 y 5. Dado que hay tres ventanas en total, lo cual es no es un múltiplo del tamaño del lote, el último lote solo contiene una ventana en lugar de dos.

Otra forma de obtener el mismo resultado es utilizar el método `window()` de la clase `Dataset` de `tf.data`. Es más complejo, pero le brinda control total, lo que le resultará útil más adelante en este capítulo, así que veamos cómo funciona. El método `window()` devuelve un conjunto de datos de conjuntos de datos de ventana:

```

>>> para window_dataset en
tf.data.Dataset.range(6).window(4, shift=1):

```

```
...     para elemento en window_dataset:  
...         imprimir(f"elemento", end=" ") imprimir()  
...  
...  
0 1 2 3  
1 2 3 4  
2 3 4 5  
3 4 5  
4 5  
5
```

En este ejemplo, el conjunto de datos contiene seis ventanas, cada una desplazada un paso en comparación con la anterior, y las últimas tres ventanas son más pequeñas porque han llegado al final de la serie. En general, querrás deshacerte de estas ventanas más pequeñas pasando `drop_remainder=True` al método `window()`.

El método `window()` devuelve un conjunto de datos anidado, análogo a una lista de listas. Esto es útil cuando desea transformar cada ventana llamando a sus métodos de conjunto de datos (por ejemplo, para mezclarlos o agruparlos).

Sin embargo, no podemos usar un conjunto de datos anidado directamente para el entrenamiento, ya que nuestro modelo esperará tensores como entrada, no conjuntos de datos.

Por lo tanto, debemos llamar al método `flat_map()`: convierte un conjunto de datos anidado en un conjunto de datos plano (uno que contiene tensores, no conjuntos de datos). Por ejemplo, supongamos que `{1, 2, 3}` representa un conjunto de datos que contiene la secuencia de tensores 1, 2 y 3. Si aplana el conjunto de datos anidado `[[1, 2], [3, 4, 5, 6]]`, obtienes el conjunto de datos plano `[1, 2, 3, 4, 5, 6]`.

Además, el método `flat_map()` toma una función como argumento, lo que le permite transformar cada conjunto de datos en el conjunto de datos anidado antes de aplanarlo. Por ejemplo, si pasa la función `lambda ds: ds.batch(2)` a `flat_map()`, transformará el conjunto de datos anidado `[[1, 2], [3, 4, 5, 6]]` en el conjunto de datos plano `[[[1, 2], [3, 4]], [[5, 6]]]`: es un conjunto de datos que contiene 3 tensores, cada uno de tamaño 2.

Con eso en mente, estamos listos para aplanar nuestro conjunto de datos:

```
>>> conjunto de datos = tf.data.Dataset.range(6).window(4, shift=1,
drop_remainder=True) >>>
conjunto de datos = dataset.flat_map(lambda window_dataset:
window_dataset.batch(4)) >>>
para window_tensor en el conjunto de datos:
...     print(f"{{window_tensor}}")
...
[0 1 2 3] [1 2
3 4] [2 3 4 5]
```

Dado que cada conjunto de datos de ventana contiene exactamente cuatro elementos, llamar al lote (4) en una ventana produce un único tensor de tamaño 4. ¡Genial! Ahora tenemos un conjunto de datos que contiene ventanas consecutivas representadas como tensores. Creamos una pequeña función auxiliar para que sea más fácil extraer ventanas de un conjunto de datos:

```
def to_windows(conjunto de datos, longitud):
    conjunto de datos = conjunto de datos.window(longitud,
shift=1, drop_remainder=True)
    return dataset.flat_map(lambda window_ds:
window_ds.batch(longitud))
```

El último paso es dividir cada ventana en entradas y destinos, utilizando el método map(). También podemos agrupar las ventanas resultantes en lotes de tamaño 2:

```
>>> conjunto de datos = to_windows(tf.data.Dataset.range(6), 4) # 3 entradas
+ 1 destino = 4 >>> conjunto
de datos = conjunto de datos.map( ventana lambda : (ventana[:-1],
ventana[ -1]))
>>> lista(dataset.batch(2))
[(<tf.Tensor: forma=(2, 3), dtype=int64, numpy=
array([[0, 1, 2], [1, 2, 3]])>,
<tf.Tensor:
forma=(2,), dtype=int64, numpy=array([3, 4])>) , (<tf.Tensor: forma=(1, 3),
dtype=int64, numpy=array([[2, 3, 4]])>, <tf.Tensor: forma=(1,), dtype=int64 ,
numpy=matriz([5])>)]
```

Como puede ver, ahora tenemos el mismo resultado que obtuvimos antes con la función `timeseries_dataset_from_array()` (con un poco más de esfuerzo, pero pronto valdrá la pena).

Ahora, antes de comenzar a entrenar, debemos dividir nuestros datos en un período de entrenamiento, un período de validación y un período de prueba. Por ahora nos centraremos en el número de pasajeros en tren. También lo reduciremos en un factor de un millón, para garantizar que los valores estén cerca del rango 0-1; Esto funciona muy bien con la inicialización de peso predeterminada y la tasa de aprendizaje:

```
rail_train = df["rail"]["2016-01":"2018-12"] / 1e6 rail_valid = df["rail"]
["2019-01":"2019-05"] / 1e6 rail_test = df[ "ferrocarril"]["2019-06":] / 1e6
```

## NOTA

Cuando se trata de series temporales, generalmente conviene dividirlas en el tiempo. Sin embargo, en algunos casos es posible que puedas dividirlo en otras dimensiones, lo que te dará un período de tiempo más largo para entrenar. Por ejemplo, si tiene datos sobre la salud financiera de 10 000 empresas entre 2001 y 2019, es posible que pueda dividir estos datos entre las diferentes empresas. Sin embargo, es muy probable que muchas de estas empresas estén fuertemente correlacionadas (por ejemplo, sectores económicos enteros pueden subir o bajar conjuntamente), y si tiene empresas correlacionadas en el conjunto de capacitación y el conjunto de pruebas, su conjunto de pruebas no será tan útil, ya que su medida del error de generalización estará sesgada de manera optimista.

A continuación, usemos `timeseries_dataset_from_array()` para crear conjuntos de datos para entrenamiento y validación. Dado que el descenso de gradiente espera que las instancias en el conjunto de entrenamiento sean independientes y estén distribuidas de manera idéntica (IID), como vimos en el [Capítulo 4](#), debemos establecer el argumento `shuffle=True` para mezclar las ventanas de entrenamiento (pero no su contenido):

```

seq_length = 56
train_ds = tf.keras.utils.timeseries_dataset_from_array( rail_train.to_numpy(),
    objetivos=rail_train[seq_length:],
    secuencia_longitud=seq_length, tamaño de
    lote=32, shuffle=True, semilla=42

) valid_ds = tf.keras.utils.timeseries_dataset_from_array( rail_valid.to_numpy(),
    objetivos=rail_valid[seq_length:],
    secuencia_longitud=seq_length,
    tamaño_lote=32

)

```

¡Y ahora estamos listos para construir y entrenar cualquier modelo de regresión que queramos!

## Previsión utilizando un modelo lineal

Probemos primero un modelo lineal básico. Usaremos la pérdida de Huber, que generalmente funciona mejor que minimizar directamente el MAE, como se analizó en [el Capítulo 10](#).

También usaremos la detención anticipada:

```

tf.random.set_seed(42) modelo
= tf.keras.Sequential([ tf.keras.layers.Dense(1,
    input_shape=[seq_length])
])
early_stopping_cb = tf.keras.callbacks.EarlyStopping( monitor="val_mae",
    paciencia=50, restaurar_best_weights=True)
opt =
tf.keras.optimizers.SGD(learning_rate=0.02, impulso=0.9)
model.compile(loss=
tf.keras.losses.Huber(), optimizador=opt, métricas=["mae"]) historial =
model.fit(train_ds,
validation_data=valid_ds, epochs=500,
devoluciones de llamada=[early_stopping_cb])

```

Este modelo alcanza un MAE de validación de aproximadamente 37,866 (su kilometraje puede variar). Eso es mejor que un pronóstico ingenuo, pero peor que el Modelo **5SARIMA**.

¿Podemos hacerlo mejor con un RNN? ¡Vamos a ver!

## Previsión utilizando un RNN simple

Probemos el RNN más básico, que contiene una única capa recurrente con solo una neurona recurrente, como vimos en la [Figura 15-1](#):

```
modelo =
    tf.keras.Sequential([
        tf.keras.layers.SimpleRNN(1, input_shape=[None, 1])
    ])
```

Todas las capas recurrentes en Keras esperan entradas 3D de forma [tamaño de lote, pasos de tiempo, dimensionalidad], donde la dimensionalidad es 1 para series temporales univariadas y más para series temporales multivariadas. Recuerde que el argumento `input_shape` ignora la primera dimensión (es decir, el tamaño del lote) y, dado que las capas recurrentes pueden aceptar secuencias de entrada de cualquier longitud, podemos establecer la segunda dimensión en Ninguna, que significa "cualquier tamaño". Por último, dado que estamos tratando con una serie de tiempo univariada, necesitamos que el tamaño de la última dimensión sea 1. Es por eso que especificamos la forma de entrada [Ninguna, 1]: significa "secuencias univariadas de cualquier longitud". Tenga en cuenta que los conjuntos de datos en realidad contienen entradas de forma [tamaño de lote, pasos de tiempo], por lo que nos falta la última dimensión, de tamaño 1, pero Keras tiene la amabilidad de agregarla en este caso.

Este modelo funciona exactamente como vimos antes: el estado inicial  $h$  se establece en 0 y se pasa a una única neurona recurrente, junto con el valor del primer paso de tiempo,  $x$ . La neurona calcula una suma ponderada de estos valores más el término de sesgo y aplica la función de activación al resultado, utilizando la función tangente hiperbólica de forma predeterminada. El resultado es la primera salida,  $y$ . En un RNN simple, esta salida también es el nuevo estado  $h$ . Este nuevo estado se pasa a la misma neurona recurrente junto con el siguiente valor de entrada,  $x(1)$ , y el

El proceso se repite hasta el último paso de tiempo. Al final, la capa simplemente genera el último valor: en nuestro caso, las secuencias tienen 56 pasos, por lo que el último valor es  $y_{55}$ . Todo esto se realiza simultáneamente para cada secuencia del lote, de los cuales son 32 en este caso.

### NOTA

De forma predeterminada, las capas recurrentes en Keras solo devuelven el resultado final. Para que devuelvan una salida por paso de tiempo, debes configurar `return_sequences=True`, como verás.

¡Ese es nuestro primer modelo recurrente! Es un modelo de secuencia a vector. Como hay una única neurona de salida, el vector de salida tiene un tamaño de 1.

Ahora, si compila, entrena y evalúa este modelo como el modelo anterior, encontrará que no sirve para nada: ¡su validación MAE es superior a 100.000! Ay. Era de esperarse por dos razones:

1. El modelo solo tiene una neurona recurrente, por lo que los únicos datos que puede usar para hacer una predicción en cada paso de tiempo son el valor de entrada en el paso de tiempo actual y el valor de salida del paso de tiempo anterior. ¡Eso no es mucho para continuar! En otras palabras, la memoria del RNN es extremadamente limitada: es solo un número, su salida anterior. Y contemos cuántos parámetros tiene este modelo: dado que solo hay una neurona recurrente con solo dos valores de entrada, todo el modelo solo tiene tres parámetros (dos pesos más un término de sesgo). Eso está lejos de ser suficiente para esta serie temporal. Por el contrario, nuestro modelo anterior podía analizar los 56 valores anteriores a la vez y tenía un total de 57 parámetros.
2. La serie de tiempo contiene valores de 0 a aproximadamente 1,4, pero como la función de activación predeterminada es `tanh`, la capa recurrente puede

sólo genera valores entre  $-1$  y  $+1$ . No hay forma de que pueda predecir valores entre  $1,0$  y  $1,4$ .

Solucionemos ambos problemas: crearemos un modelo con una capa recurrente más grande, que contenga 32 neuronas recurrentes, y agregaremos una capa de salida densa encima con una única neurona de salida y sin función de activación. La capa recurrente podrá transportar mucha más información de un paso de tiempo al siguiente, y la capa de salida densa proyectará la salida final desde 32 dimensiones hasta 1, sin ninguna restricción de rango de valores:

```
univar_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32,
        input_shape=[None, 1]),
    tf.keras.layers.Dense(1) # sin función de activación por
                           # defecto
])
```

Ahora bien, si compilas, ajustas y evalúas este modelo como el anterior, encontrarás que su MAE de validación llega a 27,703. Ese es el mejor modelo que hemos entrenado hasta ahora, e incluso supera al modelo SARIMA: ¡lo estamos haciendo bastante bien!

#### CONSEJO

Solo hemos normalizado la serie temporal, sin eliminar la tendencia y la estacionalidad, y aún así el modelo aún funciona bien. Esto es conveniente porque permite buscar rápidamente modelos prometedores sin preocuparse demasiado por el preprocessamiento. Sin embargo, para obtener el mejor rendimiento, es posible que desee intentar hacer que la serie temporal sea más estacionaria; por ejemplo, usando la diferenciación.

## Pronóstico utilizando un RNN profundo

Es bastante común apilar varias capas de celdas, como se muestra en la [Figura 15-10](#). Esto le brinda un RNN profundo.

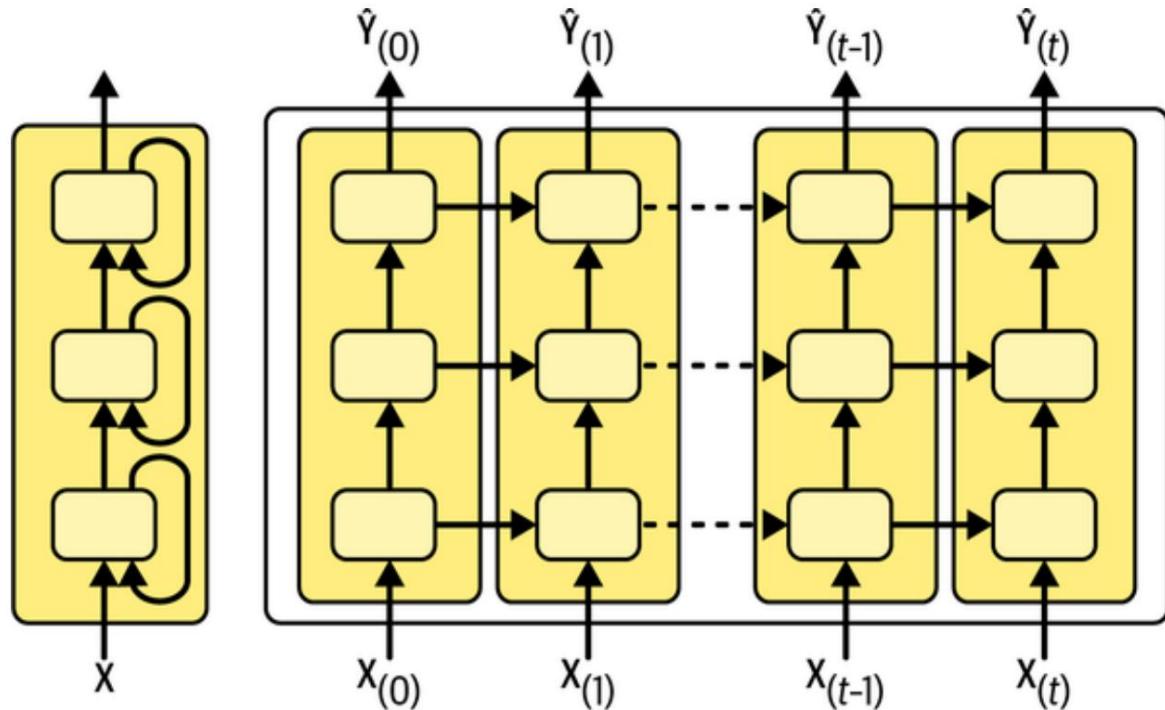


Figura 15-10. Un RNN profundo (izquierda) desplegado a través del tiempo (derecha)

Implementar un RNN profundo con Keras es sencillo: simplemente apile capas recurrentes. En el siguiente ejemplo, usamos tres capas SimpleRNN (pero podríamos usar cualquier otro tipo de capa recurrente, como una capa LSTM o una capa GRU, que discutiremos en breve).

Las dos primeras son capas de secuencia a secuencia y la última es una capa de secuencia a vector. Finalmente, la capa Densa produce el pronóstico del modelo (puede considerarla como una capa de vector a vector). Entonces, este modelo es igual al modelo representado en [la Figura 15-10](#), excepto que las salidas  $\hat{Y}$  a  $\hat{Y}$  se ignoran y hay una capa densa encima de la capa final para producir el pronóstico real:

```
deep_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32,
        return_sequences=True, input_shape=[None, 1]),
    tf.keras.layers.SimpleRNN(32,
        return_sequences=True),
    tf.keras.layers.Dense(1)
])
```

### ADVERTENCIA

Asegúrese de configurar `return_sequences=True` para todas las capas recurrentes (excepto la última, si solo le importa la última salida). Si olvida configurar este parámetro para una capa recurrente, generará una matriz 2D que contiene solo la salida del último paso de tiempo, en lugar de una matriz 3D que contiene salidas para todos los pasos de tiempo. La siguiente capa recurrente se quejará de que no le está suministrando secuencias en el formato 3D esperado.

Si entrena y evalúa este modelo, encontrará que alcanza un MAE de aproximadamente 31,211. Eso es mejor que ambas líneas de base, pero no supera a nuestro RNN "menos profundo". Parece que este RNN es demasiado grande para nuestra tarea.

## Previsión de series temporales multivariadas

Una gran cualidad de las redes neuronales es su flexibilidad: en particular, pueden manejar series temporales multivariadas casi sin cambios en su arquitectura. Por ejemplo, intentemos pronosticar la serie temporal del ferrocarril utilizando como entrada los datos del autobús y del ferrocarril. De hecho, ¡incorporemos también el tipo de día! Dado que siempre podemos saber de antemano si mañana será un día laborable, un fin de semana o un día festivo, podemos desplazar la serie de tipos de días un día hacia el futuro, de modo que el modelo reciba como entrada el tipo de día de mañana. Para simplificar, haremos este procesamiento usando Pandas:

```
df_mulvar = df[["bus", "rail"]] / 1e6 # usa series de bus y ferrocarril como entrada
df_mulvar["next_day_type"] =
df["day_type"].shift(-1) # conocemos el tipo de mañana df_mulvar = pd.get_dummies(df_mulvar)
# codificación one-hot del tipo
de día
```

Ahora `df_mulvar` es un DataFrame con cinco columnas: los datos del autobús y del ferrocarril, más tres columnas que contienen la codificación one-hot del siguiente

tipo de día (recuerde que hay tres tipos de días posibles, W, A y U). A continuación podemos proceder de forma muy parecida a como lo hicimos antes. Primero dividimos los datos en tres períodos, para entrenamiento, validación y prueba:

```
mulvar_train = df_mulvar["2016-01":"2018-12"] mulvar_valid =
df_mulvar["2019-01":"2019-05"] mulvar_test = df_mulvar["2019-06":]
```

Luego creamos los conjuntos de datos:

```
train_mulvar_ds =
tf.keras.utils.timeseries_dataset_from_array( mulvar_train.to_numpy(), # usar
las 5 columnas como entrada objetivos=mulvar_train["rail"][:seq_length:], # pronóstico
sólo la serie de rieles [...] # los
otros 4 argumentos son los mismos que antes

) valid_mulvar_ds =
tf.keras.utils.timeseries_dataset_from_array( mulvar_valid.to_numpy(),
target=mulvar_valid["rail"][:seq_length:],
[...] # los otros 2 argumentos son los mismos que antes

)
```

Y finalmente creamos el RNN:

```
mulvar_model = tf.keras.Sequential([
tf.keras.layers.SimpleRNN(32,
input_shape=[None, 5]),
tf.keras.layers.Dense(1)

])
```

Tenga en cuenta que la única diferencia con el RNN univar\_model que construimos anteriormente es la forma de la entrada: en cada paso de tiempo, el modelo ahora recibe cinco entradas en lugar de una. De hecho, este modelo alcanza una validación MAE de 22.062. ¡Ahora estamos haciendo grandes progresos!

De hecho, no es demasiado difícil hacer que RNN pronostique el número de pasajeros tanto en autobús como en tren. Solo necesita cambiar los objetivos al crear los conjuntos de datos, configurándolos en mulvar\_train[["bus", "rail"]]

[seq\_length:] para el conjunto de entrenamiento y mulvar\_valid[["bus", "rail"]][seq\_length:] para el conjunto de validación. También debe agregar una neurona adicional en la capa Densa de salida, ya que ahora debe hacer dos pronósticos: uno para el número de pasajeros en autobús de mañana y el otro para el tren. ¡Eso es todo al respecto!

Como analizamos en [el Capítulo 10](#), usar un modelo único para múltiples tareas relacionadas a menudo resulta en un mejor desempeño que usar un modelo separado para cada tarea, ya que las características aprendidas para una tarea pueden ser útiles para las otras tareas, y también porque tener que desempeñarse bien en múltiples tareas evita que el modelo se sobreajuste (es una forma de regularización). Sin embargo, depende de la tarea y, en este caso particular, el RNN multitarea que pronostica tanto el número de pasajeros en autobús como en tren no funciona tan bien como los modelos dedicados que pronostican uno u otro (utilizando las cinco columnas como entrada). Aún así, alcanza un MAE de validación de 25.330 para ferrocarril y 26.369 para autobús, lo cual es bastante bueno.

Pronosticar varios pasos de tiempo por delante Hasta ahora sólo hemos predicho el valor en el siguiente paso de tiempo, pero podríamos haberlo predicho con la misma facilidad varios pasos por delante cambiando los objetivos apropiadamente (por ejemplo, para predecir el número de pasajeros dentro de 2 semanas, podría simplemente cambiar los objetivos para que sean el valor con 14 días de anticipación en lugar de 1 día de anticipación). Pero ¿qué pasa si queremos predecir los siguientes 14 valores?

La primera opción es tomar el RNN univar\_model que entrenamos anteriormente para la serie temporal del ferrocarril, hacer que prediga el siguiente valor y agregar ese valor a las entradas, actuando como si el valor predicho realmente hubiera ocurrido; Luego usaríamos el modelo nuevamente para predecir el siguiente valor, y así sucesivamente, como en el siguiente código:

```
importar numpy como np
```

```

X = rail_valid.to_numpy()[:, np.newaxis, :seq_length, np.newaxis] para
step_ahead en
el rango(14): y_pred_one =
    univar_model.predict(X)
    X = np.concatenate([X, y_pred_one.reshape(1, 1, 1)], eje=1)

```

En este código, tomamos el número de pasajeros en tren de los primeros 56 días del período de validación y convertimos los datos a una matriz NumPy de forma [1, 56, 1] (recuerde que las capas recurrentes esperan entradas 3D). Luego usamos repetidamente el modelo para pronosticar el siguiente valor y agregamos cada pronóstico a la serie de entrada, a lo largo del eje de tiempo (eje = 1). Los pronósticos resultantes se representan en la Figura 15-11.

#### ADVERTENCIA

Si el modelo comete un error en un paso de tiempo, entonces los pronósticos para los siguientes pasos de tiempo también se ven afectados: los errores tienden a acumularse. Por lo tanto, es preferible utilizar esta técnica sólo en un pequeño número de pasos.

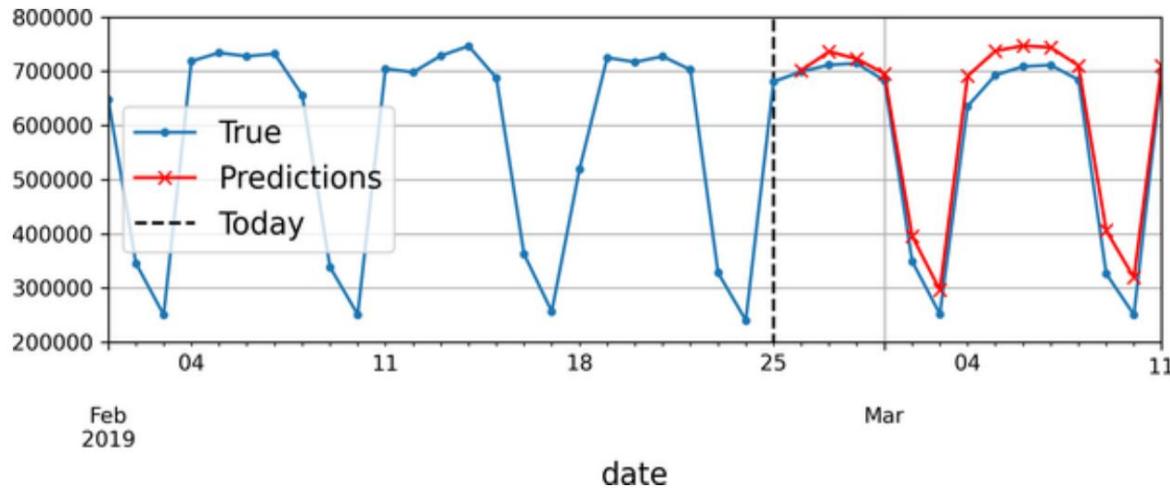


Figura 15-11. Pronosticando 14 pasos adelante, 1 paso a la vez

La segunda opción es entrenar un RNN para que prediga los siguientes 14 valores de una sola vez. Todavía podemos usar un modelo de secuencia a vector, pero generará 14 valores en lugar de 1. Sin embargo, primero debemos cambiar el

Los objetivos serán vectores que contengan los siguientes 14 valores. Para hacer esto, podemos usar `timeseries_dataset_from_array()` nuevamente, pero esta vez pidiéndole que cree conjuntos de datos sin objetivos (`objetivos=Ninguno`) y con secuencias más largas, de longitud `seq_length + 14`. Luego podemos usar el método `map()` de los conjuntos de datos para aplicar una función personalizada a cada lote de secuencias, dividiéndolas en entradas y objetivos. En este ejemplo, utilizamos la serie temporal multivariada como entrada (usando las cinco columnas) y pronosticamos el número de pasajeros en tren para los próximos 14 días:

6

```
def split_inputs_and_targets(mulvar_series, forward=14, target_col=1): devuelve
    mulvar_series[:, :-forward], mulvar_series[:, -forward:, target_col]

forward_train_ds =
    tf.keras.utils.timeseries_dataset_from_array( mulvar_train.to_numpy(),
        target=None, secuencia_length=seq_length
        + 14, [...] # los otros
        3 argumentos son los mismos que
        antes ).map(split_inputs_and_targets) forward_valid_ds =
           

tf .keras.utils.timeseries_dataset_from_array( mulvar_valid.to_numpy(),
    objetivos=Ninguno,
    secuencia_longitud=seq_length + 14,
    tamaño_lote=32 ).map(split_inputs_and_targets)
```

Ahora sólo necesitamos que la capa de salida tenga 14 unidades en lugar de 1:

```
forward_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32,
        input_shape=[Ninguno, 5]),
    tf.keras.layers.Dense(14)
])
```

Después de entrenar este modelo, puedes predecir los siguientes 14 valores a la vez de esta manera:

```
X = mulvar_valid.to_numpy()[:, np.newaxis, :seq_length] # forma [1, 56, 5]
```

```
Y_pred = forward_model.predict(X) # forma [1, 14]
```

Este enfoque funciona bastante bien. Sus pronósticos para el día siguiente son obviamente mejores que sus pronósticos para 14 días en el futuro, pero no acumula errores como lo hacía el enfoque anterior. Sin embargo, aún podemos hacerlo mejor utilizando un modelo de secuencia a secuencia (o seq2seq) .

## Pronóstico utilizando una secuencia a secuencia Modelo

En lugar de entrenar el modelo para pronosticar los siguientes 14 valores solo en el último paso de tiempo, podemos entrenarlo para pronosticar los siguientes 14 valores en todos y cada uno de los pasos de tiempo. En otras palabras, podemos convertir este RNN de secuencia a vector en un RNN de secuencia a secuencia. La ventaja de esta técnica es que la pérdida contendrá un término para la salida del RNN en todos y cada uno de los pasos de tiempo, no solo para la salida en el último paso de tiempo.

Esto significa que habrá muchos más gradientes de error fluyendo a través del modelo y no tendrán que fluir tanto a través del tiempo ya que provendrán de la salida de cada paso de tiempo, no solo del último. Esto estabilizará y acelerará el entrenamiento.

Para ser claros, en el paso de tiempo 0 el modelo generará un vector que contiene los pronósticos para los pasos de tiempo 1 a 14, luego en el paso de tiempo 1 el modelo pronosticará los pasos de tiempo 2 a 15, y así sucesivamente. En otras palabras, los objetivos son secuencias de ventanas consecutivas, desplazadas un paso de tiempo en cada paso de tiempo. El objetivo ya no es un vector, sino una secuencia de la misma longitud que las entradas, que contiene un vector de 14 dimensiones en cada paso.

Preparar los conjuntos de datos no es trivial, ya que cada instancia tiene una ventana como entrada y una secuencia de ventanas como salida. Una manera de

Hacer esto es usar la función de utilidad `to_windows()` que creamos anteriormente, dos veces seguidas, para obtener ventanas de ventanas consecutivas. Por ejemplo, conviertamos la serie de números del 0 al 6 en un conjunto de datos que contenga secuencias de 4 ventanas consecutivas, cada una de longitud 3:

```
>>> mi_series = tf.data.Dataset.range(7) >>> conjunto de
datos = to_windows(to_windows(mi_series, 3), 4) >>> lista(conjunto de datos)
[<tf.Tensor: forma=(4, 3 ),
dtype=int64, numpy= matriz([[0, 1, 2], [1, 2, 3], [2, 3, 4], [3, 4, 5]])>,
<tf.Tensor: forma=(4, 3), dtype=int64, numpy= matriz([[1, 2, 3], [2, 3,
4], [3, 4, 5], [4, 5 , 6]])>]
```

Ahora podemos usar el método `map()` para dividir estas ventanas de ventanas en entradas y destinos:

```
>>> conjunto de datos = conjunto de datos.map(lambda S: (S[:, 0], S[:, 1:])) >>>
lista(conjunto de datos)
[(<tf.Tensor: forma=(4,), dtype=int64, numpy=array([0, 1, 2, 3])>, <tf.Tensor: forma=(4,
2),
dtype=int64, numpy= array([[1, 2], [2 , 3], [3, 4], [4, 5]])>), (<tf.Tensor:
forma=(4,),
```

$$\text{dtype}=\text{int64}, \text{numpy}=\text{array}([1, 2, 3, 4] )\>, <\!\!\text{tf.Tensor: forma}=(4, 2), \text{dtype}=\text{int64}, \text{numpy}=\text{matriz}([[2, 3], [3, 4], [4, 5], [5, 6]])\!\!>)]$$

Ahora el conjunto de datos contiene secuencias de longitud 4 como entradas, y los objetivos son secuencias que contienen los dos pasos siguientes, para cada paso de tiempo. Por ejemplo, la primera secuencia de entrada es [0, 1, 2, 3] y sus objetivos correspondientes son [[1, 2], [2, 3], [3, 4], [4, 5]], lo que son los dos valores siguientes para cada paso de tiempo. Si eres como yo, probablemente necesitarás unos minutos para entender esto. ¡Tome su tiempo!

## NOTA

Puede resultar sorprendente que los objetivos contengan valores que aparecen en las entradas. ¿No es eso hacer trampa? Afortunadamente, no en absoluto: en cada paso de tiempo, un RNN solo conoce los pasos de tiempo pasados; no puede mirar hacia adelante. Se dice que es un modelo causal .

Creemos otra pequeña función de utilidad para preparar los conjuntos de datos para nuestro modelo de secuencia a secuencia. También se encargará de barajar (opcional) y agrupar:

```
def to_seq2seq_dataset(serie, seq_length=56, adelante=14, target_col=1,
                      tamaño_lote = 32, aleatorio = Falso,
                      semilla = Ninguna):
    ds =
        to_windows(tf.data.Dataset.from_tensor_slices(series), forward + 1) ds =
        to_windows(ds,
                   seq_length).map(lambda S: (S[:, 0], S[:, 1:, 1])) si se mezcla:
                    ds = ds.shuffle(8 * tamaño_lote, semilla=semilla)
                    devolver ds.batch(tamaño_batch)
```

Ahora podemos usar esta función para crear los conjuntos de datos:

```
seq2seq_train = to_seq2seq_dataset(mulvar_train, shuffle=True,
                                    semilla=42) seq2seq_valid =
        to_seq2seq_dataset(mulvar_valid)
```

Y por último, podemos construir el modelo secuencia a secuencia:

```
seq2seq_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, return_sequences=True, input_shape=[None, 5]),
    tf.keras.layers.Dense(14)

])
```

Es casi idéntico a nuestro modelo anterior: la única diferencia es que configuramos `return_sequences=True` en la capa SimpleRNN. De esta manera, generará una secuencia de vectores (cada uno de tamaño 32), en lugar de generar un solo vector en el último paso de tiempo. La capa Densa es lo suficientemente inteligente como para manejar secuencias como entrada: se aplicará en cada paso de tiempo, tomando un vector de 32 dimensiones como entrada y generando un vector de 14 dimensiones. De hecho, otra forma de obtener exactamente el mismo resultado es utilizar una capa Conv1D con un tamaño de kernel de 1: `Conv1D(14, kernel_size=1)`.

#### CONSEJO

Keras ofrece una capa `TimeDistributed` que le permite aplicar cualquier capa de vector a vector a cada vector en las secuencias de entrada, en cada paso de tiempo. Lo hace de manera eficiente, remodelando las entradas para que cada paso de tiempo se trate como una instancia separada, luego remodela las salidas de la capa para recuperar la dimensión de tiempo. En nuestro caso, no lo necesitamos ya que la capa Densa ya admite secuencias como entradas.

El código de entrenamiento es el mismo de siempre. Durante el entrenamiento, se utilizan todos los resultados del modelo, pero después del entrenamiento solo importa el resultado del último paso de tiempo y el resto se puede ignorar. Por ejemplo, podemos pronosticar el número de pasajeros en tren para los próximos 14 días de esta manera:

```
X = mulvar_valid.to_numpy()[:, np.newaxis, :seq_length] y_pred_14 =
seq2seq_model.predict(X)[0, -1] # solo el
```

## salida del último paso

Si evalúas los pronósticos de este modelo para  $t + 1$ , encontrarás un MAE de validación de 25,519. Para  $t + 2$  es 26,274 y el rendimiento continúa cayendo gradualmente a medida que el modelo intenta hacer pronósticos en el futuro. En  $t + 14$ , el MAE es 34,322.

### CONSEJO

Puede combinar ambos enfoques para pronosticar con varios pasos de anticipación: por ejemplo, puede entrenar un modelo que pronostique con 14 días de anticipación, luego tomar su resultado y agregarlo a las entradas, luego ejecutar el modelo nuevamente para obtener pronósticos para los siguientes 14 días. y posiblemente repetir el proceso.

Los RNN simples pueden ser bastante buenos para pronosticar series de tiempo o manejar otros tipos de secuencias, pero no funcionan tan bien en series o secuencias de tiempo largas. Discutamos por qué y veamos qué podemos hacer al respecto.

## Manejo de secuencias largas

Para entrenar un RNN en secuencias largas, debemos ejecutarlo en muchos pasos de tiempo, lo que hace que el RNN desenrollado sea una red muy profunda. Al igual que cualquier red neuronal profunda, puede sufrir el problema de los gradientes inestables, que se analiza en [el Capítulo 11](#): puede tardar una eternidad en entrenarse, o el entrenamiento puede ser inestable. Además, cuando un RNN procesa una secuencia larga, gradualmente olvidará las primeras entradas de la secuencia. Veamos ambos problemas, comenzando con el problema de los gradientes inestables.

## Luchando contra el problema de los gradientes inestables

Muchos de los trucos que usamos en redes profundas para aliviar el problema de los gradientes inestables también se pueden usar para RNN: buen parámetro

inicialización, optimizadores más rápidos, abandono, etc. Sin embargo, las funciones de activación no saturantes (p. ej., ReLU) pueden no ser de mucha ayuda en este caso. De hecho, pueden hacer que el RNN sea aún más inestable durante el entrenamiento. ¿Por qué? Bueno, supongamos que el descenso de gradiente actualiza los pesos de una manera que aumenta ligeramente las salidas en el primer paso. Debido a que se utilizan los mismos pesos en cada paso de tiempo, las salidas en el segundo paso de tiempo también pueden aumentar ligeramente, y las del tercero, y así sucesivamente hasta que las salidas exploten (y una función de activación no saturante no impide eso).

Puede reducir este riesgo utilizando una tasa de aprendizaje menor o puede utilizar una función de activación de saturación como la tangente hiperbólica (esto explica por qué es la opción predeterminada).

De la misma manera, los propios gradientes pueden explotar. Si nota que el entrenamiento es inestable, es posible que desee controlar el tamaño de los gradientes (por ejemplo, usando TensorBoard) y tal vez utilizar el recorte de gradientes.

Además, la normalización por lotes no se puede utilizar de manera tan eficiente con RNN como con redes de retroalimentación profunda. De hecho, no puedes usarlo entre pasos de tiempo, sólo entre capas recurrentes.

Para ser más precisos, es técnicamente posible agregar una capa BN a una celda de memoria (como verá en breve) para que se aplique en cada paso de tiempo (tanto en las entradas para ese paso de tiempo como en el estado oculto de el paso anterior). Sin embargo, se utilizará la misma capa BN en cada paso de tiempo, con los mismos parámetros, independientemente de la escala real y el desplazamiento de las entradas y el estado oculto. En la práctica esto no produce buenos resultados, como lo demostraron César Laurent et al. en un [artículo de 2015](#):

Los autores descubrieron que BN era <sup>7</sup> ligeramente beneficioso sólo cuando se aplicaba a las entradas de la capa, no a los estados ocultos. En otras palabras, fue ligeramente mejor que nada cuando se aplicó entre capas recurrentes (es decir, verticalmente en la [Figura 15-10](#)), pero no dentro de capas recurrentes (es decir, horizontalmente). En Keras, puedes aplicar BN entre capas simplemente agregando un

capa BatchNormalization antes de cada capa recurrente, pero ralentizará el entrenamiento y puede que no ayude mucho.

Otra forma de normalización suele funcionar mejor con los RNN: la normalización de capas. Esta idea fue introducida por Jimmy Lei Ba et al. en un [artículo de 2016](#):<sup>8</sup> Es muy similar a la normalización por lotes, pero en lugar de normalizar en la dimensión del lote, la normalización de capas se normaliza en la dimensión de las entidades. Una ventaja es que puede calcular las estadísticas requeridas sobre la marcha, en cada paso de tiempo, de forma independiente para cada instancia. Esto también significa que se comporta de la misma manera durante el entrenamiento y las pruebas (a diferencia de BN), y no necesita usar promedios móviles exponenciales para estimar las estadísticas de características en todas las instancias del conjunto de entrenamiento, como lo hace BN. Al igual que BN, la normalización de capas aprende una escala y un parámetro de compensación para cada entrada. En un RNN, normalmente se usa justo después de la combinación lineal de las entradas y los estados ocultos.

Usemos Keras para implementar la normalización de capas dentro de una celda de memoria simple. Para hacer esto, necesitamos definir una celda de memoria personalizada, que es como una capa normal, excepto que su método `call()` toma dos argumentos: las entradas en el paso de tiempo actual y los estados ocultos del paso de tiempo anterior.

Tenga en cuenta que el argumento de estados es una lista que contiene uno o más tensores. En el caso de una celda RNN simple, contiene un único tensor igual a las salidas del paso de tiempo anterior, pero otras celdas pueden tener múltiples tensores de estado (por ejemplo, una LSTMCell tiene un estado a largo plazo y un estado a corto plazo, como lo verás en breve). Una celda también debe tener un atributo `state_size` y un atributo `output_size`. En un RNN simple, ambos son simplemente iguales al número de unidades. El siguiente código implementa una celda de memoria personalizada que se comportará como `SimpleRNNCell`, excepto que también aplicará la normalización de capa en cada paso de tiempo:

```
clase LNSimpleRNNCell(tf.keras.layers.Layer): def __init__(self,  
 unidades, activación="tanh", **kwargs):
```

```

super().__init__(**kwargs) self.state_size =
unidades self.output_size = unidades
self.simple_rnn_cell =
tf.keras.layers.SimpleRNNCell(unidades,
activación=Ninguno)
self.layer_norm =
tf.keras.layers.LayerNormalization()
auto.activación =
tf.keras.activations.get(activación)

llamada def (auto, entradas, estados):
    salidas, nuevos_estados = self.simple_rnn_cell(entradas,
estados)
    salidas_norma =
self.activation(self.layer_norm(salidas))
    devolver salidas_normas, [salidas_normas]

```

Repasemos este código:

- Nuestra clase LNSimpleRNNCell hereda de la clase `tf.keras.layers.Layer`, como cualquier capa personalizada.
- El constructor toma el número de unidades y la función de activación deseada y establece los atributos `state_size` y `output_size`, luego crea una `SimpleRNNCell` sin función de activación (porque queremos realizar la normalización de capas después de la operación lineal pero antes de la función de activación)<sup>9</sup>. Luego, el constructor crea la capa `LayerNormalization` y finalmente obtiene la función de activación deseada.
- El método `call()` comienza aplicando `simpleRNNCell`, que calcula una combinación lineal de las entradas actuales y los estados ocultos anteriores, y devuelve el resultado dos veces (de hecho, en `SimpleRNNCell`, las salidas son exactamente iguales a los estados ocultos: en otras palabras, `new_states[0]` es igual a las salidas, por lo que podemos ignorar con seguridad `new_states` en el resto de

el método call()). A continuación, el método call() aplica la normalización de capas, seguido de la función de activación. Finalmente, devuelve las salidas dos veces: una como salidas y otra como los nuevos estados ocultos. Para usar esta celda personalizada, todo lo que necesitamos hacer es crear una capa tf.keras.layers.RNN y pasarle una instancia de celda:

```
custom_ln_model = tf.keras.Sequential([
    tf.keras.layers.RNN(LNSimpleRNNCell(32),
        return_sequences=True,
        input_shape=[None, 5]),
    tf.keras.layers.Dense(14)
])
```

De manera similar, puede crear una celda personalizada para aplicar la exclusión entre cada paso de tiempo. Pero hay una forma más sencilla: la mayoría de las capas y celdas recurrentes proporcionadas por Keras tienen hiperparámetros de abandono y abandono\_recurrente: el primero define la tasa de abandono que se aplicará a las entradas y el segundo define la tasa de abandono para los estados ocultos, entre pasos de tiempo. Por lo tanto, no es necesario crear una celda personalizada para aplicar la exclusión en cada paso de tiempo en un RNN.

Con estas técnicas, puede aliviar el problema de los gradientes inestables y entrenar un RNN de manera mucho más eficiente. Ahora veamos cómo abordar el problema de la memoria a corto plazo.

#### CONSEJO

Al pronosticar series temporales, suele resultar útil tener algunas barras de error junto con las predicciones. Para esto, un enfoque es usar la deserción de MC, presentada en [el Capítulo 11](#): usar recurrent\_dropout durante el entrenamiento, luego mantener la deserción activa en el momento de la inferencia llamando al modelo usando model(X, entrenamiento=True). Repita esto varias veces para obtener múltiples pronósticos ligeramente diferentes, luego calcule la media y la desviación estándar de estas predicciones para cada paso de tiempo.

Abordar el problema de la memoria a corto plazo Debido a las transformaciones que atraviesan los datos al atravesar un RNN, parte de la información se pierde en cada paso de tiempo. Después de un tiempo, el estado del RNN prácticamente no contiene rastros de las primeras entradas. Esto puede ser espectacular. Imagínate al pez Dory intentando<sup>10</sup> traducir una frase larga; Cuando termina de leerlo, no tiene idea de cómo empezó. Para abordar este problema se han introducido varios tipos de células con memoria a largo plazo. Han tenido tanto éxito que las células básicas ya no se utilizan mucho. Veamos primero la más popular de estas células de memoria a largo plazo: la célula LSTM.

## células LSTM

La célula de memoria a largo plazo (LSTM) se propuso en 1997 por Sepp<sup>11</sup> Hochreiter y Jürgen Schmidhuber y mejorado gradualmente a lo largo de los años por varios investigadores, como Alex Graves, Haşim Sak y Wojciech Zaremba.<sup>12</sup> Si considera la celda LSTM como una caja negra, se puede usar de manera muy similar a una celda básica, excepto que funcionará mucho mejor; el entrenamiento convergerá más rápido y detectará patrones a más largo plazo en los datos. En Keras, simplemente puedes usar la capa LSTM en lugar de la capa SimpleRNN:

```
modelo = tf.keras.Sequential([
    tf.keras.layers.LSTM(32, return_sequences=True,
    input_shape=[None, 5]),
    tf.keras.layers.Dense(14)
])
```

Alternativamente, puede usar la capa `tf.keras.layers.RNN` de propósito general y darle un `LSTMCell` como argumento. Sin embargo, la capa LSTM utiliza una implementación optimizada cuando se ejecuta en una GPU (consulte [el Capítulo 19](#)), por lo que en general es preferible usarla (la capa RNN es más útil cuando define celdas personalizadas, como hicimos antes).

Entonces, ¿cómo funciona una celda LSTM? Su arquitectura se muestra en [la Figura 15-12](#). Si no miras lo que hay dentro de la caja, la celda LSTM se ve exactamente como una celda normal, excepto que su estado se divide en dos vectores:  $h$  y  $c$  ("c" significa "celda"). Puedes pensar en  $h$  como el estado de corto plazo y  $c$  como el estado de largo plazo.

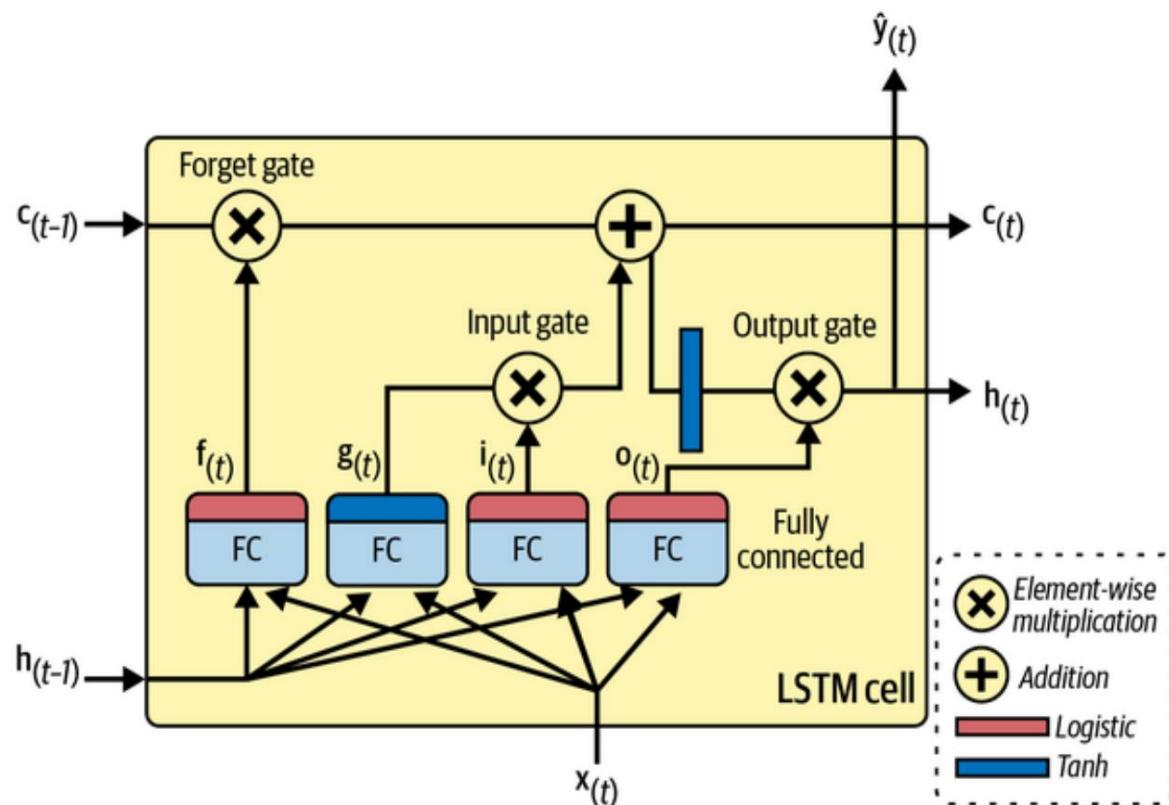


Figura 15-12. Una celda LSTM

¡Ahora abramos la caja! La idea clave es que la red pueda aprender qué almacenar a largo plazo, qué desechar y qué leer de él. A medida que el estado a largo plazo  $c$  atraviesa la red de izquierda a derecha, puede ver que primero pasa por una puerta de olvido, eliminando algunos recuerdos, y luego agrega algunos recuerdos nuevos mediante la operación de suma (que agrega los recuerdos que fueron seleccionados). Por una puerta de entrada. El resultado  $c$  se envía directamente, sin ninguna transformación adicional. Entonces, en cada paso de tiempo, algunos recuerdos se eliminan y otros se agregan. Además, después de la operación de suma, el estado a largo plazo se copia y pasa a través de la función tanh, y luego la puerta de salida filtra el resultado. Este

produce el estado a corto plazo  $h_t$  (que es igual a la salida de la celda ( $t$ ) para este paso de tiempo,  $y_t$ ). Ahora veamos de dónde vienen los nuevos recuerdos y cómo funcionan las puertas.

Primero, el vector de entrada actual  $x_t$  y el estado anterior a corto plazo ( $t-1$ )  $h_{t-1}$  se alimentan a cuatro capas diferentes completamente conectadas. Todos tienen un propósito ( $t-1$ ) diferente:

- La capa principal es la que genera  $g_t$ . Tiene la función habitual de analizar las entradas actuales  $x_t$  y el estado anterior (a corto plazo) ( $t-1$ )  $h_{t-1}$ . En una celda básica, no hay nada más que esta capa, y su salida va directamente a  $y_t$  y  $h_t$ . Pero en una celda LSTM, la salida de esta capa no sale directamente; en cambio, sus partes más importantes se almacenan en un estado a largo plazo ( $y_t$  y el resto se descarta).
- Las otras tres capas son controladores de puerta. Dado que utilizan la función de activación logística, las salidas varían de 0 a 1. Como puede ver, las salidas de los controladores de puerta se alimentan a operaciones de multiplicación por elementos: si generan 0, cierran la puerta, y si generan 1, abren la puerta. Específicamente:
  - La puerta de olvido (controlada por  $f_t$ ) controla qué partes del estado a largo plazo deben borrarse.
  - La puerta de entrada (controlada por  $i_t$ ) controla qué partes del estado a largo plazo deben agregarse al estado a largo plazo.
  - Finalmente, la puerta de salida (controlada por  $o_t$ ) controla qué partes del estado a largo plazo deben leerse y emitirse en este paso de tiempo, tanto a  $h_t$  como a  $y_t$ .

En resumen, una celda LSTM puede aprender a reconocer una entrada importante (esa es la función de la puerta de entrada), almacenarla en un estado a largo plazo y conservarla durante el tiempo que sea necesario (esa es la función de la puerta de olvido). y extráigalo cuando sea necesario. Esto explica por qué estos

Las células han tenido un éxito sorprendente en capturar datos a largo plazo. patrones en series temporales, textos largos, grabaciones de audio y más.

**La ecuación 15-4** resume cómo calcular el largo plazo de la celda. estado, su estado a corto plazo y su salida en cada paso de tiempo para un instancia única (las ecuaciones para un mini lote completo son muy similar).

### Ecuación 15-4. Cálculos LSTM

$$i(t) = \sigma(Wxi \quad x(t) + Whi \quad h(t-1) + bi)$$

$$f(t) = \sigma(Wxf \quad x(t) + Whf \quad h(t-1) + bf )$$

$$o(t) = \sigma(Wxo \quad x(t) + Quién \quad h(t-1) + bo)$$

$$g(t) = \tanh (Wxg \quad x(t) + Whg \quad h(t-1) + bg)$$

$$c(t) = f(t) \quad c(t-1) + i(t) \quad g(t)$$

$$y(t) = h(t) = o(t) \quad \tanh (c(t))$$

En esta ecuación:

- $W_{xi}, W_{xf}, W_{xo}$ , y  $W$  son las matrices de pesos de cada uno de los cuatro capas para su conexión al vector de entrada  $x$ .  $(t)$
- $W_{quién}$  es  $h(t-1)$ ? y  $W$  son las matrices de pesos de cada uno de los cuatro capas para su conexión con el estado anterior a corto plazo  $h_{(t-1)}$
- $b_{yo}, b_{bf}$  para , y  $b$  son los términos de sesgo para cada una de las cuatro capas. Tenga en cuenta que TensorFlow inicializa  $b$  en un vector lleno de unos en lugar de 0s. Esto evita olvidar todo al principio de capacitación.

Existen varias variantes de la celda LSTM. Uno particularmente popular La variante es la celda GRU, que veremos ahora.

### Células GRU

La celda de unidad recurrente cerrada (GRU) (ver [Figura 15-13](#)) fue propuesta por Kyunghyun Cho et al. en un [artículo de 2014](#)<sup>14</sup> eso también introdujo la red codificador-decodificador que discutimos anteriormente.

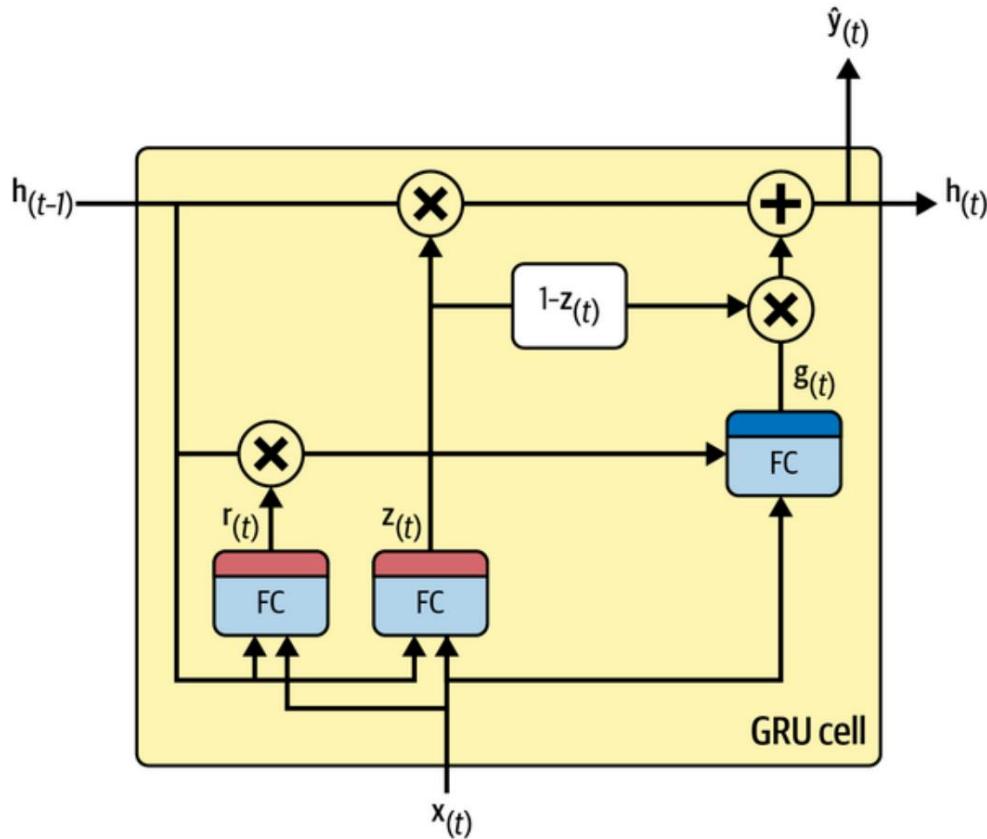


Figura 15-13. celda GRU

La celda GRU es una versión simplificada de la celda LSTM y parece funcionar igual de bien (lo que explica su creciente popularidad).<sup>15</sup>

Estas son las principales simplificaciones:

- Ambos vectores de estado se fusionan en un solo vector  $h . (t)$
- Un único controlador de puerta  $z$  controla tanto la puerta de olvido como la puerta de entrada. Si el controlador de puerta genera un 1, la puerta de olvido está abierta ( $= 1$ ) y la puerta de entrada está cerrada ( $1 - 1 = 0$ ). Si genera un 0, sucede lo contrario. En otras palabras, siempre que se debe almacenar un recuerdo, primero se borra la ubicación donde se almacenará. En realidad, esta es una variante frecuente de la celda LSTM en sí misma.

- No hay puerta de salida; el vector de estado completo se genera en cada paso de tiempo. Sin embargo, hay un nuevo controlador de puerta  $r$  que controla qué parte del estado anterior se mostrará a la capa principal ( $g$ ). ( $t$ )

**La ecuación 15-5** resume cómo calcular el estado de la celda en cada paso de tiempo para una sola instancia.

#### Ecuación 15-5. Cálculos de GRU

$$z(t) = \sigma(W_{xz}x(t) + W_{hz}h(t-1) + bz)$$

$$r(t) = \sigma(W_{xr}x(t) + W_{hr}h(t-1) + br)$$

$$g(t) = \tanh(W_{xg}x(t) + W_{hg}(r(t)h(t-1)) + bg)$$

$$h(t) = z(t)h(t-1) + (1 - z(t))g(t)$$

Keras proporciona una capa `tf.keras.layers.GRU`: usarla es solo cuestión de reemplazar `SimpleRNN` o `LSTM` con `GRU`. También proporciona `tf.keras.layers.GRUCell`, en caso de que desee crear una celda personalizada basada en una celda `GRU`.

Las células LSTM y GRU son una de las principales razones del éxito de los RNN. Sin embargo, si bien pueden abordar secuencias mucho más largas que los RNN simples, todavía tienen una memoria a corto plazo bastante limitada y les resulta difícil aprender patrones a largo plazo en secuencias de 100 pasos de tiempo o más, como muestras de audio, mucho tiempo. series o frases largas. Una forma de solucionar esto es acortar las secuencias de entrada; por ejemplo, utilizando capas convolucionales 1D.

#### Uso de capas convolucionales 1D para procesar secuencias

En [el Capítulo 14](#), vimos que una capa convolucional 2D funciona deslizando varios núcleos (o filtros) bastante pequeños a través de una imagen, produciendo múltiples mapas de características 2D (uno por núcleo). De manera similar, una capa convolucional 1D desliza varios núcleos a lo largo de una secuencia, produciendo un mapa de características 1D por núcleo. Cada núcleo aprenderá a

detectar un único patrón secuencial muy corto (no más largo que el tamaño del núcleo). Si usa 10 núcleos, entonces la salida de la capa estará compuesta por 10 secuencias 1D (todas de la misma longitud) o, de manera equivalente, puede ver esta salida como una única secuencia 10D. Esto significa que puede construir una red neuronal compuesta por una combinación de capas recurrentes y capas convolucionales 1D (o incluso capas de agrupación 1D). Si utiliza una capa convolucional 1D con un paso de 1 y el "mismo" relleno, entonces la secuencia de salida tendrá la misma longitud que la secuencia de entrada. Pero si usa un relleno "válido" o una zancada mayor que 1, entonces la secuencia de salida será más corta que la secuencia de entrada, así que asegúrese de ajustar los objetivos en consecuencia.

Por ejemplo, el siguiente modelo es el mismo que el anterior, excepto que comienza con una capa convolucional 1D que reduce la secuencia de entrada en un factor de 2, usando un paso de 2. El tamaño del núcleo es mayor que el paso, por lo que todas las entradas usarse para calcular la salida de la capa y, por lo tanto, el modelo puede aprender a preservar la información útil, eliminando solo los detalles sin importancia. Al acortar las secuencias, la capa convolucional puede ayudar a las capas GRU a detectar patrones más largos, por lo que podemos permitirnos duplicar la longitud de la secuencia de entrada a 112 días. Tenga en cuenta que también debemos recortar los primeros tres pasos de tiempo en los objetivos: de hecho, el tamaño del núcleo es 4, por lo que la primera salida de la capa convolucional se basará en los pasos de tiempo de entrada 0 a 3, y los primeros pronósticos serán para los pasos de tiempo 4 a 17 (en lugar de los pasos de tiempo 1 a 14). Además, debemos reducir la muestra de los objetivos en un factor de 2, debido al paso:

```
conv_rnn_model = tf.keras.Sequential([
    tf.keras.layers.Conv1D(filters=32,
        kernel_size=4, strides=2,
        activation="relu", input_shape=
        [None, 5]),
    tf.keras.layers.GRU(32, return_sequences=True),
    tf.keras.layers.Dense(14)
])

tren_más_largo = to_seq2seq_dataset(tren_mulvar,
```

```

longitud_secuencia=112,
                                barajar = Verdadero,
semilla=42)

long_valid = to_seq2seq_dataset(mulvar_valid, seq_length=112)
downsampled_train =
long_train.map(lambda X, Y: (X, Y[:, 3::2])) downsampled_valid = long_valid.map(lambda X, Y: (X,
Y[:, 3::2]))

[...] # compilar y ajustar el modelo utilizando los conjuntos de datos reducidos

```

Si entrena y evalúa este modelo, encontrará que supera al modelo anterior (por un pequeño margen). De hecho, ¡es posible utilizar solo capas convolucionales 1D y eliminar las capas recurrentes por completo!

## OndaNet

En un [artículo de 2016](#), Aaron van den Oord y otros investigadores de DeepMind introdujeron una arquitectura novedosa llamada WaveNet. Apilaron capas convolucionales 1D, duplicando la tasa de dilatación (qué tan separadas están las entradas de cada neurona) en cada capa: la primera capa convolucional vislumbra solo dos pasos de tiempo a la vez, mientras que la siguiente ve cuatro pasos de tiempo (su receptivo El campo tiene cuatro pasos de tiempo), el siguiente tiene ocho pasos de tiempo, y así sucesivamente ([consulte la Figura 15-14](#)). De esta manera, las capas inferiores aprenden patrones a corto plazo, mientras que las capas superiores aprenden patrones a largo plazo. Gracias a la duplicación de la tasa de dilatación, la red puede procesar secuencias extremadamente grandes de manera muy eficiente.

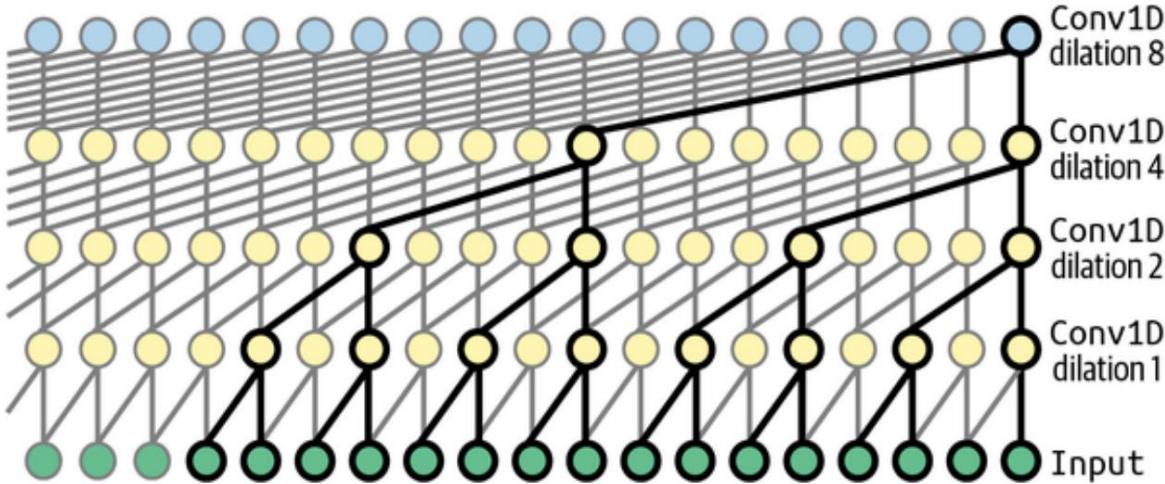


Figura 15-14. Arquitectura WaveNet

Los autores del artículo en realidad apilaron 10 capas convolucionales con tasas de dilatación de 1, 2, 4, 8,..., 256, 512, luego apilaron otro grupo de 10 capas idénticas (también con tasas de dilatación de 1, 2, 4, 8, ..., 256, 512), luego nuevamente otro grupo idéntico de 10 capas. Justificaron esta arquitectura señalando que una sola pila de 10 capas convolucionales con estas tasas de dilatación actuará como una capa convolucional súper eficiente con un núcleo de tamaño 1024 (excepto que es mucho más rápida, más poderosa y utiliza muchos menos parámetros).

También rellenaron a la izquierda las secuencias de entrada con una cantidad de ceros igual a la tasa de dilatación antes de cada capa, para preservar la misma longitud de secuencia en toda la red.

A continuación se explica cómo implementar una WaveNet simplificada para abordar las mismas secuencias que antes.<sup>17</sup>

```
wavenet_model = tf.keras.Sequential()
wavenet_model.add(tf.keras.layers.Input(shape=[None, 5])) para tasa en (1, 2,
4, 8) * 2:
    wavenet_model.add(tf .keras.capas.Conv1D(
        filtros=32, kernel_size=2, padding="causal", activación="relu",
        dilation_rate=rate))
wavenet_model.add(tf.keras.layers.Conv1D(filters=14, kernel_size=1))
```

Este modelo secuencial comienza con una capa de entrada explícita; esto es más sencillo que intentar establecer `input_shape` solo en la primera capa. Luego continúa con una capa convolucional 1D usando un relleno "causal", que es como el "mismo" relleno excepto que los ceros se agregan solo al inicio de la secuencia de entrada, en lugar de en ambos lados. Esto garantiza que la capa convolucional no mire hacia el futuro al hacer predicciones. Luego agregamos pares similares de capas usando tasas de dilatación crecientes: 1, 2, 4, 8, y nuevamente 1, 2, 4, 8. Finalmente, agregamos la capa de salida: una capa convolucional con 14 filtros de tamaño 1 y sin ninguna función de activación. Como vimos anteriormente, una capa convolucional de este tipo equivale a una capa densa con 14 unidades.

Gracias al relleno causal, cada capa convolucional genera una secuencia de la misma longitud que su secuencia de entrada, por lo que los objetivos que utilizamos durante el entrenamiento pueden ser secuencias completas de 112 días: no es necesario recortarlas ni reducirlas.

Los modelos que hemos analizado en esta sección ofrecen un rendimiento similar para la tarea de pronóstico de número de pasajeros, pero pueden variar significativamente según la tarea y la cantidad de datos disponibles. En el artículo de WaveNet, los autores lograron un rendimiento de vanguardia en varias tareas de audio (de ahí el nombre de la arquitectura), incluidas tareas de conversión de texto a voz, produciendo voces increíblemente realistas en varios idiomas. También utilizaron el modelo para generar música, una muestra de audio a la vez. Esta hazaña es aún más impresionante cuando te das cuenta de que un solo segundo de audio puede contener decenas de miles de pasos de tiempo; ni siquiera los LSTM y GRU pueden manejar secuencias tan largas.

### ADVERTENCIA

Si evalúa nuestros mejores modelos de número de pasajeros en Chicago en el período de prueba, que comienza en 2020, descubrirá que su desempeño es mucho peor de lo esperado. ¿Por qué es eso? Bueno, fue entonces cuando comenzó la pandemia de Covid-19, que afectó mucho al transporte público. Como se mencionó anteriormente, estos modelos sólo funcionarán bien si los patrones que aprendieron del pasado continúan en el futuro. En cualquier caso, antes de implementar un modelo en producción, verifique que funcione bien con datos recientes. Y una vez que esté en producción, asegúrese de controlar su rendimiento con regularidad.

¡Con eso, ahora puedes abordar todo tipo de series temporales! En [el Capítulo 16](#), continuaremos explorando las RNN y veremos cómo pueden abordar también varias tareas de PNL.

## Ejercicios

1. ¿Se te ocurren algunas aplicaciones para un RNN de secuencia a secuencia? ¿Qué pasa con un RNN de secuencia a vector y un RNN de vector a secuencia?
2. ¿Cuántas dimensiones deben tener las entradas de una capa RNN? ¿Qué representa cada dimensión? ¿Qué pasa con sus resultados?
3. Si desea crear un RNN profundo de secuencia a secuencia, ¿qué capas de RNN deberían tener `return_sequences=True`? ¿Qué pasa con un RNN de secuencia a vector?
4. Suponga que tiene una serie temporal univariada diaria y desea pronosticar los próximos siete días. ¿Qué arquitectura RNN debería utilizar?
5. ¿Cuáles son las principales dificultades a la hora de formar RNN? ¿Cómo puedes manejarlos?
6. ¿Puedes esbozar la arquitectura de la celda LSTM?

7. ¿Por qué querrías utilizar capas convolucionales 1D en un RNN?
8. ¿Qué arquitectura de red neuronal podrías utilizar para clasificar? vídeos?
9. Entrene un modelo de clasificación para el conjunto de datos SketchRNN. disponible en conjuntos de datos de TensorFlow.
10. Descarga los **corales de Bach** conjunto de datos y descomprímalo. Es compuesto por 382 corales compuestos por Johann Sebastian Bach. Cada coral tiene una duración de 100 a 640 pasos de tiempo, y cada paso de tiempo contiene 4 números enteros, donde cada número entero corresponde al índice de una nota en un piano (excepto el valor 0, que significa que no se toca ninguna nota). Entrene un modelo (recurrente, convolucional o ambos) que pueda predecir el siguiente paso de tiempo (cuatro notas), dada una secuencia de pasos de tiempo de un coral. Luego use este modelo para generar música tipo Bach, una nota a la vez: puede hacer esto dándole al modelo el comienzo de un coral y pidiéndole que prediga el siguiente paso de tiempo, luego agregando estos pasos de tiempo a la secuencia de entrada y pedirle al modelo la siguiente nota, y así sucesivamente.

También asegúrese de consultar **el modelo Coconet de Google**, que se utilizó para un bonito doodle de Google sobre Bach.

Las soluciones a estos ejercicios están disponibles al final del cuaderno de este capítulo, en <https://homl.info/colab3>.

---

<sup>1</sup> Tenga en cuenta que muchos investigadores prefieren utilizar la función de activación tangente hiperbólica ( $\tanh$ ) en RNN en lugar de la función de activación ReLU. Por ejemplo, consulte [el artículo de 2013](#) de Vu Pham et al. "La deserción mejora las redes neuronales recurrentes para el reconocimiento de escritura a mano". Los RNN basados en ReLU también son posibles, como se muestra en [el artículo de 2015](#) de Quoc V. Le et al. "Una forma sencilla de inicializar redes recurrentes de unidades lineales rectificadas".

<sup>2</sup> Nal Kalchbrenner y Phil Blunsom, "Modelos de traducción continua recurrente", Actas de la Conferencia de 2013 sobre métodos empíricos en el procesamiento del lenguaje natural (2013): 1700–1709.

- 3** Los datos más recientes de la Autoridad de Tránsito de Chicago están disponibles en el [Portal de Datos de Chicago](#).
- 4** Existen otros enfoques más basados en principios para seleccionar buenos hiperparámetros, basados en el análisis de la función de autocorrelación (ACF) y la función de autocorrelación parcial (PACF), o minimizando las métricas AIC o BIC (introducidas en el [Capítulo 9](#)) para penalizar los modelos que usan demasiados parámetros y reducir el riesgo de sobreajuste de los datos, pero La búsqueda en cuadrícula es un buen lugar para comenzar. Para obtener más detalles sobre el enfoque ACF-PACF, consulte esta muy buena [publicación de Jason Brownlee](#).
- 5** Tenga en cuenta que el período de validación comienza el 1 de enero de 2019, por lo que la primera predicción es para el 26 de febrero de 2019, ocho semanas después. Cuando evaluamos los modelos de referencia utilizamos predicciones a partir del 1 de marzo, pero esto debería ser lo suficientemente cercano.
- 6** Siéntete libre de jugar con este modelo. Por ejemplo, puedes intentar pronosticando el número de pasajeros de autobús y tren para los próximos 14 días. Deberá modificar los objetivos para incluir ambos y hacer que su modelo genere 28 pronósticos en lugar de 14.
- 7** César Laurent et al., “Redes neuronales recurrentes normalizadas por lotes”, *Actas de la Conferencia Internacional IEEE sobre Acústica, Habla y Procesamiento de Señales* (2016): 2657–2661.
- 8** Jimmy Lei Ba et al., “Normalización de capas”, preimpresión de arXiv arXiv:1607.06450 (2016).
- 9** Habría sido más sencillo heredar de SimpleRNNCell para no tener que crear una SimpleRNNCell interna o manejar los atributos state\_size y output\_size, pero el objetivo aquí era mostrar cómo crear una celda personalizada desde cero.
- 10** Un personaje de las películas animadas Buscando a Nemo y Buscando a Dory que tiene pérdida de memoria a corto plazo.
- 11** Sepp Hochreiter y Jürgen Schmidhuber, “Long Short-Term Memory”, *Neural Computation* 9, no. 8 (1997): 1735-1780.
- 12** Haşim Sak et al., “Arquitecturas de redes neuronales recurrentes basadas en memoria a corto plazo para el reconocimiento de voz con vocabulario amplio”, preimpresión de arXiv arXiv:1402.1128 (2014).
- 13** Wojciech Zaremba et al., “Recurrent Neural Network Regularization”, preimpresión de arXiv arXiv:1409.2329 (2014).
- 14** Kyunghyun Cho et al., “Aprendizaje de representaciones de frases utilizando RNN Codificador-Decodificador para traducción automática estadística”, *Actas del*

Conferencia de 2014 sobre métodos empíricos en el procesamiento del lenguaje natural (2014): 1724–1734.

**15** Véase Klaus Greff et al., “[LSTM: A Search Space Odyssey](#)”, IEEE Transacciones sobre redes neuronales y sistemas de aprendizaje 28, no. 10 (2017): 2222–2232. Este artículo parece mostrar que todas las variantes de LSTM funcionan aproximadamente igual.

**16** Aaron van den Oord et al., “[WaveNet: A Generative Model for Raw Audio](#)”, Preimpresión de arXiv arXiv:1609.03499 (2016).

**17** El WaveNet completo utiliza algunos trucos más, como omitir conexiones como en un ResNet y unidades de activación cerradas similares a las que se encuentran en una celda GRU. Consulte el cuaderno de este capítulo para obtener más detalles.

# Capítulo 16. Procesamiento del lenguaje natural con RNN y atención

---

Cuando Alan Turing imaginó su famoso [test de Turing](#) En 1950, <sup>1</sup>propuso una forma de evaluar la capacidad de una máquina para igualar la inteligencia humana. Podría haber evaluado muchas cosas, como la capacidad de reconocer gatos en imágenes, jugar al ajedrez, componer música o escapar de un laberinto, pero, curiosamente, eligió una tarea lingüística. Más concretamente, ideó un chatbot capaz de engañar a su interlocutor haciéndole creer que era humano. Esta prueba tiene sus debilidades: un conjunto de reglas codificadas pueden engañar a humanos desprevenidos o ingenuos (por ejemplo, la máquina podría dar respuestas vagas predefinidas en respuesta a algunas palabras clave, podría fingir que está bromeando o que está borracha para pasar por alto su punto más extraño). respuestas, o podría escapar de preguntas difíciles respondiéndolas con sus propias preguntas), y muchos aspectos de la inteligencia humana se ignoran por completo (por ejemplo, la capacidad de interpretar la comunicación no verbal, como las expresiones faciales, o aprender una tarea manual). Pero la prueba resalta el hecho de que dominar el lenguaje es posiblemente la mayor capacidad cognitiva del Homo sapiens .

¿Podemos construir una máquina que pueda dominar el lenguaje hablado y escrito? Este es el objetivo final de la investigación de PNL, pero es demasiado amplio, por lo que en la práctica los investigadores se centran en tareas más específicas, como clasificación de texto, traducción, resumen, respuesta a preguntas y muchas más.

Un enfoque común para las tareas de lenguaje natural es utilizar redes neuronales recurrentes. Por lo tanto, continuaremos explorando los RNN (presentados en [el Capítulo 15](#)), comenzando con un carácter RNN, o char-RNN, entrenado para predecir el siguiente carácter en una oración. Esto nos permitirá generar algún texto original. Primero usaremos un RNN sin estado (que aprende en porciones aleatorias de texto en cada iteración, sin ninguna información sobre el resto del texto), luego construiremos un RNN con estado (que preserva el estado oculto entre iteraciones de entrenamiento y continúa leyendo donde lo dejó, permitiéndole aprender patrones más largos).

A continuación, crearemos un RNN para realizar un análisis de sentimientos (por ejemplo, leer reseñas de películas y extraer los sentimientos del evaluador sobre la película), esta vez tratando las oraciones como secuencias de palabras, en lugar de personajes. Luego, mostraremos cómo se pueden usar los RNN para construir una arquitectura codificador-decodificador capaz de realizar traducción automática neuronal (NMT), traduciendo del inglés al español.

En la segunda parte de este capítulo, exploraremos los mecanismos de atención. Como sugiere su nombre, estos son componentes de redes neuronales que aprenden a seleccionar la parte de las entradas en las que debe centrarse el resto del modelo en cada paso de tiempo. Primero, aumentaremos el rendimiento de una arquitectura de codificador-decodificador basada en RNN utilizando la atención. A continuación, eliminaremos los RNN por completo y utilizaremos una arquitectura de solo atención muy exitosa, llamada transformador, para construir un modelo de traducción. Luego discutiremos algunos de los avances más importantes en PNL en los últimos años, incluidos modelos de lenguaje increíblemente poderosos como GPT y BERT, ambos basados en transformadores. Por último, te mostraré cómo empezar con la excelente biblioteca Transformers de Hugging Face.

Comencemos con un modelo simple y divertido que puede escribir como Shakespeare (más o menos).

## Generando texto de Shakespeare usando un carácter RNN

En una famosa [publicación de blog de 2015](#) Titulado "La eficacia irrazonable de las redes neuronales recurrentes", Andrej Karpathy mostró cómo entrenar un RNN para predecir el siguiente carácter en una oración. Este char-RNN se puede utilizar para generar texto novedoso, un carácter a la vez. Aquí hay una pequeña muestra del texto generado por un modelo char-RNN después de entrenarlo en todas las obras de Shakespeare:

PANDARUS:

Ay, creo que se le acercará y el día  
 Cuando se llegara a sentir un poco de estrés y nunca se pudiera alimentar,  
 ¿Y quién no es más que una cadena y sujetos de su muerte,  
 No debería dormir.

No es exactamente una obra maestra, pero sigue siendo impresionante que el modelo haya podido aprender palabras, gramática, puntuación adecuada y más, simplemente aprendiendo a predecir el siguiente carácter en una oración. Este es nuestro primer ejemplo de un modelo de lenguaje; Modelos de lenguaje similares (pero mucho más potentes), que se analizan más adelante en este capítulo, son el núcleo de la PNL moderna. En el resto de esta sección construiremos un char-RNN paso a paso, comenzando con la creación del conjunto de datos.

### Crear el conjunto de datos de entrenamiento

Primero, usando la práctica función `tf.keras.utils.get_file()` de Keras, descarguemos todas las obras de Shakespeare. Los datos se cargan desde [el proyecto char-rnn de Andrej Karpathy](#):

```
importar tensorflow como tf

shakespeare_url = "https://homl.info/shakespeare" # URL de acceso directo ruta de archivo =
tf.keras.utils.get_file("shakespeare.txt", shakespeare_url) con open(filepath) como f: shakespeare_text =
f.read()
```

Imprimamos las primeras líneas:

```
>>> imprimir(texto_shakespeare[:80])
Primer ciudadano:
Antes de continuar, escúchame hablar.
```

Todo:  
 Hablar hablar.

¡Parece Shakespeare, sí!

A continuación, usaremos una capa `tf.keras.layers.TextVectorization` (introducida en el [Capítulo 13](#)) para codificar este texto. Configuramos `split="character"` para obtener codificación a nivel de caracteres en lugar de la codificación predeterminada a nivel de palabra, y usamos `estandarizar="lower"` para convertir el texto a minúsculas (lo que simplificará la tarea):

```
text_vec_layer = tf.keras.layers.TextVectorization(split="carácter", estandarizar="inferior")

text_vec_layer.adapt([shakespeare_text]) codificado =
text_vec_layer([shakespeare_text])[0]
```

Cada carácter ahora se asigna a un número entero, comenzando en 2. La capa `TextVectorization` reservó el valor 0 para tokens de relleno y reservó 1 para caracteres desconocidos. No necesitaremos ninguno de estos tokens por ahora, así que restemos 2 de los ID de los personajes y calculemos el número de caracteres distintos y el número total de caracteres:

```
codificado -= 2 # soltar tokens 0 (pad) y 1 (desconocido), que no usaremos n_tokens =
text_vec_layer.vocabulary_size() - 2 # número de caracteres distintos = 39 dataset_size = len(codificado) # número total
de caracteres = 1.115.394
```

A continuación, tal como lo hicimos en el Capítulo 15, podemos convertir esta secuencia muy larga en un conjunto de datos de ventanas que luego podemos usar para entrenar un RNN de secuencia a secuencia. Los objetivos serán similares a los insumos, pero desplazados un paso en el tiempo hacia el "futuro". Por ejemplo, una muestra en el conjunto de datos puede ser una secuencia de ID de caracteres que representan el texto "ser o no b" (sin la "e" final), y el objetivo correspondiente: una secuencia de ID de caracteres que representa el texto "o ser o no ser" (con la "e" final, pero sin la "t" inicial). Escribamos una pequeña función de utilidad para convertir una secuencia larga de ID de caracteres en un conjunto de datos de pares de ventanas de entrada/destino:

```
def to_dataset(secuencia, longitud, shuffle=False, semilla=Ninguno, tamaño_lote=32):
    ds = tf.data.Dataset.from_tensor_slices(secuencia) ds =
    ds.window(length + 1, shift=1, drop_remainder=True) ds = ds.flat_map(lambda
    window_ds: window_ds.batch(length + 1)) if shuffle :
        ds = ds.shuffle(buffer_size=100_000, semilla=semilla)
    ds = ds.batch(batch_size) return
    ds.map( ventana lambda: (ventana[:, :-1], ventana[:, 1:])).prefetch(1)
```

Esta función comienza de manera muy similar a la función de utilidad personalizada `to_windows()` que creamos en el Capítulo 15:

- Toma una secuencia como entrada (es decir, el texto codificado) y crea un conjunto de datos que contiene todas las ventanas de la longitud deseada.
- Aumenta la longitud en uno, ya que necesitamos el siguiente carácter para el objetivo.
- Luego, mezcla las ventanas (opcionalmente), las agrupa, las divide en pares de entrada/salida y activa la captación previa.

La Figura 16-1 resume los pasos de preparación del conjunto de datos: muestra ventanas de longitud 11 y un tamaño de lote de 3. El índice de inicio de cada ventana se indica al lado.

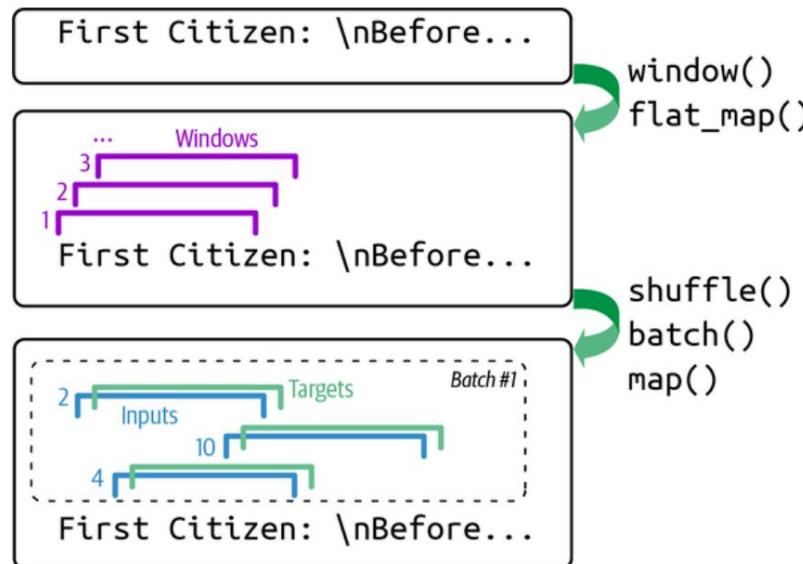


Figura 16-1. Preparar un conjunto de datos de ventanas mezcladas

Ahora estamos listos para crear el conjunto de entrenamiento, el conjunto de validación y el conjunto de prueba. Usaremos aproximadamente el 90% del texto para capacitación, 5% para validación y 5% para pruebas:

```
longitud = 100
tf.random.set_seed(42)
train_set = to_dataset(codificado[:1_000_000], longitud=longitud, shuffle=True,
semilla=42)
```

```
conjunto_válido = to_dataset(codificado[1_000_000:1_060_000], longitud=longitud) test_set =
to_dataset(codificado[1_060_000:], longitud=longitud)
```

## CONSEJO

Establecemos la longitud de la ventana en 100, pero puedes intentar ajustarla: es más fácil y rápido entrenar RNN en secuencias de entrada más cortas, pero el RNN no podrá aprender ningún patrón de mayor longitud, así que no lo hagas demasiado pequeño. .

¡Eso es todo! Preparar el conjunto de datos fue la parte más difícil. Ahora creemos el modelo.

## Construcción y entrenamiento del modelo Char-RNN

Dado que nuestro conjunto de datos es razonablemente grande y el lenguaje de modelado es una tarea bastante difícil, necesitamos más que un simple RNN con algunas neuronas recurrentes. Construyamos y entrenemos un modelo con una capa GRU compuesta por 128 unidades (puede intentar ajustar la cantidad de capas y unidades más adelante, si es necesario):

```
modelo = tf.keras.Sequential([
    tf.keras.layers.Embedding(input_dim=n_tokens, output_dim=16), tf.keras.layers.GRU(128,
    return_sequences=True), tf.keras.layers.Dense(n_tokens,
    activation="softmax")
])
model.compile(loss="sparse_categorical_crossentropy", optimizer="nadam",
    metrics=["precisión"])
model_ckpt = tf.keras.callbacks.ModelCheckpoint(
    "my_shakespeare_model", monitor="val_accuracy", save_best_only=True)
historia = model.fit(train_set, validation_data=valid_set, épocas=10,
    devoluciones de llamada = [model_ckpt])
```

Repasemos este código:

- Usamos una capa de incrustación como primera capa para codificar las ID de los caracteres (las incrustaciones se introdujeron en el [Capítulo 13](#)). El número de dimensiones de entrada de la capa de incrustación es el número de ID de caracteres distintos, y el número de dimensiones de salida es un hiperparámetro que puede ajustar; lo configuraremos en 16 por ahora. Mientras que las entradas de la capa de incrustación serán tensores de forma 2D [tamaño de lote, longitud de ventana], la salida de la capa de incrustación será un tensor de forma 3D [tamaño de lote, longitud de ventana, tamaño de incrustación].
- Usamos una capa Densa para la capa de salida: debe tener 39 unidades (`n_tokens`) porque hay 39 caracteres distintos en el texto y queremos generar una probabilidad para cada carácter posible (en cada paso de tiempo). Las 39 probabilidades de salida deben sumar 1 en cada paso de tiempo, por lo que aplicamos la función de activación softmax a las salidas de la capa Densa.
- Por último, compilamos este modelo usando la pérdida "sparse\_categorical\_crossentropy" y un optimizador Nadam, y entrenamos el modelo para varias épocas, usando una devolución de llamada <sup>3</sup>[ModelCheckpoint](#) para guardar el mejor modelo (en términos de precisión de validación) a medida que avanza el entrenamiento.

## CONSEJO

Si ejecuta este código en Colab con una GPU activada, la capacitación debería durar aproximadamente de una a dos horas. Puede reducir el número de épocas si no quiere esperar tanto, pero, por supuesto, la precisión del modelo probablemente será menor. Si la sesión de Colab se agota, asegúrese de volver a conectarse rápidamente; de lo contrario, se destruirá el tiempo de ejecución de Colab.

Este modelo no maneja el preprocesamiento de texto, así que vamos a incluirlo en un modelo final que contenga la capa `tf.keras.layers.TextVectorization` como primera capa, más una capa `tf.keras.layers.Lambda` para restar 2 de los ID de los caracteres, ya que No estamos usando el relleno ni los tokens desconocidos por ahora:

```
shakespeare_model = tf.keras.Sequential([
    text_vec_layer,
    tf.keras.layers.lambda (lambda x: x - 2), # no <mad> o tokens
    modelo
])
```

Y ahora usémoslo para predecir el siguiente carácter en una oración:

```
>>> y_proba = shakespeare_model.predict([" Ser o no b"])[0, -1] >>> y_pred = tf.argmax(y_proba) # elige el ID de
personaje más probable >>> text_vec_layer.get_vocabulary ()[y_pred + 2] 'e'
```

Genial, el modelo predijo correctamente el siguiente personaje. ¡Ahora usemos este modelo para fingir que somos Shakespeare!

## Generando texto falso de Shakespeare

Para generar texto nuevo usando el modelo char-RNN, podríamos alimentarlo con algo de texto, hacer que el modelo prediga la siguiente letra más probable, agregarlo al final del texto y luego darle el texto extendido al modelo para que adivine la siguiente letra., etcétera. A esto se le llama decodificación codiciosa. Pero en la práctica esto a menudo lleva a que las mismas palabras se repitan una y otra vez. En su lugar, podemos muestrear el siguiente carácter al azar, con una probabilidad igual a la probabilidad estimada, usando la función `tf.random.categorical()` de TensorFlow. Esto generará un texto más diverso e interesante. La función `categorical()` muestra índices de clase aleatorios, dadas las probabilidades logarítmicas de clase (logits). Por ejemplo:

```
>>> log_probas = tf.math.log([[0.5, 0.4, 0.1]]) # probas = 50%, 40% y 10% >>> tf.random.set_seed(42) >>>
tf.random .categorical(log_probas,
num_samples=8) # extrae 8 muestras <tf.Tensor: shape=(1, 8), dtype=int64, numpy=array([[0, 1, 0, 2, 1, 0, 0, 1]])>
```

Para tener más control sobre la diversidad del texto generado, podemos dividir los logits por un número llamado `temperatura`, que podemos modificar como queramos. Una temperatura cercana a cero favorece a los personajes de alta probabilidad, mientras que una temperatura alta da a todos los personajes la misma probabilidad. Por lo general, se prefieren temperaturas más bajas cuando se genera texto bastante rígido y preciso, como ecuaciones matemáticas, mientras que se prefieren temperaturas más altas cuando se genera texto más diverso y creativo. La siguiente función auxiliar personalizada `next_char()` utiliza este enfoque para seleccionar el siguiente carácter para agregar al texto de entrada:

```
def next_char(texto, temperatura=1): y_proba =
    shakespeare_model.predict([texto])[0, -1:] rescaled_logits = tf.math.log(y_proba) /
    temperatura char_id = tf.random.categorical(rescaled_logits, num_samples =1)[0, 0]
    devolver text_vec_layer.get_vocabulary()[char_id + 2]
```

A continuación, podemos escribir otra pequeña función auxiliar que llamará repetidamente a `next_char()` para obtener el siguiente carácter y agregarlo al texto dado:

```
def extender_text(texto, n_chars=50, temperatura=1):
    para _ en rango(n_chars):
```

```

    texto += next_char(texto, temperatura)
    devolver texto

```

¡Ahora estamos listos para generar algo de texto! Probemos con diferentes valores de temperatura:

```

>>> tf.random.set_seed(42) >>>
print(extend_text("Ser o no ser", temperatura=0.01))
Ser o no ser duque
ya que es una muerte extraña propiamente
dicha, y el
>>> print(extend_text("Ser o no ser", temperatura=1))
¿Ser o no contemplar?

segundo empujón:
gremio, señor todo, un hermanastro, >>>
print(extend_text("Ser o no ser", temperatura=100))
Ser o no ser bef ,mt'&o3fpadm! wh!nse?bws3est-
vgerdjh?cy-ewzqnq

```

Shakespeare parece estar sufriendo una ola de calor. Para generar texto más convincente, una técnica común es tomar muestras solo de los k caracteres superiores, o solo del conjunto más pequeño de caracteres superiores cuya probabilidad total excede algún umbral (esto se llama muestreo de núcleo). Alternativamente, puede intentar usar la búsqueda por haz, que discutiremos más adelante en este capítulo, o usar más capas GRU y más neuronas por capa, entrenar durante más tiempo y agregar algo de regularización si es necesario. También tenga en cuenta que el modelo actualmente es incapaz de aprender patrones de más de 100 caracteres. Podría intentar agrandar esta ventana, pero también hará que el entrenamiento sea más difícil, e incluso las células LSTM y GRU no pueden manejar secuencias muy largas. Un enfoque alternativo es utilizar un RNN con estado.

## RNN con estado

Hasta ahora, solo hemos usado RNN sin estado: en cada iteración de entrenamiento, el modelo comienza con un estado oculto lleno de ceros, luego actualiza este estado en cada paso de tiempo y, después del último paso de tiempo, lo descarta porque no lo es. necesita más. ¿Qué pasaría si le ordenáramos al RNN que preservara este estado final después de procesar un lote de entrenamiento y lo usara como estado inicial para el siguiente lote de entrenamiento? De esta manera, el modelo podría aprender patrones a largo plazo a pesar de que solo se propaga hacia atrás a través de secuencias cortas. Esto se llama RNN con estado. Repasemos cómo construir uno.

Primero, tenga en cuenta que un RNN con estado solo tiene sentido si cada secuencia de entrada en un lote comienza exactamente donde terminó la secuencia correspondiente en el lote anterior. Entonces, lo primero que debemos hacer para construir un RNN con estado es usar secuencias de entrada secuenciales y no superpuestas (en lugar de las secuencias aleatorias y superpuestas que usamos para entrenar RNN sin estado). Por lo tanto, al crear `tf.data.Dataset`, debemos usar `shift=length` (en lugar de `shift=1`) al llamar al método `window()`. Además, no debemos llamar al método `shuffle()`.

Desafortunadamente, el procesamiento por lotes es mucho más difícil cuando se prepara un conjunto de datos para un RNN con estado que para un RNN sin estado. De hecho, si llamaramos al lote (32), entonces se colocarían 32 ventanas consecutivas en el mismo lote y el siguiente lote no continuaría con cada una de estas ventanas donde lo dejó. El primer lote contendría las ventanas 1 a 32 y el segundo lote contendría las ventanas 33 a 64, por lo que si considera, digamos, la primera ventana de cada lote (es decir, las ventanas 1 y 33), puede ver que no son consecutivas. . La solución más sencilla a este problema es utilizar simplemente un tamaño de lote de 1. La siguiente función de utilidad personalizada `to_dataset_for_stateful_rnn()` utiliza esta estrategia para preparar un conjunto de datos para un RNN con estado:

```

def to_dataset_for_stateful_rnn(secuencia, longitud):
    ds = tf.data.Dataset.from_tensor_slices(secuencia) ds = ds.window(longitud
    + 1, shift=longitud, drop_remainder=True)

```

```

ds = ds.flat_map( ventana lambda: ventana.batch(longitud + 1).batch(1) return ds.map( ventana lambda :
(ventana[:, :-1], ventana[:, 1:])) .precarga(1)

stateful_train_set = to_dataset_for_stateful_mn(codificado[1_000_000], longitud) stateful_valid_set =
to_dataset_for_stateful_mn(codificado[1_000_000:1_060_000],
longitud)

stateful_test_set = to_dataset_for_stateful_rnn(codificado[1_060_000:], longitud)

```

La Figura 16-2 resume los pasos principales de esta función.

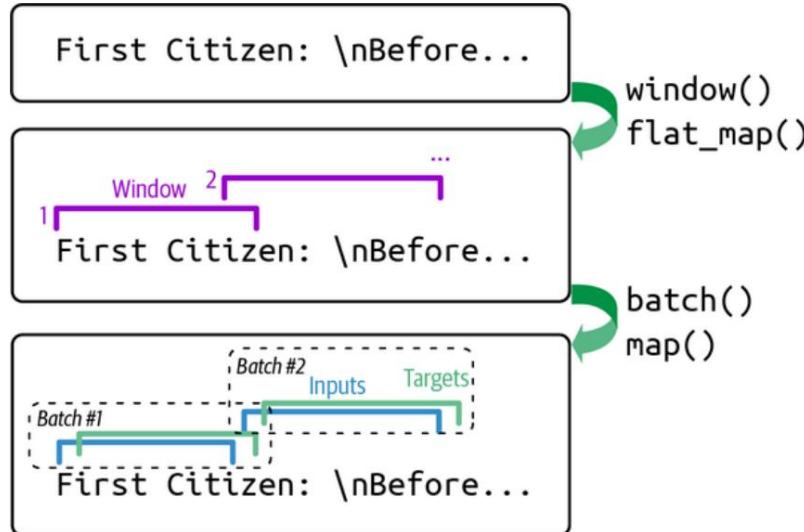


Figura 16-2. Preparación de un conjunto de datos de fragmentos de secuencia consecutivos para un RNN con estado

El procesamiento por lotes es más difícil, pero no imposible. Por ejemplo, podríamos dividir el texto de Shakespeare en 32 textos de igual longitud, crear un conjunto de datos de secuencias de entrada consecutivas para cada uno de ellos y, finalmente, usar `tf.data.Dataset.zip(datasets).map(lambda *windows: tf.stack(windows))` para crear lotes consecutivos adecuados, donde la  $n$  secuencia de entrada en un lote comienza exactamente donde terminó la  $n$  secuencia de entrada en el lote anterior (consulte el cuaderno para obtener el código completo).

Ahora, creamos el RNN con estado. Necesitamos establecer el argumento `con_estado` en Verdadero al crear cada capa recurrente, y porque el RNN con estado necesita conocer el tamaño del lote (ya que preservará un estado para cada secuencia de entrada en el lote). Por lo tanto, debemos establecer el argumento `batch_input_shape` en la primera capa. Tenga en cuenta que podemos dejar la segunda dimensión sin especificar, ya que las secuencias de entrada podrían tener cualquier longitud:

```

modelo = tf.keras.Sequential([
    tf.keras.layers.Embedding(input_dim=n_tokens, output_dim=16,
                             batch_input_shape=[1, Ninguno]),
    tf.keras.layers.GRU(128, return_sequences=True, stateful=True), tf.keras.layers.Dense(n_tokens,
activación="softmax")
])

```

Al final de cada época, debemos restablecer los estados antes de volver al principio del texto.

Para esto, podemos usar una pequeña devolución de llamada personalizada de Keras:

```

clase ResetStatesCallback(tf.keras.callbacks.Callback): def on_epoch_begin(self, epoch,
logs): self.model.reset_states()

```

Y ahora podemos compilar el modelo y entrenarlo usando nuestra devolución de llamada:

```
model.compile(loss="sparse_categorical_crossentropy", optimizador="nadam",
méticas=["exactitud"])
historial = model.fit(stateful_train_set, validation_data=stateful_valid_set,
épocas=10, devoluciones de llamada=[ResetStatesCallback(), model_ckpt])
```

## CONSEJO

Una vez entrenado este modelo, solo será posible utilizarlo para hacer predicciones para lotes del mismo tamaño que se utilizaron durante el entrenamiento. Para evitar esta restricción, cree un modelo sin estado idéntico y copie los pesos del modelo con estado en este modelo.

Curiosamente, aunque un modelo char-RNN solo está entrenado para predecir el siguiente personaje, esta tarea aparentemente simple en realidad requiere que también aprenda algunas tareas de nivel superior. Por ejemplo, para encontrar el siguiente carácter después de "Gran película, de verdad", es útil comprender que la oración es positiva, por lo que es más probable que lo que sigue sea la letra "I" (de "amado") en lugar de "h". (por "odiado"). De hecho, un [artículo de 2017](#) de Alec Radford y otros investigadores de OpenAI describe cómo los autores entrenaron un gran modelo similar a char-RNN en un gran conjunto de datos y descubrieron que una de las neuronas actuaba como un excelente clasificador de análisis de sentimientos: aunque el modelo se entrenó sin etiquetas, la neurona del sentimiento, como la llamarón, alcanzó un rendimiento de vanguardia en los puntos de referencia del análisis del sentimiento. Esto presagió y motivó el entrenamiento previo no supervisado en PNL.

Pero antes de explorar el entrenamiento previo no supervisado, dirigamos nuestra atención a los modelos a nivel de palabra y cómo usarlos de manera supervisada para el análisis de sentimientos. En el proceso, aprenderá a manejar secuencias de longitudes variables mediante enmascaramiento.

### Análisis de sentimientos Generar

El texto puede ser divertido e instructivo, pero en proyectos de la vida real, una de las aplicaciones más comunes de la PNL es la clasificación de texto, especialmente el análisis de sentimientos. Si la clasificación de imágenes en el conjunto de datos MNIST es "¡Hola mundo!" de visión por computadora, entonces el análisis de sentimientos en el conjunto de datos de reseñas de IMDb es "¡Hola mundo!" del procesamiento del lenguaje natural. El conjunto de datos de IMDb consta de 50.000 reseñas de películas en inglés (25.000 para formación, 25.000 para pruebas) extraídas de la famosa [Internet Movie Database](#), junto con un objetivo binario simple para cada revisión que indica si es negativa (0) o positiva (1). Al igual que MNIST, el conjunto de datos de reseñas de IMDb es popular por buenas razones: es lo suficientemente simple como para abordarlo en una computadora portátil en un período de tiempo razonable, pero lo suficientemente desafiante como para ser divertido y gratificante.

Carguemos el conjunto de datos de IMDb usando la biblioteca TensorFlow Datasets (presentada en [el Capítulo 13](#)). Usaremos el primer 90% del conjunto de entrenamiento para entrenamiento y el 10% restante para validación:

```
importar tensorflow_datasets como tfds

raw_train_set, raw_valid_set, raw_test_set = tfds.load( nombre="imdb_reviews",
split=[ "entrenar[:90%]", "entrenar[90%:]", "prueba"], as_supervised=True

) tf.random.set_seed(42)
train_set = raw_train_set.shuffle(5000, semilla=42).batch(32).prefetch(1) valid_set =
raw_valid_set.batch(32).prefetch(1) test_set =
raw_test_set.batch(32).captura previa(1)
```

## CONSEJO

Keras también incluye una función para cargar el conjunto de datos de IMDb, si lo prefiere: `tf.keras.datasets.imdb.load_data()`. Las reseñas ya están preprocesadas como secuencias de identificaciones de palabras.

Revisemos algunas reseñas:

```
>>> para revisión, etiqueta en raw_train_set.take(4):
...     print(review.numpy().decode("utf-8")) print("Etiqueta:",
...     label.numpy())
...
Esta fue una película absolutamente terrible. No se deje engañar por Christopher [...]
Etiqueta: 0
Se sabe que me quedo dormido durante las películas, pero esto generalmente se debe [...]
Etiqueta: 0
Mann fotografía las Montañas Rocosas de Alberta de manera magnífica y [...]
Etiqueta: 0
Este es el tipo de película para una tarde de domingo nevada cuando el resto de la [...]
Etiqueta: 1
```

Algunas reseñas son fáciles de clasificar. Por ejemplo, la primera reseña incluye las palabras "película terrible" en la primera oración. Pero en muchos casos las cosas no son tan sencillas. Por ejemplo, la tercera reseña comienza positivamente, aunque al final sea una reseña negativa (etiqueta 0).

Para construir un modelo para esta tarea, necesitamos preprocessar el texto, pero esta vez lo cortaremos en palabras en lugar de caracteres. Para ello podemos volver a utilizar la capa `tf.keras.layers.TextVectorization`.

Tenga en cuenta que utiliza espacios para identificar los límites de las palabras, lo que no funcionará bien en algunos idiomas. Por ejemplo, la escritura china no usa espacios entre palabras, la vietnamita usa espacios incluso dentro de las palabras y el alemán a menudo une varias palabras juntas, sin espacios. Incluso en inglés, los espacios no siempre son la mejor manera de tokenizar el texto: piense en "San Francisco" o "#ILoveDeepLearning".

Afortunadamente, existen soluciones para abordar estos problemas. En un [artículo de 2016](#),<sup>5</sup> Rico Sennrich et al. de la Universidad de Edimburgo exploró varios métodos para tokenizar y detokenizar texto a nivel de subpalabra.

De esta manera, incluso si su modelo encuentra una palabra rara que nunca antes había visto, aún puede adivinar razonablemente lo que significa. Por ejemplo, incluso si el modelo nunca vio la palabra "más inteligente" durante el entrenamiento, si aprendió la palabra "inteligente" y también aprendió que el sufijo "est" significa "el más", puede inferir el significado de "más inteligente". Una de las técnicas que evaluaron los autores es la codificación de pares de bytes (BPE). BPE funciona dividiendo todo el conjunto de entrenamiento en caracteres individuales (incluidos espacios) y luego fusionando repetidamente los pares adyacentes más frecuentes hasta que el vocabulario alcance el tamaño deseado.

Un [artículo de 2018](#) por Taku Kudo en Google mejoró aún más la tokenización de subpalabras, eliminando a menudo la necesidad de un preprocessamiento específico del idioma antes de la tokenización. Además, el artículo propone una técnica de regularización novedosa llamada regularización de subpalabras, que mejora la precisión y la solidez al introducir cierta aleatoriedad en la tokenización durante el entrenamiento: por ejemplo, "Nueva Inglaterra" puede tokenizarse como "Nueva" + "Inglaterra" o "Nueva". + "Eng" + "land", o simplemente "Nueva Inglaterra" (solo una ficha).

[Frase](#) de Google El proyecto proporciona una implementación de código abierto, que se describe en un [documento](#) por Taku Kudo y John Richardson.<sup>7</sup>

El [texto de TensorFlow](#) La biblioteca también implementa varias estrategias de tokenización, incluido [WordPiece](#). (una variante de BPE) y, por último, pero no menos importante, la [biblioteca Tokenizers de Hugging Face](#) implementa una amplia gama de tokenizadores extremadamente rápidos.<sup>8</sup>

Sin embargo, para la tarea de IMDb en inglés, usar espacios para los límites de los tokens debería ser suficiente. Entonces, sigamos adelante con la creación de una capa `TextVectorization` y adaptémosla al conjunto de entrenamiento. Limitaremos el vocabulario a 1000 tokens, incluidas las 998 palabras más frecuentes más un token de relleno y un

token para palabras desconocidas, ya que es poco probable que palabras muy raras sean importantes para esta tarea, y limitar el tamaño del vocabulario reducirá la cantidad de parámetros que el modelo necesita aprender:

```
vocab_size = 1000
text_vec_layer = tf.keras.layers.TextVectorization(max_tokens=vocab_size)
text_vec_layer.adapt(train_set.map( revisiones lambda, etiquetas: revisiones))
```

Finalmente, podemos crear el modelo y entrenarlo:

```
embed_size = 128
tf.random.set_seed(42) model
= tf.keras.Sequential([ text_vec_layer,
    tf.keras.layers.Embedding(vocab_size, embed_size),
    tf.keras.layers.GRU(128), tf.keras.
        capas.Dense(1, activación="sigmoide")
])
model.compile(loss="binary_crossentropy", optimizador="nadam", metrics=["accuracy"])

historia = model.fit(conjunto_tren, datos_validación=conjunto_válido, épocas=2)
```

La primera capa es la capa TextVectorization que acabamos de preparar, seguida de una capa de Incrustación que convertirá los ID de palabras en incrustaciones. La matriz de incrustación debe tener una fila por token en el vocabulario (vocab\_size) y una columna por dimensión de incrustación (este ejemplo usa 128 dimensiones, pero este es un hiperparámetro que puede ajustar). A continuación usamos una capa GRU y una capa Densa con una sola neurona y la función de activación sigmoidea, ya que esta es una tarea de clasificación binaria: la salida del modelo será la probabilidad estimada de que la reseña exprese un sentimiento positivo con respecto a la película. Luego compilamos el modelo y lo ajustamos al conjunto de datos que preparamos anteriormente durante un par de épocas (o puedes entrenar durante más tiempo para obtener mejores resultados).

Lamentablemente, si ejecuta este código, generalmente encontrará que el modelo no aprende nada en absoluto: la precisión se mantiene cerca del 50%, no mejor que la probabilidad aleatoria. ¿Por qué es eso? Las revisiones tienen diferentes longitudes, por lo que cuando la capa TextVectorization las convierte en secuencias de ID de token, rellena las secuencias más cortas usando el token de relleno (con ID 0) para que sean tan largas como la secuencia más larga del lote. Como resultado, la mayoría de las secuencias terminan con muchos tokens de relleno, a menudo docenas o incluso cientos de ellos. Aunque estamos usando una capa GRU, que es mucho mejor que una capa SimpleRNN, su memoria a corto plazo aún no es excelente, por lo que cuando pasa por muchos tokens de relleno, ¡termina olvidando de qué se trataba la revisión! Una solución es alimentar el modelo con lotes de oraciones de igual longitud (lo que también acelera el entrenamiento). Otra solución es hacer que RNN ignore los tokens de relleno. Esto se puede hacer usando enmascaramiento.

**Enmascaramiento** Hacer que el modelo ignore los tokens de relleno es trivial usando Keras: simplemente agregue mask\_zero=True al crear la capa de incrustación. Esto significa que todas las capas posteriores ignorarán los tokens de relleno (cuyo ID es 0). ¡Eso es todo! Si vuelve a entrenar el modelo anterior durante algunas épocas, encontrará que la precisión de la validación alcanza rápidamente más del 80%.

La forma en que esto funciona es que la capa de incrustación crea un tensor de máscara igual a `tf.math.not_equal(inputs, 0)`: es un tensor booleano con la misma forma que las entradas y es igual a False en cualquier lugar donde se encuentren los ID de los tokens. 0, o Verdadero en caso contrario. Luego, el modelo propaga automáticamente este tensor de máscara a la siguiente capa. Si el método `call()` de esa capa tiene un argumento de máscara, automáticamente recibe la máscara. Esto permite que la capa ignore los pasos de tiempo apropiados. Cada capa puede manejar la máscara de manera diferente, pero en general simplemente ignoran los pasos de tiempo enmascarados.

(es decir, pasos de tiempo para los cuales la máscara es Falsa). Por ejemplo, cuando una capa recurrente encuentra un paso de tiempo enmascarado, simplemente copia la salida del paso de tiempo anterior.

A continuación, si el atributo `supports_masking` de la capa es Verdadero, la máscara se propaga automáticamente a la siguiente capa. Sigue propagándose de esta manera mientras las capas tengan `support_masking=True`.

Como ejemplo, el atributo `support_masking` de una capa recurrente es Verdadero cuando

`return_sequences=True`, pero es Falso cuando `return_sequences=False` ya que en este caso ya no es necesaria una máscara. Entonces, si tiene un modelo con varias capas recurrentes con `return_sequences=True`, seguidas de una capa recurrente con `return_sequences=False`, entonces la máscara se propagará automáticamente hasta la última capa recurrente: esa capa usará la máscara para ignorar los pasos enmascarados, pero no propagará más la máscara. De manera similar, si configura `mask_zero=True` al crear la capa de incrustación en el modelo de análisis de sentimiento que acabamos de crear, entonces la capa GRU recibirá y usará la máscara automáticamente, pero no la propagará más, ya que `return_sequences` no está configurado en True. .

#### CONSEJO

Algunas capas necesitan actualizar la máscara antes de propagarla a la siguiente capa: lo hacen implementando el método `Compute_mask()`, que toma dos argumentos: las entradas y la máscara anterior. Luego calcula la máscara actualizada y la devuelve. La implementación predeterminada de `Compute_mask()` simplemente devuelve la máscara anterior sin cambios.

Muchas capas de Keras admiten enmascaramiento: SimpleRNN, GRU, LSTM, Bidireccional, Dense, TimeDistributed, Add y algunas otras (todas en el paquete `tf.keras.layers`). Sin embargo, las capas convolucionales (incluida Conv1D) no admiten el enmascaramiento; de todos modos, no es obvio cómo lo harían.

Si la máscara se propaga hasta la salida, también se aplica a las pérdidas, por lo que los pasos de tiempo enmascarados no contribuirán a la pérdida (su pérdida será 0). Esto supone que el modelo genera secuencias, lo que no es el caso en nuestro modelo de análisis de sentimiento.

#### ADVERTENCIA

Las capas LSTM y GRU tienen una implementación optimizada para GPU, basada en la biblioteca cuDNN de Nvidia. Sin embargo, esta implementación solo admite el enmascaramiento si todos los tokens de relleno están al final de las secuencias. También requiere que utilice los valores predeterminados para varios hiperparámetros: `activación`, `activación_recurrente`, `abandono_recurrente`, `unroll`, `use_bias` y `reset_after`. Si ese no es el caso, entonces estas capas recurrirán a la implementación de GPU predeterminada (mucho más lenta).

Si desea implementar su propia capa personalizada con soporte de enmascaramiento, debe agregar un argumento de máscara al método `call()` y, obviamente, hacer que el método use la máscara. Además, si la máscara debe propagarse a las siguientes capas, entonces debes configurar `self.supports_masking=True` en el constructor. Si la máscara debe actualizarse antes de propagarse, debe implementar el método `Compute_mask()`.

Si su modelo no comienza con una capa de incrustación, puede usar la capa `tf.keras.layers.Masking` en su lugar: de forma predeterminada, establece la máscara en `tf.math.reduce_any(tf.math.not_equal(X, 0), eje = -1)`, lo que significa que los pasos de tiempo en los que la última dimensión está llena de ceros se enmascararán en capas posteriores.

El uso de capas de máscara y la propagación automática de máscaras funciona mejor para modelos simples. No siempre funcionará para modelos más complejos, como cuando necesitas mezclar capas Conv1D con capas recurrentes. En

En tales casos, deberá calcular explícitamente la máscara y pasársela a las capas apropiadas, utilizando la API funcional o la API de subclase. Por ejemplo, el siguiente modelo es equivalente al modelo anterior, excepto que se crea utilizando la API funcional y maneja el enmascaramiento manualmente. También añade un poco de abandono ya que el modelo anterior estaba ligeramente sobreajustado:

```
entradas = tf.keras.layers.Input(forma=[], dtype=tf.string) token_ids = text_vec_layer(entradas)
máscara = tf.math.not_equal(token_ids, 0)

Z = tf.keras.layers.Embedding(vocab_size, embed_size)(token_ids)
Z = tf.keras.layers.GRU(128, abandono=0.2)(Z, máscara=máscara) salidas =
tf.keras.layers.Dense(1, activación="sigmoide")(Z) modelo = tf.keras. Modelo(entradas=[entradas],
salidas=[salidas])
```

Una última forma de enmascarar es alimentar el modelo con tensores irregulares. En la práctica, todo lo que necesitas hacer es establecer `ragged=True` al crear la capa `TextVectorization`, para que las secuencias de entrada se representen como tensores irregulares:<sup>9</sup>

```
>>> text_vec_layer_ragged = tf.keras.layers.TextVectorization(
...     max_tokens=vocab_size, irregular=True)
...
>>> text_vec_layer_ragged.adapt(train_set.map(lambda reviews, etiquetas: reviews)) >>> text_vec_layer_ragged(["¡Gran
película!", "Este es el mejor papel de DiCaprio."]) <tf.RaggedTensor [[86, 18], [11, 7, 1, 116, 217]]>
```

Compare esta representación tensor irregular con la representación tensorial normal, que utiliza tokens de relleno:

```
>>> text_vec_layer(["¡Gran película!", "Este es el mejor papel de DiCaprio."]) <tf.Tensor: shape=(2, 5),
dtype=int64, numpy= array([[ 86, 18, [ 11, 7,
0, 0], 1, 116, 217]])>
```

Las capas recurrentes de Keras tienen soporte incorporado para tensores irregulares, por lo que no necesitas hacer nada más: simplemente usa esta capa `TextVectorization` en su modelo. No es necesario pasar `mask_zero=True` ni manejar máscaras explícitamente; todo está implementado para usted. ¡Eso es conveniente! Sin embargo, a principios de 2022, el soporte para tensores irregulares en Keras todavía es bastante reciente, por lo que existen algunas asperezas. Por ejemplo, actualmente no es posible utilizar tensores irregulares como objetivos cuando se ejecuta en la GPU (pero esto puede resolverse cuando lea estas líneas).

Cualquiera que sea el enfoque de enmascaramiento que prefiera, después de entrenar este modelo durante algunas épocas, será bastante bueno para juzgar si una revisión es positiva o no. Si usa la devolución de llamada `tf.keras.callbacks.TensorBoard()`, puede visualizar las incrustaciones en TensorBoard a medida que se aprenden: es fascinante ver palabras como "impresionante" y "asombroso" agruparse gradualmente en un lado de la incrustación. espacio, mientras que palabras como "horrible" y "terrible" se agrupan en el otro lado.

Algunas palabras no son tan positivas como cabría esperar (al menos en este modelo), como la palabra "bueno", presumiblemente porque muchas reseñas negativas contienen la frase "no es bueno".

## Reutilización de incrustaciones y modelos de lenguaje previamente entrenados

Es impresionante que el modelo sea capaz de aprender incrustaciones de palabras útiles basándose en sólo 25.000 reseñas de películas. ¡Imagínese lo buenas que serían las incorporaciones si tuviéramos miles de millones de revisiones sobre las que entrenar! Desafortunadamente, no lo hacemos, pero quizás podamos reutilizar incrustaciones de palabras entrenadas en algún otro corpus de texto (muy) grande (por ejemplo, reseñas de Amazon, disponibles en conjuntos de datos de TensorFlow), incluso si no está compuesto de reseñas de películas. Después de todo, la palabra "asombroso" generalmente tiene el mismo significado ya sea que la uses para hablar de películas o cualquier otra cosa. Además, quizás las incrustaciones serían útiles para el análisis de sentimientos.

incluso si fueron entrenados en otra tarea: dado que palabras como "impresionante" y "sorprendente" tienen un significado similar, probablemente se agruparán en el espacio de incrustación incluso para tareas como predecir la siguiente palabra en una oración. Si todas las palabras positivas y todas las palabras negativas forman grupos, esto será útil para el análisis de sentimientos. Entonces, en lugar de entrenar incrustaciones de palabras, podríamos simplemente descargar y usar incrustaciones previamente entrenadas, como las incrustaciones Word2vec de Google, Incrustaciones GloVe de Stanford, o las incrustaciones FastText de Facebook.

El uso de incrustaciones de palabras previamente entrenadas fue popular durante varios años, pero este enfoque tiene sus límites. En particular, una palabra tiene una única representación, sin importar el contexto. Por ejemplo, la palabra "correcto" está codificada de la misma manera en "izquierda y derecha" y "correcto e incorrecto", aunque significa dos cosas muy diferentes. Para abordar esta limitación, un [artículo de 2018](#) de Matthew<sup>10</sup> Peters presentó Incrustaciones de modelos de lenguaje (ELMo): son incrustaciones de palabras contextualizadas aprendidas de los estados internos de un modelo de lenguaje bidireccional profundo. En lugar de simplemente utilizar incrustaciones previamente entrenadas en su modelo, reutiliza parte de un modelo de lenguaje previamente entrenado.

Aproximadamente al mismo tiempo, el [documento Universal Language Model Fine-Tuning \(ULMFiT\)](#) de Jeremy Howard y Sebastian Ruder demostraron la eficacia del entrenamiento previo no supervisado para tareas de PNL: los autores entrenaron un modelo de lenguaje LSTM en un enorme corpus de texto utilizando el aprendizaje autosupervisado (es decir, generando las etiquetas automáticamente a partir de los datos), luego ajustaron en diversas tareas.

Su modelo superó al estado del arte en seis tareas de clasificación de texto por un amplio margen (reduciendo la tasa de error entre un 18% y un 24% en la mayoría de los casos). Además, los autores demostraron que un modelo previamente entrenado y ajustado con solo 100 ejemplos etiquetados podría lograr el mismo rendimiento que uno entrenado desde cero con 10.000 ejemplos. Antes del artículo ULMFiT, el uso de modelos previamente entrenados era sólo la norma en visión por computadora; En el contexto de la PNL, el entrenamiento previo se limitó a la incorporación de palabras. Este artículo marcó el comienzo de una nueva era en PNL: hoy en día, la norma es reutilizar modelos de lenguaje previamente entrenados.

Por ejemplo, creemos un clasificador basado en Universal Sentence Encoder, una arquitectura de modelo presentada en un [artículo de 2018](#), por un equipo de<sup>12</sup> investigadores de Google. Este modelo se basa en la arquitectura del transformador, que veremos más adelante en este capítulo. Convenientemente, el modelo está disponible en TensorFlow Hub:

```
importar
sistema operativo importar tensorflow_hub como centro

os.environ["TFHUB_CACHE_DIR"] = "my_tfhub_cache"
modelo =
tf.keras.Sequential([
    hub.KerasLayer("https://
        tflib.dev/google/universal-sentence-encoder/4",
        trainable=True, dtype=tf.string, input_shape=[]),
    tf.keras.layers.Dense(64, activación="relu"), tf.keras.layers.Dense(1,
        activación="sigmoide")
])
modelo.compile(loss="binary_crossentropy", optimizador="nadam", metrics=["accuracy"])
modelo.fit(train_set,
validation_data=valid_set, epochs=10)
```

#### CONSEJO

Este modelo es bastante grande (cerca de 1 GB), por lo que la descarga puede tardar un poco. De forma predeterminada, los módulos de TensorFlow Hub se guardan en un directorio temporal y se descargan una y otra vez cada vez que ejecuta su programa. Para evitarlo, debe configurar la variable de entorno TFHUB\_CACHE\_DIR en un directorio de su elección: los módulos se guardarán allí y solo se descargará una vez.

Tenga en cuenta que la última parte de la URL del módulo TensorFlow Hub especifica que queremos la versión 4 del modelo. Este control de versiones garantiza que si se lanza una nueva versión del módulo en TF Hub, no dañará nuestro modelo. Convenientemente, si ingresa esta URL en un navegador web, obtendrá la documentación para este módulo.

También tenga en cuenta que configuramos trainable=True al crear hub.KerasLayer. De esta manera, el codificador de oraciones universal previamente entrenado se ajusta durante el entrenamiento: algunos de sus pesos se ajustan mediante backprop. No todos los módulos de TensorFlow Hub se pueden ajustar con precisión, así que asegúrese de consultar la documentación de cada módulo previamente entrenado que le interese.

Después del entrenamiento, este modelo debería alcanzar una precisión de validación superior al 90%. Eso es realmente bueno: si intentas realizar la tarea tú mismo, probablemente lo harás solo un poco mejor, ya que muchas reseñas contienen comentarios tanto positivos como negativos. Clasificar estas reseñas ambiguas es como lanzar una moneda al aire.

Hasta ahora, hemos analizado la generación de texto utilizando un char-RNN y el análisis de sentimientos con modelos RNN a nivel de palabra (basados en incrustaciones entrenables) y utilizando un potente modelo de lenguaje previamente entrenado de TensorFlow Hub. En la siguiente sección, exploraremos otra tarea importante de la PNL: la traducción automática neuronal (NMT).

### Una red codificador-decodificador para la traducción automática neuronal

Comencemos con un [modelo NMT](#) simple. que traducirá oraciones del inglés al español (ver [Figura 16-3](#) ).

En resumen, la arquitectura es la siguiente: las oraciones en inglés se envían como entradas al codificador y el decodificador genera las traducciones al español. Tenga en cuenta que las traducciones al español también se utilizan como entradas para el decodificador durante el entrenamiento, pero se retroceden un paso. En otras palabras, durante el entrenamiento, al decodificador se le proporciona como entrada la palabra que debería haber emitido en el paso anterior, independientemente de lo que realmente haya generado. A esto se le llama fuerza docente, una técnica que acelera significativamente el entrenamiento y mejora el rendimiento del modelo. Para la primera palabra, el decodificador recibe el token de inicio de secuencia (SOS) y se espera que finalice la oración con un token de fin de secuencia (EOS).

Cada palabra está inicialmente representada por su ID (por ejemplo, 854 para la palabra "fútbol"). A continuación, una capa de Incrustación devuelve la palabra incrustación. Estas incrustaciones de palabras luego se envían al codificador y al decodificador.

En cada paso, el decodificador genera una puntuación para cada palabra del vocabulario de salida (es decir, español), luego la función de activación softmax convierte estas puntuaciones en probabilidades. Por ejemplo, en el primer paso la palabra "Yo" puede tener una probabilidad del 7%, "Yo" puede tener una probabilidad del 1%, y así sucesivamente. Se emite la palabra con mayor probabilidad. Esto es muy parecido a una tarea de clasificación normal y, de hecho, puedes entrenar el modelo utilizando la pérdida "sparse\_categorical\_crossentropy", muy parecido a lo que hicimos en el modelo char-RNN.

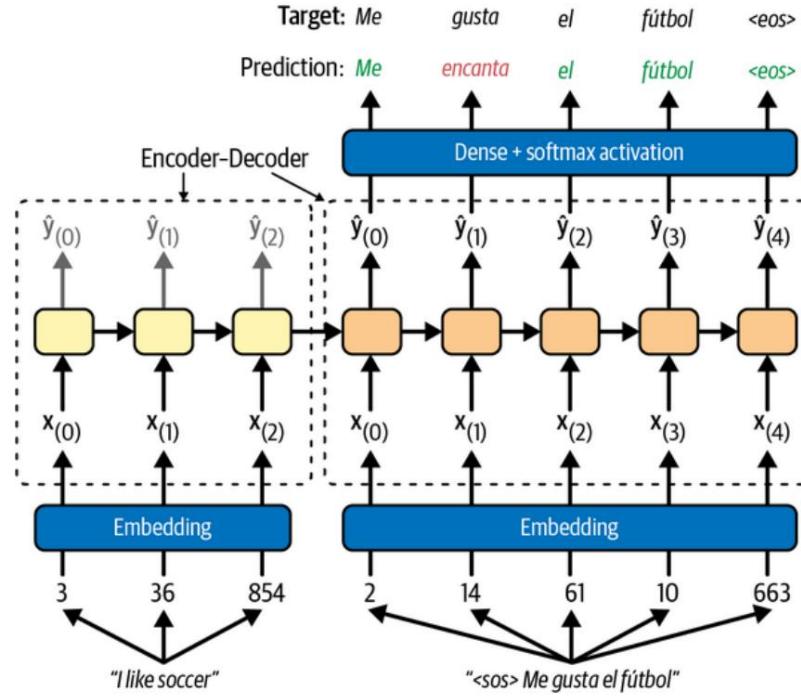


Figura 16-3. Un modelo de traducción automática simple

Tenga en cuenta que en el momento de la inferencia (después del entrenamiento), no tendrá la oración objetivo para alimentar al decodificador. En su lugar, debe introducirle la palabra que acaba de generar en el paso anterior, como se muestra en la Figura 16-4 (esto requerirá una búsqueda de incrustación que no se muestra en el diagrama).

CONSEJO

En un artículo de 2015, Samy Bengio et al. propuso cambiar gradualmente de alimentar al decodificador con el token de destino anterior a alimentarlo con el token de salida anterior durante el entrenamiento.

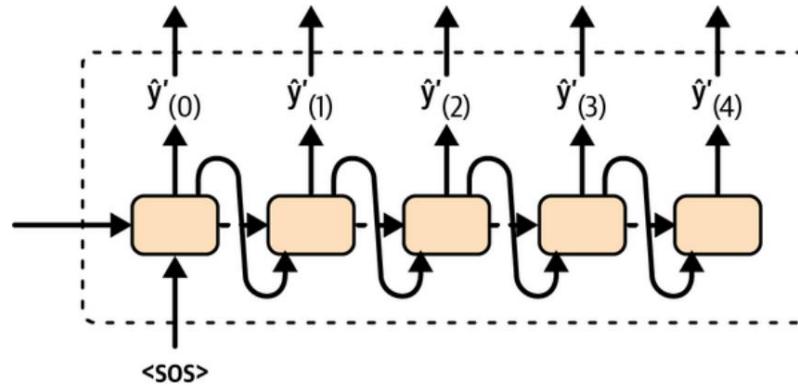


Figura 16-4. En el momento de la inferencia, el decodificador recibe como entrada la palabra que acaba de emitir en el paso de tiempo anterior.

¡Construyamos y entrenemos este modelo! Primero, necesitamos descargar un conjunto de datos de pares de oraciones en inglés/español.<sup>15</sup>

```
url = "https://storage.googleapis.com/download.tensorflow.org/data/spa-eng.zip" ruta = tf.keras.utils.get_file("spa-eng.zip", origin=url, cache_dir ="conjuntos de datos", extraer=True) texto = (Ruta(ruta).with_name("spa-eng") / "spa.txt").read_text()
```

Cada línea contiene una frase en inglés y su correspondiente traducción al español, separadas por una tabulación. Comenzaremos eliminando los caracteres en español “” y “¿”, que la capa TextVectorization no maneja, luego analizaremos los pares de oraciones y los mezclaremos. Finalmente, los dividiremos en dos listas separadas, una por idioma:

```
importar numpy como np

texto = text.replace(";", "").replace("¿", "") pares = [line.split("\t") para
la línea en text.splitlines()] np.random.shuffle( pares) oraciones_en, oraciones_es =
zip(*pares) # separa los pares en 2
listas
```

Echemos un vistazo a los primeros tres pares de oraciones:

```
>>> para i en el rango(3):
...     print(frases_es[i], "=>", frases_en[i])
...
¡Qué aburrido! => ¡Qué aburrimiento!
Amo los deportes. => Adoro el deporte.
¿Te gustaría intercambiar trabajos? => ¿Te gustaría que intercambiáramos los trabajos?
```

A continuación, creemos dos capas de TextVectorization (una por idioma) y las adaptaremos al texto:

```
vocab_size = 1000
max_length = 50
text_vec_layer_en = tf.keras.layers.TextVectorization( vocab_size,
    output_sequence_length=max_length) text_vec_layer_es =
tf.keras.layers.TextVectorization( vocab_size, output_sequence_length=max_length)
    text_vec_layer_en.adapt(sentences_en) text_ve c_layer_es.adapt([f
"startofseq {s} endofseq" para s en oraciones_es])
```

Hay algunas cosas a tener en cuenta aquí:

- Limitamos el tamaño del vocabulario a 1000, que es bastante pequeño. Esto se debe a que el conjunto de entrenamiento no es muy grande y a que usar un valor pequeño acelerará el entrenamiento. Los modelos de traducción de última generación suelen utilizar un vocabulario mucho mayor (por ejemplo, 30.000), un conjunto de entrenamiento mucho mayor (gigabytes) y un modelo mucho mayor (cientos o incluso miles de megabytes). Por ejemplo, consulte los modelos Opus-MT de la Universidad de Helsinki o el modelo M2M-100 de Facebook.
- Dado que todas las oraciones en el conjunto de datos tienen un máximo de 50 palabras, configuraremos output\_sequence\_length en 50: de esta manera, las secuencias de entrada se completarán automáticamente con ceros hasta que tengan 50 tokens. Si hubiera alguna oración de más de 50 fichas en el conjunto de entrenamiento, se recortaría a 50 fichas.
- Para el texto en español, agregamos “startofseq” y “endofseq” a cada oración al adaptar la capa TextVectorization: usaremos estas palabras como tokens SOS y EOS. Puedes usar cualquier otra palabra, siempre que no sean palabras en español.

Inspeccionemos las primeras 10 fichas en ambos vocabularios. Comienzan con el token de relleno, el token desconocido, los tokens SOS y EOS (solo en el vocabulario en español), luego las palabras reales, ordenadas por frecuencia decreciente:

```
>>> text_vec_layer_en.get_vocabulary()[:10]
[", '[UNK]', 'el', 'yo', 'a', 'tú', 'tom', 'a', 'es', 'é!'] >>> text_vec_layer_es.get_vocabulary () [:10]

[", '[UNK]', 'startofseq', 'endofseq', 'de', 'que', 'a', 'no', 'tom', 'la']
```

A continuación, creamos el conjunto de entrenamiento y el conjunto de validación (también puede crear un conjunto de prueba si lo necesita). Usaremos los primeros 100.000 pares de oraciones para entrenamiento y el resto para validación. Las entradas del decodificador son las frases en español más un prefijo de token SOS. Los objetivos son las frases en español más un sufijo EOS:

```
X_train = tf.constant(sentencias_en[:100_000])
X_valid = tf.constant(sentencias_en[100_000:])
X_train_dec = tf.constant(["startofseq " + s for s in oraciones_es[:100_000]])
X_valid_dec = tf.constant(["startofseq " + s for s in oraciones_es[100_000:]])
Y_train = text_vec_layer_es(["s " + endofseq for s in oraciones_es[:100_000]])
Y_valid = text_vec_layer_es(["s " + endofseq for s in oraciones_es[100_000:]])
```

Bien, ahora estamos listos para construir nuestro modelo de traducción. Usaremos la API funcional para eso ya que el modelo no es secuencial. Requiere dos entradas de texto, una para el codificador y otra para el decodificador, así que comencemos con eso:

```
encoder_inputs = tf.keras.layers.Input(shape=[], dtype=tf.string) decoder_inputs =
tf.keras.layers.Input(shape=[], dtype=tf.string)
```

A continuación, debemos codificar estas oraciones usando las capas TextVectorization que preparamos anteriormente, seguidas de una capa de incrustación para cada idioma, con mask\_zero=True para garantizar que el enmascaramiento se maneje automáticamente. El tamaño de incrustación es un hiperparámetro que puedes ajustar, como siempre:

```
embed_size = 128
encoder_input_ids = text_vec_layer_en(encoder_inputs) decoder_input_ids =
= text_vec_layer_es(decoder_inputs) encoder_embedding_layer =
tf.keras.layers.Embedding( vocab_size, embed_size, mask_zero=True) decoder_embedding_layer =
tf.keras.layers.Embedding( tamaño_vocab, tamaño_incrustado, máscara_cero=True )
encoder_embeddings
= encoder_embedding_layer(encoder_input_ids) decoder_embeddings =
decoder_embedding_layer(decoder_input_ids)
```

#### CONSEJO

Cuando los idiomas comparten muchas palabras, es posible obtener un mejor rendimiento utilizando la misma capa de incrustación tanto para el codificador como para el decodificador.

Ahora creamos el codificador y le pasamos las entradas integradas:

```
codificador = tf.keras.layers.LSTM(512, return_state=True) encoder_outputs,
*encoder_state = codificador(encoder_embeddings)
```

Para simplificar las cosas, solo usamos una única capa LSTM, pero puedes apilar varias. También configuramos return\_state=True para obtener una referencia al estado final de la capa. Como usamos una capa LSTM, en realidad hay dos estados: el estado a corto plazo y el estado a largo plazo. La capa devuelve estos estados por separado, por lo que tuvimos que escribir \*encoder\_state para agrupar ambos estados en una lista. Ahora podemos usar este (doble) estado como estado inicial del decodificador:

```
decodificador = tf.keras.layers.LSTM(512, return_sequences=True)
decodificador_outputs = decodificador(decoder_embeddings, estado_inicial=estado_codificador)
```

A continuación, podemos pasar las salidas del decodificador a través de una capa Densa con la función de activación softmax para obtener las probabilidades de palabras para cada paso:

```
capa_salida = tf.keras.layers.Dense(vocab_size, activation="softmax")
Y_proba = capa_salida(salidas_decodificador)
```

## OPTIMIZANDO LA CAPA DE SALIDA

Cuando el vocabulario de salida es grande, generar una probabilidad para todas y cada una de las palabras posibles puede ser bastante lento. Si el vocabulario de destino contuviera, digamos, 50.000 palabras en español en lugar de 1.000, entonces el decodificador generaría vectores de 50.000 dimensiones, y calcular la función softmax sobre un vector tan grande sería muy intensivo desde el punto de vista computacional. Para evitar esto, una solución es observar únicamente los logits generados por el modelo para la palabra correcta y una muestra aleatoria de palabras incorrectas, y luego calcular una aproximación de la pérdida basada únicamente en estos logits. Esta técnica softmax de muestra se [introdujo en 2015](#)<sup>17</sup> por Sébastien Jean et al. En TensorFlow, puede usar la función `tf.nn.sampled_softmax_loss()` para esto durante el entrenamiento y usar la función softmax normal en el momento de la inferencia (el softmax muestrado no se puede usar en el momento de la inferencia porque requiere conocer el objetivo).

Otra cosa que puede hacer para acelerar el entrenamiento, que es compatible con softmax muestrado, es vincular los pesos de la capa de salida a la transposición de la matriz de incrustación del decodificador (verá cómo vincular los pesos en el Capítulo 17). Esto reduce significativamente la cantidad de parámetros del modelo, lo que acelera el entrenamiento y, en ocasiones, también puede mejorar la precisión del modelo, especialmente si no tiene muchos datos de entrenamiento. La matriz de incrustación es equivalente a una codificación one-hot seguida de una capa lineal sin término de sesgo y sin función de activación que asigna los vectores one-hot al espacio de incrustación. La capa de salida hace lo contrario. Entonces, si el modelo puede encontrar una matriz de incrustación cuya transpuesta sea cercana a su inversa (dicha matriz se llama matriz ortogonal), entonces no hay necesidad de aprender un conjunto separado de pesos para la capa de salida.

¡Y eso es! Sólo necesitamos crear el modelo Keras, compilarlo y entrenarlo:

```
modelo = tf.keras.Model(entradas=[encoder_inputs, decoder_inputs], salidas=[Y_proba])

modelo.compile(loss="sparse_categorical_crossentropy", optimizador="nadam",
               métricas=["precisión"])
modelo.fit((X_train, X_train_dec), Y_train, épocas=10,
           datos_validación=((X_valid, X_valid_dec), Y_valid))
```

Después del entrenamiento, podemos usar el modelo para traducir nuevas oraciones del inglés al español. Pero no es tan simple como llamar a `model.predict()`, porque el decodificador espera como entrada la palabra que se predijo en el paso de tiempo anterior. Una forma de hacer esto es escribir una celda de memoria personalizada que realice un seguimiento de la salida anterior y la envíe al codificador en el siguiente paso. Sin embargo, para simplificar las cosas, podemos llamar al modelo varias veces, prediciendo una palabra adicional en cada ronda. Escribamos una pequeña función de utilidad para eso:

```
def traducir(sentence_en): traducción =
    para word_idx en rango (max_length): X =
        np.array([sentence_en]) # entrada del codificador + traducción] #
        np.array(["startofseq"]) y_proba = modelo.predict((X)) # entrada del decodificador X_dec =
        X_dec) [0, word_idx] # probas del último token predicted_word_id = np.argmax(y_proba) predicted_word =
        text_vec_layer_es.get_vocabulary()[predicted_word_id] if
        predicted_word == "endofseq": break

        traducción += " " + predicted_word
    devolver traducción.strip()
```

La función simplemente continúa prediciendo una palabra a la vez, completando gradualmente la traducción y se detiene una vez que alcanza el token EOS. ¡Hagamos un intento!

```
>>> traducir("Me gusta el fútbol") 'me gusta  
el fútbol'
```

¡Hurra, funciona! Bueno, al menos lo hace con frases muy cortas. Si intentas jugar con este modelo por un tiempo, descubrirás que aún no es bilingüe y, en particular, tiene dificultades con oraciones más largas. Por ejemplo:

```
>>> traducir("Me gusta el fútbol y también ir a la playa") 'me gusta el fútbol ya veces  
mismo al bus'
```

La traducción dice "Me gusta el fútbol y a veces hasta el autobús". Entonces, ¿cómo puedes mejorarlo? Una forma es aumentar el tamaño del conjunto de entrenamiento y agregar más capas LSTM tanto en el codificador como en el decodificador. Pero esto sólo le llevará hasta cierto punto, así que veamos técnicas más sofisticadas, comenzando con capas recurrentes bidireccionales.

## RNN bidireccionales

En cada paso de tiempo, una capa recurrente regular solo mira las entradas pasadas y presentes antes de generar su salida. En otras palabras, es causal, lo que significa que no puede mirar hacia el futuro. Este tipo de RNN tiene sentido al pronosticar series temporales o en el decodificador de un modelo secuencia a secuencia (seq2seq). Pero para tareas como la clasificación de texto o en el codificador de un modelo seq2seq, a menudo es preferible mirar hacia adelante a las siguientes palabras antes de codificar una palabra determinada.

Por ejemplo, consideremos las frases "el brazo derecho", "la persona adecuada" y "el derecho a criticar": para codificar correctamente la palabra "derecho", es necesario mirar hacia adelante. Una solución es ejecutar dos capas recurrentes en las mismas entradas, una leyendo las palabras de izquierda a derecha y la otra leyéndolas de derecha a izquierda, luego combinar sus salidas en cada paso de tiempo, generalmente concatenándolas. Esto es lo que hace una capa recurrente bidireccional (ver [Figura 16-5](#)).

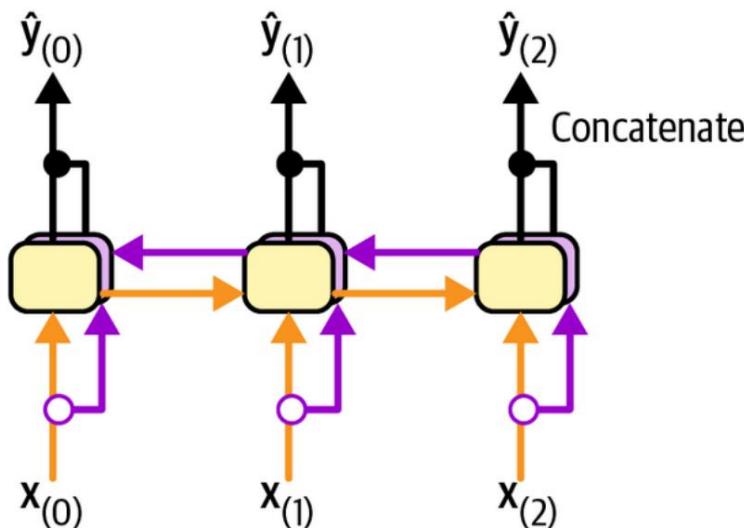


Figura 16-5. Una capa recurrente bidireccional

Para implementar una capa recurrente bidireccional en Keras, simplemente envuelva una capa recurrente en una capa `tf.keras.layers.Bidirectional`. Por ejemplo, la siguiente capa bidireccional podría usarse como codificador en nuestro modelo de traducción:

```
codificador = tf.keras.layers.Bidireccional(
    tf.keras.layers.LSTM(256, return_state=True))
```

### NOTA

La capa bidireccional creará un clon de la capa GRU (pero en la dirección inversa), ejecutará ambas y concatenará sus salidas. Entonces, aunque la capa GRU tiene 10 unidades, la capa bidireccional generará 20 valores por paso de tiempo.

Sólo hay un problema. Esta capa ahora devolverá cuatro estados en lugar de dos: los estados finales a corto y largo plazo de la capa LSTM directa, y los estados finales a corto y largo plazo de la capa LSTM inversa. No podemos usar este estado cuádruple directamente como el estado inicial de la capa LSTM del decodificador, ya que solo espera dos estados (a corto y largo plazo). No podemos hacer que el decodificador sea bidireccional, ya que debe seguir siendo causal: de lo contrario, haría trampa durante el entrenamiento y no funcionaría.

En cambio, podemos concatenar los dos estados de corto plazo y también concatenar los dos estados de largo plazo:

```
encoder_outputs, *encoder_state = codificador(encoder_embeddings) encoder_state =
[tf.concat(encoder_state[:2], eje=-1), # corto plazo (0 y 2) tf.concat(encoder_state[1::2], eje = -1)] # largo plazo (1 y 3)
```

Ahora veamos otra técnica popular que puede mejorar en gran medida el rendimiento de un modelo de traducción en el momento de la inferencia: la búsqueda por haz.

### Búsqueda de haz

Supongamos que ha entrenado un modelo codificador-decodificador y lo utiliza para traducir la frase "Me gusta el fútbol" al español. Esperas que genere la traducción adecuada "me gusta el fútbol", pero desafortunadamente muestra "me gustan los jugadores", que significa "Me gustan los jugadores". Al observar el conjunto de entrenamiento, se observan muchas oraciones como "I like cars", que se traduce como "me gustan los autos", por lo que no fue absurdo que el modelo mostrara "me gustan los" después de ver "I like". Lamentablemente, en este caso fue un error ya que "soccer" es singular. La modelo no podía volver atrás y arreglarlo, por lo que intentó completar la frase lo mejor que pudo, en este caso usando la palabra "jugadores". ¿Cómo podemos darle al modelo la oportunidad de retroceder y corregir los errores que cometió antes? Una de las soluciones más comunes es la búsqueda por haz: realiza un seguimiento de una lista corta de las k oraciones más prometedoras (por ejemplo, las tres primeras), y en cada paso del decodificador intenta ampliarlas en una palabra, manteniendo sólo las k más prometedoras . frases probables. El parámetro k se llama ancho de viga.

Por ejemplo, supongamos que utiliza el modelo para traducir la frase "Me gusta el fútbol" usando una búsqueda de haz con un ancho de haz de 3 (consulte [la Figura 16-6](#)). En el primer paso del decodificador, el modelo generará una probabilidad estimada para cada posible primera palabra en la oración traducida. Supongamos que las tres palabras principales son "yo" (75% de probabilidad estimada), "a" (3%) y "como" (1%). Esa es nuestra lista corta hasta ahora. A continuación, usamos el modelo para encontrar la siguiente palabra para cada oración. Para la primera oración ("yo"), quizás el modelo genere una probabilidad del 36% para la palabra "gustan", del 32% para la palabra "gusta", del 16% para la palabra "encanta", y así sucesivamente. Tenga en cuenta que en realidad se trata de probabilidades condicionales , dado que la oración comienza con "yo". Para la segunda oración ("a"), el modelo podría generar una probabilidad condicional del 50% para la palabra "mi", y así sucesivamente. Suponiendo que el vocabulario tiene 1000 palabras, terminaremos con 1000 probabilidades por oración.

A continuación, calculamos las probabilidades de cada una de las 3000 oraciones de dos palabras que consideramos ( $3 \times 1000$ ). Hacemos esto multiplicando la probabilidad condicional estimada de cada palabra por la probabilidad estimada de la oración que completa. Por ejemplo, la probabilidad estimada de la oración "yo" fue del 75%, mientras que la probabilidad condicional estimada de la palabra "gustan" (dado que la primera palabra es "yo") fue del 36%, por lo que la probabilidad estimada de la oración " me gustan" es  $75\% \times 36\% = 27\%$ . Después de calcular las probabilidades de las 3000 oraciones de dos palabras, nos quedamos solo con las 3 primeras. En este ejemplo, todas comienzan

con la palabra “yo”: “me gustan” (27%), “me gusta” (24%) y “me encanta” (12%). Ahora mismo gana la frase “me gustan”, pero “me gusta” no ha sido eliminada.



Figura 16-6. Búsqueda de haz, con un ancho de haz de 3

Luego repetimos el mismo proceso: usamos el modelo para predecir la siguiente palabra en cada una de estas tres oraciones y calculamos las probabilidades de las 3000 oraciones de tres palabras que consideramos. Quizás los tres primeros sean ahora “me gustan los” (10%), “me gusta el” (8%) y “me gusta mucho” (2%). En el siguiente paso podemos obtener “me gusta el fútbol” (6%), “me gusta mucho el” (1%) y “me gusta el deporte”. (0,2%). Observe que se eliminó “me gustan” y ahora está la traducción correcta. Mejoramos el rendimiento de nuestro modelo codificador-decodificador sin ningún entrenamiento adicional, simplemente usándolo de manera más inteligente.

#### CONSEJO

La biblioteca de complementos de TensorFlow incluye una API seq2seq completa que le permite crear modelos de codificador-decodificador con atención, incluida la búsqueda de haces y más. Sin embargo, su documentación es actualmente muy limitada. Implementar la búsqueda por haz es un buen ejercicio, ¡así que pruébalo! Consulte el cuaderno de este capítulo para encontrar una posible solución.

Con todo esto, puedes conseguir traducciones razonablemente buenas para frases bastante cortas. Desafortunadamente, este modelo será realmente malo traduciendo oraciones largas. Una vez más, el problema proviene de la limitada memoria a corto plazo de los RNN. Los mecanismos de atención son la innovación revolucionaria que abordó este problema.

## Mecanismos de atención

Considere el camino desde la palabra “soccer” hasta su traducción “fútbol” en la [Figura 16-3](#): ¡es bastante largo! Esto significa que una representación de esta palabra (junto con todas las demás palabras) debe realizarse durante muchos pasos antes de que se utilice realmente. ¿No podemos acortar este camino?

Esta fue la idea central de un [artículo histórico de 2014](#), por Dzmitry Bahdanau et al., donde los autores introdujeron una técnica que permitía al decodificador centrarse en las palabras apropiadas (codificadas por el codificador) en cada paso de tiempo. Por ejemplo, en el paso de tiempo en el que el decodificador necesita generar la palabra “fútbol”, centrará su atención en la palabra “soccer”. Esto significa que el camino desde una palabra de entrada hasta su traducción ahora es mucho más corto, por lo que las limitaciones de memoria a corto plazo de los RNN tienen mucho menos impacto.

Los mecanismos de atención revolucionaron la traducción automática neuronal (y el aprendizaje profundo en general), permitiendo una mejora significativa en el estado del arte, especialmente para oraciones largas (por ejemplo, más de 30 palabras).

## NOTA

La métrica más común utilizada en NMT es la puntuación del suplemento de evaluación bilingüe (BLEU), que compara cada traducción producida por el modelo con varias buenas traducciones producidas por humanos: cuenta el número de n-gramas (secuencias de n palabras) que aparecen en cualquiera de las traducciones de destino y ajusta la puntuación para tener en cuenta la frecuencia de los n-gramas producidos en las traducciones de destino.

**La Figura 16-7** muestra nuestro modelo codificador-decodificador con un mecanismo de atención adicional. A la izquierda tienes el codificador y el decodificador. En lugar de simplemente enviar el estado oculto final del codificador al decodificador, así como la palabra objetivo anterior en cada paso (lo cual todavía se hace, aunque no se muestra en la figura), ahora enviamos todas las salidas del codificador al decodificador. también. Dado que el decodificador no puede manejar todas estas salidas del codificador a la vez, es necesario agregarlas: en cada paso de tiempo, la celda de memoria del decodificador calcula una suma ponderada de todas las salidas del codificador. Esto determina en qué palabras se centrará en este paso. El peso  $\alpha$  es el peso de la salida del codificador  $i$  en el paso de tiempo del decodificador  $t$  ( $\alpha_{(t,i)}$ ). Por ejemplo, si el peso  $\alpha$  es mucho mayor que los pesos para las otras salidas del codificador, el decodificador priorizará más “fútbol” (‘fútbol’) que a la otra dos salidas, al menos en este paso de tiempo. El resto del decodificador funciona igual que antes: en cada paso de tiempo la celda de memoria recibe las entradas que acabamos de comentar, más el estado oculto del paso de tiempo anterior, y finalmente (aunque no está representado en el diagrama) recibe el objetivo. palabra del paso de tiempo anterior (o en el momento de la inferencia, la salida del paso de tiempo anterior).

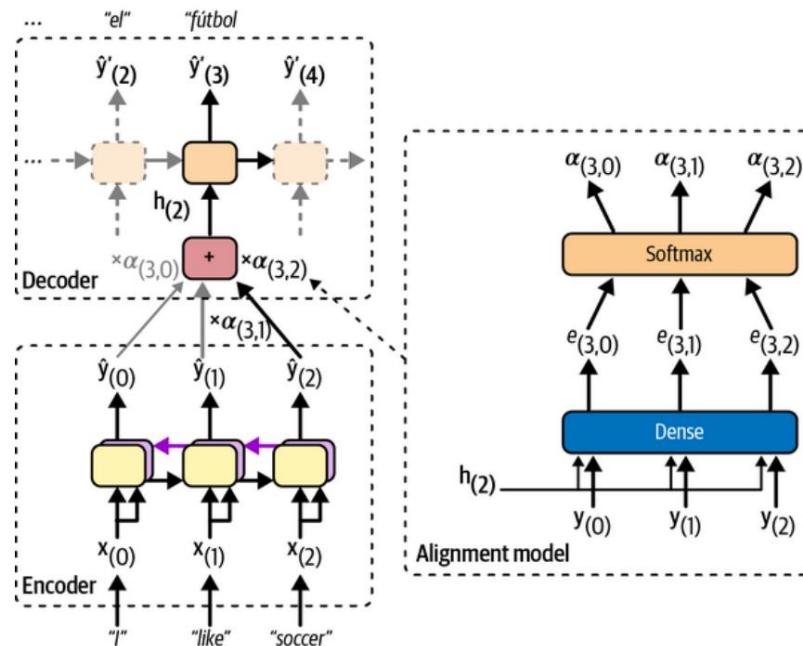


Figura 16-7. Traducción automática neuronal utilizando una red codificador-decodificador con un modelo de atención

¿ Pero de dónde vienen estos pesos  $\alpha$ ? Bueno, son generados por una pequeña red neuronal llamada modelo de alineación (o capa de atención), que se entrena junto con el resto del modelo codificador-decodificador. Este modelo de alineación se ilustra en el lado derecho de [la Figura 16-7](#). Comienza con una capa Densa compuesta por una sola neurona que procesa cada una de las salidas del codificador, junto con el estado oculto anterior del decodificador (por ejemplo,  $h$ ). Esta capa genera una puntuación (o energía) para cada salida del codificador (por ejemplo,  $e$ ): esta puntuación mide qué tan bien está alineada cada salida con el estado oculto anterior del decodificador. Por ejemplo, en [la Figura 16-7](#), el modelo ya generó “me gusta el”, por lo que ahora espera un sustantivo: la palabra “soccer” es la que mejor se alinea con el estado actual. por lo que obtiene una puntuación alta. Finalmente, todas las puntuaciones pasan por una capa softmax para obtener un peso final para cada una.

salida del codificador (por ejemplo,  $\alpha$ ). Todos los pesos para un paso de tiempo dado del decodificador suman 1. Este mecanismo de atención particular (3,2) se llama atención de Bahdanau (llamado así por el primer autor del artículo de 2014). Dado que concatena la salida del codificador con el estado oculto anterior del decodificador, a veces se le llama atención concatenativa (o atención aditiva).

### NOTA

Si la oración de entrada tiene  $n$  palabras y se supone que la oración de salida tiene aproximadamente la misma longitud, entonces este modelo necesitará calcular alrededor de  $n$  pesos. Afortunadamente, esta complejidad computacional cuadrática todavía es manejable porque incluso las oraciones largas no tienen miles de palabras.

Otro mecanismo de atención común, conocido como atención Luong o atención multiplicativa, fue propuesto poco después, en 2015, por Minh-Thang Luong et al. Debido a que el objetivo del modelo de alineación es medir la similitud entre una de las salidas del codificador y el estado oculto previo del decodificador, los autores propusieron simplemente calcular el producto escalar (ver Capítulo 4) de estos dos vectores, ya que esto suele ser un resultado bastante buena medida de similitud y el hardware moderno puede calcularla de manera muy eficiente. Para que esto sea posible, ambos vectores deben tener la misma dimensionalidad. El producto escalar proporciona una puntuación, y todas las puntuaciones (en un paso de tiempo determinado del decodificador) pasan por una capa softmax para dar los pesos finales, como en la atención de Bahdanau. Otra simplificación, Luong et al. propuesto fue utilizar el estado oculto del decodificador en el paso de tiempo actual en lugar del paso de tiempo anterior (es decir,  $h$  en lugar de  $h$  ), luego ( $t$ )

(t-1)

utilizar la salida del mecanismo de atención (anotado como  $h^*(t)$ ) directamente para calcular las predicciones del decodificador, en lugar de usarlo para calcular el estado oculto actual del decodificador. Los investigadores también propusieron una variante del mecanismo del producto escalar en el que las salidas del codificador pasan primero por una capa completamente conectada (sin un término de sesgo) antes de que se calculen los productos escalar. Esto se denomina enfoque "general" del producto escalar. Los investigadores compararon ambos enfoques del producto escalar con el mecanismo de atención concatenativa (agregando un vector de parámetro de reescalado  $v$ ) y observaron que las variantes del producto escalar funcionaron mejor que la atención concatenativa. Por esta razón, la atención concatenativa se utiliza mucho menos ahora. Las ecuaciones para estos tres mecanismos de atención se resumen en la Ecuación 16-1.

Ecuación 16-1. Mecanismos de atención

$$h^*(t) = \sum_i \alpha(t,i)y(i) \text{ con } \alpha(t,i) = \frac{\exp(e(t,i))}{\sum_i \exp(e(t,i))}$$

	y(yo)	punto
	$h(t) h(t) W y(i)$	géneros
$v \quad \tanh(W[h(t); y(i)])$	concat	

Keras proporciona una capa `tf.keras.layers.Attention` para la atención de Luong y una capa `AdditiveAttention` para la atención de Bahdanau. Agreguemos la atención de Luong a nuestro modelo codificador-decodificador. Dado que necesitaremos pasar todas las salidas del codificador a la capa de Atención, primero debemos configurar `return_sequences=True` al crear el codificador:

```
codificador = tf.keras.layers.Bidireccional(
    tf.keras.layers.LSTM(256, return_sequences=True, return_state=True))
```

A continuación, debemos crear la capa de atención y pasarele los estados del decodificador y las salidas del codificador. Sin embargo, para acceder a los estados del decodificador en cada paso necesitaríamos escribir una celda de memoria personalizada. Para simplificar, usemos las salidas del decodificador en lugar de sus estados: en la práctica esto también funciona bien y es mucho más fácil de codificar. Luego simplemente pasamos las salidas de la capa de atención directamente a la capa de salida, como se sugiere en el documento de atención de Luong:

```
capa_atención = tf.keras.layers.Attention() salidas_atención =
capa_atención([salidas_decodificador, salidas_codificador]) capa_salida = tf.keras.layers.Dense(vocab_size,
activación="softmax")
Y_proba = capa_salida(salidas_atención)
```

¡Y eso es! Si entrena este modelo, encontrará que ahora maneja oraciones mucho más largas. Por ejemplo:

```
>>> traducir("Me gusta el fútbol y también ir a la playa") 'me gusta el fútbol y también ir
a la playa'
```

En resumen, la capa de atención proporciona una forma de centrar la atención del modelo en parte de las entradas. Pero hay otra forma de pensar en esta capa: actúa como un mecanismo de recuperación de memoria diferenciable.

Por ejemplo, supongamos que el codificador analizó la oración de entrada "Me gusta el fútbol" y logró entender que la palabra "yo" es el sujeto y la palabra "me gusta" es el verbo, por lo que codificó esta información en sus salidas para estas palabras. Ahora supongamos que el decodificador ya ha traducido el sujeto y cree que debería traducir el verbo a continuación. Para ello, necesita recuperar el verbo de la oración de entrada. Esto es análogo a una búsqueda en un diccionario: es como si el codificador hubiera creado un diccionario {"sujeto": "Ellos", "verbo": "jugaron", ...} y el decodificador quisiera buscar el valor que corresponde a la clave. "verbo".

Sin embargo, el modelo no tiene tokens discretos para representar las claves (como "sujeto" o "verbo"); en cambio, tiene representaciones vectorizadas de estos conceptos que aprendió durante el entrenamiento, por lo que la consulta que utilizará para la búsqueda no coincidirá perfectamente con ninguna clave del diccionario. La solución es calcular una medida de similitud entre la consulta y cada clave en el diccionario, y luego usar la función softmax para convertir estas puntuaciones de similitud en pesos que suman 1. Como vimos anteriormente, eso es exactamente lo que hace la capa de atención. Si la clave que representa el verbo es, con diferencia, la más similar a la consulta, entonces el peso de esa clave será cercano a 1.

A continuación, la capa de atención calcula una suma ponderada de los valores correspondientes: si el peso de la clave "verbo" es cercano a 1, entonces la suma ponderada será muy cercana a la representación de la palabra "jugado".

Es por eso que las capas Keras Attention y AdditiveAttention esperan una lista como entrada, que contiene dos o tres elementos: las consultas, las claves y, opcionalmente, los valores. Si no pasa ningún valor, automáticamente serán iguales a las claves. Entonces, mirando nuevamente el ejemplo de código anterior, las salidas del decodificador son las consultas y las salidas del codificador son tanto las claves como los valores. Para cada salida del decodificador (es decir, cada consulta), la capa de atención devuelve una suma ponderada de las salidas del codificador (es decir, las claves/valores) que son más similares a la salida del decodificador.

La conclusión es que un mecanismo de atención es un sistema de recuperación de memoria entrenable. Es tan poderoso que puedes construir modelos de última generación utilizando únicamente mecanismos de atención. Ingrese a la arquitectura del transformador.

## Atención es todo lo que necesita: la arquitectura original del transformador

En un [artículo innovador de 2017](#), Un equipo de investigadores de Google sugirió que "todo lo que necesitas es atención".<sup>20</sup> Crearon una arquitectura llamada transformador, que mejoró significativamente el estado del arte en NMT sin usar capas recurrentes o convolucionales, solo mecanismos de atención (más capas de incrustación, capas densas, capas de normalización y algunas otras partes). ). Debido a que el modelo no es recurrente, no sufre tantos problemas de gradientes que desaparecen o explotan como los RNN, se puede entrenar en menos pasos, es más fácil de paralelizar entre múltiples GPU y puede capturar mejor patrones de largo alcance que RNN. La arquitectura del transformador original de 2017 se representa en la [Figura 16-8](#).

En resumen, la parte izquierda de la Figura 16-8 es el codificador y la parte derecha es el decodificador. Cada capa de incrustación genera un tensor de forma 3D [tamaño de lote, longitud de secuencia, tamaño de incrustación]. Después de eso, los tensores se transforman gradualmente a medida que fluyen a través del transformador, pero su forma sigue siendo la misma.

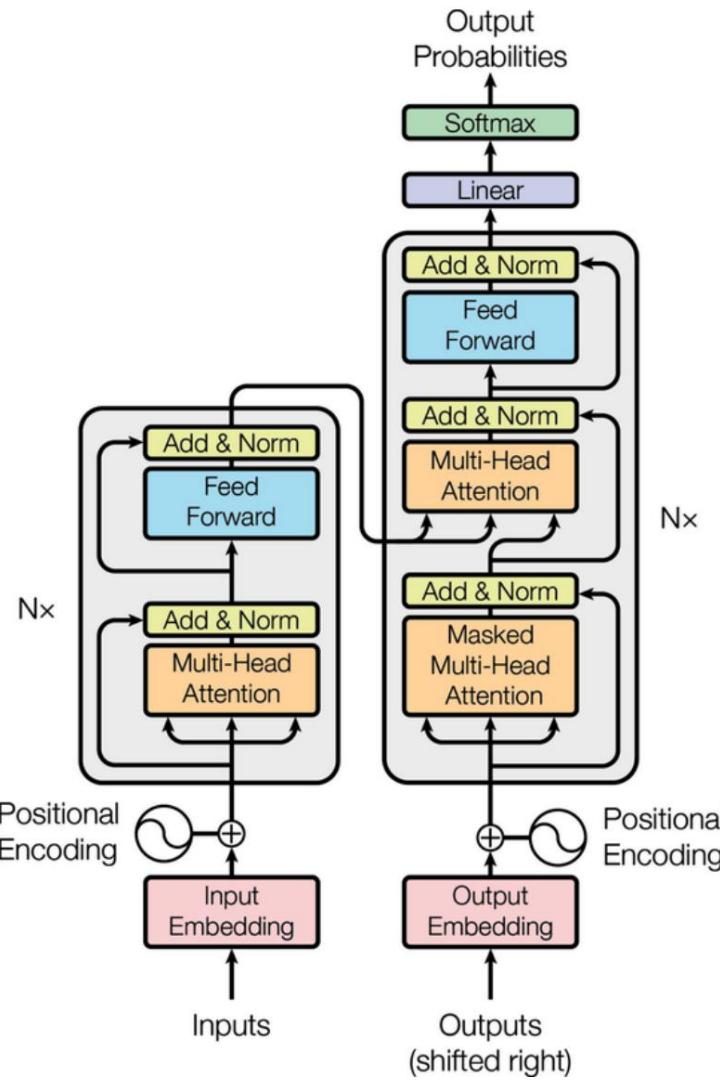


Figura 16-8. La arquitectura transformadora original de 2017.<sup>22</sup>

Si usa el transformador para NMT, durante el entrenamiento debe enviar las oraciones en inglés al codificador y las correspondientes traducciones al español al decodificador, con un token SOS adicional insertado al comienzo de cada oración. En el momento de la inferencia, debe llamar al transformador varias veces, producir las traducciones una palabra a la vez y enviar las traducciones parciales al decodificador en cada ronda, tal como lo hicimos anteriormente en la función traducir().

La función del codificador es transformar gradualmente las entradas (representaciones de palabras de la oración en inglés) hasta que la representación de cada palabra capture perfectamente el significado de la palabra, en el contexto de la oración. Por ejemplo, si alimenta el codificador con la frase "Me gusta el fútbol", entonces la palabra "me gusta" comenzará con una representación bastante vaga, ya que esta palabra podría significar diferentes cosas en diferentes contextos: piense en "Me gusta el fútbol". versus "Es así". Pero después de pasar por el codificador, la representación de la palabra debe capturar el significado correcto de "me gusta" en la oración dada (es decir, ser aficionado), así como cualquier otra información que pueda ser necesaria para la traducción (por ejemplo, es un verbo).

La función del decodificador es transformar gradualmente cada representación de palabra en la oración traducida en una representación de palabra de la siguiente palabra en la traducción. Por ejemplo, si la oración a traducir es “Me gusta el fútbol”, y la oración de entrada del decodificador es “<SOS> me gusta el fútbol”, luego de pasar por el decodificador, la representación de la palabra “el” terminará transformado en una representación de la palabra “fútbol”. De igual forma, la representación de la palabra “fútbol” se transformará en una representación del token EOS.

Después de pasar por el decodificador, cada representación de palabra pasa por una capa Densa final con una función de activación softmax, que con suerte generará una alta probabilidad para la siguiente palabra correcta y una baja probabilidad para todas las demás palabras. La frase prevista debería ser “me gusta el fútbol <EOS>”.

Ese era el panorama general; Ahora veamos [la Figura 16-8](#) con más detalle:

- Primero, observe que tanto el codificador como el decodificador contienen módulos que están apilados N veces. En el artículo, N = 6. Las salidas finales de toda la pila del codificador se envían al decodificador en cada uno de estos N niveles.
- Al hacer zoom, puede ver que ya está familiarizado con la mayoría de los componentes: hay dos capas de incrustación; varias conexiones de salto, cada una de ellas seguida de una capa de normalización de capas; varios módulos de avance que se componen de dos capas densas cada uno (el primero usa la función de activación ReLU, el segundo sin función de activación); y finalmente la capa de salida es una capa densa que utiliza la función de activación softmax. También puede agregar un poco de abandono después de las capas de atención y los módulos de avance, si es necesario. Dado que todas estas capas están distribuidas en el tiempo, cada palabra se trata de forma independiente de todas las demás. Pero, ¿cómo podemos traducir una oración mirando las palabras por separado? Bueno, no podemos, así que ahí es donde entran los nuevos componentes:
  - La capa de atención de múltiples cabezales del codificador actualiza la representación de cada palabra atendiendo (es decir, prestando atención) a todas las demás palabras en la misma oración. Ahí es donde la vaga representación de la palabra “me gusta” se convierte en una representación más rica y precisa, capturando su significado preciso en la oración dada. Discutiremos exactamente cómo funciona esto en breve.
  - La capa de atención enmascarada de múltiples cabezales del decodificador hace lo mismo, pero cuando procesa una palabra, no atiende a las palabras ubicadas después de ella: es una capa causal. Por ejemplo, cuando procesa la palabra “gusta”, solo atiende a las palabras “<SOS> me gusta”, e ignora las palabras “el fútbol” (o de lo contrario sería hacer trampa).
  - La capa superior de atención de múltiples cabezales del decodificador es donde el decodificador presta atención a las palabras de la oración en inglés. A esto se le llama atención cruzada, no atención propia en este caso. Por ejemplo, el decodificador probablemente prestará mucha atención a la palabra “soccer” cuando procese la palabra “el” y transforme su representación en una representación de la palabra “fútbol”.
  - Las codificaciones posicionales son vectores densos (muy parecidos a las incrustaciones de palabras) que representan la posición de cada palabra en la oración.<sup>11</sup> La codificación posicional n se agrega a la incrustación de palabras<sup>11</sup> de n palabras en cada oración. Esto es necesario porque todas las capas de la arquitectura del transformador ignoran las posiciones de las palabras: sin codificaciones posicionales, se podrían mezclar las secuencias de entrada, y las secuencias de salida se mezclarían de la misma manera. Obviamente, el orden de las palabras importa, por lo que necesitamos proporcionar información posicional al transformador de alguna manera: agregar codificaciones posicionales a las representaciones de palabras es una buena manera de lograrlo.

### NOTA

Las dos primeras flechas que van a cada capa de atención de múltiples cabezales en la Figura 16-8 representan las claves y los valores, y la tercera flecha representa las consultas. En las capas de autoatención, las tres son iguales a las representaciones de palabras generadas por la capa anterior, mientras que en la capa de atención superior del decodificador, las claves y los valores son iguales a las representaciones de palabras finales del codificador, y las consultas son iguales a la palabra. representaciones generadas por la capa anterior.

Repasemos con más detalle los nuevos componentes de la arquitectura del transformador, comenzando con las codificaciones posicionales.

#### Codificaciones posicionales

Una codificación posicional es un vector denso que codifica la posición de una palabra dentro de una oración: la codificación posicional  $i$  se agrega a la incrustación de la palabra  $i$  en la oración. La forma más sencilla de implementar esto es usar una capa de incrustación y hacer que codifique todas las posiciones desde 0 hasta la longitud máxima de secuencia en el lote, luego agregar el resultado a las incrustaciones de palabras. Las reglas de transmisión garantizarán que las codificaciones posicionales se apliquen a cada secuencia de entrada. Por ejemplo, aquí se explica cómo agregar codificaciones posicionales a las entradas del codificador y del decodificador:

```
max_length = 50 # longitud máxima en todo el conjunto de entrenamiento embed_size =
128 pos_embed_layer =
tf.keras.layers.Embedding(max_length, embed_size) batch_max_len_enc = tf.shape(encoder_embeddings)
[1] encoder_in = encoder_embeddings + pos_embed_layer(tf.range(batch_max_len_enc ))
lote_max_len_dec = tf.shape(decoder_embeddings)[1] decoder_in = decoder_embeddings +
pos_embed_layer(tf.range(batch_max_len_dec))
```

Tenga en cuenta que esta implementación supone que las incorporaciones se representan como tensores regulares, no como tensores irregulares.<sup>23</sup> El codificador y el decodificador comparten la misma capa de incrustación para las codificaciones posicionales, ya que tienen el mismo tamaño de incrustación (este suele ser el caso).

En lugar de utilizar codificaciones posicionales entrenables, los autores del artículo sobre transformadores optaron por utilizar codificaciones posicionales fijas, basadas en las funciones seno y coseno en diferentes frecuencias. La matriz de codificación posicional  $P$  se define en la ecuación 16-2 y se representa en la parte superior de la figura 16-9 (transpuesta).  
th donde  $P_{pi}$  es el componente  $i$  de la codificación de la palabra ubicada en la posición  $p$  de la oración.

Ecuación 16-2. Codificaciones posicionales seno/coseno

$$P_{pi} = \begin{cases} \operatorname{sen}(p/10000 i/d) & \text{si } i \text{ es par} \\ \operatorname{cos}(p/10000 (i-1)/d) & \text{si } i \text{ es impar} \end{cases}$$

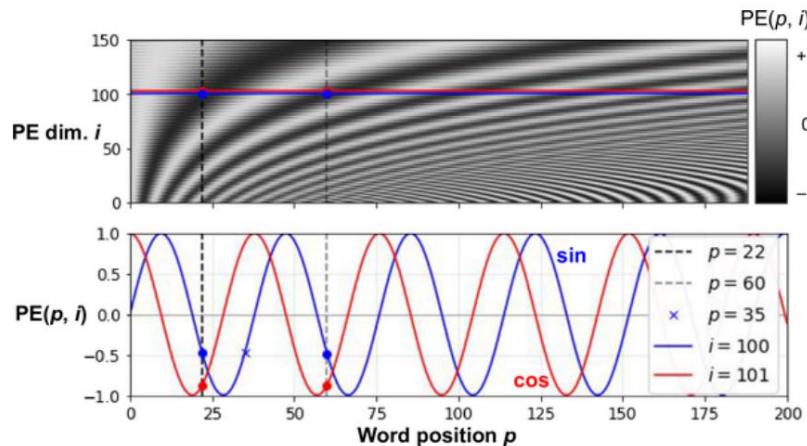


Figura 16-9. Matriz de codificación posicional seno/coseno (transpuesta, arriba) centrándose en dos valores de  $i$  (abajo)

Esta solución puede ofrecer el mismo rendimiento que las codificaciones posicionales entrenables y puede extenderse a oraciones arbitrariamente largas sin agregar ningún parámetro al modelo (sin embargo, cuando hay una gran cantidad de datos de preentrenamiento, generalmente se prefieren las codificaciones posicionales entrenables). Después de agregar estas codificaciones posicionales a las incrustaciones de palabras, el resto del modelo tiene acceso a la posición absoluta de cada palabra en la oración porque hay una codificación posicional única para cada posición (por ejemplo, la codificación posicional para la palabra ubicada en la La posición 22 de una oración está representada por la línea discontinua vertical en la parte superior izquierda de la Figura 16-9, y puede ver que es exclusiva de esa posición). Además, la elección de funciones oscilantes (seno y coseno) hace posible que el modelo también aprenda posiciones relativas. Por ejemplo, las palabras ubicadas a 38 palabras de distancia (por ejemplo, en las posiciones  $p = 22$  y  $p = 60$ ) siempre tienen los mismos valores de codificación posicional en las dimensiones de codificación  $i = 100$  e  $i = 101$ , como se puede ver en la Figura 16-9. Esto explica por qué necesitamos tanto el seno como el coseno para cada frecuencia: si solo usáramos el seno (la onda azul en  $i = 100$ ), el modelo no sería capaz de distinguir las posiciones  $p = 22$  y  $p = 35$  (marcadas por al otro lado de).

No existe una capa PositionalEncoding en TensorFlow, pero no es demasiado difícil crear una. Por razones de eficiencia, precalculamos la matriz de codificación posicional en el constructor. El método call() simplemente trunca esta matriz de codificación a la longitud máxima de las secuencias de entrada y las agrega a las entradas. También configuramos support\_masking=True para propagar la máscara automática de entrada a la siguiente capa:

```

clase Codificación posicional (tf.keras.layers.Layer):
    def __init__(self, max_length, embed_size, dtype=tf.float32, **kwargs):
        super().__init__(dtype=dtype, **kwargs) afirmar
        embed_size % 2 == 0, "embed_size debe ser par" p, i =
        np.meshgrid(np.arange(max_length), 2 *
                    np.arange(embed_size // 2)) pos_emb =
        np.empty((1, max_length, embed_size)) pos_emb[0, :, ::2] = np.sin(p /
        10_000 ** (i / embed_size)).T pos_emb[0, :, 1::2] = np.cos(p / 10_000 ** (i / embed_size)).T
        self.pos_encodings = tf.constant(pos_emb.astype(self.dtype)) self.supports_masking = True

    def llamada(self, entradas):
        lote_max_length = tf.shape(entradas)[1] devolver
        entradas + self.pos_encodings[:, :batch_max_length]

```

Usemos esta capa para agregar la codificación posicional a las entradas del codificador:

```

pos_embed_layer = PositionalEncoding(max_length, embed_size) encoder_in =
pos_embed_layer(encoder_embeddings) decoder_in =
pos_embed_layer(decoder_embeddings)

```

Ahora profundicemos en el corazón del modelo de transformador, en la capa de atención de múltiples cabezales.

#### Atención multicabezal

Para comprender cómo funciona una capa de atención de múltiples cabezas, primero debemos comprender el producto escalado. capa de atención , en la que se basa. Su ecuación se muestra en la [Ecuación 16-3](#), en forma vectorizada. Es el igual que la atención de Luong, excepto por un factor de escala.

#### Ecuación 16-3. Atención de productos escalados

$$\text{Atención } (Q, K, V) = \text{softmax} (QK^\top / \sqrt{d_{\text{keys}}})V$$

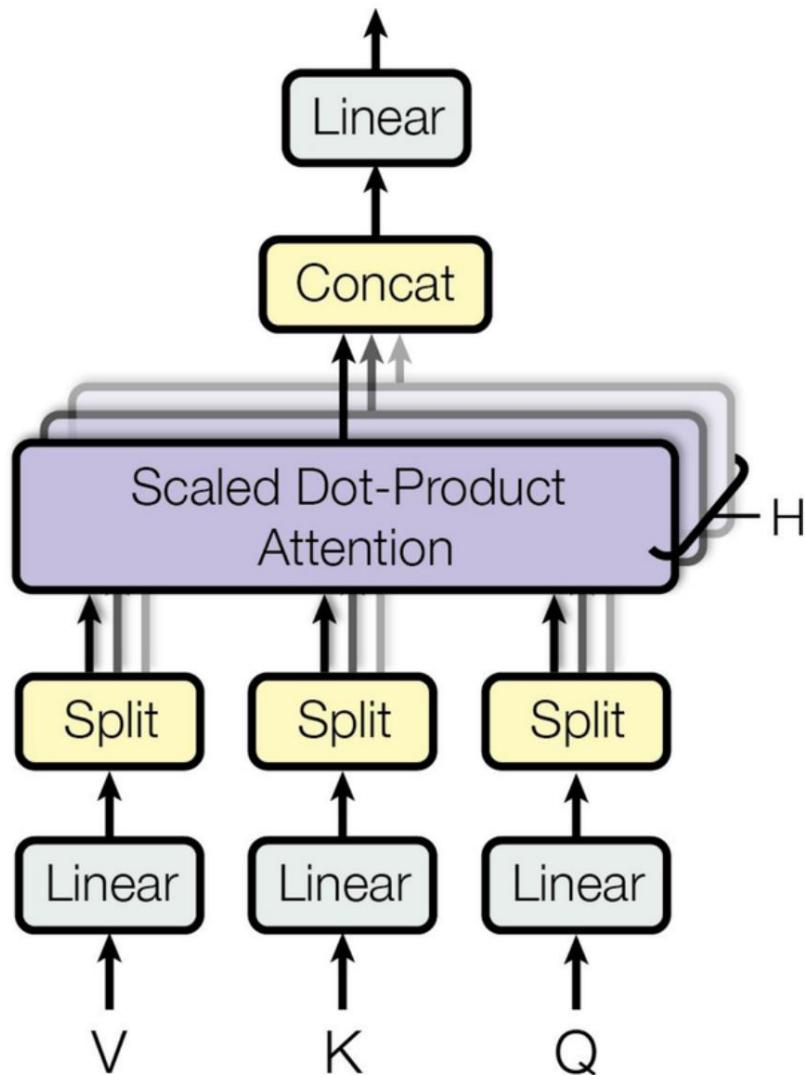
En esta ecuación:

- $Q$  es una matriz que contiene una fila por consulta. Su forma es  $[nd]$ , donde  $n$  es el número de consultas y  $d$  es el número de dimensiones de cada consulta y de cada clave.
- $K$  es una matriz que contiene una fila por clave. Su forma es  $[nd]$ , donde  $n$  es el número de claves y valores.
- $V$  es una matriz que contiene una fila por valor. Su forma es  $[nd]$ , donde  $d$  es el número de dimensiones de cada valor.
- La forma de  $QK$  es  $[n \text{ claves de consulta}]$ : contiene una puntuación de similitud para cada consulta/par de claves. A Para evitar que esta matriz sea enorme, las secuencias de entrada no deben ser demasiado largas (discutiremos cómo superar esta limitación más adelante en este capítulo). La salida de la función softmax tiene el mismo forma, pero todas las filas suman 1. La salida final tiene una forma de  $[nd]$ : hay una fila  $\frac{\text{valores}}{\text{claves}} \text{ de consultas}$  por consulta, donde cada fila representa el resultado de la consulta (una suma ponderada de los valores).
- El factor de escala  $1 / (\sqrt{d_{\text{keys}}})$  reduce las puntuaciones de similitud para evitar saturar el softmax. función, lo que conduciría a pequeños gradientes.
- Es posible enmascarar algunos pares clave/valor agregando un valor negativo muy grande a las puntuaciones de similitud correspondientes, justo antes de calcular el softmax. Esto es útil en personas enmascaradas. capa de atención de múltiples cabezas.

Si establece `use_scale=True` al crear una capa `tf.keras.layers.Attention`, se creará un parámetro adicional que permite a la capa aprender cómo reducir correctamente las puntuaciones de similitud. El La atención del producto escalado utilizada en el modelo del transformador es casi la misma, excepto que siempre escala la similitud puntúa por el mismo factor,  $1 / (\sqrt{d_{\text{keys}}})$ .

Tenga en cuenta que las entradas de la capa Atención son como  $Q$ ,  $K$  y  $V$ , excepto con una dimensión de lote adicional. (la primera dimensión). Internamente, la capa calcula todas las puntuaciones de atención de todas las oraciones del lote. con solo una llamada a `tf.matmul(consultas, claves)`: esto lo hace extremadamente eficiente. De hecho, en TensorFlow, si  $A$  y  $B$  son tensores con más de dos dimensiones, digamos, de forma  $[2, 3, 4, 5]$  y  $[2, 3, 5, 6]$ , respectivamente, entonces `tf.matmul(A, B)` tratará estos tensores como matrices de  $2 \times 3$  donde cada celda contiene una matriz y multiplicará las matrices correspondientes: la matriz en la fila  $i$  y la columna  $j$  en  $A$  se multiplicará por la matriz en la fila  $i$  y la columna  $j$  en  $B$ . Dado que el producto de una matriz de  $4 \times 5$  con una matriz de  $5 \times 6$  es una matriz de  $4 \times 6$ , `tf.matmul(A, B)` devolverá una matriz de forma  $[2, 3, 4, 6]$ .

Ahora estamos listos para observar la capa de atención de múltiples cabezas. Su arquitectura se muestra en [la Figura 16-10](#).

Figura 16-10. Arquitectura de capa de atención de múltiples cabezales<sup>24</sup>

Como puede ver, es solo un conjunto de capas de atención de productos escalados, cada una precedida por una transformación lineal de los valores, claves y consultas (es decir, una capa densa distribuida en el tiempo sin función de activación). Todas las salidas simplemente se concatenan y pasan por una transformación lineal final (nuevamente, distribuida en el tiempo).

¿Pero por qué? ¿Cuál es la intuición detrás de esta arquitectura? Bueno, consideremos una vez más la palabra "me gusta" en la frase "Me gusta el fútbol". El codificador fue lo suficientemente inteligente como para codificar el hecho de que es un verbo. Pero la palabra representación también incluye su posición en el texto, gracias a las codificaciones posicionales, y probablemente incluye muchas otras características que son útiles para su traducción, como el hecho de que está en tiempo presente. En resumen, la representación de la palabra codifica muchas características diferentes de la palabra. Si solo usáramos una única capa de atención de producto escalado, solo podríamos consultar todas estas características de una sola vez.

Es por eso que la capa de atención de múltiples cabezas aplica múltiples transformaciones lineales diferentes de los valores, claves y consultas: esto permite que el modelo aplique muchas proyecciones diferentes de la representación de la palabra en diferentes subespacios, cada uno de los cuales se centra en un subconjunto de las características de la palabra. Quizás una de las capas lineales proyecte la representación de la palabra en un subespacio donde todo lo que queda es la información de que la palabra es un verbo, otra capa lineal extraerá solo el hecho de que está en tiempo presente,

etcétera. Luego, las capas de atención del producto escalado implementan la fase de búsqueda y, finalmente, concatenamos todos los resultados y los proyectamos de nuevo al espacio original.

Keras incluye una capa `tf.keras.layers.MultiHeadAttention`, por lo que ahora tenemos todo lo que necesitamos para construir el resto del transformador. Comencemos con el codificador completo, que es exactamente como en [la Figura 16-8](#), excepto que usamos una pila de dos bloques ( $N = 2$ ) en lugar de seis, ya que no tenemos un conjunto de entrenamiento enorme, y agregamos un poco de abandono también:

```
N = 2 # en lugar de 6 num_heads
= 8 dropout_rate =
0.1 n_units = 128 # para la
primera capa densa en cada bloque de avance encoder_pad_mask = tf.math.not_equal(encoder_input_ids,
0);, tf.newaxis]
Z = encoder_in para
dentro del rango (N): skip
= Z
attn_layer = tf.keras.layers.MultiHeadAttention(
    num_heads=num_heads, key_dim=embed_size, dropout=dropout_rate)
Z = attn_layer(Z, valor=Z, atención_mask=encoder_pad_mask)
Z = tf.keras.layers.LayerNormalization()(tf.keras.layers.Add()(Z, skip)) skip = ZZ = tf.keras.layers.Dense(n_units,
activación="relu")(z)
Z = tf.keras.layers.Dense(embed_size)(Z)
Z = tf.keras.layers.Abandono(tasa_deserción)(Z)
Z = tf.keras.layers.LayerNormalization()(tf.keras.layers.Add()(Z, omitir)))
```

Este código debería ser bastante sencillo, excepto por una cosa: el enmascaramiento. Al momento de escribir este artículo, la capa `MultiHeadAttention` no admite el enmascaramiento automático, por lo que debemos manejarlo [25](#) manualmente. ¿Cómo podemos hacer eso?

La capa `MultiHeadAttention` acepta un argumento de máscara de atención, que es un tensor booleano de forma [tamaño de lote, longitud máxima de la consulta, longitud máxima del valor]: para cada token en cada secuencia de consulta, esta máscara indica qué tokens en la secuencia de valores correspondiente deben ser atendidos. . Queremos decirle a la capa `MultiHeadAttention` que ignore todos los tokens de relleno en los valores. Entonces, primero calculamos la máscara de relleno usando `tf.math.not_equal(encoder_input_ids, 0)`. Esto devuelve un tensor booleano de forma [tamaño de lote, longitud máxima de secuencia]. Luego insertamos un segundo eje usando `[:, tf.newaxis]`, para obtener una máscara de forma [tamaño de lote, 1, longitud máxima de secuencia]. Esto nos permite usar esta máscara como máscara de atención al llamar a la capa `MultiHeadAttention`: gracias a la transmisión, se usará la misma máscara para todos los tokens en cada consulta. De esta manera, los tokens de relleno en los valores se ignorarán correctamente.

Sin embargo, la capa calculará los resultados de cada token de consulta, incluidos los tokens de relleno. Necesitamos enmascarar las salidas que corresponden a estos tokens de relleno. Recuerde que usamos `mask_zero` en las capas de incrustación y configuramos `support_masking` en `True` en la capa `PositionalEncoding`, por lo que la máscara automática se propagó hasta las entradas de la capa `MultiHeadAttention` (`encoder_in`). Podemos usar esto a nuestro favor en la conexión de omisión: de hecho, la capa Agregar admite el enmascaramiento automático, por lo que cuando agregamos `Z` y `omitimos` (que inicialmente es igual a `encoder_in`), las salidas se enmascaran automáticamente y correctamente. ¡Ay! El enmascaramiento requirió mucha más explicación que el código.

26

¡Ahora al decodificador! Una vez más, el enmascaramiento será la única parte complicada, así que comencemos con eso. La primera capa de atención de múltiples cabezas es una capa de autoatención, como en el codificador, pero es una capa de atención de múltiples cabezas enmascarada , lo que significa que es causal: debería ignorar todos los tokens en el futuro. Entonces, necesitamos dos máscaras: una máscara acolchada y una máscara causal. Vamos a crearlos:

```
decoder_pad_mask = tf.math.not_equal(decoder_input_ids, 0)[:, tf.newaxis] causal_mask = tf.linalg.band_part(#
crea una matriz triangular inferior
```

```
tf.ones((batch_max_len_dec, lote_max_len_dec), tf.bool), -1, 0)
```

La máscara de relleno es exactamente igual a la que creamos para el codificador, excepto que se basa en las entradas del decodificador en lugar de las del codificador. La máscara causal se crea usando la función `tf.linalg.band_part()`, que toma un tensor y devuelve una copia con todos los valores fuera de una banda diagonal establecidos en cero. Con estos argumentos, obtenemos una matriz cuadrada de tamaño `lote_max_len_dec` (la longitud máxima de las secuencias de entrada en el lote), con 1 en el triángulo inferior izquierdo y 0 en la parte superior derecha. Si usamos esta máscara como máscara de atención, obtendremos exactamente lo que queremos: el primer token de consulta solo atenderá al primer token de valor, el segundo solo atenderá a los dos primeros, el tercero solo atenderá a los tres primeros , etcétera. En otras palabras, los tokens de consulta no pueden atender a ningún token de valor en el futuro.

Ahora construyamos el decodificador:

```
encoder_outputs = Z # guardemos las salidas finales del codificador Z = decoder_in # el
decodificador comienza con sus propias entradas en el rango (N): skip = Z attn_layer
para _ =
    tf.keras.layers.MultiHeadAttention(
        num_heads=num_heads, key_dim=embed_size, dropout=dropout_rate)
    Z = attn_layer(Z, valor=Z, máscara_atención=máscara_causal y máscara_pad_decodificador)
    Z = tf.keras.layers.LayerNormalization()(tf.keras.layers.Add()([Z, skip])) skip = Z attn_layer =
        tf.keras.layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_size, dropout=dropout_rate)
        Z = attn_layer(Z, valor=encoder_outputs, atención_mask=encoder_pad_mask)
        Z = tf.keras.layers.LayerNormalization()(tf.keras.layers.Add()([Z, skip])) skip = ZZ = tf.keras.layers.Dense(n_units,
activación="relu")(z)
Z = tf.keras.layers.Dense(embed_size)(Z)
Z = tf.keras.layers.LayerNormalization()(tf.keras.layers.Add()([Z, omitir]))
```

Para la primera capa de atención, utilizamos `causal_mask` y `decoder_pad_mask` para enmascarar tanto los tokens de relleno como los tokens futuros. La máscara causal solo tiene dos dimensiones: le falta la dimensión del lote, pero está bien ya que la transmisión garantiza que se copie en todas las instancias del lote.

Para la segunda capa de atención, no hay nada especial. Lo único a tener en cuenta es que estamos usando `encoder_pad_mask`, no `decoder_pad_mask`, porque esta capa de atención utiliza las salidas finales del codificador como valores.

Ya casi hemos terminado. Sólo necesitamos agregar la capa de salida final, crear el modelo, compilarlo y entrenarlo:

```
Y_proba = tf.keras.layers.Dense(vocab_size, activación="softmax")(Z) modelo =
tf.keras.Model(entradas=[encoder_inputs, decoder_inputs], salidas=[Y_proba]) model.compile(loss="sparse_categorical_crossentropy", optimizador="nadam",
métricas=["precisión"])
model.fit((X_train, X_train_dec), Y_train, épocas=10,
datos_validación=((X_valid, X_valid_dec), Y_valid))
```

¡Felicitaciones! Construiste un transformador completo desde cero y lo entrenaste para la traducción automática. ¡Esto se está poniendo bastante avanzado!

#### CONSEJO

El equipo de Keras ha creado un nuevo [proyecto Keras NLP](#), incluyendo una API para construir un transformador más fácilmente. También te puede interesar el nuevo [proyecto Keras CV](#) para visión por computadora.

Pero el campo no se detuvo allí. Exploraremos ahora algunos de los avances recientes.

## Una avalancha de modelos de transformadores

El año 2018 ha sido denominado el “momento ImageNet para la PNL”. Desde entonces, el progreso ha sido asombroso, con arquitecturas basadas en transformadores cada vez más grandes entrenadas en inmensos conjuntos de datos.

Primero, el [documento GPT<sup>27</sup>](#) por Alec Radford y otros investigadores de OpenAI demostraron una vez más la efectividad del preentrenamiento no supervisado, como los artículos anteriores de ELMo y ULMFiT, pero esta vez utilizando una arquitectura similar a un transformador. Los autores entrenaron previamente una arquitectura grande pero bastante simple compuesta por una pila de 12 módulos transformadores utilizando solo capas de atención enmascaradas de múltiples cabezales, como en el decodificador del transformador original. Lo entrenaron en un conjunto de datos muy grande, utilizando la misma técnica autorregresiva que usamos para nuestro char-RNN de Shakespeare: simplemente predice el siguiente token. Esta es una forma de aprendizaje autosupervisado. Luego lo perfeccionaron en varias tareas lingüísticas, utilizando sólo adaptaciones menores para cada tarea. Las tareas eran bastante diversas: incluían clasificación de texto, vinculación (si la oración A impone, involucra o implica la oración B como una consecuencia necesaria), similitud (por ejemplo, “Hoy hace buen tiempo” es muy similar<sup>28</sup> a “Hace sol”), y respuesta a preguntas (dados algunos párrafos de texto que brindan algo de contexto, el modelo debe responder algunas preguntas de opción múltiple).

Luego [el documento BERT](#) de Google<sup>29</sup> salió: también demostró la efectividad del preentrenamiento autosupervisado en un corpus grande, usando una arquitectura similar a GPT pero solo con capas de atención de múltiples cabezales sin máscara, como en el codificador del transformador original. Esto significa que el modelo es naturalmente bidireccional; de ahí la B en BERT (Representaciones de codificador bidireccional de transformadores). Lo más importante es que los autores propusieron dos tareas de preentrenamiento que explican la mayor parte de las fortalezas del modelo:

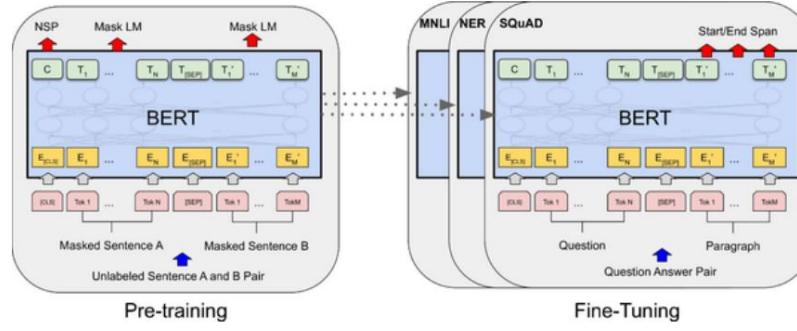
### Modelo de lenguaje enmascarado (MLM)

Cada palabra de una oración tiene un 15% de probabilidad de estar enmascarada y el modelo está entrenado para predecir las palabras enmascaradas. Por ejemplo, si la oración original es “Ella se divirtió en la fiesta de cumpleaños”, entonces al modelo se le puede dar la oración “Ella <máscara> se divirtió en la fiesta <máscara>” y debe predecir las palabras “tenía” y “cumpleaños” (las otras salidas serán ignoradas). Para ser más precisos, cada palabra seleccionada tiene un 80% de posibilidades de ser enmascarada, un 10% de posibilidades de ser reemplazada por una palabra aleatoria (para reducir la discrepancia entre el entrenamiento previo y el ajuste fino, ya que el modelo no verá tokens <máscara> durante el ajuste fino) y un 10% de posibilidades de quedarse solo (para sesgar el modelo hacia la respuesta correcta).

### Predicción de la siguiente oración (NSP)

El modelo está entrenado para predecir si dos oraciones son consecutivas o no. Por ejemplo, debería predecir que “El perro duerme” y “Ronca fuerte” son frases consecutivas, mientras que “El perro duerme” y “La Tierra orbita alrededor del Sol” no son consecutivas. Investigaciones posteriores demostraron que NSP no era tan importante como se pensaba inicialmente, por lo que se eliminó en la mayoría de las arquitecturas posteriores.

El modelo se entrena en estas dos tareas simultáneamente (consulte [la Figura 16-11](#)). Para la tarea NSP, los autores insertaron un token de clase (<CLS>) al comienzo de cada entrada, y el token de salida correspondiente representa la predicción del modelo: la oración B sigue a la oración A, o no. Las dos oraciones de entrada se concatenan, se separan únicamente por un token de separación especial (<SEP>) y se introducen como entrada al modelo. Para ayudar al modelo a saber a qué oración pertenece cada token de entrada, se agrega una incrustación de segmento encima de las incrustaciones posicionales de cada token: solo hay dos incrustaciones de segmentos posibles, una para la oración A y otra para la oración B. Para la tarea MLM, algunas las palabras de entrada están enmascaradas (como acabamos de ver) y el modelo intenta predecir cuáles eran esas palabras. La pérdida sólo se calcula en la predicción del NSP y en los tokens enmascarados, no en los desenmascarados.

Figura 16-11. Proceso de formación y ajuste de BERT<sup>30</sup>

Después de esta fase de preentrenamiento no supervisado en un corpus de texto muy grande, el modelo se ajusta en muchas tareas diferentes, cambiando muy poco para cada tarea. Por ejemplo, para la clasificación de texto, como el análisis de sentimientos, se ignoran todos los tokens de salida excepto el primero, correspondiente al token de clase, y una nueva capa de salida reemplaza a la anterior, que era solo una capa de clasificación binaria para NSP.

En febrero de 2019, apenas unos meses después de la publicación de BERT, Alec Radford, Jeffrey Wu y otros investigadores de OpenAI publicaron el artículo GPT-2, que propone<sup>31</sup> una arquitectura muy similar a GPT, pero aún más grande (¡con más de 1.500 millones de parámetros!). Los investigadores demostraron que el nuevo y mejorado modelo GPT podía realizar un aprendizaje de disparo cero (ZSL), lo que significa que podía lograr un buen rendimiento en muchas tareas sin ningún ajuste. Este fue solo el comienzo de una carrera hacia modelos cada vez más grandes: los Switch Transformers de Google, (introducido en enero de 2021) utilizó 1 billón de parámetros y pronto aparecieron modelos mucho más grandes, como el modelo Wu Dao 2.0 de la Academia de Inteligencia Artificial de Beijing (BAII), anunciado en junio de 2021.

Una consecuencia desafortunada de esta tendencia hacia modelos gigantes es que sólo las organizaciones bien financiadas pueden darse el lujo de entrenar tales modelos: fácilmente puede costar cientos de miles de dólares o más. Y la energía necesaria para entrenar un único modelo corresponde al consumo eléctrico de un hogar estadounidense durante varios años; no es nada ecológico. Muchos de estos modelos son demasiado grandes para usarse incluso en hardware normal: no cabrían en la RAM y serían terriblemente lentos. Por último, algunos son tan costosos que no se hacen públicos.

Afortunadamente, investigadores ingeniosos están encontrando nuevas formas de reducir el tamaño de los transformadores y hacerlos más eficiente en datos. Por ejemplo, el modelo DistilBERT, presentado en octubre de 2019 por Victor Sanh et al. de Hugging Face, es un modelo de transformador pequeño y rápido basado en BERT. Está disponible en el excelente centro de modelos de Hugging Face, junto con miles de otros; verá un ejemplo más adelante en este capítulo.

DistilBERT se entrenó mediante destilación (de ahí el nombre): esto significa transferir conocimiento de un modelo de profesor a uno de estudiante, que suele ser mucho más pequeño que el modelo de profesor. Por lo general, esto se hace utilizando las probabilidades previstas por el profesor para cada instancia de capacitación como objetivos para el estudiante.

Sorprendentemente, la destilación a menudo funciona mejor que entrenar al estudiante desde cero con el mismo conjunto de datos que el profesor. De hecho, el estudiante se beneficia de las etiquetas más matizadas del profesor.

Muchas más arquitecturas de transformadores surgieron después de BERT, casi mensualmente, y a menudo mejoraron el estado del arte en todas las tareas de PNL: XLNet (junio de 2019), RoBERTa (julio de 2019), StructBERT (agosto de 2019), ALBERT (septiembre de 2019), T5 (octubre de 2019), ELECTRA (marzo de 2020), GPT3 (mayo de 2020), DeBERTa (junio de 2020), Switch Transformers (enero de 2021), Wu Dao 2.0 (junio de 2021), Gopher (diciembre de 2021), GPT-NeoX -20B (febrero de 2022), Chinchilla (marzo de 2022), OPT (mayo de 2022) y el

la lista sigue y sigue. Cada uno de estos modelos aportó nuevas ideas y técnicas, pero a mí me gusta especialmente el papel T5, por investigadores<sup>35</sup> de Google: enumera todas las tareas de PNL como texto a texto, utilizando un transformador codificador-decodificador. Por ejemplo, para traducir "Me gusta el fútbol" al español, puedes simplemente llamar al modelo con la oración de entrada "traducir inglés al español: me gusta el fútbol" y generará "me gusta el fútbol". Para resumir un párrafo, simplemente ingrese "resumir:" seguido del párrafo, y generará el

<sup>34</sup>

<sup>35</sup> de Google: enumera todas las tareas de PNL como texto a texto, utilizando un transformador codificador-decodificador. Por ejemplo, para traducir "Me gusta el fútbol" al español, puedes simplemente llamar al modelo con la oración de entrada "traducir inglés al español: me gusta el fútbol" y generará "me gusta el fútbol". Para resumir un párrafo, simplemente ingrese "resumir:" seguido del párrafo, y generará el

resumen. Para la clasificación, solo necesita cambiar el prefijo a "clasificar:" y el modelo genera el nombre de la clase, como texto. Esto simplifica el uso del modelo y también permite entrenarlo previamente en aún más tareas.

Por último, pero no menos importante, en abril de 2022, los investigadores de Google utilizaron una nueva plataforma de capacitación a gran escala llamada Pathways (que discutiremos brevemente en [el Capítulo 19](#)) para entrenar un modelo de lenguaje enorme llamado [Pathways Language Model \(PaLM\)](#)<sup>36</sup>. con la friolera de 540 mil millones de parámetros, utilizando más de 6000 TPU. Aparte de su increíble tamaño, este modelo es un transformador estándar, que utiliza sólo decodificadores (es decir, con capas de atención enmascaradas de múltiples cabezales), con sólo unos pocos ajustes (consulte el documento para obtener más detalles). Este modelo logró un rendimiento increíble en todo tipo de tareas de PNL, particularmente en comprensión del lenguaje natural (NLU). Es capaz de realizar hazañas impresionantes, como explicar chistes, dar respuestas detalladas paso a paso a preguntas e incluso codificar. Esto se debe en parte al tamaño del modelo, pero también gracias a una técnica llamada [Cadena de pensamientos](#), que fue presentado <sup>37</sup> un par de meses antes por otro equipo de investigadores de Google.

En las tareas de respuesta a preguntas, las indicaciones habituales suelen incluir algunos ejemplos de preguntas y respuestas, como: "P: Roger tiene 5 pelotas de tenis. Compra 2 latas más de pelotas de tenis. Cada lata tiene 3 pelotas de tenis. ¿Cuántas pelotas de tenis tiene ahora? R: 11". Luego, el mensaje continúa con la pregunta real, como "P: John cuida 10 perros. Cada perro dedica 0,5 horas al día a pasear y ocuparse de sus asuntos. ¿Cuántas horas a la semana dedica a cuidar perros? A:", y el trabajo del modelo es agregar la respuesta: en este caso, "35".

Pero con una cadena de pensamiento, las respuestas de ejemplo incluyen todos los pasos de razonamiento que conducen a la conclusión. Por ejemplo, en lugar de "A: 11", el mensaje contiene "A: Roger empezó con 5 bolas. 2 latas de 3 pelotas de tenis cada una son 6 pelotas de tenis.  $5 + 6 = 11$ ". Esto anima al modelo a dar una respuesta detallada a la pregunta real, como por ejemplo "John cuida 10 perros. Cada perro dedica 0,5 horas al día a pasear y ocuparse de sus asuntos. Entonces eso es  $10 \times 0,5 = 5$  horas al día. 5 horas al día  $\times$  7 días a la semana = 35 horas a la semana. La respuesta es 35 horas a la semana". ¡Este es un ejemplo real del artículo!

El modelo no solo da la respuesta correcta con mucha más frecuencia que el uso de indicaciones regulares (alentamos al modelo a pensar las cosas detenidamente), sino que también proporciona todos los pasos de razonamiento, que pueden ser útiles para comprender mejor el fundamento detrás de la respuesta de un modelo. .

Los transformadores se hicieron cargo de la PNL, pero no se detuvieron ahí: pronto se expandieron también a la visión por computadora.

## Transformadores de visión

Una de las primeras aplicaciones de los mecanismos de atención más allá de NMT fue la generación de títulos de imágenes utilizando la [atención visual](#)<sup>38</sup>: una red neuronal convolucional primero procesa la imagen y genera algunos mapas de características, luego un decodificador RNN equipado con un mecanismo de atención genera el título, una palabra a la vez.

En cada paso de tiempo del decodificador (es decir, cada palabra), el decodificador utiliza el modelo de atención para centrarse sólo en la parte correcta de la imagen. Por ejemplo, en [la Figura 16-12](#), el modelo generó el título "Una mujer está lanzando un frisbee en un parque", y se puede ver en qué parte de la imagen de entrada centró su atención el decodificador cuando estaba a punto de emitir la palabra "frisbee": claramente, la mayor parte de su atención se centró en el frisbee.



Figura 16-12. Atención visual: una imagen de entrada (izquierda) y el enfoque del modelo antes de producir la palabra "frisbee" (derecha)<sup>39</sup>

#### EXPLICACIÓN

Un beneficio adicional de los mecanismos de atención es que facilitan la comprensión de qué llevó al modelo a producir su resultado. A esto se le llama explicabilidad. Puede resultar especialmente útil cuando el modelo comete un error: por ejemplo, si una imagen de un perro caminando en la nieve está etiquetada como "un lobo caminando en la nieve", entonces puedes regresar y comprobar en qué se centró el modelo cuando muestra la palabra "lobo". Es posible que descubras que estaba prestando atención no sólo al perro, sino también a la nieve, lo que sugiere una posible explicación: tal vez la forma en que el modelo aprendió a distinguir a los perros de los lobos es comprobando si hay mucha nieve alrededor. Luego puedes solucionar este problema entrenando el modelo con más imágenes de lobos sin nieve y perros con nieve. Este ejemplo proviene de un excelente [artículo de 2016](#) de Marco Tulio Ribeiro et al. que utiliza un enfoque diferente para la explicabilidad: aprender un modelo interpretable localmente en torno a la predicción de un clasificador.<sup>40</sup>

En algunas aplicaciones, la explicabilidad no es sólo una herramienta para depurar un modelo; puede ser un requisito legal; piense en un sistema que decide si debe concederle un préstamo o no.

Cuando aparecieron los transformadores en 2017 y la gente empezó a experimentar con ellos más allá de la PNL, se utilizaron por primera vez junto con las CNN, sin reemplazarlas. En cambio, los transformadores se usaban generalmente para reemplazar los RNN, por ejemplo, en modelos de subtítulos de imágenes. Los transformadores se volvieron un poco más visuales en un [artículo de 2020](#)<sup>41</sup> por investigadores de Facebook, que propusieron una arquitectura híbrida CNN-transformador para la detección de objetos. Una vez más, la CNN primero procesa las imágenes de entrada y genera un conjunto de mapas de características, luego estos mapas de características se convierten en secuencias y se alimentan a un transformador, que genera predicciones de cuadros delimitadores. Pero nuevamente, la mayor parte del trabajo visual todavía lo realiza la CNN.

Luego, en octubre de 2020, un equipo de investigadores de Google publicó un [artículo](#).<sup>42</sup> que introdujo un modelo de visión totalmente basado en transformadores, llamado transformador de visión (ViT). La idea es sorprendentemente simple: simplemente corte la imagen en pequeños cuadrados de  $16 \times 16$  y trate la secuencia de cuadrados como si fuera una secuencia de representaciones de palabras. Para ser más precisos, los cuadrados primero se aplana en  $16 \times 16 \times 3 =$  vectores de 768 dimensiones (el 3 es para los canales de color RGB), luego estos vectores pasan por una capa lineal que los transforma pero conserva su dimensionalidad. La secuencia resultante de vectores se puede tratar como una secuencia de incrustaciones de palabras: esto significa agregar incrustaciones posicionales y pasar el resultado al transformador. ¡Eso es todo! Este modelo superó los últimos avances en clasificación de imágenes de ImageNet, pero para ser justos, los autores tuvieron que utilizar más de 300 millones de imágenes adicionales para el entrenamiento. Esto tiene sentido ya que los transformadores no tienen tantos sesgos inductivos como las redes neuronales convolucionales, por lo que necesitan datos adicionales solo para aprender cosas que las CNN asumen implícitamente.

### NOTA

Un sesgo inductivo es una suposición implícita hecha por el modelo, debido a su arquitectura. Por ejemplo, los modelos lineales suponen implícitamente que los datos son lineales. Las CNN suponen implícitamente que los patrones aprendidos en un lugar probablemente también serán útiles en otros lugares. Los RNN suponen implícitamente que las entradas están ordenadas y que los tokens recientes son más importantes que los más antiguos. Cuantos más sesgos inductivos tenga un modelo, suponiendo que sean correctos, menos datos de entrenamiento requerirá el modelo. Pero si los supuestos implícitos son incorrectos, entonces el modelo puede funcionar mal incluso si se entrena con un conjunto de datos grande.

Sólo dos meses después, un equipo de investigadores de Facebook publicó [un artículo](#)<sup>43</sup>, que introdujo transformadores de imágenes eficientes en datos (DeiT). Su modelo logró resultados competitivos en ImageNet sin requerir datos adicionales para el entrenamiento. La arquitectura del modelo es prácticamente la misma que la del ViT original, pero los autores utilizaron una técnica de destilación para transferir conocimientos de los modelos CNN de última generación a su modelo.

Luego, en marzo de 2021, DeepMind publicó [un artículo](#) importante, que [introdujo la arquitectura Perceiver](#). Es un transformador multimodal, lo que significa que puedes alimentarlo con texto, imágenes, audio o prácticamente cualquier otra modalidad. Hasta entonces, los transformadores habían estado restringidos a secuencias bastante cortas debido al rendimiento y al cuello de botella de RAM en las capas de atención. Esto excluyó modalidades como el audio o el video y obligó a los investigadores a tratar las imágenes como secuencias de parches, en lugar de secuencias de píxeles. El cuello de botella se debe a la autoatención, donde cada token debe atender a todos los demás tokens: si la secuencia de entrada tiene  $M$  tokens, entonces la capa de atención debe calcular una matriz  $M \times M$ , que puede ser enorme si  $M$  es muy grande. El Perceiver resuelve este problema mejorando gradualmente una representación latente bastante corta de las entradas, compuesta por  $N$  tokens (normalmente sólo unos pocos cientos). (La palabra latente significa oculta o interna). El modelo utiliza únicamente capas de atención cruzada, alimentándolas con la representación latente como consultas y las entradas (posiblemente grandes) como valores. Esto sólo requiere calcular una matriz  $M \times N$ , por lo que la complejidad computacional es lineal con respecto a  $M$ , en lugar de cuadrática. Después de pasar por varias capas de atención cruzada, si todo va bien, la representación latente acaba captando todo lo que importa en las entradas. Los autores también sugirieron compartir los pesos entre capas de atención cruzada consecutivas: si haces eso, entonces el Perceiver se convierte efectivamente en un RNN. De hecho, las capas de atención cruzada compartidas pueden verse como la misma celda de memoria en diferentes pasos de tiempo, y la representación latente corresponde al vector de contexto de la celda. Las mismas entradas se envían repetidamente a la celda de memoria en cada paso de tiempo. ¡Parece que los RNN no están muertos después de todo!

Tan solo un mes después, Mathilde Caron et al. presentó [DINO](#), un impresionante transformador de visión entrenado completamente sin etiquetas, mediante autosupervisión y capaz de realizar una segmentación semántica de alta precisión. El modelo se duplica durante la formación, con una red actuando como profesor y la otra como estudiante. El descenso de gradiente solo afecta al estudiante, mientras que los pesos del profesor son solo un promedio móvil exponencial de los pesos del estudiante. Se entrena al estudiante para que coincida con las predicciones del profesor: como son casi el mismo modelo, esto se llama autodestilación. En cada paso del entrenamiento, las imágenes de entrada se aumentan de diferentes maneras para el profesor y el estudiante, de modo que no vean exactamente la misma imagen, pero sus predicciones deben coincidir. Esto los obliga a presentar representaciones de alto nivel. Para evitar el colapso del modo, donde tanto el estudiante como el profesor siempre producirían lo mismo, ignorando completamente las entradas, DINO realiza un seguimiento de un promedio móvil de las salidas del profesor y modifica las predicciones del profesor para garantizar que permanezcan centradas en cero., de media. DINO también obliga al profesor a tener mucha confianza en sus predicciones: a esto se le llama agudización. Juntas, estas técnicas preservan la diversidad en las producciones del profesor.

En un [artículo de 2021](#), Los investigadores de Google mostraron cómo aumentar o reducir los ViT, según la cantidad de datos. Se las arreglaron para crear un enorme modelo de 2 mil millones de parámetros que alcanzó más del 90,4% de precisión en ImageNet. Por el contrario, también entrenaron un modelo reducido que alcanzó más del 84,8% de precisión en ImageNet, utilizando solo 10.000 imágenes: ¡eso es solo 10 imágenes por clase!

Y el progreso en los transformadores visuales ha continuado de manera constante hasta el día de hoy. Por ejemplo, en marzo de 2022, un <sup>47</sup>[artículo](#) por Mitchell Wortsman et al. demostró que es posible entrenar primero varios transformadores y luego promediar sus pesos para crear un modelo nuevo y mejorado. Esto es similar a un conjunto (ver [Capítulo 7](#)), excepto que al final solo hay un modelo, lo que significa que no hay penalización en el tiempo de inferencia.

La última tendencia en transformadores consiste en construir grandes modelos multimodales, a menudo capaces de realizar un aprendizaje de pocos o cero disparos. Por ejemplo, el documento [CLIP 2021](#) de <sup>48</sup>[OpenAI](#) propuso un modelo de transformador grande preentrenado para unir títulos con imágenes: esta tarea le permite aprender excelentes representaciones de imágenes, y luego el modelo se puede usar directamente para tareas como la clasificación de imágenes utilizando mensajes de texto simples como "una foto de un gato". Poco después, OpenAI anunció [DALL-E](#), capaz de generar imágenes <sup>49</sup>[asombrosas](#) basadas en indicaciones de texto. El [DALL-E 2](#), que genera imágenes de calidad aún mayor <sup>50</sup>[utilizando](#) un modelo de difusión (ver [Capítulo 17](#)).

En abril de 2022, DeepMind publicó el [artículo Flamingo](#), que introdujo <sup>51</sup>[una familia](#) de modelos previamente entrenados en una amplia variedad de tareas en múltiples modalidades, incluidos texto, imágenes y videos. Se puede utilizar un único modelo en tareas muy diferentes, como respuesta a preguntas, subtítulos de imágenes y más. Poco después, en mayo de 2022, DeepMind presentó [GATO](#), un modelo multimodal que se puede utilizar como política para <sup>52</sup>[un agente de aprendizaje por refuerzo](#) (RL se presentará en el [Capítulo 18](#)). El mismo transformador puede chatear contigo, subtítular imágenes, jugar juegos de Atari, controlar brazos robóticos (simulados) y más, todo con "sólo" 1.200 millones de parámetros. ¡Y la aventura continúa!

#### NOTA

Estos asombrosos avances han llevado a algunos investigadores a afirmar que la IA a nivel humano está cerca, que "todo lo que se necesita es escala" y que algunos de estos modelos pueden ser "ligeramente conscientes". Otros señalan que a pesar del asombroso progreso, estos modelos todavía carecen de la confiabilidad y adaptabilidad de la inteligencia humana, de nuestra capacidad de razonar simbólicamente, de generalizar basándose en un solo ejemplo, y más.

Como puedes ver, ¡los transformadores están en todas partes! Y la buena noticia es que, por lo general, no tendrá que implementar transformadores usted mismo, ya que muchos modelos excelentes previamente entrenados están disponibles para descargar a través de TensorFlow Hub o el centro de modelos de Hugging Face. Ya has visto cómo usar un modelo de TF Hub, así que cerremos este capítulo echando un vistazo rápido al ecosistema de Hugging Face.

## Biblioteca de Transformers de Hugging Face

Es imposible hablar de transformadores hoy en día sin mencionar a Hugging Face, una empresa de inteligencia artificial que ha creado todo un ecosistema de herramientas de código abierto fáciles de usar para PNL, visión y más. El componente central de su ecosistema es la biblioteca Transformers, que le permite descargar fácilmente un modelo previamente entrenado, incluido su tokenizador correspondiente, y luego ajustarlo en su propio conjunto de datos, si es necesario. Además, la biblioteca es compatible con TensorFlow, PyTorch y JAX (con la biblioteca Flax).

La forma más sencilla de utilizar la biblioteca Transformers es utilizar la función `transformadores.pipeline()`: simplemente especifica qué tarea deseas, como el análisis de sentimientos, y descarga un modelo predeterminado previamente entrenado, listo para ser utilizado; realmente no podría ser más simple:

[de tubería de importación de transformadores](#)

```
clasificador = canalización("análisis de sentimientos") # muchas otras tareas están disponibles
resultado = clasificador("Los actores fueron muy convincentes.")
```

El resultado es una lista de Python que contiene un diccionario por texto de entrada:

```
>>> resultado
[{'etiqueta': 'POSITIVO', 'puntuación': 0.9998071789741516}]
```

En este ejemplo, el modelo encontró correctamente que la oración es positiva, con alrededor del 99,98% de confianza. Por supuesto, también puedes pasar un lote de oraciones al modelo:

```
>>> clasificador(["Soy de la India.", "Soy de Irak."]) [{"label": "POSITIVO", "puntuación": 0.9896161556243896}, {"label": "NEGATIVO", "puntuación": 0.9811071157455444}]
```

## SESGO Y JUSTICIA

Como sugiere el resultado, este clasificador específico ama a los indios, pero tiene un severo sesgo contra los iraquíes. Puedes probar este código con tu propio país o ciudad. Este sesgo indeseable generalmente proviene en gran parte de los propios datos de entrenamiento: en este caso, había muchas frases negativas relacionadas con las guerras en Irak en los datos de entrenamiento. Este sesgo se amplificó luego durante el proceso de ajuste, ya que el modelo se vio obligado a elegir entre sólo dos clases: positiva o negativa. Si se agrega una clase neutral al realizar ajustes, entonces el sesgo hacia el país desaparece en su mayor parte. Pero los datos de entrenamiento no son la única fuente de sesgo: la arquitectura del modelo, el tipo de pérdida o regularización utilizada para el entrenamiento, el optimizador; todo esto puede afectar lo que el modelo termina aprendiendo. Incluso un modelo mayoritariamente imparcial se puede utilizar de forma sesgada, de la misma manera que las preguntas de una encuesta pueden ser sesgadas.

Comprender el sesgo en la IA y mitigar sus efectos negativos sigue siendo un área de investigación activa, pero una cosa es segura: debes hacer una pausa y pensar antes de apresurarte a implementar un modelo en producción. Pregúntese cómo el modelo podría causar daño, aunque sea indirectamente. Por ejemplo, si las predicciones del modelo se utilizan para decidir si conceder o no un préstamo a alguien, el proceso debería ser justo. Por lo tanto, asegúrese de evaluar el rendimiento del modelo no sólo en promedio en todo el conjunto de pruebas, sino también en varios subconjuntos: por ejemplo, puede encontrar que aunque el modelo funciona muy bien en promedio, su rendimiento es abismal para algunas categorías de gente. También es posible que desee ejecutar pruebas contrafactuals: por ejemplo, es posible que desee comprobar que las predicciones del modelo no cambian cuando simplemente cambia el género de alguien.

Si el modelo funciona bien en promedio, es tentador llevarlo a producción y pasar a otra cosa, especialmente si es solo un componente de un sistema mucho más grande. Pero, en general, si usted no soluciona estos problemas, nadie más lo hará y su modelo puede terminar haciendo más daño que bien. La solución depende del problema: puede requerir reequilibrar el conjunto de datos, realizar ajustes en un conjunto de datos diferente, cambiar a otro modelo previamente entrenado, ajustar la arquitectura o los hiperparámetros del modelo, etc.

La función pipeline() utiliza el modelo predeterminado para la tarea dada. Por ejemplo, para tareas de clasificación de texto como el análisis de sentimientos, al momento de escribir este artículo, el valor predeterminado es distilbert-base-uncased-finetuned-sst-2-english: un modelo DistilBERT con un tokenizador sin caja, entrenado en Wikipedia en inglés y un corpus de libros en inglés y afinado en la tarea Stanford Sentiment Treebank v2 (SST 2). También es posible especificar manualmente un modelo diferente. Por ejemplo, podría utilizar un modelo DistilBERT ajustado en la tarea Multi-Genre Natural Language Inference (MultiNLI), que clasifica dos oraciones en tres clases: contradicción, neutral o vinculación. Aquí es cómo:

```
>>> model_name = "huggingface/distilbert-base-uncased-finetuned-mnli" >>> classifier_mnli =
pipeline("clasificación de texto", modelo=model_name) >>> classifier_mnli("Ella me ama. [SEP] Ella ama Yo
no.") [{"label": "contradicción", "puntuación": 0.9790192246437073}]
```

## CONSEJO

Puedes encontrar los modelos disponibles en <https://huggingface.co/models>, y la lista de tareas en <https://huggingface.co/tasks>.

La API de canalización es muy simple y conveniente, pero a veces necesitarás más control. Para tales casos, la biblioteca Transformers proporciona muchas clases, incluidos todo tipo de tokenizadores, modelos, configuraciones, devoluciones de llamadas y mucho más. Por ejemplo, carguemos el mismo modelo DistilBERT, junto con su tokenizador correspondiente, usando las clases `TFAutoModelForSequenceClassification` y `AutoTokenizer`:

```
desde transformadores importe AutoTokenizer, TFAutoModelForSequenceClassification

tokenizer = AutoTokenizer.from_pretrained(nombre_modelo) modelo =
TFAutoModelForSequenceClassification.from_pretrained(nombre_modelo)
```

A continuación, tokenizaremos un par de pares de oraciones. En este código, activamos el relleno y especificamos que queremos tensores de TensorFlow en lugar de listas de Python:

CONSEJO

En lugar de pasar la "Oración 1 [SEP] Oración 2" al tokenizador, puede pasársela de manera equivalente una tupla: ("Oración 1", "Oración 2").

La salida es una instancia similar a un diccionario de la clase BatchEncoding, que contiene las secuencias de ID de tokens, así como una máscara que contiene ceros para los tokens de relleno:

Si configura `return_token_type_ids=True` al llamar al tokenizador, también obtendrá un tensor adicional que indica a qué oración pertenece cada token. Algunos modelos lo necesitan, pero no DistilBERT.

A continuación, podemos pasar directamente este objeto BatchEncoding al modelo; devuelve un Objeto TFSequenceClassifierOutput que contiene sus logits de clase predichos:

```
>>> salidas = modelo(token_ids) >>> salidas
```

TFSequenceClassifierOutput(loss=None, logits=[<tf.Tensor: [...] numpy= array([-2.1123817, 1.4101017], [-0.01478387, 1.0962474 - 0.99199441786793], ...-float32>)], [...])

Por último, podemos aplicar la función de activación softmax para convertir estos logits en probabilidades de clase y usar la función argmax() para predecir la clase con la mayor probabilidad para cada par de oraciones de entrada:

```
>>> Y_probas = tf.keras.activations.softmax(salidas.logits)
>>> Y_probas
<tf.Tensor: forma=(2, 3), dtype=float32, numpy= matriz([[0.01619702
0.43523544 0.54856761
```

```
[0.08672056, 0.85204804, 0.06123142]], dtype=float32)>
>>> Y_pred = tf.argmax(Y_probas, eje=1)
>>> Y_pred # 0 = contradicción, 1 = vinculación, 2 = neutral <tf.Tensor: forma=(2,),
dtype=int64, numpy=array([2, 1])>
```

En este ejemplo, el modelo clasifica correctamente el primer par de oraciones como neutral (el hecho de que a mí me guste el fútbol no implica que a todos los demás les guste) y el segundo par como vinculante (Joe debe ser bastante mayor).

Si desea ajustar este modelo en su propio conjunto de datos, puede entrenar el modelo como de costumbre con Keras, ya que es solo un modelo Keras normal con algunos métodos adicionales. Sin embargo, debido a que el modelo genera logits en lugar de probabilidades, debe usar la pérdida

`tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)` en lugar de la pérdida habitual "sparse\_categorical\_crossentropy". Además, el modelo no admite entradas de BatchEncoding durante el entrenamiento, por lo que debes usar su atributo de datos para obtener un diccionario normal:

```
oraciones = [("El cielo es azul", "El cielo es rojo"), ("La amo", "Ella me ama")]
X_train = tokenzier(sentences, padding=True, return_tensors="tf").data y_train = tf.constant([0, 2]) #
contradicción, pérdida neutral = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
modelo. compilar(pérdida=pérdida, optimizador="nadam", métricas=["precisión"]) historial = model.fit(X_train,
y_train, épocas=2)
```

Hugging Face también ha creado una biblioteca de conjuntos de datos que puede utilizar para descargar fácilmente un conjunto de datos estándar (como IMDb) o uno personalizado, y utilizarlo para ajustar su modelo. Es similar a TensorFlow Datasets, pero también proporciona herramientas para realizar tareas de preprocesamiento comunes sobre la marcha, como el enmascaramiento. La lista de conjuntos de datos está disponible en <https://huggingface.co/datasets>.

Esto debería ayudarte a comenzar con el ecosistema de Hugging Face. Para obtener más información, puedes dirigirte a <https://huggingface.co/docs> para la documentación, que incluye muchos cuadernos de tutoriales, videos, la API completa y más. También te recomiendo que consultes el libro de O'Reilly **Natural Language Processing with Transformers: Building Language Applications with Hugging Face** de Lewis Tunstall, Leandro von Werra y Thomas Wolf, todos del equipo de Hugging Face.

En el próximo capítulo, discutiremos cómo aprender representaciones profundas sin supervisión usando codificadores automáticos, y usaremos redes generativas adversarias para producir imágenes y más.

## Ejercicios

1. ¿Cuáles son las ventajas y desventajas de utilizar un RNN con estado frente a un RNN sin estado?
2. ¿Por qué la gente utiliza RNN codificadores-descodificadores en lugar de RNN simples de secuencia a secuencia para la traducción automática?
3. ¿Cómo se pueden manejar secuencias de entrada de longitud variable? ¿Qué pasa con la salida de longitud variable? secuencias?
4. ¿Qué es la búsqueda por haz y por qué debería utilizarla? ¿Qué herramienta puedes utilizar para implementarlo?
5. ¿Qué es un mecanismo de atención? ¿Cómo ayuda?
6. ¿Cuál es la capa más importante en la arquitectura del transformador? ¿Cuál es su propósito?
7. ¿Cuándo necesitarías utilizar softmax muestrado?
8. Hochreiter y Schmidhuber utilizaron gramáticas integradas de Reber en su artículo. acerca de LSTM. Son gramáticas artificiales que producen cadenas como "BPBTSXXVPSEPE". Verificar

**La bonita introducción** de Jenny Orr. a este tema, luego elija una gramática Reber incorporada particular (como la representada en la página de Orr), luego entrene un RNN para identificar si una cadena respeta esa gramática o no. Primero necesitarás escribir una función capaz de generar un lote de entrenamiento que contenga aproximadamente un 50% de cadenas que respeten la gramática y un 50% que no.

9. Entrene un modelo codificador-decodificador que pueda convertir una cadena de fecha de un formato a otro (p. ej., del “22 de abril de 2019” al “22-04-2019”).

10. Consulte el ejemplo en el sitio web de Keras para “[Búsqueda de imágenes en lenguaje natural con doble Codificador](#)”. Aprenderá a construir un modelo capaz de representar imágenes y texto dentro del mismo espacio de incrustación. Esto hace posible buscar imágenes mediante un mensaje de texto, como en el modelo CLIP de OpenAI.

11. Utilice la biblioteca Hugging Face Transformers para descargar un modelo de lenguaje previamente entrenado capaz de generar texto (por ejemplo, GPT) e intentar generar un texto de Shakespeare más convincente. Deberá utilizar el método `generate()` del modelo; consulte la documentación de Hugging Face para obtener más detalles.

Las soluciones a estos ejercicios están disponibles al final del cuaderno de este capítulo, en <https://homl.info/colab3>.

<sup>1</sup> Alan Turing, “Maquinaria informática e inteligencia”, *Mind* 49 (1950): 433–460.

<sup>2</sup> Por supuesto, la palabra chatbot llegó mucho después. Turing llamó a su prueba el juego de imitación: la máquina A y el humano B conversan con el interrogador humano C a través de mensajes de texto; el interrogador hace preguntas para determinar cuál es la máquina (A o B). La máquina pasa la prueba si puede engañar al interrogador, mientras que el humano B debe intentar ayudar al interrogador.

<sup>3</sup> Dado que las ventanas de entrada se superponen, el concepto de época no es tan claro en este caso: durante cada época (tal como se implementó por Keras), el modelo verá el mismo personaje varias veces.

<sup>4</sup> Alec Radford et al., “Aprender a generar reseñas y descubrir sentimientos”, preimpresión de arXiv arXiv:1704.01444 (2017).

<sup>5</sup> Rico Sennrich et al., “Neural Machine Translation of Rare Words with Subword Units”, Actas de la 54.<sup>a</sup> reunión anual de la Asociación de Lingüística Computacional 1 (2016): 1715–1725.

<sup>6</sup> Taku Kudo, “Regularización de subpalabras: mejora de los modelos de traducción de redes neuronales con múltiples candidatos de subpalabras”, preimpresión de arXiv arXiv:1804.10959 (2018).

<sup>7</sup> Taku Kudo y John Richardson, “SentencePiece: un tokenizador de subpalabras simple e independiente del lenguaje y Detokenizer for Neural Text Processing”, preimpresión de arXiv arXiv:1808.06226 (2018).

<sup>8</sup> Yonghui Wu et al., “Sistema de traducción automática neuronal de Google: uniendo la brecha entre la traducción humana y la automática”, preimpresión de arXiv arXiv:1609.08144 (2016).

<sup>9</sup> Los tensores irregulares se introdujeron en [el Capítulo 12](#) y se detallan en [el Apéndice C](#).

<sup>10</sup> Matthew Peters et al., “Deep Contextualized Word Representations”, Actas de la Conferencia de 2018 del Capítulo Norteamericano de la Asociación de Lingüística Computacional: Tecnologías del Lenguaje Humano 1 (2018): 2227–2237.

<sup>11</sup> Jeremy Howard y Sebastian Ruder, “Universal Language Model Fine-Tuning for Text Classification”, Actas de la 56.<sup>a</sup> Reunión Anual de la Asociación de Lingüística Computacional 1 (2018): 328–339.

<sup>12</sup> Daniel Cer et al., “Universal Sentence Encoder”, preimpresión de arXiv arXiv:1803.11175 (2018).

<sup>13</sup> Ilya Sutskever et al., “Sequence to Sequence Learning with Neural Networks”, preimpresión de arXiv (2014).

<sup>14</sup> Samy Bengio et al., “Muestreo programado para predicción de secuencias con redes neuronales recurrentes”, preimpresión de arXiv arXiv:1506.03099 (2015).

<sup>15</sup> Este conjunto de datos está compuesto por pares de oraciones creados por contribuyentes del [proyecto Tatoeba](#). Los autores del sitio web <https://manythings.org/anki> seleccionaron alrededor de 120.000 pares de frases . Este conjunto de datos se publica bajo la licencia Creative Commons Attribution 2.0 Francia. Hay otros pares de idiomas disponibles.

<sup>16</sup> En Python, si ejecuta `a, *b = [1, 2, 3, 4]`, entonces `a` es igual a 1 y `b` es igual a [2, 3, 4].

<sup>17</sup> Sébastien Jean et al., “On Used Very Large Target Vocabulary for Neural Machine Translation”, Actas de la 53<sup>a</sup> Reunión Anual de la Asociación de Lingüística Computacional y la 7<sup>a</sup> Conferencia Conjunta Internacional sobre

Procesamiento del lenguaje natural de la Federación Asiática de Procesamiento del Lenguaje Natural 1 (2015): 1–10.

18 Dzmitry Bahdanau et al., "Traducción automática neuronal mediante el aprendizaje conjunto de alineación y traducción", preimpresión de arXiv arXiv:1409.0473 (2014).

19 Minh-Thang Luong et al., "Enfoques efectivos para la traducción automática neuronal basada en la atención", Actas del Conferencia de 2015 sobre métodos empíricos en el procesamiento del lenguaje natural (2015): 1412–1421.

20 Ashish Vaswani et al., "Attention Is All You Need", Actas de la 31<sup>a</sup> Conferencia Internacional sobre Sistemas de Procesamiento de Información Neural (2017): 6000–6010.

21 Dado que el transformador utiliza capas densas distribuidas en el tiempo, se podría argumentar que utiliza capas convolucionales 1D con una tamaño de grano de 1.

22 Esta es la figura 1 del artículo "La atención es todo lo que necesitas", reproducida con el amable permiso de los autores.

23 Es posible utilizar tensores irregulares en su lugar, si está utilizando la última versión de TensorFlow.

24 Esta es la parte derecha de la figura 2 de "Atención es todo lo que necesitas", reproducida con la amable autorización del autores.

25 Lo más probable es que esto haya cambiado cuando leas esto; consulte el número 16248 de Keras para más detalles. Cuando esto suceda, no será necesario establecer el argumento atencion\_mask y, por lo tanto, no será necesario crear encoder\_pad\_mask.

26 Actualmente Z + skip no admite el enmascaramiento automático, por eso tuvimos que escribir tf.keras.layers.Add()  
([Z, saltar]) en su lugar. Nuevamente, esto puede cambiar cuando lea esto.

27 Alec Radford et al., "Mejorar la comprensión del lenguaje mediante el entrenamiento previo generativo" (2018).

28 Por ejemplo, la oración "Jane se divirtió mucho en la fiesta de cumpleaños de su amiga" implica "Jane disfrutó la fiesta", pero se contradice con "Todos odiaban el partido" y no tiene relación con "La Tierra es plana".

29 Jacob Devlin et al., "BERT: Entrenamiento previo de transformadores bidireccionales profundos para la comprensión del lenguaje", Actas de la Conferencia de 2018 del Capítulo Norteamericano de la Asociación de Lingüística Computacional: Tecnologías del Lenguaje Humano 1 (2019).

30 Esta es la figura 1 del artículo, reproducida con la amable autorización de los autores.

31 Alec Radford et al., "Los modelos lingüísticos son estudiantes mult tarea sin supervisión" (2019).

32 William Fedus et al., "Transformadores de conmutación: escalamiento a modelos de billones de parámetros con dispersión simple y eficiente" (2021).

33 Victor Sanh et al., "DistilBERT, A Distilled Version of Bert: Smaller, Faster, Cheaper and Lighter", preimpresión de arXiv arXiv:1910.01108 (2019).

34 Mariya Yao resumió muchos de estos modelos en esta publicación: <https://homl.info/yaopost>.

35 Colin Raffel et al., "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer", preimpresión de arXiv arXiv:1910.10683 (2019).

36 Aakanksha Chowdhery et al., "PaLM: Scaling Language Modeling with Pathways", preimpresión de arXiv arXiv:2204.02311 (2022).

37 Jason Wei et al., "Chain of Thought Prompting Elicits Reasoning in Large Language Models", preimpresión de arXiv arXiv:2201.11903 (2022).

38 Kelvin Xu et al., "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", Actas del 32<sup>a</sup> Conferencia Internacional sobre Aprendizaje Automático (2015): 2048–2057.

39 Esta es una parte de la figura 3 del artículo. Se reproduce con la amable autorización de los autores.

40 Marco Tulio Ribeiro et al., "¿Por qué debería confiar en usted?": Explicando las predicciones de cualquier clasificador", Actas de la 22<sup>a</sup> Conferencia Internacional ACM SIGKDD sobre Descubrimiento de Conocimiento y Minería de Datos (2016): 1135–1144.

41 Nicolas Carion et al., "Detección de objetos de extremo a extremo con transformadores", preimpresión de arXiv arxiv:2005.12872 (2020).

42 Alexey Dosovitskiy et al., "Una imagen vale 16 x 16 palabras: transformadores para el reconocimiento de imágenes a escala", preimpresión de arXiv arxiv:2010.11929 (2020).

43 Hugo Touvron et al., "Training Data-Efficient Image Transformers & Distillation Through Attention", preimpresión de arXiv arxiv:2012.12877 (2020).

44 Andrew Jaegle et al., "Perceiver: General Perception with Iterative Attention", preimpresión de arXiv arxiv:2103.03206 (2021).

45 Mathilde Caron et al., "Propiedades emergentes en transformadores de visión autosupervisados", preimpresión de arXiv arxiv:2104.14294 (2021).

- [46](#) Xiaohua Zhai et al., "Scaling Vision Transformers", preimpresión de arXiv arxiv:2106.04560v1 (2021).
- [47](#) Mitchell Wortsman et al., "Sopas de modelos: el promedio de pesos de múltiples modelos ajustados mejora la precisión sin Increasing Inference Time", preimpresión de arXiv arxiv:2203.05482v1 (2022).
- [48](#) Alec Radford et al., "Aprendizaje de modelos visuales transferibles a partir de la supervisión del lenguaje natural", preimpresión de arXiv arxiv:2103.00020 (2021).
- [49](#) Aditya Ramesh et al., "Zero-Shot Text-to-Image Generation", preimpresión de arXiv arxiv:2102.12092 (2021).
- [50](#) Aditya Ramesh et al., "Generación de imágenes condicionales de texto jerárquico con CLIP Latents", preimpresión de arXiv arxiv:2204.06125 (2022).
- [51](#) Jean-Baptiste Alayrac et al., "Flamingo: a Visual Language Model for Few-Shot Learning", preimpresión de arXiv arxiv:2204.14198 (2022).
- [52](#) Scott Reed et al., "A Generalist Agent", preimpresión de arXiv arxiv:2205.06175 (2022).

# Capítulo 17. Autocodificadores, GAN y modelos de difusión

---

Los codificadores automáticos son redes neuronales artificiales capaces de aprender representaciones densas de los datos de entrada, llamadas representaciones latentes o codificaciones, sin ninguna supervisión (es decir, el conjunto de entrenamiento no está etiquetado). Estas codificaciones suelen tener una dimensionalidad mucho más baja que los datos de entrada, lo que hace que los codificadores automáticos sean útiles para la reducción de dimensionalidad (consulte el Capítulo 8), especialmente para fines de visualización. Los codificadores automáticos también actúan como detectores de características y pueden usarse para el preentrenamiento no supervisado de redes neuronales profundas (como analizamos en el Capítulo 11). Por último, algunos codificadores automáticos son modelos generativos: son capaces de generar aleatoriamente nuevos datos que se parecen mucho a los datos de entrenamiento. Por ejemplo, podría entrenar un codificador automático con imágenes de caras y luego podría generar caras nuevas.

Las redes generativas adversarias (GAN) también son redes neuronales capaces de generar datos. De hecho, pueden generar imágenes de rostros tan convincentes que resulta difícil creer que las personas que representan no existen.

Puedes juzgarlo por ti mismo visitando <https://thispersondoesnotexist.com>, un sitio web que muestra rostros generados por una arquitectura GAN llamada StyleGAN. También puedes consultar <https://thisrentaldoesnotexist.com> para ver algunos listados generados en Airbnb.

Las GAN ahora se usan ampliamente para super resolución (aumentando la resolución de una imagen), coloración, potente edición de imágenes (por ejemplo, reemplazar fotobombarderos con fondos realistas), convertir bocetos simples en imágenes fotorrealistas, predecir los siguientes cuadros en un video, aumentar un conjunto de datos (para entrenar otros modelos), generar otros tipos de datos (como texto, audio, y series de tiempo), identificando las debilidades de otros modelos para fortalecerlos, y más.

Una incorporación más reciente al grupo del aprendizaje generativo son los modelos de difusión. En 2021, lograron generar más diversidad y mayor-

imágenes de calidad que las GAN, y al mismo tiempo son mucho más fáciles de entrenar.

Sin embargo, los modelos de difusión son mucho más lentos de ejecutar.

Los codificadores automáticos, las GAN y los modelos de difusión no están supervisados, todos aprenden representaciones latentes, todos pueden usarse como modelos generativos y tienen muchas aplicaciones similares. Sin embargo, funcionan de manera muy diferente:

- Los codificadores automáticos simplemente aprenden a copiar sus entradas a sus salidas. Esto puede parecer una tarea trivial, pero como verá, restringir la red de varias maneras puede hacerlo bastante difícil. Por ejemplo, puede limitar el tamaño de las representaciones latentes o puede agregar ruido a las entradas y entrenar la red para recuperar las entradas originales.

Estas restricciones impiden que el codificador automático copie trivialmente las entradas directamente en las salidas, lo que lo obliga a aprender formas eficientes de representar los datos. En resumen, las codificaciones son subproductos del autocodificador que aprende la función de identidad bajo algunas restricciones.

- Las GAN se componen de dos redes neuronales: un generador que intenta generar datos similares a los datos de entrenamiento y un discriminador que intenta distinguir los datos reales de los datos falsos. Esta arquitectura es muy original en el aprendizaje profundo en el sentido de que el generador y el discriminador compiten entre sí durante el entrenamiento: el generador a menudo se compara con un criminal que intenta hacer dinero falso realista, mientras que el discriminador es como el investigador de la policía que intenta distinguir el dinero real de falso. El entrenamiento adversario (entrenamiento de redes neuronales competitivas) se considera ampliamente una de las innovaciones más importantes de la década de 2010. En 2016, Yann LeCun incluso dijo que era “la idea más interesante de los últimos 10 años en aprendizaje automático”.

- Se entrena un modelo probabilístico de difusión de eliminación de ruido (DDPM) para eliminar una pequeña cantidad de ruido de una imagen. Si luego toma una imagen completamente llena de ruido gaussiano y ejecuta repetidamente el modelo de difusión en esa imagen, emergirá gradualmente una imagen de alta calidad, similar a las imágenes de entrenamiento (pero no idéntica).

En este capítulo comenzaremos explorando con más profundidad cómo funcionan los codificadores automáticos y cómo usarlos para la reducción de dimensionalidad, extracción de características,

preentrenamiento no supervisado, o como modelos generativos. Esto nos llevará naturalmente a las GAN. Construiremos una GAN simple para generar imágenes falsas, pero veremos que el entrenamiento suele ser bastante difícil. Discutiremos las principales dificultades que encontrará con el entrenamiento adversario, así como algunas de las principales técnicas para solucionar estas dificultades. Y, por último, construiremos y entrenaremos un DDPM y lo usaremos para generar imágenes. ¡Empecemos con los codificadores automáticos!

## Representaciones de datos eficientes

¿Cuál de las siguientes secuencias numéricas te parece más fácil de memorizar?

- 40, 27, 25, 36, 81, 57, 10, 73, 19, 68
- 50, 48, 46, 44, 42, 40, 38, 36, 34, 32, 30, 28, 26, 24, 22, 20, 18, 16, 14

A primera vista, parecería que la primera secuencia debería ser más sencilla, ya que es mucho más corta. Sin embargo, si observas detenidamente la segunda secuencia, notarás que es solo la lista de números pares del 50 al 14. Una vez que notes este patrón, la segunda secuencia se vuelve mucho más fácil de memorizar que la primera porque solo necesitas recordar el patrón (es decir, números pares decrecientes) y los números iniciales y finales (es decir, 50 y 14). Tenga en cuenta que si pudiera memorizar rápida y fácilmente secuencias muy largas, no le importaría mucho la existencia de un patrón en la segunda secuencia. Simplemente aprenderías cada número de memoria y eso sería todo. El hecho de que sea difícil memorizar secuencias largas es lo que hace que sea útil reconocer patrones y, con suerte, esto aclara por qué restringir un codificador automático durante el entrenamiento lo empuja a descubrir y explotar patrones en los datos.

**William Chase y Herbert Simon** estudiaron la relación entre memoria, percepción y coincidencia de patrones. a principios de los años 1970. Observaron<sup>1</sup> que los jugadores de ajedrez expertos eran capaces de memorizar las posiciones de todas las piezas de una partida con sólo mirar el tablero durante cinco segundos, una tarea que a la mayoría de la gente le resultaría imposible. Sin embargo, esto sólo era así cuando las piezas se colocaban en posiciones realistas.

(de juegos reales), no cuando las piezas se colocaron al azar. Los expertos en ajedrez no tienen mucha mejor memoria que tú y yo; simplemente ven los patrones de ajedrez más fácilmente gracias a su experiencia con el juego.

Notar patrones les ayuda a almacenar información de manera eficiente.

Al igual que los jugadores de ajedrez en este experimento de memoria, un codificador automático analiza las entradas, las convierte en una representación latente eficiente y luego escupe algo que (con suerte) se parece mucho a las entradas. Un codificador automático siempre se compone de dos partes: un codificador (o red de reconocimiento) que convierte las entradas en una representación latente, seguido de un decodificador (o red generativa) que convierte la representación interna en salidas (consulte la Figura 17-1).

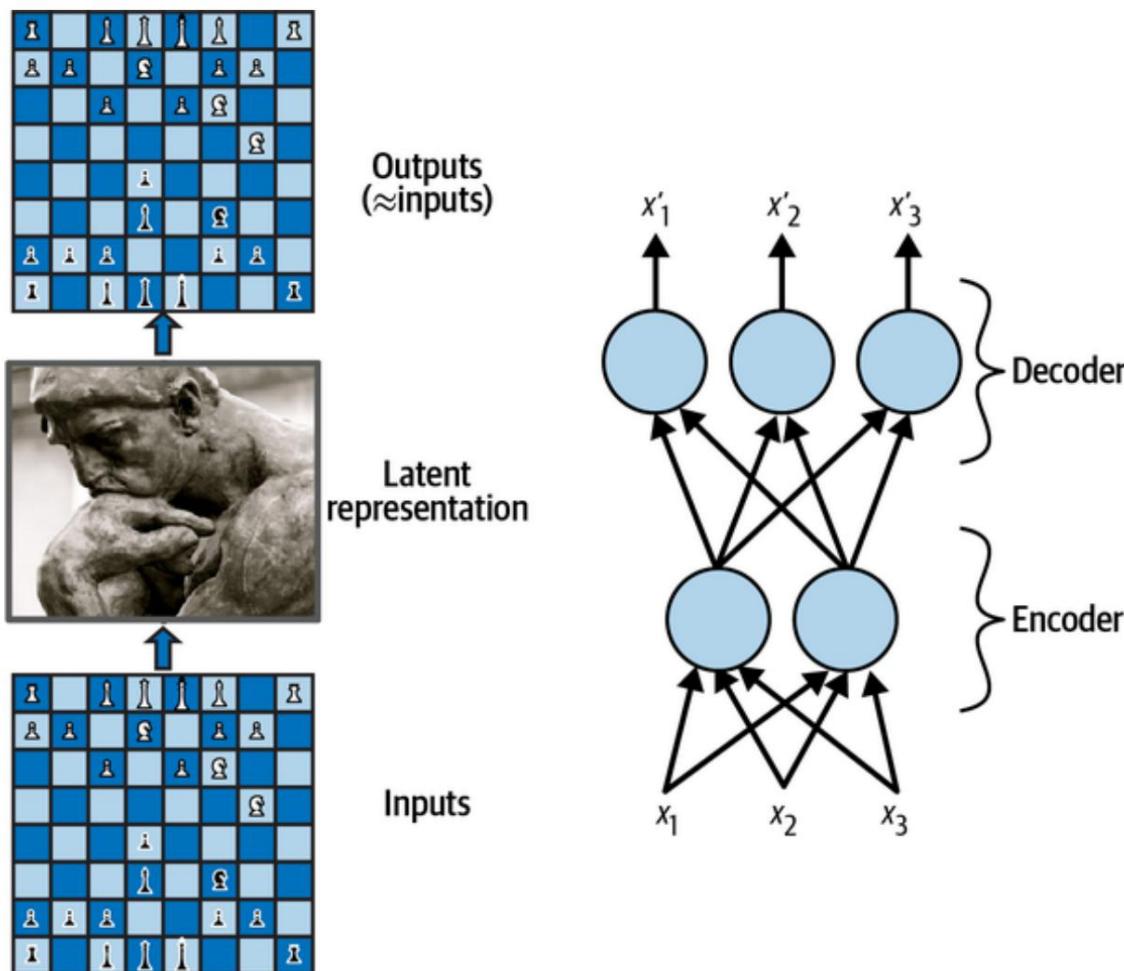


Figura 17-1. El experimento de la memoria de ajedrez (izquierda) y un codificador automático simple (derecha)

Como puede ver, un codificador automático normalmente tiene la misma arquitectura que un perceptrón multicapa (MLP; consulte [el Capítulo 10](#)), excepto que el número de

Las neuronas en la capa de salida deben ser iguales al número de entradas. En este ejemplo, hay solo una capa oculta compuesta por dos neuronas (el codificador) y una capa de salida compuesta por tres neuronas (el decodificador).

Las salidas a menudo se denominan reconstrucciones porque el codificador automático intenta reconstruir las entradas. La función de costo contiene una pérdida de reconstrucción que penaliza al modelo cuando las reconstrucciones son diferentes de las entradas.

Debido a que la representación interna tiene una dimensionalidad menor que los datos de entrada (es 2D en lugar de 3D), se dice que el codificador automático está incompleto. Un codificador automático incompleto no puede copiar trivialmente sus entradas a las codificaciones, pero debe encontrar una manera de generar una copia de sus entradas. Se ve obligado a aprender las características más importantes de los datos de entrada (y descartar las que no son importantes).

Veamos cómo implementar un codificador automático subcompleto muy simple para reducir la dimensionalidad.

## Realizar PCA con un lineal incompleto codificador automático

Si el codificador automático usa solo activaciones lineales y la función de costo es el error cuadrático medio (MSE), entonces termina realizando un análisis de componentes principales (PCA; consulte el Capítulo 8).

El siguiente código crea un codificador automático lineal simple para realizar PCA en un conjunto de datos 3D, proyectándolo en 2D:

```
import tensorflow como tf

codificador = tf.keras.Sequential([tf.keras.layers.Dense(2)]) decodificador =
tf.keras.Sequential([tf.keras.layers.Dense(3)]) autoencoder =
tf.keras.Sequential( [codificador, decodificador])

optimizador = tf.keras.optimizers.SGD(learning_rate=0.5)
autoencoder.compile(loss="mse", optimizador=optimizador)
```

En realidad, este código no es muy diferente de todos los MLP que creamos en capítulos anteriores, pero hay algunas cosas a tener en cuenta:

- Organizamos el codificador automático en dos subcomponentes: el codificador y el decodificador. Ambos son modelos secuenciales regulares con una única capa Densa cada uno, y el codificador automático es un modelo secuencial que contiene el codificador seguido del decodificador (recuerde que un modelo se puede usar como capa en otro modelo).
- El número de salidas del codificador automático es igual al número de entradas (es decir, 3).
- Para realizar PCA, no utilizamos ninguna función de activación (es decir, todas las neuronas son lineales) y la función de costo es el MSE. Esto se debe a que PCA es una transformación lineal. En breve veremos codificadores automáticos más complejos y no lineales.

Ahora entrenemos el modelo en el mismo conjunto de datos 3D generado simple que usamos en [el Capítulo 8](#) y usémoslo para codificar ese conjunto de datos (es decir, proyectarlo en 2D):

```
X_train = [...] # generar un conjunto de datos 3D, como en el Capítulo 8
historia = autoencoder.fit(X_train, X_train, epochs=500, verbose=False)
codings =
encoder.predict(X_train)
```

Tenga en cuenta que `X_train` se utiliza tanto como entradas como como objetivos. La Figura 17-2 muestra el conjunto de datos 3D original (a la izquierda) y la salida de la capa oculta del codificador automático (es decir, la capa de codificación, a la derecha). Como puede ver, el codificador automático encontró el mejor plano 2D para proyectar los datos, preservando tanta variación en los datos como pudo (al igual que PCA).

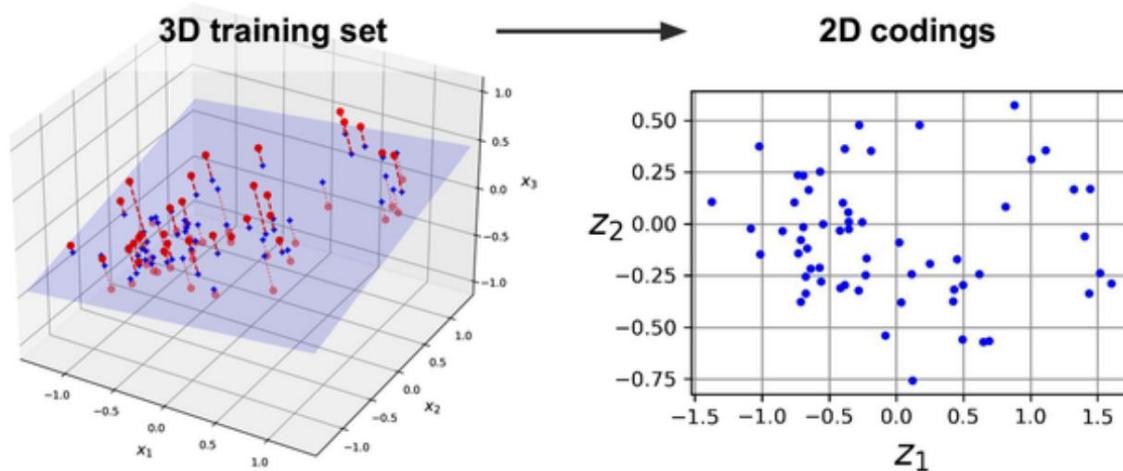


Figura 17-2. PCA aproximada realizada por un codificador automático lineal incompleto

## NOTA

Puede pensar que un codificador automático realiza una forma de aprendizaje autosupervisado, ya que se basa en una técnica de aprendizaje supervisado con etiquetas generadas automáticamente (en este caso, simplemente iguales a las entradas).

### Codificadores automáticos apilados

Al igual que otras redes neuronales que hemos analizado, los codificadores automáticos pueden tener varias capas ocultas. En este caso se denominan codificadores automáticos apilados (o codificadores automáticos profundos). Agregar más capas ayuda al codificador automático a aprender codificaciones más complejas. Dicho esto, hay que tener cuidado de no hacer que el codificador automático sea demasiado potente. Imagine un codificador tan poderoso que simplemente aprende a asignar cada entrada a un único número arbitrario (y el decodificador aprende el mapeo inverso). Obviamente, un codificador automático de este tipo reconstruirá los datos de entrenamiento perfectamente, pero no habrá aprendido ninguna representación de datos útil en el proceso y es poco probable que se generalice bien a nuevas instancias.

La arquitectura de un codificador automático apilado suele ser simétrica con respecto a la capa oculta central (la capa de codificación). En pocas palabras, parece un sándwich. Por ejemplo, un codificador automático para Fashion MNIST (presentado en [el Capítulo 10](#)) puede tener 784 entradas, seguidas de una capa oculta con 100 neuronas, luego una capa oculta central de 30 neuronas, luego otra capa oculta con 100 neuronas y una capa de salida con 784 neuronas. Este codificador automático apilado se representa en [la Figura 17-3](#).

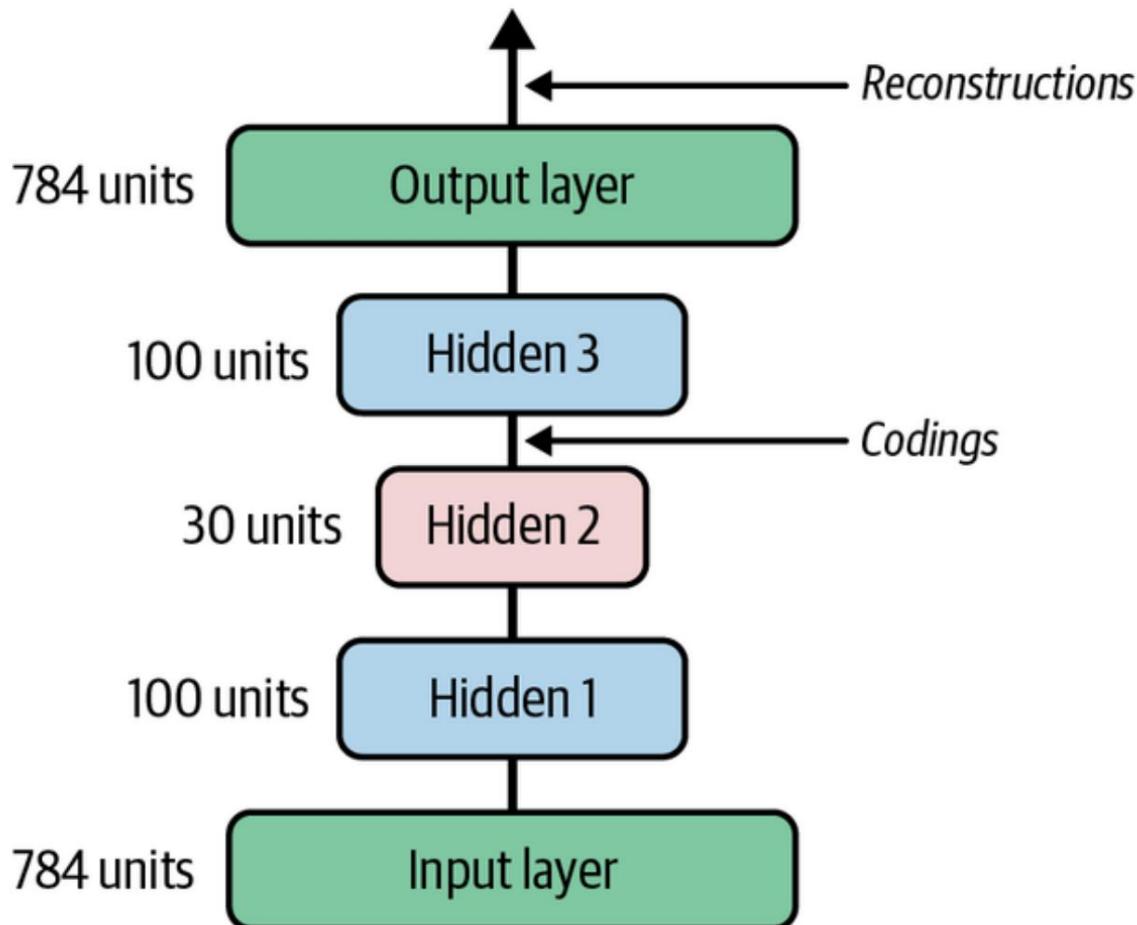


Figura 17-3. Codificador automático apilado

## Implementación de un codificador automático apilado usando Keras

Puede implementar un codificador automático apilado de forma muy parecida a un codificador profundo normal. MLP:

```

stacked_encoder =
    tf.keras.Sequential([
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(100, activation="relu"),
        tf.keras.layers.Dense(30, activation="relu")
    ])
stacked_decoder =
    tf.keras.Sequential([
        tf.keras.layers.Dense(100, activation="relu"),
        tf.keras.layers.Dense(28 * 28),
        tf.keras.layers.Reshape([28, 28])
    ])
stacked_ae = tf.keras.Sequential([stacked_encoder, stacked_decoder])
  
```

```
stacked_ae.compile(loss="mse", optimizador="nadam") historial =
stacked_ae.fit(X_train, X_train, épocas=20,
                datos_validación=(X_valido, X_valido))
```

Repasemos este código:

- Al igual que antes, dividimos el modelo de codificador automático en dos submodelos: el codificador y el decodificador.
- El codificador toma imágenes en escala de grises de  $28 \times 28$  píxeles, las aplana para que cada imagen se represente como un vector de tamaño 784, luego procesa estos vectores a través de dos capas densas de tamaños decrecientes (100 unidades y luego 30 unidades), ambas usando la activación ReLU. función. Para cada imagen de entrada, el codificador genera un vector de tamaño 30.
- El decodificador toma codificaciones de tamaño 30 (salida del codificador) y las procesa a través de dos capas densas de tamaños crecientes (100 unidades y luego 784 unidades), y reforma los vectores finales en matrices de  $28 \times 28$  para que las salidas del decodificador tengan la misma forma. como entradas del codificador.
- Al compilar el codificador automático apilado, utilizamos la pérdida MSE y la optimización Nadam.
- Finalmente, entrenamos el modelo usando X\_train como entradas y objetivos. De manera similar, usamos X\_valid como entradas y objetivos de validación.

## Visualizando las reconstrucciones

Una forma de garantizar que un codificador automático esté entrenado adecuadamente es comparar las entradas y las salidas: las diferencias no deben ser demasiado significativas. Tracemos algunas imágenes del conjunto de validación, así como sus reconstrucciones:

```
importar numpy como np
```

```
def plot_reconstructions(modelo, imágenes=X_valid, n_images=5):
```

```

reconstrucciones = np.clip(model.predict(images[:n_images]), 0, 1) fig =
plt.figure(figsize=(n_images * 1.5, 3)) para image_index in
range(n_images): plt.subplot(2 , n_images, 1 +
image_index) plt.imshow(images[image_index],
cmap="binary") plt.axis("off") plt.subplot(2, n_images, 1 +
n_images +
image_index) plt.imshow(reconstrucciones[ image_index], cmap="binary")
plt.axis("apagado")

plot_reconstructions(stacked_ae) plt.show()

```

La Figura 17-4 muestra las imágenes resultantes.



Figura 17-4. Imágenes originales (arriba) y sus reconstrucciones (abajo)

Las reconstrucciones son reconocibles, pero con demasiada pérdida. Es posible que necesitemos entrenar el modelo por más tiempo, hacer que el codificador y el decodificador sean más profundos, o agrandar las codificaciones. Pero si hacemos que la red sea demasiado poderosa, logrará hacer reconstrucciones perfectas sin haber aprendido ningún patrón útil en los datos. Por ahora, vayamos con este modelo.

## Visualización del conjunto de datos MNIST de moda

Ahora que hemos entrenado un codificador automático apilado, podemos usarlo para reducir la dimensionalidad del conjunto de datos. Para la visualización, esto no da grandes resultados en comparación con otros algoritmos de reducción de dimensionalidad (como los que analizamos en el Capítulo 8), pero una gran ventaja de los codificadores automáticos es que pueden manejar grandes conjuntos de datos con muchas instancias.

y muchas características. Entonces, una estrategia es usar un codificador automático para reducir la dimensionalidad a un nivel razonable y luego usar otro algoritmo de reducción de dimensionalidad para la visualización. Usemos esta estrategia para visualizar Fashion MNIST. Primero usaremos el codificador de nuestro codificador automático apilado para reducir la dimensionalidad a 30, luego usaremos la implementación de Scikit-Learn del algoritmo t-SNE para reducir la dimensionalidad a 2 para la visualización:

```
desde sklearn.manifold importar TSNE
```

```
X_valid_compressed = stacked_encoder.predict(X_valid) tsne =  
TSNE(init="pca", learning_rate="auto", random_state=42)  
X_valid_2D = tsne.fit_transform(X_valid_comprimido)
```

Ahora podemos trazar el conjunto de datos:

```
plt.scatter(X_valid_2D[:, 0], X_valid_2D[:, 1], c=y_valid, s=10, cmap="tab10") plt.show()
```

La Figura 17-5 muestra el diagrama de dispersión resultante, un poco embellecido al mostrar algunas de las imágenes. El algoritmo t-SNE identificó varios grupos que coinciden razonablemente bien con las clases (cada clase está representada por un color diferente).

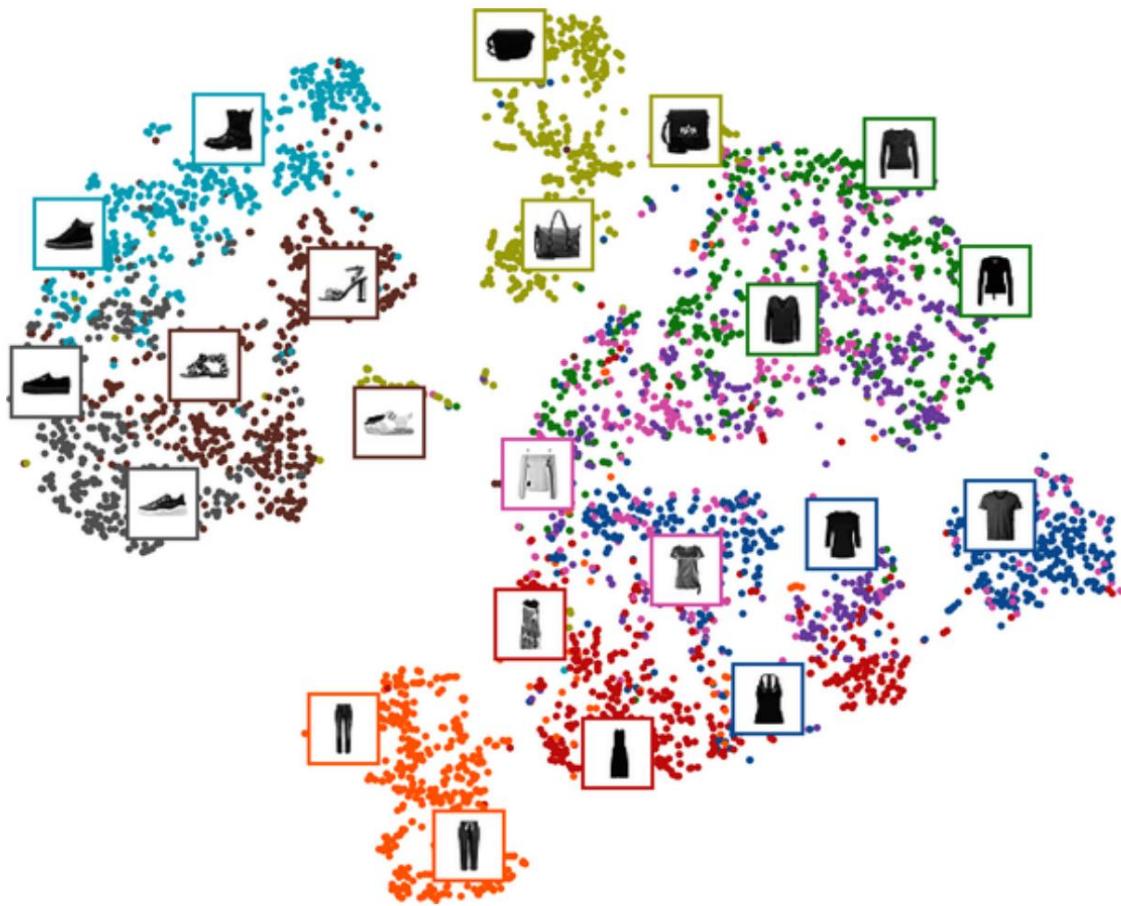


Figura 17-5. Visualización de moda MNIST utilizando un codificador automático seguido de t-SNE

Por tanto, los codificadores automáticos se pueden utilizar para reducir la dimensionalidad. Otra aplicación es para el preentrenamiento no supervisado.

#### Entrenamiento previo no supervisado usando Stacked codificadores automáticos

Como analizamos en [el Capítulo 11](#), si está abordando una tarea supervisada compleja pero no tiene muchos datos de entrenamiento etiquetados, una solución es encontrar una red neuronal que realice una tarea similar y reutilizar sus capas inferiores. Esto hace posible entrenar un modelo de alto rendimiento utilizando pocos datos de entrenamiento porque su red neuronal no tendrá que aprender todas las características de bajo nivel; simplemente reutilizará los detectores de funciones aprendidos por la red existente.

De manera similar, si tiene un conjunto de datos grande pero la mayor parte no está etiquetado, primero puede entrenar un codificador automático apilado usando todos los datos y luego reutilizar el inferior.

capas para crear una red neuronal para su tarea real y entrenarla utilizando los datos etiquetados. Por ejemplo, la Figura 17-6 muestra cómo utilizar un codificador automático apilado para realizar un entrenamiento previo no supervisado para una red neuronal de clasificación. Al entrenar el clasificador, si realmente no tiene muchos datos de entrenamiento etiquetados, es posible que desee congelar las capas previamente entrenadas (al menos las inferiores).

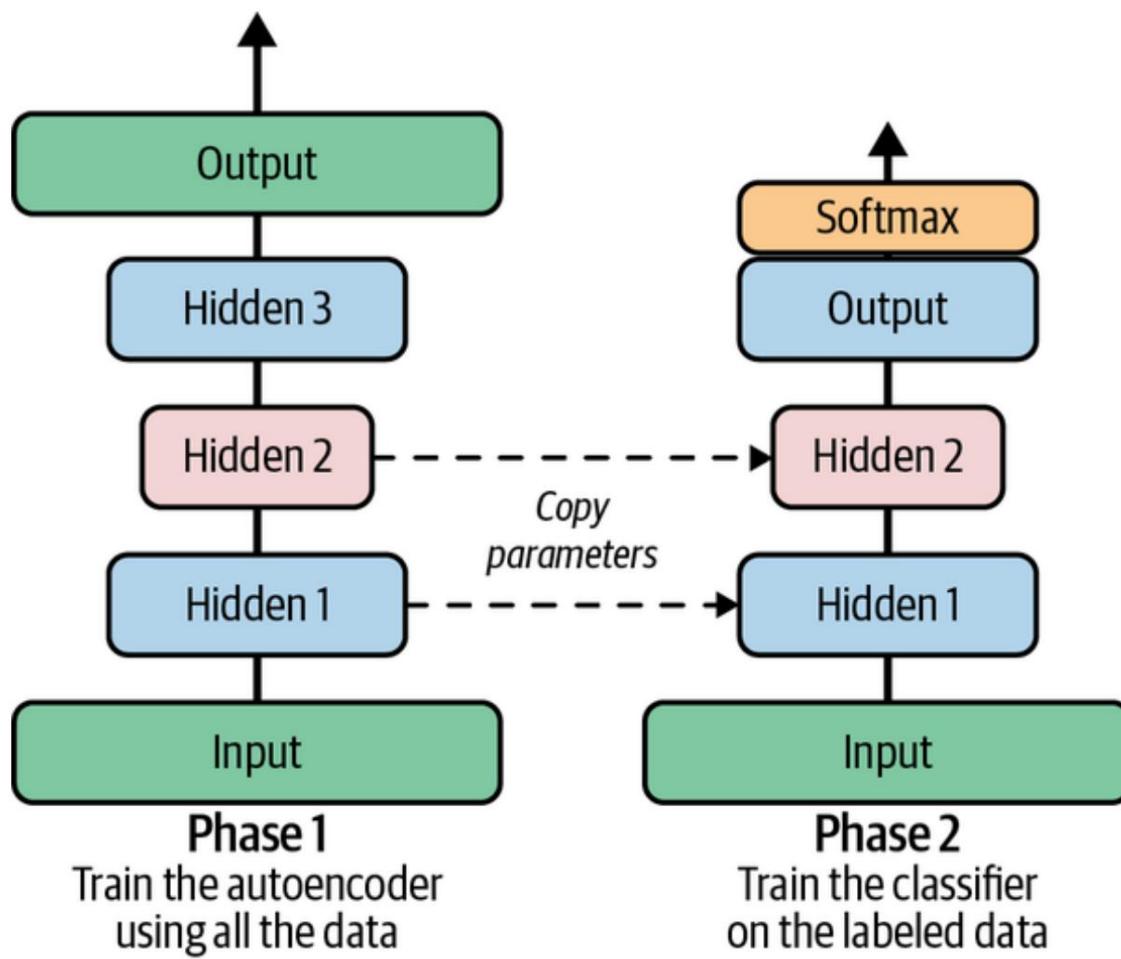


Figura 17-6. Entrenamiento previo no supervisado utilizando codificadores automáticos

## NOTA

Es común tener muchos datos sin etiquetar y pocos datos etiquetados. Construir un gran conjunto de datos sin etiquetar suele ser barato (por ejemplo, un script simple puede descargar millones de imágenes de Internet), pero etiquetar esas imágenes (por ejemplo, clasificarlas como lindas o no) generalmente solo lo pueden hacer de manera confiable los humanos. Etiquetar instancias lleva mucho tiempo y es costoso, por lo que es normal tener solo unos pocos miles de instancias etiquetadas por humanos, o incluso menos.

No hay nada especial en la implementación: simplemente entrene un codificador automático usando todos los datos de entrenamiento (etiquetados y sin etiquetar), luego reutilice sus capas de codificador para crear una nueva red neuronal (consulte los ejercicios al final de este capítulo para ver un ejemplo).

A continuación, veamos algunas técnicas para entrenar codificadores automáticos apilados.

## Atar pesos

Cuando un codificador automático es claramente simétrico, como el que acabamos de construir, una técnica común es vincular los pesos de las capas del decodificador con los pesos de las capas del codificador. Esto reduce a la mitad el número de pesos en el modelo, acelerando el entrenamiento y limitando el riesgo de sobreajuste. Específicamente, si el codificador automático tiene un total de  $N$  capas (sin contar la capa de entrada) y  $W$  representa los pesos de conexión de la capa  $L$  (por ejemplo, la capa  $^{th}1$  es la primera capa oculta, la capa  $N/2$  es la capa de codificación y la capa  $N$  es la capa de salida), entonces los pesos de la capa del decodificador se pueden definir como  $W = W^T$  (con  $L = N / 2 + 1, \dots, N$ ).

$L+1$

Para vincular pesos entre capas usando Keras, definamos una capa personalizada:

```

clase DenseTranspose(tf.keras.layers.Layer): def __init__(self, denso,
activación=None, **kwargs):
    super().__init__(**kwargs)
    self.dense = denso
    self.activación = tf.keras.activations.get(activación)

    def build(self, lote_input_shape): self.biases =
        self.add_weight(nombre="sesgo",
        forma = self.dense.input_shape [-1],
```

```

    inicializador="ceros")
super().build(batch_input_shape)

llamada def (auto, entradas):
    Z = tf.matmul(entradas, self.dense.weights[0], transpose_b=True) devuelve
    self.activation(Z + self.biases)

```

Esta capa personalizada actúa como una capa Densa normal, pero utiliza los pesos de otra capa Densa, transpuesta (establecer transpose\_b=True es equivalente a transponer el segundo argumento, pero es más eficiente ya que realiza la transposición sobre la marcha dentro de la operación matmul() ).

Sin embargo, utiliza su propio vector de sesgo. Ahora podemos construir un nuevo codificador automático apilado, muy parecido al anterior pero con las capas Densa del decodificador unidas a las capas Densa del codificador:

```

denso_1 = tf.keras.layers.Dense(100, activación="relu") denso_2 = tf.keras.layers.Dense(30,
activación="relu")

codificador_atado = tf.keras.Sequential([
    tf.keras.layers.Flatten(), denso_1, denso_2

])

atado_decoder = tf.keras.Sequential([ DenseTranspose(dense_2,
    activación="relu"), DenseTranspose(dense_1), tf.keras.layers.Reshape([28,
    28])]

)

atado_ae = tf.keras.Sequential([tied_encoder, atado_decodificador])

```

Este modelo logra aproximadamente el mismo error de reconstrucción que el modelo anterior, utilizando casi la mitad de parámetros.

## Entrenamiento de un codificador automático a la vez

En lugar de entrenar todo el codificador automático apilado de una vez como acabamos de hacer, es posible entrenar un codificador automático superficial a la vez y luego apilarlos todos en un solo codificador automático apilado (de ahí el nombre), como se muestra en

Figura 17-7. Esta técnica no se usa mucho hoy en día, pero es posible que aún encuentres artículos que hablan sobre "entrenamiento codicioso por capas", por lo que es bueno saber qué significa.

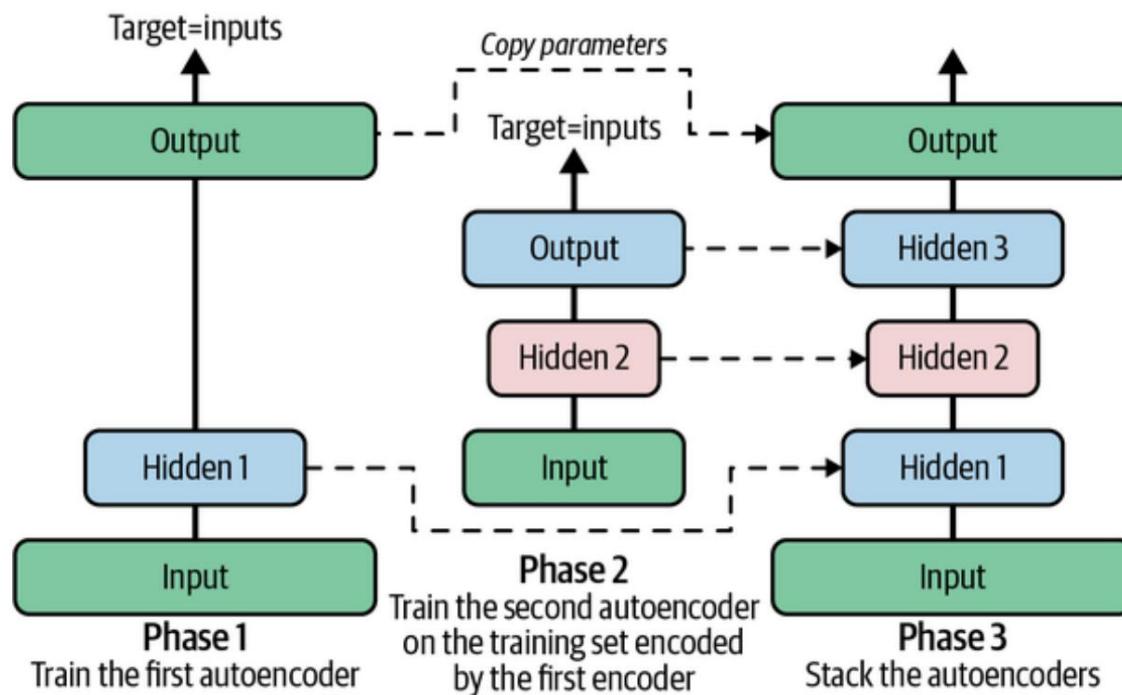


Figura 17-7. Entrenando un codificador automático a la vez

Durante la primera fase del entrenamiento, el primer codificador automático aprende a reconstruir las entradas. Luego codificamos todo el conjunto de entrenamiento usando este primer codificador automático, y esto nos da un nuevo conjunto de entrenamiento (comprimido). Luego entrenamos un segundo codificador automático en este nuevo conjunto de datos. Esta es la segunda fase del entrenamiento. Finalmente, construimos un sándwich grande usando todos estos codificadores automáticos, como se muestra en la Figura 17-7 (es decir, primero apilamos las capas ocultas de cada codificador automático, luego las capas de salida en orden inverso). Esto nos da el codificador automático apilado final (consulte la sección "Entrenamiento de un codificador automático a la vez" en el cuaderno del capítulo para obtener una implementación). Podríamos entrenar fácilmente más codificadores automáticos de esta manera, construyendo un codificador automático muy profundo.

Como mencioné anteriormente, uno de los desencadenantes del tsunami del aprendizaje profundo fue el descubrimiento en 2006 por Geoffrey Hinton et al. que las redes neuronales profundas se pueden entrenar previamente sin supervisión, utilizando este codicioso enfoque por capas. Usaron máquinas Boltzmann restringidas (RBM; ver <https://homl.info/extras-anns>) para este propósito, pero en 2007 Yoshua

Bengio et al.<sup>2</sup> demostró que los codificadores automáticos funcionaban igual de bien.

Durante varios años ésta fue la única forma eficiente de entrenar redes profundas, hasta que muchas de las técnicas introducidas en el [Capítulo 11](#) permitieron entrenar una red profunda de un solo golpe.

Los codificadores automáticos no se limitan a redes densas: también puede crear codificadores automáticos convolucionales. Miremos estos ahora.

## Codificadores automáticos convolucionales

Si se trata de imágenes, los codificadores automáticos que hemos visto hasta ahora no funcionarán bien (a menos que las imágenes sean muy pequeñas): como vio en el [Capítulo 14](#), las redes neuronales convolucionales son mucho más adecuadas que las redes densas para trabajar con imágenes. Entonces, si desea crear un codificador automático para imágenes (por ejemplo, para entrenamiento previo no supervisado o reducción de dimensionalidad), necesitará crear un codificador automático <sup>3</sup> [convolucional](#). El codificador es una CNN normal compuesta por capas convolucionales y capas de agrupación. Por lo general, reduce la dimensionalidad espacial de las entradas (es decir, altura y ancho) al tiempo que aumenta la profundidad (es decir, el número de mapas de características). El decodificador debe hacer lo contrario (aumentar la imagen y reducir su profundidad a las dimensiones originales), y para ello puede usar transponer capas convolucionales (alternativamente, puede combinar capas de muestreo superior con capas convolucionales). Aquí hay un codificador automático convolucional básico para Fashion MNIST:

```
conv_encoder = tf.keras.Sequential([
    tf.keras.layers.Reshape([28, 28, 1]),
    tf.keras.layers.Conv2D(16, 3, padding="mismo", activation="relu"),
    tf.keras.layers.MaxPool2D( pool_size=2), # salida: 14 × 14 x 16 tf.keras.layers.Conv2D(32, 3,
    padding="same", activation="relu"), tf.keras.layers.MaxPool2D(pool_size=2) ,
    # salida: 7 × 7 x
    32
    tf.keras.layers.Conv2D(64, 3, padding="same", activation="relu"),
    tf.keras.layers.MaxPool2D(pool_size=2), # salida: 3 × 3 x
    64
    tf.keras.layers.Conv2D(30, 3, padding="igual",
```

```

activación="relu"),
tf.keras.layers.GlobalAvgPool2D() # salida: 30
])
conv_decoder = tf.keras.Sequential([
    tf.keras.layers.Dense(3 * 3 * 16),
    tf.keras.layers.Reshape((3, 3, 16)),
    tf.keras.layers.Conv2DTranspose(32, 3, zancadas=2, activación= "relu"),

    tf.keras.layers.Conv2DTranspose(16, 3, zancadas=2, padding="igual",
                                  activación="relu"),
    tf.keras.layers.Conv2DTranspose(1, 3, zancadas=2, padding="igual"),
    tf.keras.layers.Reshape([28,
                           28]))
])
conv_ae = tf.keras.Sequential([conv_encoder, conv_decoder])

```

También es posible crear codificadores automáticos con otros tipos de arquitectura, como RNN (consulte el cuaderno para ver un ejemplo).

Bien, retrocedamos un segundo. Hasta ahora hemos analizado varios tipos de codificadores automáticos (básicos, apilados y convolucionales) y cómo entrenarlos (ya sea de una sola vez o capa por capa). También analizamos un par de aplicaciones: visualización de datos y entrenamiento previo no supervisado.

Hasta ahora, para obligar al codificador automático a aprender características interesantes, hemos limitado el tamaño de la capa de codificación, haciéndola incompleta.

En realidad, existen muchos otros tipos de restricciones que se pueden utilizar, incluidas aquellas que permiten que la capa de codificación sea tan grande como las entradas, o incluso más grande, lo que da como resultado un codificador automático sobrecompleto . Luego, en las siguientes secciones veremos algunos tipos más de codificadores automáticos: codificadores automáticos con eliminación de ruido, codificadores automáticos dispersos y codificadores automáticos variacionales.

### Eliminación de ruido de codificadores automáticos

Otra forma de obligar al codificador automático a aprender funciones útiles es agregar ruido a sus entradas, entrenándolo para recuperar las entradas originales sin ruido. Esta idea ha existido desde la década de 1980 (por ejemplo, se menciona en la tesis de maestría de Yann LeCun de 1987).

En un [artículo de 2008](#), Pascal Vicente et al.

demostró que los codificadores automáticos también podrían usarse para la extracción de características. En un [artículo de 2010<sup>5</sup>](#), Vicente y cols. Se introdujeron codificadores automáticos con eliminación de ruido apilados.

El ruido puede ser ruido gaussiano puro agregado a las entradas, o pueden ser entradas apagadas aleatoriamente, como en la desconexión (introducida en el [Capítulo 11](#)). [La Figura 17-8](#) muestra ambas opciones.

La implementación es sencilla: es un codificador automático apilado normal con una capa de abandono adicional aplicada a las entradas del codificador (o podría usar una capa GaussianNoise en su lugar). Recuerde que la capa Dropout solo está activa durante el entrenamiento (al igual que la capa GaussianNoise):

```
dropout_encoder =  
  
    tf.keras.Sequential([ tf.keras.layers.Flatten(),  
        tf.keras.layers.Dropout(0.5), tf.keras.layers.Dense(100,  
        activación="relu"), tf.keras .layers.Dense(30, activación="relu")  
    ])  
dropout_decoder =  
    tf.keras.Sequential([ tf.keras.layers.Dense(100, activación="relu"),  
        tf.keras.layers.Dense(28 * 28),  
        tf.keras.layers.Reshape([ 28, 28])  
    ])  
dropout_ae = tf.keras.Sequential([dropout_encoder, dropout_decoder])
```

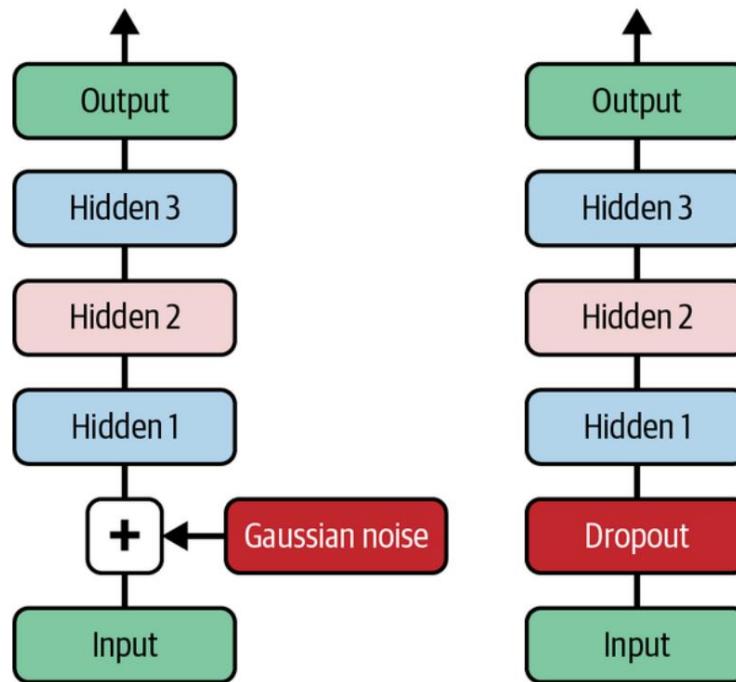


Figura 17-8. Codificadores automáticos con eliminación de ruido, con ruido gaussiano (izquierda) o abandono (derecha)

**La Figura 17-9** muestra algunas imágenes ruidosas (con la mitad de los píxeles desactivados) y las imágenes reconstruidas por el codificador automático de eliminación de ruido basado en eliminación. Observe cómo el codificador automático adivina detalles que en realidad no están en la entrada, como la parte superior de la camisa blanca (fila inferior, cuarta imagen). Como puede ver, los codificadores automáticos de eliminación de ruido no solo se pueden usar para visualización de datos o entrenamiento previo no supervisado, como los otros codificadores automáticos que hemos discutido hasta ahora, sino que también se pueden usar de manera bastante simple y eficiente para eliminar el ruido de las imágenes.



Figura 17-9. Imágenes ruidosas (arriba) y sus reconstrucciones (abajo)

## Codificadores automáticos dispersos

Otro tipo de restricción que a menudo conduce a una buena extracción de características es la escasez: al agregar un término apropiado a la función de costo, el codificador automático se ve obligado a reducir el número de neuronas activas en la capa de codificación. Por ejemplo, se puede exigir que tenga en promedio sólo un 5% de neuronas significativamente activas en la capa de codificación. Esto obliga al codificador automático a representar cada entrada como una combinación de una pequeña cantidad de activaciones. Como resultado, cada neurona en la capa de codificación normalmente termina representando una característica útil (si pudieras decir sólo unas pocas palabras al mes, probablemente intentarías que valga la pena escucharlas).

Un enfoque simple es usar la función de activación sigmoidea en la capa de codificación (para restringir las codificaciones a valores entre 0 y 1), usar una capa de codificación grande (por ejemplo, con 300 unidades) y agregar algo de regularización de  $\ell$  a las activaciones de la capa de codificación. . El decodificador es simplemente un decodificador normal:

```
sparse_l1_encoder = tf.keras.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(100,
        activation="relu"), tf.keras.layers.Dense(300, activation="sigmoide "),
    tf.keras.layers.ActivityRegularization(l1=1e-4)

])
sparse_l1_decoder = tf.keras.Sequential([
    tf.keras.layers.Dense(100, activation="relu"), tf.keras.layers.Dense(28
        * 28), tf.keras.layers.Reshape([28, 28])

])
sparse_l1_ae = tf.keras.Sequential([sparse_l1_encoder, sparse_l1_decoder])
```

Esta capa ActivityRegularization simplemente devuelve sus entradas, pero como efecto secundario agrega una pérdida de entrenamiento igual a la suma de los valores absolutos de sus entradas. Esto sólo afecta al entrenamiento. De manera equivalente, puede eliminar la capa ActivityRegularization y configurar

`Activity_regularizer=tf.keras.regularizers.l1(1e-4)` en la capa anterior. Esta penalización alentará a la red neuronal a producir codificaciones cercanas a 0, pero como también será penalizada si no reconstruye las entradas correctamente, tendrá que generar al menos algunos valores distintos de cero. Usar la norma  $\ell$  en lugar de la norma  $\ell$  impulsará el

red neuronal para preservar las codificaciones más importantes y al mismo tiempo eliminar las que no son necesarias para la imagen de entrada (en lugar de simplemente reducir todas las codificaciones).

Otro enfoque, que a menudo produce mejores resultados, es medir la escasez real de la capa de codificación en cada iteración de entrenamiento y penalizar el modelo cuando la escasez medida difiere de la escasez objetivo. Lo hacemos calculando la activación promedio de cada neurona en la capa de codificación, durante todo el lote de entrenamiento. El tamaño del lote no debe ser demasiado pequeño, de lo contrario la media no será exacta.

Una vez que tenemos la activación media por neurona, queremos penalizar a las neuronas que son demasiado activas o no lo suficientemente activas agregando una pérdida de escasez a la función de costo. Por ejemplo, si medimos que una neurona tiene una activación promedio de 0,3, pero la escasez objetivo es 0,1, hay que penalizarla para que se active menos. Un enfoque podría ser simplemente agregar el error al cuadrado ( $0,3 - 0,1$ ) a la función de costo<sup>2</sup>, pero en la práctica un mejor enfoque es utilizar la divergencia de Kullback-Leibler (KL) (que se analiza brevemente en el Capítulo 4), que tiene gradientes mucho más **fuertes** que el error cuadrático medio, como se puede ver en la Figura 17-10.

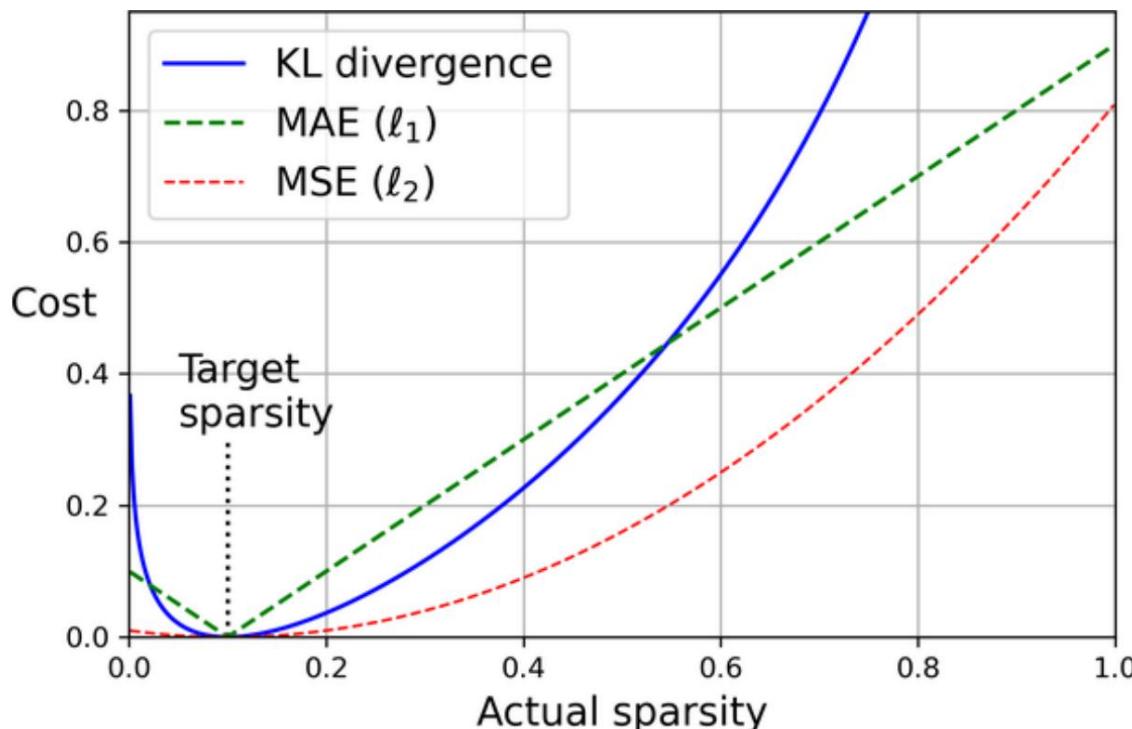


Figura 17-10. Pérdida de escasez

Dadas dos distribuciones de probabilidad discretas P y Q, la divergencia KL entre estas distribuciones, denominada D (P || Q), se puede calcular utilizando la ecuación 17-1.

Ecuación 17-1. Divergencia Kullback-Leibler

$$DKL(P \parallel Q) = \sum_i P(i) \text{ registro} \frac{P(i)}{Q(y)}$$

En nuestro caso, queremos medir la divergencia entre la probabilidad objetivo p de que se active una neurona en la capa de codificación y la probabilidad real q, estimada midiendo la activación media durante el lote de entrenamiento. Entonces, la divergencia KL se simplifica a la ecuación 17-2.

Ecuación 17-2. Divergencia de KL entre la escasez objetivo p y la escasez real q

$$DKL(p \parallel q) = p \log \frac{p}{q} + (1 - p) \log \frac{1 - p}{1 - q}$$

Una vez que hemos calculado la pérdida de escasez para cada neurona en la capa de codificación, sumamos estas pérdidas y sumamos el resultado a la función de costo. Para controlar la importancia relativa de la pérdida de escasez y la pérdida de reconstrucción, podemos multiplicar la pérdida de escasez por un hiperparámetro de peso de escasez. Si este peso es demasiado alto, el modelo se apegará estrechamente a la escasez objetivo, pero es posible que no reconstruya las entradas adecuadamente, lo que hará que el modelo sea inútil. Por el contrario, si es demasiado bajo, el modelo ignorará en gran medida el objetivo de escasez y no aprenderá ninguna característica interesante.

Ahora tenemos todo lo que necesitamos para implementar un codificador automático disperso basado en la divergencia KL. Primero, creemos un regularizador personalizado para aplicar la regularización de divergencia de KL:

```
kl_divergence = tf.keras.losses.kullback_leibler_divergence
clase
KLDivergenceRegularizer(tf.keras.regularizers.Regularizer): def __init__(self, peso,
objetivo): self.weight = peso self.target = objetivo
```

```

def __call__(self, entradas):
    mean_activities = tf.reduce_mean(inputs, axis=0) return self.weight * (
        kl_divergence(self.objetivo, media_actividades) + kl_divergence(1. -
        self.objetivo, 1. - media_actividades))

```

Ahora podemos construir el codificador automático disperso, usando el KLDivergenceRegularizer para las activaciones de la capa de codificación:

```

kld_reg = KLDivergenceRegularizer(peso=5e-3, objetivo=0.1) sparse_kl_encoder =
tf.keras.Sequential([ tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(100,
        activación="relu"), tf.keras.layers.Dense(300, activación="sigmoide",
        actividad_regularizer=kld_reg)

])
sparse_kl_decoder = tf.keras.Sequential([
    tf.keras.layers.Dense(100, activación="relu"), tf.keras.layers.Dense(28
    * 28), tf.keras.layers.Reshape([28, 28])

])
sparse_kl_ae = tf.keras.Sequential([sparse_kl_encoder, sparse_kl_decoder])

```

Después de entrenar este codificador automático disperso en Fashion MNIST, la capa de codificación tendrá aproximadamente un 10 % de escasez.

## Autocodificadores variacionales

Diederik Kingma y Max Welling introdujeron una categoría importante de codificadores automáticos en 2013. y rápidamente se convirtió<sup>6</sup>en una de las variantes más populares: los codificadores automáticos variacionales (VAE).

Los VAE son bastante diferentes de todos los codificadores automáticos que hemos analizado hasta ahora, en estos aspectos particulares:

- Son codificadores automáticos probabilísticos, lo que significa que sus resultados están determinados en parte por el azar, incluso después del entrenamiento (a diferencia de

codificadores automáticos sin ruido, que utilizan la aleatoriedad sólo durante el entrenamiento).

- Lo más importante es que son codificadores automáticos generativos, lo que significa que pueden generar nuevas instancias que parecen tomadas del conjunto de entrenamiento.

Ambas propiedades hacen que los VAE sean bastante similares a los RBM, pero son más fáciles de entrenar y el proceso de muestreo es mucho más rápido (con los RBM es necesario esperar a que la red se estabilice en un "equilibrio térmico" antes de poder muestrear una nueva instancia). . Como sugiere su nombre, los autocodificadores variacionales realizan inferencia bayesiana variacional, que es una forma eficaz de realizar inferencia bayesiana aproximada. Recuerde que la inferencia bayesiana significa actualizar una distribución de probabilidad basada en nuevos datos, utilizando ecuaciones derivadas del teorema de Bayes. La distribución original se llama anterior, mientras que la distribución actualizada se llama posterior. En nuestro caso, queremos encontrar una buena aproximación de la distribución de datos.

Una vez que tengamos eso, podemos probarlo.

Echemos un vistazo a cómo funcionan los VAE. [La Figura 17-11](#) (izquierda) muestra un codificador automático variacional. Puede reconocer la estructura básica de todos los codificadores automáticos, con un codificador seguido de un decodificador (en este ejemplo, ambos tienen dos capas ocultas), pero hay un giro: en lugar de producir directamente una codificación para una entrada determinada, el codificador produce una codificación media  $\mu$  y una desviación estándar  $\sigma$ . Luego, la codificación real se muestrea aleatoriamente a partir de una distribución gaussiana con media  $\mu$  y desviación estándar  $\sigma$ . Después de eso, el decodificador decodifica normalmente la codificación muestreada.

La parte derecha del diagrama muestra una instancia de entrenamiento que pasa por este codificador automático. Primero, el codificador produce  $\mu$  y  $\sigma$ , luego se muestrea una codificación aleatoriamente (obsérvese que no está ubicada exactamente en  $\mu$ ) y finalmente esta codificación se decodifica; el resultado final se parece a la instancia de capacitación.

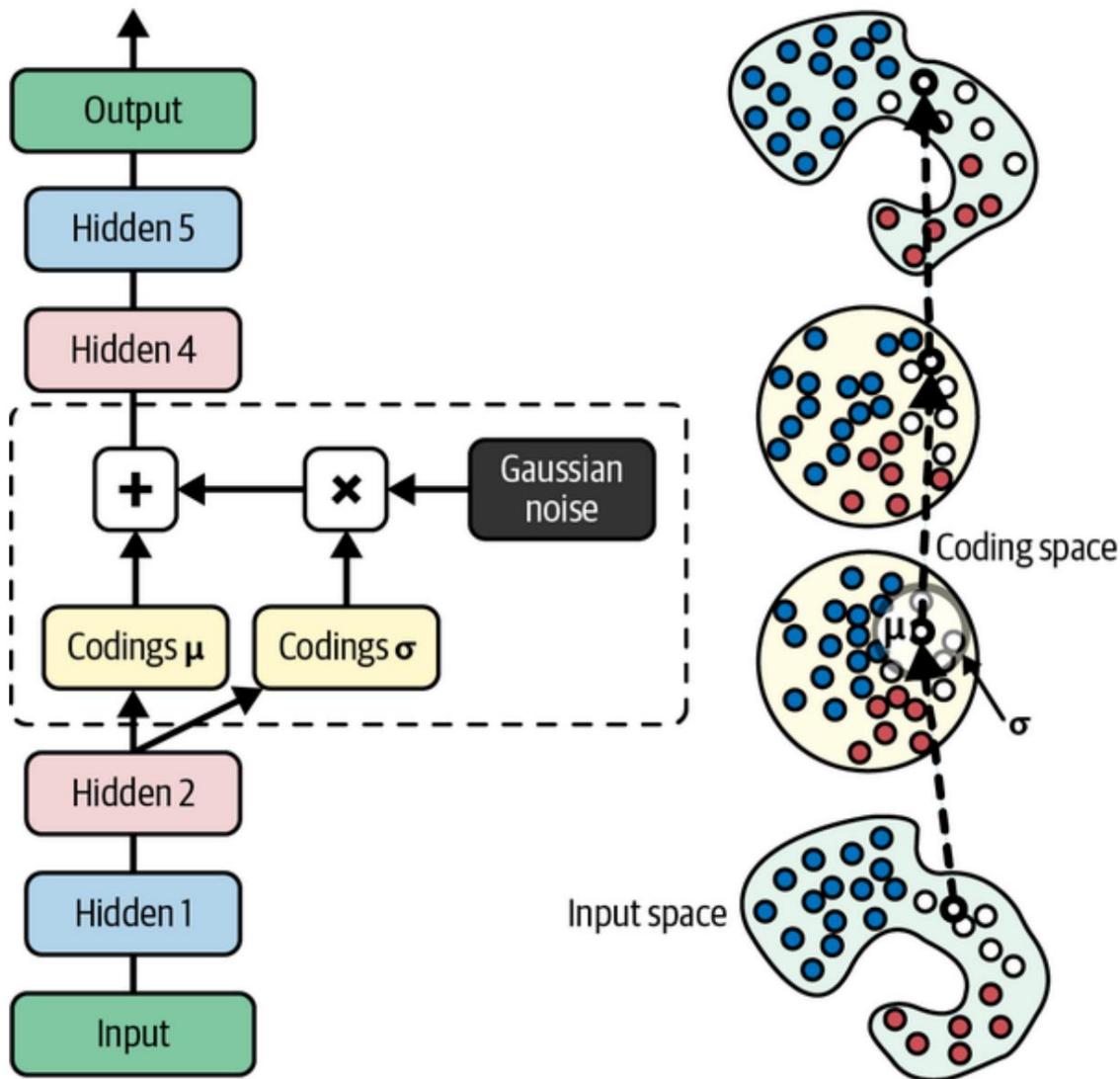


Figura 17-11. Un codificador automático variacional (izquierda) y una instancia que lo atraviesa (derecha)

Como se puede ver en el diagrama, aunque las entradas pueden tener una distribución muy complicada, un codificador automático variacional tiende a producir codificaciones que parecen tomadas de una distribución gaussiana simple: durante el entrenamiento, la función de costo (que se analiza a continuación) empuja las codificaciones migran gradualmente dentro del espacio de codificación (también llamado espacio latente) para terminar pareciendo una nube de puntos gaussianos. Una gran consecuencia es que después de entrenar un codificador automático variacional, puedes generar muy fácilmente una nueva instancia: simplemente toma una muestra de una codificación aleatoria de la distribución gaussiana, decodificala y ¡listo!

Ahora, veamos la función de costo. Está compuesto de dos partes. La primera es la pérdida de reconstrucción habitual que empuja al codificador automático a reproducirse.

sus entradas. Podemos usar el MSE para esto, como hicimos antes. La segunda es la pérdida latente que empuja al codificador automático a tener codificaciones que parecen tomadas de una distribución gaussiana simple: es la divergencia KL entre la distribución objetivo (es decir, la distribución gaussiana) y la distribución real de las codificaciones. Las matemáticas son un poco más complejas que con el codificador automático disperso, en particular debido al ruido gaussiano, que limita la cantidad de información que se puede transmitir a la capa de codificación. Esto empuja al codificador automático a aprender funciones útiles.

Afortunadamente, las ecuaciones se simplifican, por lo que la pérdida latente se puede calcular utilizando [la ecuación 17-3](#).

8

Ecuación 17-3. Pérdida latente del codificador automático variacional.

$$L = -\frac{1}{n} \sum_{i=1}^n [1 + \log(\sigma_i^2) - \sigma_i^{-2} - \mu_i^2]$$

En esta ecuación,  $L$  es la pérdida latente,  $n$  es la dimensionalidad de las codificaciones y  $\mu$  y  $\sigma$  son la media y la desviación estándar del componente  $i$  de las codificaciones. Los vectores  $\mu$  y  $\sigma$  (que contienen todos los  $\mu$  y  $\sigma$ ) son generados por el codificador<sup>th</sup>, como se muestra en [la Figura 17-11](#) (izquierda).

Un ajuste común en la arquitectura del codificador automático variacional es hacer que la salida del codificador sea  $y = \log(\sigma)$  en lugar de  $\sigma$ . Luego, la pérdida latente se puede calcular como se muestra en [la ecuación 17-4](#). Este enfoque es más estable numéricamente y acelera el entrenamiento.

Ecuación 17-4. Pérdida latente del codificador automático variacional, reescrita usando  $y = \log(\sigma^2)$

$$L = -\frac{1}{n} \sum_{i=1}^n [1 + y_i - \exp(y_i) - \mu_i^2]$$

Comencemos a construir un codificador automático variacional para Fashion MNIST (como se muestra en [la Figura 17-11](#), pero usando el ajuste  $y$ ). Primero, necesitaremos una capa personalizada para muestrear las codificaciones, dados  $\mu$  y  $y$ :

```
muestreo de clase (tf.keras.layers.Layer): llamada def (auto,
entradas):
    media, log_var = entradas
```

```
    devolver tf.random.normal(tf.shape(log_var)) * tf.exp(log_var / 2) +
media
```

Esta capa de muestreo toma dos entradas: media ( $\mu$ ) y  $\log_{\text{var}}$  ( $\gamma$ ). Utiliza la función `tf.random.normal()` para muestrear un vector aleatorio (de la misma forma que  $\gamma$ ) de la distribución gaussiana, con media 0 y desviación estándar 1. Luego lo multiplica por  $\exp(\gamma / 2)$  (que es igual a  $\sigma$ , como puedes comprobar matemáticamente), y finalmente suma  $\mu$  y devuelve el resultado. Esto muestra un vector de codificación de la distribución gaussiana con media  $\mu$  y desviación estándar  $\sigma$ .

A continuación, podemos crear el codificador utilizando la API funcional porque el modelo no es completamente secuencial:

```
codificaciones_tamaño = 10

entradas = tf.keras.layers.Input(forma=[28, 28])
Z = tf.keras.layers.Flatten()(entradas)
Z = tf.keras.layers.Dense(150, activación="relu")(Z)
Z = tf.keras.layers.Dense(100, activación="relu")(Z) codings_mean =
tf.keras.layers.Dense(codificaciones_tamaño)(Z) #  $\mu$  codings_log_var =
tf.keras.layers.Dense(codificaciones_tamaño)(Z) # codificaciones  $\gamma$  = muestreo()([codificaciones_media,
codificaciones_log_var]) variacional_encoder = tf.keras.Model(entradas=[entradas],
salidas=[codificaciones_media, codificaciones_log_var,
codificaciones])
```

Tenga en cuenta que las capas densas que generan `codings_mean` ( $\mu$ ) y `codings_log_var` ( $\gamma$ ) tienen las mismas entradas (es decir, las salidas de la segunda capa densa). Luego pasamos tanto `codings_mean` como `codings_log_var` a la capa de muestreo. Finalmente, el modelo `variacional_encoder` tiene tres salidas. Solo se requieren las codificaciones, pero también agregamos `codings_mean` y `codings_log_var`, en caso de que queramos inspeccionar sus valores. Ahora construyamos el decodificador:

```
decoder_inputs = tf.keras.layers.Input(shape=[codificaciones_tamaño]) x = tf.keras.layers.Dense(100,
activación="relu") (decoder_inputs) x = tf.keras.layers.Dense(150, activación
="relu")(x) x =
tf.keras.layers.Dense(28 * 28)(x) salidas = tf.keras.layers.Reshape([28, 28])(x)
```

```
variacional_decodificador = tf.keras.Model(entradas=[decodificador_entradas],
salidas=[salidas])
```

Para este decodificador, podríamos haber usado la API secuencial en lugar de la API funcional, ya que en realidad es solo una simple pila de capas, prácticamente idéntica a muchos de los decodificadores que hemos creado hasta ahora. Finalmente, construyamos el modelo de codificador automático variacional:

```
_ , _ , codificaciones = codificador_variante(entradas)
reconstrucciones = decodificador_variante(codificaciones)
variación_ae = tf.keras.Model(entradas = [entradas], salidas = [reconstrucciones])
```

Ignoramos las dos primeras salidas del codificador (solo queremos enviar las codificaciones al decodificador). Por último hay que sumar la pérdida latente y la pérdida de reconstrucción:

```
pérdida_latente = -0.5 * tf.reduce_sum(
    1 + codificaciones_log_var - tf.exp(codificaciones_log_var) -
    tf.square(codificaciones_media),
    eje=-1)
variación_ae.add_loss(tf.reduce_mean(latent_loss) / 784.)
```

Primero aplicamos la ecuación 17-4 para calcular la pérdida latente para cada instancia del lote, sumando el último eje. Luego calculamos la pérdida media de todas las instancias del lote y dividimos el resultado por 784 para asegurarnos de que tenga la escala adecuada en comparación con la pérdida de reconstrucción.

De hecho, se supone que la pérdida de reconstrucción del codificador automático variacional es la suma de los errores de reconstrucción de píxeles, pero cuando Keras calcula la pérdida "mse", calcula la media de los 784 píxeles, en lugar de la suma. Entonces, la pérdida de reconstrucción es 784 veces menor de lo que necesitamos. Podríamos definir una pérdida personalizada para calcular la suma en lugar de la media, pero es más sencillo dividir la pérdida latente entre 784 (la pérdida final será 784 veces menor de lo que debería ser, pero esto solo significa que debemos usar una pérdida mayor). tasa de aprendizaje).

¡Y finalmente, podemos compilar y ajustar el codificador automático!

```
variación_ae.compile(loss="mse", optimizador="nadam") historial =
variación_ae.fit(X_train, X_train, épocas=25,
```

```
tamaño_lote = 128,
          datos_validación=(X_válido,
X_válido))
```

### Generando imágenes MNIST de moda

Ahora usemos este codificador automático variacional para generar imágenes que parecen artículos de moda. Todo lo que necesitamos hacer es muestrear codificaciones aleatorias de una distribución gaussiana y decodificarlas:

```
codificaciones = tf.random.normal(forma=[3 * 7, codificaciones_tamaño]) imágenes =
variacional_decoder(codificaciones).numpy()
```

La Figura 17-12 muestra las 12 imágenes generadas.

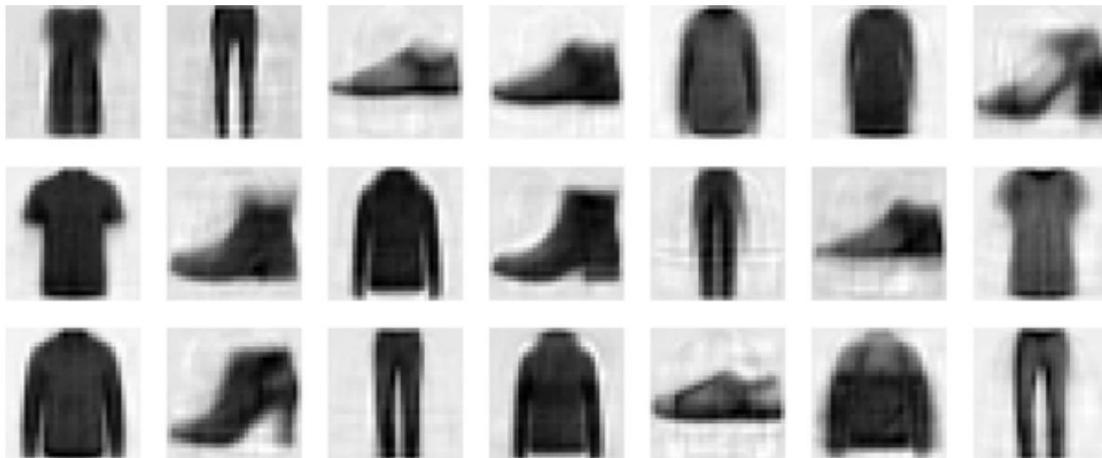


Figura 17-12. Imágenes MNIST de moda generadas por el codificador automático variacional

La mayoría de estas imágenes parecen bastante convincentes, aunque un poco confusas. El resto no es muy bueno, pero no seas demasiado duro con el codificador automático: ¡sólo tuvo unos minutos para aprender!

Los codificadores automáticos variacionales permiten realizar interpolación semántica: en lugar de interpolar entre dos imágenes a nivel de píxeles, lo que parecería como si las dos imágenes estuvieran simplemente superpuestas, podemos interpolar a nivel de codificaciones. Por ejemplo, tomemos algunas codificaciones a lo largo de una línea arbitraria en el espacio latente y decodifíquelas. Obtenemos una secuencia de imágenes que van gradualmente desde pantalones hasta suéteres (ver Figura 17-13):

```

codificaciones = np.zeros([7, codings_size])
codificaciones[:, 3] = np.linspace(-0.8, 0.8, 7) # el eje 3 se ve mejor en este caso
imágenes =
variacional_decoder(codings).numpy()

```



Figura 17-13. Interpolación semántica

Ahora dirijamos nuestra atención a las GAN: son más difíciles de entrenar, pero cuando logras hacerlas funcionar, producen imágenes bastante sorprendentes.

## Redes generativas de confrontación

Las redes generativas adversarias se propusieron en un [artículo de 2014](#). por Ian Goodfellow et al., y aunque la idea entusiasmó a los investigadores casi instantáneamente, tomó algunos años superar algunas de las dificultades del entrenamiento de GAN. Como muchas grandes ideas, parece simple en retrospectiva: hacer que las redes neuronales compitan entre sí con la esperanza de que esta competencia las impulse a sobresalir. Como se muestra en [la Figura 17-14](#), una GAN se compone de dos redes neuronales:

### Generador

Toma una distribución aleatoria como entrada (normalmente gaussiana) y genera algunos datos, normalmente una imagen. Puede pensar en las entradas aleatorias como representaciones latentes (es decir, codificaciones) de la imagen que se va a generar. Entonces, como puede ver, el generador ofrece la misma funcionalidad que un decodificador en un codificador automático variacional, y se puede usar de la misma manera para generar nuevas imágenes: simplemente alimente un poco de ruido gaussiano y generará una imagen completamente nueva. . Sin embargo, se entrena de manera muy diferente, como pronto verás.

### Discriminado

Toma como entrada una imagen falsa del generador o una imagen real del conjunto de entrenamiento y debe adivinar si la imagen de entrada es falsa o real.

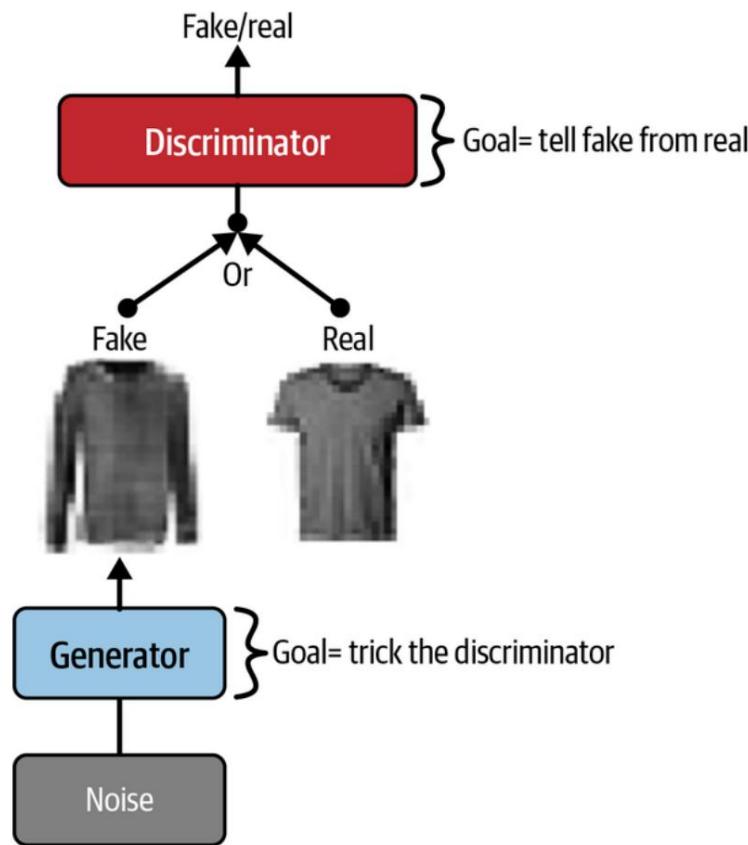


Figura 17-14. Una red generativa de confrontación

Durante el entrenamiento, el generador y el discriminador tienen objetivos opuestos: el discriminador intenta distinguir imágenes falsas de imágenes reales, mientras que el generador intenta producir imágenes que parezcan lo suficientemente reales como para engañar al discriminador. Debido a que la GAN se compone de dos redes con objetivos diferentes, no se puede entrenar como una red neuronal normal. Cada iteración de entrenamiento se divide en dos fases:

- En la primera fase, entrenamos al discriminador. Se toma una muestra de un lote de imágenes reales del conjunto de entrenamiento y se completa con una cantidad igual de imágenes falsas producidas por el generador. Las etiquetas se establecen en 0 para imágenes falsas y en 1 para imágenes reales, y el discriminador se entrena en este lote etiquetado durante un paso, utilizando la pérdida binaria de entropía cruzada. Es importante destacar que la retropropagación solo optimiza los pesos del discriminador durante esta fase.
- En la segunda fase, entrenamos el generador. Primero lo usamos para producir otro lote de imágenes falsas y, una vez más, se usa el discriminador para determinar si las imágenes son falsas o reales. Este

En ese momento no agregamos imágenes reales en el lote y todas las etiquetas se establecen en 1 (real): en otras palabras, queremos que el generador produzca imágenes que el discriminador creerá (erróneamente) que son reales. Fundamentalmente, los pesos del discriminador se congelan durante este paso, por lo que la retropropagación solo afecta los pesos del generador.

### NOTA

En realidad, el generador nunca ve imágenes reales, pero poco a poco aprende a producir imágenes falsas convincentes. Todo lo que obtiene son los gradientes que regresan a través del discriminador. Afortunadamente, cuanto mejor es el discriminador, más información sobre las imágenes reales contienen estos gradientes de segunda mano, por lo que el generador puede lograr avances significativos.

Sigamos adelante y construyamos una GAN simple para Fashion MNIST.

Primero, necesitamos construir el generador y el discriminador. El generador es similar al decodificador de un codificador automático y el discriminador es un clasificador binario normal: toma una imagen como entrada y termina con una capa densa que contiene una sola unidad y utiliza la función de activación sigmoidea. Para la segunda fase de cada iteración de entrenamiento, también necesitamos el modelo GAN completo que contiene el generador seguido del discriminador:

```
codificaciones_tamaño = 30

Denso = tf.keras.layers.Dense generador =
tf.keras.Sequential([ Denso(100, activación="relu",
kernel_initializer="él_normal"),
Denso(150, activación="relu",
kernel_initializer="he_normal"), Denso(28 * 28,
activación="sigmoide"), tf.keras.layers.Reshape([28, 28])

])

discriminador = tf.keras.Sequential([
tf.keras.layers.Flatten(), Denso(150,
activación="relu",
kernel_initializer="él_normal"),
Denso(100, activación="relu",
kernel_initializer="él_normal"),
```

```

        Denso(1, activación="sigmoide")
    ])
gan = tf.keras.Sequential([generador, discriminador])

```

A continuación, necesitamos compilar estos modelos. Como el discriminador es un clasificador binario, naturalmente podemos utilizar la pérdida de entropía cruzada binaria. El modelo gan también es un clasificador binario, por lo que también puede utilizar la pérdida binaria de entropía cruzada. Sin embargo, el generador solo se entrenará a través del modelo gan, por lo que no es necesario compilarlo en absoluto. Es importante destacar que el discriminador no debe entrenarse durante la segunda fase, por lo que lo hacemos no entrenable antes de compilar el modelo gan:

```

discriminator.compile(loss="binary_crossentropy", optimizador="rmsprop")
discriminator.trainable = False
gan.compile(loss="binary_crossentropy",
            optimizador="rmsprop")

```

### NOTA

Keras solo tiene en cuenta el atributo entrenable al compilar un modelo, por lo que después de ejecutar este código, el discriminador se puede entrenar si llamamos a su método fit() o a su método train\_on\_batch() (que usaremos), mientras que no se puede entrenar cuando llamamos a estos métodos en el modelo gan.

Dado que el ciclo de entrenamiento es inusual, no podemos utilizar el método fit() normal. En su lugar, escribiremos un ciclo de entrenamiento personalizado. Para esto, primero necesitamos crear un conjunto de datos para recorrer las imágenes:

```

tamaño_por_lotes = 32
conjunto de
datos = tf.data.Dataset.from_tensor_slices(X_train).shuffle(tamaño_búfer =1000) conjunto de
datos =
conjunto de datos.batch(tamaño_por_lotes,
drop_remainder=True).prefetch(1)

```

Ahora estamos listos para escribir el ciclo de entrenamiento. Vamos a envolverlo en una función train\_gan():

```

def train_gan(gan, conjunto de datos, tamaño de lote, tamaño de codificación,
n_epochs):
    generador, discriminador = gan.layers para época en
    rango (n_epochs): para X_batch en conjunto
        de datos: # fase 1 - entrenamiento
            del ruido discriminador = tf.random.normal(forma
                =[tamaño_lote,
                codificaciones_tamaño])
                imágenes_generadas = generador(ruido)
                X_fake_and_real = tf.concat([generated_images, X_batch], eje=0) y1
                = tf.constant([[0.]]) *
                    tamaño_lote + [[1.]] *
                    tamaño del lote)
                discriminator.train_on_batch(X_fake_and_real, y1) # fase 2 - entrenamiento
                del ruido del generador =
                    tf.random.normal(shape=[batch_size,
                    codificaciones_tamaño])
                    y2 = tf.constant([[1.]]) * tamaño_lote)
                    gan.train_on_batch(ruido, y2)

train_gan(gan, conjunto de datos, tamaño_lote, tamaño_codificaciones, n_epochs=50)

```

Como se mencionó anteriormente, puede ver las dos fases en cada iteración:

- En la fase uno, alimentamos ruido gaussiano al generador para producir imágenes falsas y completamos este lote concatenando un número igual de imágenes reales. Los objetivos y1 se establecen en 0 para imágenes falsas y en 1 para imágenes reales. Luego entrenamos al discriminador en este lote. Recuerde que el discriminador es entrenable en esta fase, pero no estamos tocando el generador.
- En la fase dos, alimentamos a la GAN con algo de ruido gaussiano. Su generador comenzará produciendo imágenes falsas, luego el discriminador intentará adivinar si estas imágenes son falsas o reales. En esta fase, estamos intentando mejorar el generador, lo que significa que queremos que el discriminador falle: es por eso que los objetivos y2 están todos configurados en 1, aunque las imágenes son falsas. En esta fase, el discriminador no se puede entrenar, por lo que la única parte del modelo gan que mejorará es el generador.

¡Eso es todo! Después del entrenamiento, puede muestrear aleatoriamente algunas codificaciones de una distribución gaussiana y alimentarlas al generador para producir nuevas

imágenes:

```
codificaciones = tf.random.normal(forma=[tamaño_lote, tamaño_codificaciones])
imágenes_generadas = generador.predict(codificaciones)
```

Si muestra las imágenes generadas (ver [Figura 17-15](#)), verá que al final de la primera época, ya comienzan a verse (muy ruidosas) imágenes MNIST de moda.



Figura 17-15. Imágenes generadas por GAN después de una época de entrenamiento.

Desafortunadamente, las imágenes nunca son mucho mejores que eso, e incluso puedes encontrar épocas en las que la GAN parece estar olvidando lo que aprendió. ¿Por qué es eso? Bueno, resulta que entrenar una GAN puede ser un desafío. Veamos por qué.

## Las dificultades de entrenar GAN

Durante el entrenamiento, el generador y el discriminador intentan constantemente ser más astutos que el otro, en un juego de suma cero. A medida que avanza el entrenamiento, el juego puede terminar en un estado que los teóricos de juegos llaman equilibrio de Nash, llamado así en honor del matemático John Nash: aquí es cuando ningún jugador estaría mejor si cambiara su propia estrategia, suponiendo que los otros jugadores no cambien la suya. Por ejemplo, se alcanza un equilibrio de Nash cuando todo el mundo conduce por el lado izquierdo de la carretera: ningún conductor estaría mejor si fuera el único en cambiar de lado. Por supuesto, hay una segunda posible

Equilibrio de Nash: cuando todos conducen por el lado derecho de la carretera. Diferentes estados iniciales y dinámicas pueden conducir a un equilibrio u otro. En este ejemplo, hay una única estrategia óptima una vez que se alcanza un equilibrio (es decir, conducir por el mismo lado que todos los demás), pero un equilibrio de Nash puede implicar múltiples estrategias competitivas (por ejemplo, un depredador persigue a su presa, la presa intenta escapar, y ninguno de los dos estaría mejor si cambiara su estrategia).

Entonces, ¿cómo se aplica esto a las GAN? Bueno, los autores del artículo sobre GAN demostraron que una GAN solo puede alcanzar un único equilibrio de Nash: ahí es cuando el generador produce imágenes perfectamente realistas y el discriminador se ve obligado a adivinar (50% real, 50% falso). Este hecho es muy alentador: parecería que basta con entrenar la GAN durante el tiempo suficiente y eventualmente alcanzará este equilibrio, proporcionándose un generador perfecto. Desgraciadamente, no es tan sencillo: nada garantiza que algún día se alcance el equilibrio.

La mayor dificultad se llama colapso modal: esto ocurre cuando las salidas del generador se vuelven gradualmente menos diversas. ¿Cómo puede suceder esto? Supongamos que el generador produce mejores zapatos que cualquier otra clase. Engañará un poco más al discriminador con los zapatos, y esto le animará a producir aún más imágenes de zapatos. Poco a poco se olvidará de cómo producir cualquier otra cosa. Mientras tanto, las únicas imágenes falsas que verá el discriminador serán los zapatos, por lo que también olvidará cómo discriminar imágenes falsas de otras clases. Finalmente, cuando el discriminador logra discriminar los zapatos falsos de los reales, el generador se verá obligado a pasar a otra clase. Entonces puede volverse bueno con las camisas, olvidándose de los zapatos, y el discriminador lo seguirá.

La GAN puede recorrer gradualmente algunas clases y nunca llegar a ser muy buena en ninguna de ellas.

Además, debido a que el generador y el discriminador se empujan constantemente entre sí, sus parámetros pueden terminar oscilando y volviéndose inestables. El entrenamiento puede comenzar correctamente y luego de repente divergir sin razón aparente, debido a estas inestabilidades. Y dado que muchos factores afectan esta dinámica compleja, las GAN son muy sensibles a los hiperparámetros: es posible que tengas que esforzarte mucho para ajustarlos.

De hecho, es por eso que usé RMSProp en lugar de Nadam al compilar los modelos: cuando usé Nadam, me encontré con un colapso de modo severo.

Estos problemas han mantenido a los investigadores muy ocupados desde 2014: se han publicado muchos artículos sobre este tema, algunos de los cuales proponen <sup>10</sup> nuevas funciones de costos (aunque un artículo de 2018 por investigadores de Google cuestiona su eficiencia) o técnicas para estabilizar el entrenamiento o evitar el problema del colapso del modo. Por ejemplo, una técnica popular llamada repetición de experiencia consiste en almacenar las imágenes producidas por el generador en cada iteración en un búfer de reproducción (eliminando gradualmente las imágenes generadas más antiguas) y entrenar al discriminador utilizando imágenes reales más imágenes falsas extraídas de este búfer (en lugar de simplemente imágenes falsas producidas por el generador actual). Esto reduce las posibilidades de que el discriminador se sobre adapte a las salidas del último generador. Otra técnica común se llama discriminación de mini lotes: mide qué tan similares son las imágenes en el lote y proporciona esta estadística al discriminador, para que pueda rechazar fácilmente un lote completo de imágenes falsas que carecen de diversidad. Esto anima al generador a producir una mayor variedad de imágenes, reduciendo la posibilidad de colapso del modo. Otros artículos simplemente proponen arquitecturas específicas que funcionan bien.

En resumen, este es todavía un campo de investigación muy activo y la dinámica de las GAN aún no se comprende perfectamente. Pero la buena noticia es que se han logrado grandes avances y algunos de los resultados son realmente sorprendentes. Entonces, veamos algunas de las arquitecturas más exitosas, comenzando con las GAN convolucionales profundas, que eran lo último en tecnología hace apenas unos años. Luego veremos dos arquitecturas más recientes (y más complejas).

## GAN convolucionales profundas Los

autores del artículo original sobre GAN experimentaron con capas convolucionales, pero solo intentaron generar imágenes pequeñas. Poco después, muchos investigadores intentaron construir GAN basadas en redes convolucionales más profundas para imágenes más grandes. Esto resultó ser complicado, ya que el entrenamiento era muy inestable, pero Alec Radford et al. Finalmente lo logró a finales de 2015, después de experimentar con muchas arquitecturas e hiperparámetros diferentes. Llama

arquitectura **GAN convolucional profunda** (DCGAN). Estas son las <sup>12</sup> principales pautas que propusieron para construir GAN convolucionales estables:

- Reemplace las capas de agrupación con convoluciones escalonadas (en el discriminador) y convoluciones transpuestas (en el generador).
- Utilice la normalización por lotes tanto en el generador como en el discriminador, excepto en la capa de salida del generador y en la capa de entrada del discriminador.
- Elimine las capas ocultas completamente conectadas para obtener arquitecturas más profundas.
- Utilice la activación ReLU en el generador para todas las capas excepto la capa de salida, que debería utilizar tanh.
- Utilice activación ReLU con fugas en el discriminador para todas las capas.

Estas pautas funcionarán en muchos casos, pero no siempre, por lo que es posible que aún tengas que experimentar con diferentes hiperparámetros. De hecho, a veces basta con cambiar la semilla aleatoria y entrenar exactamente el mismo modelo nuevamente. Aquí hay un pequeño DCGAN que funciona razonablemente bien con Fashion MNIST:

```
codificaciones_tamaño = 100

generador = tf.keras.Sequential([
    tf.keras.layers.Dense(7 * 7 * 128),
    tf.keras.layers.Reshape([7, 7, 128]),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2DTranspose(64, tamaño_núcleo=5, zancadas=2,
                                  relleno="igual",
                                  activación="relu"),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2DTranspose(1, kernel_size=5, strides=2,
                                  relleno="igual",
                                  activación="tanh"), ])

discriminador = tf.keras.Sequential([
    tf.keras.layers.Conv2D(64, kernel_size=5, strides=2, padding="igual",
```

```

activación=tf.keras.layers.LeakyReLU(0.2)),
tf.keras.layers.Dropout(0.4),
tf.keras.layers.Conv2D(128, kernel_size=5, strides=2, padding="mismo",

activación=tf.keras.layers.LeakyReLU(0.2)),
tf.keras.layers.Dropout(0.4),
tf.keras.layers.Flatten(),
tf.keras.layers.Dense(1, activación="sigmoide" )
])
gan = tf.keras.Sequential([generador, discriminador])

```

El generador toma codificaciones de tamaño 100, las proyecta a 6272 dimensiones ( $7 \times 7 \times 128$ ) y reforma el resultado para obtener un tensor de  $7 \times 7 \times 128$ . Este tensor se normaliza por lotes y se envía a una capa convolucional transpuesta con un paso de 2, lo que lo aumenta de  $7 \times 7$  a  $14 \times 14$  y reduce su profundidad de 128 a 64. El resultado se normaliza por lotes nuevamente y se envía a otro convolucional transpuesto. capa con una zancada de 2, que la aumenta de  $14 \times 14$  a  $28 \times 28$  y reduce la profundidad de 64 a 1. Esta capa utiliza la función de activación tanh, por lo que las salidas oscilarán entre -1 y 1. Por esta razón, Antes de entrenar la GAN, debemos reescalar el conjunto de entrenamiento al mismo rango. También necesitamos remodelarlo para agregar la dimensión del canal:

```
X_train_dcgan = X_train.reshape(-1, 28, 28, 1) * 2. - 1. # remodelar y reescalar
```

El discriminador se parece mucho a una CNN normal para clasificación binaria, excepto que en lugar de usar capas de agrupación máxima para reducir la resolución de la imagen, usamos convoluciones con zancadas (zancadas = 2). Tenga en cuenta que utilizamos la función de activación ReLU con fugas. En general, respetamos las pautas de DCGAN, excepto que reemplazamos las capas BatchNormalization en el discriminador con capas Dropout; de lo contrario, en este caso el entrenamiento fue inestable. Siéntase libre de modificar esta arquitectura: verá cuán sensible es a los hiperparámetros, especialmente las tasas de aprendizaje relativas de las dos redes.

Por último, para construir el conjunto de datos y luego compilar y entrenar este modelo, podemos usar el mismo código que antes. Después de 50 épocas de entrenamiento, el generador

produce imágenes como las que se muestran en [la Figura 17-16](#). Todavía no es perfecto, pero muchas de estas imágenes son bastante convincentes.



Figura 17-16. Imágenes generadas por el DCGAN después de 50 épocas de entrenamiento

Si amplía esta arquitectura y la entrena en un gran conjunto de datos de rostros, puede obtener imágenes bastante realistas. De hecho, las DCGAN pueden aprender representaciones latentes bastante significativas, como se puede ver en [la Figura 17-17](#): se generaron muchas imágenes y nueve de ellas se seleccionaron manualmente (arriba a la izquierda), incluidas tres que representan a hombres con gafas, tres hombres sin gafas, y tres mujeres sin gafas. Para cada una de estas categorías, se promediaron las codificaciones que se utilizaron para generar las imágenes y se generó una imagen basada en las codificaciones medias resultantes (abajo a la izquierda). En resumen, cada una de las tres imágenes de la parte inferior izquierda representa la media de las tres imágenes ubicadas encima. Pero esta no es una media simple calculada a nivel de píxel (esto daría como resultado tres caras superpuestas), es una media calculada en el espacio latente, por lo que las imágenes aún parecen caras normales. Sorprendentemente, si calculas a los hombres con gafas, menos los hombres sin gafas, más las mujeres sin gafas (donde cada término corresponde a una de las codificaciones medias) y generas la imagen que corresponde a esta codificación, obtienes la imagen en el centro de la codificación. Cuadrícula 3×3 de rostros a la derecha: ¡una mujer con gafas! Las otras ocho imágenes a su alrededor se generaron en base al mismo vector más un poco de ruido, para ilustrar las capacidades de interpolación semántica de las DCGAN. ¡Poder hacer aritmética con caras parece ciencia ficción!

Sin embargo, las DCGAN no son perfectas. Por ejemplo, cuando intentas generar imágenes muy grandes usando DCGAN, a menudo terminas con características localmente convincentes pero inconsistencias generales, como camisas con una manga mucho más larga que la otra, aretes diferentes u ojos que miran en direcciones opuestas. ¿Cómo puedes arreglar esto?

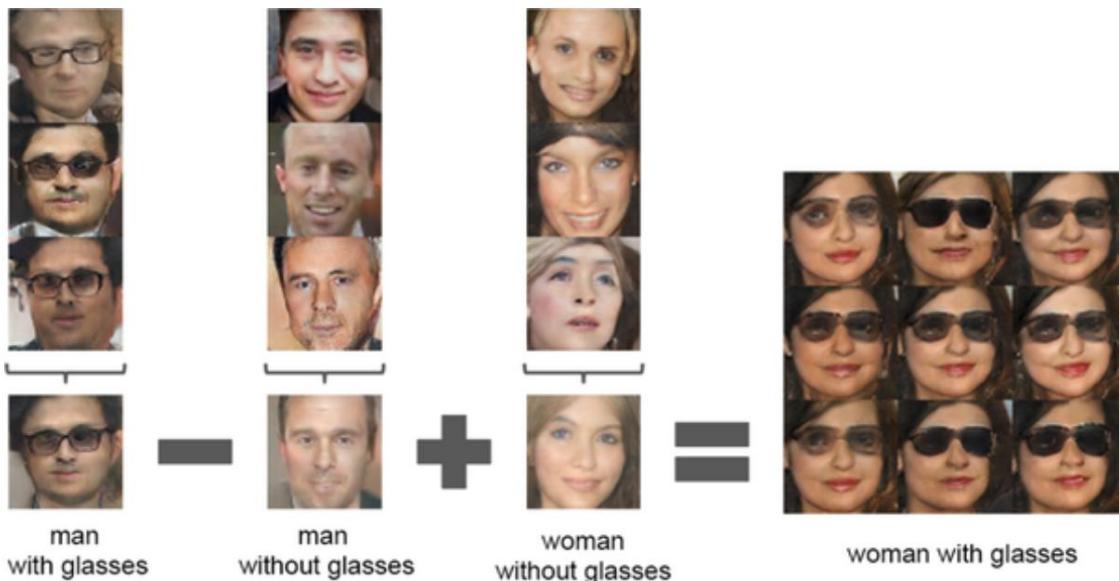


Figura 17-17. Aritmética vectorial para conceptos visuales (parte de la figura 7 del artículo de DCGAN)<sup>13</sup>

#### CONSEJO

Si agrega la clase de cada imagen como entrada adicional tanto al generador como al discriminador, ambos aprenderán cómo se ve cada clase y, por lo tanto, podrá controlar la clase de cada imagen producida por el generador. Esto se llama **condicional GAN(CGAN)**.<sup>14</sup>

## Crecimiento progresivo de las GAN

En un [artículo de 2018](#), Los investigadores de Nvidia, Tero Keras et al. propusieron una técnica importante: sugirieron generar imágenes pequeñas al comienzo del entrenamiento y luego agregar gradualmente capas convolucionales tanto al generador como al discriminador para producir imágenes cada vez más grandes ( $4 \times 4$ ,  $8 \times 8$ ,  $16 \times 16$ , ...,  $512 \times 512$ ,  $1.024 \times 1.024$ ). Este enfoque se asemeja al entrenamiento codicioso por capas de codificadores automáticos apilados. El extra

las capas se agregan al final del generador y al comienzo del discriminador, y las capas previamente entrenadas siguen siendo entrenables.

Por ejemplo, al aumentar las salidas del generador de  $4 \times 4$  a  $8 \times 8$  (consulte la Figura 17-18), se agrega una capa de muestreo superior (usando el filtrado del vecino más cercano) a la capa convolucional existente ("Conv 1") para producir  $8 \times 8$  mapas de características. Estos se envían a la nueva capa convolucional ("Conv 2"), que a su vez se alimenta a una nueva capa convolucional de salida. Para evitar romper los pesos entrenados de Conv 1, gradualmente desvanecemos las dos nuevas capas convolucionales (representadas con líneas discontinuas en la Figura 17-18) y desvanecemos la capa de salida original. Las salidas finales son una suma ponderada de las nuevas salidas (con peso  $\alpha$ ) y las salidas originales (con peso  $1 - \alpha$ ), aumentando lentamente  $\alpha$  de 0 a 1. Se utiliza una técnica de aparición/desaparición similar cuando se agrega una nueva capa convolucional al discriminador (seguida de una capa de agrupación promedio para la reducción de resolución). Tenga en cuenta que todas las capas convolucionales utilizan el "mismo" relleno y pasos de 1, por lo que conservan la altura y el ancho de sus entradas. Esto incluye la capa convolucional original, por lo que ahora produce salidas de  $8 \times 8$  (ya que sus entradas ahora son  $8 \times 8$ ). Por último, las capas de salida utilizan el tamaño de kernel 1. Simplemente proyectan sus entradas hasta el número deseado de canales de color.

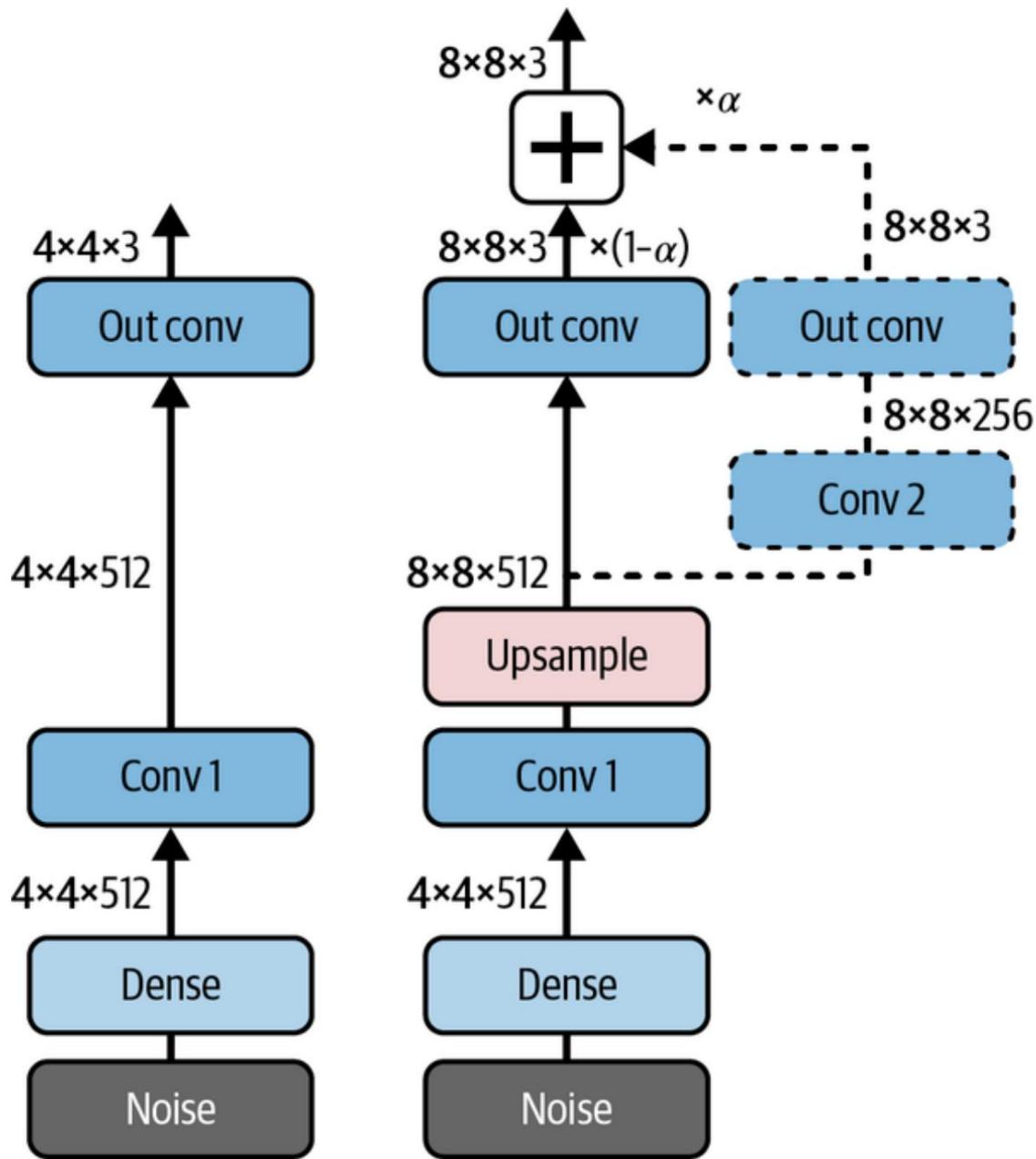


Figura 17-18. Una GAN en crecimiento progresivo: un generador de GAN genera imágenes en color de  $4 \times 4$  (izquierdo); lo ampliamos para generar imágenes de  $8 \times 8$  (derecha)

El artículo también introdujo varias otras técnicas destinadas a aumentar la diversidad de los resultados (para evitar el colapso del modo) y hacer que el entrenamiento sea más estable:

Capa de desviación estándar de mini lotes

Agregado cerca del final del discriminador. Para cada posición en las entradas, calcula la desviación estándar en todos los canales y en todos los

instancias en el lote ( $S = \text{tf.math.reduce\_std(inputs, axis=[0, -1])}$ ). Luego, estas desviaciones estándar se promedian en todos los puntos para obtener un valor único ( $v = \text{tf.reduce_mean}(S)$ ).

Finalmente, se agrega un mapa de características adicional a cada instancia del lote y se completa con el valor calculado ( $\text{tf.concat}([\text{inputs}, \text{tf.fill}([\text{batch\_size}, \text{height}, \text{width}, 1], v)], \text{axis}=-1)$ ). ¿Cómo ayuda esto? Bueno, si el generador produce imágenes con poca variedad, habrá una pequeña desviación estándar entre los mapas de características en el discriminador. Gracias a esta capa, el discriminador tendrá fácil acceso a esta estadística, por lo que será menos probable que se deje engañar por un generador que produce muy poca diversidad. Esto alentará al generador a producir productos más diversos, reduciendo el riesgo de colapso modal.

### Tasa de aprendizaje igualada

Inicializa todos los pesos utilizando una distribución gaussiana con media 0 y desviación estándar 1 en lugar de utilizar la inicialización He. Sin embargo, los pesos se reducen en tiempo de ejecución (es decir, cada vez que se ejecuta la capa) por el mismo factor que en la inicialización de He: se dividen por  $\sqrt{2/n}$  entradas, donde

n artículos demostraron es el número de entradas a la capa. las entradas que esta técnica mejoró significativamente el rendimiento de la GAN cuando utilizando RMSProp, Adam u otros optimizadores de gradiente adaptativos. De hecho, estos optimizadores normalizan las actualizaciones de gradiente según su desviación estándar estimada (consulte [el Capítulo 11](#)), por lo que los parámetros que tienen un rango dinámico mayor tardarán más en entrenarse, mientras que los parámetros con un rango dinámico pequeño pueden actualizarse demasiado rápido, lo que genera inestabilidades. Al reescalar los pesos como parte del modelo en sí, en lugar de simplemente reescalarlos durante la inicialización, este enfoque garantiza que el rango dinámico sea el mismo para todos los parámetros durante el entrenamiento, de modo que todos aprendan a la misma velocidad.

Esto acelera y estabiliza el entrenamiento.

### Capa de normalización por píxeles

Agregado después de cada capa convolucional en el generador. Normaliza cada activación en función de todas las activaciones en la misma imagen y en la misma ubicación, pero en todos los canales (dividiendo por la raíz cuadrada

de la activación cuadrática media). En el código de TensorFlow, esto es `inputs / tf.sqrt(tf.reduce_mean(tf.square(X), axis=-1, keepdims=True) + 1e-8)` (el término de suavizado `1e-8` es necesario para evitar la división por cero). Esta técnica evita explosiones en las activaciones por excesiva competencia entre el generador y el discriminador.

La combinación de todas estas técnicas permitió a los autores generar **imágenes de rostros en alta definición extremadamente convincentes**. Pero, ¿a qué llamamos exactamente “convinciente”? La evaluación es uno de los grandes desafíos cuando se trabaja con GAN: aunque es posible evaluar automáticamente la diversidad de las imágenes generadas, juzgar su calidad es una tarea mucho más complicada y subjetiva. Una técnica consiste en utilizar evaluadores humanos, pero esto es costoso y requiere mucho tiempo. Entonces, los autores propusieron medir la similitud entre la estructura de imagen local de las imágenes generadas y las imágenes de entrenamiento, considerando todas las escalas. Esta idea los llevó a otra innovación revolucionaria: StyleGAN.

## EstiloGAN

El estado del arte en generación de imágenes de alta resolución fue avanzado una vez más por el mismo equipo de Nvidia en un [artículo de 2018](#).<sup>17</sup> que introdujo la popular arquitectura StyleGAN. Los autores utilizaron técnicas de transferencia de estilo en el generador para garantizar que las imágenes generadas tengan la misma estructura local que las imágenes de entrenamiento, en todas las escalas, mejorando enormemente la calidad de las imágenes generadas. El discriminador y la función de pérdida no fueron modificados, sólo el generador. Un generador StyleGAN se compone de dos redes (consulte [la Figura 17-19](#)):

### Red de mapeo

Un MLP de ocho capas que asigna las representaciones latentes  $z$  (es decir, las codificaciones) a un vector  $w$ . Luego, este vector se envía a través de múltiples transformaciones afines (es decir, capas densas sin funciones de activación, representadas por los cuadros "A" en la [Figura 17-19](#)), lo que produce múltiples vectores. Estos vectores controlan el estilo de la imagen generada en diferentes niveles, desde textura de grano fino (por ejemplo, color de cabello) hasta

características de alto nivel (por ejemplo, adulto o niño). En resumen, la red de mapeo asigna las codificaciones a múltiples vectores de estilo.

### Red de síntesis

Responsable de generar las imágenes. Tiene una entrada aprendida constante (para ser claros, esta entrada será constante después del entrenamiento, pero durante el entrenamiento sigue siendo modificada por la propagación hacia atrás). Procesa esta entrada a través de múltiples capas convolucionales y de muestreo ascendente, como antes, pero hay dos giros. Primero, se agrega algo de ruido a la entrada y a todas las salidas de las capas convolucionales (antes de la función de activación).

En segundo lugar, a cada capa de ruido le sigue una capa de normalización de instancia adaptativa (AdaIN): estandariza cada mapa de características de forma independiente (restando la media del mapa de características y dividiéndola por su desviación estándar), luego usa el vector de estilo para determinar la escala y el desplazamiento. de cada mapa de características (el vector de estilo contiene una escala y un término de sesgo para cada mapa de características).

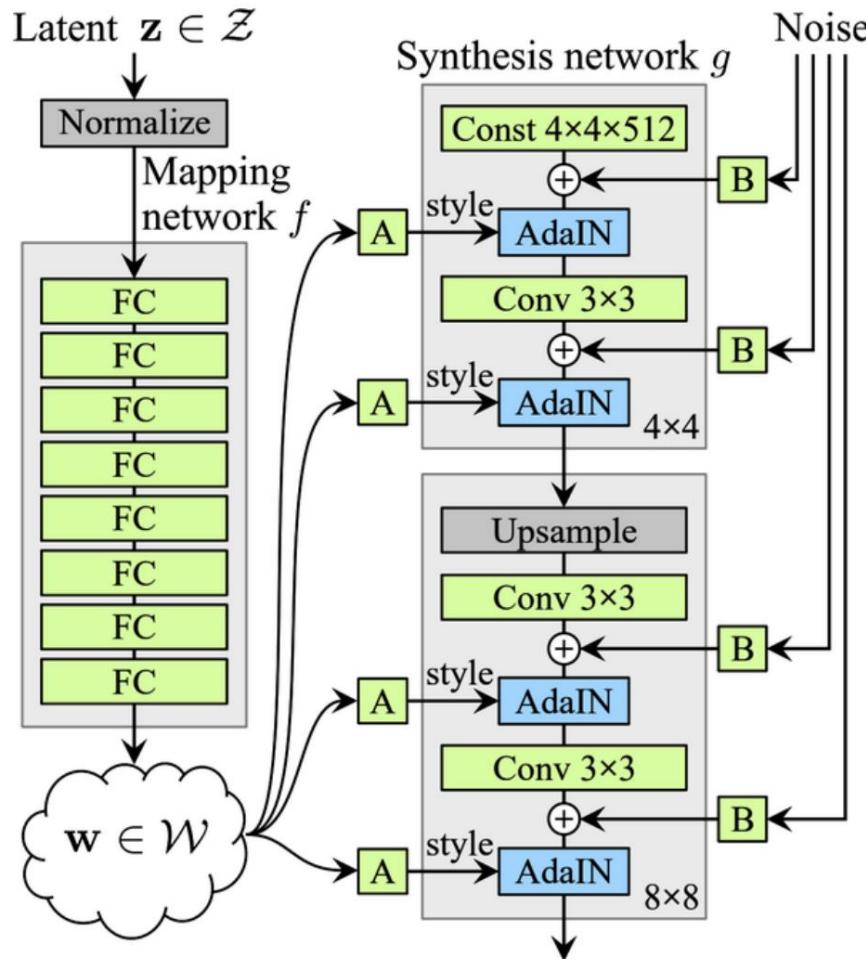


Figura 17-19. Arquitectura del generador de StyleGAN (parte de la Figura 1 del artículo de StyleGAN<sup>18</sup>)

La idea de añadir ruido independientemente de las codificaciones es muy importante. Algunas partes de una imagen son bastante aleatorias, como la posición exacta de cada peca o cabello. En las GAN anteriores, esta aleatoriedad tenía que provenir de las codificaciones o ser algún ruido pseudoaleatorio producido por el propio generador. Si procedía de las codificaciones, significaba que el generador tenía que dedicar una parte significativa del poder de representación de las codificaciones a almacenar ruido, lo cual es un desperdicio considerable. Además, el ruido tenía que poder fluir a través de la red y llegar a las capas finales del generador: esto parece una limitación innecesaria que probablemente ralentizó el entrenamiento. Y finalmente, pueden aparecer algunos artefactos visuales porque se utilizó el mismo ruido en diferentes niveles. Si, en cambio, el generador intentara producir su propio ruido pseudoaleatorio, este ruido podría no parecer muy convincente, lo que daría lugar a más artefactos visuales. Además, parte del peso del generador se dedicaría a

generando ruido pseudoaleatorio, lo que nuevamente parece un desperdicio. Al agregar entradas de ruido adicionales, se evitan todos estos problemas; la GAN puede utilizar el ruido proporcionado para agregar la cantidad correcta de estocasticidad a cada parte de la imagen.

El ruido añadido es diferente para cada nivel. Cada entrada de ruido consta de un único mapa de características lleno de ruido gaussiano, que se transmite a todos los mapas de características (del nivel dado) y se escala usando factores de escala aprendidos por característica (esto está representado por los cuadros "B" en la Figura 17-19) antes de agregarlo.

Finalmente, StyleGAN utiliza una técnica llamada regularización de mezcla (o mezcla de estilos), donde un porcentaje de las imágenes generadas se produce utilizando dos codificaciones diferentes. Específicamente, las codificaciones  $c$  y  $c_{\perp}$  se envían a través de la red de mapeo, dando dos vectores de estilo  $w$  y  $w_{\perp}$ . Luego, la red de síntesis genera una imagen basada en los estilos  $w$  para los primeros niveles y los estilos  $w_{\perp}$  para los niveles restantes. El nivel de corte se elige al azar. Esto evita que la red suponga que los estilos en niveles adyacentes están correlacionados, lo que a su vez fomenta la localidad en la GAN, lo que significa que cada vector de estilo solo afecta a un número limitado de rasgos en la imagen generada.

Existe una variedad tan amplia de GAN que se necesitaría un libro completo para cubrirlas todas. Esperamos que esta introducción le haya dado las ideas principales y, lo más importante, el deseo de aprender más. Anímate a implementar tu propia GAN, y no te desanimes si al principio tiene problemas para aprender: lamentablemente, esto es normal y requerirá bastante paciencia para que funcione, pero el resultado vale la pena. Si tiene dificultades con un detalle de implementación, hay muchas implementaciones de Keras o TensorFlow que puede consultar. De hecho, si todo lo que desea es obtener resultados sorprendentes rápidamente, puede utilizar un modelo previamente entrenado (por ejemplo, hay modelos StyleGAN previamente entrenados disponibles para Keras).

Ahora que hemos examinado los codificadores automáticos y las GAN, veamos un último tipo de arquitectura: los modelos de difusión.

## Modelos de difusión

Las ideas detrás de los modelos de difusión existen desde hace muchos años, pero se formalizaron por primera vez en su forma moderna en un [artículo de 2015](#), por<sup>19</sup> Jascha Sohl-Dickstein et al. de la Universidad de Stanford y UC Berkeley. Los autores aplicaron herramientas de la termodinámica para modelar un proceso de difusión, similar a una gota de leche que se difunde en una taza de té. La idea central es entrenar un modelo para que aprenda el proceso inverso: comenzar desde el estado completamente mezclado y gradualmente "desmezclar" la leche del té. Utilizando esta idea, obtuvieron resultados prometedores en la generación de imágenes, pero como las GAN producían imágenes más convincentes en aquel entonces, los modelos de difusión no recibieron tanta atención.

Luego, en 2020, [Jonathan Ho et al.](#), también de la UC Berkeley, lograron construir un modelo de difusión capaz de generar imágenes muy realistas, al que llamaron modelo probabilístico de difusión de eliminación de ruido (DDPM). Unos meses después, un [artículo de 2021](#). Los investigadores<sup>21</sup> de OpenAI, Alex Nichol y Prafulla Dhariwal, analizaron la arquitectura DDPM y propusieron varias mejoras que permitieron a los DDPM vencer finalmente a las GAN: no solo son mucho más fáciles de entrenar que las GAN, sino que las imágenes generadas son más diversas y de calidad aún mayor. La principal desventaja de los DDPM, como verá, es que tardan mucho en generar imágenes, en comparación con las GAN o VAE.

Entonces, ¿cómo funciona exactamente un DDPM? Bueno, supongamos que comienzas con una imagen de un gato (como la que verás en [la Figura 17-20](#)), anotada  $x$  y en cada paso  $t$  agregas un poco de ruido gaussiano a la imagen, con media 0 y varianza  $\beta$ . Este ruido es independiente para cada píxel: lo llamamos isotrópico. Primero se obtiene la imagen  $x$  y así sucesivamente, hasta que el gato queda completamente oculto por el ruido, imposible de ver. El último paso de tiempo se anota como  $T$ . En el artículo original de DDPM, los autores usaron  $T = 1000$  y programaron la varianza  $\beta$  de tal manera que la señal del gato se desvanece linealmente entre los pasos de tiempo 0 y  $T$ . En el artículo de DDPM mejorado,  $T$  aumentó a 4000 y el programa de variación se modificó para que cambiara más lentamente al principio y al final. En resumen, poco a poco vamos ahogando al gato en ruido: esto se llama proceso de avance.

A medida que agregamos más y más ruido gaussiano en el proceso de avance, la distribución de los valores de píxeles se vuelve cada vez más gaussiana. Un detalle importante que omití es que los valores de píxeles se reescalán ligeramente en

cada paso, por un factor de  $\sqrt{1 - \beta t}$ . Esto garantiza que la media de los valores de píxeles se acerque gradualmente a 0, ya que el factor de escala es un poco menor que 1 (imagínese multiplicar repetidamente un número por 0,99). También asegura que la varianza convergerá gradualmente a 1. Esto se debe a que

la desviación estándar de los valores de los píxeles también se escala en  $\sqrt{1 - \beta t}$ , por lo que la varianza se escala en  $1 - \beta$  (es decir, el cuadrado del factor de escala). Pero la varianza no puede reducirse a 0 ya que estamos agregando ruido gaussiano con varianza  $\beta$  en cada paso. Y dado que las varianzas se suman cuando se suman las distribuciones gaussianas, se puede ver que la varianza solo puede converger a  $1 - \beta + \beta = 1$ .

$t$        $t$

El proceso de difusión directa se resume en [la ecuación 17-5](#). Esta ecuación no le enseñará nada nuevo sobre el proceso de avance, pero es útil para comprender este tipo de notación matemática, como se usa a menudo en artículos de aprendizaje automático. Esta ecuación define la distribución de probabilidad  $q$  de  $x$  dado  $x$  como una distribución gaussiana con media  $x$  multiplicada por el factor de escala  $y$  con una matriz de covarianza igual a  $\beta I$ . Esta es la matriz identidad  $I$  multiplicada por  $\beta$ , lo que significa que el ruido es isotrópico con varianza  $\beta$ .

$t$

Ecuación 17-5. Distribución de probabilidad  $q$  del proceso de difusión directa

$$q(x_t | x_{t-1}) = N(\sqrt{1 - \beta} x_{t-1}, \beta I)$$

Curiosamente, hay un atajo para el proceso de avance: es posible muestrear una imagen  $x$  dado  $x$  sin tener que calcular primero  $x_1, x_2, \dots, x_{t-1}$

1. De hecho, dado que la suma de múltiples distribuciones gaussianas también es una Distribución gaussiana, todo el ruido se puede agregar de una sola vez usando [la ecuación 17-6](#). Esta es la ecuación que usaremos, ya que es mucho más rápida.

Ecuación 17-6. Atajo para el proceso de difusión directa

$$q(x_t | x_0) = N(\sqrt{\alpha} x_0, (1 - \alpha)t I)$$

Nuestro objetivo, por supuesto, no es ahogar a los gatos en el ruido. ¡Al contrario, queremos crear muchos gatos nuevos! Podemos hacerlo entrenando un modelo que pueda realizar el proceso inverso: pasar de  $x$  a  $x_0$ . Luego podemos usarlo para

elimine un poquito de ruido de una imagen y repita la operación muchas veces hasta que desaparezca todo el ruido. Si entrenamos el modelo en un conjunto de datos que contiene muchas imágenes de gatos, entonces podemos darle una imagen completamente llena de ruido gaussiano y el modelo hará aparecer gradualmente un gato nuevo (consulte la Figura 17-20) .

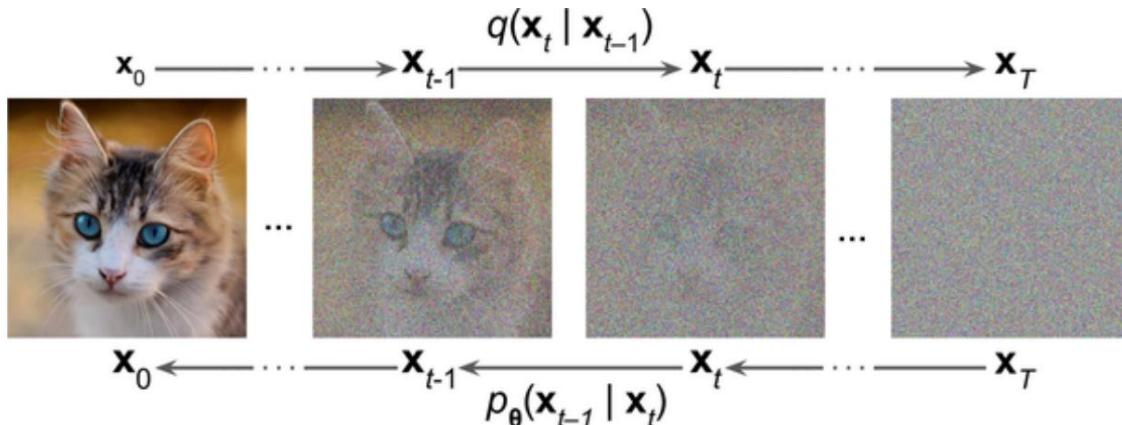


Figura 17-20. El proceso directo q y el proceso inverso p

Bien, ¡comencemos a codificar! Lo primero que debemos hacer es codificar el proceso de avance. Para ello, primero necesitaremos implementar el programa de variación. ¿Cómo podemos controlar la rapidez con la que desaparece el gato? Inicialmente, el 100% de la variación proviene de la imagen del gato original. Luego, en cada paso de tiempo t, la varianza se multiplica por  $1 - \beta$ , como se explicó anteriormente, y se agrega ruido. Entonces, la parte de la varianza que proviene de la distribución inicial se reduce en un factor de  $1 - \beta$  en cada paso. Si definimos  $\alpha = 1 - \beta$ , entonces después de t pasos de tiempo, la señal cat se habrá multiplicado por un factor de  $\alpha^t = \alpha \times \dots \times \alpha = \prod_{i=1}^t \alpha$ . Es este factor de "señal de gato" α que queremos programar para que se reduzca gradualmente de 1 a 0 entre los pasos de tiempo 0 y T. En el artículo mejorado de DDPM, los autores programan α según la [Ecuación 17-7](#). Este cronograma se representa en la [Figura 17-21](#).

Ecuación 17-7. Ecuaciones del programa de varianza para el proceso de difusión directa.

$$\beta_t = 1 - \frac{\alpha_t}{\alpha_{t-1}}, \text{ con } \alpha_t = \frac{f(t)}{f(0)} \text{ y } f(t) = \cos\left(\frac{t\pi}{T} + s\right)^{\frac{2}{1+s}}$$

En estas ecuaciones:

- $s$  es un valor pequeño que evita que  $\beta$  sea demasiado pequeño cerca de  $t = 0$ . En el artículo, los autores utilizaron  $s = 0,008$ .
- $\beta$  se recorta para que no sea mayor que 0,999, para evitar inestabilidades cercanas a  $t = T$ .

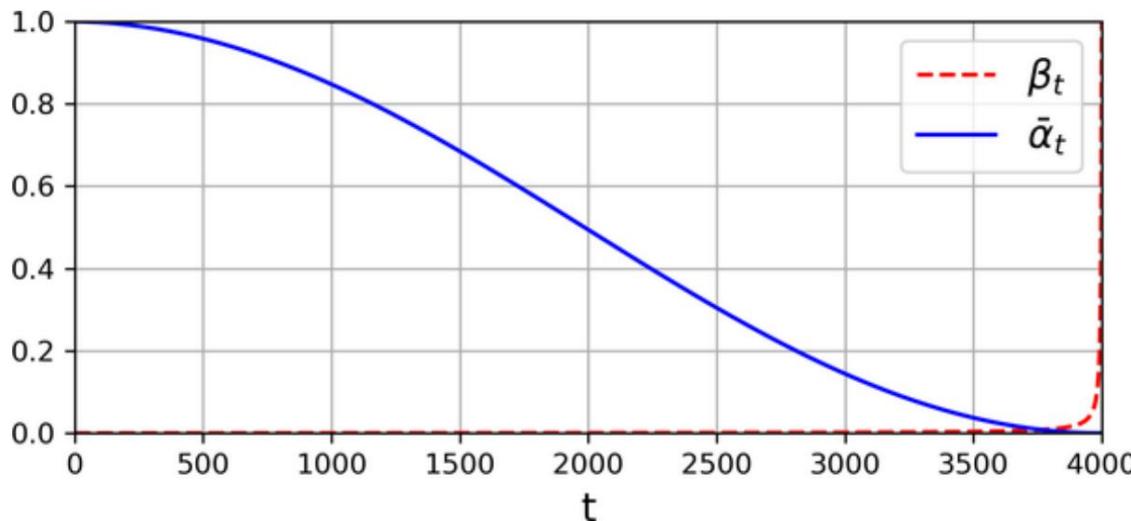


Figura 17-21. Programa de variación de ruido  $\beta$  y variación de señal restante  $\alpha_t$

Creemos una pequeña función para calcular  $\alpha_{4000}$ :  $\beta_t$  y  $\alpha_t$ , y llámalo con  $T =$

```
def varianza_programa(T, s=0,008, max_beta=0,999):
    t = np.arange(T + 1)
    f = np.cos((t / T + s) / (1 + s) * np.pi / 2) ** 2
    alfa = np.clip(f[1:] / f[:-1], 1 - max_beta, 1)
    alfa = np.append(1, alfa).astype(np.float32) # agregar 1
    beta = 1 - alfa
    alfa_cumprod = np.cumprod(alfa)
    devolver alfa, alfa_cumprod, beta #  $\alpha_t$ ,  $\beta_t$  para  $t = 0$ 
a T
T = 4000
alfa, alfa_cumprod, beta = varianza_programa(T)
```

Para entrenar nuestro modelo para revertir el proceso de difusión, necesitaremos ruido. imágenes de diferentes pasos de tiempo del proceso de avance. Para esto vamos a crear una función `prepare_batch()` que tomará un lote de limpieza imágenes del conjunto de datos y prepararlas:

```

def preparar_batch(X):
    X = tf.cast(X[..., tf.newaxis], tf.float32) * 2 - 1 #
    escala de -1 a +1 X_shape =
        tf.shape(X) t =
            tf.random.uniform([X_shape[0]], minval=1, maxval=T + 1, dtype=tf.int32) alpha_cm = tf.
            reunir(alpha_cumprod,
                t) alpha_cm = tf.reshape(alpha_cm, [X_shape[0]] + [1] *
                (len(X_shape) - 1)) ruido = tf.random.normal(X_shape) return { "X_noisy":alpha_cm ** 0,5 * X + (1 -
                alpha_cm) ** 0,5 * ruido, "tiempo": t, }, ruido

```

Repasemos este código:

- Para simplificar usaremos Fashion MNIST, por lo que la función primero debe agregar un eje de canal. También ayudará a escalar los valores de los píxeles de  $-1$  a  $1$ , para que estén más cerca de la distribución gaussiana final con media  $0$  y varianza  $1$ .
- A continuación, la función crea  $t$ , un vector que contiene un paso de tiempo aleatorio para cada imagen del lote, entre  $1$  y  $T$ .
- Luego usa `tf.gather()` para obtener el valor de `alpha_cumprod` para cada uno de los pasos de tiempo en el vector  $t$ . Esto nos da el vector `alpha_cm`, que contiene un valor de  $\alpha_t$  para cada imagen.
- La siguiente línea cambia la forma de `alpha_cm` de `[tamaño de lote]` a `[tamaño de lote, 1, 1, 1]`. Esto es necesario para garantizar que `alpha_cm` se pueda transmitir con el lote `X`.
- Luego generamos algo de ruido gaussiano con media  $0$  y varianza  $1$ .
- Por último, utilizamos [la ecuación 17-6](#) para aplicar el proceso de difusión a las imágenes. Tenga en cuenta que  $x^{0,5}$  es igual a la raíz cuadrada de  $x$ . La función devuelve una tupla que contiene las entradas y los objetivos. Las entradas se representan como un dictado de Python que contiene el ruido.

imágenes y los pasos de tiempo utilizados para generarlas. Los objetivos son el ruido gaussiano utilizado para generar cada imagen.

### NOTA

Con esta configuración, el modelo predecirá el ruido que se debe restar de la imagen de entrada para obtener la imagen original. ¿Por qué no predecir directamente la imagen original? Bueno, los autores lo intentaron: simplemente no funciona tan bien.

A continuación, crearemos un conjunto de datos de entrenamiento y un conjunto de validación que aplicará la función `prepare_batch()` a cada lote. Como antes, `X_train` y `X_valid` contienen las imágenes Fashion MNIST con valores de píxeles que van de 0 a 1:

```
def prepare_dataset(X, tamaño_lote=32, aleatorio=False):
    ds = tf.data.Dataset.from_tensor_slices(X) si se reproduce
    aleatoriamente:
        ds = ds.shuffle(buffer_size=10_000) devuelve
    ds.batch(batch_size).map(prepare_batch).prefetch(1)

train_set = prepare_dataset(X_train, lote_size=32, shuffle=True) valid_set =
    prepare_dataset(X_valid, lote_size=32)
```

Ahora estamos listos para construir el modelo de difusión real. Puede ser cualquier modelo que desee, siempre que tome las imágenes ruidosas y los pasos de tiempo como entradas, y prediga el ruido que se restará de las imágenes de entrada:

```
def build_diffusion_model():
    X_noisy = tf.keras.layers.Input(forma=[28, 28, 1], nombre="X_noisy")
    time_input =
        tf.keras.layers.Input(forma=[], dtype=tf.int32, nombre="hora")
    [...] # construir el modelo basado en las imágenes ruidosas y los pasos de tiempo
    salidas = [...] # predecir el ruido (la misma forma que las imágenes de entrada) return
    tf.keras.Model(entradas=[X_noisy, time_input], salidas=[salidas])
```

Los autores de DDPM utilizaron una arquitectura U-Net modificada,<sup>22</sup> que tiene muchas similitudes con la arquitectura FCN que analizamos en el Capítulo 14 para la segmentación semántica: es una red neuronal convolucional que reduce gradualmente las imágenes de entrada y luego las aumenta gradualmente nuevamente, con conexiones de salto que se cruzan desde cada nivel de la parte de reducción de resolución al correspondiente nivel en la parte de muestreo ascendente. Para tener en cuenta los pasos de tiempo, los codificaron utilizando la misma técnica que las codificaciones posicionales en la arquitectura del transformador (ver Capítulo 16). En todos los niveles de la arquitectura U-Net, pasaron estas codificaciones de tiempo a través de capas densas y las alimentaron a U-Net. Por último, también utilizaron capas de atención de múltiples cabezas en varios niveles. Consulte el cuaderno de este capítulo para obtener una implementación básica o <https://homl.info/ddpmcode> para la implementación oficial: está basado en TF 1.x, que está en desuso, pero es bastante legible.

Ahora podemos entrenar el modelo normalmente. Los autores observaron que utilizar la pérdida MAE funcionó mejor que el MSE. También puedes utilizar la pérdida de Huber:

```
modelo = build_diffusion_model()
model.compile(loss=tf.keras.losses.Huber(), optimizer="nadam") historial =
model.fit(train_set, validation_data=valid_set, epochs=100)
```

Una vez entrenado el modelo, puede usarlo para generar nuevas imágenes. Desafortunadamente, no hay atajos en el proceso de difusión inversa, por lo que hay que muestrear  $x$  aleatoriamente de una distribución gaussiana con media 0 y varianza 1, luego pasarlo al modelo para predecir el ruido; réstalo de la imagen usando la ecuación 17-8 y obtendrás  $x$ . Repita el proceso 3999 veces más hasta obtener  $x$ : si todo salió bien, debería verse como una imagen MNIST de moda normal.

Ecuación 17-8. Un paso atrás en el proceso de difusión

$$x_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left( x_t - \frac{\beta_t}{\sqrt{1-\alpha_t}} \theta(x_t, t) \right) + \sqrt{\beta_t} z$$

En esta ecuación,  $\theta(x_t, t)$  representa el ruido predicho por el modelo dada la imagen de entrada  $x$  y el paso de tiempo  $t$ . La  $\theta$  representa el modelo.

parámetros. Además,  $z$  es ruido gaussiano con media 0 y varianza 1. Esto hace que el proceso inverso sea estocástico: si lo ejecuta varias veces, obtendrá imágenes diferentes.

Escribamos una función que implemente este proceso inverso y llamémosla para generar algunas imágenes:

```
def generar (modelo, tamaño_lote = 32): X = tf.random.normal
    ([tamaño_lote, 28, 28, 1]) para t en el rango (T, 0, -1):

        ruido = (tf.random.normal si t > 1 sino tf.zeros)
        (tf.forma(X))
        X_noisy = modelo({"X_noisy": X, "time": tf.constant([t] * tamaño_lote)})

        X = (
            1 / alfa[t] ** 0,5 * (X - beta[t] / (1
            - alfa_cumprod[t])) ** 0,5 *
            X_ruido)
            + (1 - alfa[t]) ** 0,5 * ruido
        )
        volver X

X_gen = generar (modelo) # imágenes generadas
```

Esto puede tardar uno o dos minutos. Ese es el principal inconveniente de los modelos de difusión: la generación de imágenes es lenta ya que es necesario llamar al modelo muchas veces. Es posible hacer esto más rápido usando un valor  $T$  más pequeño o usando el mismo modelo de predicción para varios pasos a la vez, pero es posible que las imágenes resultantes no se vean tan bien. Dicho esto, a pesar de esta limitación de velocidad, los modelos de difusión producen imágenes diversas y de alta calidad, como se puede ver en [la Figura 17-22](#).



Figura 17-22. Imágenes generadas por el DDPM

Los modelos de difusión han logrado enormes avances recientemente. En particular, un artículo publicado en diciembre de 2021 por [Robin Rombach, Andreas 23 Blattmann y otros](#) introdujeron modelos de difusión latente, donde el proceso de difusión tiene lugar en el espacio latente, en lugar de en el espacio de píxeles. Para lograr esto, se utiliza un potente codificador automático para comprimir cada imagen de entrenamiento en un espacio latente mucho más pequeño, donde tiene lugar el proceso de difusión, luego se utiliza el codificador automático para descomprimir la representación latente final, generando la imagen de salida. Esto acelera considerablemente la generación de imágenes y reduce drásticamente el tiempo y el coste de formación.

Es importante destacar que la calidad de las imágenes generadas es excepcional.

Además, los investigadores también adaptaron varias técnicas de condicionamiento para guiar el proceso de difusión utilizando indicaciones de texto, imágenes o cualquier otra entrada. Esto hace posible producir rápidamente una hermosa imagen de alta resolución de una salamandra leyendo un libro o cualquier otra cosa que se le ocurra. También puede condicionar el proceso de generación de imágenes utilizando una imagen de entrada. Esto permite muchas aplicaciones, como la pintura exterior (donde una imagen de entrada se extiende más allá de sus bordes) o la pintura interior (donde se rellenan los agujeros de una imagen).

Por último, en agosto de 2022 se abrió un potente modelo de difusión latente previamente entrenado llamado Stable Diffusion gracias a una colaboración entre LMU Munich y algunas empresas, incluidas StabilityAI y Runway, con el apoyo de EleutherAI y LAION. En septiembre de 2022, fue

portado a TensorFlow e incluido en [KerasCV](#), una biblioteca de visión por computadora construida por el equipo de Keras. Ahora cualquiera puede generar imágenes alucinantes en segundos, de forma gratuita, incluso en una computadora portátil normal (consulte el último ejercicio de este capítulo). ¡Las posibilidades son infinitas!

En el próximo capítulo pasaremos a una rama completamente diferente del aprendizaje profundo: el aprendizaje por refuerzo profundo.

## Ejercicios

1. ¿Cuáles son las principales tareas para las que se utilizan los codificadores automáticos?
2. Suponga que desea entrenar un clasificador y tiene muchos datos de entrenamiento sin etiquetar, pero solo unos pocos miles de instancias etiquetadas. ¿Cómo pueden ayudar los codificadores automáticos? ¿Cómo procederías?
3. Si un autocodificador reconstruye perfectamente las entradas, ¿es necesariamente un buen autocodificador? ¿Cómo se puede evaluar el rendimiento de un codificador automático?
4. ¿Qué son los codificadores automáticos subcompletos y sobrecompletos? ¿Cuál es el principal riesgo de un codificador automático demasiado incompleto? ¿Qué pasa con el principal riesgo de un codificador automático demasiado completo?
5. ¿Cómo se vinculan los pesos en un codificador automático apilado? Cuál es el punto de hacerlo?
6. ¿Qué es un modelo generativo? ¿Puedes nombrar un tipo de codificador automático generativo?
7. ¿Qué es una GAN? ¿Puedes nombrar algunas tareas en las que las GAN pueden brillar?
8. ¿Cuáles son las principales dificultades a la hora de entrenar GAN?
9. ¿Para qué son buenos los modelos de difusión? ¿Cuál es su principal limitación?
10. Intente utilizar un codificador automático de eliminación de ruido para preparar previamente un clasificador de imágenes.  
Puede usar MNIST (la opción más simple) o un conjunto de datos de imágenes más complejo como [CIFAR10](#) si quieres un desafío mayor. Independientemente del conjunto de datos que esté utilizando, siga estos pasos:

- a. Divida el conjunto de datos en un conjunto de entrenamiento y un conjunto de prueba. Entrene un codificador automático de eliminación de ruido profundo en el conjunto de entrenamiento completo.
  - b. Compruebe que las imágenes estén bastante bien reconstruidas. Visualice las imágenes que más activan cada neurona en la capa de codificación.
  - C. Construya una clasificación DNN, reutilizando las capas inferiores de la codificador automático. Entrénelo usando solo 500 imágenes del conjunto de entrenamiento. ¿Se desempeña mejor con o sin entrenamiento previo?
11. Entrene un codificador automático variacional en el conjunto de datos de imágenes de su elección y utilícelo para generar imágenes. Alternativamente, puede intentar encontrar un conjunto de datos sin etiquetar que le interese y ver si puede generar nuevas muestras.
12. Entrene un DCGAN para que aborde el conjunto de datos de imágenes de su elección y utilícelo para generar imágenes. Agregue la repetición de la experiencia y vea si esto ayuda. Conviértelo en una GAN condicional donde puedas controlar la clase generada.
13. Consulte el excelente [tutorial de Difusión estable de KerasCV](#). y genera un hermoso dibujo de una salamandra leyendo un libro. Si publicas tu mejor dibujo en Twitter, etiquétame en [@aureliengeron](#).  
¡Me encantaría ver tus creaciones!

Las soluciones a estos ejercicios están disponibles al final del cuaderno de este capítulo, en <https://homl.info/colab3>.

---

<sup>1</sup> William G. Chase y Herbert A. Simon, “Percepción en el ajedrez”, *Psicología cognitiva* 4, no. 1 (1973): 55–81.

<sup>2</sup> Yoshua Bengio et al., “Greedy Layer-Wise Training of Deep Networks”, *Actas de la 19<sup>a</sup> Conferencia Internacional sobre Sistemas de Procesamiento de Información Neural* (2006): 153–160.

<sup>3</sup> Jonathan Masci et al., “Codificadores automáticos convolucionales apilados para Feature Extraction”, *Actas de la 21<sup>a</sup> Conferencia Internacional sobre Redes Neuronales Artificiales* 1 (2011): 52–59.

<sup>4</sup> Pascal Vincent et al., “Extracting and Composing Robust Features with Denoising Autoencoders”, *Actas de la 25<sup>a</sup> Conferencia Internacional sobre Aprendizaje Automático* (2008): 1096–1103.

- <sup>5</sup> Pascal Vincent et al., “Codificadores automáticos con eliminación de ruido apilados: aprendizaje útil Representaciones en una red profunda con un criterio de eliminación de ruido local”, *Journal of Machine Learning Research* 11 (2010): 3371–3408.
- <sup>6</sup> Diederik Kingma y Max Welling, “Auto-Encoding Variational Bayes”, preimpresión de arXiv arXiv:1312.6114 (2013).
- <sup>7</sup> Los codificadores automáticos variacionales son en realidad más generales; las codificaciones no se limitan a Distribuciones gaussianas.
- <sup>8</sup> Para obtener más detalles matemáticos, consulte el artículo original sobre variabilidad. codificadores automáticos, o [el gran tutorial](#) de Carl Doersch (2016).
- <sup>9</sup> Ian Goodfellow et al., “Generative Adversarial Nets”, *Actas de la 27<sup>a</sup> Conferencia Internacional sobre Sistemas de Procesamiento de Información Neural* 2 (2014): 2672–2680.
- <sup>10</sup> Para obtener una buena comparación de las principales pérdidas de GAN, consulte este gran proyecto de [GitHub de Hwalsuk Lee](#).
- <sup>11</sup> Mario Lucic et al., “¿Las GAN son creadas iguales? Un estudio a gran escala”, *Actas de la 32<sup>a</sup> Conferencia Internacional sobre Sistemas de Procesamiento de Información Neural* (2018): 698–707.
- <sup>12</sup> Alec Radford et al., “Aprendizaje de representación no supervisado con profundidad Redes adversarias generativas convolucionales”, preimpresión de arXiv arXiv:1511.06434 (2015).
- <sup>13</sup> Reproducido con la amable autorización de los autores.
- <sup>14</sup> Mehdi Mirza y Simon Osindero, “Conditional Generative Adversarial Nets”, preimpresión de arXiv arXiv:1411.1784 (2014).
- <sup>15</sup> Tero Karras et al., “Crecimiento progresivo de GAN para mejorar la calidad, la estabilidad y la variación”, *Actas de la Conferencia Internacional sobre Representaciones del Aprendizaje* (2018).
- <sup>16</sup> El rango dinámico de una variable es la relación entre el valor más alto y el más bajo que puede tomar.
- <sup>17</sup> Tero Karras et al., “Una arquitectura de generador basada en estilos para redes generativas adversarias”, preimpresión de arXiv arXiv:1812.04948 (2018).
- <sup>18</sup> Reproducido con la amable autorización de los autores.
- <sup>19</sup> Jascha Sohl-Dickstein et al., “Aprendizaje profundo no supervisado utilizando el método de desequilibrio Termodinámica”, preimpresión de arXiv arXiv:1503.03585 (2015).
- <sup>20</sup> Jonathan Ho et al., “Modelos probabilísticos de difusión de eliminación de ruido” (2020).
- <sup>21</sup> Alex Nichol y Prafulla Dhariwal, “Probabilística de difusión de eliminación de ruido mejorada Modelos” (2021).

- 22** Olaf Ronneberger et al., “U-Net: Redes convolucionales para imágenes biomédicas Segmentación”, preimpresión de arXiv arXiv:1505.04597 (2015).
- 23** Robin Rombach, Andreas Blattmann, et al., “Síntesis de imágenes de alta resolución con modelos de difusión latente”, preimpresión de arXiv arXiv:2112.10752 (2021).

# Capítulo 18. Aprendizaje por refuerzo

---

El aprendizaje por refuerzo (RL) es uno de los campos del aprendizaje automático más apasionantes en la actualidad, y también uno de los más antiguos. Ha existido desde la década de 1950, produciendo muchas aplicaciones interesantes a lo largo de los años, particularmente en juegos (por ejemplo, TD-Gammon, un programa para jugar Backgammon) y en el control de máquinas, pero rara vez aparece en los titulares de las noticias. Sin embargo, <sup>1</sup>en 2013 se produjo una revolución , cuando investigadores de una startup británica llamada DeepMind demostraron un sistema que podía aprender a jugar casi cualquier juego de Atari desde cero, superando eventualmente a los humanos <sup>2</sup>en la mayoría de ellos, utilizando únicamente píxeles sin procesar como entradas y sin ningún conocimiento previo de las reglas de los juegos<sup>3</sup>. Esta fue la primera de una serie de hazañas asombrosas, que culminaron con la victoria de su sistema AlphaGo contra Lee Sedol, un legendario jugador profesional de Go, en marzo de 2016 y contra Ke Jie, el campeón mundial, en mayo de 2017. El programa había estado alguna vez cerca de vencer a un maestro de este juego, y mucho menos al campeón mundial. Hoy en día todo el campo de la RL está hirviendo de nuevas ideas, con una amplia gama de aplicaciones.

Entonces, ¿cómo logró DeepMind (comprada por Google por más de 500 millones de dólares en 2014) todo esto? En retrospectiva, parece bastante simple: aplicaron el poder del aprendizaje profundo al campo del aprendizaje por refuerzo y funcionó más allá de sus sueños más descabellados. En este capítulo, primero explicaré qué es el aprendizaje por refuerzo y para qué es bueno, luego presentaré dos de las técnicas más importantes en el aprendizaje por refuerzo profundo: gradientes de políticas y redes Q profundas, incluida una discusión sobre los procesos de decisión de Markov.

¡Empecemos!

## Aprender a optimizar las recompensas

En el aprendizaje por refuerzo, un agente de software hace observaciones y realiza acciones dentro de un entorno y, a cambio, recibe recompensas del entorno. Su objetivo es aprender a actuar de una manera que maximice las recompensas esperadas con el tiempo. Si no le importa un poco el antropomorfismo, puede pensar en las recompensas positivas como placer y en las recompensas negativas como dolor (el término "recompensa" es un poco engañoso en este caso). En resumen, el agente actúa en el entorno y aprende por prueba y error a maximizar su placer y minimizar su dolor.

Se trata de un entorno bastante amplio, que puede aplicarse a una amplia variedad de tareas. A continuación se muestran algunos ejemplos (consulte la Figura 18-1):

1. El agente puede ser el programa que controla un robot. En este caso, el entorno es el mundo real, el agente observa el entorno a través de un conjunto de sensores como cámaras y sensores táctiles, y sus acciones consisten en enviar señales para activar motores. Puede programarse para obtener recompensas positivas cada vez que se acerca al destino objetivo y recompensas negativas cada vez que pierde el tiempo o va en la dirección equivocada.
2. El agente puede ser el programa que controla a la Sra. Pac-Man. En este caso, el medio ambiente es un Simulación del juego Atari, las acciones son las nueve posiciones posibles del joystick (arriba a la izquierda, abajo, centro, etc.), las observaciones son capturas de pantalla y las recompensas son solo los puntos del juego.
3. De manera similar, el agente puede ser el programa que juega un juego de mesa como Go. Sólo obtiene una recompensa si gana.
4. El agente no tiene que controlar una cosa que se mueve física (o virtualmente). Por ejemplo, puede ser un termostato inteligente, que obtiene recompensas positivas siempre que se acerca a la temperatura objetivo y ahorra energía, y recompensas negativas cuando los humanos necesitan ajustar la temperatura, por lo que el agente debe aprender a anticipar las necesidades humanas.

5. El agente puede observar los precios del mercado de valores y decidir cuánto comprar o vender cada segundo.

Las recompensas son obviamente las ganancias y pérdidas monetarias.

Tenga en cuenta que es posible que no haya ninguna recompensa positiva; por ejemplo, el agente puede moverse en un laberinto y obtener una recompensa negativa en cada paso, por lo que será mejor que encuentre la salida lo más rápido posible.

Hay muchos otros ejemplos de tareas para las que el aprendizaje por refuerzo es adecuado, como los coches autónomos, los sistemas de recomendación, la colocación de anuncios en una página web o el control de dónde debe centrar su atención un sistema de clasificación de imágenes.

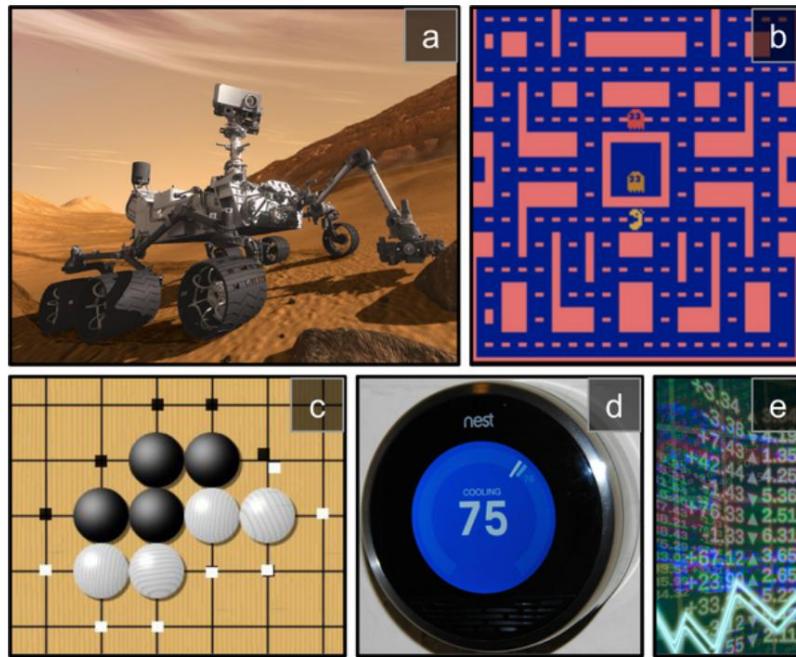


Figura 18-1. Ejemplos de aprendizaje por refuerzo: (a) robótica, (b) Sra. Pac-Man, (c) jugador Go, (d) termostato, (e) comerciante automático

5

### Búsqueda de políticas

El algoritmo que utiliza un agente de software para determinar sus acciones se llama política. La política podría ser una red neuronal que tome observaciones como entradas y genere la acción a realizar (consulte la Figura 18-2).

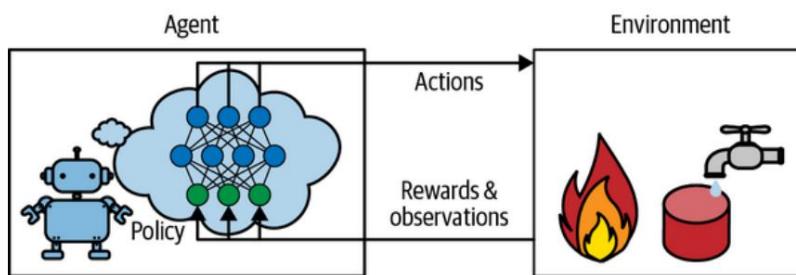


Figura 18-2. Aprendizaje por refuerzo utilizando una política de red neuronal.

La política puede ser cualquier algoritmo que se le ocurra y no tiene por qué ser determinista. De hecho, ¡en algunos casos ni siquiera es necesario observar el entorno! Por ejemplo, pensemos en un robot aspirador cuya recompensa es la cantidad de polvo que recoge en 30 minutos. Su política podría ser avanzar con cierta probabilidad  $p$  cada segundo, o rotar aleatoriamente hacia la izquierda o hacia la derecha con una probabilidad  $1 - p$ . El ángulo de rotación sería un ángulo aleatorio entre  $-r$  y  $+r$ . Dado que esta política implica cierta aleatoriedad, se denomina política estocástica. El robot tendrá una trayectoria errática, lo que garantiza

que eventualmente llegará a cualquier lugar al que pueda llegar y recogerá todo el polvo. La pregunta es ¿cuánto polvo recogerá en 30 minutos?

¿Cómo entrenarías a un robot así? Solo hay dos parámetros de política que puedes modificar: la probabilidad  $p$  y el rango de ángulo  $r$ . Un posible algoritmo de aprendizaje podría ser probar muchos valores diferentes para estos parámetros y elegir la combinación que funcione mejor (consulte la [Figura 18-3](#)). Este es un ejemplo de búsqueda de políticas, en este caso utilizando un enfoque de fuerza bruta. Cuando el espacio de políticas es demasiado grande (que es generalmente el caso), encontrar un buen conjunto de parámetros de esta manera es como buscar una aguja en un pajar gigantesco.

Otra forma de explorar el espacio político es utilizar algoritmos genéticos. Por ejemplo, podrías crear aleatoriamente una primera generación de 100 políticas y probarlas, luego "eliminar" las 80 peores políticas y hacer que los 20 supervivientes produzcan 4 descendientes cada uno. Una descendencia es una copia de su parente más <sup>7</sup>alguna variación aleatoria.

Las políticas supervivientes más sus descendientes juntos constituyen la segunda generación. Puede continuar [iterando](#) a través de generaciones de esta manera hasta que encuentre una buena política.

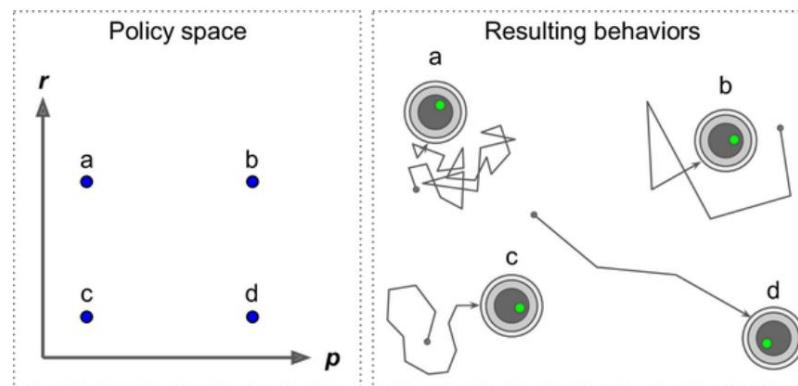


Figura 18-3. Cuatro puntos en el espacio de políticas (izquierda) y el comportamiento correspondiente del agente (derecha)

Otro enfoque más es utilizar técnicas de optimización, evaluando los gradientes de las recompensas con respecto a los parámetros de la política y luego ajustando estos parámetros siguiendo los gradientes hacia recompensas más altas. Analizaremos este <sup>9</sup>enfoque, llamado gradientes de políticas (PG), con más detalle más adelante en este capítulo. Volviendo al robot aspirador, podrías aumentar ligeramente  $p$  y evaluar si al hacerlo aumenta la cantidad de polvo que recoge el robot en 30 minutos; si es así, aumente  $p$  un poco más o reduzca  $p$ . Implementaremos un algoritmo PG popular usando TensorFlow, pero antes de hacerlo, necesitamos crear un entorno en el que viva el agente, por lo que es hora de presentar OpenAI Gym.

## Introducción al gimnasio OpenAI

Uno de los desafíos del aprendizaje por refuerzo es que para capacitar a un agente, primero es necesario tener un ambiente de trabajo. Si deseas programar un agente que aprenda a jugar un juego de Atari, necesitarás un simulador de juego de Atari. Si deseas programar un robot andante, entonces el entorno es el mundo real y puedes entrenar directamente a su robot en ese entorno. Sin embargo, esto tiene sus límites: si el robot se cae por un precipicio, no puedes simplemente hacer clic en Deshacer. Tampoco se puede acelerar el tiempo (añadir más potencia informática no hará que el robot se mueva más rápido) y, en general, es demasiado caro entrenar 1.000 robots en paralelo.

En resumen, el entrenamiento es duro y lento en el mundo real, por lo que generalmente se necesita un entorno simulado al menos para el entrenamiento inicial. Por ejemplo, podrías usar una biblioteca como [PyBullet](#) o [MuJoCo](#) para simulación de física 3D.

[Gimnasio OpenAI](#) es un conjunto de herramientas que proporciona una amplia variedad de entornos simulados (juegos de Atari, juegos de mesa, simulaciones físicas 2D y 3D, etc.), que puedes utilizar para entrenar agentes, compararlos o desarrollar nuevos algoritmos de RL.

OpenAI Gym está preinstalado en Colab, pero es una versión anterior, por lo que deberás reemplazarla por la última. También necesitas instalar algunas de sus dependencias. Si está codificando en su propia máquina en lugar de Colab y siguió las instrucciones de instalación en <https://homl.info/install>, entonces puedes saltarte este paso; de lo contrario, ingrese estos comandos:

```
# ¡Ejecute estos comandos solo en Colab o Kaggle! %pip install
-q -U gymnasio %pip install -q
-U gymnasio[classic_control,box2d,atari,accept-rom-license]
```

El primer comando %pip actualiza Gym a la última versión. La opción -q significa silencio: hace que la salida sea menos detallada. La opción -U significa actualización. El segundo comando %pip instala las bibliotecas necesarias para ejecutar varios tipos de entornos. Esto incluye entornos clásicos de la teoría del control (la ciencia de controlar sistemas dinámicos), como equilibrar un poste en un carro. También incluye entornos basados en la biblioteca Box2D, un motor de física 2D para juegos. Por último, incluye entornos basados en Arcade Learning Environment (ALE), que es un emulador de juegos de Atari 2600. Varias ROM de juegos de Atari se descargan automáticamente y, al ejecutar este código, acepta las licencias de ROM de Atari.

Con eso, estás listo para usar OpenAI Gym. Importémoslo y creemos un entorno:

```
import gymnasio

env = gymnasio.make("CartPole-v1", render_mode="rgb_array")
```

Aquí, hemos creado un entorno CartPole. Esta es una simulación 2D en la que se puede acelerar un carro hacia la izquierda o hacia la derecha para equilibrar un poste colocado encima (consulte [la Figura 18-4](#)). Esta es una tarea de control clásica.

CONSEJO

El diccionario `gym.envs.registry` contiene los nombres y especificaciones de todos los entornos disponibles.

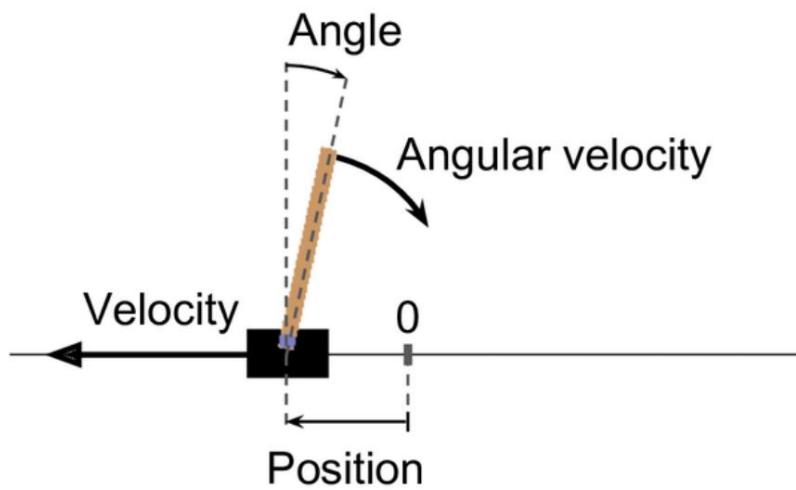


Figura 18-4. El entorno CartPole

Una vez creado el entorno, debe inicializarlo utilizando el método `reset()`, especificando opcionalmente una semilla aleatoria. Esto devuelve la primera observación. Las observaciones dependen del tipo de entorno. Para el entorno CartPole, cada observación es una matriz NumPy 1D que contiene cuatro flotadores que representan la posición horizontal del carro (0,0 = centro), su velocidad (positivo significa derecha), el ángulo del poste (0,0 =

vertical) y su velocidad angular (positiva significa en el sentido de las agujas del reloj). El método `reset()` también devuelve un diccionario que puede contener información adicional específica del entorno. Esto puede resultar útil para depurar o para entrenar. Por ejemplo, en muchos entornos Atari, contiene el número de vidas que quedan. Sin embargo, en el entorno CartPole, este diccionario está vacío.

```
>>> obs, info = env.reset(seed=42) >>> matriz obs
([ 0.0273956
>>> información {} , -0.00611216, 0.03585979, 0.0197368 ], dtype=float32)
```

Llamemos al método `render()` para representar este entorno como una imagen. Dado que configuramos `render_mode="rgb_array"` al crear el entorno, la imagen se devolverá como una matriz NumPy:

```
>>> img = env.render() >>>
img.shape # alto, ancho, canales (3 = Rojo, Verde, Azul) (400, 600, 3)
```

Luego puede usar la función `imshow()` de Matplotlib para mostrar esta imagen, como de costumbre.

Ahora preguntemos al entorno qué acciones son posibles:

```
>>> env.action_space
Discreto(2)
```

Discreto (2) significa que las acciones posibles son números enteros 0 y 1, que representan la aceleración hacia la izquierda o hacia la derecha. Otros entornos pueden tener acciones discretas adicionales u otros tipos de acciones (por ejemplo, continuas). Como el poste está inclinado hacia la derecha (`obs[2] > 0`), aceleraremos el carro hacia la derecha:

```
>>> acción = 1 # acelerar a la derecha >>> obs,
recompensa, hecho, truncado, info = env.step(acción) >>> obs array([ 0.02727336,
0.18847767,
0.03625453, -0.26141977], dtype=float32) >>> recompensa 1.0 >>> hecho Falso >>> truncado Falso >>>
información {}
```

El método `step()` ejecuta la acción deseada y devuelve cinco valores:

observación

Esta es la nueva observación. El carrito ahora se mueve hacia la derecha (`obs[1] > 0`). El polo todavía está inclinado hacia la derecha (`obs[2] > 0`), pero su velocidad angular ahora es negativa (`obs[3] < 0`), por lo que probablemente estará inclinado hacia la izquierda después del siguiente paso.

premio

En este entorno, obtienes una recompensa de 1.0 en cada paso, sin importar lo que hagas, por lo que el objetivo es mantener el episodio funcionando el mayor tiempo posible.

hecho

Este valor será Verdadero cuando finalice el episodio. Esto sucederá cuando el poste se incline demasiado, o se salga de la pantalla, o después de 200 pasos (en este último caso, has ganado). Después de eso, el entorno debe restablecerse antes de poder volver a utilizarlo.

truncado

Este valor será Verdadero cuando un episodio se interrumpa antes de tiempo, por ejemplo, por un contenedor de entorno que impone un número máximo de pasos por episodio (consulte la documentación de Gym para obtener más detalles sobre los contenedores de entorno). Algunos algoritmos de RL tratan los episodios truncados de manera diferente a los episodios terminados normalmente (es decir, cuando finaliza es Verdadero), pero en este capítulo los trataremos de manera idéntica.

información

Este diccionario específico del entorno puede proporcionar información adicional, como la que devuelve el método `reset()`.

CONSEJO

Una vez que haya terminado de usar un entorno, debe llamar a su método `close()` para liberar recursos.

Codifiquemos una política simple que acelere hacia la izquierda cuando el polo se inclina hacia la izquierda y acelere hacia la derecha cuando el polo se inclina hacia la derecha. Ejecutaremos esta política para ver las recompensas promedio que obtiene por cada 500 episodios:

```
def política_básica(obs): ángulo
    = obs[2] devuelve 0
    si ángulo < 0 en caso contrario 1

totales = [] para
el episodio en el rango (500):
    episodio_recompensas = 0
    obs, información = env.reset(seed=episode) para el
    paso en el rango (200): acción =
        basic_policy(obs) obs, recompensa,
        hecho, truncado, información = env.step(action) episodio_rewards +=
            recompensa si se realiza o se trunca:
            pausa

    totales.append(episodio_recompensas)
```

Este código se explica por sí mismo. Veamos el resultado:

```
>>> importar numpy como np
>>> np.mean(totales), np.std(totales), min(totales), max(totales)
(41.698, 8.389445512070509,
 24.0, 63.0)
```

Incluso con 500 intentos, esta política nunca logró mantener el poste en posición vertical durante más de 63 pasos consecutivos. No es bueno. Si observa la simulación en el cuaderno de este capítulo, verá que el carro oscila hacia la izquierda y hacia la derecha cada vez con más fuerza hasta que el poste se inclina demasiado. Veamos si una red neuronal puede generar una política mejor.

## Políticas de redes neuronales

Creemos una política de red neuronal. Esta red neuronal tomará una observación como entrada y generará la acción que se ejecutará, tal como la política que codificamos anteriormente. Más precisamente, estimará una probabilidad para cada acción y luego seleccionaremos una acción al azar, de acuerdo con las probabilidades estimadas (ver [Figura 18-5](#)). En el caso del entorno CartPole, solo hay dos acciones posibles (izquierda o derecha), por lo que solo necesitamos una neurona de salida. Generará la probabilidad  $p$  de la acción 0 (izquierda) y, por supuesto, la probabilidad de la acción 1 (derecha) será  $1 - p$ . Por ejemplo, si genera 0,7, elegiremos la acción 0 con un 70% de probabilidad o la acción 1 con un 30% de probabilidad.

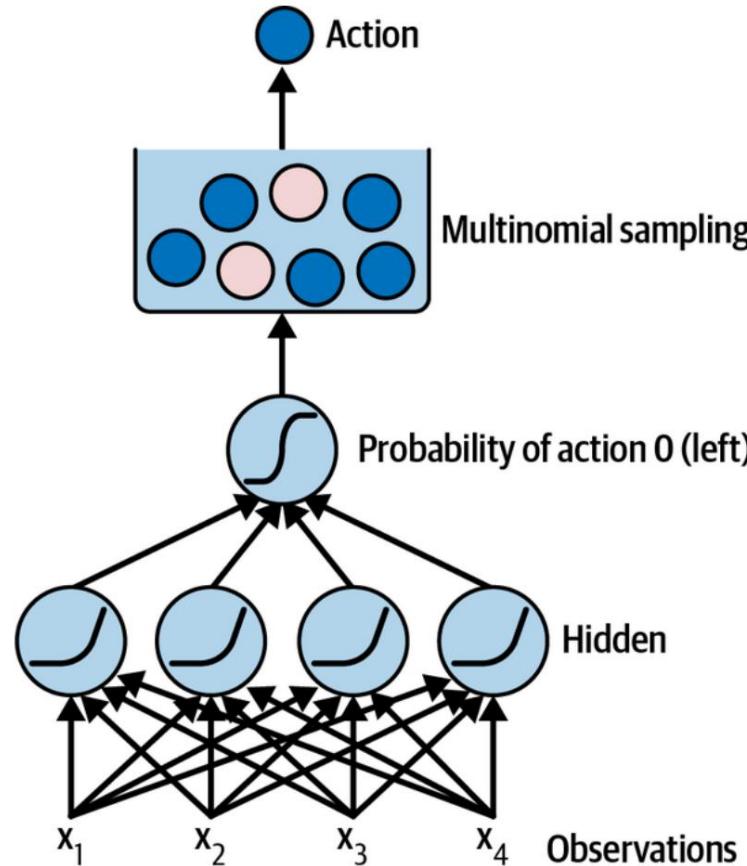


Figura 18-5. Política de redes neuronales

Quizás se pregunte por qué elegimos una acción aleatoria en función de las probabilidades dadas por la red neuronal, en lugar de simplemente elegir la acción con la puntuación más alta. Este enfoque permite al agente encontrar el equilibrio adecuado entre explorar nuevas acciones y explotar las acciones que se sabe que funcionan bien. He aquí una analogía: supongamos que vas a un restaurante por primera vez y todos los platos parecen igualmente atractivos, así que eliges uno al azar. Si resulta bueno, puedes aumentar la probabilidad de que lo pidas la próxima vez, pero no debes aumentar esa probabilidad hasta el 100%, o de lo contrario nunca probarás los otros platos, algunos de los cuales pueden ser incluso mejor que el que probaste. Este dilema de exploración/explotación es central en el aprendizaje por refuerzo.

Tenga en cuenta también que en este entorno particular, las acciones y observaciones pasadas pueden ignorarse con seguridad, ya que cada observación contiene el estado completo del entorno. Si hubiera algún estado oculto, es posible que también deba considerar acciones y observaciones pasadas. Por ejemplo, si el entorno solo revelara la posición del carro pero no su velocidad, tendría que considerar no solo la observación actual sino también la observación anterior para estimar la velocidad actual. Otro ejemplo es cuando las observaciones son ruidosas; en ese caso, generalmente querrás utilizar las últimas observaciones para estimar el estado actual más probable. Por tanto, el problema de CartPole es tan simple como puede ser; las observaciones son libres de ruido y contienen el estado completo del medio ambiente.

Aquí está el código para construir una política de red neuronal básica usando Keras:

```
importar tensorflow como tf

modelo = tf.keras.Sequential([
    tf.keras.layers.Dense(5,
        activación="relu"),
    tf.keras.layers.Dense(1, activación="sigmoide"),
])


```

Utilizamos un modelo secuencial para definir la red de políticas. El número de entradas es el tamaño del espacio de observación (que en el caso de CartPole es 4) y solo tenemos cinco unidades ocultas porque es una tarea bastante simple. Finalmente, queremos generar una probabilidad única (la probabilidad de ir a la izquierda), por lo que tenemos una única neurona de salida que utiliza la función de activación sigmoidea. Si hubiera más de dos acciones posibles, habría una neurona de salida por acción y en su lugar usaríamos la función de activación softmax.

Bien, ahora tenemos una política de red neuronal que tomará observaciones y generará probabilidades de acción. ¿Pero cómo lo entrenamos?

**Evaluación de acciones: el problema de la asignación de crédito** Si supiéramos cuál es la mejor acción en cada paso, podríamos entrenar la red neuronal como de costumbre, minimizando la entropía cruzada entre la distribución de probabilidad estimada y la distribución de probabilidad objetivo. Sería simplemente un aprendizaje supervisado regular. Sin embargo, en el aprendizaje por refuerzo la única guía que recibe el agente es a través de recompensas, y las recompensas suelen ser escasas y retrasadas. Por ejemplo, si el agente logra equilibrar el poste durante 100 pasos, ¿cómo puede saber cuáles de las 100 acciones que realizó fueron buenas y cuáles fueron malas? Lo único que sabe es que el poste cayó después de la última acción, pero seguramente esta última acción no es del todo responsable. Esto se denomina problema de asignación de crédito: cuando el agente obtiene una recompensa, le resulta difícil saber qué acciones deben acreditarse (o culparse) por ello. Piense en un perro que es recompensado horas después de haberse portado bien; ¿Entenderá por qué está siendo recompensado?

Para abordar este problema, una estrategia común es evaluar una acción en función de la suma de todas las recompensas que vienen después, generalmente aplicando un factor de descuento,  $\gamma$  (gamma), en cada paso. Esta suma de recompensas descontadas se denomina retorno de la acción. Considere el ejemplo de [la Figura 18-6](#). Si un agente decide ir a la derecha tres veces seguidas y obtiene una recompensa de +10 después del primer paso, 0 después del segundo paso y finalmente -50 después del tercer paso, suponiendo que usamos un factor de descuento  $\gamma = 0.8$ , el primero La acción tendrá un retorno de  $10 + \gamma \times 0 + \gamma \times (-50) = -22$ . Si el factor de descuento es cercano a 0, las recompensas futuras no contarán mucho en comparación con las recompensas inmediatas. Por el contrario, si el factor de descuento es cercano a 1, las recompensas en el futuro lejano contarán casi tanto como las recompensas inmediatas. Los factores de descuento típicos varían de 0,9 a 0,99. Con un factor de descuento de 0,95, las recompensas de 13 pasos hacia el futuro cuentan aproximadamente la mitad que las recompensas inmediatas (ya que  $0,95 \approx 0,5$ ), mientras que con un factor de descuento de 0,99, las recompensas de 69 pasos hacia el futuro cuentan la mitad que las inmediatas. <sup>13</sup> recompensas. En el entorno CartPole, las acciones tienen efectos a bastante corto plazo, por lo que parece razonable elegir un factor de descuento de 0,95.

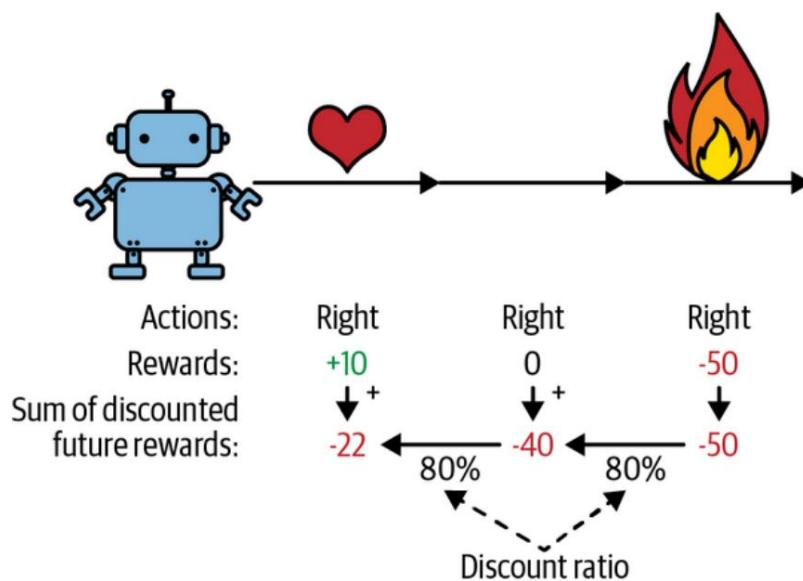


Figura 18-6. Calcular el retorno de una acción: la suma de las recompensas futuras descontadas

Por supuesto, una buena acción puede ir seguida de varias malas acciones que hacen que el poste caiga rápidamente, lo que da como resultado que la buena acción obtenga un bajo rendimiento. De la misma manera, un buen actor a veces puede protagonizar una película terrible. Sin embargo, si jugamos suficientes veces, en promedio las buenas acciones obtendrán un mayor rendimiento que las malas. Queremos estimar cuánto mejor o peor es una acción, en comparación con otras acciones posibles, en promedio. Esto se llama ventaja de acción. Para ello, debemos ejecutar muchos episodios y normalizar todos los retornos de la acción, restando la media y dividiéndolo por la desviación estándar.

Después de eso, podemos suponer razonablemente que las acciones con una ventaja negativa fueron malas mientras que las acciones con una ventaja positiva fueron buenas. Bien, ahora que tenemos una forma de evaluar cada acción, estamos listos para capacitar a nuestro primer agente utilizando gradientes de políticas. Veamos cómo.

## gradientes de políticas

Como se analizó anteriormente, los algoritmos de PG optimizan los parámetros de una política siguiendo los gradientes hacia recompensas más altas. Una clase popular de algoritmos PG, llamados algoritmos REINFORCE, se [introdujo en 1992](#) por Ronald Williams. Aquí hay una variante común:

[11](#)

1. Primero, deje que la política de la red neuronal juegue el juego varias veces y, en cada paso, calcule el gradientes que harían que la acción elegida fuera aún más probable, pero no aplique estos gradientes todavía.
2. Una vez que hayas ejecutado varios episodios, calcula la ventaja de cada acción, usando el método descrito en el apartado anterior.
3. Si la ventaja de una acción es positiva, significa que la acción probablemente fue buena y quieres aplicar los gradientes calculados anteriormente para que sea aún más probable que la acción sea elegida en el futuro. Sin embargo, si la ventaja de la acción es negativa, significa que la acción probablemente fue mala y deseas aplicar los gradientes opuestos para que esta acción sea un poco menos probable en el futuro. La solución es multiplicar cada vector de gradiente por la ventaja de la acción correspondiente.
4. Finalmente, calcule la media de todos los vectores de gradiente resultantes y utilícela para realizar un gradiente.

Usemos Keras para implementar este algoritmo. Entrenaremos la política de red neuronal que construimos anteriormente para que aprenda a equilibrar el poste en el carro. Primero, necesitamos una función que desempeñe un paso. Por ahora fingiremos que cualquier acción que se tome es la correcta para que podamos calcular la pérdida y su

gradientes. Estos gradientes simplemente se guardarán por un tiempo y los modificaremos más adelante dependiendo de qué tan buena o mala resultó la acción:

```
def play_one_step(env, obs, model, loss_fn): con
    tf.GradientTape() como cinta: left_proba =
        model(obs[np.newaxis]) action =
            (tf.random.uniform([1, 1]) > left_proba) y_target = tf.constant([[1.]]) -
            tf.cast(action, tf.float32) pérdida = tf.reduce_mean(loss_fn(y_target, left_proba))

    grads = tape.gradient(loss, model.trainable_variables) obs, recompensa,
    hecho, truncado, info = env.step(int(action)) return obs, recompensa, hecho,
    truncado, grads
```

Repasemos esta función:

- Dentro del bloque GradientTape (ver [Capítulo 12](#)), comenzamos llamando al modelo, dándole una sola observación. Reformamos la observación para que se convierta en un lote que contenga una sola instancia, ya que el modelo espera un lote. Esto genera la probabilidad de ir a la izquierda.
- A continuación, tomamos una muestra de un flotante aleatorio entre 0 y 1 y comprobamos si es mayor que left\_proba. La acción será Falsa con probabilidad left\_proba, o Verdadera con probabilidad  $1 - \text{left\_proba}$ . Una vez que convertimos este booleano en un número entero, la acción será 0 (izquierda) o 1 (derecha) con las probabilidades apropiadas.
- Ahora definimos la probabilidad objetivo de ir a la izquierda: es 1 menos la acción (lanzar a un flotador). Si la acción es 0 (izquierda), entonces la probabilidad objetivo de ir a la izquierda será 1. Si la acción es 1 (derecha), entonces la probabilidad objetivo será 0.
- Luego calculamos la pérdida usando la función de pérdida dada y usamos la cinta para calcular el gradiente de la pérdida con respecto a las variables entrenables del modelo. Nuevamente, estos gradientes se modificarán más adelante, antes de aplicarlos, dependiendo de qué tan buena o mala resultó ser la acción.
- Finalmente, reproducimos la acción seleccionada y devolvemos la nueva observación, la recompensa, si el episodio finalizó o no, si está truncado o no y, por supuesto, los gradientes que acabamos de calcular.

Ahora creemos otra función que se basará en la función play\_one\_step() para reproducir múltiples episodios, devolviendo todas las recompensas y gradientes para cada episodio y cada paso:

```
def play_multiple_episodes(env, n_episodes, n_max_steps, modelo, loss_fn):
    all_rewards = []
    all_grads = [] para
    el episodio dentro del rango (n_episodes):
        current_rewards = []
        current_grads = [] obs,
        info = env.reset() para el paso
        dentro del rango (n_max_steps): obs,
            recompensa, hecho, truncado, graduados = jugar_un_paso(
                env, obs, modelo, loss_fn)
            current_rewards.append(reward)
            current_grads.append(grads) si está
            hecho o truncado: pausa

    all_rewards.append(recompensas_actuales)
    all_grads.append(graduados_actuales)

    devolver todas las recompensas, todos los graduados
```

Este código devuelve una lista de listas de recompensas: una lista de recompensas por episodio, que contiene una recompensa por paso. También devuelve una lista de listas de gradientes: una lista de gradientes por episodio, cada una de las cuales contiene una tupla de gradientes por paso y cada tupla contiene un tensor de gradiente por variable entrenable.

El algoritmo utilizará la función `play_multiple_episodes()` para jugar varias veces (por ejemplo, 10 veces), luego retrocederá y observará todas las recompensas, las descontará y las normalizará. Para hacer eso, necesitamos un par de funciones más; el primero calculará la suma de futuras recompensas descontadas en cada paso, y el segundo normalizará todas estas recompensas descontadas (es decir, los retornos) a lo largo de muchos episodios restando la media y dividiéndola por la desviación estándar:

```
def recompensas_descuento(recompensas, factor_descuento):
    descontado = np.array(recompensas) para el
    paso en el rango(len(recompensas) - 2, -1, -1): desuento [paso] +=
        desuento[paso + 1] * retorno_factor_descuento descontado

def descuento_y_normalizar_recompensas (todas_recompensas, factor_descuento):
    all_discounted_rewards = [discount_rewards(rewards, discount_factor) para recompensas en all_rewards]
    flat_rewards =
        np.concatenate(all_discounted_rewards) recompensa_media = flat_rewards.mean()
    recompensa_std = flat_rewards.std() return
    [(recompensas_descuentos -
        recompensa_media) / recompensa_std
        para recompensas_con descuento en todas_recompensas_con descuento]
```

Comprobemos que esto funciona:

```
>>> recompensas_descuento([10, 0, -50], factor_descuento=0.8) matriz([-22, -40,
-50]) >>>
recompensas_descuento_y_normalización([[10, 0, -50], [10, 20 ]], factor_descuento=0.8)
...
...
[matriz (-0.28435071, -0.86597718, -1.18910299], matriz ([1.26665318,
1.0727777])]
```

La llamada a `discount_rewards()` devuelve exactamente lo que esperamos (ver [Figura 18-6](#)). Puedes verificar que la función `discount_and_normalize_rewards()` efectivamente devuelve las ventajas de acción normalizadas para cada acción en ambos episodios. Observe que el primer episodio fue mucho peor que el segundo, por lo que sus ventajas normalizadas son todas negativas; todas las acciones del primer episodio se considerarían malas y, a la inversa, todas las acciones del segundo episodio se considerarían buenas.

¡Estamos casi listos para ejecutar el algoritmo! Ahora definamos los hiperparámetros. Ejecutaremos 150 iteraciones de entrenamiento, reproduciremos 10 episodios por iteración y cada episodio tendrá una duración máxima de 200 pasos. Usaremos un factor de descuento de 0,95:

```
n_iteraciones = 150
n_episodios_por_actualización = 10
n_max_steps = 200
factor_descuento = 0.95
```

También necesitamos un optimizador y la función de pérdida. Un optimizador Nadam normal con una tasa de aprendizaje de 0,01 funcionará bien y usaremos la función de pérdida de entropía cruzada binaria porque estamos entrenando un clasificador binario (hay dos acciones posibles: izquierda o derecha):

```
optimizador = tf.keras.optimizers.Nadam(learning_rate=0.01) loss_fn =
tf.keras.losses.binary_crossentropy
```

¡Ahora estamos listos para construir y ejecutar el ciclo de capacitación!

```

para iteración en rango(n_iterations): all_rewards,
    all_grads = play_multiple_episodes(
        env, n_episodes_per_update, n_max_steps, modelo, loss_fn) all_final_rewards
        = descuento_y_normalizar_recompensas (todas_recompensas, factor_descuento)

    all_mean_grads = [] para
    var_index en el rango (len (model.trainable_variables)):
        media_graduada = tf.reduce_mean(
            [final_reward * all_grads[episode_index][paso][var_index] para episodio_index,
            final_rewards en enumerate(all_final_rewards) para el paso, final_reward en
            enumerate(final_rewards)], eje=0) all_mean_grads.append(mean_grads)

    optimizador.apply_gradients(zip(all_mean_grads, model.trainable_variables))

```

Repasemos este código:

- En cada iteración de entrenamiento, este bucle llama a la función `play_multiple_episodes()`, que reproduce 10 episodios y devuelve las recompensas y gradientes de cada paso en cada episodio.
- Luego llamamos a la función `discount_and_normalize_rewards()` para calcular la ventaja normalizada de cada acción, llamada `final_reward` en este código. Esto proporciona una medida de qué tan buena o mala fue realmente cada acción, en retrospectiva.
- A continuación, analizamos cada variable entrenable y para cada una de ellas calculamos la media ponderada de los gradientes de esa variable en todos los episodios y todos los pasos, ponderados por la `recompensa_final`.
- Finalmente, aplicamos estos gradientes medios usando el optimizador: las variables entrenables del modelo se modificarán y, con suerte, la política será un poco mejor.

¡Y hemos terminado! Este código entrenará la política de la red neuronal y aprenderá con éxito a equilibrar el poste en el carro. La recompensa media por episodio se acercará mucho a 200. De forma predeterminada, ese es el máximo para este entorno. ¡Éxito!

El algoritmo simple de gradientes de políticas que acabamos de entrenar resolvió la tarea CartPole, pero no se adaptaría bien a tareas más grandes y complejas. De hecho, es muy ineficiente en cuanto a muestras, lo que significa que necesita explorar el juego durante mucho tiempo antes de poder lograr un progreso significativo. Esto se debe a que debe ejecutar varios episodios para estimar la ventaja de cada acción, como hemos visto. Sin embargo, es la base de algoritmos más potentes, como los algoritmos actor-crítico (que discutiremos brevemente al final de este capítulo).

#### CONSEJO

Los investigadores intentan encontrar algoritmos que funcionen bien incluso cuando el agente inicialmente no sabe nada sobre el entorno. Sin embargo, a menos que esté escribiendo un artículo, no debe dudar en inyectar conocimientos previos al agente, ya que acelerará drásticamente la formación. Por ejemplo, como sabes que el poste debe estar lo más vertical posible, puedes agregar recompensas negativas proporcionales al ángulo del poste. Esto hará que las recompensas sean mucho menos escasas y acelerará el entrenamiento. Además, si ya tiene una política razonablemente buena (por ejemplo, codificada), es posible que desee entrenar la red neuronal para que la imite antes de utilizar gradientes de políticas para mejorarla.

Ahora veremos otra familia popular de algoritmos. Mientras que los algoritmos PG intentan optimizar directamente la política para aumentar las recompensas, los algoritmos que exploraremos ahora son menos directos: el agente aprende a estimar el rendimiento esperado para cada estado, o para cada acción en cada estado, y luego utiliza este conocimiento para decidir. como actuar. Para comprender estos algoritmos, primero debemos considerar los procesos de decisión de Markov (MDP).

## Procesos de decisión de Markov

A principios del siglo XX, el matemático Andrey Markov estudió procesos estocásticos sin memoria, llamadas cadenas de Markov. Este proceso tiene un número fijo de estados y evoluciona aleatoriamente de un estado a otro en cada paso. La probabilidad de que evolucione de un estado  $s$  a un estado  $s'$  es fija, y depende sólo del par  $(s, s')$ , no de estados pasados. Por eso decimos que el sistema no tiene memoria.

La figura 18-7 muestra un ejemplo de una cadena de Markov con cuatro estados.

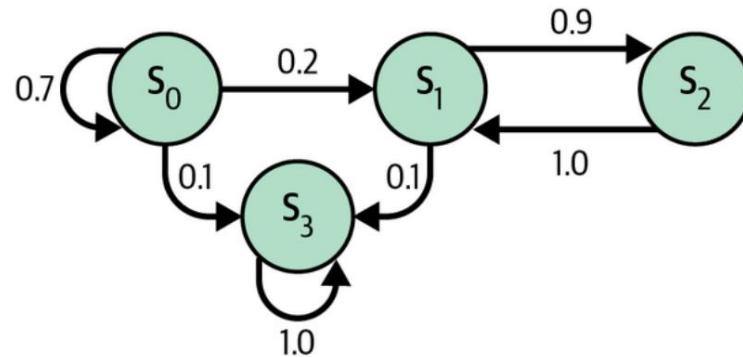


Figura 18-7. Ejemplo de cadena de Markov

Supongamos que el proceso comienza en el estado  $s_0$ , y hay un 70% de posibilidades de que permanezca en ese estado en el siguiente paso. Al final, está obligado a abandonar ese estado y no volver jamás, porque ningún otro estado apunta de nuevo al  $s_0$ . Si va al estado  $s_1$ , entonces lo más probable es que vaya al estado  $s_2$  (90% de probabilidad), entonces inmediatamente regresa al estado  $s_1$  (con 100% de probabilidad). Puede alternar varias veces entre estos dos estados, pero eventualmente caerá en el estado  $s_3$  y permanecerá allí para siempre, ya que no hay salida: esto se llama estado terminal. Las cadenas de Markov pueden tener dinámicas muy diferentes y se utilizan mucho en termodinámica, química, estadística y mucho más.

Los procesos de decisión de Markov fueron descritos por primera vez en la década de 1950 por [Richard Bellman](#). Se parecen <sup>12</sup> a las Cadenas de Markov, pero con un giro: en cada paso, un agente puede elegir una de varias acciones posibles, y las probabilidades de transición dependen de la acción elegida. Además, algunas transiciones estatales devuelven algunos recompensa (positiva o negativa), y el objetivo del agente es encontrar una política que maximice la recompensa con el tiempo.

Por ejemplo, el MDP representado en la Figura 18-8 tiene tres estados (representados por círculos) y hasta tres posibles acciones discretas en cada paso (representadas por diamantes).

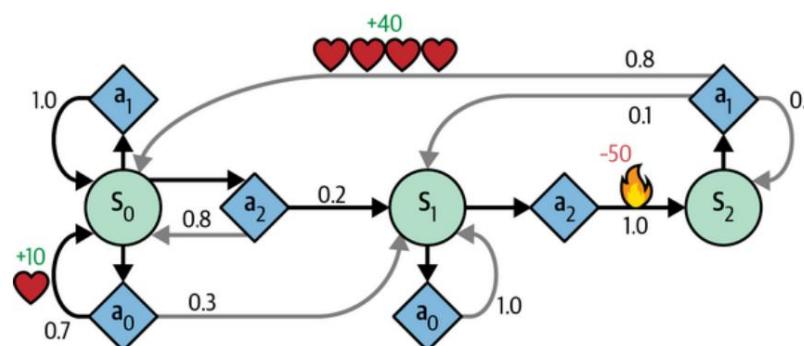


Figura 18-8. Ejemplo de un proceso de decisión de Markov

Si comienza en el estado  $s_0$ , el agente puede elegir entre acciones  $a_0$  (0.7),  $a_1$  (1.0) o  $a_2$  (0.8). Si elige acción  $a_1$ , es solo permanece en el estado  $s_0$  con certeza y sin recompensa alguna. Por tanto, puede decidir quedarse allí para siempre si quiere. Pero si elige acción  $a_0$ , tiene un 70% de probabilidad de obtener una recompensa de +10 y permanecer en estado  $s_0$ . Luego puede intentar una y otra vez obtener la mayor recompensa posible, pero en un momento se va.

para terminar en el estado  $s_0$ . En el estado  $s_0$  sólo tiene dos acciones posibles:  $a_0$  o  $a_1$ . Puede optar por quedarse dicho eligiendo repetidamente la acción  $a_0$ , o puede optar por pasar al estado  $s_1$  y obtener una recompensa negativa  $-50$  (ay). En los estados  $s_1$  no tiene más remedio que actuar  $a_1$ , lo que probablemente lo llevará de regreso al estado  $s_0$ , obteniendo una recompensa de  $+40$  en el camino. Te dan la imagen. Al mirar este MDP, ¿puedes adivinar? ¿Qué estrategia obtendrá la mayor recompensa con el tiempo? En el estado  $s_0$  es claro que la acción  $a_0$  es la mejor opción, y en los estados  $s_1$  el agente no tiene más opción que tomar la acción  $a_1$  pero en los estados  $s_1$  no es obvio si el agente debe quedarse quieto ( $a_0$ ) o atravesar el fuego ( $a_1$ ). 2

Bellman encontró una manera de estimar el valor de estado óptimo de cualquier estado  $s$ , anotó  $V^*(s)$ , que es la suma de todas las recompensas futuras con descuento que el agente puede esperar en promedio después de llegar al estado, suponiendo que actúa de manera óptima. Demostró que si el agente actúa de manera óptima, entonces se aplica la ecuación de optimización de Bellman. (ver [Ecuación 18-1](#)). Esta ecuación recursiva dice que si el agente actúa de manera óptima, entonces el valor óptimo del estado actual es igual a la recompensa que obtendrá en promedio después de realizar una acción óptima, más la Valor óptimo esperado de todos los posibles estados siguientes a los que esta acción puede conducir.

Ecuación 18-1. Ecuación de optimización de Bellman

$$V^*(s) = \max_{a \in A(s)} \sum_{s' \in S} T(s, a, s') [R(s, a, s') + \gamma V^*(s')] \text{ para todos los } s$$

En esta ecuación:

- $T(s, a, s')$  es la probabilidad de transición del estado  $s$  al estado  $s'$ , dado que el agente eligió la acción  $a$ . Por ejemplo, en [la figura 18-8](#),  $T(s_0, a_0, s_1) = 0.8$ .
- $R(s, a, s')$  es la recompensa que obtiene el agente cuando pasa del estado  $s$  al estado  $s'$ , dado que el agente eligió la acción  $a$ . Por ejemplo, en [la figura 18-8](#),  $R(s_0, a_0, s_1) = +40$ .
- $\gamma$  es el factor de descuento.

Esta ecuación conduce directamente a un algoritmo que puede estimar con precisión el valor de estado óptimo de cada estado posible: primero inicialice todas las estimaciones de valores de estado a cero y luego actualícelas iterativamente usando el algoritmo de iteración de valores (ver [Ecuación 18-2](#)). Un resultado notable es que, con el tiempo suficiente, estos Se garantiza que las estimaciones convergerán a los valores estatales óptimos, correspondientes a la política óptima.

Ecuación 18-2. Algoritmo de iteración de valor

$$V^{k+1}(s) \leftarrow \max_{a \in A(s)} \sum_{s' \in S} T(s, a, s') [R(s, a, s') + \gamma V^k(s')] \text{ para todo } s$$

En esta ecuación,  $V^k(s)$  es el valor estimado del estado  $s$  en la  $k$  iteración del algoritmo.<sup>th</sup>

#### NOTA

Este algoritmo es un ejemplo de programación dinámica, que descompone un problema complejo en soluciones manejables. subproblemas que pueden abordarse de forma iterativa.

Conocer los valores estatales óptimos puede ser útil, en particular para evaluar una política, pero no nos proporciona la política óptima para el agente. Afortunadamente, Bellman encontró un algoritmo muy similar para estimar el valor óptimo. valores estado-acción, generalmente llamados valores  $Q$  (valores de calidad). El valor  $Q$  óptimo de la acción del estado. El par  $(s, a)$ , anotado como  $Q^*(s, a)$ , es la suma de las recompensas futuras descontadas que el agente puede esperar en promedio. después de que alcanza el estado  $s$  y elige la acción  $a$ , pero antes de ver el resultado de esta acción, suponiendo actúa de manera óptima después de esa acción.

Veamos cómo funciona. Una vez más, se comienza inicializando todas las estimaciones del valor Q a cero y luego se actualizan utilizando el algoritmo de iteración del valor Q (consulte la ecuación 18-3).

Ecuación 18-3. Algoritmo de iteración del valor Q

$$Q_{k+1}(s, a) \leftarrow \sum s' T(s, a, s') [R(s, a, s') + \gamma \max_a Q_k(s', a')] \text{ para todo } (s, a)$$

Una vez que se tienen los valores Q óptimos, definir la política óptima, anotada como  $\pi^*(s)$ , es trivial; cuando el agente está en el estado s, debe elegir la acción con el valor Q más alto para ese estado:

$$\pi^*(s) = \operatorname{argmax}_a Q^*(s, a).$$

Apliquemos este algoritmo al MDP representado en la Figura 18-8. Primero, necesitamos definir el MDP:

```
probabilidades_de_transición = [ # forma=[s, a, s']
    [[0.7, 0.3, 0.0], [1.0, 0.0, 0.0], [0.8, 0.2, 0.0]], [[0.0, 1.0, 0.0], Ninguno, [0.0, 0.0, 1.0]],
    [Ninguno, [0.8, 0.1, 0.1], Ninguno]

] recompensas = [ # forma=[s, a, s'] [+10, 0, 0], [0,
    0, 0], [0, 0, 0]], [[0, 0, 0], [0, 0, 0], [0, 0, -50]], [[0, 0, 0],
    [+40, 0, 0], [0, 0, 0]]

] posibles_acciones = [[0, 1, 2], [0, 2], [1]]
```

Por ejemplo, para conocer la probabilidad de transición de pasar de s a  $s'$  después de realizar la acción  $a$ , miraremos  $a$  en  $\text{Transition\_probabilities}[2][1][0]$  (que es 0,8). De manera similar, para obtener la recompensa correspondiente, buscaremos  $\text{recompensas}[2][1][0]$  (que es +40). Y para obtener la lista de posibles acciones en s buscaremos  $\text{posibles\_acciones}[2]$  (en este caso, solo es posible la acción  $a$ ). A continuación, debemos inicializar todos los valores Q a cero (excepto las acciones imposibles, para las cuales establecemos los valores Q en  $-\infty$ ):

```
Q_values = np.full((3, 3), -np.inf) # -np.inf para acciones imposibles para el estado, acciones en enumerar
(possibles_acciones):
    Q_values[estado, acciones] = 0.0 # para todas las acciones posibles
```

Ahora ejecutemos el algoritmo de iteración del valor Q. Aplica la Ecuación 18-3 repetidamente, a todos los valores Q, para cada estado y cada acción posible:

```
gamma = 0.90 # el factor de descuento

para iteración en rango(50): Q_prev =
    Q_values.copy() para s en rango(3):
        para a en posibles_acciones[s]:
            Q_values[s, a] = np.sum([ Transition_probabilities[s]
                [a] [sp] * (recompensas[s][a][sp] + gamma
                * Q_prev[sp].max()) para sp en el rango(3)])
```

¡Eso es todo! Los valores Q resultantes se ven así:

```
>>> Matriz Q_values
([[18.91891892, 17.02702702, 13.62162162], -inf, -4.87971488], -inf,
 [0. [ , 50.13365013,
 -inf]])
```

Por ejemplo, cuando el agente está en el estado  $s$  y elige una acción, la recompensa  $1$ , la suma esperada de descuento futura es aproximadamente  $17,0$ .

Para cada estado, podemos encontrar la acción que tiene el valor  $Q$  más alto:

```
>>> Q_values.argmax(axis=1) # acción óptima para cada matriz de estado([0, 0, 1])
```

Esto nos da la política óptima para este MDP cuando usamos un factor de descuento de  $0,90$ : en el estado  $s$ , elija la acción  $0$  (una acción  $0$ ), en el estado  $s$ , elija la acción  $a$  (es decir, quedarse quieto), y en el estado  $s_2$  elija la acción  $a$  (la única opción posible).

Curiosamente, si aumentamos el factor de descuento a  $0,95$ , la política óptima cambia: en el estado  $s$  la mejor acción se convierte en  $1$  en (pasar por el fuego!). Esto tiene sentido porque cuanto más valoras las recompensas futuras, más dispuesto estás a soportar algo de dolor ahora por la promesa de una felicidad futura.

## Aprendizaje de diferencias temporales

Los problemas de aprendizaje por refuerzo con acciones discretas a menudo pueden modelarse como procesos de decisión de Markov, pero el agente inicialmente no tiene idea de cuáles son las probabilidades de transición (no conoce  $T(s, a, s')$ ) y no sabe cuáles son las probabilidades de transición. las recompensas serán cualquiera de las dos (no conoce  $R(s, a, s')$ ). Debe experimentar cada estado y cada transición al menos una vez para conocer las recompensas, y debe experimentarlas varias veces si quiere tener una estimación razonable de las probabilidades de transición.

El algoritmo de aprendizaje de diferencia temporal (TD) es muy similar al algoritmo de iteración del valor  $Q$ , pero modificado para tener en cuenta el hecho de que el agente sólo tiene un conocimiento parcial del MDP. En general suponemos que el agente inicialmente sólo conoce los posibles estados y acciones, y nada más. El agente utiliza una política de exploración (por ejemplo, una política puramente aleatoria) para explorar el MDP y, a medida que avanza, el algoritmo de aprendizaje TD actualiza las estimaciones de los valores del estado en función de las transiciones y recompensas que realmente se observan (consulte la Ecuación 18) . -4).

Ecuación 18-4. Algoritmo de aprendizaje TD

$$V_{k+1}(s) \leftarrow (1 - \alpha)V_k(s) + \alpha(r + \gamma V_k(s')), \text{ o, equivalentemente: } V_{k+1}(s) \leftarrow V_k(s) + \alpha \delta_k(s, r, s')w$$

En esta ecuación:

- $\alpha$  es la tasa de aprendizaje (por ejemplo,  $0,01$ ).
- $r + \gamma \cdot V(s')$  se denomina objetivo TD.
- $\delta_k(s, r, s')$  se llama error TD.

Una forma más concisa de escribir la primera forma de esta ecuación es usar la notación  $a \leftarrow a \cdot b$ , lo que significa  $a \leftarrow (1 - \alpha) \cdot a + \alpha \cdot b$ . Entonces, la primera línea de la ecuación 18-4 se puede reescribir así:

$$V(s) \leftarrow \alpha r + \gamma V(s').$$

### CONSEJO

El aprendizaje TD tiene muchas similitudes con el descenso de gradiente estocástico, incluido el hecho de que maneja una muestra a la vez. Además, al igual que SGD, solo puede converger verdaderamente si se reduce gradualmente la tasa de aprendizaje; de lo contrario, seguirá rebotando alrededor de los valores  $Q$  óptimos.

Para cada estado , este algoritmo realiza un seguimiento de un promedio móvil de las recompensas inmediatas que obtiene el agente al abandonar ese estado, más las recompensas que espera obtener más adelante, suponiendo que actúa de manera óptima.

### **Q-Learning De**

manera similar, el algoritmo Q-learning es una adaptación del algoritmo de iteración del valor Q a la situación en la que las probabilidades de transición y las recompensas se desconocen inicialmente (consulte la [ecuación 18-5](#)). El Q-learning funciona observando a un agente jugar (por ejemplo, de forma aleatoria) y mejorando gradualmente sus estimaciones de los valores Q. Una vez que tiene estimaciones precisas del valor Q (o lo suficientemente cercanas), entonces la política óptima es simplemente elegir la acción que tiene el valor Q más alto (es decir, la política codiciosa).

Ecuación 18-5. Algoritmo de aprendizaje Q

$$Q(s, a) \leftarrow \alpha r + \gamma \max_{a'} Q(s', a')$$

Para cada par estado-acción ( $s, a$ ), este algoritmo realiza un seguimiento de un promedio móvil de las recompensas  $r$  que obtiene el agente al abandonar el estado  $s$  con la acción  $a$ , más la suma de las recompensas futuras descontadas que espera obtener. Para estimar esta suma, tomamos el máximo de las estimaciones del valor Q para el siguiente estado  $s'$ , ya que suponemos que la política objetivo actuará de manera óptima a partir de ese momento.

Implementemos el algoritmo Q-learning. Primero, necesitaremos hacer que un agente explore el entorno.

Para esto, necesitamos una función de paso para que el agente pueda ejecutar una acción y obtener el estado resultante y la recompensa:

```
def paso(estado, acción):
    probas = probabilidades_de_transición[estado][acción] estado_siguiente
    = np.random.choice([0, 1, 2], p=probas) recompensa = recompensas[estado]
    [acción][estado_siguiente] devolver estado_siguiente, recompensa
```

Ahora implementemos la política de exploración del agente. Dado que el espacio de estados es bastante pequeño, una simple política aleatoria será suficiente. Si ejecutamos el algoritmo durante el tiempo suficiente, el agente visitará cada estado muchas veces y también intentará todas las acciones posibles muchas veces:

```
def política_exploración(estado):
    devolver np.random.choice(posibles_acciones[estado])
```

A continuación, después de inicializar los valores Q como antes, estamos listos para ejecutar el algoritmo Q-learning con caída de la tasa de aprendizaje (usando la programación de energía, presentada en el Capítulo 11 ) :

```
alfa0 = 0,05 # caída de la tasa de aprendizaje inicial =
0,005 # caída de la tasa de aprendizaje gamma = 0,90
# estado del factor de descuento = 0 # estado
inicial

para iteración en el rango (10_000):
    acción = política_exploración(estado)
    siguiente_estado, recompensa = paso(estado, acción)
    siguiente_valor = Q_valores[siguiente_estado].max() # política codiciosa en el siguiente paso alfa = alfa0 / (1 +
    iteración * decaimiento)
    Valores_Q[estado, acción] *= 1 - alfa Valores_Q[estado,
    acción] += alfa * (recompensa + estado gamma = estado_siguiente
    * valor_siguiente)
```

Este algoritmo convergerá a los valores Q óptimos, pero requerirá muchas iteraciones y posiblemente bastante ajuste de hiperparámetros. Como puede ver en [la Figura 18-9](#), el algoritmo de iteración del valor Q (izquierda) converge muy rápidamente, en menos de 20 iteraciones, mientras que el algoritmo de aprendizaje Q (derecha) necesita alrededor de 8000 iteraciones para converger. Obviamente, no conocer las probabilidades de transición o las recompensas hace que encontrar la política óptima sea mucho más difícil.

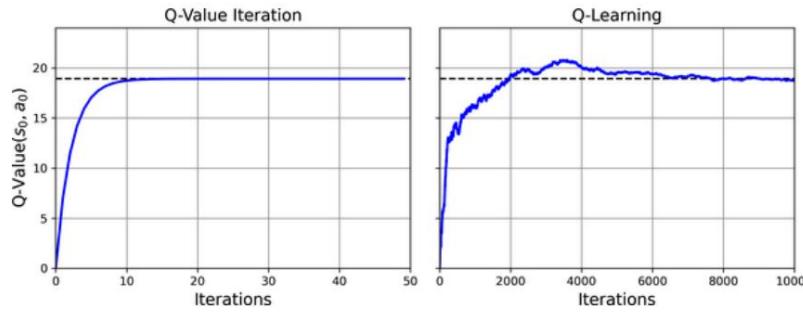


Figura 18-9. Curva de aprendizaje del algoritmo de iteración del valor Q versus el algoritmo Q-learning

El algoritmo Q-learning se denomina algoritmo fuera de políticas porque la política que se entrena no es necesariamente la que se utiliza durante el entrenamiento. Por ejemplo, en el código que acabamos de ejecutar, la política que se estaba ejecutando (la política de exploración) era completamente aleatoria, mientras que la política que se estaba entrenando nunca se usó. Después del entrenamiento, la política óptima corresponde a elegir sistemáticamente la acción con el valor Q más alto. Por el contrario, el algoritmo de gradientes de políticas es un algoritmo basado en políticas : explora el mundo utilizando la política que se está entrenando. Es algo sorprendente que Q-learning sea capaz de aprender la política óptima con solo observar a un agente actuar al azar. Imagínate aprender a jugar golf cuando tu profesor es un mono con los ojos vendados. ¿Podemos hacerlo mejor?

## Políticas de exploración

Por supuesto, Q-learning sólo puede funcionar si la política de exploración explora el MDP lo suficientemente a fondo. Aunque se garantiza que una política puramente aleatoria eventualmente visitará cada estado y cada transición muchas veces, puede llevar mucho tiempo hacerlo. Por lo tanto, una mejor opción es utilizar la política  $\epsilon$ -codiciosa ( $\epsilon$  es épsilon): en cada paso actúa aleatoriamente con probabilidad  $\epsilon$ , o con avidez con probabilidad  $1-\epsilon$  (es decir, eligiendo la acción con el valor Q más alto). La ventaja de la política  $\epsilon$ -codiciosa (en comparación con una política completamente aleatoria) es que pasará cada vez más tiempo explorando las partes interesantes del entorno, a medida que las estimaciones del valor Q mejoran cada vez más, sin dejar de dedicar algo de tiempo a visitar Regiones desconocidas del MDP. Es bastante común comenzar con un valor alto para  $\epsilon$  (por ejemplo, 1,0) y luego reducirlo gradualmente (por ejemplo, hasta 0,05).

Alternativamente, en lugar de depender únicamente de la oportunidad para la exploración, otro enfoque es alentar a la política de exploración a intentar acciones que no ha intentado mucho antes. Esto se puede implementar como una bonificación agregada a las estimaciones del valor Q, como se muestra en la ecuación 18-6.

Ecuación 18-6. Q-learning utilizando una función de exploración

$$Q(s, a) \leftarrow \alpha r + \gamma \max_{a'} f(Q(s', a'), N(s', a'))$$

En esta ecuación:

- $N(s', a')$  cuenta el número de veces que se eligió la acción  $a'$  en el estado  $s'$ .
- $f(Q, N)$  es una función de exploración, como  $f(Q, N) = Q + \kappa/(1 + N)$ , donde  $\kappa$  es un hiperparámetro de curiosidad que mide cuánto se siente atraído el agente por lo desconocido.

## Q-Learning aproximado y Q-Learning profundo

El principal problema con Q-learning es que no se adapta bien a MDP grandes (o incluso medianos) con muchos estados y acciones. Por ejemplo, supongamos que desea utilizar Q-learning para entrenar a un agente para que interprete a la Sra. Pac-Man (consulte la Figura 18-1). Hay alrededor de 150 bolitas que la Sra. Pac-Man puede comer, cada una de las cuales puede estar presente o ausente (es decir, ya consumida). Entonces, el número de estados posibles es mayor que  $2^{150} \approx 10^{45}$

<sup>1045</sup> Y si sumas todas las combinaciones posibles de posiciones para todos los fantasmas y la Sra. Pac-Man, la cantidad de estados posibles se vuelve mayor que la cantidad de átomos en nuestro planeta, por lo que no hay absolutamente ninguna manera de que puedas realizar un seguimiento de una estimación de cada valor Q.

La solución es encontrar una función Q ( $s_\theta a$ ) que se aproxime al valor Q de cualquier par estado-acción (s, a) utilizando un número manejable de parámetros (dado por el vector de parámetros  $\theta$ ). Esto se llama Q-learning aproximado. Durante años se recomendó utilizar combinaciones lineales de características artesanales extraídas del estado (por ejemplo, las distancias de los fantasmas más cercanos, sus direcciones, etc.) para estimar los valores Q, pero en 2013, DeepMind demostró que el uso de redes neuronales profundas puede funcionar mucho mejor, especialmente para problemas complejos, y no requiere ninguna ingeniería de funciones. Un DNN utilizado para estimar los valores Q se denomina red Q profunda (DQN), y el uso de un DQN para un aprendizaje Q aproximado se denomina aprendizaje Q profundo.

Ahora bien, ¿cómo podemos entrenar un DQN? Bueno, considere el valor Q aproximado calculado por el DQN para un par estado-acción dado (s, a). Gracias a Bellman, sabemos que queremos que este valor Q aproximado sea lo más cercano posible a la recompensa r que realmente observamos después de realizar la acción a en el estado s, más el valor descontado de jugar de manera óptima a partir de ese momento. Para estimar esta suma de recompensas descontadas futuras, podemos simplemente ejecutar el DQN en el siguiente estado s', para todas las acciones posibles a'. Obtenemos un valor Q futuro aproximado para cada acción posible. Luego elegimos el más alto (ya que asumimos que jugaremos de manera óptima) y lo descontamos, y esto nos da una estimación de la suma de futuras recompensas descontadas. Al sumar la recompensa r y la estimación del valor descontado futuro, obtenemos un valor Q objetivo y(s, a) para el par estado-acción (s, a), como se muestra en la ecuación 18-7.

Ecuación 18-7. Valor Q objetivo

$$y(s, a) = r + \gamma \max_{a'} Q(s', a')$$

Con este valor Q objetivo, podemos ejecutar un paso de entrenamiento utilizando cualquier algoritmo de descenso de gradiente. Específicamente, generalmente intentamos minimizar el error al cuadrado entre el valor Q estimado Q (s, a) y el valor Q objetivo y (s, a), o la pérdida de Huber para reducir la sensibilidad del algoritmo a errores grandes. ¡Y ese es el algoritmo de Q-learning profundo! Veamos cómo implementarlo para resolver el entorno CartPole.

## Implementación de Q-Learning profundo

Lo primero que necesitamos es una red Q profunda. En teoría, necesitamos una red neuronal que tome un par estado-acción como entrada y genere un valor Q aproximado. Sin embargo, en la práctica es mucho más eficiente utilizar una red neuronal que solo toma un estado como entrada y genera un valor Q aproximado para cada acción posible. Para resolver el entorno CartPole, no necesitamos una red neuronal muy complicada; un par de capas ocultas servirán:

```
input_shape = [4] # == env.observation_space.shape n_outputs = 2 # ==
env.action_space.n

modelo = tf.keras.Sequential([
    tf.keras.layers.Dense(32,
        activation="elu", input_shape=input_shape),
    tf.keras.layers.Dense(32, activation="elu"),
    tf.keras.layers.Dense(n_outputs)
])
```

Para seleccionar una acción usando este DQN, elegimos la acción con el valor Q previsto más grande. Para asegurar que el agente explore el entorno, usaremos una política  $\epsilon$ -codiciosa (es decir, elegiremos una acción aleatoria con probabilidad  $\epsilon$ ):

```
def epsilon_greedy_policy(estado, epsilon=0):
    if np.random.rand() < epsilon:
```

```

    devolver np.random.randint(n_outputs) # acción aleatoria más:

    Q_values = model.predict(state[np.newaxis], verbose=0)[0] return Q_values.argmax()
    # acción óptima según el DQN

```

En lugar de entrenar el DQN basándose únicamente en las últimas experiencias, almacenaremos todas las experiencias en un búfer de reproducción (o memoria de reproducción) y tomaremos muestras de un lote de entrenamiento aleatorio en cada iteración de entrenamiento. Esto ayuda a reducir las correlaciones entre las experiencias en un lote de capacitación, lo que ayuda enormemente a la capacitación. Para esto, usaremos una cola de doble extremo (deque):

```

de colecciones importar deque

replay_buffer = deque(maxlen=2000)

```

## CONSEJO

Una deque es una cola que se puede agregar o eliminar de manera eficiente en ambos extremos. Insertar y eliminar elementos al final de la cola es muy rápido, pero el acceso aleatorio puede ser lento cuando la cola se vuelve larga. Si necesita un búfer de reproducción muy grande, debe usar un búfer circular (consulte el cuaderno para ver una implementación) o consulte la [biblioteca Reverb de DeepMind](#).

Cada experiencia estará compuesta por seis elementos: un estado s, la acción a que realizó el agente, la recompensa resultante r, el siguiente estado s' que alcanzó, un booleano que indica si el episodio terminó en ese punto (hecho) y finalmente otro booleano que indica si el episodio se truncó en ese momento. Necesitaremos una pequeña función para muestrear un lote aleatorio de experiencias del búfer de reproducción. Devolverá seis matrices NumPy correspondientes a los seis elementos de la experiencia:

```

def experiencias_muestra(tamaño_lote):
    índices = np.random.randint(len(replay_buffer), tamaño=batch_size) lote = [replay_buffer[index]
    para índice en índices] return [ np.array([experiencia[field_index] para
    experiencia
    en lote]) para field_index en rango (6) ] # [estados, acciones, recompensas,
    próximos_estados, hechos, truncados]

```

También creemos una función que reproducirá un solo paso usando la política  $\epsilon$ -greedy y luego almacenaremos la experiencia resultante en el búfer de reproducción:

```

def play_one_step(env, state, épsilon): acción =
    epsilon_greedy_policy(state, épsilon) next_state, recompensa,
    hecho, truncado, info = env.step(action) replay_buffer.append((estado, acción,
    recompensa, next_state, hecho, truncado )) devolver next_state, recompensa, hecho, truncado, información

```

Finalmente, creemos una última función que muestreará un lote de experiencias del búfer de reproducción y entrenará el DQN realizando un único paso de descenso de gradiente en este lote:

```

tamaño_lote = 32
factor_descuento = 0.95
optimizador = tf.keras.optimizers.Nadam(learning_rate=1e-2) loss_fn =
tf.keras.losses.mean_squared_error

def paso_de_entrenamiento(tamaño_de_lote):
    experiencias = experiencias_de_muestra(tamaño_de_lote)
    estados, acciones, recompensas, estados_siguientes, hechos, truncados = experiencias
    valores_Q_siguientes = model.predict(estados_siguientes, detallado=0)
    valores_Q_siguientes máximos = valores_Q_siguientes.max(eje=1)

```

```

ejecuciones = 1.0 - (hechos | truncados) # episodio no finalizado o truncado target_Q_values = recompensas
+ ejecuciones * factor_descuento * max_next_Q_values target_Q_values = target_Q_values.reshape(-1, 1)
máscara = tf.one_hot(actions, n_outputs) con tf.GradientTape () como
cinta: all_Q_values = modelo(estados)

Valores_Q = tf.reduce_sum(all_Q_values * máscara, eje=1, keepdims=True) pérdida =
tf.reduce_mean(loss_fn(valores_Q_objetivo, valores_Q))

graduados = tape.gradient(loss, model.trainable_variables)
optimizador.apply_gradients(zip(grads, model.trainable_variables))

```

Esto es lo que sucede en este código:

- Primero definimos algunos hiperparámetros y creamos el optimizador y la función de pérdida.
- Luego creamos la función Training\_step(). Comienza muestreando un lote de experiencias, luego usa el DQN para predecir el valor Q para cada acción posible en el siguiente estado de cada experiencia.  
Como asumimos que el agente jugará de manera óptima, solo mantenemos el valor Q máximo para cada estado siguiente. A continuación, utilizamos [la ecuación 18-7](#) para calcular el valor Q objetivo para el par estado-acción de cada experiencia.
- Queremos usar el DQN para calcular el valor Q para cada par de estado-acción experimentado, pero el DQN también generará los valores Q para las otras acciones posibles, no solo para la acción que realmente eligió el agente. Por lo tanto, debemos enmascarar todos los valores Q que no necesitamos. La función tf.one\_hot() permite convertir una serie de índices de acción en dicha máscara.  
Por ejemplo, si las tres primeras experiencias contienen las acciones 1, 1, 0, respectivamente, entonces la máscara comenzará con [[0, 1], [0, 1], [1, 0]...]. Luego podemos multiplicar la salida del DQN con esta máscara, y esto pondrá a cero todos los valores Q que no queremos. Luego sumamos sobre el eje 1 para eliminar todos los ceros, manteniendo solo los valores Q de los pares estado-acción experimentados. Esto nos da el tensor Q\_values, que contiene un valor Q predicho para cada experiencia del lote.
- A continuación, calculamos la pérdida: es el error cuadrático medio entre los valores Q objetivo y predicho para los pares estado-acción experimentados.
- Finalmente, realizamos un paso de descenso de gradiente para minimizar la pérdida con respecto a las variables entrenables del modelo.

Esta fue la parte más difícil. Ahora entrenar el modelo es sencillo:

```

para episodio dentro del rango (600):
    obs, info = env.reset() para el paso
    en el rango(200): epsilon = max(
        - episodio / 500, 0.01) obs, recompensa, hecho, truncado,
    info = play_one_step(env, obs, epsilon) si está hecho o truncado:
        romper
    si episodio > 50:
        paso_entrenamiento(tamaño_lote)

```

Ejecutamos 600 episodios, cada uno con un máximo de 200 pasos. En cada paso, primero calculamos el valor épsilon para la política  $\epsilon$ -codiciosa: irá de 1 a 0,01, linealmente, en poco menos de 500 episodios. Luego llamamos a la función play\_one\_step(), que utilizará la política  $\epsilon$ -greedy para elegir una acción, luego ejecutarla y registrar la experiencia en el búfer de reproducción. Si el episodio finaliza o se trunca, salimos del bucle. Finalmente, si ya pasamos el episodio 50, llamamos a la función Training\_step() para entrenar el modelo en un lote muestreado del búfer de reproducción. La razón por la que reproducimos muchos episodios sin entrenamiento es para dar

el búfer de reproducción necesita algo de tiempo para llenarse (si no esperamos lo suficiente, entonces no habrá suficiente diversidad en el búfer de reproducción). Y eso es todo: ¡acabamos de implementar el algoritmo Deep Q-learning!

La Figura 18-10 muestra las recompensas totales que obtuvo el agente durante cada episodio.

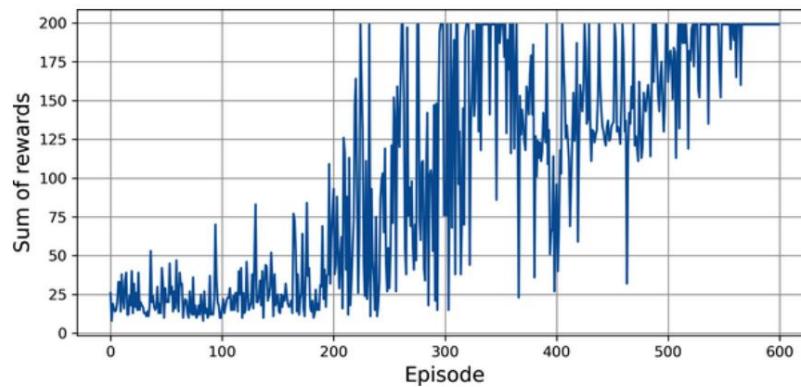


Figura 18-10. Curva de aprendizaje del algoritmo Q-learning profundo

Como puede ver, el algoritmo tardó un poco en empezar a aprender algo, en parte porque  $\epsilon$  era muy alto al principio. Luego, su progreso fue errático: primero alcanzó la recompensa máxima alrededor del episodio 220, pero inmediatamente cayó, luego rebotó hacia arriba y hacia abajo unas cuantas veces, y poco después parecía que finalmente se había estabilizado cerca de la recompensa máxima, alrededor del episodio 320. Su puntuación volvió a caer drásticamente. A esto se le llama olvido catastrófico y es uno de los grandes problemas que enfrentan prácticamente todos los algoritmos de RL: a medida que el agente explora el entorno, actualiza su política, pero lo que aprende en una parte del entorno puede alterar lo que aprendió anteriormente en otras partes del medio ambiente. Las experiencias están bastante correlacionadas y el entorno de aprendizaje sigue cambiando; ¡esto no es ideal para el descenso de gradientes! Si aumenta el tamaño del búfer de reproducción, el algoritmo estará menos sujeto a este problema. Ajustar la tasa de aprendizaje también puede ayudar. Pero la verdad es que el aprendizaje por refuerzo es difícil: el entrenamiento suele ser inestable y es posible que tengas que probar muchos valores de hiperparámetros y semillas aleatorias antes de encontrar una combinación que funcione bien. Por ejemplo, si intenta cambiar la función de activación de "elu" a "relu", el rendimiento será mucho menor.

#### NOTA

El aprendizaje por refuerzo es notoriamente difícil, en gran parte debido a las inestabilidades del entrenamiento y la enorme sensibilidad a la elección de valores de hiperparámetros y semillas aleatorias. Como lo expresó el investigador Andrej Karpathy, “[el aprendizaje supervisado] quiere funcionar. [...] Hay que obligar a RL a trabajar”. Necesitarás tiempo, paciencia, perseverancia y quizás también un poco de suerte. Ésta es una de las principales razones por las que RL no se adopta tan ampliamente como el aprendizaje profundo normal (por ejemplo, redes convolucionales). Pero hay algunas aplicaciones del mundo real, más allá de los juegos AlphaGo y Atari: por ejemplo, Google usa RL para optimizar los costos de su centro de datos y se usa en algunas aplicaciones de robótica, para ajuste de hiperparámetros y en sistemas de recomendación.

Quizás se pregunte por qué no planeamos la pérdida. Resulta que la pérdida es un mal indicador del desempeño del modelo. La pérdida podría disminuir, pero el agente podría tener un peor desempeño (por ejemplo, esto puede suceder cuando el agente se queda atascado en una pequeña región del entorno y el DQN comienza a sobreajustar esta región).

Por el contrario, la pérdida podría aumentar, pero el agente podría desempeñarse mejor (por ejemplo, si el DQN estaba subestimando los valores Q y comienza a aumentar correctamente sus predicciones, el agente probablemente se desempeñará mejor y obtendrá más recompensas, pero la pérdida podría aumentar). porque el DQN también fija los objetivos, que también serán mayores). Por tanto, es preferible trazar las recompensas.

El algoritmo básico de Q-learning profundo que hemos estado usando hasta ahora sería demasiado inestable para aprender a jugar juegos de Atari. Entonces, ¿cómo lo hizo DeepMind? Bueno, ¡modificaron el algoritmo!

## Variantes de Q-Learning profundo

Veamos algunas variantes del algoritmo de Q-learning profundo que pueden estabilizar y acelerar el entrenamiento.

### Objetivos de valor Q fijos En el

algoritmo básico de Q-learning profundo, el modelo se utiliza tanto para hacer predicciones como para establecer sus propios objetivos. Esto puede llevar a una situación análoga a la de un perro persiguiéndose su propia cola. Este circuito de retroalimentación puede hacer que la red sea inestable: puede divergir, oscilar, congelarse, etc. Para resolver este problema, en su artículo de 2013, los investigadores de DeepMind utilizaron dos DQN en lugar de uno: el primero es el modelo en línea, que aprende en cada paso y se usa para mover al agente, y el otro es el modelo objetivo que se usa solo para definir los objetivos. El modelo de destino es sólo un clon del modelo en línea:

```
target = tf.keras.models.clone_model(model) # clonar la arquitectura del modelo target.set_weights(model.get_weights())
# copiar los pesos
```

Luego, en la función `Training_step()`, solo necesitamos cambiar una línea para usar el modelo objetivo en lugar del modelo en línea al calcular los valores Q de los siguientes estados:

```
next_Q_values = target.predict(next_states, detallado=0)
```

Finalmente, en el ciclo de entrenamiento, debemos copiar los pesos del modelo en línea al modelo de destino, a intervalos regulares (por ejemplo, cada 50 episodios):

```
si episodio % 50 == 0:
    objetivo.set_weights(modelo.get_weights())
```

Dado que el modelo objetivo se actualiza con mucha menos frecuencia que el modelo en línea, los objetivos de valor Q son más estables, el ciclo de retroalimentación que analizamos anteriormente se amortigua y sus efectos son menos severos. Este enfoque fue una de las principales contribuciones de los investigadores de DeepMind en su artículo de 2013, permitiendo a los agentes aprender a jugar juegos de Atari a partir de píxeles sin procesar. Para estabilizar el entrenamiento, utilizaron una pequeña tasa de aprendizaje de 0,00025, actualizaron el modelo objetivo solo cada 10.000 pasos (en lugar de 50) y utilizaron un búfer de reproducción muy grande de 1 millón de experiencias. Disminuyeron épsilon muy lentamente, de 1 a 0,1 en 1 millón de pasos, y dejaron que el algoritmo se ejecutara durante 50 millones de pasos. Además, su DQN era una red convolucional profunda.

Ahora echemos un vistazo a otra variante de DQN que logró superar el estado del arte una vez más.

## Doble DQN

En un [artículo de 2014](#), Los investigadores de DeepMind modificaron su algoritmo DQN, aumentando su rendimiento y estabilizando en cierta medida el entrenamiento. A esta variante la llamaron doble DQN. La actualización se basó en la observación de que la red objetivo es propensa a sobreestimar los valores Q. De hecho, supongamos que todas las acciones son igualmente buenas: los valores Q estimados por el modelo objetivo deberían ser idénticos, pero como son aproximaciones, algunos pueden ser ligeramente mayores que otros, por pura casualidad. El modelo objetivo siempre seleccionará el valor Q más grande, que será ligeramente mayor que el valor Q medio, muy probablemente sobreestimando el valor Q verdadero (un poco como contar la altura de la ola aleatoria más alta al medir la profundidad de una piscina). Para solucionar este problema, los investigadores propusieron utilizar el modelo en línea en lugar del modelo objetivo al seleccionar las mejores acciones para los siguientes estados, y utilizar el modelo objetivo sólo para estimar los valores Q de estas mejores acciones. Aquí está la función `Training_step()` actualizada:

```
def paso_entrenamiento(tamaño_lote):
    experiencias = experiencias_muestra(tamaño_lote) estados,
    acciones, recompensas, próximos_estados, hechos, truncados = experiencias
```

```

next_Q_values = model.predict(next_states, detallado=0) # ≠ target.predict() best_next_actions =
next_Q_values.argmax(axis=1) next_mask = tf.one_hot(best_next_actions,
n_outputs).numpy() max_next_Q_values = (target.predict( next_states, detallado = 0) * next_mask

).sum(axis=1) [...] #
el resto es igual que antes

```

Pocos meses después, se propuso otra mejora al algoritmo DQN; veremos eso a continuación.

### Reproducción de experiencias priorizadas En

lugar de muestrear experiencias de manera uniforme desde el búfer de reproducción, ¿por qué no muestrear experiencias importantes con más frecuencia? Esta idea se llama muestreo de importancia (IS) o repetición de experiencia priorizada (PER), y se introdujo en un [artículo de 2015](#).<sup>15</sup> por investigadores de DeepMind (¡una vez más!).

Más específicamente, las experiencias se consideran “importantes” si es probable que conduzcan a un rápido progreso en el aprendizaje. Pero ¿cómo podemos estimar esto? Un enfoque razonable es medir la magnitud del error TD  $\delta = r + \gamma \cdot V(s') - V(s)$ . Un error TD grande indica que una transición  $(s, a, s')$  es muy sorprendente y, por lo tanto, probablemente vale la pena aprender de ella. Cuando se registra una experiencia en el búfer de reproducción, su prioridad se establece en un valor muy grande, para garantizar que se muestree al menos una vez. Sin embargo, una vez que se muestrea (y cada vez que se muestrea), se calcula el error TD  $\delta$  y la prioridad de esta experiencia se establece en  $p = |\delta|$  (más una pequeña constante para garantizar que cada experiencia tenga una probabilidad distinta de cero de ser muestreada). La probabilidad  $\zeta P$  de muestrear una experiencia con prioridad  $p$  es proporcional a  $p^\zeta$ , donde  $\zeta$  es un hiperparámetro que controla qué tan codicioso queremos que sea el muestreo de importancia: cuando  $\zeta = 0$ , simplemente obtenemos un muestreo uniforme, y cuando  $\zeta = 1$ , obtenga un muestreo de importancia completa. En el artículo, los autores utilizaron  $\zeta = 0,6$ , pero el valor óptimo dependerá de la tarea.

Sin embargo, hay un inconveniente: dado que las muestras estarán sesgadas hacia experiencias importantes, debemos compensar este sesgo durante el entrenamiento restando importancia a las experiencias según su importancia, o de lo contrario el modelo simplemente sobreajustará las experiencias importantes. Para ser claros, queremos que se muestreen experiencias importantes con más frecuencia, pero esto también significa que debemos darles un peso menor durante el entrenamiento  $-\beta$ . Para hacer esto, definimos el peso de entrenamiento de cada experiencia como  $w = (n P)$ , donde  $n$  es el número de experiencias en el búfer de repetición y  $\beta$  es un hiperparámetro que controla cuánto queremos compensar el sesgo de muestreo de importancia (0 significa nada, mientras que 1 significa completamente). En el artículo, los autores utilizaron  $\beta = 0,4$  al comienzo del entrenamiento y lo aumentaron linealmente a  $\beta = 1$  al final del entrenamiento. Nuevamente, el valor óptimo dependerá de la tarea, pero si aumenta uno, generalmente querrá aumentar también el otro.

Ahora veamos una última variante importante del algoritmo DQN.

## Duelo DQN

El algoritmo de duelo DQN (DDQN, que no debe confundirse con el doble DQN, aunque ambas técnicas se pueden combinar fácilmente) se introdujo en otro [artículo de 2015](#).<sup>17</sup> por investigadores de DeepMind. Para entender cómo funciona, primero debemos tener en cuenta que el valor Q de un par estado-acción  $(s, a)$  se puede expresar como  $Q(s, a) = V(s) + A(s, a)$ , donde  $V(s)$  es el valor del estado  $s$  y  $A(s, a)$  es la ventaja de realizar la acción  $a$  en el estado  $s$ , en comparación con todas las demás acciones posibles en ese estado. Además, el valor de un estado es igual al valor Q de la mejor acción a para ese estado (ya que asumimos que la política óptima elegirá la mejor acción), por lo que  $V(s) = Q(s, a_*)$ , que implica que  $A(s, a_*) = 0$ . En un DQN de duelo, el modelo estima tanto el valor del estado como la ventaja de cada acción posible. Dado que la mejor acción debería tener una ventaja de 0, el modelo resta la ventaja máxima prevista de todas las ventajas previstas. Aquí hay un modelo DDQN simple, implementado usando la API funcional:

```

estados_entrada = tf.keras.layers.Input(forma=[4]) oculto1 =
tf.keras.layers.Dense(32, activación="elu")(estados_entrada)

```

```

oculto2 = tf.keras.layers.Dense(32, activación="elu")(hidden1) state_values =
tf.keras.layers.Dense(1)(hidden2) raw_advantages =
tf.keras.layers.Dense(n_outputs)(hidden2 ) ventajas = raw_advantages -
tf.reduce_max(raw_advantages, eje=1, keepdims=True)

Valores_Q = valores_estado + modelo de ventajas
= tf.keras.Model(entradas=[estados_entrada], salidas=[valores_Q])

```

El resto del algoritmo es el mismo que antes. De hecho, ¡puedes crear un DQN de duelo doble y combinarlo con una repetición de experiencia priorizada! De manera más general, se pueden combinar muchas técnicas de RL, como demostró DeepMind en un [artículo de 2017](#): Los autores del artículo combinaron seis técnicas diferentes en un agente llamado Rainbow, que superó ampliamente el estado del arte.

Como puede ver, el aprendizaje por refuerzo profundo es un campo en rápido crecimiento y ¡hay mucho más por descubrir!

## Descripción general de algunos algoritmos RL populares

Antes de cerrar este capítulo, echemos un breve vistazo a algunos otros algoritmos populares:

AlfaGo [19](#)

AlphaGo utiliza una variante de la búsqueda de árboles de Monte Carlo (MCTS) basada en redes neuronales profundas para vencer a los campeones humanos en el juego de Go. MCTS fue inventado en 1949 por Nicholas Metropolis y Stanislaw Ulam. Selecciona el mejor movimiento después de ejecutar muchas simulaciones, explorar repetidamente el árbol de búsqueda a partir de la posición actual y dedicar más tiempo a las ramas más prometedoras.

Cuando llega a un nodo que no ha visitado antes, juega aleatoriamente hasta que finaliza el juego y actualiza sus estimaciones para cada nodo visitado (excluyendo los movimientos aleatorios), aumentando o disminuyendo cada estimación según el resultado final. AlphaGo se basa en el mismo principio, pero utiliza una red de políticas para seleccionar movimientos, en lugar de jugar al azar. Esta red de políticas se entrena utilizando gradientes de políticas. El algoritmo original involucraba tres redes neuronales más y era más complicado, pero se simplificó en el [artículo de AlphaGo Zero](#), que utiliza una única red neuronal para seleccionar movimientos y evaluar los estados del juego. El [artículo AlphaZero](#) generalizó este algoritmo, haciéndolo capaz de abordar no sólo el <sup>20</sup> juego de Go, sino también el ajedrez y el shogi (ajedrez japonés).

Por último, el [artículo de MuZero](#) <sup>22</sup> continuó mejorando este algoritmo, superando las iteraciones anteriores a pesar de que el agente comienza sin siquiera conocer las reglas del juego!

## Algoritmos actor-crítico

Los actores críticos son una familia de algoritmos RL que combinan gradientes de políticas con redes Q profundas. Un agente actor-crítico contiene dos redes neuronales: una red de políticas y un DQN. El DQN se entrena normalmente, aprendiendo de las experiencias del agente. La red de políticas aprende de manera diferente (y mucho más rápida) que en el PG normal: en lugar de estimar el valor de cada acción pasando por múltiples episodios, luego sumar las recompensas descontadas futuras para cada acción y finalmente normalizarlas, el agente (actor) confía en sobre los valores de acción estimados por el DQN (crítico). Es un poco como un atleta (el agente) que aprende con la ayuda de un entrenador (el DQN).

Ventaja asincrónica actor-crítico (A3C)

[23](#)

Esta es una importante variante actor-crítico introducida por investigadores de DeepMind en 2016, donde múltiples agentes aprenden en paralelo, explorando diferentes copias del entorno. A intervalos regulares, pero de forma asincrónica (de ahí el nombre), cada agente envía algunas actualizaciones de peso a una red maestra y luego extrae los pesos más recientes de esa red. Cada agente contribuye así a mejorar la red maestra y se beneficia de lo que los demás agentes han aprendido. Además, en lugar de estimar los valores Q, el DQN estima la ventaja de cada acción (de ahí la segunda A en el nombre), lo que estabiliza el entrenamiento.

### Ventaja actor-crítico (A2C)

A2C es una variante del algoritmo A3C que elimina la asincronicidad. Todas las actualizaciones del modelo son sincrónicas, por lo que las actualizaciones de gradiente se realizan en lotes más grandes, lo que permite que el modelo utilice mejor la potencia de la GPU.

### Actor-crítico suave (SAC) <sup>24</sup>

SAC es una variante actor-crítica propuesta en 2018 por Tuomas Haarnoja y otros investigadores de UC Berkeley. Aprende no sólo a recompensar, sino también a maximizar la entropía de sus acciones. En otras palabras, intenta ser lo más impredecible posible y al mismo tiempo obtener tantas recompensas como sea posible. Esto anima al agente a explorar el entorno, lo que acelera el entrenamiento y hace que sea menos probable que ejecute repetidamente la misma acción cuando el DQN produce estimaciones imperfectas. Este algoritmo ha demostrado una eficiencia de muestra asombrosa (a diferencia de todos los algoritmos anteriores, que aprenden muy lentamente).

### Optimización de políticas próximas (PPO) <sup>25</sup>

Este algoritmo de John Schulman y otros investigadores de OpenAI se basa en A2C, pero recorta la función de pérdida para evitar actualizaciones de peso excesivamente grandes (que a menudo conducen a inestabilidades en el entrenamiento).

PPO es una simplificación de la [optimización de la política de región de confianza](#) anterior (<sup>26</sup>TRPO), también de OpenAI. OpenAI fue noticia en abril de 2019 con su IA llamada OpenAI Five, basada en el algoritmo PPO, que derrotó a los campeones del mundo en el juego multijugador Dota 2.

### Exploración basada en la curiosidad<sup>27</sup>

Un problema recurrente en RL es la escasez de recompensas, lo que hace que el aprendizaje sea muy lento e inefficiente. Deepak Pathak y otros investigadores de UC Berkeley han propuesto una manera interesante de abordar este problema: ¿por qué no ignorar las recompensas y simplemente hacer que el agente tenga mucha curiosidad por explorar el entorno? De este modo, las recompensas se vuelven intrínsecas al agente, en lugar de provenir del entorno. De manera similar, es más probable que estimular la curiosidad en un niño dé buenos resultados que simplemente recompensarlo por obtener buenas calificaciones. ¿Cómo funciona esto? El agente intenta continuamente predecir el resultado de sus acciones y busca situaciones en las que el resultado no coincide con sus predicciones. En otras palabras, quiere sorprenderse. Si el resultado es predecible (aburrido), se va a otra parte. Sin embargo, si el resultado es impredecible pero el agente nota que no tiene control sobre él, también se aburre al cabo de un tiempo. Sólo por curiosidad, los autores lograron entrenar a un agente en muchos videojuegos: aunque el agente no recibe ninguna penalización por perder, el juego comienza de nuevo, lo cual es aburrido y aprende a evitarlo.

### Aprendizaje abierto (OEL)

El objetivo de OEL es formar agentes capaces de aprender infinitamente tareas nuevas e interesantes, normalmente generadas de forma procedimental. Aún no hemos llegado a ese punto, pero ha habido algunos avances sorprendentes en los últimos años. Por ejemplo, un [artículo de 2019](#) Un equipo de investigadores de Uber AI presentó el algoritmo POET, que genera múltiples entornos 2D simulados con baches y agujeros y entrena a un agente por entorno: el objetivo del agente es caminar lo más rápido posible evitando los obstáculos. El algoritmo comienza con entornos simples, pero gradualmente se vuelven más difíciles con el tiempo: esto se llama aprendizaje curricular. Además, aunque cada agente sólo se entrena en un entorno, debe competir periódicamente contra otros agentes en todos los entornos. En cada entorno, el ganador se copia y reemplaza al agente que estaba allí antes. De esta manera, el conocimiento se transfiere periódicamente entre entornos y se seleccionan los agentes más adaptables. Al final, los agentes caminan mucho mejor que los agentes entrenados en una sola tarea y pueden afrontar entornos mucho más difíciles. Por supuesto, este principio también se puede aplicar a otros entornos y tareas. Si está interesado en OEL, asegúrese de consultar el [artículo Enhanced POET](#), así como el [artículo 2021](#) de DeepMind. sobre este tema.

## CONSEJO

Si desea obtener más información sobre el aprendizaje por refuerzo, consulte el libro [Aprendizaje por refuerzo](#). por Phil Winder (O'Reilly).

Cubrimos muchos temas en este capítulo: gradientes de políticas, cadenas de Markov, procesos de decisión de Markov, Q-learning, Q-learning aproximado y Q-learning profundo y sus principales variantes (objetivos de valor Q fijo, DQN doble, DQN en duelo y repetición de experiencia priorizada), y finalmente echamos un vistazo rápido a algunos otros algoritmos populares. El aprendizaje por refuerzo es un campo enorme y apasionante, en el que cada día surgen nuevas ideas y algoritmos, así que espero que este capítulo haya despertado tu curiosidad: ¡hay todo un mundo por explorar!

## Ejercicios

1. ¿Cómo definirías el aprendizaje por refuerzo? ¿En qué se diferencia del programa regular supervisado o ¿aprendizaje sin supervisión?
2. ¿Puedes pensar en tres posibles aplicaciones de RL que no se mencionaron en este capítulo? Para cada uno de ellos, ¿cuál es el medio ambiente? ¿Cuál es el agente? ¿Cuáles son algunas acciones posibles? ¿Cuáles son las recompensas?
3. ¿Cuál es el factor de descuento? ¿Puede cambiar la política óptima si modifica el factor de descuento?
4. ¿Cómo se mide el desempeño de un agente de aprendizaje por refuerzo?
5. ¿Cuál es el problema de asignación de crédito? ¿Cuándo ocurre? ¿Cómo puedes aliviarlo?
6. ¿Cuál es el punto de utilizar un búfer de reproducción?
7. ¿Qué es un algoritmo RL fuera de política?
8. Utilice gradientes de políticas para resolver el entorno LunarLander-v2 de OpenAI Gym.
9. Utilice un DQN de doble duelo para entrenar a un agente que pueda alcanzar un nivel sobrehumano en el famoso Juego Atari Breakout ("ALE/Breakout-v5"). Las observaciones son imágenes. Para simplificar la tarea, debes convertirlos a escala de grises (es decir, promedio sobre el eje del canal), luego recortarlos y reducir la resolución, para que sean lo suficientemente grandes para reproducirlos, pero no más. Una imagen individual no te dice en qué dirección van la pelota y las paletas, por lo que debes fusionar dos o tres imágenes consecutivas para formar cada estado. Por último, el DQN debería estar compuesto principalmente por capas convolucionales.
10. Si te sobran unos 100 dólares, puedes comprar una Raspberry Pi 3 y algo de robótica barata componentes, instala TensorFlow en el Pi y ¡vívelte loco! Para ver un ejemplo, mira esta [divertida publicación](#). de Lukas Biewald, o eche un vistazo a GoPiGo o BrickPi. Comience con objetivos simples, como hacer que el robot gire para encontrar el ángulo más brillante (si tiene un sensor de luz) o el objeto más cercano (si tiene un sensor de sonar), y muévase en esa dirección. Entonces puedes empezar a utilizar el aprendizaje profundo: por ejemplo, si el robot tiene una cámara, puedes intentar implementar un algoritmo de detección de objetos para que detecte personas y se mueva hacia ellas. También puedes intentar usar RL para que el agente aprenda por sí solo cómo usar los motores para lograr ese objetivo. ¡Divirtirse!

Las soluciones a estos ejercicios están disponibles al final del cuaderno de este capítulo, en <https://homl.info/colab3>.

- 
- <sup>1</sup> Para obtener más detalles, asegúrese de consultar el libro de Richard Sutton y Andrew Barto sobre RL, Reinforcement Learning: An Introducción (Presa MIT).
- <sup>2</sup> Volodymyr Mnih et al., "Playing Atari with Deep Reinforcement Learning", preimpresión de arXiv arXiv:1312.5602 (2013).
- <sup>3</sup> Volodymyr Mnih et al., "Control a nivel humano mediante el aprendizaje por refuerzo profundo", Nature 518 (2015): 529–533.
- <sup>4</sup> Vea los videos del sistema de DeepMind aprendiendo a jugar Space Invaders, Breakout y otros videojuegos en <https://hml.info/dqn3>.
- <sup>5</sup> Las imágenes (a), (d) y (e) son de dominio público. La imagen (b) es una captura de pantalla del juego Ms. Pac-Man , copyright Atari (uso legítimo en este capítulo). La imagen (c) es una reproducción de Wikipedia; Fue creado por el usuario Stevertigo y publicado bajo Creative Commons BY-SA 2.0.
- <sup>6</sup> A menudo es mejor dar a los que tienen un desempeño deficiente una pequeña oportunidad de sobrevivir, para preservar cierta diversidad en el "acervo genético".
- <sup>7</sup> Si hay un solo progenitor, esto se llama reproducción asexual. Con dos (o más) padres, se llama sexual. reproducción. El genoma de una descendencia (en este caso, un conjunto de parámetros políticos) se compone aleatoriamente de partes de los genomas de sus padres.
- <sup>8</sup> Un ejemplo interesante de un algoritmo genético utilizado para el aprendizaje por refuerzo es el algoritmo NeuroEvolution of Augmenting Topologies (NEAT).
- <sup>9</sup> Esto se llama ascenso en gradiente. Es como el descenso de gradiente, pero en la dirección opuesta: maximizar en lugar de minimizando.
- <sup>10</sup> OpenAI es una empresa de investigación de inteligencia artificial, financiada en parte por Elon Musk. Su objetivo declarado es promover y Desarrollar IA amigables que beneficien a la humanidad (en lugar de exterminarla).
- <sup>11</sup> Ronald J. Williams, "Algoritmos estadísticos simples de seguimiento de gradientes para la inclinación por el refuerzo conexiónista", Aprendizaje automático 8 (1992): 229–256.
- <sup>12</sup> Richard Bellman, "Un proceso de decisión markoviano", Revista de Matemáticas y Mecánica 6, no. 5 (1957): 679–684.
- <sup>13</sup> Una gran [publicación de 2018](#) de Alex Irpan expone muy bien las mayores dificultades y limitaciones de RL.
- <sup>14</sup> Hado van Hasselt et al., "Aprendizaje por refuerzo profundo con doble Q-Learning", Actas de la 30<sup>a</sup> Conferencia AAAI sobre Inteligencia Artificial (2015): 2094–2100.
- <sup>15</sup> Tom Schaul et al., "Prioritized Experience Replay", preimpresión de arXiv arXiv:1511.05952 (2015).
- <sup>16</sup> También podría ser simplemente que las recompensas sean ruidosas, en cuyo caso existen mejores métodos para estimar el impacto de una experiencia. importancia (consulte el documento para ver algunos ejemplos).
- <sup>17</sup> Ziyu Wang et al., "Arquitecturas de red en duelo para el aprendizaje por refuerzo profundo", preimpresión de arXiv arXiv:1511.06581 (2015).
- <sup>18</sup> Matteo Hessel et al., "Rainbow: Combinando mejoras en el aprendizaje por refuerzo profundo", preimpresión de arXiv arXiv:1710.02298 (2017): 3215–3222.
- <sup>19</sup> David Silver et al., "Dominar el juego del Go con redes neuronales profundas y búsqueda de áboles", Nature 529 (2016): 484–489.
- <sup>20</sup> David Silver et al., "Dominar el juego de ir sin conocimiento humano", Nature 550 (2017): 354–359.
- <sup>21</sup> David Silver et al., "Dominar el ajedrez y el shogi mediante el juego autónomo con un algoritmo de aprendizaje por refuerzo general", preimpresión de arXiv arXiv:1712.01815.
- <sup>22</sup> Julian Schrittwieser et al., "Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model", preimpresión de arXiv arXiv:1911.08265 (2019).
- <sup>23</sup> Volodymyr Mnih et al., "Métodos asincrónicos para el aprendizaje por refuerzo profundo", Actas del 33<sup>o</sup> Conferencia internacional sobre aprendizaje automático (2016): 1928–1937.
- <sup>24</sup> Tuomas Haarnoja et al., "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor", Actas de la 35<sup>a</sup> Conferencia Internacional sobre Aprendizaje Automático (2018): 1856–1865.
- <sup>25</sup> John Schulman et al., "Algoritmos de optimización de políticas próximas", preimpresión de arXiv arXiv:1707.06347 (2017).
- <sup>26</sup> John Schulman et al., "Trust Region Policy Optimization", Actas de la 32<sup>a</sup> Conferencia Internacional sobre Aprendizaje automático (2015): 1889–1897.
- <sup>27</sup> Deepak Pathak et al., "Exploración impulsada por la curiosidad mediante predicción autosupervisada", Actas del 34<sup>o</sup> Conferencia internacional sobre aprendizaje automático (2017): 2778–2787.
- <sup>28</sup> Rui Wang et al., "Paired Open-Ended Trailblazer (POET): Endless Generating Increasingly Complex and Diverse Learning Environments and Their Solutions", preimpresión de arXiv arXiv:1901.01753 (2019).

**29** Rui Wang et al., "POET mejorado: aprendizaje por refuerzo abierto a través de una invención ilimitada del aprendizaje Desafíos y sus soluciones", preimpresión de arXiv arXiv:2003.08536 (2020).

**30** Open-Ended Learning Team et al., "El aprendizaje abierto conduce a agentes generalmente capaces", preimpresión de arXiv arXiv:2107.12808 (2021).

# Capítulo 19. Entrenamiento e implementación de modelos de TensorFlow a escala

---

Una vez que tienes un modelo hermoso que hace predicciones sorprendentes, ¿qué haces con él? Bueno, ¡necesitas ponerlo en producción! Esto podría ser tan simple como ejecutar el modelo en un lote de datos y quizás escribir un script que ejecute este modelo todas las noches. Sin embargo, suele ser mucho más complicado. Es posible que varias partes de su infraestructura necesiten usar este modelo en datos en vivo, en cuyo caso probablemente querrá envolver su modelo en un servicio web: de esta manera, cualquier parte de su infraestructura puede consultar el modelo en cualquier momento usando un REST simple. API (o algún otro protocolo), como analizamos en [el Capítulo 2](#). Pero a medida que pasa el tiempo, necesitará volver a entrenar su modelo periódicamente con datos nuevos y enviar la versión actualizada a producción. Debe manejar el control de versiones del modelo, realizar una transición elegante de un modelo al siguiente, posiblemente retroceder al modelo anterior en caso de problemas y tal vez ejecutar varios modelos diferentes en paralelo para realizar experimentos A/ B. Si su producto tiene éxito, su servicio puede comenzar a recibir una gran cantidad de consultas por segundo (QPS) y debe ampliarse para soportar la carga. Una <sup>1</sup> gran solución para ampliar su servicio, como verá en este capítulo, es utilizar TF Serving, ya sea en su propia infraestructura de hardware o mediante un servicio en la nube como Google Vertex AI. Se encargará de servir eficientemente a su modelo, manejar transiciones elegantes del modelo y más. Si utiliza la plataforma en la nube, también obtendrá muchas funciones adicionales, como potentes herramientas de monitoreo.

<sup>2</sup>

Además, si tiene muchos datos de entrenamiento y modelos con uso intensivo de computación, el tiempo de entrenamiento puede ser prohibitivamente largo. Si su producto necesita adaptarse a los cambios rápidamente, entonces un largo tiempo de capacitación puede ser una

espectacular (por ejemplo, piense en un sistema de recomendación de noticias que promocione noticias de la semana pasada). Quizás lo más importante es que un largo tiempo de formación le impedirá experimentar con nuevas ideas. En el aprendizaje automático (como en muchos otros campos), es difícil saber de antemano qué ideas funcionarán, por lo que debes probar tantas como sea posible, lo más rápido posible. Una forma de acelerar el entrenamiento es utilizar aceleradores de hardware como GPU o TPU. Para ir aún más rápido, puedes entrenar un modelo en varias máquinas, cada una equipada con múltiples aceleradores de hardware. La API de estrategias de distribución simple pero poderosa de TensorFlow hace que esto sea fácil, com  
ver.

En este capítulo veremos cómo implementar modelos, primero usando TF Serving y luego usando Vertex AI. También echaremos un vistazo rápido a la implementación de modelos en aplicaciones móviles, dispositivos integrados y aplicaciones web. Luego, discutiremos cómo acelerar los cálculos usando GPU y cómo entrenar modelos en múltiples dispositivos y servidores usando la API de estrategias de distribución. Por último, exploraremos cómo entrenar modelos y ajustar sus hiperparámetros a escala utilizando Vertex AI. Son muchos temas para discutir, ¡así que profundicemos!

## Sirviendo un modelo TensorFlow

Una vez que haya entrenado un modelo de TensorFlow, podrá usarlo fácilmente en cualquier código Python: si es un modelo de Keras, ¡simplemente llame a su método predict()! Pero a medida que su infraestructura crece, llega un punto en el que es preferible envolver su modelo en un pequeño servicio cuya única función es hacer predicciones y hacer que el resto de la infraestructura lo consulte (por ejemplo, a través de una<sup>3</sup>API REST o gRPC). Esto desacopla su modelo del resto de la infraestructura, lo que permite cambiar fácilmente las versiones del modelo o ampliar el servicio según sea necesario (independientemente del resto de su infraestructura), realizar experimentos A/B y garantizar que todos sus componentes de software dependan en las mismas versiones del modelo. También simplifica las pruebas y el desarrollo, y más. Tú

Podrías crear tu propio microservicio usando cualquier tecnología que deseas (por ejemplo, usando la biblioteca Flask), pero ¿por qué reinventar la rueda cuando solo puedes usar TF Serving?

## Usando el servicio TensorFlow

TF Serving es un servidor modelo muy eficiente y probado en batalla, escrito en C++. Puede soportar una carga elevada, ofrecer múltiples versiones de sus modelos y observar un repositorio de modelos para implementar automáticamente las últimas versiones y más (consulte la [Figura 19-1](#)).

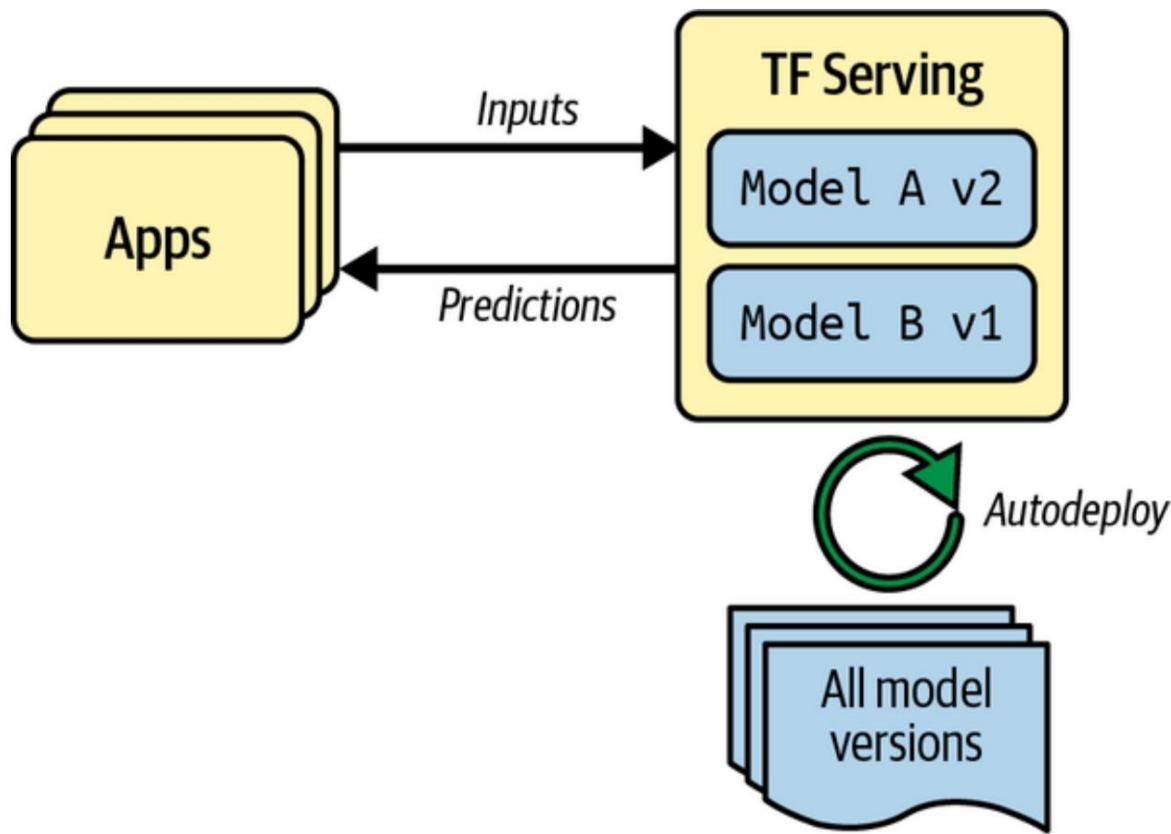


Figura 19-1. TF Serving puede servir a múltiples modelos e implementar automáticamente la última versión de cada modelo.

Entonces, supongamos que ha entrenado un modelo MNIST usando Keras y desea implementarlo en TF Serving. Lo primero que debe hacer es exportar este modelo al formato SavedModel, presentado en el [Capítulo 10](#).

## Exportación de modelos

guardados Ya sabes cómo guardar el modelo: simplemente llama a `model.save()`.

Ahora, para versionar el modelo, sólo necesita crear un subdirectorio para cada versión del modelo. ¡Fácil!

```
desde pathlib import Ruta importar
tensorflow como tf

X_train, X_valid, X_test = [...] # cargar y dividir el conjunto de datos MNIST

modelo = [...] # construir y entrenar un modelo MNIST (también maneja el preprocesamiento
de imágenes)

model_name = "my_mnist_model"
model_version = "0001"
model_path = Ruta(model_name) / model_version
model.save(model_path, save_format="tf")
```

Generalmente es una buena idea incluir todas las capas de preprocesamiento en el modelo final que exporta para que pueda ingerir datos en su forma natural una vez que se implementen en producción. Esto evita tener que encargarse del preprocesamiento por separado dentro de la aplicación que utiliza el modelo.

Agrupar los pasos de preprocesamiento dentro del modelo también simplifica su actualización más adelante y limita el riesgo de discrepancia entre un modelo y los pasos de preprocesamiento que requiere.

### ADVERTENCIA

Dado que `SavedModel` guarda el gráfico de cálculo, solo se puede usar con modelos que se basan exclusivamente en operaciones de TensorFlow, excluyendo la operación `tf.py_function()`, que encapsula código Python arbitrario.

TensorFlow viene con una pequeña interfaz de línea de comandos `save_model_cli` para inspeccionar `SavedModels`. Úselo para inspeccionar nuestros productos exportados.

modelo:

```
$ save_model_cli show --dir my_mnist_model/0001
```

El SavedModel proporcionado contiene los siguientes conjuntos de etiquetas:  
'servir'

¿Qué significa esta salida? Bueno, un SavedModel contiene uno o más metagrafos. Un metagrafo es un gráfico de cálculo más algunas definiciones de firma de función, incluidos sus nombres, tipos y formas de entrada y salida. Cada metagrafo se identifica mediante un conjunto de etiquetas. Por ejemplo, es posible que desee tener un metagrama que contenga el gráfico de cálculo completo, incluidas las operaciones de entrenamiento: normalmente etiquetaría este como "entrenamiento". Y es posible que tenga otro metagrafo que contenga un gráfico de cálculo recortado con solo las operaciones de predicción, incluidas algunas operaciones específicas de la GPU: este podría etiquetarse como "servir", "gpu". Es posible que también quieras tener otros metagramas. Esto se puede hacer utilizando la API SavedModel de bajo nivel de TensorFlow . Sin embargo, cuando guarda un modelo de Keras usando su método save(), guarda un único metagrafo etiquetado como "servir". Inspeccionemos este conjunto de etiquetas de "servicio":

```
$ save_model_cli show --dir 0001/my_mnist_model --tag_set
atender
```

El MetaGraphDef de SavedModel proporcionado contiene SignatureDefs con estas claves:

Clave SignatureDef: "\_\_saved\_model\_init\_op"

Clave SignatureDef: "serving\_default"

Este metagrama contiene dos definiciones de firma: una función de inicialización llamada "\_\_saved\_model\_init\_op", de la que no necesita preocuparse, y una función de servicio predeterminada llamada "serving\_default". Al guardar un modelo de Keras, la función de servicio predeterminada es el método call() del modelo, lo que hace

predicciones, como ya sabes. Conozcamos más detalles sobre esta función de publicación:

```
$ save_model_cli show --dir 0001/my_mnist_model --tag_set servir \
```

```
    --signature_def servicio_predeterminado
```

El SavedModel SignatureDef proporcionado contiene las siguientes entradas:

```
entradas['flatten_input'] tensor_info: dtype: DT_UINT8  
    forma: (-1, 28, 28)  
    nombre:  
        server_default_flatten_input:0
```

El SavedModel SignatureDef proporcionado contiene las siguientes salidas:

```
salidas['dense_1'] tensor_info: dtype: DT_FLOAT  
    forma: (-1, 10)  
    nombre:  
        StatefulPartitionedCall:0
```

El nombre del método es: tensorflow/serving/predict

Tenga en cuenta que la entrada de la función se denomina "flatten\_input" y la salida se denomina "dense\_1". Estos corresponden a los nombres de las capas de entrada y salida del modelo Keras. También puede ver el tipo y la forma de los datos de entrada y salida. ¡Se ve bien!

Ahora que tiene un modelo guardado, el siguiente paso es instalar TF Serving.

## Instalación e inicio de TensorFlow Serving

Hay muchas formas de instalar TF Serving: usando el administrador de paquetes del sistema, usando una imagen de Docker<sup>4</sup>, instalando desde la fuente y más. Dado que Colab se ejecuta en Ubuntu, podemos usar el administrador de paquetes apt de Ubuntu de esta manera:

```
url = "https://storage.googleapis.com/tensorflow-serving-apt" src = "servidor-  
    modelo-tensorflow estable tensorflow-servidor-modelo-universal"
```

```

!echo 'deb {url} {src}' > /etc/apt/
sources.list.d/tensorflow-serving.list
curl '{url}/tensorflow-
serving.release.pub.gpg' | apt-key add -
apt update -q && apt-get install -y tensorflow-model-
servidor
%pip install -q -U tensorflow-serving-api

```

Este código comienza agregando el repositorio de paquetes de TensorFlow a la lista de fuentes de paquetes de Ubuntu. Luego descarga la clave GPG pública de TensorFlow y la agrega a la lista de claves del administrador de paquetes para que pueda verificar las firmas de los paquetes de TensorFlow. A continuación, utiliza apt para instalar el paquete tensorflow-model-server. Por último, instala la biblioteca tensorflow-serving-api, que necesitaremos para comunicarnos con el servidor.

Ahora queremos iniciar el servidor. El comando requerirá la ruta absoluta del directorio del modelo base (es decir, la ruta a my\_mnist\_model, no 0001), así que guardémosla en la variable de entorno MODEL\_DIR:

```

importar sistema operativo

os.environ["MODEL_DIR"] = str(model_path.parent.absolute())

```

Luego podemos iniciar el servidor:

```

%%bash --bg
tensorflow_model_server \
    --port=8500
    --rest_api_port=8501
    --model_name=my_mnist_model
    --model_base_path="${MODEL_DIR}"
>my_server.log 2>&1

```

En Jupyter o Colab, el comando mágico %%bash --bg ejecuta la celda como un script bash, ejecutándolo en segundo plano. La parte >my\_server.log 2>&1 redirige la salida estándar y el error estándar al archivo my\_server.log . ¡Y eso es! El servicio TF es

ahora se ejecuta en segundo plano y sus registros se guardan en `my_server.log`. Cargó nuestro modelo MNIST (versión 1) y ahora está esperando solicitudes de gRPC y REST, respectivamente, en los puertos 8500 y 8501.

## EJECUTANDO TF SIRVIENDO EN UN CONTENEDOR DOCKER

Si está ejecutando el portátil en su propia máquina y ha instalado Docker, puede ejecutar docker pull tensorflow/serving en una terminal para descargar la imagen de TF Serving. El equipo de TensorFlow recomienda encarecidamente este método de instalación porque es simple, no afectará su sistema y ofrece un alto rendimiento. Para iniciar el servidor dentro de un contenedor Docker, puede ejecutar<sup>5</sup> el siguiente comando en una terminal:

```
$ docker run -it --rm -v "/ruta/a/  
mi_modelo_mnist:/models/mi_modelo_mnist" \ -p 8500:8500 -p  
8501:8501 -e  
MODEL_NAME=my_mnist_model tensorflow/servicio
```

Esto es lo que significan todas estas opciones de línea de comandos:

-íl

Hace que el contenedor sea interactivo (para que pueda presionar Ctrl-C para detenerlo) y muestra la salida del servidor.

--rm

Elimina el contenedor cuando lo detienes: no es necesario saturar tu máquina con contenedores interrumpidos. Sin embargo, no elimina la imagen.

-v

"/ruta/a/mi\_modelo\_mnist:/modelos/mi\_modelo\_mnist"

Hace que el directorio my\_mnist\_model del host esté disponible para el contenedor en la ruta /models/mnist\_model. Debe reemplazar /path/to/my\_mnist\_model con la ruta absoluta de este directorio. En Windows, recuerde usar \ en lugar de / en el

ruta del host, pero no en la ruta del contenedor (ya que el contenedor se ejecuta en Linux).

-p 8500:8500

Hace que el motor Docker reenvíe el puerto TCP 8500 del host al puerto TCP 8500 del contenedor. De forma predeterminada, TF Serving usa este puerto para servir la API gRPC.

-p 8501:8501

Reenvía el puerto TCP 8501 del host al puerto TCP 8501 del contenedor. La imagen de Docker está configurada para usar este puerto de forma predeterminada para servir la API REST.

-e MODEL\_NAME=mi\_modelo\_mnist

Establece la variable de entorno MODEL\_NAME del contenedor, para que TF Serving sepa qué modelo servir. De forma predeterminada, buscará modelos en el directorio /models y automáticamente ofrecerá la última versión que encuentre.

tensorflow/servicio

Este es el nombre de la imagen a ejecutar.

Ahora que el servidor está en funcionamiento, consultémoslo, primero usando la API REST y luego la API gRPC.

Consulta de TF Serving a través de la API REST

Comencemos creando la consulta. Debe contener el nombre de la firma de la función que desea llamar y, por supuesto, los datos de entrada.

Dado que la solicitud debe utilizar el formato JSON, tenemos que convertir las imágenes de entrada de una matriz NumPy a una lista de Python:

```

importar json

X_new = X_test[:3] # finge que tenemos 3 imágenes de dígitos nuevos para
clasificar
request_json = json.dumps({
    "signature_name": "serving_default",
    "instancias": X_new.tolist(),
})

```

Tenga en cuenta que el formato JSON está 100% basado en texto. La cadena de solicitud  
Se ve como esto:

```

>>> solicitud_json
'{"signature_name": "serving_default", "instancias": [[[0,
0, 0, 0, ... ]]]}'

```

Ahora enviamos esta solicitud a TF Serving a través de una POST HTTP  
pedido. Esto se puede hacer usando la biblioteca de solicitudes (no forma parte  
de la biblioteca estándar de Python, pero está preinstalada en Colab):

[solicitudes de importación](#)

```

URL_servidor =
"http://localhost:8501/v1/models/my_mnist_model:predict"
respuesta = solicitudes.post(server_url, datos=request_json)
respuesta.raise_for_status() # generar una excepción en caso
de error
respuesta = respuesta.json()

```

Si todo va bien, la respuesta debería ser un diccionario que contenga un  
tecla única de "predicciones". El valor correspondiente es la lista de  
predicciones Esta lista es una lista de Python, así que convirtámosla en NumPy.  
matriz y redondea los flotantes que contiene al segundo decimal:

```

>>> importar números como np
>>> y_proba = np.array(respuesta["predicciones"])
>>> y_proba.ronda(2)
 matriz([[0., 0.0], , 0. , 0. , 0. , 0. , 0. , 1. , 0.
,

```

```
[0.      ,  0.      , 0.99, 0.01, 0.          ,  0.      ,  0.      ,  0.      ,  0.
,  0. ],
[0.      , 0.97, 0.01, 0.          ,  0.      ,  0.      ,  0.      , 0.01, 0.
,  0. ]])
```

¡Hurra, tenemos las predicciones! El modelo está cerca del 100%.

Confía en que la primera imagen es un 7, 99% confía en que la segunda. imagen es un 2, y el 97% confía en que la tercera imagen es un 1. Eso es correcto.

La API REST es agradable y sencilla, y funciona bien cuando la entrada y los datos de salida no son demasiado grandes. Además, casi cualquier cliente La aplicación puede realizar consultas REST sin necesidad de realizar consultas adicionales. dependencias, mientras que otros protocolos no siempre son tan fáciles disponible. Sin embargo, está basado en JSON, que está basado en texto y bastante detallado. Por ejemplo, tuvimos que convertir la matriz NumPy en una Lista de Python, y cada flotante terminó representado como una cadena. Esto es muy ineficiente, tanto en términos de tiempo de serialización/deserialización, tenemos que convertir todos los flotadores en cuerdas y viceversa, y en términos de Tamaño de carga útil: muchos flotadores terminan siendo representados usando más de 15 caracteres, lo que se traduce en más de 120 bits para flotantes de 32 bits. Esta voluntad resulta en una alta latencia y uso de ancho de banda al transferir grandes Matrices numerosas.<sup>6</sup> Entonces, veamos cómo usar gRPC en su lugar.

#### CONSEJO

Al transferir grandes cantidades de datos, o cuando la latencia es importante, es Es mucho mejor usar la API gRPC, si el cliente la admite, ya que utiliza un formato binario compacto y un protocolo de comunicación eficiente basado en tramas HTTP/2.

Consultar TF Serving a través de la API gRPC

La API gRPC espera un protocolo PredictRequest serializado buffer como entrada, y genera un PredictResponse serializado

búfer de protocolo. Estos protobufs son parte de la biblioteca tensorflow-serving-api, que instalamos anteriormente. Primero, creemos la solicitud:

```
desde tensorflow_serving.apis.predict_pb2 importar PredictRequest
```

```
request = PredictRequest()
request.model_spec.name = model_name
request.model_spec.signature_name = "serving_default" input_name =
model.input_names[0] # == "flatten_input"
request.inputs[input_name].CopyFrom(tf.make_tensor_proto(X_new ))
```

Este código crea un búfer de protocolo PredictRequest y completa los campos obligatorios, incluido el nombre del modelo (definido anteriormente), el nombre de la firma de la función que queremos llamar y, finalmente, los datos de entrada, en forma de un búfer de protocolo Tensor. La función `tf.make_tensor_proto()` crea un búfer de protocolo Tensor basado en el tensor o matriz NumPy dado, en este caso `X_new`.

A continuación, enviaremos la solicitud al servidor y obtendremos su respuesta. Para ello necesitaremos la biblioteca grpcio, que está preinstalada en Colab:

```
importar grpc
desde tensorflow_serving.apis importar
predict_service_pb2_grpc

canal = grpc.insecure_channel('localhost:8500') predict_service =
predict_service_pb2_grpc.PredictionServiceStub(canal) respuesta =
predict_service.Predict(solicitud, tiempo de espera = 10.0)
```

El código es bastante sencillo: después de las importaciones, creamos un canal de comunicación gRPC para localhost en el puerto TCP 8500, luego creamos un servicio gRPC a través de este canal y lo usamos para enviar una solicitud, con un tiempo de espera de 10 segundos. Tenga en cuenta que la llamada es sincrónica:

se bloqueará hasta que reciba la respuesta o cuando expire el período de tiempo de espera. En este ejemplo, el canal es inseguro (sin cifrado ni autenticación), pero gRPC y TF Serving también admiten canales seguros a través de SSL/TLS.

A continuación, conviertamos el búfer del protocolo PredictResponse en un tensor:

```
nombre_salida = model.nombres_salida[0] # == "denso_1" salidas_proto =
respuesta.salidas[nombre_salida] y_proba = tf.make_ndarray(salidas_proto)
```

Si ejecuta este código e imprime y\_proba.round(2), obtendrá exactamente las mismas probabilidades de clase estimadas que antes. Y eso es todo: en solo unas pocas líneas de código, ahora puedes acceder a tu modelo de TensorFlow de forma remota, usando REST o gRPC.

Implementación de una nueva versión del modelo

Ahora creemos una nueva versión del modelo y exportemos un modelo guardado, esta vez al directorio my\_mnist\_model/0002 :

```
modelo = [...] # construir y entrenar una nueva versión del modelo MNIST

model_version = "0002" model_path
= Ruta(model_name) / model_version model.save(model_path,
save_format="tf")
```

A intervalos regulares (el retraso es configurable), TF Serving comprueba el directorio de modelos en busca de nuevas versiones de modelos. Si encuentra uno, maneja automáticamente la transición con elegancia: de forma predeterminada, responde las solicitudes pendientes (si las hay) con la versión del modelo anterior, mientras maneja las nuevas solicitudes con la nueva versión. Tan pronto como se responde a cada solicitud pendiente, se descarga la versión del modelo anterior. Puede ver esto en funcionamiento en los registros de TF Serving (en my\_server.log):

[...]

Leyendo SavedModel de: /models/my\_mnist\_model/0002 Leyendo meta gráfico con etiquetas {serve} [...]

Versión de servicio cargada correctamente {nombre: my\_mnist\_model versión: 2}

Versión de servicio inactivo {nombre: my\_mnist\_model versión: 1}

Se ha desactivado la versión servible {nombre: my\_mnist\_model versión: 1}

Descargando la versión servible {nombre: my\_mnist\_model versión: 1}

CONSEJO

Si SavedModel contiene algunas instancias de ejemplo en el directorio activos/extra , puede configurar TF Serving para ejecutar el nuevo modelo en estas instancias antes de comenzar a usarlo para atender solicitudes. A esto se le llama calentamiento del modelo: garantizará que todo esté cargado correctamente, evitando largos tiempos de respuesta para las primeras solicitudes.

Este enfoque ofrece una transición fluida, pero puede utilizar demasiada RAM, especialmente la RAM de la GPU, que generalmente es la más limitada. En este caso, puede configurar TF Serving para que maneje todas las solicitudes pendientes con la versión del modelo anterior y las descargue antes de cargar y usar la nueva versión del modelo. Esta configuración evitará tener dos versiones de modelo cargadas al mismo tiempo, pero el servicio estará indisponible por un corto período.

Como puede ver, TF Serving facilita la implementación de nuevos modelos. Además, si descubre que la versión 2 no funciona tan bien como esperaba, volver a la versión 1 es tan sencillo como eliminar el directorio my\_mnist\_model/0002 .

## CONSEJO

Otra gran característica de TF Serving es su capacidad de procesamiento por lotes automático, que puede activar usando la opción `--enable_batching` al inicio. Cuando TF Serving recibe varias solicitudes en un corto período de tiempo (el retraso es configurable), las agrupará automáticamente antes de usar el modelo. Esto ofrece un aumento significativo del rendimiento al aprovechar la potencia de la GPU. Una vez que el modelo devuelve las predicciones, TF Serving envía cada predicción al cliente correcto.

Puede cambiar un poco de latencia por un mayor rendimiento aumentando el retraso del procesamiento por lotes (consulte la opción `--batching_parameters_file`).

Si espera recibir muchas consultas por segundo, deberá implementar TF Serving en varios servidores y equilibrar la carga de las consultas (consulte la [Figura 19-2](#)). Esto requerirá implementar y administrar muchos contenedores TF Serving en estos servidores. Una forma de manejar esto es usar una herramienta como [Kubernetes](#), que es un sistema de código abierto para simplificar la orquestación de contenedores en muchos servidores. Si no desea comprar, mantener y actualizar toda la infraestructura de hardware, querrá utilizar máquinas virtuales en una plataforma en la nube como Amazon AWS, Microsoft Azure, Google Cloud Platform, IBM Cloud, Alibaba Cloud, Oracle Cloud o alguna otra oferta de plataforma como servicio (PaaS). Administrar todas las máquinas virtuales, manejar la orquestación de contenedores (incluso con la ayuda de Kubernetes), encargarse de la configuración, el ajuste y el monitoreo de TF Serving: todo esto puede ser un trabajo de tiempo completo. Afortunadamente, algunos proveedores de servicios pueden encargarse de todo esto por usted. En este capítulo usaremos Vertex AI: es la única plataforma con TPU en la actualidad; es compatible con TensorFlow 2, Scikit-Learn y XGBoost; y ofrece un buen conjunto de servicios de inteligencia artificial. Sin embargo, hay varios otros proveedores en este espacio que también son capaces de ofrecer modelos de TensorFlow, como Amazon AWS SageMaker y Microsoft AI Platform, así que asegúrese de revisarlos también.

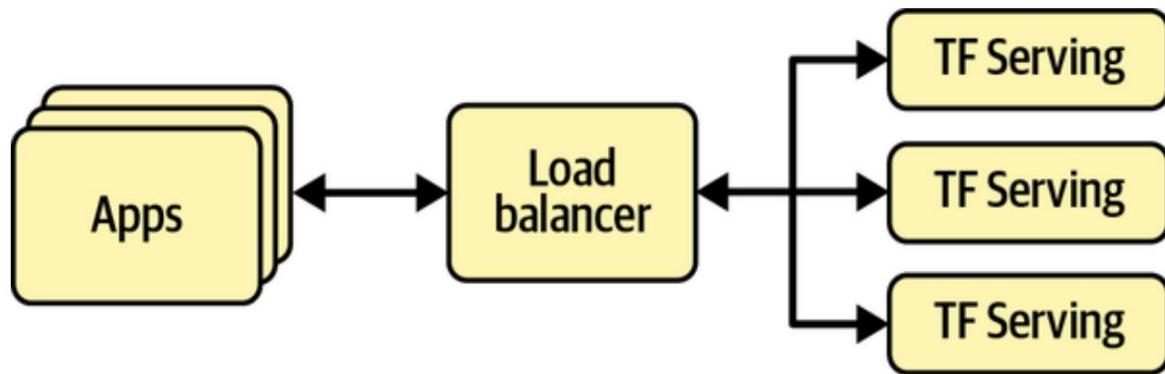


Figura 19-2. Ampliación de TF Serving con equilibrio de carga

¡Ahora veamos cómo servir nuestro maravilloso modelo MNIST en la nube!

## Creación de un servicio de predicción en Vertex AI

Vertex AI es una plataforma dentro de Google Cloud Platform (GCP) que ofrece una amplia gama de herramientas y servicios relacionados con la IA. Puede cargar conjuntos de datos, hacer que los humanos los etiqueten, almacenar funciones de uso común en un almacén de funciones y usarlas para entrenamiento o en producción, y entrenar modelos en muchos servidores GPU o TPU con ajuste automático de hiperparámetros o búsqueda de arquitectura de modelo (AutoML). También puede administrar sus modelos entrenados, usarlos para hacer predicciones por lotes sobre grandes cantidades de datos, programar múltiples trabajos para sus flujos de trabajo de datos, servir sus modelos a través de REST o gRPC a escala y experimentar con sus datos y modelos dentro de un entorno alojado de Jupyter. Llamado Banco de Trabajo. Incluso hay un servicio Matching Engine que le permite comparar vectores de manera muy eficiente (es decir, vecinos más cercanos aproximados). GCP también incluye otros servicios de inteligencia artificial, como API para visión por computadora, traducción, conversión de voz a texto y más.

Antes de comenzar, hay que encargarse de una pequeña configuración:

1. Inicie sesión en su cuenta de Google y luego vaya a la [consola de Google Cloud Platform](#) (consulte [la Figura 19-3](#)). Si no tienes una cuenta de Google, tendrás que crear una.

2. Si es la primera vez que utilizas GCP, deberás leer y aceptar los términos y condiciones. A los nuevos usuarios se les ofrece una prueba gratuita, que incluye \$300 de crédito de GCP que pueden usar en el transcurso de 90 días (a partir de mayo de 2022). Sólo necesitará una pequeña parte de esa cantidad para pagar los servicios que utilizará en este capítulo. Al registrarse para una prueba gratuita, aún deberá crear un perfil de pago e ingresar su número de tarjeta de crédito: se usa con fines de verificación (probablemente para evitar que las personas usen la prueba gratuita varias veces), pero no se le facturará. los primeros \$300, y después de eso solo se le cobrará si opta por actualizarse a una cuenta paga.

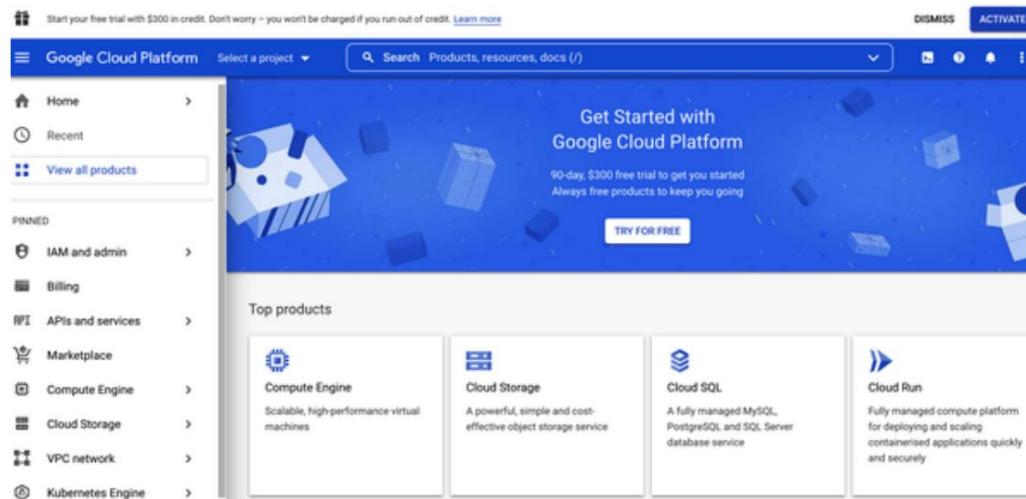


Figura 19-3. Consola de Google Cloud Platform

3. Si ha utilizado GCP antes y su prueba gratuita ha caducado, entonces los servicios que utilizará en este capítulo le costarán algo de dinero. No debería ser demasiado, especialmente si recuerdas desactivar los servicios cuando ya no los necesites. Asegúrese de comprender y aceptar las condiciones de precios antes de ejecutar cualquier servicio. ¡Por la presente declino cualquier responsabilidad si los servicios terminan costando más de lo que esperaba! También asegúrese de que su cuenta de facturación esté activa. Para comprobarlo, abra el menú de navegación en la parte superior izquierda y haga clic en Facturación, luego asegúrese de haber configurado un método de pago y de que la cuenta de facturación esté activa.

4. Cada recurso en GCP pertenece a un proyecto. Esto incluye todas las máquinas virtuales que pueda utilizar, los archivos que almacene y los trabajos de capacitación que execute. Cuando creas una cuenta, GCP crea automáticamente un proyecto para ti, llamado "Mi primer proyecto". Si lo desea, puede cambiar su nombre para mostrar yendo a la configuración del proyecto: en el menú de navegación , seleccione "IAM y administrador → Configuración", cambie el nombre para mostrar del proyecto y haga clic en GUARDAR. Tenga en cuenta que el proyecto también tiene un ID y un número únicos. Puede elegir el ID del proyecto cuando crea un proyecto, pero no puede cambiarlo más adelante. El número de proyecto se genera automáticamente y no se puede cambiar. Si desea crear un nuevo proyecto, haga clic en el nombre del proyecto en la parte superior de la página, luego haga clic en NUEVO PROYECTO e ingrese el nombre del proyecto. También puede hacer clic en EDITAR para configurar el ID del proyecto. Asegúrese de que la facturación esté activa para este nuevo proyecto para que se puedan facturar las tarifas del servicio (a sus créditos gratuitos, si corresponde).

#### ADVERTENCIA

Configure siempre una alarma para recordarse que debe apagar los servicios cuando sepa que solo los necesitará durante unas pocas horas; de lo contrario, podría dejarlos funcionando durante días o meses, incurriendo en costos potencialmente significativos.

5. Ahora que tienes una cuenta de GCP y un proyecto, y la facturación está activada, debes activar las API que necesitas. En el menú de navegación , seleccione "API y servicios" y asegúrese de que la API de Cloud Storage esté habilitada. Si es necesario, haga clic en + HABILITAR APIs Y SERVICIOS, busque Cloud Storage y habilítelo. Habilite también la API de Vertex AI.

Podrías continuar haciendo todo a través de la consola de GCP, pero recomiendo usar Python en su lugar: de esta manera puedes escribir scripts para automatizar casi todo lo que quieras con GCP, y a menudo es más

más conveniente que hacer clic en menús y formularios, especialmente para tareas comunes.

## CLI Y SHELL DE LA NUBE DE GOOGLE

La interfaz de línea de comandos (CLI) de Google Cloud incluye el comando gcloud, que le permite controlar casi todo en GCP, y gsutil, que le permite interactuar con Google Cloud Storage. Esta CLI está preinstalada en Colab: todo lo que necesita hacer es autenticarse usando google.auth.authenticate\_user() y listo. Por ejemplo, la lista de configuración de !gcloud mostrará la configuración.

GCP también ofrece un entorno de shell preconfigurado llamado Google Cloud Shell, que puede utilizar directamente en su navegador web; se ejecuta en una máquina virtual Linux (Debian) gratuita con el SDK de Google Cloud ya preinstalado y configurado para usted, por lo que no es necesario autenticarse. Cloud Shell está disponible en cualquier lugar de GCP: simplemente haga clic en el ícono Activar Cloud Shell en la parte superior derecha de la página (consulte la [Figura 19-4](#)).

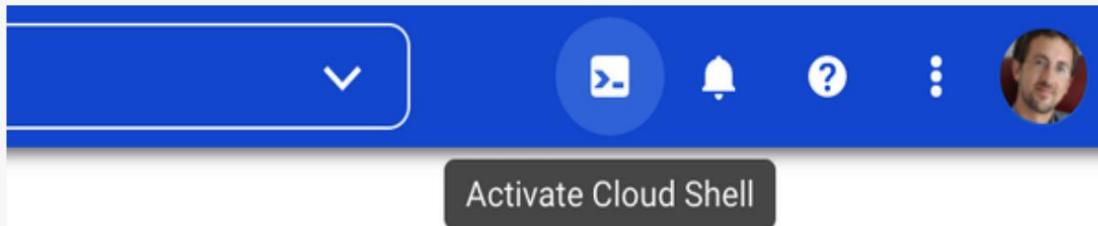


Figura 19-4. Activando Google Cloud Shell

Si prefiere [instalar la CLI en su máquina](#), luego, después de la instalación, debes inicializarlo ejecutando gcloud init: sigue las instrucciones para iniciar sesión en GCP y otorgar acceso a tus recursos de GCP, luego selecciona el proyecto de GCP predeterminado que deseas usar (si tienes más de uno) y el proyecto predeterminado. región donde desea que se ejecuten sus trabajos.

Lo primero que debe hacer antes de poder utilizar cualquier servicio de GCP es autenticarse. La solución más sencilla al utilizar Colab es ejecutar el siguiente código:

```
desde google.colab importar autenticación
```

```
autenticación.authenticate_user()
```

El proceso de autenticación se basa en **OAuth 2.0**: una ventana emergente le pedirá que confirme que desea que el cuaderno Colab acceda a sus credenciales de Google. Si acepta, debe seleccionar la misma cuenta de Google que utilizó para GCP. Luego se le pedirá que confirme que acepta otorgarle a Colab acceso completo a todos sus datos en Google Drive y GCP. Si permite el acceso, solo el cuaderno actual tendrá acceso y solo hasta que caduque el tiempo de ejecución de Colab. Obviamente, sólo debes aceptar esto si confías en el código del cuaderno.

#### ADVERTENCIA

Si no está trabajando con los cuadernos oficiales de <https://github.com/ageron/handson-ml3>, entonces debes tener mucho cuidado: si el autor del cuaderno es travieso, podría incluir código para hacer lo que quiera con tus datos.

## AUTENTICACIÓN Y AUTORIZACIÓN EN GCP

En general, el uso de la autenticación OAuth 2.0 solo se recomienda cuando una aplicación debe acceder a los datos o recursos personales del usuario desde otra aplicación, en nombre del usuario. Por ejemplo, algunas aplicaciones permiten al usuario guardar datos en su Google Drive, pero para ello la aplicación primero necesita que el usuario se autentique en Google y permita el acceso a Google Drive. Por lo general, la aplicación sólo te pedirá el nivel de acceso que necesita; No será un acceso ilimitado: por ejemplo, la aplicación sólo solicitará acceso a Google Drive, no a Gmail ni a ningún otro servicio de Google. Además, la autorización suele caducar al cabo de un tiempo y siempre se puede revocar.

Cuando una aplicación necesita acceder a un servicio en GCP en su propio nombre, no en nombre del usuario, generalmente debe usar una cuenta de servicio. Por ejemplo, si crea un sitio web que necesita enviar solicitudes de predicción a un punto final de Vertex AI, entonces el sitio web accederá al servicio en su propio nombre. No hay datos ni recursos a los que deba acceder en la cuenta de Google del usuario. De hecho, muchos usuarios del sitio web ni siquiera tendrán una cuenta de Google. Para este escenario, primero debe crear una cuenta de servicio. Seleccione "IAM y administrador → Cuentas de servicio" en el menú de navegación de la consola GCP (o use el cuadro de búsqueda), luego haga clic en + CREAR CUENTA DE SERVICIO, complete la primera página del formulario (nombre de la cuenta de servicio, ID, descripción) y haga clic en CREAR Y CONTINUAR. A continuación, debe otorgarle a esta cuenta algunos derechos de acceso. Seleccione la función "Usuario de Vertex AI": esto permitirá que la cuenta de servicio haga predicciones y utilice otros servicios de Vertex AI, pero nada más. Haga clic en CONTINUAR. Ahora, opcionalmente, puedes otorgar acceso a algunos usuarios a la cuenta de servicio: esto es útil cuando tu cuenta de usuario de GCP es parte de una organización y deseas autorizar a otros usuarios de la organización a implementar aplicaciones que se basarán en esta.

cuenta de servicio, o para administrar la cuenta de servicio en sí. A continuación, haga clic en LISTO.

Una vez que haya creado una cuenta de servicio, su aplicación debe autenticarse como esa cuenta de servicio. Hay varias maneras de hacer eso. Si su aplicación está alojada en GCP (por ejemplo, si está codificando un sitio web alojado en Google Compute Engine), entonces la solución más sencilla y segura es adjuntar la cuenta de servicio al recurso de GCP que aloja su sitio web, como una instancia de VM o un servicio de Google App Engine. Esto se puede hacer al crear el recurso de GCP, seleccionando la cuenta de servicio en la sección "Identidad y acceso a API".

Algunos recursos, como las instancias de VM, también le permiten adjuntar la cuenta de servicio después de crear la instancia de VM: debe detenerla y editar su configuración. En cualquier caso, una vez que se adjunta una cuenta de servicio a una instancia de VM, o cualquier otro recurso de GCP que ejecute su código, las bibliotecas cliente de GCP (que se analizan en breve) se autenticarán automáticamente como la cuenta de servicio elegida, sin necesidad de realizar ningún paso adicional.

Si su aplicación está alojada en Kubernetes, debe utilizar el servicio Workload Identity de Google para asignar la cuenta de servicio correcta a cada cuenta de servicio de Kubernetes. Si su aplicación no está alojada en GCP (por ejemplo, si solo está ejecutando el cuaderno Jupyter en su propia máquina), puede usar el servicio Workload Identity Federation (esa es la opción más segura pero más difícil) o simplemente generar una clave de acceso para su cuenta de servicio, guárdelo en un archivo JSON y apúntelo con la variable de entorno GOOGLE\_APPLICATION\_CREDENTIALS para que su aplicación cliente pueda acceder a él. Puede administrar las claves de acceso haciendo clic en la cuenta de servicio que acaba de crear y luego abriendo la pestaña LLAVES. Asegúrese de mantener el archivo de clave en secreto: es como una contraseña para la cuenta de servicio.

Para obtener más detalles sobre cómo configurar la autenticación y la autorización para que su aplicación pueda acceder a los servicios de GCP, consulte la

documentación.

Ahora creamos un depósito de Google Cloud Storage para almacenar nuestros SavedModels (un depósito de GCS es un contenedor para sus datos). Para ello utilizaremos la biblioteca google-cloud-storage, que está preinstalada en Colab. Primero creamos un objeto Cliente, que servirá como interfaz con GCS, luego lo usamos para crear el depósito:

```
desde google.cloud importar almacenamiento
```

```
project_id = "my_project" # cambia esto a tu ID de proyecto bucket_name = "my_bucket" #
cambia esto a un depósito único
nombre
ubicación = "us-central1"
```

```
cliente_almacenamiento = almacenamiento.Cliente(proyecto=id_proyecto) depósito
= cliente_almacenamiento.create_bucket(nombre_depósito, ubicación=ubicación)
```

#### CONSEJO

Si desea reutilizar un depósito existente, reemplace la última línea con depósito = almacenamiento\_cliente.bucket(nombre\_depósito). Asegúrese de que la ubicación esté configurada en la región del depósito.

GCS utiliza un único espacio de nombres mundial para los depósitos, por lo que lo más probable es que nombres simples como "aprendizaje automático" no estén disponibles. Asegúrese de que el nombre del depósito cumpla con las convenciones de nomenclatura DNS, ya que puede usarse en registros DNS. Además, los nombres de los depósitos son públicos, así que no pongas nada privado en el nombre. Es común usar su nombre de dominio, el nombre de su empresa o su ID de proyecto como prefijo para garantizar la unicidad, o simplemente usar un número aleatorio como parte del nombre.

Puede cambiar la región si lo desea, pero asegúrese de elegir una que admita GPU. Además, es posible que desee considerar el hecho de que

los precios varían mucho entre regiones, algunas regiones producen mucho más CO que otras, algunas regiones no admiten todos los servicios y el uso de un segmento de una sola región mejora el rendimiento. Consulte [la lista de regiones de Google Cloud](#) y [la documentación de Vertex AI sobre ubicaciones](#) para más detalles. Si no está seguro, lo mejor sería seguir con "nosotros-central1".

A continuación, carguemos el directorio `my_mnist_model` en el nuevo depósito. Los archivos en GCS se denominan blobs (u objetos) y, en el fondo, todos se colocan en el depósito sin ninguna estructura de directorio. Los nombres de blobs pueden ser cadenas Unicode arbitrarias e incluso pueden contener barras diagonales (/). La consola GCP y otras herramientas utilizan estas barras para dar la ilusión de que hay directorios. Entonces, cuando cargamos el directorio `my_mnist_model`, solo nos preocupamos por los archivos, no por los directorios:

```
def upload_directory(bucket, dirpath): dirpath =  
    Ruta(dirpath) para la ruta del  
    archivo en dirpath.glob("**/*"): if filepath.is_file():  
        blob =  
  
        bucket.blob(filepath.relative_to(dirpath.parent).as_posix() )  
  
        blob.upload_from_filename(ruta de archivo)  
  
upload_directory(depósito, "my_mnist_model")
```

Esta función funciona bien ahora, pero sería muy lenta si hubiera muchos archivos para cargar. No es demasiado difícil acelerarlo enormemente mediante subprocessos múltiples (consulte el cuaderno para ver una implementación). Alternativamente, si tiene la CLI de Google Cloud, puede usar el siguiente comando:

```
!gsutil -m cp -r my_mnist_model gs://{bucket_name}/
```

A continuación, hablemos de Vertex AI sobre nuestro modelo MNIST. Para comunicarnos con Vertex AI, podemos utilizar la biblioteca `google-cloud-aiplatform`

(todavía usa el antiguo nombre de AI Platform en lugar de Vertex AI). No está preinstalado en Colab, por lo que debemos instalarlo. Después de eso, podemos importar la biblioteca e inicializarla, solo para especificar algunos valores predeterminados para el ID del proyecto y la ubicación, luego podemos crear un nuevo modelo de Vertex AI: especificamos un nombre para mostrar, la ruta GCS a nuestro modelo (en este caso la versión 0001) y la URL del contenedor Docker que queremos que Vertex AI use para ejecutar este modelo. Si visita esa URL y navega hacia arriba un nivel, encontrará otros contenedores que puede usar. Este es compatible con TensorFlow 2.8 con una GPU:

```
desde google.cloud importar aiplatform

server_image = "gcr.io/cloud-aiplatform/prediction/tf2-gpu.2-8:latest"

aiplatform.init(proyecto=proyecto_id, ubicación=ubicación) mnist_model =
aiplatform.Model.upload( display_name="mnist",
    artefacto_uri=f"gs://
{bucket_name}/my_mnist_model/0001", server_container_image_uri=server_image,
)

)
```

Ahora implementemos este modelo para que podamos consultararlo mediante gRPC o API REST para hacer predicciones. Para esto primero necesitamos crear un punto final. Esto es a lo que se conectan las aplicaciones cliente cuando quieren acceder a un servicio. Luego necesitamos implementar nuestro modelo en este punto final:

```
punto_final = aiplatform.Endpoint.create(display_name="mnist- endpoint")
```

```
endpoint.deploy( mnist_model,
    min_replica_count=1,
    max_replica_count=5, machine_type="n1-
standard-4", accelerator_type="NVIDIA_TESLA_K80",
    accelerator_count=1
)
```

Este código puede tardar unos minutos en ejecutarse, porque Vertex AI necesita configurar una máquina virtual. En este ejemplo, utilizamos una máquina bastante básica de tipo n1-standard-4 (consulte <https://homl.info/machinetypes> para otros tipos). También utilizamos una GPU básica de tipo NVIDIA\_TESLA\_K80 (ver <https://homl.info/accelerators> para otros tipos). Si seleccionó otra región que no sea "us-central1", es posible que deba cambiar el tipo de máquina o el tipo de acelerador a valores admitidos en esa región (por ejemplo, no todas las regiones tienen GPU Nvidia Tesla K80).

### NOTA

Google Cloud Platform impone varias cuotas de GPU, tanto a nivel mundial como por región: no se pueden crear miles de nodos de GPU sin la autorización previa de Google. Para verificar sus cuotas, abra "IAM y administrador → Cuotas" en la consola de GCP. Si algunas cuotas son demasiado bajas (por ejemplo, si necesita más GPU en una región en particular), puede solicitar que se aumenten; suele tardar unas 48 horas.

Vertex AI generará inicialmente la cantidad mínima de nodos de cómputo (solo uno en este caso), y cada vez que la cantidad de consultas por segundo sea demasiado alta, generará más nodos (hasta el número máximo que definiste, cinco en este caso). y equilibrará la carga de las consultas entre ellos. Si la tasa de QPS baja por un tiempo, Vertex AI detendrá los nodos de cálculo adicionales automáticamente. Por lo tanto, el coste está directamente relacionado con la carga, así como con los tipos de máquina y acelerador que haya seleccionado y la cantidad de datos que almacene en GCS. Este modelo de precios es ideal para usuarios ocasionales y para servicios con picos de uso importantes. También es ideal para startups: el precio se mantiene bajo hasta que la startup realmente se pone en marcha.

¡Felicitaciones, ha implementado su primer modelo en la nube!

Ahora consultemos este servicio de predicción:

```
respuesta = endpoint.predict(instancias=X_new.tolist())
```

Primero necesitamos convertir las imágenes que queremos clasificar a Python. lista, como lo hicimos anteriormente cuando enviamos solicitudes a TF Serving utilizando el API DESCANSO. El objeto de respuesta contiene las predicciones, representado como una lista Python de listas de flotadores. Redondeémoslos a dos lugares decimales y convertirlos a una matriz NumPy:

```
>>> importar números como np
>>> np.ronda(respuesta.predicciones, 2)
matriz([[0. , 0. 0. ], 0. , 0. , , 0. , 0. , 0. , 1. , 0.
,
[0. , 0. , 0.99, 0.01, 0. , , 0. , 0. , 0. , 0. ],
, 0. ],
[0. , 0.97, 0.01, 0. , , 0. , 0. , 0. , 0.01, 0.
,
0. ]])
```

¡Sí! Obtenemos exactamente las mismas predicciones que antes. ahora tenemos un Buen servicio de predicción que se ejecuta en la nube desde el que podemos consultar. en cualquier lugar de forma segura y que puede ampliarse o reducirse automáticamente dependiendo del número de QPS. Cuando haya terminado de usar el punto final, no olvides eliminarlo para evitar pagar por nada:

```
endpoint.undeploy_all() # anular la implementación de todos los modelos del
punto final
punto final.eliminar()
```

Ahora veamos cómo ejecutar un trabajo en Vertex AI para hacer predicciones en un lote de datos potencialmente muy grande.

## Ejecución de trabajos de predicción por lotes en Vertex AI

Si tenemos un gran número de predicciones que hacer, entonces en lugar de llamando a nuestro servicio de predicción repetidamente, podemos pedirle a Vertex AI que se ejecute un trabajo de predicción para nosotros. Esto no requiere un punto final, sólo un modelo. Por ejemplo, ejecutemos un trabajo de predicción en las primeras 100 imágenes. del conjunto de prueba, utilizando nuestro modelo MNIST. Para esto primero necesitamos Prepare el lote y cárguelo en GCS. Una manera de hacer esto es

cree un archivo que contenga una instancia por línea, cada una formateada como un valor JSON (este formato se llama Líneas JSON ) y luego pase este archivo a Vertex AI. Entonces, creamos un archivo JSON Lines en un nuevo directorio y luego carguemos este directorio en GCS:

```

ruta_batch = Ruta("my_mnist_batch")
ruta_batch.mkdir(exist_ok=True) con
open(batch_path / "my_mnist_batch.jsonl", "w") como jsonl_file:

    para imagen en X_test[:100].tolist():
        jsonl_file.write(json.dumps(imagen))
        jsonl_file.write("\n")

directorio_carga(depósito, ruta_lote)

```

Ahora estamos listos para iniciar el trabajo de predicción, especificando el nombre del trabajo, el tipo y número de máquinas y aceleradores a usar, el Ruta GCS al archivo JSON Lines que acabamos de crear y la ruta al archivo Directorio GCS donde Vertex AI guardará las predicciones del modelo:

```
batch_prediction_job =
```

```

mnist_model.batch_predict( job_display_name="my_batch_prediction_job",
    machine_type="n1-standard-4", Starting_replica_count=1, max_replica_count=5, accelerator_type="#"
    gcs_destination_prefix=f"gs://{bucket_name}/my_mnist_predictions/", sync=True
# configúrelo en False si no desea esperar a que se complete )

```

## CONSEJO

Para lotes grandes, puede dividir las entradas en varios archivos de líneas JSON y enumerarlos todos mediante el argumento gcs\_source.

Esto llevará unos minutos, principalmente para generar los nodos de cómputo en Vertex AI. Una vez que se complete este comando, las predicciones estarán disponibles en un conjunto de archivos llamado algo así como prediction.results-00001-of-00002. Estos archivos utilizan el formato JSON Lines de forma predeterminada y cada valor es un diccionario que contiene una instancia y su predicción correspondiente (es decir, 10 probabilidades). Las instancias se enumeran en el mismo orden que las entradas. El trabajo también genera archivos de errores de predicción\*, que pueden resultar útiles para depurar si algo sale mal. Podemos iterar a través de todos estos archivos de salida usando batch\_prediction\_job.iter\_outputs(), así que repasemos todas las predicciones y almacenémoslas.

```
y_probas = [] para
blob en batch_prediction_job.iter_outputs():
    si "predicción.resultados" en blob.name:
        para la línea en blob.download_as_text().splitlines(): y_proba =
            json.loads(line)["prediction"] y_probas.append(y_proba)
```

Ahora veamos qué tan buenas son estas predicciones:

```
>>> y_pred = np.argmax(y_probas, eje=1) >>> exactitud =
np.sum(y_pred == y_test[:100]) / 100 0.98
```

¡Bien, 98% de precisión!

El formato JSON Lines es el predeterminado, pero cuando se trata de instancias grandes, como imágenes, es demasiado detallado. Afortunadamente, el método batch\_predict() acepta un argumento instances\_formato que le permite elegir otro formato si lo desea. Por defecto

a "jsonl", pero puedes cambiarlo a "csv", "tf-record", "tf-record-gzip", "bigquery" o "file-list". Si lo configura en "lista de archivos", entonces el argumento gcs\_source debería apuntar a un archivo de texto que contiene una ruta de archivo de entrada por línea; por ejemplo, apuntando a archivos de imagen PNG. Vertex AI leerá estos archivos como binarios, los codificará usando Base64 y pasará las cadenas de bytes resultantes al modelo.

Esto significa que debe agregar una capa de preprocesamiento en su modelo para analizar las cadenas Base64, usando `tf.io.decode_base64()`. Si los archivos son imágenes, debe analizar el resultado usando una función como `tf.io.decode_image()` o `tf.io.decode_png()`, como se analiza en el [Capítulo 13](#).

Cuando haya terminado de usar el modelo, puede eliminarlo si lo desea ejecutando `mnist_model.delete()`. También puedes eliminar los directorios que creaste en tu depósito de GCS, opcionalmente el depósito en sí (si está vacío) y el trabajo de predicción por lotes:

```
para prefijo en ["my_mnist_model/", "my_mnist_batch/", "my_mnist_predictions/"]:  
    blobs  
    = bucket.list_blobs(prefix=prefix) para  
        blob en blobs:  
  
        blob.eliminar()  
  
    bucket.delete() # si el depósito está vacío batch_prediction_job.delete()
```

Ahora sabe cómo implementar un modelo en Vertex AI, crear un servicio de predicción y ejecutar trabajos de predicción por lotes. Pero, ¿qué sucede si desea implementar su modelo en una aplicación móvil? ¿O a un dispositivo integrado, como un sistema de control de calefacción, un rastreador de actividad física o un vehículo autónomo?

## Implementación de un modelo en un dispositivo móvil o integrado

### Dispositivo

Los modelos de aprendizaje automático no se limitan a ejecutarse en grandes servidores centralizados con múltiples GPU: pueden ejecutarse más cerca de la fuente de datos (esto se llama computación de borde), por ejemplo, en el dispositivo móvil del usuario o en un dispositivo integrado. Hay muchos beneficios al descentralizar los cálculos y moverlos hacia el borde: permite que el dispositivo sea inteligente incluso cuando no está conectado a Internet, reduce la latencia al no tener que enviar datos a un servidor remoto y reduce la carga en los servidores, y puede mejorar la privacidad, ya que los datos del usuario pueden permanecer en el dispositivo.

Sin embargo, implementar modelos hasta el borde también tiene sus desventajas. Los recursos informáticos del dispositivo son generalmente pequeños en comparación con un servidor robusto con múltiples GPU. Es posible que un modelo grande no quiera en el dispositivo, que utilice demasiada RAM y CPU y que la descarga tarde demasiado. Como resultado, es posible que la aplicación deje de responder y que el dispositivo se caliente y se quede sin batería rápidamente. Para evitar todo esto, es necesario realizar un modelo ligero y eficiente, sin sacrificar demasiado de su precisión. La [TFLite](#) La biblioteca proporciona varias herramientas para ayudarle a implementar sus modelos en el borde, con tres objetivos principales:

- Reduzca el tamaño del modelo para acortar el tiempo de descarga y reducir el uso de RAM.
- Reduzca la cantidad de cálculos necesarios para cada predicción, para reducir la latencia, el uso de la batería y el calentamiento.
- Adapte el modelo a las restricciones específicas del dispositivo.

Para reducir el tamaño del modelo, el conversor de modelos de TFLite puede tomar un SavedModel y comprimirlo a un formato mucho más ligero basado en [FlatBuffers](#). Esta es una biblioteca de serialización multiplataforma eficiente (un poco como buffers de protocolo) creada inicialmente por Google para juegos. Está diseñado para que puedas cargar FlatBuffers directamente en la RAM sin ningún preprocessamiento: esto reduce el tiempo de carga y el uso de memoria.

Una vez que el modelo se carga en un dispositivo móvil o integrado, el

El intérprete TFLite lo ejecutará para hacer predicciones. Así es como puedes convertir un SavedModel en un FlatBuffer y guardarlo en un archivo .tflite :

```
convertidor =
tf.lite.TFLiteConverter.from_saved_model(str(model_path)) tflite_model = convertidor.convert()
con open("my_converted_savedmodel.tflite", "wb") como
f: f.write(tflite_model)
```

## CONSEJO

También puede guardar un modelo de Keras directamente en un FlatBuffer usando `tf.lite.TFLiteConverter.from_keras_model(model)`.

El conversor también optimiza el modelo, tanto para reducirlo como para reducir su latencia. Poda todas las operaciones que no son necesarias para hacer predicciones (como operaciones de entrenamiento) y optimiza los cálculos siempre que sea posible; por ejemplo,  $3 \times a + 4 \times a + 5 \times a$  se convertirá en  $12 \times a$ . Además, intenta fusionar operaciones siempre que sea posible. Por ejemplo, si es posible, las capas de normalización por lotes terminan plegadas en las operaciones de suma y multiplicación de la capa anterior. Para tener una buena idea de cuánto TFLite puede optimizar un modelo, descargue uno de los **modelos TFLite previamente entrenados**, como `Inception_V1_quant` (haga clic en `tflite&pb`), descomprima el archivo y luego abra la excelente **herramienta de visualización de gráficos Netron** y cargue el archivo .pb para ver el modelo original. Es un gráfico grande y complejo, ¿verdad? A continuación, abre el modelo .tflite optimizado y ¡maravíllate ante su belleza!

Otra forma de reducir el tamaño del modelo (además de simplemente utilizar arquitecturas de redes neuronales más pequeñas) es mediante el uso de anchos de bits más pequeños: por ejemplo, si utiliza medios flotantes (16 bits) en lugar de flotantes normales (32 bits), el modelo el tamaño se reducirá en un factor de 2, a costa de una caída (generalmente pequeña) de precisión. Además, el entrenamiento será más rápido y utilizará aproximadamente la mitad de la cantidad de RAM de la GPU.

El conversor de TFLite puede ir más allá, al cuantificar los pesos del modelo hasta números enteros de 8 bits de punto fijo. Esto conduce a una reducción de tamaño cuatro veces mayor en comparación con el uso de flotantes de 32 bits. El enfoque más simple se llama cuantificación post-entrenamiento: simplemente cuantifica los pesos después del entrenamiento, utilizando una técnica de cuantificación simétrica bastante básica pero eficiente. Encuentra el valor de peso absoluto máximo,  $m$ , luego asigna el rango de punto flotante  $-m$  a  $+m$  al rango de punto fijo (entero)  $-127$  a  $+127$ . Por ejemplo, si los pesos oscilan entre  $-1.5$  y  $+0.8$ , entonces los bytes  $-127$ ,  $0$  y  $+127$  corresponderán a los flotantes  $-1.5$ ,  $0.0$  y  $+1.5$ , respectivamente (consulte la Figura 19-5). Tenga en cuenta que  $0.0$  siempre se asigna a  $0$  cuando se utiliza cuantificación simétrica. Tenga en cuenta también que los valores de bytes  $+68$  a  $+127$  no se utilizarán en este ejemplo, ya que se asignan a flotantes mayores que  $+0.8$ .

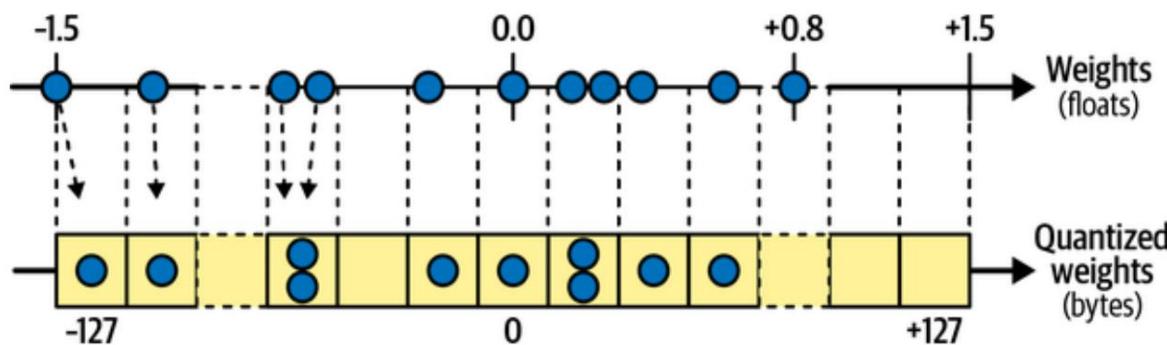


Figura 19-5. Desde flotantes de 32 bits hasta enteros de 8 bits, utilizando cuantificación simétrica

Para realizar esta cuantificación posterior al entrenamiento, simplemente agregue `DEFAULT` a la lista de optimizaciones del convertidor antes de llamar al método `convert()`:

```
convertidor.optimizaciones = [tf.lite.Optimize.DEFAULT]
```

Esta técnica reduce drásticamente el tamaño del modelo, lo que hace que su descarga sea mucho más rápida y utiliza menos espacio de almacenamiento. En tiempo de ejecución, los pesos cuantificados se vuelven a convertir en flotantes antes de usarse. Estos flotadores recuperados no son perfectamente idénticos a los originales, pero no están demasiado alejados, por lo que la pérdida de precisión suele ser aceptable. Para evitar volver a calcular los valores flotantes, todos los

tiempo, lo que ralentizaría gravemente el modelo, TFLite los almacena en caché: desafortunadamente, esto significa que esta técnica no reduce el uso de RAM y tampoco acelera el modelo. Es sobre todo útil para reducir el tamaño de la aplicación.

La forma más eficaz de reducir la latencia y el consumo de energía es cuantificar también las activaciones para que los cálculos se puedan realizar completamente con números enteros, sin necesidad de operaciones de punto flotante. Incluso cuando se utiliza el mismo ancho de bits (por ejemplo, enteros de 32 bits en lugar de flotantes de 32 bits), los cálculos de enteros utilizan menos ciclos de CPU, consumen menos energía y producen menos calor. Y si también reduce el ancho de bits (por ejemplo, hasta enteros de 8 bits), puede obtener enormes aceleraciones. Además, algunos dispositivos aceleradores de redes neuronales (como el Edge TPU de Google) solo pueden procesar números enteros, por lo que es obligatoria la cuantificación completa tanto de los pesos como de las activaciones. Esto se puede hacer después del entrenamiento; requiere un paso de calibración para encontrar el valor absoluto máximo de las activaciones, por lo que debe proporcionar una muestra representativa de datos de entrenamiento a TFLite (no es necesario que sea enorme), y procesará los datos a través del modelo y medirá la estadísticas de activación necesarias para la cuantificación. Este paso suele ser rápido.

El principal problema de la cuantización es que pierde un poco de precisión: es similar a añadir ruido a los pesos y activaciones. Si la caída de la precisión es demasiado grave, es posible que deba utilizar un entrenamiento consciente de la cuantificación. Esto significa agregar operaciones de cuantificación falsas al modelo para que pueda aprender a ignorar el ruido de cuantificación durante el entrenamiento; Las ponderaciones finales serán entonces más resistentes a la cuantificación. Además, el paso de calibración se puede realizar automáticamente durante el entrenamiento, lo que simplifica todo el proceso.

He explicado los conceptos básicos de TFLite, pero llegar hasta el final para codificar una aplicación móvil o integrada requeriría un libro dedicado. Afortunadamente, existen algunos: si desea obtener más información sobre cómo crear aplicaciones TensorFlow para dispositivos móviles e integrados, consulte los libros de O'Reilly [TinyML: Machine Learning with](#)

[TensorFlow en Arduino y microcontroladores de potencia ultrabaja](#), por Pete Warden (ex líder del equipo TFLite) y Daniel Situnayake y [AI y Machine Learning para el desarrollo en dispositivos](#), por Laurence Moroney.

Ahora, ¿qué sucede si desea utilizar su modelo en un sitio web, ejecutándolo directamente en el navegador del usuario?

## Ejecutar un modelo en una página web

Ejecutar su modelo de aprendizaje automático en el lado del cliente, en el navegador del usuario, en lugar de en el lado del servidor, puede resultar útil en muchos escenarios, como por ejemplo:

- Cuando su aplicación web se usa a menudo en situaciones donde la conectividad del usuario es intermitente o lenta (por ejemplo, un sitio web para excursionistas), ejecutar el modelo directamente en el lado del cliente es la única forma de hacer que su sitio web sea confiable.
- Cuando necesitas que las respuestas del modelo sean lo más rápidas posible (por ejemplo, para un juego en línea). Eliminar la necesidad de consultar el servidor para hacer predicciones definitivamente reducirá la latencia y hará que el sitio web sea mucho más receptivo.
- Cuando su servicio web hace predicciones basadas en algunos datos privados del usuario y desea proteger la privacidad del usuario haciendo predicciones en el lado del cliente para que los datos privados nunca tengan que salir de la máquina del usuario.

Para todos estos escenarios, puede utilizar [TensorFlow.js \(TFJS\)](#)

[Biblioteca de JavaScript](#). Esta biblioteca puede cargar un modelo TFLite y hacer predicciones directamente en el navegador del usuario. Por ejemplo, el siguiente módulo de JavaScript importa la biblioteca TFJS, descarga un modelo MobileNet previamente entrenado y utiliza este modelo para clasificar una imagen y registrar las predicciones.

Puedes jugar con el código en <https://homl.info/tfjscode>, usando Glitch.com, un sitio web que le permite

cree aplicaciones web en su navegador de forma gratuita; haga clic en el botón VISTA PREVIA en la esquina inferior derecha de la página para ver el código en acción:

```
importar
"https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest"; importar "https://cdn.jsdelivr.net/npm/
@tensorflow-models/mobilenet@1.0.0";

imagen constante = document.getElementById("imagen");

mobilenet.load().entonces(modelo => {
    modelo.classify(imagen).luego(predicciones => {
        for (var i = 0; i < predicciones.longitud; i++) { let className =
            predicciones[i].className let proba = (predicciones[i].probabilidad *
            100).toFixed(1)
            console.log(nombredeclase + " : " + probablemente + "%");
        }
    });
});
```

Incluso es posible convertir este sitio web en una aplicación web progresiva (PWA): se trata de un sitio web que respeta una serie de criterios que permiten visualizarlo en cualquier navegador e incluso instalarlo como una aplicación independiente en un dispositivo móvil. Por ejemplo, intente visitar <https://homl.info/tfjswpa> en un dispositivo móvil: la mayoría de los navegadores modernos le preguntarán si desea agregar TFJS Demo a su pantalla de inicio. Si aceptas, verás un nuevo ícono en tu lista de aplicaciones. Al hacer clic en este ícono, se cargará el sitio web de demostración de TFJS dentro de su propia ventana, como una aplicación móvil normal. Una PWA incluso se puede configurar para que funcione sin conexión mediante el uso de un trabajador de servicio: se trata de un módulo de JavaScript que se ejecuta en su propio hilo separado en el navegador e intercepta las solicitudes de red, lo que le permite almacenar en caché los recursos para que la PWA pueda ejecutarse más rápido, o incluso completamente fuera de línea. También puede enviar mensajes push, ejecutar tareas en segundo plano y más. Las PWA le permiten administrar una base de código única para la web y para dispositivos móviles. También facilitan la tarea de garantizar que todos los usuarios ejecuten

la misma versión de su aplicación. Puedes jugar con el código PWA de esta demostración TFJS en Glitch.com en <https://homl.info/wpacode>.

## CONSEJO

Vea muchas más demostraciones de modelos de aprendizaje automático que se ejecutan en su navegador en <https://tensorflow.org/js/demos>.

¡TFJS también admite el entrenamiento de un modelo directamente en su navegador web! Y en realidad es bastante rápido. Si su computadora tiene una tarjeta GPU, TFJS generalmente puede usarla, incluso si no es una tarjeta Nvidia. De hecho, TFJS usará WebGL cuando esté disponible y, dado que los navegadores web modernos generalmente admiten una amplia gama de tarjetas GPU, TFJS en realidad admite más tarjetas GPU que TensorFlow normal (que solo admite tarjetas Nvidia).

Entrenar un modelo en el navegador web de un usuario puede resultar especialmente útil para garantizar que los datos de este usuario permanezcan privados. Un modelo se puede entrenar de forma centralizada y luego ajustarlo localmente, en el navegador, en función de los datos de ese usuario. Si está interesado en este tema, consulte [el aprendizaje federado](#).

Una vez más, hacer justicia a este tema requeriría un libro completo. Si desea obtener más información sobre TensorFlow.js, consulte los libros de O'reilly [Practical Deep Learning for Cloud, Mobile y Edge](#). por Anirudh Koul et al., o [Learning TensorFlow.js](#), Por Gant Laborde.

Ahora que ha visto cómo implementar modelos de TensorFlow en TF Serving, en la nube con Vertex AI, en dispositivos móviles e integrados usando TFLite, o en un navegador web usando TFJS, analicemos cómo usar GPU para acelerar los cálculos. .

## Uso de GPU para acelerar los cálculos

En el Capítulo 11 analizamos varias técnicas que pueden acelerar considerablemente el entrenamiento: una mejor inicialización del peso, optimizadores sofisticados, etc. Pero incluso con todas estas técnicas, entrenar una gran red neuronal en una sola máquina con una sola CPU puede llevar horas, días o incluso semanas, según la tarea. Gracias a las GPU, este tiempo de entrenamiento se puede reducir a minutos u horas.

Esto no solo ahorra una enorme cantidad de tiempo, sino que también significa que puede experimentar con varios modelos mucho más fácilmente y volver a entrenar sus modelos con frecuencia con datos nuevos.

En los capítulos anteriores, utilizamos tiempos de ejecución habilitados para GPU en Google Colab. Todo lo que tiene que hacer para esto es seleccionar "Cambiar tipo de tiempo de ejecución" en el menú Tiempo de ejecución y elegir el tipo de acelerador de GPU; TensorFlow detecta automáticamente la GPU y la usa para acelerar los cálculos, y el código es exactamente el mismo que sin una GPU. Luego, en este capítulo vio cómo implementar sus modelos en Vertex AI en múltiples nodos de cómputo habilitados para GPU: solo es cuestión de seleccionar la imagen de Docker habilitada para GPU correcta al crear el modelo de Vertex AI y seleccionar el tipo de GPU deseado cuando llamando a `endpoint.deploy()`. Pero, ¿qué pasa si quieres comprar tu propia GPU?

¿Y qué sucede si desea distribuir los cálculos entre la CPU y varios dispositivos GPU en una sola máquina (consulte la [Figura 19-6](#))?

Esto es lo que discutiremos ahora y luego, más adelante en este capítulo, discutiremos cómo distribuir los cálculos entre múltiples servidores.

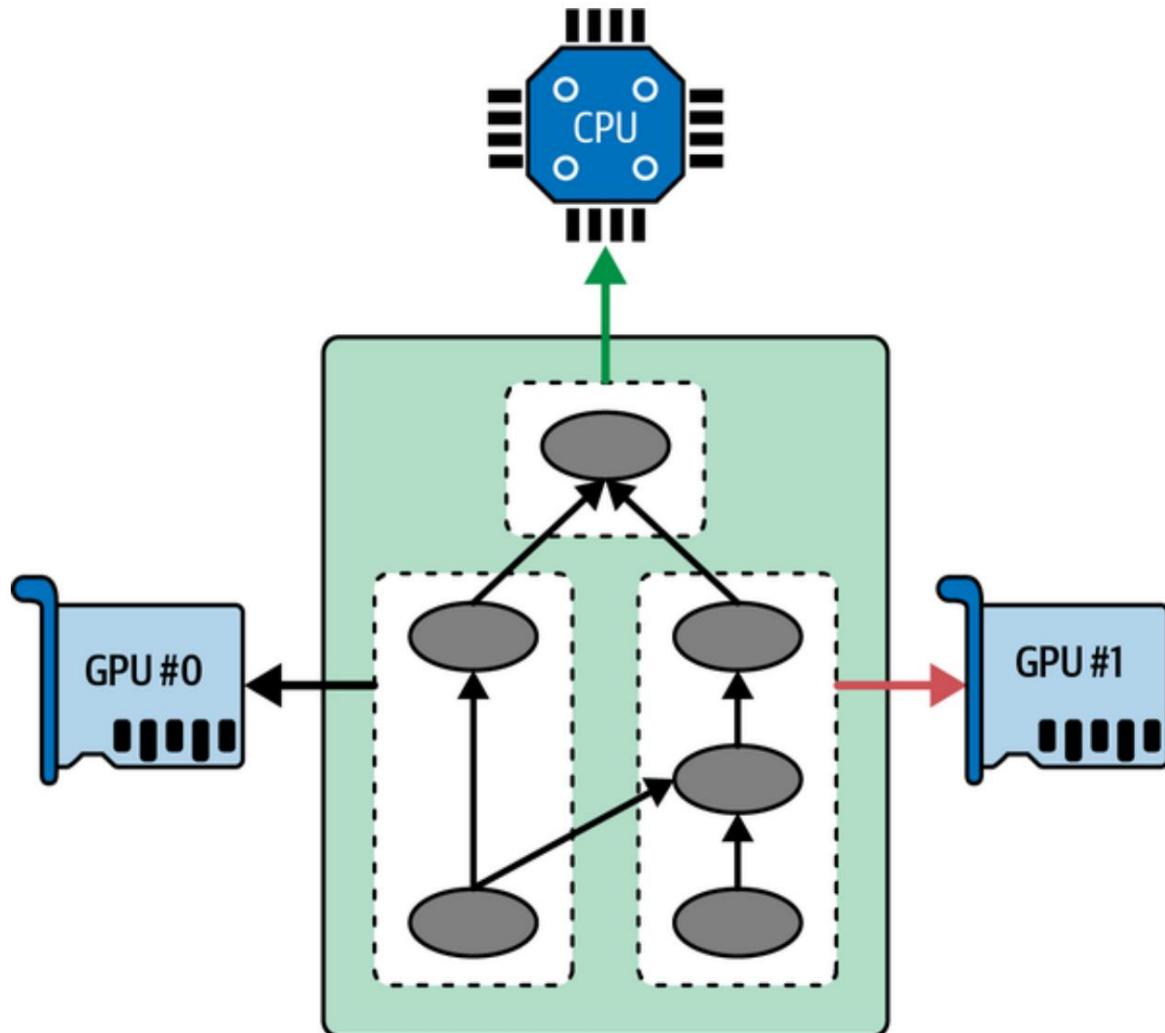


Figura 19-6. Ejecutar un gráfico de TensorFlow en varios dispositivos en paralelo

## Obtener su propia GPU

Si sabe que utilizará una GPU en gran medida y durante un largo período de tiempo, comprar una propia puede tener sentido financiero. Es posible que también desees entrenar tus modelos localmente porque no deseas cargar tus datos en la nube. O tal vez simplemente quieras comprar una tarjeta GPU para juegos y también te gustaría usarla para el aprendizaje profundo.

Si decide comprar una tarjeta GPU, tómese un tiempo para tomar la decisión correcta. Deberá considerar la cantidad de RAM que necesitará para sus tareas (por ejemplo, normalmente al menos 10 GB para imágenes).

procesamiento o PNL), el ancho de banda (es decir, qué tan rápido puede enviar datos dentro y fuera de la GPU), la cantidad de núcleos, el sistema de enfriamiento, etc. Tim Dettmers escribió una excelente publicación en el blog . Para ayudarte a elegir: te animo a que lo leas atentamente. Al momento de escribir este artículo, TensorFlow solo admite [tarjetas Nvidia con CUDA Compute Capability 3.5+](#) (así como las TPU de Google, por supuesto), pero puede extender su soporte a otros fabricantes, así que asegúrese de consultar la [documentación de TensorFlow](#). para ver qué dispositivos son compatibles hoy.

Si opta por una tarjeta GPU Nvidia, deberá instalar los controladores Nvidia adecuados y varias bibliotecas de Nvidia. Estos incluyen la biblioteca Compute Unified Device Architecture (CUDA). Toolkit, que permite a los desarrolladores utilizar GPU habilitadas para CUDA para todo tipo de cálculos (no solo aceleración de gráficos), y la biblioteca CUDA Deep Neural Network (cuDNN), una biblioteca acelerada por GPU de cálculos DNN comunes, como capas de activación, normalización, convoluciones hacia adelante y hacia atrás, y agrupación (consulte [el Capítulo 14](#)). cuDNN es parte del SDK de aprendizaje profundo de Nvidia. Tenga en cuenta que deberá crear una cuenta de desarrollador de Nvidia para poder descargarlo. TensorFlow usa CUDA y cuDNN para controlar las tarjetas GPU y acelerar los cálculos (consulte [la Figura 19-7](#)).

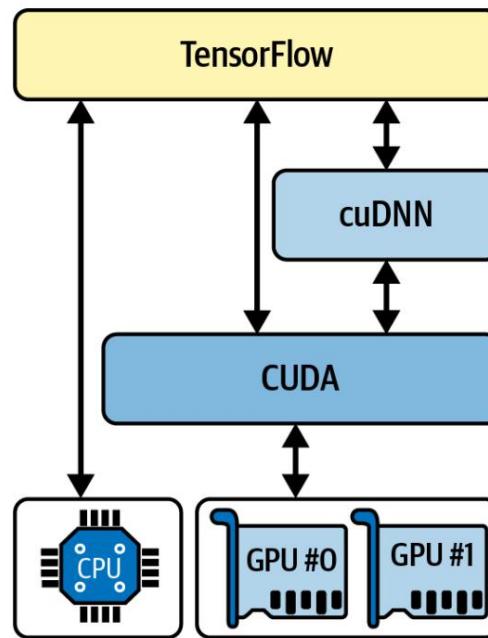


Figura 19-7. TensorFlow usa CUDA y cuDNN para controlar las GPU y aumentarlas DNN

Una vez que haya instalado las tarjetas GPU y todos los controladores necesarios y bibliotecas, puede utilizar el comando nvidia-smi para comprobar que todo está correctamente instalado. Este comando enumera los disponibles Tarjetas GPU, así como todos los procesos que se ejecutan en cada tarjeta. En esto Por ejemplo, es una tarjeta GPU Nvidia Tesla T4 con aproximadamente 15 GB de RAM disponible y no hay ningún proceso ejecutándose actualmente en ella:

```

$ nvidia-smi
Mon 10 abr 04:52:10 2022
+-----+
| NVIDIA-SMI 460.32.03 Versión:      | Versión del controlador: 460.32.03 | CUDA
| 11.2          |                               |           |
+-----+-----+-----+
| Nombre de GPU      | Persistencia-M| ID de autobús | Disp.A |
| volátil no corregido. CEC |             |          |          |
| Fan Temp Perf Pwr:Uso/Cap| GPU-Util Compute M. | Uso de memoria |
| MIG M. |          |          |          |
  
```

```

=====
|=====+=====+=====+=====+
|=====+=====+=====+=====+
| 0 Tesla T4 0 | Apagado | 00000000:00:04.0 Apagado |
| N/D 34C 0% P8 9W / 70W | 3MiB / 15109MiB |
| Predeterminado | | |
| N/A | | |
+-----+-----+---+
-----+
-----+
| Procesos:
| GPU GI CI Tipo PID Nombre del proceso
Memoria GPU | HICE
| Uso |
+-----+-----+-----+-----+
| No se encontraron procesos en ejecución
| |
+-----+-----+
-----+

```

Para comprobar que TensorFlow realmente ve su GPU, ejecute lo siguiente comandos y asegúrese de que el resultado no esté vacío:

```

>>> físico_gpus = tf.config.list_physical_devices("GPU")
>>> físico_gpus
[Dispositivo físico(nombre='/dispositivo_físico:GPU:0',
tipo_dispositivo='GPU')]

```

## Administrar la RAM de la GPU

Por defecto, TensorFlow toma automáticamente casi toda la RAM en todos GPU disponibles la primera vez que ejecuta un cálculo. Hace esto para limitar la fragmentación de la RAM de la GPU. Esto significa que si intenta iniciar un segundo programa TensorFlow (o cualquier programa que requiera el

GPU), rápidamente se quedará sin RAM. Esto no sucede con tanta frecuencia como podría pensar, ya que la mayoría de las veces tendrá un único programa TensorFlow ejecutándose en una máquina: generalmente un script de entrenamiento, un nodo TF Serving o un cuaderno Jupyter. Si necesita ejecutar varios programas por algún motivo (por ejemplo, para entrenar dos modelos diferentes en paralelo en la misma máquina), deberá dividir la RAM de la GPU entre estos procesos de manera más uniforme.

Si tiene varias tarjetas GPU en su máquina, una solución sencilla es asignar cada una de ellas a un único proceso. Para hacer esto, puede configurar la variable de entorno CUDA\_VISIBLE\_DEVICES para que cada proceso solo vea las tarjetas GPU apropiadas. También configure la variable de entorno CUDA\_DEVICE\_ORDER en PCI\_BUS\_ID para garantizar que cada ID siempre haga referencia a la misma tarjeta GPU. Por ejemplo, si tiene cuatro tarjetas GPU, puede iniciar dos programas, asignando dos GPU a cada uno de ellos, ejecutando comandos como los siguientes en dos ventanas de terminal separadas:

```
$ CUDA_DEVICE_ORDER=PCI_BUS_ID CUDA_VISIBLE_DEVICES=0,1
python3 program_1.py # y
en otra terminal: $
CUDA_DEVICE_ORDER=PCI_BUS_ID CUDA_VISIBLE_DEVICES=3,2
python3 program_2.py
```

Entonces, el programa 1 solo verá las tarjetas GPU 0 y 1, denominadas "/gpu:0" y "/gpu:1", respectivamente, en TensorFlow, y el programa 2 solo verá las tarjetas GPU 2 y 3, denominadas "/gpu:1". " y "/gpu:0", respectivamente (tenga en cuenta el orden). Todo funcionará bien (ver [Figura 19-8](#)).

Por supuesto, también puedes definir estas variables de entorno en Python configurando

```
os.environ["CUDA_DEVICE_ORDER"] y
os.environ["CUDA_VISIBLE_DEVICES"], siempre que lo hagas antes de usar
TensorFlow.
```

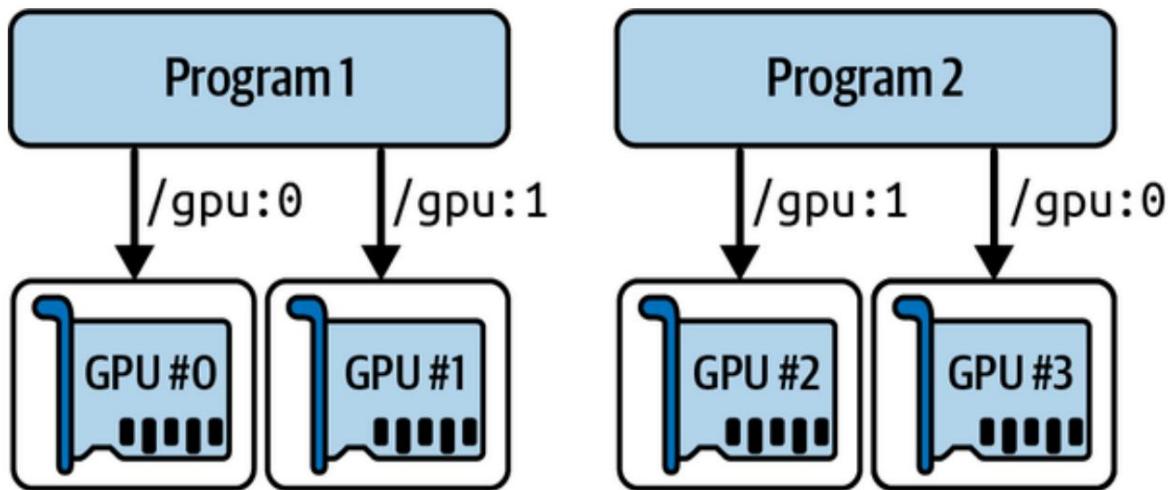


Figura 19-8. Cada programa obtiene dos GPU

Otra opción es decirle a TensorFlow que tome solo una cantidad específica de RAM de GPU. Esto debe hacerse inmediatamente después de importar TensorFlow. Por ejemplo, para hacer que TensorFlow tome solo 2 GiB de RAM en cada GPU, debe crear un dispositivo GPU lógico (a veces llamado dispositivo GPU virtual) para cada dispositivo GPU físico y establecer su límite de memoria en 2 GiB (es decir, 2048 MiB). :

```

para gpu en Physical_gpus:
    tf.config.set_logical_device_configuration(
        GPU,
        [tf.config.LogicalDeviceConfiguration(memory_limit=2048)])
    )

```

Supongamos que tiene cuatro GPU, cada una con al menos 4 GiB de RAM: en este caso, dos programas como este pueden ejecutarse en paralelo, cada uno usando las cuatro tarjetas GPU (consulte la Figura 19-9) . Si ejecuta el comando nvidia-smi mientras ambos programas se están ejecutando, debería ver que cada proceso tiene 2 GiB de RAM en cada tarjeta.

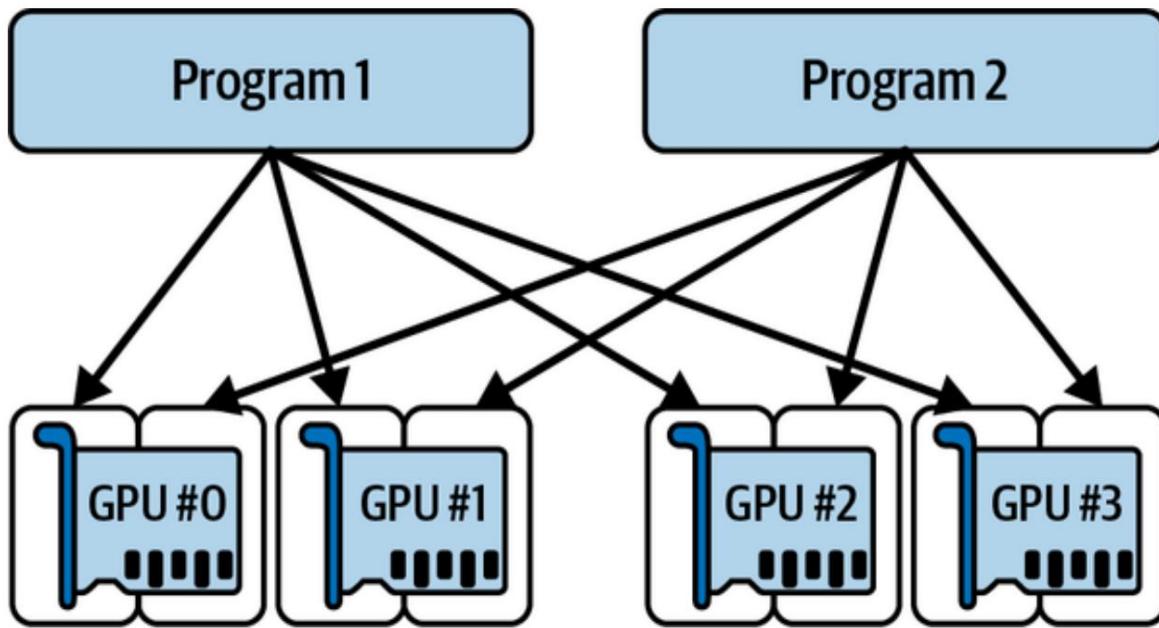


Figura 19-9. Cada programa obtiene las cuatro GPU, pero con solo 2 GiB de RAM en cada GPU

Otra opción más es decirle a TensorFlow que tome memoria solo cuando la necesite. Nuevamente, esto debe hacerse inmediatamente después de importar TensorFlow:

para gpu en `físico_gpus`:

```
tf.config.experimental.set_memory_growth(gpu, Verdadero)
```

Otra forma de hacer esto es establecer la variable de entorno `TF_FORCE_GPU_ALLOW_GROWTH` en verdadero. Con esta opción, TensorFlow nunca liberará memoria una vez que la haya capturado (nuevamente, para evitar la fragmentación de la memoria), excepto, por supuesto, cuando finalice el programa. Puede ser más difícil garantizar un comportamiento determinista usando esta opción (por ejemplo, un programa puede fallar porque el uso de memoria de otro programa se disparó), por lo que en producción probablemente querrás seguir con una de las opciones anteriores. Sin embargo, hay algunos casos en los que es muy útil: por ejemplo, cuando usas una máquina para ejecutar múltiples notebooks Jupyter, varios de los cuales usan TensorFlow. El

La variable de entorno TF\_FORCE\_GPU\_ALLOW\_GROWTH está configurada en verdadero en los tiempos de ejecución de Colab.

Por último, en algunos casos es posible que desees dividir una GPU en dos o más dispositivos lógicos. Por ejemplo, esto es útil si solo tiene una GPU física (como en un tiempo de ejecución de Colab) pero desea probar un algoritmo de múltiples GPU. El siguiente código divide la GPU #0 en dos dispositivos lógicos, con 2 GiB de RAM cada uno (nuevamente, esto debe hacerse inmediatamente después de importar TensorFlow):

```
tf.config.set_logical_device_configuration( físico_gpus[0],  
  
[tf.config.LogicalDeviceConfiguration(memory_limit=2048),  
  
tf.config.LogicalDeviceConfiguration(memory_limit=2048)] )
```

Estos dos dispositivos lógicos se llaman "/gpu:0" y "/gpu:1", y puedes usarlos como si fueran dos GPU normales. Puede enumerar todos los dispositivos lógicos de esta manera:

```
>>> logic_gpus = tf.config.list_logical_devices("GPU") >>> logic_gpus  
[Dispositivo lógico(nombre='/dispositivo:GPU:0', tipo_dispositivo='GPU'),  
 Dispositivo lógico(nombre='/dispositivo:GPU: 1', tipo_dispositivo='GPU')]
```

Ahora veamos cómo TensorFlow decide qué dispositivos debe usar para colocar variables y ejecutar operaciones.

Colocar operaciones y variables en dispositivos Keras y tf.data generalmente hacen un buen trabajo al colocar operaciones y variables donde pertenecen, pero también puedes colocar operaciones y variables manualmente en cada dispositivo, si quieres tener más control:

- Por lo general, desea colocar las operaciones de preprocessamiento de datos en la CPU y las operaciones de la red neuronal en las GPU.
- Las GPU suelen tener un ancho de banda de comunicación bastante limitado, por lo que es importante evitar transferencias de datos innecesarias dentro y fuera de las GPU.
- Agregar más RAM de CPU a una máquina es simple y bastante económico, por lo que generalmente hay suficiente, mientras que la RAM de la GPU está integrada en la GPU: es un recurso costoso y, por lo tanto, limitado, por lo que si no se necesita una variable en los próximos pasos de entrenamiento, probablemente debería colocarse en la CPU (por ejemplo, los conjuntos de datos generalmente pertenecen a la CPU).

De forma predeterminada, todas las variables y todas las operaciones se colocarán en la primera GPU (la que se llama "/gpu:0"), excepto las variables y operaciones que no tienen un kernel de GPU: estas se colocan en la CPU (siempre llamada "/CPU:0"). Un dispositivo de tensor o variable.<sup>10</sup>

El atributo le indica en qué dispositivo se colocó:<sup>11</sup>

```
>>> a = tf.Variable([1., 2., 3.]) # la variable float32 va a la GPU
>>> a.device '/'
job:localhost/replica:0/task:0/device:GPU:0' >>> b = tf.Variable([1,
2, 3]) # la variable int32 va a la CPU
>>> b.dispositivo
'/trabajo:localhost/réplica:0/tarea:0/dispositivo:CPU:0'
```

Puede ignorar con seguridad el prefijo / job:localhost/replica:0/task:0 por ahora; Analizaremos trabajos, réplicas y tareas más adelante en este capítulo. Como puede ver, la primera variable se colocó en la GPU n.º 0, que es el dispositivo predeterminado.

Sin embargo, la segunda variable se colocó en la CPU: esto se debe a que no hay núcleos de GPU para variables enteras o para

operaciones que involucran tensores enteros, por lo que TensorFlow recurrió a la CPU.

Si desea realizar una operación en un dispositivo diferente al predeterminado, use un contexto `tf.device()`:

```
>>> con tf.device("/cpu:0"): c =
...     tf.Variable([1., 2., 3.])
...
>>> c.dispositivo
'/trabajo:localhost/réplica:0/tarea:0/dispositivo:CPU:0'
```

### NOTA

La CPU siempre se trata como un único dispositivo ("/cpu:0"), incluso si su máquina tiene varios núcleos de CPU. Cualquier operación realizada en la CPU puede ejecutarse en paralelo en varios núcleos si tiene un núcleo multiproceso.

Si intenta explícitamente colocar una operación o variable en un dispositivo que no existe o para el cual no hay un kernel, TensorFlow recurrirá silenciosamente al dispositivo que habría elegido de forma predeterminada. Esto es útil cuando desea poder ejecutar el mismo código en diferentes máquinas que no tienen la misma cantidad de GPU. Sin embargo, puede ejecutar `tf.config.set_soft_device_placement(False)` si prefiere obtener una excepción.

Ahora bien, ¿cómo ejecuta exactamente TensorFlow las operaciones en múltiples dispositivos?

## Ejecución paralela en múltiples dispositivos

Como vimos en [el Capítulo 12](#), uno de los beneficios de usar funciones TF es el paralelismo. Miremos esto un poco más de cerca. Cuando TensorFlow ejecuta una función TF, comienza analizando su gráfico para encontrar la lista de operaciones que deben evaluarse y cuenta cuántas dependencias tiene cada una de ellas. TensorFlow luego agrega cada

operación con cero dependencias (es decir, cada operación de origen) a la cola de evaluación del dispositivo de esta operación (consulte [la Figura 19-10](#)). Una vez evaluada una operación, se decrementa el contador de dependencia de cada operación que depende de ella. Una vez que el contador de dependencia de una operación llega a cero, se envía a la cola de evaluación de su dispositivo. Y una vez que se han calculado todos los resultados, se devuelven.

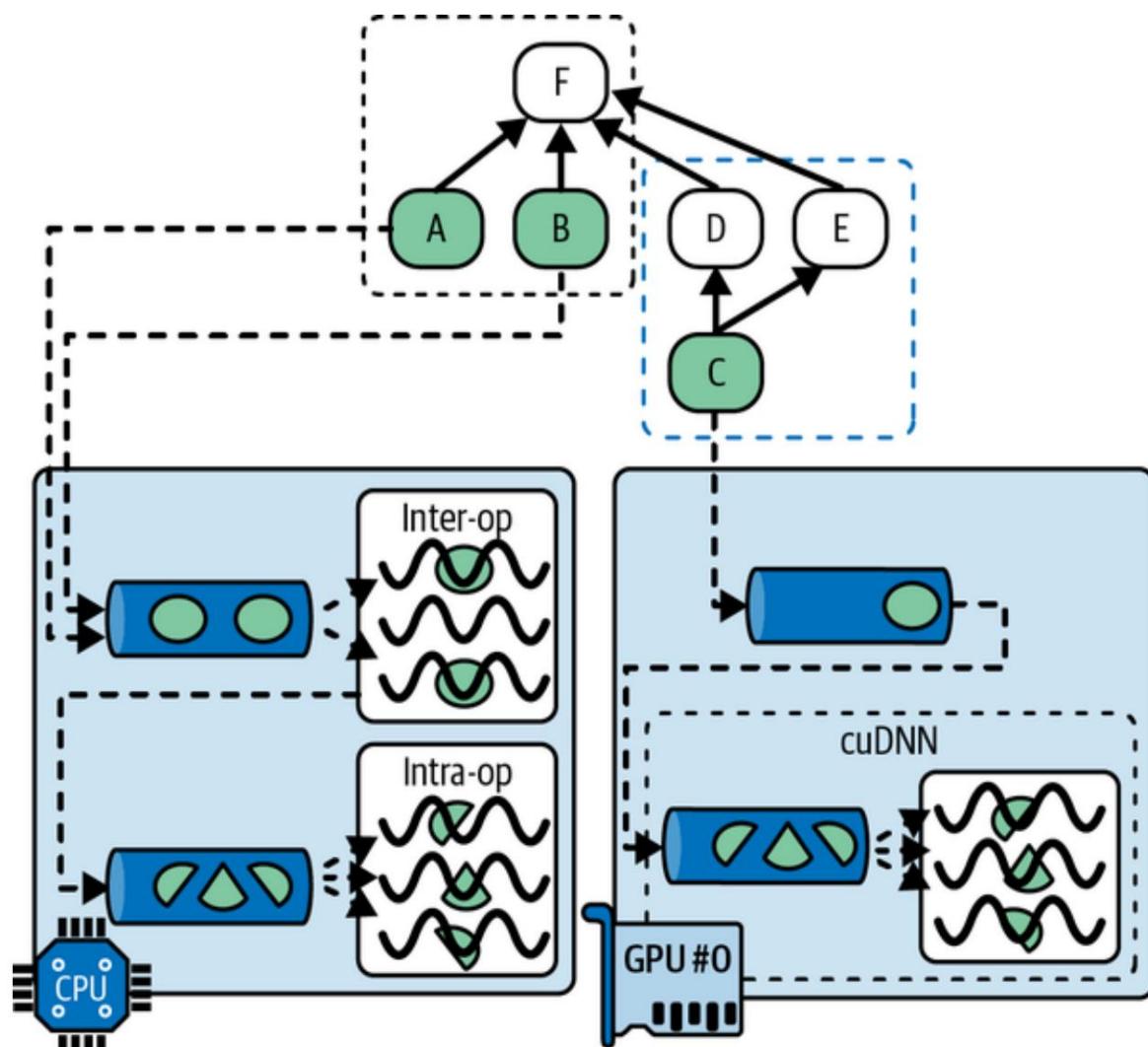


Figura 19-10. Ejecución paralela de un gráfico TensorFlow

Las operaciones en la cola de evaluación de la CPU se envían a un grupo de subprocessos llamado grupo de subprocessos de interoperabilidad. Si la CPU tiene varios núcleos, estas operaciones se evaluarán efectivamente en paralelo. Algunas operaciones tienen núcleos de CPU multiproceso: estos núcleos dividen sus

tareas en múltiples suboperaciones, que se colocan en otra cola de evaluación y se envían a un segundo grupo de subprocessos llamado grupo de subprocessos intraoperativos (compartido por todos los núcleos de CPU multiproceso). En resumen, se pueden evaluar múltiples operaciones y suboperaciones en paralelo en diferentes núcleos de CPU.

Para la GPU, las cosas son un poco más sencillas. Las operaciones en la cola de evaluación de una GPU se evalúan secuencialmente. Sin embargo, la mayoría de las operaciones tienen núcleos de GPU multiproceso, generalmente implementados por bibliotecas de las que depende TensorFlow, como CUDA y cuDNN. Estas implementaciones tienen sus propios grupos de subprocessos y, por lo general, explotan tantos subprocessos de GPU como pueden (razón por la cual no hay necesidad de un grupo de subprocessos interoperativos en las GPU: cada operación ya inunda la mayoría de los subprocessos de GPU).

Por ejemplo, en [la Figura 19-10](#), las operaciones A, B y C son operaciones de origen, por lo que pueden evaluarse inmediatamente. Las operaciones A y B se colocan en la CPU, por lo que se envían a la cola de evaluación de la CPU, luego se envían al grupo de subprocessos de interoperabilidad y se evalúan inmediatamente en paralelo. Resulta que la operación A tiene un núcleo multiproceso; sus cálculos se dividen en tres partes, que el grupo de subprocessos intraoperatorios ejecuta en paralelo. La operación C va a la cola de evaluación de la GPU n.º 0 y, en este ejemplo, su núcleo de GPU usa cuDNN, que administra su propio grupo de subprocessos intraoperativos y ejecuta la operación en muchos subprocessos de GPU en paralelo. Supongamos que C termina primero. Los contadores de dependencia de D y E disminuyen y llegan a 0, por lo que ambas operaciones se envían a la cola de evaluación de la GPU n.º 0 y se ejecutan secuencialmente. Tenga en cuenta que C sólo se evalúa una vez, aunque tanto D como E dependen de ello.

Supongamos que B termina a continuación. Luego, el contador de dependencia de F se reduce de 4 a 3 y, como no es 0, no se ejecuta todavía. Una vez que A, D y E terminan, el contador de dependencia de F llega a 0, se envía a la cola de evaluación de la CPU y se evalúa. Finalmente, TensorFlow devuelve las salidas solicitadas.

Un poco de magia adicional que realiza TensorFlow es cuando la función TF modifica un recurso con estado, como una variable: garantiza que el orden de ejecución coincide con el orden en el código, incluso si no existe una dependencia explícita entre las declaraciones. Por ejemplo, si su función TF contiene `v.assign_add(1)` seguido de `v.assign(v * 2)`, TensorFlow se asegurará de que estas operaciones se ejecuten en ese orden.

## CONSEJO

Puede controlar la cantidad de subprocessos en el grupo de subprocessos de interoperabilidad llamando a `tf.config.threading.set_inter_op_parallelism_threads()`. Para establecer el número de subprocessos intraoperativos, utilice `tf.config.threading.set_intra_op_parallelism_threads()`. Esto es útil si no desea que TensorFlow use todos los núcleos de la CPU o si desea que sea de un solo subprocesso.<sup>12</sup>

¡Con eso, tienes todo lo que necesitas para ejecutar cualquier operación en cualquier dispositivo y explotar el poder de tus GPU! Estas son algunas de las cosas que podrías hacer:

- Podrías entrenar varios modelos en paralelo, cada uno en su propia GPU: simplemente escribe un script de entrenamiento para cada modelo y ejecútalo en paralelo, configurando CUDA\_DEVICE\_ORDER y CUDA\_VISIBLE\_DEVICES para que cada script solo vea un único dispositivo GPU. Esto es excelente para el ajuste de hiperparámetros, ya que puede entrenar en paralelo varios modelos con diferentes hiperparámetros. Si tiene una sola máquina con dos GPU y se necesita una hora para entrenar un modelo en una GPU, entonces entrenar dos modelos en paralelo, cada uno en su propia GPU dedicada, tomará solo una hora. ¡Simple!
- Podrías entrenar un modelo en una sola GPU y realizar todo el preprocessamiento en paralelo en la CPU, utilizando el conjunto de datos.

método `prefetch()` para preparar los siguientes lotes con anticipación para que estén listos cuando la GPU los necesite (consulte el [Capítulo 13](#)).

- Si su modelo toma dos imágenes como entrada y las procesa usando dos CNN antes de unir sus salidas, entonces probablemente se ejecutará mucho más rápido si coloca cada CNN en una GPU diferente.
- Puede crear un conjunto eficiente: simplemente coloque un modelo entrenado diferente en cada GPU para que pueda obtener todas las predicciones mucho más rápido para producir la predicción final del conjunto.

Pero, ¿qué sucede si desea acelerar el entrenamiento mediante el uso de varias GPU?

## Modelos de entrenamiento en múltiples dispositivos

Hay dos enfoques principales para entrenar un solo modelo en múltiples dispositivos: paralelismo de modelos, donde el modelo se divide entre los dispositivos, y paralelismo de datos, donde el modelo se replica en cada dispositivo y cada réplica se entrena en un subconjunto diferente del datos. Veamos estas dos opciones.

### Paralelismo modelo

Hasta ahora hemos entrenado cada red neuronal en un solo dispositivo. ¿Qué pasa si queremos entrenar una única red neuronal en varios dispositivos?

Esto requiere dividir el modelo en fragmentos separados y ejecutar cada fragmento en un dispositivo diferente. Desafortunadamente, este paralelismo de modelos resulta bastante complicado y su efectividad realmente depende de la arquitectura de su red neuronal. Para redes totalmente conectadas, generalmente no se gana mucho con este enfoque (consulte la [Figura 19-11](#)). Intuitivamente, puede parecer que una manera fácil de dividir el modelo es colocar cada capa en un lugar diferente.

dispositivo, pero esto no funciona porque cada capa necesita esperar la salida de la capa anterior antes de poder hacer algo. Entonces, ¿tal vez puedes dividirlo verticalmente, por ejemplo, con la mitad izquierda de cada capa en un dispositivo y la parte derecha en otro dispositivo? Esto es ligeramente mejor, ya que ambas mitades de cada capa pueden funcionar en paralelo, pero el problema es que cada mitad de la siguiente capa requiere la salida de ambas mitades, por lo que habrá mucha comunicación entre dispositivos (representada por el flechas discontinuas). Es probable que esto anule por completo el beneficio del cálculo paralelo, ya que la comunicación entre dispositivos es lenta (y más aún cuando los dispositivos están ubicados en máquinas diferentes).

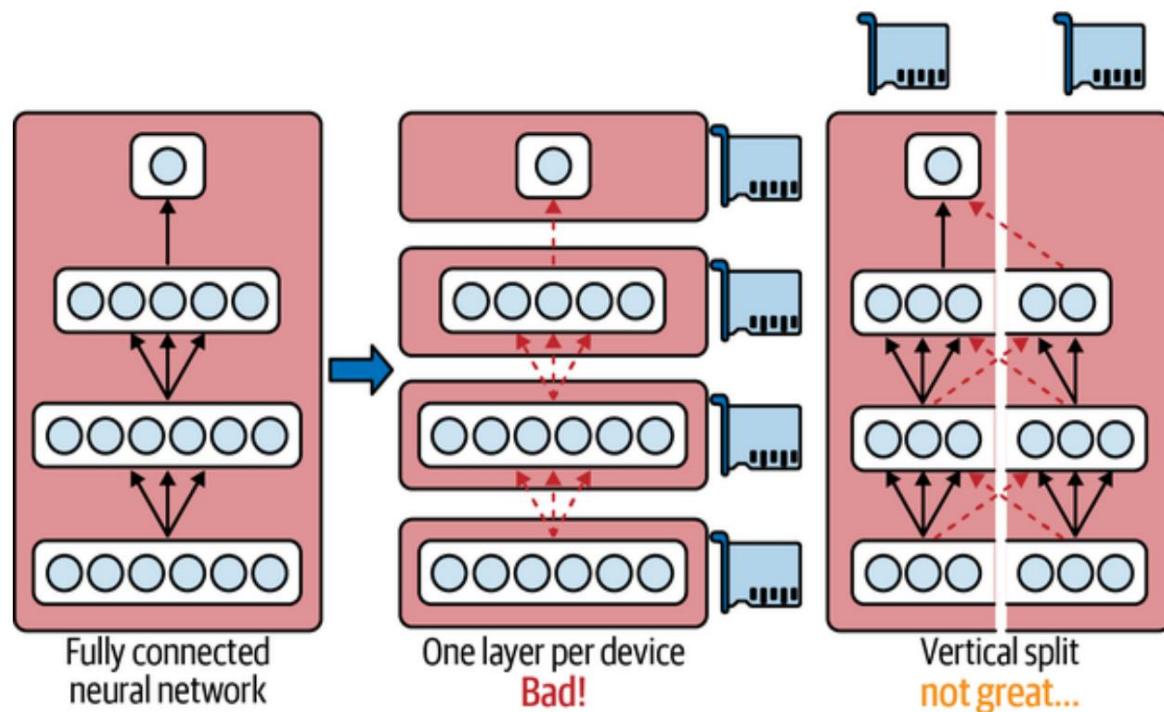


Figura 19-11. Dividiendo una red neuronal completamente conectada

Algunas arquitecturas de redes neuronales, como las redes neuronales convolucionales (consulte [el Capítulo 14](#)), contienen capas que están sólo parcialmente conectadas a las capas inferiores, por lo que es mucho más fácil distribuir fragmentos entre dispositivos de manera eficiente ([Figura 19-12](#)).

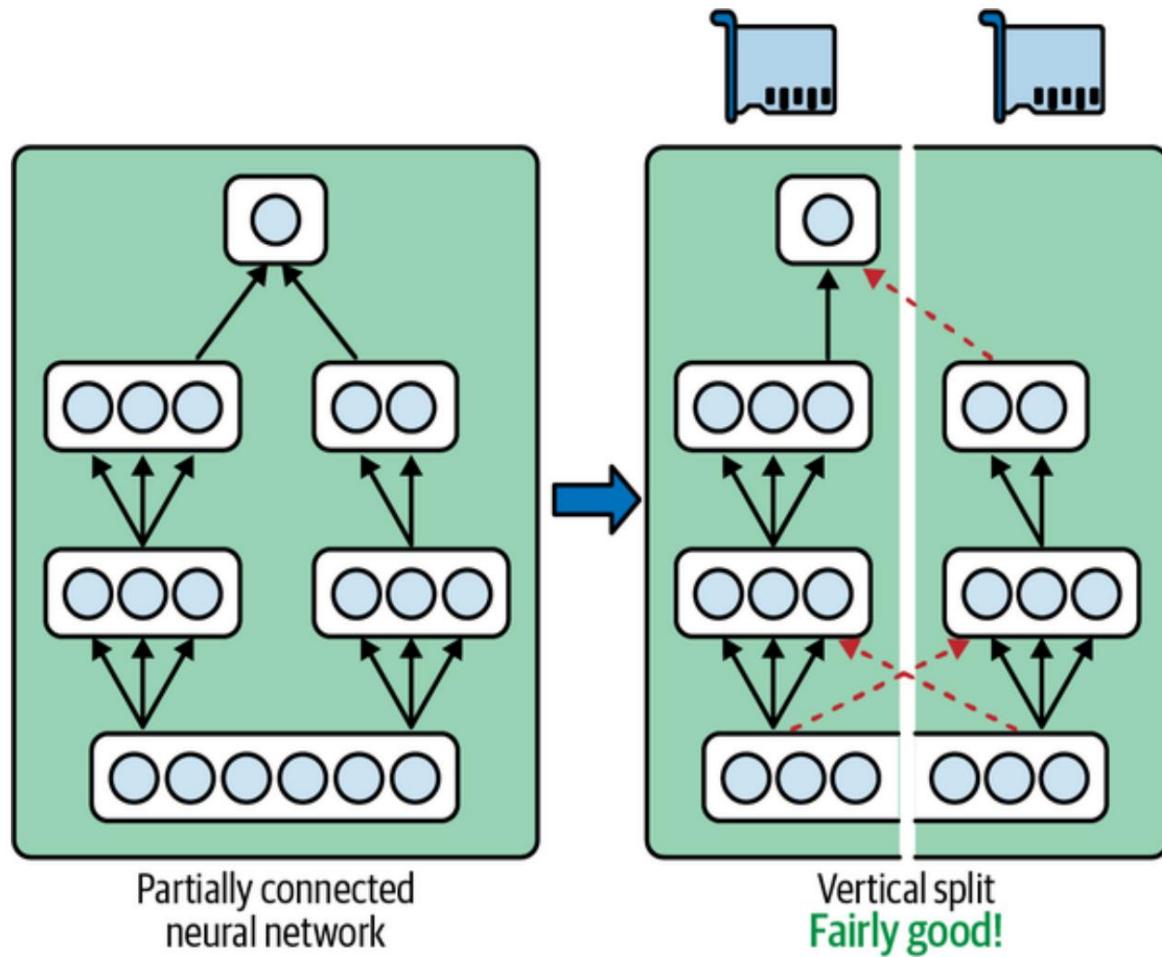


Figura 19-12. Dividir una red neuronal parcialmente conectada

Las redes neuronales recurrentes profundas (consulte [el Capítulo 15](#)) se pueden dividir de manera un poco más eficiente en múltiples GPU.

Si divide la red horizontalmente colocando cada capa en un dispositivo diferente y alimenta la red con una secuencia de entrada para procesar, entonces en el primer paso solo estará activo un dispositivo (trabajando en el primer valor de la secuencia), en el segundo El paso dos estará activo (la segunda capa manejará la salida de la primera capa para el primer valor, mientras que la primera capa manejará el segundo valor), y cuando la señal se propague a la capa de salida, todos los dispositivos estarán activos simultáneamente ([Figura 19-13](#)). Todavía hay mucha comunicación entre dispositivos, pero dado que cada celda puede ser bastante compleja, el beneficio de ejecutar varias celdas en paralelo puede (en teoría) superar la penalización de la comunicación. Sin embargo, en la p

La pila de capas LSTM que se ejecutan en una sola GPU en realidad se ejecuta mucho más rápido.

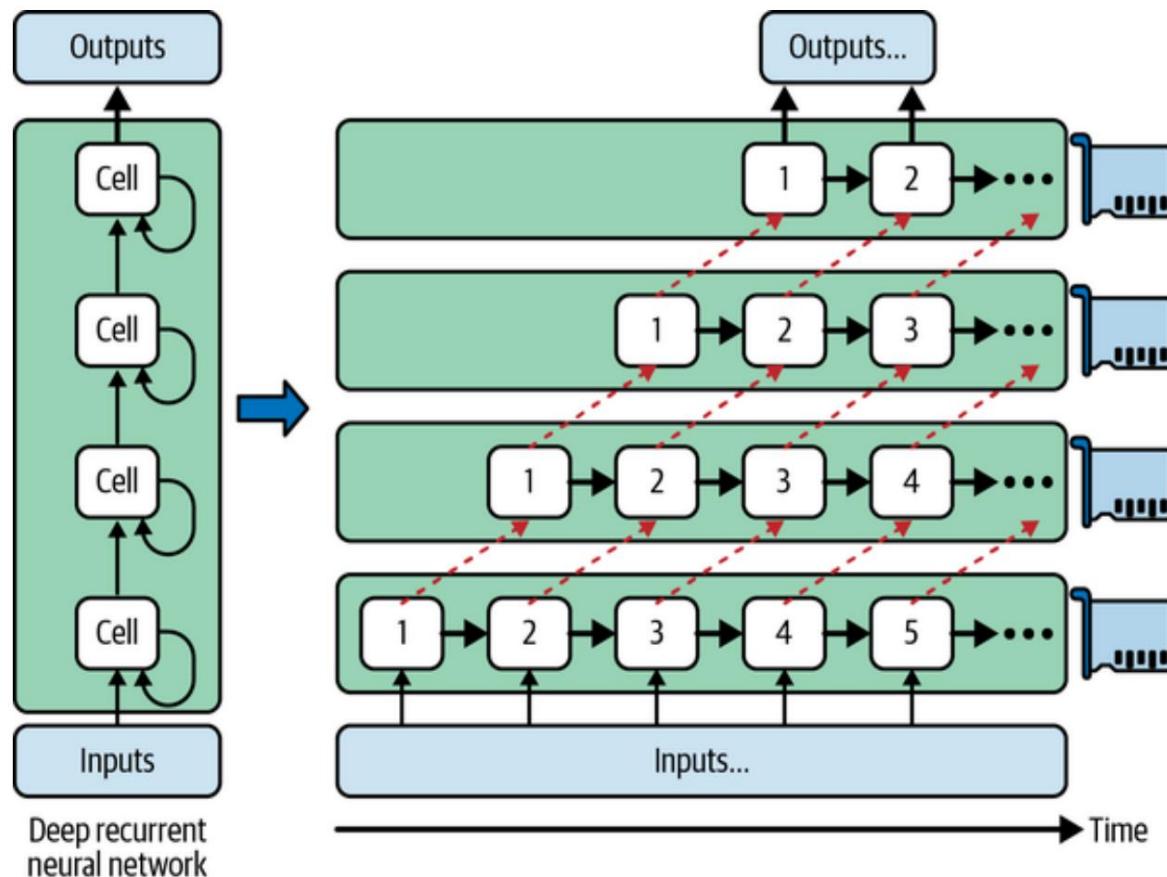


Figura 19-13. Dividiendo una red neuronal recurrente profunda

En resumen, el paralelismo de modelos puede acelerar la ejecución o el entrenamiento de algunos tipos de redes neuronales, pero no de todas, y requiere cuidados y ajustes especiales, como asegurarse de que los dispositivos que necesitan comunicarse al máximo se ejecuten en la misma máquina. A continuación veremos una<sup>15</sup> opción mucho más simple y generalmente más eficiente: el paralelismo de datos.

## Paralelismo de datos

Otra forma de paralelizar el entrenamiento de una red neuronal es replicarla en cada dispositivo y ejecutar cada paso de entrenamiento simultáneamente en todas las réplicas, utilizando un mini lote diferente para cada una. Luego se promedian los gradientes calculados por cada réplica y se

El resultado se utiliza para actualizar los parámetros del modelo. Esto se denomina paralelismo de datos o, a veces, programa único, datos múltiples (SPMD). Hay muchas variantes de esta idea, así que veamos las más importantes.

#### Paralelismo de datos utilizando la estrategia reflejada

Podría decirse que el enfoque más simple es reflejar completamente todos los parámetros del modelo en todas las GPU y aplicar siempre exactamente las mismas actualizaciones de parámetros en cada GPU. De este modo, todas las réplicas son siempre perfectamente idénticas. Esto se llama estrategia reflejada y resulta bastante eficiente, especialmente cuando se utiliza una sola máquina (consulte [la Figura 19-14](#)).

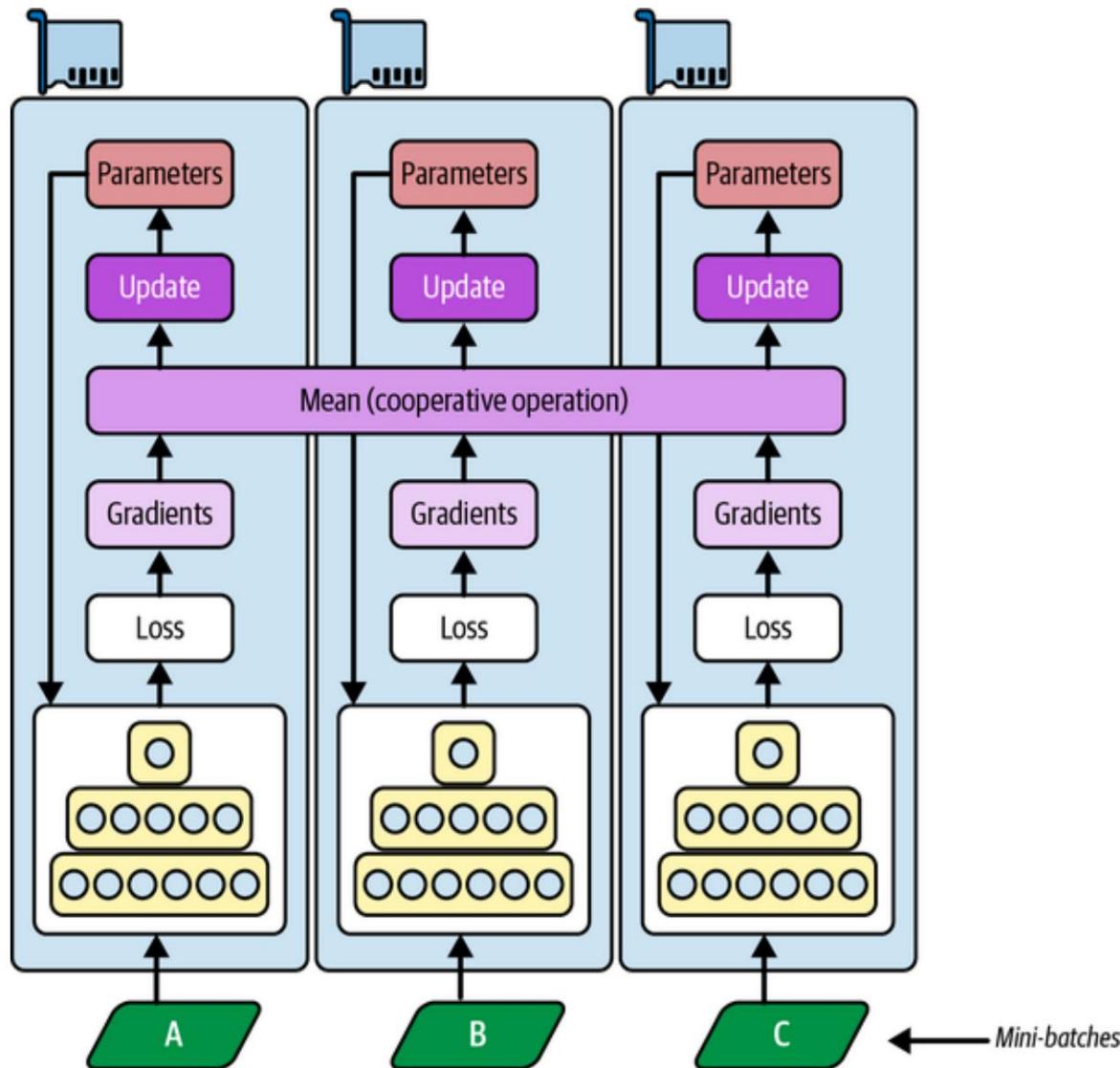


Figura 19-14. Paralelismo de datos utilizando la estrategia reflejada.

La parte complicada al utilizar este enfoque es calcular de manera eficiente la media de todos los gradientes de todas las GPU y distribuir el resultado entre todas las GPU. Esto se puede hacer utilizando un algoritmo AllReduce , una clase de algoritmos en los que varios nodos colaboran para realizar de manera eficiente una operación de reducción (como calcular la media, la suma y el máximo), al tiempo que se garantiza que todos los nodos obtengan el mismo resultado final. Afortunadamente, existen implementaciones estándar de dichos algoritmos, como verá.

Paralelismo de datos con parámetros centralizados.

Otro enfoque es almacenar los parámetros del modelo fuera de los dispositivos GPU que realizan los cálculos (llamados trabajadores); por ejemplo, en la CPU (consulte [la Figura 19-15](#)). En una configuración distribuida, puede colocar todos los parámetros en uno o más servidores de CPU llamados servidores de parámetros, cuya única función es alojar y actualizar los parámetros.

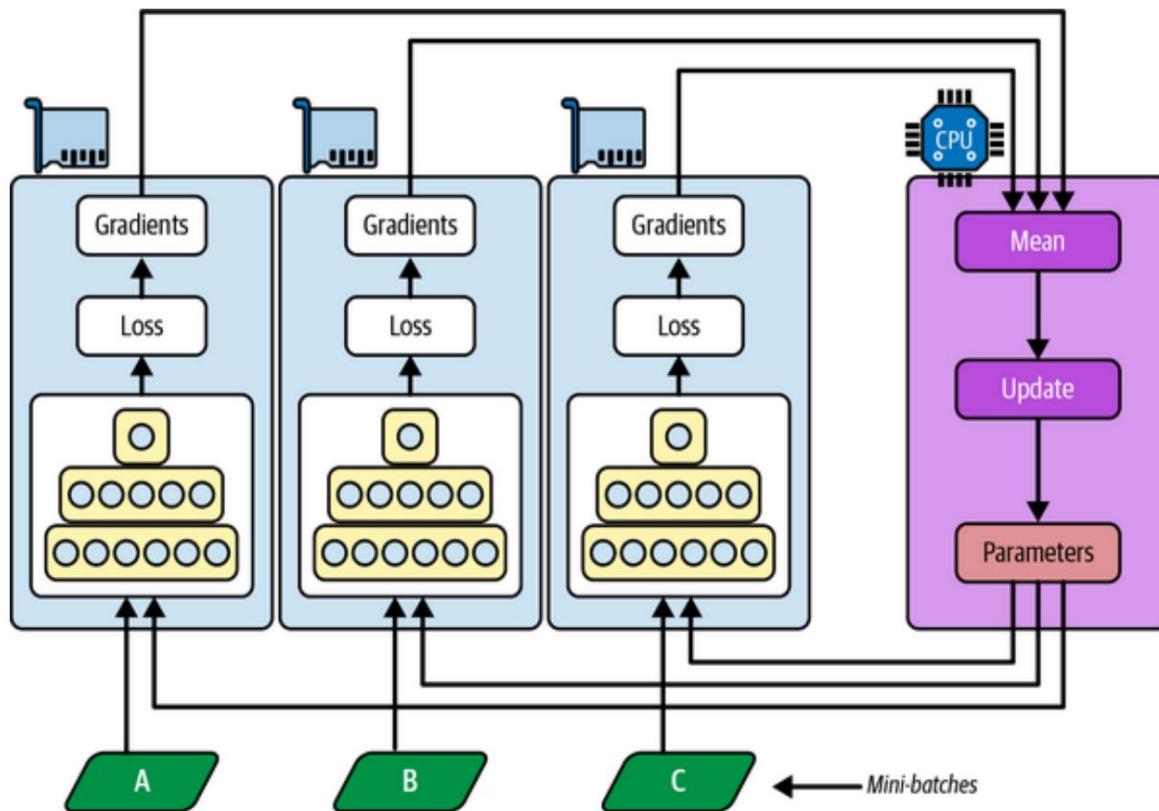


Figura 19-15. Paralelismo de datos con parámetros centralizados.

Mientras que la estrategia reflejada impone actualizaciones de peso sincrónicas en todas las GPU, este enfoque centralizado permite actualizaciones sincrónicas o asincrónicas. Echemos un vistazo a los pros y los contras de ambas opciones.

#### Actualizaciones sincrónicas

Con las actualizaciones sincrónicas, el agregador espera hasta que todos los gradientes estén disponibles antes de calcular los gradientes promedio y pasarlo al optimizador, que actualizará los parámetros del modelo. Una vez que una réplica ha terminado de calcular sus gradientes, debe esperar a que

Los parámetros se actualizarán antes de poder pasar al siguiente mini lote.

La desventaja es que algunos dispositivos pueden ser más lentos que otros, por lo que los dispositivos rápidos tendrán que esperar a los lentos en cada paso, haciendo que todo el proceso sea tan lento como el dispositivo más lento. Además, los parámetros se copiarán en todos los dispositivos casi al mismo tiempo (inmediatamente después de aplicar los gradientes), lo que puede saturar el ancho de banda de los servidores de parámetros.

CONSEJO

Para reducir el tiempo de espera en cada paso, puede ignorar los gradientes de las pocas réplicas más lentas (normalmente ~10%). Por ejemplo, podría ejecutar 20 réplicas, pero solo agregar los gradientes de las 18 réplicas más rápidas en cada paso e ignorar los gradientes de las últimas 2. Tan pronto como se actualicen los parámetros, las primeras 18 réplicas pueden comenzar a funcionar nuevamente inmediatamente., sin tener que esperar a las 2 réplicas más lentas. Generalmente se describe que esta configuración tiene 18 réplicas más 2 réplicas de repuesto.

## Actualizaciones asincrónicas

Con las actualizaciones asincrónicas, cada vez que una réplica ha terminado de calcular los gradientes, los gradientes se utilizan inmediatamente para actualizar los parámetros del modelo. No hay agregación (elimina el paso "medio" en [la Figura 19-15](#)) ni sincronización. Las réplicas funcionan independientemente de las otras réplicas. Como no hay que esperar a las otras réplicas, este enfoque ejecuta más pasos de entrenamiento por minuto. Además, aunque todavía es necesario copiar los parámetros a cada dispositivo en cada paso, esto sucede en diferentes momentos para cada réplica, por lo que se reduce el riesgo de saturación del ancho de banda.

El paralelismo de datos con actualizaciones asincrónicas es una opción atractiva debido a su simplicidad, la ausencia de retraso en la sincronización y su mejor uso del ancho de banda. Sin embargo, aunque funciona razonablemente bien en la práctica, ¡es casi sorprendente que funcione!

De hecho, cuando una réplica haya terminado de calcular los gradientes en función de algunos valores de parámetros, estos parámetros habrán sido actualizados varias veces por otras réplicas (en promedio  $N - 1$  veces, si hay  $N$  réplicas), y no hay garantía de que los gradientes calculados seguirán apuntando en la dirección correcta (consulte la Figura 19-16).

Cuando los gradientes están muy desactualizados, se les llama gradientes obsoletos: pueden ralentizar la convergencia, introduciendo ruido y efectos de oscilación (la curva de aprendizaje puede contener oscilaciones temporales), o incluso pueden hacer que el algoritmo de entrenamiento diverja.

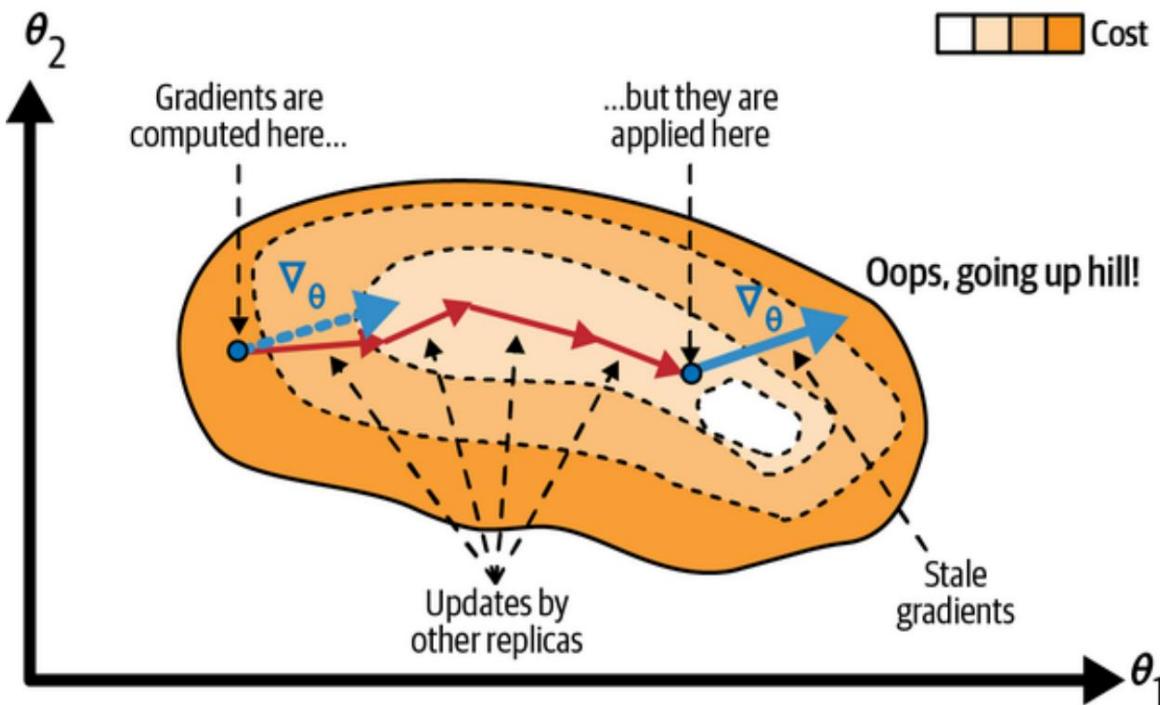


Figura 19-16. Gradiéntes obsoletos cuando se utilizan actualizaciones asincrónicas

Hay algunas formas de reducir el efecto de los gradiéntes obsoletos:

- Reducir la tasa de aprendizaje.
- Elimine los degradados obsoletos o redúzcalos.
- Ajuste el tamaño del mini lote.
- Comience las primeras épocas usando solo una réplica (esto se llama fase de calentamiento). Los gradiéntes obsoletos tienden a ser más dañinos en

al comienzo del entrenamiento, cuando los gradientes suelen ser grandes y los parámetros aún no se han asentado en un valle de la función de costos, por lo que diferentes réplicas pueden empujar los parámetros en direcciones bastante diferentes.

Un artículo publicado por el equipo de Google Brain en 2016 comparó <sup>17</sup> varios enfoques y descubrió que usar actualizaciones sincrónicas con algunas réplicas de repuesto era más eficiente que usar actualizaciones asincrónicas, no solo convergiendo más rápido sino también produciendo un mejor modelo. Sin embargo, esta sigue siendo un área de investigación activa, por lo que no debes descartar las actualizaciones asincrónicas todavía.

#### Saturación de ancho de banda

Ya sea que utilice actualizaciones sincrónicas o asincrónicas, el paralelismo de datos con parámetros centralizados aún requiere comunicar los parámetros del modelo desde los servidores de parámetros a cada réplica al comienzo de cada paso de entrenamiento, y los gradientes en la otra dirección al final de cada paso de entrenamiento. De manera similar, cuando se utiliza la estrategia reflejada, los gradientes producidos por cada GPU deberán compartirse con todas las demás GPU. Desafortunadamente, a menudo llega un punto en el que agregar una GPU adicional no mejorará el rendimiento en absoluto porque el tiempo dedicado a mover los datos dentro y fuera de la RAM de la GPU (y a través de la red en una configuración distribuida) superará la velocidad obtenida al dividir el cálculo. carga. En ese punto, agregar más GPU solo empeorará la saturación del ancho de banda y, de hecho, ralentizará el entrenamiento.

La saturación es más grave para los modelos grandes y densos, ya que tienen muchos parámetros y gradientes para transferir. Es menos severo para modelos pequeños (pero la ganancia de paralelización es limitada) y para modelos grandes y dispersos, donde los gradientes suelen ser en su mayoría ceros y, por lo tanto, pueden comunicarse de manera eficiente. Jeff Dean, iniciador y líder del proyecto Google Brain, informó aceleraciones típicas de 25 a 40 × cuando se distribuyen cálculos en 50 GPU para modelos densos, y una aceleración de 300 × para modelos más dispersos entrenados en 500 GPU. Como tu

Como puede ver, los modelos dispersos realmente escalan mejor. Aquí hay algunos ejemplos concretos:

- Traducción automática neuronal: aceleración  $6\times$  en 8 GPU
- Inception/ImageNet: aceleración  $32\times$  en 50 GPU
- RankBrain: aceleración de  $300\times$  en 500 GPU

Se están realizando muchas investigaciones para aliviar el problema de saturación del ancho de banda, con el objetivo de permitir que el entrenamiento escale linealmente con la cantidad de GPU disponibles. Por ejemplo, un [artículo de 2018](#) Un equipo de investigadores de la Universidad Carnegie Mellon, la Universidad de Stanford y Microsoft Research propusieron un sistema llamado PipeDream que logró reducir las comunicaciones de red en más del 90%, permitiendo entrenar modelos grandes en muchas máquinas. Lo lograron utilizando una nueva técnica llamada paralelismo de canalización, que combina el paralelismo de modelos y el paralelismo de datos: el modelo se divide en partes consecutivas, llamadas etapas, cada una de las cuales se entrena en una máquina diferente. Esto da como resultado una canalización asincrónica en la que todas las máquinas trabajan en paralelo con muy poco tiempo de inactividad. Durante el entrenamiento, cada etapa alterna una ronda de propagación hacia adelante y una ronda de propagación hacia atrás (consulte [la Figura 19-17](#)): extrae un mini lote de su cola de entrada, lo procesa y envía las salidas a la cola de entrada de la siguiente etapa, luego extrae un mini lote de gradientes de su cola de gradientes, propaga hacia atrás estos gradientes y actualiza sus propios parámetros de modelo, y empuja los gradientes retropropagados a la cola de gradientes de la etapa anterior. Luego repite todo el proceso una y otra vez. Cada etapa también puede utilizar paralelismo de datos regular (por ejemplo, utilizando la estrategia reflejada), independientemente de las otras etapas.

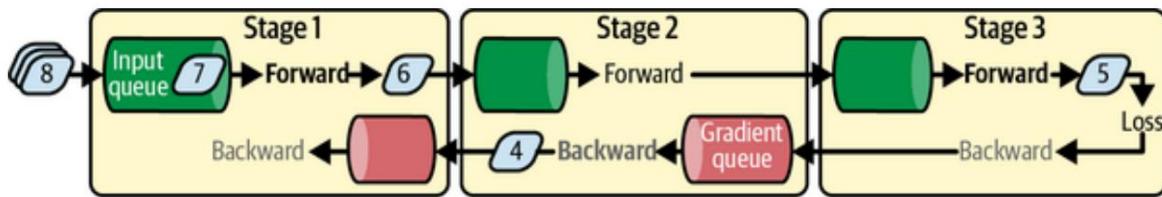


Figura 19-17. Paralelismo de canalización de PipeDream

Sin embargo, tal como se presenta aquí, PipeDream no funcionaría tan bien.

Para entender por qué, considere el minilote 5 en la Figura 19-17: cuando pasó por la etapa 1 durante el pase hacia adelante, los gradientes del minilote 4 aún no se habían propagado hacia atrás a través de esa etapa, pero cuando llegó el 5 Los gradientes regresan a la etapa 1, los gradientes del n.º 4 se habrán utilizado para actualizar los parámetros del modelo, por lo que los gradientes del n.º 5 estarán un poco obsoletos. Como hemos visto, esto puede degradar la velocidad y precisión del entrenamiento, e incluso hacerlo divergir: cuantas más etapas haya, peor se vuelve este problema. Sin embargo, los autores del artículo propusieron métodos para mitigar este problema: por ejemplo, cada etapa guarda pesos durante la propagación hacia adelante y los restaura durante la propagación hacia atrás, para garantizar que se utilicen los mismos pesos tanto para el paso hacia adelante como para el paso hacia atrás.

A esto se le llama ocultación de peso. Gracias a esto, PipeDream demuestra una impresionante capacidad de escalamiento, mucho más allá del simple paralelismo de datos.

El último avance en este campo de investigación se publicó en un [artículo de 2022](#) por investigadores<sup>19</sup> de Google: desarrollaron un sistema llamado Pathways que utiliza paralelismo de modelos automatizado, programación de grupos asíncronos y otras técnicas para alcanzar casi el 100% de utilización del hardware en miles de TPU. Programar significa organizar cuándo y dónde debe ejecutarse cada tarea, y programar en grupo significa ejecutar tareas relacionadas al mismo tiempo, en paralelo y cerca unas de otras para reducir el tiempo que las tareas tienen que esperar por los resultados de las demás. Como vimos en el Capítulo 16, este sistema se utilizó para entrenar un modelo de lenguaje masivo en más de 6.000 TPU, con una utilización de hardware cercana al 100%: es una hazaña de ingeniería alucinante.

Al momento de escribir este artículo, Pathways aún no es público, pero es probable que en un futuro cercano puedas entrenar modelos enormes en Vertex AI usando Pathways o un sistema similar. Mientras tanto, para reducir el problema de saturación, probablemente querrás usar algunas GPU potentes en lugar de muchas GPU débiles, y si necesitas entrenar un modelo en varios servidores, debes agrupar tus GPU en pocas y muy bien. servidores interconectados. También puede intentar reducir la precisión del flotador de 32 bits (`tf.float32`) a 16 bits (`tf.bfloat16`).

Esto reducirá a la mitad la cantidad de datos a transferir, a menudo sin mucho impacto en la tasa de convergencia o el rendimiento del modelo. Por último, si utiliza parámetros centralizados, puede fragmentar (dividir) los parámetros en varios servidores de parámetros: agregar más servidores de parámetros reducirá la carga de la red en cada servidor y limitará el riesgo de saturación del ancho de banda.

Bien, ahora que hemos repasado toda la teoría, ¡entrenemos un modelo en múltiples GPU!

## Entrenamiento a escala usando la API de estrategias de distribución

Afortunadamente, TensorFlow viene con una API muy buena que se encarga de toda la complejidad de distribuir su modelo en múltiples dispositivos y máquinas: la API de estrategias de distribución. Para entrenar un modelo de Keras en todas las GPU disponibles (en una sola máquina, por ahora) usando el paralelismo de datos con la estrategia reflejada, simplemente cree un objeto `MirroredStrategy`, llame a su método `alcance()` para obtener un contexto de distribución y ajuste la creación y compilación. de su modelo dentro de ese contexto. Luego llame al método `fit()` del modelo normalmente:

```
estrategia = tf.distribute.MirroredStrategy()  
  
con estrategia.scope(): modelo =  
    tf.keras.Sequential([...]) # crear un Keras
```

```

modelo normalmente
model.compile([...]) # compila el modelo normalmente

tamaño_lote = 100 # preferiblemente divisible por el número de réplicas model.fit(X_train,
y_train,
epochs=10, validation_data=(X_valid, y_valid),
batch_size=batch_size)

```

En el fondo, Keras tiene en cuenta la distribución, por lo que en este contexto de MirroredStrategy sabe que debe replicar todas las variables y operaciones en todos los dispositivos GPU disponibles. Si nos fijamos en los pesos del modelo, son del tipo MirroredVariable:

```

>>> tipo(modelo.pesos[0])
tensorflow.python.distribute.values.MirroredVariable

```

Tenga en cuenta que el método fit() dividirá automáticamente cada lote de entrenamiento entre todas las réplicas, por lo que es preferible asegurarse de que el tamaño del lote sea divisible por la cantidad de réplicas (es decir, la cantidad de GPU disponibles) para que todas las réplicas obtengan lotes de el mismo tamaño. ¡Y eso es todo! La capacitación generalmente será significativamente más rápida que usar un solo dispositivo y el cambio de código fue realmente mínimo.

Una vez que haya terminado de entrenar su modelo, puede usarlo para hacer predicciones de manera eficiente: llame al método predict() y automáticamente dividirá el lote entre todas las réplicas, haciendo predicciones en paralelo. Nuevamente, el tamaño del lote debe ser divisible por el número de réplicas. Si llama al método save() del modelo, se guardará como un modelo normal, no como un modelo reflejado con múltiples réplicas. Entonces, cuando lo cargues, se ejecutará como un modelo normal, en un solo dispositivo: de forma predeterminada en la GPU n.º 0 o en la CPU si no hay GPU. Si desea cargar un modelo y ejecutarlo en todos los dispositivos disponibles, debe llamar a tf.keras.models.load\_model() dentro de un contexto de distribución:

```
con estrategia.alcance():
    modelo = tf.keras.models.load_model("mi_modelo_espejo")
```

Si solo desea utilizar un subconjunto de todos los dispositivos GPU disponibles, puede pasar la lista al constructor de MirroredStrategy:

```
estrategia = tf.distribute.MirroredStrategy(dispositivos= ["/gpu:0", "/gpu:1"])
```

De forma predeterminada, la clase MirroredStrategy utiliza la Biblioteca de comunicaciones colectivas de NVIDIA (NCCL) para la operación media AllReduce, pero puede cambiarla estableciendo el argumento cross\_device\_ops en una instancia de la clase

`tf.distribute.HierarchicalCopyAllReduce`, o una instancia de `tf.` clase `distribuir.ReductionToOneDevice`.

La opción NCCL predeterminada se basa en la clase `tf.distribute.NcclAllReduce`, que suele ser más rápida, pero depende de la cantidad y los tipos de GPU, por lo que es posible que desee probar las alternativas.

20

Si desea intentar utilizar el paralelismo de datos con parámetros centralizados, reemplace MirroredStrategy con Estrategia de almacenamiento central:

```
estrategia =
tf.distribute.experimental.CentralStorageStrategy()
```

Opcionalmente, puede configurar el argumento `Compute_devices` para especificar la lista de dispositivos que desea usar como trabajadores (de forma predeterminada, usará todas las GPU disponibles) y, opcionalmente, puede configurar el argumento `parámetro_device` para especificar el dispositivo en el que desea almacenar los parámetros. Por defecto utilizará la CPU, o la GPU si solo hay una.

¡Ahora veamos cómo entrenar un modelo en un grupo de servidores TensorFlow!

## Entrenamiento de un modelo en un clúster de TensorFlow

Un clúster de TensorFlow es un grupo de procesos de TensorFlow que se ejecutan en paralelo, generalmente en diferentes máquinas, y se comunican entre sí para completar algún trabajo (por ejemplo, entrenar o ejecutar un modelo de red neuronal). Cada proceso TF en el clúster se denomina tarea o servidor TF. Tiene una dirección IP, un puerto y un tipo (también llamado rol o trabajo). El tipo puede ser "trabajador", "jefe", "ps" (servidor de parámetros) o "evaluador":

- Cada trabajador realiza cálculos, normalmente en una máquina con una o más GPU.
- El jefe también realiza cálculos (es un trabajador), pero también maneja trabajo adicional, como escribir registros de TensorBoard o guardar puntos de control. Hay un solo jefe en un grupo. Si no se especifica explícitamente ningún jefe, entonces, por convención, el primer trabajador es el jefe.
- Un servidor de parámetros solo realiza un seguimiento de los valores de las variables y, por lo general, se encuentra en una máquina que solo tiene CPU. Este tipo de tarea solo se utiliza con ParameterServerStrategy.
- Obviamente, un evaluador se encarga de la evaluación. Este tipo no se utiliza con frecuencia y, cuando se utiliza, suele haber un solo evaluador.

Para iniciar un clúster de TensorFlow, primero debe definir su especificación.

Esto significa definir la dirección IP, el puerto TCP y el tipo de cada tarea. Por ejemplo, la siguiente especificación de clúster define un clúster con tres tareas (dos trabajadores y un servidor de parámetros; consulte [la Figura 19-18](#)). La especificación del clúster es un diccionario con una clave por trabajo y los valores son listas de direcciones de tareas (IP:puerto):

```
cluster_spec =
  { "trabajador":
    [ "máquina-a.ejemplo.com:2222", #
```

```
/trabajo:trabajador/tarea:0
    "máquina-b.ejemplo.com:2222" /trabajo:trabajador/      #
tarea:1 ], "ps": ["máquina-
    a.ejemplo.com:2221"] # /trabajo: pd/tarea:0
}
```

En general, habrá una única tarea por máquina, pero como muestra este ejemplo, puedes configurar varias tareas en la misma máquina si lo deseas. En este caso, si comparten las mismas GPU, asegúrese de que la RAM esté dividida adecuadamente, como se mencionó anteriormente.

#### ADVERTENCIA

De forma predeterminada, cada tarea en el clúster puede comunicarse con todas las demás tareas, así que asegúrese de configurar su firewall para autorizar todas las comunicaciones entre estas máquinas en estos puertos (normalmente es más sencillo si usa el mismo puerto en cada máquina).

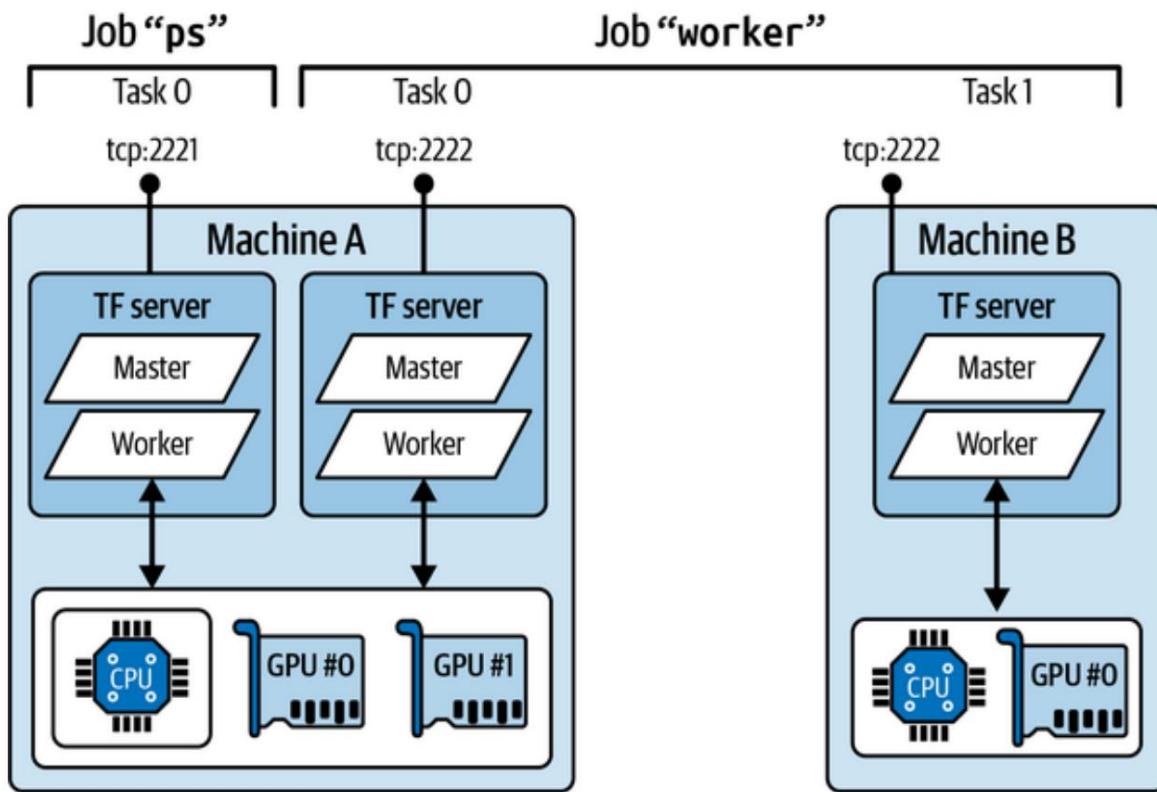


Figura 19-18. Un ejemplo de clúster de TensorFlow

Cuando inicia una tarea, debe darle la especificación del clúster y también debe decirle cuál es su tipo e índice (por ejemplo, trabajador #0). La forma más sencilla de especificar todo a la vez (tanto la especificación del clúster como el tipo e índice de la tarea actual) es configurar la variable de entorno `TF_CONFIG` antes de iniciar TensorFlow. Debe ser un diccionario codificado en JSON que contenga una especificación de clúster (bajo la clave "clúster") y el tipo e índice de la tarea actual (bajo la clave "tarea"). Por ejemplo, la siguiente variable de entorno `TF_CONFIG` utiliza el clúster que acabamos de definir y especifica que la tarea a iniciar es el trabajador n.º 0:

```
os.environ["TF_CONFIG"] = json.dumps({ "cluster": cluster_spec,
    "task": {"type": "worker", "index": 0}
})
```

## CONSEJO

En general, desea definir la variable de entorno TF\_CONFIG fuera de Python, por lo que no es necesario que el código incluya el tipo y el índice de la tarea actual (esto hace posible usar el mismo código en todos los trabajadores).

¡Ahora entrenemos un modelo en un clúster! Comenzaremos con la estrategia reflejada. Primero, debe configurar la variable de entorno TF\_CONFIG de forma adecuada para cada tarea. No debería haber ningún servidor de parámetros (elimine la clave "ps" en la especificación del clúster) y, en general, querrá un solo trabajador por máquina. Asegúrese de establecer un índice de tarea diferente para cada tarea. Finalmente, ejecute el siguiente script en cada trabajador:

```
importar archivo
temporal importar tensorflow como tf

estrategia = tf.distribute.MultiWorkerMirroredStrategy() # ¡al principio!

solucionador =
tf.distribute.cluster_resolver.TFConfigClusterResolver() print(f"Iniciando tarea
{resolver.task_type} # {resolver.task_id}") # carga y divide el
conjunto de datos MNIST

con estrategia.alcance():
    model = tf.keras.Sequential([...]) # construir el modelo Keras

    model.compile([...]) # compila el modelo

model.fit(X_train, y_train, validation_data=(X_valid, y_valid), épocas=10)

if resolutor.task_id == 0: # el jefe guarda el modelo en la ubicación correcta
    model.save("my_mnist_multiworker_model", save_format="tf")
```

demás:

```
tmpdir = tempfile.mkdtemp() # otros trabajadores guardan en un directorio temporal
model.save(tmpdir,
    save_format="tf") tf.io.gfile.rmtree(tmpdir) # y podemos
    eliminar esto
directorío al final!
```

Es casi el mismo código que usaste antes, excepto que esta vez estás usando MultiWorkerMirroredStrategy. Cuando inicie este script en los primeros trabajadores, permanecerán bloqueados en el paso AllReduce, pero el entrenamiento comenzará tan pronto como se inicie el último trabajador, y los verá a todos avanzando exactamente al mismo ritmo ya que se sincronizan en cada paso. .

#### ADVERTENCIA

Al utilizar MultiWorkerMirroredStrategy, es importante asegurarse de que todos los trabajadores hagan lo mismo, incluido guardar puntos de control del modelo o escribir registros de TensorBoard, aunque solo conservará lo que escribe el jefe. Esto se debe a que es posible que estas operaciones necesiten ejecutar las operaciones AllReduce, por lo que todos los trabajadores deben estar sincronizados.

Hay dos implementaciones de AllReduce para esta estrategia de distribución: un algoritmo de anillo AllReduce basado en gRPC para las comunicaciones de red y la implementación de NCCL. El mejor algoritmo a utilizar depende de la cantidad de trabajadores, la cantidad y los tipos de GPU y la red. De forma predeterminada, TensorFlow aplicará algunas heurísticas para seleccionar el algoritmo adecuado para usted, pero puede forzar NCCL (o RING) de esta manera:

```
estrategia = tf.distribute.MultiWorkerMirroredStrategy()
```

```
opciones_comunicación=tf.distribute.experimental.Opcionesdecomunicación(
```

```
implementación=tf.distribute.experimental.CollectiveCommunication.NCCL))
```

Si prefiere implementar paralelismo de datos asincrónicos con servidores de parámetros, cambie la estrategia a ParameterServerStrategy, agregue uno o más servidores de parámetros y configure TF\_CONFIG adecuadamente para cada tarea. Tenga en cuenta que, aunque los trabajadores funcionarán de forma asincrónica, las réplicas de cada trabajador funcionarán de forma sincrónica.

Por último, si tiene acceso a [TPU en Google Cloud, por ejemplo](#) Por ejemplo, si usa Colab y configura el tipo de acelerador en TPU, puede crear una TPUStrategy como esta:

```
solucionador =  
tf.distribute.cluster_resolver.TPUClusterResolver()  
tf.tpu.experimental.initialize_tpu_system(resolver) estrategia =  
tf.distribute.experimental.TPUStrategy(resolver)
```

Esto debe ejecutarse inmediatamente después de importar TensorFlow. Luego podrá utilizar esta estrategia normalmente.

CONSEJO

Si es investigador, puede ser elegible para utilizar TPU de forma gratuita; ver <https://tensorflow.org/tfrc> para más detalles.

Ahora puedes entrenar modelos en múltiples GPU y múltiples servidores: ¡date una palmadita en la espalda! Sin embargo, si desea entrenar un modelo muy grande, necesitará muchas GPU en muchos servidores, lo que requerirá comprar una gran cantidad de hardware o administrar muchas máquinas virtuales en la nube. En muchos casos, es menos complicado y menos costoso utilizar un servicio en la nube que se encargue de aprovisionar y administrar toda esta infraestructura por usted, justo cuando la necesita. Veamos cómo hacerlo usando Vertex AI.

## Ejecutar grandes trabajos de capacitación en Vertex AI

Vertex AI le permite crear trabajos de capacitación personalizados con su propio código de capacitación. De hecho, puedes usar casi el mismo código de entrenamiento que usarías en tu propio clúster TF. Lo principal que debes cambiar es dónde el jefe debe guardar el modelo, los puntos de control y los registros de TensorBoard. En lugar de guardar el modelo en un directorio local, el jefe debe guardarlo en GCS, utilizando la ruta proporcionada por Vertex AI en la variable de entorno AIP\_MODEL\_DIR. Para los puntos de control del modelo y los registros de TensorBoard, debe utilizar las rutas contenidas en las variables de entorno AIP\_CHECKPOINT\_DIR y AIP\_TENSORBOARD\_LOG\_DIR, respectivamente. Por supuesto, también debe asegurarse de que se pueda acceder a los datos de capacitación desde las máquinas virtuales, como en GCS u otro servicio de GCP como BigQuery, o directamente desde la web. Por último, Vertex AI establece explícitamente el tipo de tarea "jefe", por lo que debes identificar al jefe usando resolve.task\_type == "jefe" en lugar de resolve.task\_id == 0:

```

importar
sistema operativo [...] # otras importaciones, crear MultiWorkerMirroredStrategy y
resolver

if resolver.task_type == "chief": model_dir =
    os.getenv("AIP_MODEL_DIR") # rutas proporcionadas por Vertex AI

    tensorboard_log_dir =
    os.getenv("AIP_TENSORBOARD_LOG_DIR")
    checkpoint_dir = os.getenv("AIP_CHECKPOINT_DIR") más:

    tmp_dir = Path(tempfile.mkdtemp()) # otros trabajadores usan directorios
temporales model_dir
        = tmp_dir / "model" tensorboard_log_dir =
    tmp_dir / "logs" checkpoint_dir = tmp_dir / "ckpt"

devoluciones de llamada =
[tf.keras.callbacks.TensorBoard(tensorboard_log_dir),

```

```
tf.keras.callbacks.ModelCheckpoint(checkpoint_dir)] [...] # compilar y  
compilar usando el alcance de la estrategia, como antes
```

```
model.fit(X_train, y_train, validation_data=(X_valid, y_valid), epochs=10,  
callbacks=callbacks)  
    model.save(model_dir,  
save_format="tf")
```

## CONSEJO

Si coloca los datos de entrenamiento en GCS, puede crear un `tf.data.TextLineDataset` o `tf.data.TFRecordDataset` para acceder a ellos: simplemente use las rutas de GCS como nombres de archivo (por ejemplo, `gs://my_bucket/data/001.csv`). Estos conjuntos de datos dependen del paquete `tf.io.gfile` para acceder a los archivos: admite archivos locales y archivos GCS.

Ahora puede crear un trabajo de capacitación personalizado en Vertex AI, basado en este script. Deberá especificar el nombre del trabajo, la ruta a su script de entrenamiento, la imagen de Docker que usará para el entrenamiento, la que usará para las predicciones (después del entrenamiento), cualquier biblioteca de Python adicional que pueda necesitar y, por último, el depósito que Vertex La IA debería utilizarlo como directorio provisional para almacenar el guión de entrenamiento. De forma predeterminada, ahí también es donde el script de entrenamiento guardará el modelo entrenado, así como los registros de TensorBoard y los puntos de control del modelo (si los hay). Creemos el trabajo:

```
custom_training_job =  
  
    aiplatform.CustomTrainingJob( display_name="my_custom_training_job",  
        script_path="my_vertex_ai_training_task.py", container_uri="gcr.io/  
cloud-aiplatform/  
            training/tf-gpu.2-4:latest",  
        model_serving_container_image_uri=server_image, requisitos=  
        ["gcsfs==2022.3.0"], # no es necesario, esto es solo un ejemplo
```

```
    staging_bucket=f"gs://{bucket_name}/staging"
)
```

Y ahora ejecutémoslo en dos trabajadores, cada uno con dos GPU:

```
mnist_model2 = custom_training_job.run(machine_type="n1-
    standard-4", replica_count=2,
    accelerator_type="NVIDIA_TESLA_K80", accelerator_count=2,
)
```

Y eso es todo: Vertex AI aprovisionará los nodos de computación que usted solicitó (dentro de sus cuotas) y ejecutará su script de entrenamiento en ellos. Una vez que se completa el trabajo, el método run() devolverá un modelo entrenado que puede usar exactamente igual al que creó anteriormente: puede implementarlo en un punto final o usarlo para hacer predicciones por lotes. Si algo sale mal durante el entrenamiento, puede ver los registros en la consola de GCP: en el menú de navegación , seleccione Vertex AI → Entrenamiento, haga clic en su trabajo de entrenamiento y haga clic en VER REGISTROS. Alternativamente, puede hacer clic en la pestaña TRABAJOS PERSONALIZADOS y copiar el ID del trabajo (por ejemplo, 1234), luego seleccionar Registro en el menú de navegación y consultar recurso.labels.job\_id=1234.

#### CONSEJO

Para visualizar el progreso del entrenamiento, simplemente inicie TensorBoard y apunte su -logdir a la ruta GCS de los registros. Utilizará las credenciales predeterminadas de la aplicación, que puede configurar mediante el inicio de sesión predeterminado de la aplicación de autenticación de gcloud. Vertex AI también ofrece servidores TensorBoard alojados si lo prefiere.

Si desea probar algunos valores de hiperparámetros, una opción es ejecutar varios trabajos. Puede pasar los valores de los hiperparámetros a su secuencia de comandos como argumentos de línea de comandos configurando el parámetro args

al llamar al método `run()`, o puede pasárselas como variables de entorno utilizando el parámetro `Environment_variables`.

Sin embargo, si desea ejecutar un gran trabajo de ajuste de hiperparámetros en la nube, una opción mucho mejor es utilizar el servicio de ajuste de hiperparámetros de Vertex AI. Veamos cómo.

Ajuste de hiperparámetros en Vertex AI El servicio de ajuste de hiperparámetros de Vertex AI se basa en un algoritmo de optimización bayesiano, capaz de encontrar rápidamente combinaciones óptimas de hiperparámetros. Para usarlo, primero debe crear un script de entrenamiento que acepte valores de hiperparámetros como argumentos de la línea de comandos. Por ejemplo, su script podría usar la biblioteca estándar `argparse` de esta manera:

```
importar análisis de argumentos

analizador = argparse.ArgumentParser()
parser.add_argument("--n_hidden", type=int, default=2) parser.add_argument("--n_neurons", type=int, default=256) parser.add_argument("--tasa_de_aprendizaje", tipo=float, predeterminado=1e-2) parser.add_argument("--optimizer", default="adam")
args = parser.parse_args()
```

El servicio de ajuste de hiperparámetros llamará a su secuencia de comandos varias veces, cada vez con diferentes valores de hiperparámetro: cada ejecución se denomina prueba y el conjunto de pruebas se denomina estudio. Luego, su secuencia de comandos de entrenamiento debe usar los valores de hiperparámetros proporcionados para construir y compilar un modelo. Puede utilizar una estrategia de distribución reflejada si lo desea, en caso de que cada prueba se ejecute en una máquina con múltiples GPU. Luego, el script puede cargar el conjunto de datos y entrenar el modelo. Por ejemplo:

```
importar tensorflow como tf
```

```

def build_model(argumentos):
    con tf.distribute.MirroredStrategy().scope(): model = tf.keras.Sequential()
        model.add(tf.keras.layers.Flatten(input_shape=[28,
        28], dtype=tf.uint8)) para en rango (args.n_hidden):
            -
            model.add(tf.keras.layers.Dense(args.n_neurons,
            activación="relu"))
            model.add(tf.keras.layers.Dense(10, activación="softmax"))

    opt = tf.keras.optimizers.get(args.optimizer) opt.learning_rate =
    args.learning_rate

    model.compile(loss="sparse_categorical_crossentropy", optimizador=opt,
                  métricas=["precisión"])
    modelo de devolución

[...] # cargar el conjunto de datos modelo
= build_model(args) historial =
model.fit([...])

```

## CONSEJO

Puede utilizar las variables de entorno AIPE\_\* que mencionamos anteriormente para determinar dónde guardar los puntos de control, los registros de TensorBoard y el modelo final.

Por último, el script debe informar el rendimiento del modelo al servicio de ajuste de hiperparámetros de Vertex AI, para que pueda decidir qué hiperparámetros probar a continuación. Para ello, debe utilizar la biblioteca hypertune, que se instala automáticamente en las máquinas virtuales de entrenamiento de Vertex AI.

```

importar hypertune

hypertune = hypertune.HyperTune()
hypertune.report_hyperparameter_tuning_metric(
    hyperparameter_metric_tag="accuracy", # nombre de la métrica reportada

```

```

    valor_métrico=max(historia.historia["val_accuracy"]), #
valor métrico

    global_step=model.optimizer.iterations.numpy(),
)

```

Ahora que su script de entrenamiento está listo, necesita definir el tipo de máquina en la que le gustaría ejecutarlo. Para esto, debes definir un trabajo personalizado, que Vertex AI utilizará como plantilla para cada prueba:

```

trial_job = aiplatform.CustomJob.from_local_script( display_name="my_search_trial_job",
    script_path="my_vertex_ai_trial.py", # ruta a su script
    de capacitación contenedor_uri="gcr.io/cloud-aiplatform/training/tf-gpu.2-4:latest
", staging_bucket=f"gs://
{bucket_name}/staging", accelerator_type="NVIDIA_TESLA_K80",
accelerator_count=2, #
en este ejemplo, cada prueba tendrá 2 GPU

```

```
)
```

Finalmente, está listo para crear y ejecutar el trabajo de ajuste de hiperparámetros:

```
desde google.cloud.aiplatform importe hyperparameter_tuning como hpt
```

```
hp_job =
```

```

    aiplatform.HyperparameterTuningJob( display_name="my_hp_search_job",
custom_job=trial_job, metric_spec={"accuracy": "maximize"}, parámetro_spec={ "learning_rate": hpt.Double
        "n_neurons": hpt.IntegerParameterSpec(min=1, max=300,
escala="lineal"),
        "n_hidden": hpt.IntegerParameterSpec(min=1, max=10, scale="linear"),
"optimizador":
        hpt.CategoricalParameterSpec(["sgd", "adam"]),

```

```

},
max_trial_count=100,
paralelo_trial_count=20,
) hp_job.run()

```

Aquí, le decimos a Vertex AI que maximice la métrica denominada "precisión": este nombre debe coincidir con el nombre de la métrica informada por el script de entrenamiento. También definimos el espacio de búsqueda, usando una escala logarítmica para la tasa de aprendizaje y una escala lineal (es decir, uniforme) para los otros hiperparámetros. Los nombres de los hiperparámetros deben coincidir con los argumentos de la línea de comandos del script de entrenamiento. Luego establecemos el número máximo de pruebas en 100 y el número máximo de pruebas que se ejecutan en paralelo en 20. Si aumenta el número de pruebas paralelas a (digamos) 60, el tiempo total de búsqueda se reducirá significativamente, en un factor de hasta 3. Pero los primeros 60 ensayos se iniciarán en paralelo, por lo que no se beneficiarán de la retroalimentación de los otros ensayos. Por lo tanto, debes aumentar el número máximo de pruebas para compensar, por ejemplo, hasta aproximadamente 140.

Esto llevará bastante tiempo. Una vez completado el trabajo, puede obtener los resultados de la prueba utilizando `hp_job.trials`. Cada resultado de la prueba se representa como un objeto protobuf, que contiene los valores de los hiperparámetros y las métricas resultantes. Busquemos la mejor prueba:

```

def get_final_metric(trial, metric_id): para la métrica
    en prueba.final_measurement.metrics: si metric.metric_id ==
        metric_id: devuelve metric.value

juicios = hp_job.trials
juicio_accuracies = [get_final_metric(trial, "accuracy") para prueba en ensayos]
best_trial =
    juicios[np.argmax(trial_accuracies)]

```

Ahora veamos la precisión de esta prueba y sus valores de hiperparámetros:

```
>>> max(precisiones_de_prueba)
0.977400004863739
>>> mejor_trial.id '98'

>>> best_trial.parameters
[parameter_id: "tasa_de_aprendizaje" valor {número_valor: 0.001}, parámetro_id:
valor "n_hidden" {número_valor: 8.0}, parámetro_id: valor
"n_neuronas" {número_valor: 216.0}, parámetro_id: valor
"optimizador" { valor_cadena: "adam" } ]
```

¡Eso es todo! Ahora puede obtener el modelo guardado de esta versión de prueba, opcionalmente entrenarlo un poco más e implementarlo en producción.

CONSEJO

Vertex AI también incluye un servicio AutoML, que se encarga por completo de encontrar la arquitectura del modelo adecuada y entrenarla para usted. Todo lo que necesita hacer es cargar su conjunto de datos en Vertex AI usando un formato especial que depende del tipo de conjunto de datos (imágenes, texto, tabular, video, etc.), luego crear un trabajo de entrenamiento de AutoML, apuntando al conjunto de datos y especificando el Número máximo de horas de cálculo que está dispuesto a dedicar. Vea el cuaderno para ver un ejemplo.

## SINTONIZACIÓN DE HIPERPARÁMETROS USANDO KERAS TUNER ON VÉRTEX AI

En lugar de utilizar el servicio de ajuste de hiperparámetros de Vertex AI, puede utilizar Keras Tuner (presentado en el [Capítulo 10](#)) y ejecutarlo en las máquinas virtuales de Vertex AI. Keras Tuner proporciona una forma sencilla de escalar la búsqueda de hiperparámetros distribuyéndola en varias máquinas: solo requiere configurar tres variables de entorno en cada máquina y luego ejecutar su código Keras Tuner habitual en cada máquina. Puede utilizar exactamente el mismo script en todas las máquinas. Una de las máquinas actúa como jefa (es decir, el oráculo) y las demás actúan como trabajadoras. Cada trabajador le pregunta al jefe qué valores de hiperparámetros probar, luego el trabajador entrena el modelo usando estos valores de hiperparámetros y finalmente informa el desempeño del modelo al jefe, quien luego puede decidir qué valores de hiperparámetros debe probar el trabajador a continuación.

Las tres variables de entorno que debe configurar en cada máquina son:

### KERASTUNER\_TUNER\_ID

Esto equivale a "jefe" en la máquina principal o a un identificador único en cada máquina trabajadora, como "trabajador0", "trabajador1", etc.

### KERASTUNER\_ORACLE\_IP

Esta es la dirección IP o nombre de host de la máquina principal. El propio jefe generalmente debería usar "0.0.0.0" para escuchar cada dirección IP en la máquina.

### KERASTUNER\_ORACLE\_PORT

Este es el puerto TCP por el que escuchará el jefe.

Puede distribuir Keras Tuner en cualquier conjunto de máquinas. Si desea ejecutarlo en máquinas Vertex AI, puede generar un trabajo de entrenamiento regular y simplemente modificar el script de entrenamiento para configurar las variables de entorno correctamente antes de usar Keras Tuner. Vea el cuaderno para ver un ejemplo.

Ahora tiene todas las herramientas y el conocimiento que necesita para crear arquitecturas de redes neuronales de última generación y entrenarlas a escala utilizando diversas estrategias de distribución, en su propia infraestructura o en la nube, y luego implementarlas en cualquier lugar. En otras palabras, ahora tienes superpoderes: ¡úsalos bien!

## Ejercicios

1. ¿Qué contiene un modelo guardado? ¿Cómo se inspecciona su contenido?
2. ¿Cuándo debería utilizar TF Serving? ¿Cuáles son sus principales características?  
¿Cuáles son algunas de las herramientas que puede utilizar para implementarlo?
3. ¿Cómo se implementa un modelo en múltiples instancias de TF Serving?
4. ¿Cuándo debería utilizar la API gRPC en lugar de la API REST para consultar un modelo servido por TF Serving?
5. ¿Cuáles son las diferentes formas en que TFLite reduce el tamaño de un modelo para que se ejecute en un dispositivo móvil o integrado?
6. ¿Qué es la capacitación consciente de la cuantización y por qué la necesitaría?
7. ¿Qué son el paralelismo de modelos y el paralelismo de datos? Porque es el  
¿Se recomienda generalmente este último?
8. Al entrenar un modelo en varios servidores, ¿qué estrategias de distribución se pueden utilizar? ¿Cómo eliges cuál usar?

9. Entrene un modelo (cualquier modelo que desee) e impleméntelo en TF Serving o Google Vertex AI. Escriba el código del cliente para consultarla mediante la API REST o la API gRPC. Actualice el modelo e implemente la nueva versión. Su código de cliente ahora consultará la nueva versión. Regrese a la primera versión.
10. Entrene cualquier modelo en varias GPU en la misma máquina usando MirroredStrategy (si no tiene acceso a las GPU, puede usar Google Colab con un tiempo de ejecución de GPU y crear dos GPU lógicas). Entrene el modelo nuevamente usando CentralStorageStrategy y compare el tiempo de entrenamiento.
11. Ajuste un modelo de su elección en Vertex AI, utilizando Keras Tuner o el servicio de ajuste de hiperparámetros de Vertex AI.

Las soluciones a estos ejercicios están disponibles al final del cuaderno de este capítulo, en <https://homl.info/colab3>.

## ¡Gracias!

Antes de cerrar el último capítulo de este libro, me gustaría agradecerles por leerlo hasta el último párrafo. Realmente espero que te hayas divertido tanto leyendo este libro como yo escribiéndolo, y que te resulte útil para tus proyectos, grandes o pequeños.

Si encuentra errores, envíe sus comentarios. En términos más generales, me encantaría saber qué piensa, así que no dude en ponerse en contacto conmigo a través de O'Reilly, a través del proyecto GitHub [ageron/handson-ml3](#) o en Twitter en [@aureliengeron](#).

De cara al futuro, mi mejor consejo para ti es que practiques y practiques: intenta realizar todos los ejercicios (si aún no lo has hecho), juega con los cuadernos, únete a Kaggle o alguna otra comunidad de ML, mira cursos de ML, lee artículos, asistir a conferencias y conocer expertos. Las cosas avanzan rápido, así que trate de mantenerse actualizado. Varios canales de YouTube presentan periódicamente artículos sobre aprendizaje profundo en excelente

detalle, de una manera muy cercana. Recomiendo especialmente los canales de Yannic Kilcher, Letitia Parcalabescu y Xander Steenbrugge. Para debates fascinantes sobre ML y conocimientos de alto nivel, asegúrese de consultar ML Street Talk y el canal de Lex Fridman. También ayuda enormemente tener un proyecto concreto en el que trabajar, ya sea por trabajo o por diversión (idealmente para ambos), así que si hay algo que siempre has soñado construir, ¡pruébalo!

Trabajar de forma incremental; No apuntes a la luna de inmediato, pero mantente enfocado en tu proyecto y constrúyelo pieza por pieza. Requerirá paciencia y perseverancia, pero cuando tengas un robot andante, un chatbot que funcione o cualquier otra cosa que te apetezca construir, ¡será inmensamente gratificante!

Mi mayor esperanza es que este libro lo inspire a crear una maravillosa aplicación de aprendizaje automático que nos beneficie a todos. ¿Qué será?  
—Aurélien Geron

---

<sup>1</sup> Un experimento A/B consiste en probar dos versiones diferentes de su producto en diferentes subconjuntos de usuarios para comprobar qué versión funciona mejor y obtener otras ideas.

<sup>2</sup> Google AI Platform (anteriormente conocido como Google ML Engine) y Google AutoML se fusionó en 2021 para formar Google Vertex AI.

<sup>3</sup> Una API REST (o RESTful) es una API que utiliza verbos HTTP estándar, como como GET, POST, PUT y DELETE, y utiliza entradas y salidas JSON. El protocolo gRPC es más complejo pero más eficiente; los datos se intercambian utilizando buffers de protocolo (consulte [el Capítulo 13](#)).

<sup>4</sup> Si no está familiarizado con Docker, le permite descargar fácilmente un conjunto de aplicaciones empaquetadas en una imagen de Docker (incluidas todas sus dependencias y, por lo general, alguna buena configuración predeterminada) y luego ejecutarlas en su sistema utilizando un motor Docker. Cuando ejecuta una imagen, el motor crea un contenedor Docker que mantiene las aplicaciones bien aisladas de su propio sistema, pero puede darle acceso limitado si lo desea. Es similar a una máquina virtual, pero mucho más rápida y liviana, ya que el contenedor depende directamente del kernel del host. Esto significa que la imagen no necesita incluir ni ejecutar su propio kernel.

[5](#) También hay imágenes de GPU disponibles y otras opciones de instalación. Para obtener más detalles, consulte las [instrucciones de instalación oficiales](#).

[6](#) Para ser justos, esto se puede mitigar serializando los datos primero y codificándolos en Base64 antes de crear la solicitud REST. Además, las solicitudes REST se pueden comprimir mediante gzip, lo que reduce significativamente el tamaño de la carga útil.

[7](#) Consulte también [la herramienta de transformación de gráficos](#) de TensorFlow para modificar y optimizar gráficos computacionales.

[8](#) Por ejemplo, una PWA debe incluir íconos de varios tamaños para diferentes dispositivos móviles, debe entregarse a través de HTTPS y debe incluir un archivo de manifiesto que contenga metadatos como el nombre de la aplicación y el color de fondo.

[9](#) Consulte los documentos de TensorFlow para obtener una instalación detallada y actualizada. instrucciones, ya que cambian con bastante frecuencia.

[10](#) Como vimos en [el Capítulo 12](#), un núcleo es la implementación de una operación para un tipo de datos y un tipo de dispositivo específicos. Por ejemplo, hay un kernel de GPU para la operación float32 tf.matmul(), pero no hay un kernel de GPU para int32 tf.matmul(), solo un kernel de CPU.

[11](#) También puedes usar `tf.debugging.set_log_device_placement(True)` para registrar todas las ubicaciones de dispositivos.

[12](#) Esto puede ser útil si desea garantizar una reproducibilidad perfecta, como explica en [este video](#), basado en TF 1.

[13](#) Al momento de escribir este artículo, solo busca previamente los datos en la RAM de la CPU, pero usa `tf.data.experimental.prefetch_to_device()` para buscar previamente los datos y enviarlos al dispositivo de tu elección para que la GPU no los desperdicie. Tiempo de espera a que se transfieran los datos.

[14](#) Si las dos CNN son idénticas, entonces se denomina red neuronal siamesa.

[15](#) Si está interesado en ir más allá con el paralelismo de modelos, consulte [Mesh TensorFlow](#).

[16](#) Este nombre es un poco confuso porque parece que algunas réplicas son especiales y no hacen nada. En realidad, todas las réplicas son equivalentes: todas trabajan duro para estar entre las más rápidas en cada paso del entrenamiento, y los perdedores varían en cada paso (a menos que algunos dispositivos sean realmente más lentos que otros). Sin embargo, sí significa que si uno o dos servidores fallan, la capacitación continuará sin problemas.

[17](#) Jianmin Chen et al., “Revisiting Distributed Synchronous SGD”, preimpresión de arXiv arXiv:1604.00981 (2016).

[18](#) Aaron Harlap et al., “PipeDream: Fast and Efficient Pipeline Parallel DNN Training”, preimpresión de arXiv arXiv:1806.03377 (2018).

**19** Paul Barham et al., "Pathways: flujo de datos distribuido asincrónico para ML", Preimpresión de arXiv arXiv:2203.12533 (2022).

**20** Para obtener más detalles sobre los algoritmos AllReduce, lea [la publicación de Yuichiro Ueno](#) sobre las tecnologías detrás del aprendizaje profundo y [la publicación de Sylvain Jaugey](#) sobre la ampliación masiva de la formación en aprendizaje profundo con NCCL.

# Apéndice A. Lista de verificación del proyecto de aprendizaje automático

---

Esta lista de verificación puede guiarte a través de tus proyectos de aprendizaje automático. Hay ocho pasos principales:

1. Plantee el problema y observe el panorama general.
2. Obtenga los datos.
3. Explore los datos para obtener información.
4. Prepare los datos para exponer mejor los patrones de datos subyacentes a los algoritmos de aprendizaje automático.
5. Explore muchos modelos diferentes y seleccione los mejores.
6. Ajusta tus modelos y combínalos en una gran solución.
7. Presente su solución.
8. Inicie, supervise y mantenga su sistema.

Obviamente, deberías sentirte libre de adaptar esta lista de verificación a tus necesidades.

## Encuadre el problema y observe el panorama general

1. Definir el objetivo en términos de negocio.
2. ¿Cómo se utilizará su solución?
3. ¿Cuáles son las soluciones/soluciones alternativas actuales (si las hay)?
4. ¿Cómo debería plantear este problema (supervisado/no supervisado, en línea/fuera de línea, etc.)?

5. ¿Cómo se debe medir el desempeño?
6. ¿La medida de desempeño está alineada con el objetivo comercial?
7. ¿Cuál sería el desempeño mínimo necesario para alcanzar el objetivo de negocio?
8. ¿Cuáles son los problemas comparables? ¿Puedes reutilizar la experiencia o  
  ¿herramientas?
9. ¿Hay experiencia humana disponible?
10. ¿Cómo solucionarías el problema manualmente?
11. Enumere las suposiciones que usted (u otros) han hecho hasta ahora.
12. Verifique las suposiciones si es posible.

## Obtener los datos

Nota: automate tanto como sea posible para que pueda obtener datos nuevos fácilmente.

1. Enumere los datos que necesita y cuánto necesita.
2. Busque y documente dónde puede obtener esos datos.
3. Comprueba cuánto espacio ocupará.
4. Consultar las obligaciones legales y obtener autorización si es necesario.
5. Obtener autorizaciones de acceso.
6. Cree un espacio de trabajo (con suficiente espacio de almacenamiento).
7. Obtenga los datos.
8. Convierta los datos a un formato que pueda manipular fácilmente (sin  
  cambiando los datos en sí).

9. Asegúrese de que la información confidencial se elimine o proteja (por ejemplo, anonimizada).
10. Comprobar el tamaño y tipo de datos (series temporales, muestrales, geográficos, etc.).
11. Pruebe un conjunto de prueba, déjelo a un lado y nunca lo mire (¡sin espiar datos!).

#### **Explore la nota de datos:**

intente obtener información de un experto en el campo para estos pasos.

1. Cree una copia de los datos para su exploración (reduciéndola a un tamaño manejable si es necesario).
2. Cree un cuaderno de Jupyter para mantener un registro de su exploración de datos.
3. Estudiar cada atributo y sus características:
  - Nombre
  - Tipo (categórico, int/float, acotado/ilimitado, texto, estructurado, etc.)
  - % de valores faltantes
  - Ruido y tipo de ruido (estocástico, valores atípicos, errores de redondeo, etc.)
  - Utilidad para la tarea.
  - Tipo de distribución (gaussiana, uniforme, logarítmica, etc.)
4. Para tareas de aprendizaje supervisado, identifique los atributos objetivo.
5. Visualice los datos.
6. Estudiar las correlaciones entre atributos.

7. Estudia cómo resolverías el problema manualmente.
8. Identifique las transformaciones prometedoras que quizás desee aplicar.
9. Identifique datos adicionales que serían útiles (regrese a “[Obtener la Datos](#)”).
10. Documente lo que ha aprendido.

## Prepare los datos

Notas:

- Trabaje en copias de los datos (mantenga intacto el conjunto de datos original).
- Escriba funciones para todas las transformaciones de datos que aplique, durante cinco razones:
  - Para que pueda preparar fácilmente los datos la próxima vez que obtenga un conjunto de datos nuevo.
  - Para que puedas aplicar estas transformaciones en futuros proyectos
  - Para limpiar y preparar el equipo de prueba
  - Para limpiar y preparar nuevas instancias de datos una vez que su solución esté activa
  - Para facilitar el tratamiento de sus opciones de preparación como hiperparámetros

1. Limpiar los datos:

- Corregir o eliminar valores atípicos (opcional).
- Complete los valores faltantes (por ejemplo, con cero, media, mediana...) o elimine sus filas (o columnas).

2. Realice la selección de funciones (opcional):

- Elimine los atributos que no proporcionen información útil para la tarea.

3. Realizar ingeniería de características, cuando corresponda:

- Discretizar características continuas.
- Descomponer características (p. ej., categóricas, fecha/hora, etc.).
- Agregue transformaciones prometedoras de características (por ejemplo,  $\log(x)$ ,  $\sqrt{x}$ , etc.).
- Agregue funciones en funciones nuevas y prometedoras.

4. Realice el escalado de funciones:

- Estandarizar o normalizar características.

## **Lista corta de modelos prometedores**

Notas:

- Si los datos son enormes, es posible que desee tomar muestras de conjuntos de entrenamiento más pequeños para poder entrenar muchos modelos diferentes en un tiempo razonable (tenga en cuenta que esto penaliza los modelos complejos, como redes neuronales grandes o bosques aleatorios).
- Una vez más, intenta automatizar estos pasos tanto como sea posible.

1. Entrene muchos modelos rápidos y sucios de diferentes categorías.

(p. ej., lineal, Bayes ingenuo, SVM, bosque aleatorio, red neuronal, etc.) utilizando parámetros estándar.

2. Mida y compare su desempeño:

- Para cada modelo, utilice la validación cruzada de N veces y calcule la media y la desviación estándar de la medida de rendimiento en las N veces.

3. Analizar las variables más significativas para cada algoritmo.
4. Analice los tipos de errores que cometen los modelos:
  - ¿Qué datos habría utilizado un humano para evitar estos errores?
5. Realice una ronda rápida de selección e ingeniería de funciones.
6. Realice una o dos iteraciones rápidas más de las cinco anteriores.  
pasos.
7. Haga una lista corta de los tres a cinco modelos más prometedores, prefiriendo modelos que cometan diferentes tipos de errores.

## Ajustar el sistema

Notas:

- Querrá utilizar tantos datos como sea posible para este paso, especialmente a medida que avanza hacia el final del ajuste.
  - Como siempre, automatiza lo que puedas.
1. Ajuste los hiperparámetros mediante validación cruzada:
    - Trate sus opciones de transformación de datos como hiperparámetros, especialmente cuando no esté seguro de ellos (por ejemplo, si no está seguro de si reemplazar los valores faltantes con ceros o con el valor mediano, o simplemente eliminar las filas).
    - A menos que haya muy pocos valores de hiperparámetros para explorar, prefiera la búsqueda aleatoria a la búsqueda en cuadrícula. Si el entrenamiento es muy largo, es posible que prefiera un enfoque de optimización bayesiano (por ejemplo, utilizando procesos previos gaussianos, como lo describen [Jasper Snoek et al.](#) ).
  2. Pruebe métodos de conjunto. Combinar los mejores modelos a menudo producirá un mejor rendimiento que ejecutarlos individualmente.

3. Una vez que esté seguro de su modelo final, mida su rendimiento en el conjunto de prueba para estimar el error de generalización.

#### ADVERTENCIA

No modifique su modelo después de medir el error de generalización: simplemente comenzaría a sobreajustar el conjunto de prueba.

## Presente su solución

1. Documente lo que ha hecho.
2. Crea una bonita presentación:
  - Asegúrese de resaltar primero el panorama general.
3. Explique por qué su solución logra el objetivo comercial.
4. No olvides presentar los puntos interesantes que hayas notado a lo largo del forma:
  - Describe qué funcionó y qué no.
  - Enumere sus suposiciones y las limitaciones de su sistema.
5. Asegúrese de que sus hallazgos clave se comuniquen a través de visualizaciones hermosas o declaraciones fáciles de recordar (por ejemplo, “el ingreso medio es el predictor número uno de los precios de la vivienda”).

## ¡Lanzamiento!

1. Prepare su solución para la producción (conéctela a las entradas de datos de producción, escriba pruebas unitarias, etc.).
2. Escriba un código de monitoreo para verificar el rendimiento en vivo de su sistema a intervalos regulares y activar alertas cuando caiga:

- Tenga cuidado con la degradación lenta: los modelos tienden a “pudrirse” a medida que evolucionan los datos.
  - Medir el desempeño puede requerir un proceso humano (por ejemplo, a través de un servicio de crowdsourcing).
  - También supervise la calidad de sus entradas (por ejemplo, un sensor defectuoso que envía valores aleatorios o la salida de otro equipo se vuelve obsoleta). Esto es particularmente importante para los sistemas de aprendizaje en línea.
3. Vuelva a entrenar sus modelos periódicamente con datos nuevos (automáticamente cuanto más se pueda).

---

<sup>1</sup> Jasper Snoek et al., “Optimización bayesiana práctica de algoritmos de aprendizaje automático”, Actas de la 25<sup>a</sup> Conferencia Internacional sobre Sistemas de Procesamiento de Información Neural 2 (2012): 2951–2959.

# Apéndice B. Autodiff

---

Este apéndice explica cómo funciona la función de autodiferenciación (autodiff) de TensorFlow y cómo se compara con otras soluciones.

Supongamos que define una función  $f(x, y) = xy + y + \frac{2}{2}$ , y necesita sus derivadas parciales  $\partial f / \partial x$  y  $\partial f / \partial y$ , normalmente para realizar el descenso de gradiente (o algún otro algoritmo de optimización). Sus opciones principales son la diferenciación manual, la aproximación de diferencias finitas, la diferenciación automática en modo directo y la diferenciación automática en modo inverso. TensorFlow implementa autodiff en modo inverso, pero para entenderlo, es útil mirar primero las otras opciones. Entonces, repasemos cada uno de ellos, comenzando con la diferenciación manual.

## Diferenciación manual

El primer método para calcular derivadas es tomar un lápiz y una hoja de papel y utilizar sus conocimientos de cálculo para derivar la ecuación adecuada. Para la función  $f(x, y)$  que acabamos de definir, no es demasiado difícil; sólo necesitas usar cinco reglas:

- La derivada de una constante es 0.
- La derivada de  $\lambda x$  es  $\lambda$  (donde  $\lambda$  es una constante).
- La derivada de  $x$  es  $\lambda x^{\lambda - 1}$ , entonces la derivada de  $x$  es  $2x$ .
- La derivada de una suma de funciones es la suma de las derivadas de estas funciones.
- La derivada de  $\lambda$  multiplicada por una función es  $\lambda$  multiplicada por su derivada.

A partir de estas reglas, se puede derivar [la Ecuación B-1](#).

Ecuación B-1. Derivadas parciales de  $f(x, y)$

$$\frac{\partial f}{\partial x} = \frac{\partial(x^2y)}{\partial x} + \frac{\partial y}{\partial x} + \frac{\partial 2}{\partial x} = y \frac{\partial(x^2)}{\partial x} + 0 + 0 = 2xy$$

$$\frac{\partial f}{\partial y} = \frac{\partial(x^2y)}{\partial y} + \frac{\partial y}{\partial y} + \frac{\partial 2}{\partial y} = x^2 + 1 + 0 = x^2 + 1$$

Este enfoque puede resultar muy tedioso para funciones más complejas y corre el riesgo de cometer errores. Afortunadamente, existen otras opciones. Veamos ahora la aproximación en diferencias finitas.

## Aproximación de diferencias finitas

Recuerde que la derivada  $h'(x)$  de una función  $h(x)$  en un punto  $x_0$  es la pendiente de la función en ese punto. Más precisamente, la derivada se define como el límite de la pendiente de una línea recta que pasa por este punto  $x_0$  y otro punto  $x$  en la función, cuando  $x$  se acerca infinitamente a  $x_0$  (ver [Ecuación B-2](#)).

Ecuación B-2. Definición de la derivada de una función  $h(x)$  en el punto  $x_0$

$$\begin{aligned} h'(x_0) &= \lim_{x \rightarrow x_0} \frac{h(x) - h(x_0)}{x - x_0} \\ &= \lim_{\epsilon \rightarrow 0} \frac{(x_0 + \epsilon) - h(x_0)}{\epsilon} \end{aligned}$$

Entonces, si quisieramos calcular la derivada parcial de  $f(x, y)$  con respecto a  $x$  en  $x = 3$  e  $y = 4$ , podríamos calcular  $f(3 + \epsilon, 4) - f(3, 4)$  y dividir el resultado por  $\epsilon$ , usando un valor muy pequeño para  $\epsilon$ . Este tipo de aproximación numérica de la derivada se llama aproximación en diferencias finitas y esta ecuación específica se llama cociente de diferencias de Newton. Eso es exactamente lo que hace el siguiente código:

```
def f(x, y): devolver
    x**2*y + y + 2

derivada def (f, x, y, x_eps, y_eps): retorno (f(x + x_eps, y
    + y_eps) - f(x, y)) / (x_eps + y_eps)
```

```
df_dx = derivada(f, 3, 4, 0.00001, 0) df_dy = derivada(f,
3, 4, 0, 0.00001)
```

Desafortunadamente, el resultado es impreciso (y empeora con funciones más complicadas). Los resultados correctos son respectivamente 24 y 10, pero en su lugar obtenemos:

```
>>> df_dx
24.000039999805264
>>> df_dy
10.000000000331966
```

Observe que para calcular ambas derivadas parciales, tenemos que llamar a `f()` al menos tres veces (lo llamamos cuatro veces en el código anterior, pero podría optimizarse). Si hubiera 1000 parámetros, necesitaríamos llamar a `f()` al menos 1001 veces. Cuando se trata de redes neuronales grandes, esto hace que la aproximación en diferencias finitas sea demasiado ineficiente.

Sin embargo, este método es tan sencillo de implementar que es una gran herramienta para comprobar que los otros métodos se implementan correctamente. Por ejemplo, si no está de acuerdo con su función derivada manualmente, entonces su función probablemente contenga un error.

Hasta ahora, hemos considerado dos formas de calcular gradientes: usando diferenciación manual y usando aproximación en diferencias finitas. Desafortunadamente, ambos tienen fallas fatales para entrenar una red neuronal a gran escala. Entonces pasemos a la diferenciación automática, comenzando con el modo de avance.

### Diferencia automática en modo directo

La Figura B-1 muestra cómo funciona la diferenciación automática en modo directo en una función aún más simple,  $g(x, y) = 5 + xy$ . La gráfica de esa función está representada a la izquierda. Después de la diferenciación automática en modo directo, obtenemos la gráfica de la derecha, que representa la derivada parcial  $\partial g / \partial x = 0 + (0 \times x + y \times 1) = y$  (de manera similar, podríamos obtener la derivada parcial con respecto a  $y$  ).

El algoritmo recorrerá el gráfico de cálculo desde las entradas hasta las salidas (de ahí el nombre "modo directo"). Comienza obteniendo las derivadas parciales de los nodos de las hojas. El nodo constante (5) devuelve la constante 0, ya que la derivada de una constante es siempre 0. La variable  $x$  devuelve la

constante 1 ya que  $\partial x/\partial x = 1$ , y la variable y devuelve la constante 0 ya que  $\partial y/\partial x = 0$  (si buscáramos la derivada parcial respecto de y sería al revés).

Ahora tenemos todo lo que necesitamos para subir en la gráfica hasta el nodo de multiplicación en la función g. El cálculo nos dice que la derivada del producto de dos funciones u y v es  $\partial(u \times v)/\partial x = \partial v/\partial x \times u + v \times \partial u/\partial x$ . Por lo tanto, podemos construir una gran parte de la gráfica de la derecha, que representa  $0 \times x + y \times 1$ .

Finalmente, podemos subir al nodo de suma en la función g. Como se mencionó, la derivada de una suma de funciones es la suma de las derivadas de estas funciones, por lo que solo necesitamos crear un nodo de suma y conectarlo a las partes del gráfico que ya hemos calculado. Obtenemos la derivada parcial correcta:  $\partial g/\partial x = 0 + (0 \times x + y \times 1)$ .

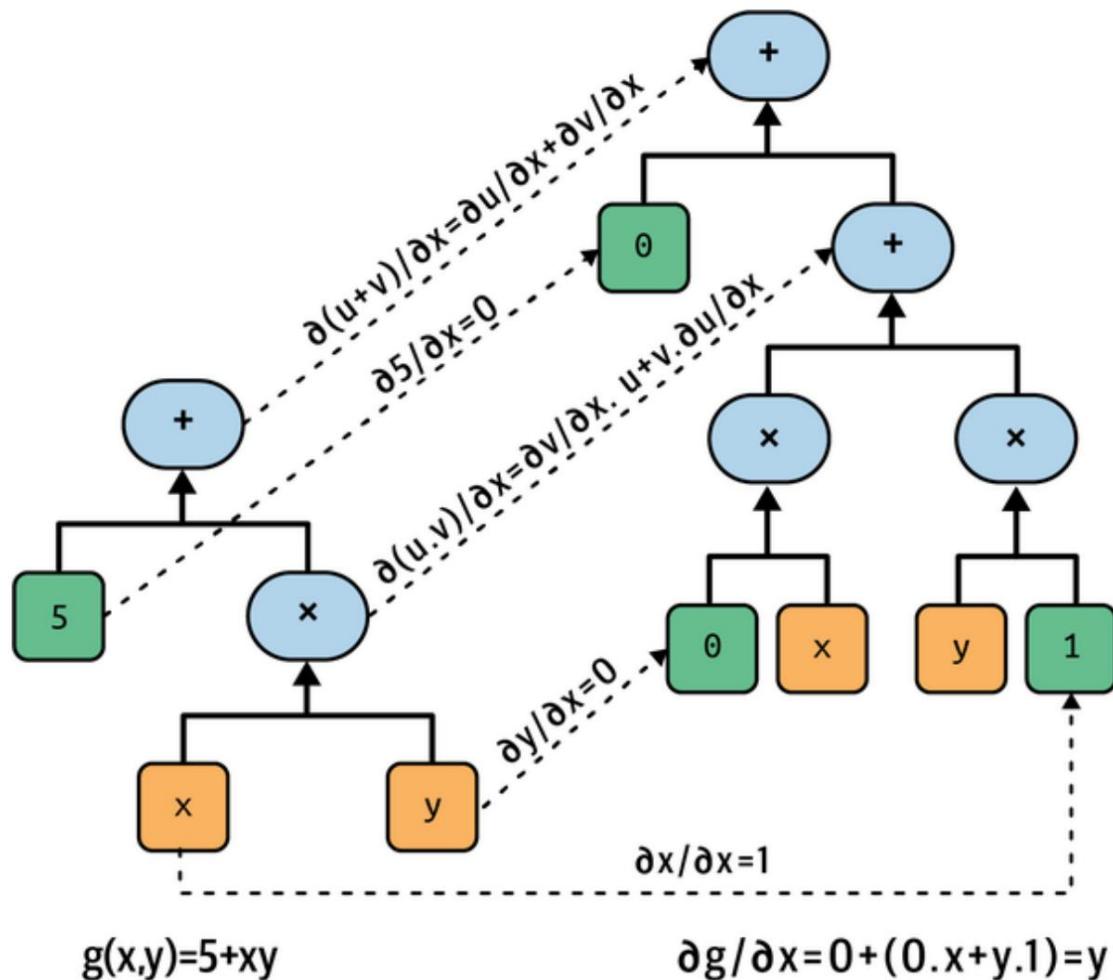


Figura B-1. Autodiff en modo directo

Sin embargo, esta ecuación se puede simplificar (mucho). Al aplicar algunos pasos de poda al gráfico de cálculo para eliminar todas las operaciones innecesarias, obtenemos un gráfico mucho más pequeño con un solo nodo:  $\partial g / \partial x = y$ . En este caso, la simplificación es bastante fácil, pero para una función más compleja, la diferenciación automática en modo directo puede producir un gráfico enorme que puede ser difícil de simplificar y conducir a un rendimiento subóptimo.

Tenga en cuenta que comenzamos con un gráfico de cálculo y la diferenciación automática en modo directo produjo otro gráfico de cálculo. Esto se llama diferenciación simbólica y tiene dos características interesantes: primero, una vez que se ha producido la gráfica de cálculo de la derivada, podemos usarla tantas veces como queramos para calcular las derivadas de la función dada para cualquier valor de  $x$  y  $y$ ; en segundo lugar, podemos ejecutar autodiff en modo directo nuevamente en el gráfico resultante para obtener derivadas de segundo orden si alguna vez lo necesitamos (es decir, derivadas de derivadas). Incluso podríamos calcular derivadas de tercer orden, etc.

Pero también es posible ejecutar autodiff en modo directo sin construir un gráfico (es decir, numéricamente, no simbólicamente), simplemente calculando resultados intermedios sobre la marcha. Una forma de hacerlo es utilizar números duales, que son números extraños pero fascinantes de la forma  $a + b\epsilon$ , donde  $a$  y  $b$  son números reales y  $\epsilon$  es un número infinitesimal tal que  $\epsilon = 0$  (pero  $\epsilon \neq 0$ ). Puedes pensar en el número dual  $42 + 24\epsilon$  como algo parecido a  $42.0000\ldots 000024$  con un número infinito de ceros (pero, por supuesto, esto está simplificado solo para darte una idea de qué son los números duales). Un número dual se representa en la memoria como un par de flotadores. Por ejemplo,  $42 + 24\epsilon$  está representado por el par  $(42.0, 24.0)$ .

Los números duales se pueden sumar, multiplicar, etc., como se muestra en la Ecuación B-3.

#### Ecuación B-3. Algunas operaciones con números duales

$$\begin{aligned} \lambda(a + b\epsilon) &= \lambda a + \lambda b\epsilon \\ (a + b\epsilon) + (c + d\epsilon) &= (a + c) + (b + d)\epsilon \\ (a + b\epsilon) \times (c + d\epsilon) &= ac + (ad + bc)\epsilon + (bd)\epsilon^2 \end{aligned}$$

$\epsilon^2 = ac + (ad + bc)\epsilon + (bd)\epsilon$

Lo más importante es que se puede demostrar que  $h(a + b\epsilon) = h(a) + b \times h'(a)\epsilon$ , por lo que calcular  $h(a + \epsilon)$  da tanto  $h(a)$  como la derivada  $h'(a)$  en un solo disparo. La Figura B-2 muestra que la derivada parcial de  $f(x, y)$  con respecto a  $x$

en  $x = 3$  e  $y = 4$  (que escribiré  $\partial f / \partial x (3, 4)$ ) se puede calcular usando números duales. Todo lo que necesitamos hacer es calcular  $f(3 + \varepsilon, 4)$ ; esto generará un número dual cuyo primer componente es igual a  $f(3, 4)$  y cuyo segundo componente es igual a  $\partial f / \partial x (3, 4)$ .

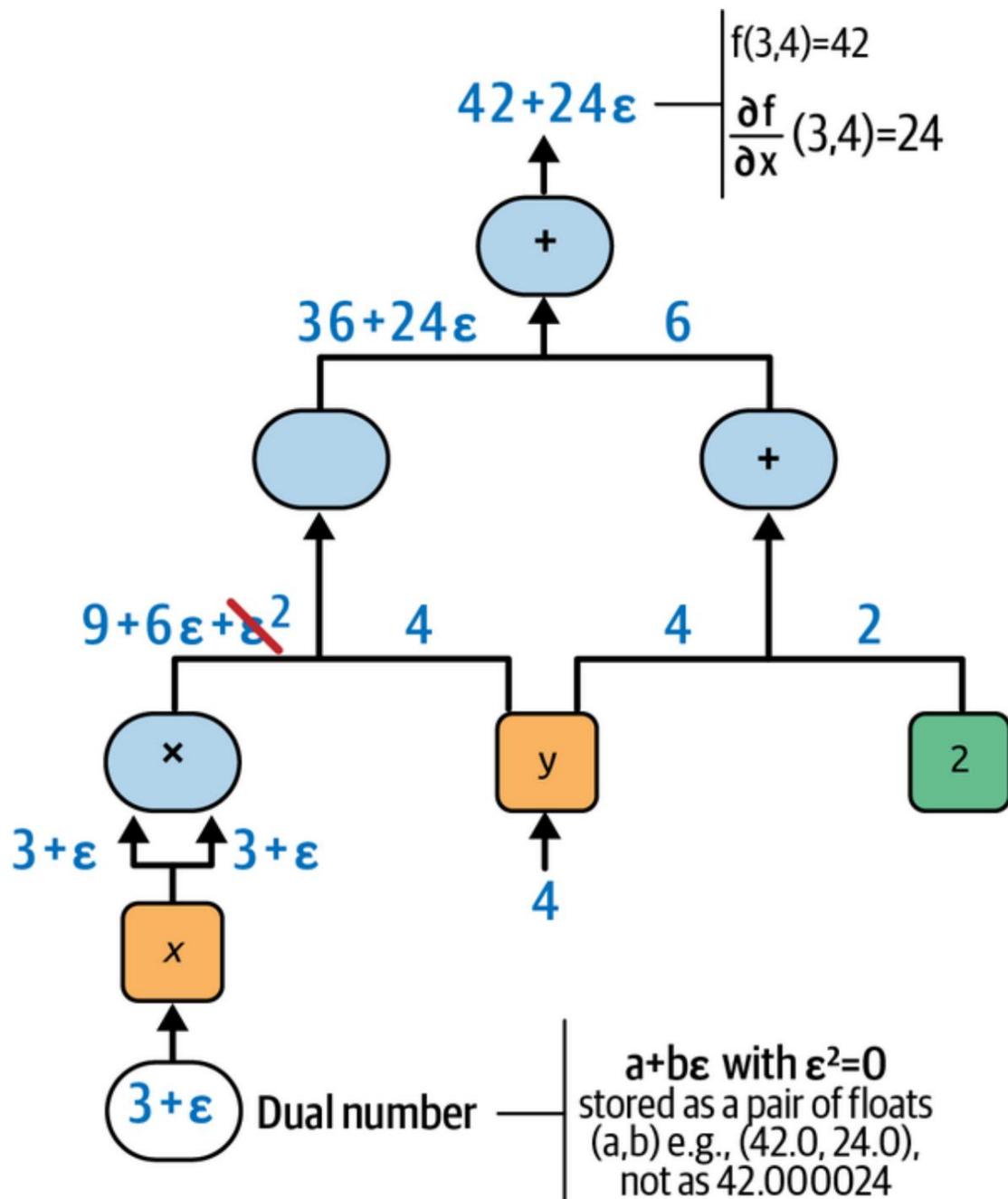


Figura B-2. Autodiff en modo directo usando números duales

Para calcular  $\partial f / \partial y (3, 4)$  tendríamos que repasar la gráfica nuevamente, pero esta vez con  $x = 3$  e  $y = 4 + \varepsilon$ .

Por lo tanto, la diferenciación automática en modo directo es mucho más precisa que la aproximación en diferencias finitas, pero adolece del mismo defecto importante, al menos cuando hay muchas entradas y pocas salidas (como es el caso cuando se trata de redes neuronales): si hubiera 1.000 parámetros, se necesitarían 1000 pasadas por el gráfico para calcular todas las derivadas parciales. Aquí es donde brilla la diferenciación automática en modo inverso: puede calcularlos todos en solo dos pasadas por el gráfico. Veamos cómo.

### Diferencial automático en modo inverso

La diferenciación automática en modo inverso es la solución implementada por TensorFlow. Primero recorre el gráfico en dirección directa (es decir, desde las entradas a la salida) para calcular el valor de cada nodo. Luego realiza una segunda pasada, esta vez en dirección inversa (es decir, desde la salida a las entradas), para calcular todas las derivadas parciales. El nombre "modo inverso" proviene de este segundo paso por el gráfico, donde los gradientes fluyen en dirección inversa. [La Figura B-3](#) representa la segunda pasada. Durante la primera pasada, se calcularon todos los valores de los nodos, comenzando desde  $x = 3$  e  $y = 4$ . Puede ver esos valores en la parte inferior derecha de cada nodo (por ejemplo,  $x \times x = 9$ ). Los nodos están etiquetados de  $n$  a  $n$  para mayor claridad. El nodo de salida es  $n : f(3, 4) = n = 42$ .

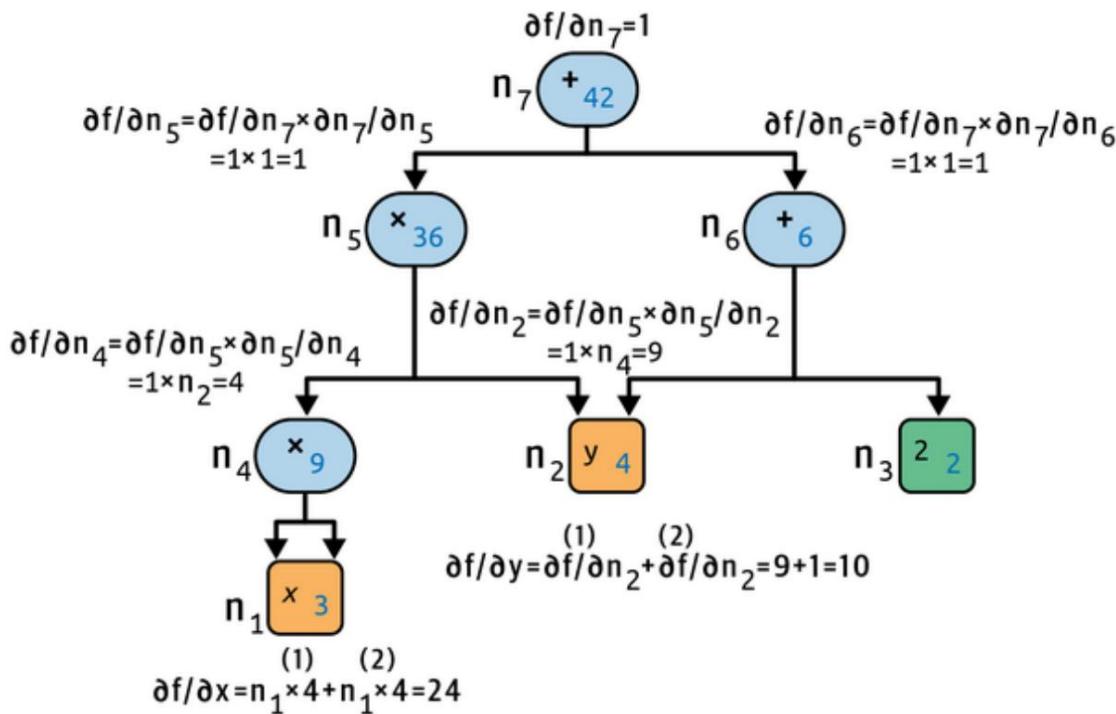


Figura B-3. Autodiff en modo inverso

La idea es ir descendiendo gradualmente por la gráfica, calculando el parcial derivada de  $f(x, y)$  con respecto a cada nodo consecutivo, hasta llegar a los nodos variables. Para esto, la diferenciación automática en modo inverso depende en gran medida de la regla de la cadena, que se muestra en la Ecuación B-4.

Ecuación B-4. Cadena de reglas

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial n_i} \times \frac{\partial n_i}{\partial x}$$

Dado que  $n$  es el nodo de salida,  $f = n$  entonces  $\frac{\partial f}{\partial n} = 1$ .

Sigamos por el gráfico hasta  $n$ : ¿cuánto varía  $f$  cuando  $n$  ?  
varía? La respuesta es  $\frac{\partial f}{\partial n} = \frac{\partial f}{\partial n} \times \frac{\partial n}{\partial n}$ . eso ya lo sabemos  
 $\frac{\partial f}{\partial n} = 1$ , entonces todo lo que necesitamos es  $\frac{\partial f}{\partial n}$ . Dado que  $n$  simplemente realiza la suma  $n_1 + n_2 + n_3 + n_4$ , encontramos que  $\frac{\partial f}{\partial n} = 1$ , entonces  $\frac{\partial f}{\partial n} = 1 \times 1 = 1$ .

Ahora podemos proceder al nodo  $n$ : ¿cuánto varía  $f$  cuando varía  $n$  ?

La respuesta es  $\frac{\partial f}{\partial n} = \frac{\partial f}{\partial n_1} \times \frac{\partial n_1}{\partial n} + \frac{\partial f}{\partial n_2} \times \frac{\partial n_2}{\partial n} + \frac{\partial f}{\partial n_3} \times \frac{\partial n_3}{\partial n} + \frac{\partial f}{\partial n_4} \times \frac{\partial n_4}{\partial n}$ . Como  $n = n_1 + n_2 + n_3 + n_4$  encontramos que  $\frac{\partial f}{\partial n} = 1 \times 4 = 4$ .

El proceso continúa hasta llegar al final del gráfico. En ese punto habremos calculado todas las derivadas parciales de  $f(x, y)$  en el punto  $x = 3$  e  $y = 4$ . En este ejemplo, encontramos  $\partial f / \partial x = 24$  y  $\partial f / \partial y = 10$ . ¡Suena bien!

La diferenciación automática en modo inverso es una técnica muy poderosa y precisa, especialmente cuando hay muchas entradas y pocas salidas, ya que requiere solo un paso directo más un paso inverso por salida para calcular todas las derivadas parciales de todas las salidas con respecto a todas las entradas. . Cuando entrenamos redes neuronales, generalmente queremos minimizar la pérdida, por lo que hay una única salida (la pérdida) y, por lo tanto, solo se necesitan dos pasadas por el gráfico para calcular los gradientes. La diferenciación automática en modo inverso también puede manejar funciones que no son completamente diferenciables, siempre y cuando le pida que calcule las derivadas parciales en puntos que son diferenciables.

En [la Figura B-3](#), los resultados numéricos se calculan sobre la marcha, en cada nodo. Sin embargo, eso no es exactamente lo que hace TensorFlow: en lugar de eso, crea un nuevo gráfico de cálculo. En otras palabras, implementa una diferenciación automática simbólica en modo inverso. De esta manera, el gráfico de cálculo para calcular los gradientes de pérdida con respecto a todos los parámetros de la red neuronal solo necesita generarse una vez y luego puede ejecutarse una y otra vez, siempre que el optimizador necesite calcular los gradientes. Además, esto hace posible calcular derivadas de orden superior si es necesario.

CONSEJO

Si alguna vez desea implementar un nuevo tipo de operación TensorFlow de bajo nivel en C++ y desea que sea compatible con autodiff, deberá proporcionar una función que devuelva las derivadas parciales de las salidas de la función con respecto a sus entradas. . Por ejemplo, suponga que implementa una función que calcula el cuadrado de su entrada:  $f(x) = x^2$ . En ese caso, necesitarías proporcionar la función derivada correspondiente:  $f'(x) = 2x$ .

# Apéndice C. Estructuras de datos especiales

---

En este apéndice, echaremos un vistazo rápido a las estructuras de datos admitidas por TensorFlow, más allá de los tensores flotantes o enteros normales.

Esto incluye cadenas, tensores irregulares, tensores dispersos, matrices de tensores, conjuntos y colas.

## Instrumentos de cuerda

Los tensores pueden contener cadenas de bytes, lo que resulta útil en particular para el procesamiento del lenguaje natural (consulte el [Capítulo 16](#)):

```
>>> tf.constant(b"holo mundo") <tf.Tensor:  
forma=(), dtype=cadena, numpy=b'holo mundo'>
```

Si intentas construir un tensor con una cadena Unicode, TensorFlow lo codifica automáticamente en UTF-8:

```
>>> tf.constant("café") <tf.Tensor:  
forma=(), dtype=cadena, numpy=b'caf\xc3\xa9'>
```

También es posible crear tensores que representen cadenas Unicode.

Simplemente cree una matriz de números enteros de 32 bits, cada uno de los cuales represente un único punto de código Unicode.<sup>1</sup>

```
>>> u = tf.constant([ord(c) para c en "café"]) >>> u  
  
<tf.Tensor: forma=(4,), [...], numpy=matriz([ 99, 97, 102, 233], dtype=int32)>
```

## NOTA

En tensores de tipo `tf.string`, la longitud de la cuerda no forma parte de la forma del tensor. En otras palabras, las cadenas se consideran valores atómicos. Sin embargo, en un tensor de cadena Unicode (es decir, un tensor `int32`), la longitud de la cadena es parte de la forma del tensor.

El paquete `tf.strings` contiene varias funciones para manipular tensores de cadena, como `length()` para contar el número de bytes en una cadena de bytes (o el número de puntos de código si configura `unit="UTF8_CHAR"`), `unicode_encode()` para convertir un tensor de cadena Unicode (es decir, tensor `int32`) a un tensor de cadena de bytes, y `unicode_decode()` para hacer lo inverso:

```
>>> b = tf.strings.unicode_encode(u, "UTF-8") >>> b
<tf.Tensor: forma=(), dtype=cadena, numpy=b'caf\xcc\x9a9'> >>> tf.strings.length(b,
unit="UTF8_CHAR") <tf.Tensor: forma=( ), dtype=int32,
numpy=4> >>> tf.strings.unicode_decode(b, "UTF-8") <tf.Tensor:
forma=(4,), [...], numpy=array([ 99, 97, 102, 233], dtype=int32)>
```

También puedes manipular tensores que contengan varias cadenas:

```
>>> p = tf.constant(["Café", "Coffee", "caffè", " "]) >>> tf.strings.length(p,
unit="UTF8_CHAR") <tf.Tensor: forma =(4,), dtype=int32,
numpy=array([4, 6, 5, 2], dtype=int32)> >>> r = tf.strings.unicode_decode (p, "UTF8") >>>
r
<tf.RaggedTensor [[67, 97, 102, 233], [67, 111, 102, 102, 101, 101], [99, 97, 102, 102,
232], [21654, 21857]]>
```

Observe que las cadenas decodificadas se almacenan en un `RaggedTensor`. ¿Qué es eso?

## Tensores irregulares Un

tensor irregular es un tipo especial de tensor que representa una lista de matrices de diferentes tamaños. De manera más general, es un tensor con una o más dimensiones irregulares, es decir, dimensiones cuyos cortes pueden tener diferentes longitudes. En el tensor irregular `r`, la segunda dimensión es una dimensión irregular. En todos los tensores irregulares, la primera dimensión es siempre una dimensión regular (también llamada dimensión uniforme).

Todos los elementos del tensor irregular `r` son tensores regulares. Por ejemplo, veamos el segundo elemento del tensor irregular:

```
>>> r[1]
<tf.Tensor: [...], numpy=array([ 67, 111, 102, 102, 101, 101], dtype=int32)>
```

El paquete `tf.ragged` contiene varias funciones para crear y manipular tensores irregulares. Creemos un segundo tensor irregular usando `tf.ragged.constant()` y concatenémoslo con el primer tensor irregular, a lo largo del eje 0:

```
>>> r2 = tf.ragged.constant([[65, 66], [], [67]]) >>> tf.concat([r, r2], eje=0)
<tf.RaggedTensor [[67, 97, 102, 233], [67,
111, 102, 102, 101, 101], [99, 97, 102, 102, 232], [21654, 21857], [65, 66], [], [67]]>
```

El resultado no es demasiado sorprendente: los tensores de `r2` se añadieron después de los tensores de `r` a lo largo del eje 0. Pero, ¿qué pasa si concatenamos `r` y otro tensor irregular a lo largo del eje 1?

```
>>> r3 = tf.ragged.constant([[68, 69, 70], [71], [], [72, 73]]) >>> print(tf.concat([r, r3], eje=1))
<tf.RaggedTensor [[67, 97, 102, 233, 68, 69, 70], [67,
111, 102, 102, 101, 101, 71], [99, 97, 102, 102, 232], [21654, 21857, 72, 73]]>
```

Esta vez, observe que el tensor  $i$  en  $r^t$  y el tensor  $i$  en  $r3$  eran concatenados. Eso es más inusual, ya que todos estos tensores puede tener diferentes longitudes.

Si llamas al método `to_tensor()`, el tensor irregular se vuelve convertido a un tensor regular, llenando tensores más cortos con ceros para obtener tensores de longitudes iguales (puede cambiar el valor predeterminado estableciendo el argumento `valor_predeterminado`):

Muchas operaciones de TF soportan tensores irregulares. Para obtener la lista completa, consulte el documentoación de la clase `tf.RaggedTensor`.

# Tensores dispersos

TensorFlow también puede representar eficientemente tensores dispersos (es decir, tensores que contienen principalmente ceros). Simplemente crea un `tf.SparseTensor`, especificando los índices y valores de los elementos distintos de cero y el la forma del tensor. Los índices deben enumerarse en “orden de lectura” (de de izquierda a derecha y de arriba a abajo). Si no está seguro, simplemente use `tf.sparse.reordenar()`. Puedes convertir un tensor disperso en un tensor denso (es decir, un tensor regular) usando `tf.sparse.to_dense()`:

```
>>> s = tf.SparseTensor(indices=[[0, 1], [1, 0], [2, 3]],  
...                                     valores=[1., 2., 3.],  
...                                     forma_densa=[3, 4])  
...  
>>> tf.sparse.to_dense(s)  
<tf.Tensor: forma=(3, 4), dtype=float32, numpy=
```

```
matriz([[0., 1., 0., 0.], [2., 0., 0., 0.],
       [0., 0., 0., 3.]],
      dtype=float32) >
```

Tenga en cuenta que los tensores dispersos no admiten tantas operaciones como los tensores densos. Por ejemplo, puedes multiplicar un tensor disperso por cualquier valor escalar y obtendrás un nuevo tensor disperso, pero no puedes agregar un valor escalar a un tensor disperso, ya que esto no devolvería un tensor disperso:

```
>>> s * 42.0
<tensorflow.python.framework.sparse_tensor.SparseTensor en 0x7f84a6749f10>
>>> s + 42.0
[...] TypeError: tipos de operandos no admitidos para +: 'SparseTensor' y 'float'
```

## Matrices tensoriales

Un `tf.TensorArray` representa una lista de tensores. Esto puede resultar útil en modelos dinámicos que contienen bucles, para acumular resultados y luego calcular algunas estadísticas. Puede leer o escribir tensores en cualquier ubicación de la matriz:

```
matriz = tf.TensorArray(dtype=tf.float32, tamaño=3) matriz = array.write(0,
tf.constant([1., 2.])) matriz = array.write(1, tf.constant([ 3., 10.])) array
= array.write(2, tf.constant([5., 7.])) tensor1 = array.read(1) # => devuelve
(jy pone a cero!) tf.constant ([3., 10.])
```

De forma predeterminada, leer un elemento también lo reemplaza con un tensor de la misma forma pero lleno de ceros. Puedes configurar `clear_after_read` en `False` si no quieres esto.

### ADVERTENCIA

Cuando escribe en la matriz, debe asignar la salida nuevamente a la matriz, como se muestra en este ejemplo de código. Si no lo hace, aunque su código funcionará bien en modo ansioso, se interrumpirá en modo gráfico (estos modos se analizan en el [Capítulo 12](#)).

De forma predeterminada, un `TensorArray` tiene un tamaño fijo que se establece en el momento de su creación. Alternativamente, puede establecer `size=0` y `dynamic_size=True` para permitir que la matriz crezca automáticamente cuando sea necesario. Sin embargo, esto afectará el rendimiento, por lo que si conoce el tamaño de antemano, es mejor utilizar una matriz de tamaño fijo. También debe especificar el tipo `d` y todos los elementos deben tener la misma forma que el primero escrito en la matriz.

Puedes apilar todos los elementos en un tensor normal llamando al método `stack()`:

```
>>> array.stack()  
<tf.Tensor: forma=(3, 2), dtype=float32, numpy= matriz([[1., 2.], [0., 0.],  
[5., 7.]], dtype=float32)>
```

### Conjuntos

TensorFlow admite conjuntos de números enteros o cadenas (pero no flotantes). Representa conjuntos que utilizan tensores regulares. Por ejemplo, el conjunto `{1, 5, 9}` simplemente se representa como el tensor `[[1, 5, 9]]`. Tenga en cuenta que el tensor debe tener al menos dos dimensiones y los conjuntos deben estar en la última dimensión. Por ejemplo, `[[1, 5, 9], [2, 5, 11]]` es un tensor que contiene dos conjuntos independientes: `{1, 5, 9}` y `{2, 5, 11}`.

El paquete `tf.sets` contiene varias funciones para manipular conjuntos. Por ejemplo, creamos dos conjuntos y calculemos su unión (la

El resultado es un tensor disperso, por lo que llamamos a `to_dense()` para mostrarlo):

```
>>> a = tf.constant([[1, 5, 9]]) >>> b = tf.constant([[5,
6, 9, 11]]) >>> u = tf.sets.union (a, b)

>>> tu
<tensorflow.python.framework.sparse_tensor.SparseTensor en 0x132b60d30>

>>> tf.sparse.to_dense(u) <tf.Tensor:
[...], numpy=array([[ 1, 5, 6, 9, 11]], dtype=int32)>
```

También puedes calcular la unión de varios pares de conjuntos simultáneamente. Si algunos conjuntos son más cortos que otros, debes rellenarlos con un valor de relleno, como 0:

```
>>> a = tf.constant([[1, 5, 9], [10, 0, 0]]) >>> b = tf.constant([[5, 6, 9,
11], [13, 0, 0, 0]]) >>> u = tf.sets.union(a, b) >>> tf.sparse.to_dense(u) <tf.Tensor:
[...] numpy=array([[ 1, 5, 6, 9, 11],
[0, 10, 13, 0, 0]],

dtipo=int32)>
```

Si prefiere utilizar un valor de relleno diferente, como -1, debe establecer `default_value=-1` (o su valor preferido) al llamar a `to_dense()`.

#### ADVERTENCIA

El valor predeterminado `default_value` es 0, por lo que cuando se trata de conjuntos de cadenas, debe establecer este parámetro (por ejemplo, en una cadena vacía).

Otras funciones disponibles en `tf.sets` incluyen `diferencia()`, `intersección()` y `tamaño()`, que se explican por sí mismas. Si desea comprobar si un conjunto contiene o no algunos valores dados, debe

Puede calcular la intersección de ese conjunto y los valores. Si desea agregar algunos valores a un conjunto, puede calcular la unión del conjunto y los valores.

## Colas

Una cola es una estructura de datos a la que puede enviar registros de datos y luego extraerlos. TensorFlow implementa varios tipos de colas en el paquete `tf.queue`. Solían ser muy importantes al implementar procesos eficientes de carga y preprocesamiento de datos, pero la API `tf.data` esencialmente los ha vuelto inútiles (excepto quizás en algunos casos raros) porque es mucho más simple de usar y proporciona todas las herramientas que necesita para construir ductos eficientes. Sin embargo, en aras de la exhaustividad, echemos un vistazo rápido a ellos.

El tipo de cola más simple es la cola de primero en entrar, primero en salir (FIFO). Para construirlo, debe especificar la cantidad máxima de registros que puede contener. Además, cada registro es una tupla de tensores, por lo que debes especificar el tipo de cada tensor y, opcionalmente, sus formas. Por ejemplo, el siguiente ejemplo de código crea una cola FIFO con un máximo de tres registros, cada uno de los cuales contiene una tupla con un entero de 32 bits y una cadena. Luego envía dos registros, mira el tamaño (que es 2 en este momento) y extrae un registro:

```
>>> q = tf.queue.FIFOQueue(3, [tf.int32, tf.string], formas=[(), ()]) >>>
q.enqueue([10,
b"windy"]) >>> q.enqueue([15, b"sunny"])
>>> q.size() <tf.Tensor: forma=(),
dtype=int32,
numpy=2> >>> q.dequeue() [ <tf.Tensor: forma=(), dtype=int32,
numpy=10>, <tf.Tensor:
forma=(), dtype=cadena, numpy=b'windy'>]
```

También es posible poner en cola y quitar de cola varios registros a la vez usando `enqueue_many()` y `dequeue_many()` (para usar

dequeue\_many(), debes especificar el argumento de formas cuando creas la cola, como hicimos anteriormente):

```
>>> q.enqueue_many([[13, 16], [b'nublado', b'lluvioso]]) >>> q.dequeue_many(3)
[<tf.Tensor: [...], numpy=
 matriz([15, 13, 16], dtype=int32)>, <tf.Tensor: [...],
 numpy=array([b'soleado',
 b'nublado',
 b'lluvioso'], dtype=object)>]
```

Otros tipos de colas incluyen:

#### RellenoFIFOCola

Igual que FIFOQueue, pero su método dequeue\_many() admite la eliminación de la cola de varios registros de diferentes formas. Rellena automáticamente los registros más cortos para garantizar que todos los registros del lote tengan la misma forma.

#### Cola de prioridad

Una cola que retira los registros en orden de prioridad. La prioridad debe ser un número entero de 64 bits incluido como primer elemento de cada registro. Sorprendentemente, los registros con menor prioridad serán retirados primero de la cola. Los registros con la misma prioridad se quitarán de la cola en orden FIFO.

#### Cola aleatoria aleatoria

Una cola cuyos registros se retiran de la cola en orden aleatorio. Esto fue útil para implementar un búfer aleatorio antes de que existiera tf.data.

Si una cola ya está llena e intenta poner en cola otro registro, el método enqueue\*() se congelará hasta que otro hilo retire un registro de la cola. De manera similar, si una cola está vacía e intenta sacar de la cola un registro, el método dequeue\*() se congelará hasta que otro subproceso envíe los registros a la cola.

---

1 Si no está familiarizado con los puntos del código Unicode, consulte  
<https://homl.info/unicode>.

# Apéndice D. Gráficos de TensorFlow

---

En este apéndice, exploraremos las gráficas generadas por funciones TF (consulte el Capítulo 12).

## Funciones TF y funciones concretas

Las funciones TF son polimórficas, lo que significa que admiten entradas de diferentes tipos (y formas). Por ejemplo, considere la siguiente función `tf_cube()`:

```
@tf.function def  
tf_cube(x): devuelve  
    x ** 3
```

Cada vez que llamas a una función TF con una nueva combinación de tipos o formas de entrada, genera una nueva función concreta, con su propio gráfico especializado para esta combinación en particular. Esta combinación de tipos y formas de argumentos se denomina firma de entrada. Si llama a la función TF con una firma de entrada que ya ha visto antes, reutilizará la función concreta que generó anteriormente. Por ejemplo, si llama a `tf_cube(tf.constant(3.0))`, la función TF reutilizará la misma función concreta que usó para

`tf_cube(tf.constant(2.0))` (para tensores escalares float32). Pero generará una nueva función concreta si llama a

`tf_cube(tf.constant([2.0]))` o

`tf_cube(tf.constant([3.0]))` (para tensores de forma float32 [1]), y otra más para `tf_cube (tf.constant([[1.0, 2.0], [3.0, 4.0]]))` (para tensores float32 de forma [2, 2]). Puede obtener la función concreta para una combinación particular de entradas llamando

el método `get_concrete_function()` de la función TF. Luego se puede llamar como una función normal, pero solo admitirá una firma de entrada (en este ejemplo, tensores escalares `float32`):

```
>>> función_concreta =
tf_cube.get_concrete_function(tf.constant(2.0)) >>>
función_concreta
<Función_concreta tf_cube(x) en 0x7F84411F4250> >>>
función_concreta(tf.constant(2.0)) <tf.Tensor:
forma=() , dtype=float32, numpy=8.0>
```

**La Figura D-1** muestra la función TF `tf_cube()`, después de llamar a `tf_cube(2)` y `tf_cube(tf.constant(2.0))`: se generaron dos funciones concretas, una para cada firma, cada una con su propio gráfico de funciones optimizado ( `FuncGraph` ) y su propia definición de función (`FunctionDef`). La definición de una función apunta a las partes del gráfico que corresponden a las entradas y salidas de la función. En cada `FuncGraph`, los nodos (óvalos) representan operaciones (por ejemplo, potencia, constantes o marcadores de posición para argumentos como `x`), mientras que los bordes (las flechas sólidas entre las operaciones) representan los tensores que fluirán a través del gráfico. La función concreta de la izquierda está especializada para  $x=2$ , por lo que TensorFlow logró simplificarla para generar solo 8 todo el tiempo (tenga en cuenta que la definición de la función ni siquiera tiene una entrada). La función concreta de la derecha está especializada para tensores escalares `float32` y no se puede simplificar. Si llamamos a `tf_cube(tf.constant(5.0))`, se llamará a la segunda función concreta, la operación de marcador de posición para `x` generará 5.0, luego la operación de potencia c

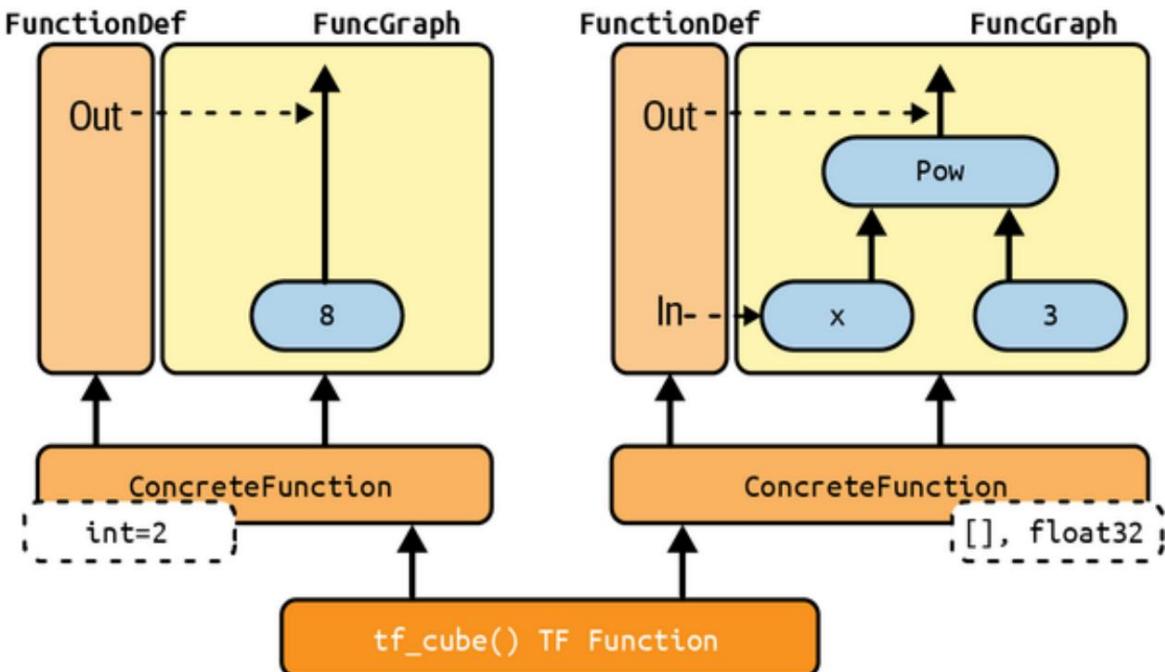


Figura D-1. La función TF `tf_cube()` , con sus `ConcreteFunctions` y sus `FuncGraphs`

Los tensores en estos gráficos son tensores simbólicos, lo que significa que no tienen un valor real, solo un tipo de datos, una forma y un nombre. Representan los tensores futuros que fluirán a través del gráfico una vez que se introduzca un valor real en el marcador de posición `x` y se ejecute el gráfico. Los tensores simbólicos permiten especificar de antemano cómo conectar las operaciones y también permiten que TensorFlow infiera recursivamente los tipos de datos y las formas de todos los tensores, dados los tipos de datos y las formas de sus entradas.

Ahora sigamos echando un vistazo más allá del capó y veamos cómo acceder a definiciones de funciones y gráficos de funciones y cómo explorar las operaciones y los tensores de un gráfico.

## Explorando definiciones de funciones y gráficos

Puede acceder al gráfico de cálculo de una función concreta utilizando el atributo `gráfico` y obtener la lista de sus operaciones llamando al método `get_operatives()` del gráfico:

```
>>> concrete_function.graph
<tensorflow.python.framework.func_graph.FuncGraph en 0x7f84411f4790>

>>> operaciones = función_concreta.graph.get_operaciones() >>>

operaciones [<tf.Operación 'x' tipo=Placeholder>,
  <tf.Operación 'pow/y' tipo=Const>, <tf.Operación
  'pow' tipo=Pow>, <tf.Operación 'Identidad'
  tipo=Identidad>]
```

En este ejemplo, la primera operación representa el argumento de entrada x (se llama marcador de posición), la segunda "operación" representa la constante 3, la tercera operación representa la operación de potencia (\*\*\*) y la operación final representa la salida de esta función (es una operación de identidad, lo que significa que no hará más que copiar la salida de la operación de energía<sup>1</sup>). Cada operación tiene una lista de tensores de entrada y salida a los que puede acceder fácilmente utilizando los atributos de entrada y salida de la operación. Por ejemplo, obtengamos la lista de entradas y salidas de la operación de energía:

```
>>> pow_op = operaciones[2]
>>> lista(pow_op.inputs)
[<tf.Tensor 'x:0' forma=() dtype=float32>, <tf.Tensor 'pow/y:0'
 forma= () dtype=float32>]
>>> pow_op.outputs
[<tf.Tensor 'pow:0' forma=() dtype=float32>]
```

Este gráfico de cálculo se representa en [la Figura D-2.](#)

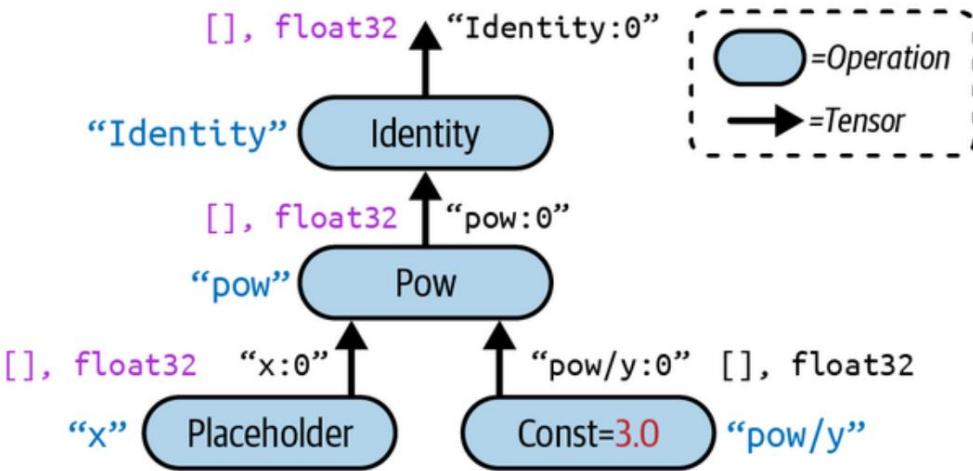


Figura D-2. Ejemplo de un gráfico de cálculo

Tenga en cuenta que cada operación tiene un nombre. El nombre predeterminado es el nombre de la operación (p. ej., "pow"), pero puede definirlo manualmente al llamar a la operación (p. ej., `tf.pow(x, 3, name="other_name")`). Si ya existe un nombre, TensorFlow agrega automáticamente un índice único (por ejemplo, "pow\_1", "pow\_2", etc.). Cada tensor también tiene un nombre único: siempre es el nombre de la operación que genera este tensor, más :0 si es la primera salida de la operación, o :1 si es la segunda salida, y así sucesivamente. Puedes buscar una operación o un tensor por nombre usando los métodos `get_operation_by_name()` o `get_tensor_by_name()` del gráfico:

```
>>> función_concreta.graph.get_operation_by_name('x') <tf.Operation
'x' tipo=Marcador de posición>
>>>
concrete_function.graph.get_tensor_by_name('Identidad:0') <tf.Tensor
'Identidad:0' forma=() dtype=float32>
```

La función concreta también contiene la definición de la función (representada como un búfer de protocolo), que incluye la firma de la función. Esta firma permite que la función concreta sepa qué marcadores de posición alimentar con los valores de entrada y qué tensores devolver:

```
>>> función_concreta.función_def.firma nombre:  

"__inference_tf_cube_3515903" input_arg { nombre:  

"x"  
  

    tipo: DT_FLOAT  
  

} salida_arg  

{ nombre: "identidad"  

    tipo: DT_FLOAT  

}
```

Ahora veamos más de cerca el rastreo.

## Una mirada más cercana al rastreo

Modifiquemos la función `tf_cube()` para imprimir su entrada:

```
@tf.function def  

tf_cube(x): print(f"x =  

{x}") devuelve x ** 3
```

Ahora llamémoslo:

```
>>> resultado = tf_cube(tf.constant(2.0)) x = Tensor("x:0",  

forma=(), dtype=float32) >>> resultado  
  

<tf.Tensor: forma=(), dtype=float32, numpy=8.0>
```

El resultado se ve bien, pero mira lo que se imprimió: ¡`x` es un tensor simbólico! Tiene una forma y un tipo de datos, pero no tiene valor. Además tiene un nombre ("`x:0`"). Esto se debe a que la función `print()` no es una operación de TensorFlow, por lo que solo se ejecutará cuando se rastree la función de Python, lo que ocurre en modo gráfico, con los argumentos reemplazados por tensores simbólicos (mismo tipo y forma, pero sin valor). Dado que la función `print()` no fue capturada en el gráfico, las próximas veces que llamemos a `tf_cube()` con tensores escalares `float32`, no se imprime nada:

```
>>> resultado = tf_cube(tf.constant(3.0)) >>>
resultado = tf_cube(tf.constant(4.0))
```

Pero si llamamos a `tf_cube()` con un tensor de un tipo o forma diferente, o con un nuevo valor de Python, la función será trazada nuevamente, por lo que se llamará a la función `print()`:

```
>>> resultado = tf_cube(2) # nuevo valor de Python: ¡rastreo! x = 2

>>> resultado = tf_cube(3) # nuevo valor de Python: ¡rastreo! x = 3

>>> resultado = tf_cube(tf.constant([[1., 2.]])) # nueva forma: ¡rastro!

x = Tensor("x:0", forma=(1, 2), dtype=float32) >>> resultado =
tf_cube(tf.constant([[3., 4.], [5., 6.]]) ) # nueva forma: ¡rastro! x = Tensor("x:0",
forma=(Ninguno, 2),
dtype=float32) >>> resultado = tf_cube(tf.constant([[7., 8.], [9., 10.]]) )
# misma forma: sin rastro
```

#### ADVERTENCIA

Si su función tiene efectos secundarios de Python (por ejemplo, guarda algunos registros en el disco), tenga en cuenta que este código solo se ejecutará cuando se rastree la función (es decir, cada vez que se llame a la función TF con una nueva firma de entrada). Es mejor asumir que la función se puede rastrear (o no) cada vez que se llama a la función TF.

En algunos casos, es posible que desee restringir una función TF a una firma de entrada específica. Por ejemplo, suponga que sabe que solo llamará a una función TF con lotes de imágenes de  $28 \times 28$  píxeles, pero los lotes tendrán tamaños muy diferentes. Es posible que no desee que TensorFlow genere una función concreta diferente para cada tamaño de lote, o que cuente con él para determinar por sí solo cuándo usar Ninguno. En este caso, puede especificar la firma de entrada de esta manera:

```
@tf.function(input_signature=[tf.TensorSpec([None, 28, 28], tf.float32)]) def
encogimiento(images):
    devuelve imágenes[:, ::2, ::2]
        # elimina la mitad de las filas y
    columnas
```

Esta función TF aceptará cualquier tensor float32 de forma [\* , 28, 28] y reutilizará la misma función concreta cada vez:

```
img_batch_1 = tf.random.uniform(shape=[100, 28, 28]) img_batch_2 =
tf.random.uniform(shape=[50, 28, 28]) preprocessed_images =
encogimiento(img_batch_1) # funciona bien, rastrea la función

preprocessed_images = encogimiento(img_batch_2) # funciona bien, la misma función
concreta
```

Sin embargo, si intenta llamar a esta función TF con un valor de Python o un tensor de un tipo o forma de datos inesperados, obtendrá una excepción:

```
img_batch_3 = tf.random.uniform(shape=[2, 2, 2]) preprocessed_images =
encogimiento(img_batch_3) # ¡ValueError!
Entradas incompatibles
```

## Uso de AutoGraph para capturar el flujo de control

Si su función contiene un bucle for simple, ¿qué espera que suceda? Por ejemplo, escribamos una función que sumará 10 a su entrada, simplemente sumando 1 10 veces:

```
@tf.function def
add_10(x): para i en
    el rango(10): x += 1

    volver x
```

Funciona bien, pero cuando miramos su gráfico, encontramos que no contiene un bucle: ¡solo contiene 10 operaciones de suma!

```

>>> add_10(tf.constant(0))
<tf.Tensor: forma=(), dtype=int32, numpy=15>
>>>
add_10.get_concrete_function(tf.constant(0)).graph.get_operations() [<tf.Operation
'x'
type=Placeholder>, [...], <tf.Operation 'add' type=AddV2>,
[ ...], <tf.Operación 'add_1' tipo=AddV2>, [...],
<tf.Operación 'add_2' tipo=AddV2>, [...], [...] <tf.Operación
'add_9' tipo=AddV2>, [...], <tf.Operation 'Identidad'
tipo=Identidad>]

```

En realidad, esto tiene sentido: cuando se trazó la función, el ciclo se ejecutó 10 veces, por lo que la operación  $x += 1$  se ejecutó 10 veces y, como estaba en modo gráfico, registró esta operación 10 veces en el gráfico. Puedes pensar en este bucle for como un bucle "estático" que se desenrolla cuando se crea el gráfico.

Si desea que el gráfico contenga un bucle "dinámico" (es decir, uno que se ejecute cuando se ejecuta el gráfico), puede crear uno manualmente usando la operación `tf.while_loop()`, pero no es muy intuitivo (consulte la sección " Uso de AutoGraph para capturar el flujo de control" del cuaderno del Capítulo 12 para ver un ejemplo). En cambio, es mucho más sencillo utilizar la función AutoGraph de TensorFlow , que se analiza en [el Capítulo 12](#). En realidad, AutoGraph está activado de forma predeterminada (si alguna vez necesita desactivarlo, puede pasar `autograph=False` a `tf.function()`). Entonces, si está activado, ¿por qué no capturó el bucle for en la función `add_10()`? Solo captura bucles for que iteran sobre tensores de objetos `tf.data.Dataset`, por lo que debes usar `tf.range()`, no `range()`. Esto es para darle la opción:

- Si usa `range()`, el bucle for será estático, lo que significa que solo se ejecutará cuando se rastree la función. El bucle se "desenrollará" en un conjunto de operaciones para cada iteración, como vimos.
- Si usa `tf.range()`, el bucle será dinámico, lo que significa que se incluirá en el gráfico mismo (pero no se ejecutará durante

rastreo).

Veamos el gráfico que se genera si simplemente reemplazamos range() con tf.range() en la función add\_10():

```
>>>
add_10.get_concrete_function(tf.constant(0)).graph.get_operations() [<tf.Operation 'x'
tipo=Placeholder>, [...], <tf.Operation 'while' tipo = StatelessWhile>, [ ...]]
```

Como puede ver, el gráfico ahora contiene una operación de bucle While, como si hubiéramos llamado a la función tf. while\_loop().

## Manejo de variables y otros recursos en TF Funciones

En TensorFlow, las variables y otros objetos con estado, como colas o conjuntos de datos, se denominan recursos. Las funciones TF las tratan con especial cuidado: cualquier operación que lea o actualice un recurso se considera con estado, y las funciones TF garantizan que las operaciones con estado se ejecuten en el orden en que aparecen (a diferencia de las operaciones sin estado, que pueden ejecutarse en paralelo, por lo que sus el orden de ejecución no está garantizado). Además, cuando pasa un recurso como argumento a una función TF, se pasa por referencia, por lo que la función puede modificarlo. Por ejemplo:

```
contador = tf.Variable(0)

@tf.function def
incremento(contador, c=1):
    devolver contador.assign_add(c)

incremento(contador) # el contador ahora es igual a 1 increment(contador) #
el contador ahora es igual a 2
```

Si echa un vistazo a la definición de la función, el primer argumento está marcado como un recurso:

```
>>> function_def =
incremento.get_concrete_function(contador).function_def >>>
function_def.signature.input_arg[0]
nombre: "contador"
tipo: DT_RESOURCE
```

También es posible utilizar una variable tf.Variable definida fuera de la función, sin pasarla explícitamente como argumento:

```
contador = tf.Variable(0)

@tf.function def
incremento(c=1): devuelve
    contador.assign_add(c)
```

La función TF tratará esto como un primer argumento implícito, por lo que en realidad terminará con la misma firma (excepto por el nombre del argumento). Sin embargo, el uso de variables globales puede volverse complicado rápidamente, por lo que generalmente debes incluir las variables (y otros recursos) dentro de las clases. La buena noticia es que @tf.function también funciona bien con métodos:

Contador [de clase](#) :

```
def __init__(self): self.contador
    = tf.Variable(0)

@tf.function def
incremento(self, c=1): devuelve
    self.counter.assign_add(c)
```

### ADVERTENCIA

No utilice `=`, `+=`, `-=` ni ningún otro operador de asignación de Python con variables TF. En su lugar, debe utilizar los métodos `asignar()`, `asignar_add()` o `asignar_sub()`. Si intenta utilizar un operador de asignación de Python, obtendrá una excepción cuando llame al método.

Un buen ejemplo de este enfoque orientado a objetos es, por supuesto, Keras. Veamos cómo usar las funciones TF con Keras.

## Usar funciones TF con Keras (o no)

De forma predeterminada, cualquier función, capa o modelo personalizado que utilice con Keras se convertirá automáticamente en una función TF; ¡No necesitas hacer nada en absoluto! Sin embargo, en algunos casos es posible que desee desactivar esta conversión automática; por ejemplo, si su código personalizado no se puede convertir en una función TF o si simplemente desea depurar su código (lo cual es mucho más fácil en el modo ansioso). Para hacer esto, simplemente puedes pasar `dinámico=True` al crear el modelo o cualquiera de sus capas:

```
modelo = MiModelo(dinámico=Verdadero)
```

Si su modelo o capa personalizada siempre será dinámica, puede llamar al constructor de la clase base con `dinámico=True`:

```
clase MyDense(tf.keras.layers.Layer):
    def __init__(self, unidades, **kwargs):
        super().__init__(dynamic=True, **kwargs) [...]
```

Alternativamente, puedes pasar `run_eagerly=True` al llamar al método `compile()`:

```
model.compile(loss=my_mse, optimizador="nadam", métricas= [my_mae],  
run_eagerly=True)
```

Ahora ya sabe cómo las funciones TF manejan el polimorfismo (con múltiples funciones concretas), cómo se generan automáticamente los gráficos usando AutoGraph y el rastreo, cómo se ven los gráficos, cómo explorar sus operaciones simbólicas y tensores, cómo manejar variables y recursos, y cómo utilizarlos. TF funciona con Keras.

---

**1** Puede ignorarlo con seguridad; solo está aquí por razones técnicas, para garantizar que Las funciones TF no filtran estructuras internas.

**2** Un formato binario popular que se analiza en [el Capítulo 13](#).

# Índice

---

## Símbolos

Operador @ (multiplicación de matrices), la ecuación normal

$\beta$  (impulso), **impulso**

Valor  $\gamma$  (gamma), núcleo RBF gaussiano

$\epsilon$  (tolerancia), **descenso de gradiente por lotes**, clases SVM y  
Complejidad computacional

$\epsilon$  política codiciosa, **Políticas de Exploración**

$\epsilon$  vecindario, **DBSCAN**

$\epsilon$  sensible, **regresión SVM**

Prueba  $\chi^2$ , **Hiperparámetros de regularización**

$\ell$  norma, **seleccione una medida de desempeño**

## A

Experimentos A/B, **entrenamiento e implementación de modelos TensorFlow en Escala**

k-medias aceleradas, **k-medias aceleradas y k-medias de mini lotes**

álgebra lineal acelerada (XLA), **funciones y gráficos de TensorFlow**

medida de rendimiento de precisión, [¿Qué es el aprendizaje automático?](#),

[Medición de la precisión mediante validación cruzada](#)

[ACF \(función de autocorrelación\)](#), [la familia de modelos ARMA](#)

[ventaja de la acción](#), [aprendizaje por refuerzo](#), [Evaluación de acciones: el Problema de asignación de crédito](#)

[Potenciales de acción \(AP\)](#), [neuronas biológicas](#)

[acciones, en aprendizaje por refuerzo](#), [Aprender a optimizar las recompensas](#),

[Evaluación de acciones: el problema de la asignación de créditos](#)

[Acciones: El problema de la cesión de crédito](#)

[funciones de activación](#), [El Perceptrón](#), [El Perceptrón Multicapa](#) y

[Retropropagación: el perceptrón multicapa y la retropropagación](#)

- para la capa Conv2D, [Implementación de capas convolucionales con Keras](#)
- modelos personalizados, [funciones de activación personalizadas](#), [inicializadores](#), [Regularizadores y restricciones](#)
- ELU, [ELU y SELU-GELU](#), Swish y Mish
- GELU, [GELU](#), Swish y Mish-GELU, Swish y Mish
- tangente hiperbólica (htan), [el perceptrón multicapa y Retropropagación](#), luchando contra el problema de los gradientes inestables
- hiperparámetros, [tasa de aprendizaje](#), [tamaño de lote](#) y otros [Hiperparámetros](#)
- parámetros de inicialización, [Glorot y He Inicialización](#)
- LeakyReLU, [Leaky ReLU-Leaky ReLU](#)
- Mish, [GELU](#), Swish y Mish
- PReLU, [ReLU con fugas](#)

- ReLU (ver ReLU)
- RReLU, **ReLU con fugas**
- SELU, **ELU** y SELU, Abandono
- sigmoide, **estimación de probabilidades**, el perceptrón multicapa y  
Propagación hacia atrás, los problemas de gradientes que desaparecen/explotan,  
Codificadores automáticos dispersos
- SiLU, **GELU**, Swish y Mish
- softmax, **regresión de Softmax**, clasificación de MLP, creación del modelo utilizando  
la API secuencial, una red de codificador-decodificador para traducción automática  
neuronal
- softplus, **MLP de regresión**
- Swish, **GELU**, Swish y Mish

aprendizaje activo, uso de agrupaciones para el aprendizaje semisupervisado

actor-crítico, descripción general de algunos algoritmos RL populares

clase real, **Matrices de confusión**

probabilidades reales versus estimadas, **la curva ROC**

AdaBoost, AdaBoost-AdaBoost

AdaGrad, **AdaGrad**

Optimización de Adam, **Adam**, regularización  $\ell_1$  y  $\ell_2$

AdaMax, **AdaMax**

Regularización AdamW, **AdamW**,  $\ell_1$  y  $\ell_2$

impulso adaptativo (AdaBoost), impulso-AdaBoost

normalización de instancias adaptativas (AdaIN), **StyleGANs**

algoritmos de tasa de aprendizaje adaptativo, AdaGrad-AdamW

estimación del momento adaptativo (Adam), Adam

atención aditiva, Mecanismos de Atención

ventaja actor-crítico (A2C), descripción general de algunas RL populares

Algoritmos

aprendizaje adversario, segmentación semántica, codificadores automáticos, GAN y modelos de difusión

transformaciones afines, StyleGAN

función de afinidad, algoritmos de agrupamiento: k-means y DBSCAN

propagación por afinidad, otros algoritmos de agrupación

agentes, aprendizaje por refuerzo, aprendizaje por refuerzo, aprender a Optimizar recompensas, gradientes de políticas, Q-Learning

agrupamiento aglomerativo, otros algoritmos de agrupamiento

Criterio de información de Akaike (AIC), selección del número de conglomerados

AlexNet, AlexNet

algoritmos, preparar datos para, preparar los datos para la máquina

Algoritmos de aprendizaje -canalizaciones de transformación

modelo de alineación, Mecanismos de Atención

Algoritmo AllReduce, paralelismo de datos utilizando la estrategia reflejada

abandono alfa, abandono

AlphaGo, Aprendizaje por refuerzo, Aprendizaje por refuerzo,

Descripción general de algunos algoritmos RL populares

Priores de anclaje, solo miras una vez

ANN (ver redes neuronales artificiales)

detección de anomalías, Ejemplos de Aplicaciones, Aprendizaje no supervisado

- agrupación para algoritmos de agrupación: k-means y DBSCAN
- GMM, uso de mezclas gaussianas para la detección de anomalías  
Mezclas gaussianas para la detección de anomalías
- bosque de aislamiento, Otros algoritmos para anomalías y novedades  
Detección

AP (precisión media), sólo miras una vez

AP (potenciales de acción), neuronas biológicas

área bajo la curva (AUC), la curva ROC

argmax(), Regresión Softmax

Modelo ARIMA, La familia de modelos ARMA-La familia de modelos ARMA

Familia de modelos ARMA, La familia de modelos ARMA-El modelo ARMA  
Familia

Redes neuronales artificiales (RNA), Introducción a las redes neuronales artificiales

Redes con tasa de aprendizaje de Keras, tamaño de lote y otros

Hiperparámetros

- codificadores automáticos (ver codificadores automáticos)
- retropropagación, el perceptrón multicapa y  
Propagación hacia atrás: el perceptrón multicapa y  
Propagación hacia atrás
- neuronas biológicas como fondo para, Neuronas biológicas-  
Neuronas biológicas
- Evolución de las neuronas biológicas a las artificiales.  
Neuronas biológicas a artificiales

- ajuste fino de hiperparámetros, [ajuste fino de la red neuronal](#)  
[Hiperparámetros: tasa de aprendizaje, tamaño de lote y otros](#)  
[Hiperparámetros](#)
- Implementación de MLP con Keras, [Implementación de MLP con Keras-Usando TensorBoard para visualización](#)
- Cálculos lógicos con neuronas, [Cálculos lógicos con Neuronas-](#) [Computaciones Lógicas con Neuronas](#)
- perceptrones (ver perceptrones multicapa)
- Políticas de aprendizaje por refuerzo, [Políticas de redes neuronales.](#)

neurona artificial, [Cálculos Lógicos con Neuronas](#)

aprendizaje de reglas de asociación, [aprendizaje no supervisado](#)

suposiciones, comprobar la construcción del modelo, discrepancia de datos, comprobar la [Suposiciones](#)

ventaja asincrónica actor-crítico (A3C), [descripción general de algunos Algoritmos RL populares](#)

programación de pandillas asíncrona, [saturación de ancho de banda](#)

Actualizaciones asincrónicas, con parámetros centralizados, [Actualizaciones asincrónicas.](#)

à-trous capa convolucional, [segmentación semántica](#)

mecanismos de atención, [procesamiento del lenguaje natural con RNN y Atención](#), [Mecanismos de Atención -Atención multicabezal , Visión transformadores](#)

- (ver también modelos de transformadores)

atributos, [eche un vistazo rápido a la estructura de datos: eche un vistazo rápido Mira la estructura de datos](#)

- categórico, [Manejo de texto y atributos categóricos](#), [Manejo Texto y atributos categóricos](#)
- combinaciones de, [Experimente con combinaciones de atributos](#)-[Experimente con combinaciones de atributos](#)
- preprocessado, [eche un vistazo rápido a la estructura de datos](#)
- objetivo, [eche un vistazo rápido a la estructura de datos](#)
- aprendizaje no supervisado, [aprendizaje supervisado](#)

AUC (área bajo la curva), [la curva ROC](#)

series de tiempo autocorrelacionadas, [Predicción de una serie de tiempo](#)

función de autocorrelación (ACF), [la familia de modelos ARMA](#)

autodiff (diferenciación automática), [Autodiff-Reverse-Mode Autodiff](#)

- para calcular gradientes, [Calcular gradientes utilizando Autodiff](#)-[Calcular gradientes usando Autodiff](#)
- aproximación en diferencias finitas, [aproximación en diferencias finitas](#)
- modo de avance, Autodiff [en modo de avance](#) -[Modo de avance Autodiff](#)
- diferenciación manual, [diferenciación manual](#)
- modo inverso, Autodiff [en modo inverso](#) -[Autodiff en modo inverso](#)

codificadores automáticos, [codificadores automáticos](#), GAN y modelos de difusión-

[Generando imágenes MNIST de moda](#)

- convolucional, Autocodificadores [convolucionales](#) -[convolucionales](#) [codificadores automáticos](#)
- Eliminación de ruido, [Codificadores automáticos con eliminación de ruido](#)- [Codificadores automáticos con eliminación de ruido](#)
- representaciones de datos eficientes, [Representaciones de datos eficientes](#)-[Representaciones de datos eficientes](#)

- Autocodificadores convolucionales sobrecompletos
- PCA con codificador automático lineal incompleto, Realización de PCA con un codificador automático lineal incompleto-Realización de PCA con un codificador automático lineal incompleto
- dispersos, codificadores automáticos dispersos- codificadores automáticos dispersos
- apilados, Autocodificadores apilados: entrenamiento de un codificador automático a la vez Tiempo
- entrenar uno a la vez, entrenar un codificador automático a la vez- Entrenamiento de un codificador automático a la vez
- Representaciones de datos incompletas y eficientes
- variacionales, Autocodificadores variacionales - Autocodificadores variacionales

Autógrafo, autógrafo y calco

AutoGraph, uso de AutoGraph para capturar el flujo de control

Servicio AutoML, ajuste fino de hiperparámetros de redes neuronales,

Ajuste de hiperparámetros en Vertex AI

modelo de media móvil integrada autorregresiva (ARIMA), el

Familia de modelos ARMA: la familia de modelos ARMA

modelo autorregresivo, la familia de modelos ARMA

familia de modelos de media móvil autorregresiva (ARMA), ARMA

Familia de modelos: la familia de modelos ARMA

tarea auxiliar, preentrenamiento en, Preentrenamiento en una tarea auxiliar

desviación absoluta promedio, Seleccione una medida de desempeño

capa de agrupación promedio, Implementación de capas de agrupación con Keras

precisión promedio (AP), solo miras una vez

## B

columna vertebral, modelo, GoogLeNet

retropropagación, el perceptrón multicapa y retropropagación-

El perceptrón multicapa y la retropropagación, el

Problemas de gradientes que desaparecen o explotan, cálculo de gradientes

Usando Autodiff, redes generativas adversarias

retropropagación a través del tiempo (BPTT), entrenamiento de RNN

embolsado (agregación de arranque), embolsado y pegado aleatorio

Parches y subespacios aleatorios

Atención de Bahdanau, Mecanismos de Atención

reducción iterativa equilibrada y agrupación mediante jerarquías

(BIRCH), otros algoritmos de agrupación

saturación de ancho de banda, saturación de ancho de banda - saturación de ancho de banda

BaseEstimator, transformadores personalizados

descenso de gradiente por lotes, descenso de gradiente por lotes- gradiente por lotes

Descenso, regresión de crestas

aprendizaje por lotes, aprendizaje por lotes- aprendizaje por lotes

normalización por lotes (BN), normalización por lotes : implementación de la normalización por lotes con Keras, lucha contra el problema de los gradientes inestables

predicciones por lotes, implementación de una nueva versión del modelo, creación de un

Servicio de predicción en Vertex AI, ejecución de trabajos de predicción por lotes en IA de vértice

Mezclas bayesianas gaussianas, modelos de mezclas bayesianas gaussianas

Criterio de información bayesiano (BIC), Selección del número de

Clústeres

búsqueda de haz, **Búsqueda de haz- Búsqueda de haz**

Ecuación de optimización de Bellman, **procesos de decisión de Markov**

sesgo, **curvas de aprendizaje**

parcialidad y equidad, transformadores de PNL, **transformadores de Hugging Face**  
**Biblioteca**

constante del término de sesgo, **regresión lineal**, ecuación normal

Compensación entre sesgo y varianza, **Curvas de aprendizaje**

BIC (criterio de información bayesiano), **Selección del número de Clústeres**

capa recurrente bidireccional, **RNN bidireccionales**

clasificadores binarios, **Entrenamiento de un clasificador binario**, Regresión logística

logaritmo binario, **Complejidad Computacional**

árboles binarios, **hacer predicciones**, otros algoritmos de agrupación

redes neuronales biológicas (BNN), **neuronas biológicas -biológicas**  
**Neuronas**

BIRCH (reducción iterativa equilibrada y agrupación mediante jerarquías), **otros algoritmos de agrupación**

modelos de caja negra, **Haciendo predicciones**

mezclando, apilando, **apilando**

BN (normalización por lotes), **normalización por lotes**, lucha contra lo inestable  
**Problema de gradientes**

BNNs (redes neuronales biológicas), **Neuronas Biológicas -Biológicas**  
**Neuronas**

aumento, **aumento de gradiente basado en histograma**

agregación de arranque (embolsado), [embolsado y pegado aleatorio](#)

**Parches y subespacios aleatorios**

arranque, [embolsado y pegado](#)

capas de cuello de botella, [GoogLeNet](#)

cuadros delimitadores, identificación de imágenes, [clasificación y localización](#)-

**Sólo miras una vez**

BPTT (propagación hacia atrás a través del tiempo), [entrenamiento de RNN](#)

agrupar una característica, [escalar y transformar características](#)

codificación de pares de bytes (BPE), [análisis de sentimiento](#)

## C

Conjunto de datos de precios de vivienda de California, [trabajando con datos reales](#):

**verifique los supuestos**

devoluciones de llamada, [Uso de devoluciones de llamada-Uso de devoluciones de llamada](#)

Algoritmo CART (Árbol de Clasificación y Regresión), [Creación](#)

**Predicciones, algoritmo de entrenamiento CART, regresión**

Olvido catastrófico, [Implementación de Deep Q-Learning](#)

atributos categóricos, [Manejo de texto y atributos categóricos](#),

[Manejo de texto y atributos categóricos](#)

características categóricas, codificación, [codificación de capa hash](#)

**Funciones categóricas mediante incrustaciones**

Capa CategoryEncoding, [La capa CategoryEncoding](#)

modelo causal, [Pronóstico utilizando un modelo de secuencia a secuencia](#),

**RNN bidireccionales**

parámetros centralizados, **Paralelismo de datos con parámetros centralizados-Actualizaciones asincrónicas**

centroide, clúster, **Algoritmos de agrupación: k-means y DBSCAN, k-means, Métodos de inicialización del algoritmo k-means -centroide**

regla de la cadena, **el perceptrón multicapa y la retropropagación**

incitaciones en cadena de pensamiento, **Una avalancha de modelos de transformadores**

**ChainClassifier, Clasificación multietiqueta**

Encadenamiento de transformaciones, **Encadenamiento de transformaciones-Encadenamiento Transformaciones**

Modelo char-RNN, generación de texto de Shakespeare utilizando un **Carácter RNN-RNN con estado**

chatbot o asistente personal, **Ejemplos de Aplicaciones**

`check_estimator()`, **Transformadores personalizados**

prueba de chi-cuadrado ( $\chi^2$ ), **hiperparámetros de regularización**

clasificación, **Clasificación- Clasificación multisalida**

- ejemplos de aplicación de, **Ejemplos de Aplicaciones-Ejemplos de Aplicaciones**
- clasificador binario, **Entrenamiento de un clasificador binario, Logística Regresión**
- CNN, **clasificación y localización: solo se mira una vez**
- análisis de errores, **Análisis de errores- Análisis de errores**
- margen duro, **clasificación de margen suave, bajo el capó de Clasificadores SVM lineales**
- Clasificadores de votación dura, **Clasificadores de votación**

- imagen (ver imágenes)
- regresión logística (ver regresión logística)
- MLP para, Clasificación MLP-Clasificación MLP
- Conjunto de datos MNIST, MNIST-MNIST
- multiclase, Clasificación multiclase- Clasificación multiclase,  
Clasificación MLP-Clasificación MLP
- multilabel, Clasificación multietiqueta- Clasificación multietiqueta
- multisalida, Clasificación multisalida- Clasificación multisalida
- medidas de desempeño, Medidas de desempeño-La República de China  
Curva
- y regresión, Aprendizaje supervisado, Clasificación multisalida
- margen suave, Clasificación de margen suave- Clasificación de margen suave
- regresión softmax, regresión softmax- regresión softmax
- SVM (ver máquinas de vectores de soporte)
- texto, Análisis de sentimiento : reutilización de incrustaciones previamente entrenadas y  
Modelos de lenguaje
- clasificadores de votación, Clasificadores de votación- Clasificadores de votación

Algoritmo de árbol de clasificación y regresión (CART), creación

Predicciones, algoritmo de entrenamiento CART, regresión

clon(), parada anticipada

ecuación/solución en forma cerrada, modelos de entrenamiento, lo normal

Ecuación, regresión de crestas, entrenamiento y función de costos

Implementación de plataforma en la nube con Vertex AI, creación de una predicción

Servicio en Vertex AI: creación de un servicio de predicción en Vertex AI

algoritmos de agrupamiento, Ejemplos de aplicaciones, Aprendizaje no supervisado,

Técnicas de aprendizaje no supervisado-Otros agrupamientos

## Algoritmos

- propagación por afinidad, otros algoritmos de agrupación
- agrupamiento aglomerativo, otros algoritmos de agrupamiento
- aplicaciones para algoritmos de agrupamiento: k-means y DBSCAN-  
Algoritmos de agrupamiento: k-means y DBSCAN
- BIRCH, otros algoritmos de agrupación
- DBSCAN, DBSCAN-DBSCAN
- GMM, mezclas gaussianas: otros algoritmos para anomalías y  
Detección de novedades
- segmentación de imágenes, algoritmos de agrupamiento: k-medias y  
DBSCAN, uso de agrupación en clústeres para la segmentación de imágenes  
Agrupación para segmentación de imágenes
- k-medias (ver algoritmo k-medias)
- cambio medio, otros algoritmos de agrupación
- responsabilidades de los clusters, por ejemplo, mezclas gaussianas
- Aprendizaje semisupervisado con uso de agrupación en clústeres para semi-supervisado.  
Aprendizaje supervisado mediante agrupación en clústeres para semisupervisado  
Aprendiendo
- agrupamiento espectral, otros algoritmos de agrupamiento

CNN (ver redes neuronales convolucionales)

Colab, Ejecutar los ejemplos de código con Google Colab-Ejecutar los ejemplos de código  
con Google Colab

canales de color, apilamiento de múltiples mapas de funciones

segmentación de color, imágenes, uso de agrupación en clústeres para imágenes

## Segmentación

vectores de columna, regresión lineal

ColumnTransformer, canalizaciones de transformación

modelos complejos con API funcional, Construcción de modelos complejos usando la API funcional-Construcción de modelos complejos usando la API funcional API

escalado compuesto, Otras arquitecturas destacadas

archivos TFRecord comprimidos, Archivos TFRecord comprimidos

compresión y descompresión, PCA, PCA para compresión-PCA para compresión

gráficos de cálculo, un recorrido rápido por TensorFlow

complejidad computacional

- DBSCAN, DBSCAN
- árboles de decisión, Complejidad Computacional
- Modelo de mezcla gaussiana, Mezclas gaussianas
- aumento de gradiente basado en histograma, gradiente basado en histograma Impulsando
- Algoritmo k-medias, El algoritmo k-medias
- Ecuación normal, Complejidad Computacional
- y clases SVM, Clases SVM y Complejidad Computacional

atención concatenativa, Mecanismos de Atención

función concreta, funciones TF y funciones concretas: exploración

Definiciones de funciones y gráficos

- GAN condicional, GAN convolucionales profundas
- probabilidad condicional, búsqueda por haz
- intervalo de confianza, evalúe su sistema en el equipo de prueba
- matriz de confusión (CM), Matrices de confusión -Matrices de confusión,
- Análisis de errores- Análisis de errores
- ConfusionMatrixDisplay, análisis de errores
- Conexionismo, de las neuronas biológicas a las artificiales
- optimización restringida, bajo el capó de los clasificadores lineales SVM
- restricciones, modelos personalizados, funciones de activación personalizadas,
- Inicializadores, regularizadores y restricciones
- tasa de convergencia, descenso de gradiente por lotes
- función convexa, descenso de gradiente
- optimización cuadrática convexa, bajo el capó de SVM lineal
- Clasificadores
- Kernels de convolución (kernels), Filtros, Arquitecturas CNN
- redes neuronales convolucionales (CNN), visión profunda por computadora usando
- Redes neuronales convolucionales : segmentación semántica
  - arquitecturas, Arquitecturas CNN: elegir la CNN adecuada  
Arquitectura
  - codificadores automáticos, codificadores automáticos convolucionales -convolucionales  
codificadores automáticos
  - clasificación y localización, Clasificación y Localización-  
Sólo miras una vez
  - capas convolucionales, Capas Convolucionales -Memoria  
Requisitos, Segmentación Semántica-Semántica

Segmentación, uso de capas convolucionales 1D para procesar secuencias,

WaveNet-WaveNet, enmascaramiento

- Evolución de la visión profunda por computadora mediante el uso de neuronas convolucionales.  
Redes
- GAN, GAN convolucionales profundas : GAN convolucionales profundas
- detección de objetos, Detección de objetos: solo miras una vez
- seguimiento de objetos, seguimiento de objetos
- capas de agrupación, Capas de agrupación : implementación de capas de agrupación con Keras
- modelos previamente entrenados de Keras, uso de modelos previamente entrenados de Keras utiliza modelos preentrenados de Keras
- ResNet-34 CNN usando Keras, implementando una ResNet-34 CNN  
Usando Keras
- segmentación semántica, Ejemplos de Aplicaciones, Uso  
Agrupación para segmentación de imágenes, segmentación semántica.  
Segmentación semántica
- división entre dispositivos, paralelismo de modelos
- transferencia de modelos preentrenados de aprendizaje, Modelos preentrenados para  
Transferencia de aprendizaje: modelos previamente entrenados para transferencia de aprendizaje
- U-Net, modelos de difusión
- y transformadores de visión, Transformadores de visión
- arquitectura de la corteza visual, La arquitectura de la corteza visual
- WaveNet, WaveNet-WaveNet

copy.deepcopy(), parada anticipada

instancia central, DBSCAN

coeficiente de correlación, [buscar correlaciones](#)-[buscar correlaciones](#)

función de costo, [aprendizaje basado en modelos](#) y un flujo de trabajo típico de aprendizaje automático, seleccione una medida de rendimiento

- en AdaBoost, [AdaBoost](#)
- en codificadores automáticos, [Realización de PCA con un lineal insuficientemente completo](#) codificador automático
- en el modelo de predicción del cuadro delimitador, [clasificación](#) y [Localización](#)
- en el algoritmo de entrenamiento CART, [El algoritmo de entrenamiento CART](#), [Regresión](#)
- en red elástica, [Regresión neta elástica](#)
- en descenso de gradiente, [modelos de entrenamiento](#), [descenso de gradiente](#) Descenso de gradiente, estocástico Descenso de gradiente-estocástico Descenso de gradiente, los problemas de gradientes que desaparecen o explotan
- en regresión de lazo, [Regresión de lazo](#)- [Regresión de lazo](#)
- en gradiente acelerado de Nesterov, [Nesterov acelerado](#) Degradado
- en regresión lineal, [Regresión lineal](#)
- en regresión logística, [Capacitación](#) y [Función de Costo-Capacitación](#) y [Función de costo](#)
- en optimización del impulso, [Momentum](#)
- en regresión de crestas, [regresión de crestas](#)
- en codificadores automáticos variacionales, [Autocodificadores variacionales](#)

problema de asignación de crédito, [Evaluación de acciones](#): el crédito

Acciones de evaluación de problemas de asignación : la asignación de crédito

## Problema

entropía cruzada, regresión Softmax

validación cruzada, ajuste de hiperparámetros y selección de modelo,

Mejor evaluación mediante validación cruzada: mejor evaluación mediante

Validación cruzada, evaluación de su sistema en el equipo de prueba, medición

Precisión mediante validación cruzada: medición de precisión mediante validación cruzada

Validación, Curvas de Aprendizaje -Curvas de Aprendizaje

`cross_val_predict()`, Matrices de confusión, Compensación precisión/recuperación, Curva ROC,

Análisis de errores, Apilamiento

`cross_val_score()`, Mejor evaluación mediante validación cruzada,

Medición de la precisión mediante validación cruzada

Biblioteca CUDA, cómo obtener su propia GPU

exploración basada en la curiosidad, descripción general de algunas RL populares

Algoritmos

aprendizaje curricular, descripción general de algunos algoritmos RL populares

modelos personalizados y algoritmos de entrenamiento, personalización de modelos y

Algoritmos de entrenamiento: bucles de entrenamiento personalizados

- funciones de activación, funciones de activación personalizadas, inicializadores, Regularizadores y restricciones
- autodiff para calcular gradientes, Calcular gradientes usando Autodiff: cálculo de gradientes mediante Autodiff
- restricciones, funciones de activación personalizadas, inicializadores, Regularizadores y restricciones
- inicializadores, funciones de activación personalizadas, inicializadores, Regularizadores y restricciones
- capas, Capas personalizadas - Capas personalizadas

- funciones de pérdida, funciones de pérdida personalizadas, pérdidas y métricas  
Basado en el modelo Internos: pérdidas y métricas basadas en el modelo Internos
- métricas, Métricas personalizadas - Métricas personalizadas, Pérdidas y Métricas  
Basado en el modelo Internos: pérdidas y métricas basadas en el modelo Internos
- modelos, Modelos personalizados - Modelos personalizados
- regularizadores, funciones de activación personalizadas, inicializadores, Regularizadores y restricciones
- guardar y cargar modelos, Guardar y cargar modelos que  
Contiene componentes personalizados: guarda y carga modelos que  
Contiene componentes personalizados
- bucles de entrenamiento, bucles de entrenamiento personalizados -bucle de entrenamiento personalizados

transformadores personalizados, Transformadores personalizados- Transformadores personalizados  
segmentación de clientes, Algoritmos de agrupamiento: k-medias y DBSCAN

D

DALL-E, Transformadores de Visión  
datos

- descargar, guardar los cambios de su código y guardar sus datos  
Tu código cambia y tus datos
- representaciones de datos eficientes, Representaciones de datos eficientes-Representaciones de datos eficientes
- poner en cola y quitar la cola, Colas

- encontrar correlaciones en, [buscar correlaciones](#)- [buscar Correlaciones](#)
- hacer suposiciones sobre [la falta de coincidencia de datos](#)
- sobreajuste (ver sobreajuste de datos)
- preparación para algoritmos de ML, [preparación de datos para la máquina](#)  
[Algoritmos de aprendizaje](#) -canalizaciones de transformación
- preparación para modelos ML, [preparación de datos para máquina](#)  
[Modelos de aprendizaje](#) : [preparación de datos para el aprendizaje automático](#)  
[Modelos](#)
- preprocesamiento (ver carga y preprocesamiento de datos)
- barajar de, [barajar los datos](#)
- datos de prueba (ver conjunto de prueba)
- series de tiempo (ver datos de series de tiempo)
- datos de entrenamiento (ver conjunto de entrenamiento)
- Desajuste de, [Capacitar y Evaluar en el Conjunto de Capacitación](#), Aprendizaje  
[Curvas: curvas de aprendizaje](#), núcleo polinómico
- eficacia irrazonable, [cantidad insuficiente de formación](#)  
[Datos](#)
- visualizando (ver visualización)
- trabajar con datos reales, [Trabajar con datos reales](#)-[Trabajar con Datos reales](#)

análisis de datos, agrupamiento para, [algoritmos de agrupamiento](#): k-medias y  
[DBSCAN](#)

aumento de datos, [Ejercicios](#), [AlexNet](#), Modelos previamente entrenados para  
[Transferir aprendizaje](#)

limpieza de datos, [Limpiar los datos-Limpiar los datos](#)  
deriva de datos, [aprendizaje por lotes](#)  
minería de datos, [¿por qué utilizar el aprendizaje automático?](#)  
discrepancia de datos, [discrepancia de datos](#)  
paralelismo de datos, [Paralelismo de datos- saturación](#) de ancho de banda  
canalización de datos, [encuadre el problema](#)  
Sesgo de espionaje de datos, [creación de un conjunto de pruebas.](#)  
estructura de datos, [eche un vistazo rápido a la estructura de datos: eche un vistazo rápido](#)  
[Mire la estructura de datos, estructuras de datos especiales: colas](#)  
Transformadores de imágenes con eficiencia de datos (DeiT), [Vision Transformers](#)  
DataFrame, [limpieza de datos, manejo de texto y categórico](#)  
Atributos, [escalamiento de características y transformación](#)  
Búsqueda de datos, [otros recursos](#)  
Biblioteca de conjuntos de datos, [Proyecto de conjuntos de datos de TensorFlow](#)  
DBSCAN, DBSCAN-DBSCAN  
DCGAN (GANS convolucionales profundos), [GAN convolucionales profundas-GAN convolucionales profundas](#)  
DDPM (modelo probabilístico de difusión de eliminación de ruido), [codificadores automáticos, GAN y modelos de difusión, modelos de difusión -modelos de difusión](#)  
DDQN (duelo en duelo con DQN), [duelo con DQN](#)  
límites de decisión, [límites de decisión- límites de decisión,](#)  
Regresión Softmax, realización de predicciones, sensibilidad al eje  
Orientación, k-medias

función de decisión, la compensación entre precisión y recuperación, bajo el capó de Clasificadores SVM lineales: bajo el capó de los clasificadores SVM lineales

tocones de decisión, AdaBoost

umbral de decisión, el equilibrio precisión/recuperación: el Compensación entre precisión y recuperación

árboles de decisión, árboles de decisión : los árboles de decisión tienen una alta variación, Aprendizaje conjunto y bosques aleatorios

- (ver también aprendizaje en conjunto)
- embolsar y pegar, Embolsar y pegar en Scikit-Learn
- Algoritmo de entrenamiento CART, El algoritmo de entrenamiento CART
- estimaciones de probabilidad de clase, Estimación de probabilidades de clase
- complejidad computacional, Complejidad Computacional
- y límites de decisión, hacer predicciones
- Medidas de impureza o entropía de GINI, ¿impureza o entropía de Gini?
- alta variación con, los árboles de decisión tienen una alta variación
- predicciones, Hacer predicciones: el algoritmo de entrenamiento CART
- tareas de regresión, Regresión-Regresión
- hiperparámetros de regularización, Regularización  
Hiperparámetros- Hiperparámetros de regularización
- sensibilidad a la orientación del eje, Sensibilidad a la orientación del eje
- entrenar y visualizar, entrenar y visualizar una decisión  
Entrenamiento de árboles y visualización de un árbol de decisiones
- en la capacitación del modelo, Capacitar y Evaluar en el Conjunto de Capacitación  
Mejor evaluación mediante validación cruzada

DecisionTreeClassifier, [entrenamiento y visualización de un árbol de decisión, ¿ impureza o entropía de Gini?](#), hiperparámetros de regularización, sensibilidad a la orientación del eje, bosques aleatorios

DecisionTreeRegressor, [entrenar y evaluar en el conjunto de entrenamiento, Árboles de decisión, regresión, aumento de gradiente](#)

`decision_function()`, el equilibrio entre precisión y recuperación

capa de deconvolución, [segmentación semántica](#)

codificadores automáticos profundos (ver codificadores automáticos apilados)

GANS convolucionales profundos (DCGAN), [GAN convolucionales profundos-GAN convolucionales profundas](#)

proceso gaussiano profundo, abandono de Monte Carlo (MC)

aprendizaje profundo, [El tsunami del aprendizaje automático](#)

- (ver también redes neuronales profundas; aprendizaje por refuerzo)

redes neuronales profundas (DNN), [entrenamiento de redes neuronales profundas-Resumen y directrices prácticas](#)

- CNN (ver redes neuronales convolucionales)
- configuración predeterminada, [resumen y pautas prácticas](#)
- optimizadores más rápidos para, [Optimizadores más rápidos-AdamW](#)
- programación de tasa de aprendizaje, [programación de tasa de aprendizaje-aprendizaje Programación de tarifas](#)
- MLP (ver perceptrones multicapa)
- Regularización, [evitando el sobreajuste mediante la regularización-Max-Regularización de normas](#)
- reutilizar capas previamente entrenadas, [Reutilizar capas previamente entrenadas-Entrenamiento previo en una tarea auxiliar](#)

- RNN (ver redes neuronales recurrentes)
- y transferencia de aprendizaje, Aprendizaje autosupervisado
- gradientes inestables, Los gradientes que desaparecen/explotan Problemas
- gradientes que desaparecen y explotan, The Vanishing/Exploding Problemas de gradientes : recorte de gradientes

neuroevolución profunda, ajuste fino de los hiperparámetros de la red neuronal

Q-learning profundo, Q-Learning aproximado y Q-Learning profundo-Duelo DQN

redes Q profundas (DQN) (ver algoritmo de aprendizaje Q)

deepcopy(), parada anticipada

DeepMind, aprendizaje por refuerzo

grados de libertad, sobreajuste de los datos de entrenamiento, curvas de aprendizaje

DeiT (transformadores de imágenes con eficiencia de datos), Vision Transformers

eliminación de ruido de codificadores automáticos, eliminación de ruido de codificadores automáticos-eliminación de ruido codificadores automáticos

modelo probabilístico de difusión de eliminación de ruido (DDPM), codificadores automáticos, GAN y modelos de difusión, modelos de difusión -modelos de difusión

Capa densa, El perceptrón, Creando el modelo usando la API secuencial, Creando el modelo usando la API secuencial, Creación de modelos complejos utilizando la API funcional

matriz densa, Escalado y transformación de características, Transformación Tuberías

estimación de densidad, Técnicas de aprendizaje no supervisado, DBSCAN-DBSCAN, mezclas gaussianas

umbral de densidad, uso de mezclas gaussianas para la detección de anomalías

capa de concatenación de profundidad, GoogLeNet

capa de convolución separable en profundidad, Xception

deque, Implementación de Q-Learning profundo

describe(), echa un vistazo rápido a la estructura de datos

conjunto de desarrollo (dev set), modelo y ajuste de hiperparámetros

Selección

diferenciación, pronóstico de series de tiempo, pronóstico de una serie de tiempo,

Predicción de una serie temporal, la familia de modelos ARMA

modelos de difusión, codificadores automáticos, GAN y modelos de difusión,

Modelos de difusión -Modelos de difusión

filtro dilatado, Segmentación Semántica

tasa de dilatación, segmentación semántica

reducción de dimensionalidad, Aprendizaje no supervisado, Dimensionalidad

Reducción-Otras técnicas de reducción de dimensionalidad

- enfoques para, principales enfoques para la reducción de dimensionalidad-Aprendizaje múltiple
- codificadores automáticos, codificadores automáticos, GAN y modelos de difusión, Representaciones de datos eficientes: PCA de rendimiento con un Codificador automático lineal subcompleto
- elegir el número correcto de dimensiones, Elegir el número correcto Número de dimensiones
- agrupamiento, Algoritmos de agrupamiento: k-means y DBSCAN
- maldición de la dimensionalidad, La maldición de la dimensionalidad-La maldición de la dimensionalidad

- para visualización de datos, **Reducción de dimensionalidad**
- pérdida de información, **reducción de dimensionalidad**
- Isomap, **otras técnicas de reducción de dimensionalidad**
- análisis discriminante lineal, **reducción de otras dimensionalidades**  
**Técnicas**
- LLE, LLE-LLE
- escalamiento multidimensional, **reducción de otras dimensionalidades**  
**Técnicas**
- PCA (ver análisis de componentes principales)
- algoritmo de proyección aleatoria, **Proyección aleatoria-Aleatoria**  
**Proyección**
- Incrustación de vecinos estocásticos distribuidos en t, **Otros**  
**Técnicas de reducción de dimensionalidad, visualizando la moda.**  
Conjunto de datos MNIST

factor de descuento y en el sistema de recompensa, **Evaluación de acciones: el crédito**  
**Problema de asignación**

recompensas con descuento, **Evaluación de acciones: la asignación de crédito**  
**Problema, gradientes de políticas**

Capa de discretización, **La capa de discretización**

discriminador, GAN, **codificadores automáticos, GAN y modelos de difusión,**  
**Redes generativas adversarias: las dificultades de entrenar GAN**

Modelo DistilBERT, **una avalancha de modelos de transformadores, abrazos**  
**Biblioteca de Transformers de Face - Abrazando la biblioteca de Transformers de Face**  
destilación, **una avalancha de modelos de transformadores**

estrategias de distribución API, [Capacitación a Escala Usando la Distribución API de estrategias](#)

Contenedor Docker, [instalación e inicio de TensorFlow Serving](#)

producto escalar, [Mecanismos de atención](#)

Dota 2, [descripción general de algunos algoritmos RL populares](#)

Doble DQN, [Doble DQN](#)

descargar datos, [guardar los cambios de su código y sus datos](#)

[Guardar los cambios de su código y sus datos](#)

DQN (redes Q profundas) (ver algoritmo de Q-learning)

`drop()`, [preparar los datos para algoritmos de aprendizaje automático](#)

`dropna()`, [Limpiar los datos](#)

Abandono y eliminación [de ruido de codificadores automáticos](#)

tasa de deserción, [abandono](#)

regularización de deserción, [Abandono-Abandono](#)

números duales, [Autodiff en modo directo](#)

problema dual, [Las SVM kernelizadas de problema dual](#)

duelo DQN (DDQN), [duelo DQN](#)

atributos ficticios, [manejo de texto y atributos categóricos](#)

Problema moribundo de ReLU, [mejores funciones de activación](#)

Modelos dinámicos con API de subclasiación. [Uso de la API de subclasiación para Cree](#) modelos dinámicos: utilice [la API de subclases para crear modelos dinámicos Modelos](#)

programación dinámica, [Procesos de Decisión de Markov](#)

mi

ejecución ansiosa (modo ansioso), **AutoGraph y Tracing**

regularización de parada anticipada, **Parada Temprana- Parada Temprana,**

**Aumento de gradiente, pronóstico utilizando un modelo lineal**

Computación de vanguardia, **implementación de un modelo en un dispositivo móvil o integrado Dispositivo**

representaciones de datos eficientes, codificadores automáticos, **datos eficientes**

**Representaciones: representaciones de datos eficientes**

red elástica, **regresión neta elástica**

**EllipticEnvelope, otros algoritmos para anomalías y novedades**

**Detección**

ELMo (incrustaciones de modelos de lenguaje), reutilización de material previamente entrenado

**Incorporaciones y modelos de lenguaje**

ELU (unidad lineal exponencial), **ELU y SELU-GELU, Swish y**

**Mismo**

EM (maximización de expectativas), **mezclas gaussianas**

dispositivo integrado, implementar modelo en, **implementar un modelo en un**

**Dispositivo móvil o integrado : implementación de un modelo en un dispositivo móvil o**

**Dispositivo integrado**

Gramáticas Reber integradas, **Ejercicios**

matriz de incrustación, **codificación de características categóricas mediante**

**Incorporaciones, una red de codificador-decodificador para máquinas neuronales**

**Traducción**

tamaño de incrustación, **una red de codificador-decodificador para máquinas neuronales**

**Traducción, Codificaciones posicionales**

incrustaciones, manejo de texto y atributos categóricos

- codificar características categóricas usando, Codificación Categórica  
Funciones que utilizan funciones categóricas de codificación de incrustaciones  
Usando incrustaciones
- reutilización de incrustaciones previamente entrenadas, reutilización de incrustaciones previamente entrenadas y  
Modelos de lenguaje : reutilización de incrustaciones previamente entrenadas y  
Modelos de lenguaje
- análisis de sentimiento, Análisis de sentimiento

Incorporaciones de modelos de lenguaje (ELMo), reutilización de materiales previamente entrenados

Incorporaciones y modelos de lenguaje

codificador, Representaciones de datos eficientes

- (ver también codificadores automáticos)

Modelos codificador-decodificador, Secuencias de entrada y salida, Natural

Procesamiento del lenguaje con RNN y atención, un codificador-

Red decodificadora para búsqueda por haz de traducción automática neuronal

- (ver también mecanismos de atención)

Ejercicio de proyecto de aprendizaje automático de un extremo a otro, Proyecto de aprendizaje automático

de un extremo a otro: ¡pruébelo !

- construyendo el modelo, mire el panorama general: verifique el  
Suposiciones
- descubrir y visualizar datos, explorar y visualizar los datos para obtener información:  
experimentar con combinaciones de atributos
- afinar su modelo, Afinar su modelo-Evaluar su  
Sistema en el equipo de prueba
- obtener los datos, Obtener los datos: crear un conjunto de prueba

- preparar datos para algoritmos de ML, preparar los datos para la máquina  
Algoritmos de aprendizaje -canalizaciones de transformación
- datos reales, ventajas de trabajar con, Trabajar con Datos Reales-  
Trabajar con datos reales
- seleccionar y entrenar un modelo, entrenar y evaluar en el  
Conjunto de entrenamiento : entrene y evalúe en el conjunto de entrenamiento

punto final, implementación del modelo en GCP, creación de un servicio de predicción en IA de vértice

aprendizaje en conjunto, aprendizaje en conjunto y bosques aleatorios.

### Apilado

- embolsar y pegar, Embolsar y pegar: parches aleatorios y subespacios aleatorios
- aumento, aumento de gradiente basado en histograma
- validación cruzada, mejor evaluación mediante validación cruzada
- afinando el sistema, Métodos de conjunto
- bosques aleatorios (ver bosques aleatorios)
- apilamiento, apilamiento-apilamiento
- clasificadores de votación, Clasificadores de votación- Clasificadores de votación

vinculación, una avalancha de modelos de transformadores

medida de impureza de entropía, ¿ impureza de Gini o entropía?

entornos, aprendizaje por refuerzo, Aprender a optimizar

Recompensas, Introducción a OpenAI Gym: Introducción a OpenAI Gym

épocas, descenso de gradiente por lotes

tasa de aprendizaje igualada, GAN, crecimiento progresivo de GAN

equivarianza, [capas de agrupación](#)

análisis de errores, clasificación, [Análisis de errores- Análisis de errores](#)

probabilidades estimadas versus reales, [la curva ROC](#)

estimadores, [Limpiar los datos](#), Canales de transformación, Votación  
[Clasificadores](#)

Norma euclidiana, seleccione una medida de desempeño

archivos de eventos, TensorBoard, [Uso de TensorBoard para visualización](#)

Ejemplo de protobuf, [TensorFlow Protobufs](#)

problema exclusivo o (XOR), [El Perceptrón](#)

ejemplares, propagación por afinidad, [otros algoritmos de agrupación](#)

maximización de expectativas (EM), [mezclas gaussianas](#)

repetición de experiencias, [Las dificultades de entrenar GAN](#)

explicabilidad, mecanismos de atención, [transformadores de visión.](#)

relación de varianza explicada, [relación de varianza explicada](#)

varianza explicada, trazado, [elección del número correcto de](#)

[Dimensiones: elegir el número correcto de dimensiones](#)

gradientes explosivos, [Los problemas de los gradientes que desaparecen/explotan](#)

- (ver también gradientes que desaparecen y explotan)

políticas de exploración, [Aprendizaje en Diferencia Temporal](#), Exploración  
[Políticas](#)

dilema exploración/explotación, aprendizaje por refuerzo, [Neural](#)  
[Políticas de red](#)

unidad lineal exponencial (ELU), [ELU y SELU-GELU](#), Swish y  
[Mismo](#)

programación exponencial, programación de tasa de aprendizaje, tasa de aprendizaje

Planificación

`export_graphviz()`, Entrenamiento y visualización de un árbol de decisiones

árboles adicionales, bosque aleatorio, árboles adicionales

conjunto de árboles extremadamente aleatorio (extra-trees), Extra-Trees

F

clasificador de reconocimiento facial, clasificación multietiqueta

falsos negativos, matriz de confusión, Matrices de confusión

tasa de falsos positivos (FPR) o caída, la curva ROC

falsos positivos, matriz de confusión, Matrices de confusión

fan-in/fan-out, inicialización de Glorot y He

fast-MCD, otros algoritmos para la detección de anomalías y novedades

FCN (redes totalmente convolucionales), redes totalmente convolucionales-

Redes Totalmente Convolucionales, Segmentación Semántica

ingeniería de características, características irrelevantes, algoritmos de agrupamiento: k-means y DBSCAN

extracción de características, aprendizaje no supervisado, características irrelevantes

mapas de características, Apilamiento de múltiples mapas de características-Apilamiento de múltiples

Mapas de características, implementación de capas convolucionales con Keras,

ResNet, SENet

escalado de características, escalado y transformación de características: escalado y transformación de características, descenso de gradiente, clasificación SVM lineal

selección de funciones, funciones irrelevantes, análisis de los mejores modelos y sus errores, regresión de lazo, importancia de las funciones

vectores de características, selección de una medida de rendimiento, regresión lineal,  
**Bajo el capó de los clasificadores SVM lineales**

características, **aprendizaje supervisado**

aprendizaje federado, **Ejecución de un modelo en una página web**

red neuronal feedforward (FNN), **el perceptrón multicapa y**  
**Retropropagación, neuronas recurrentes y capas**

`fetch_openml()`, **MNIST**

`fillna()`, **Limpiar los datos**

filtros, capas convolucionales, **Filtros, Implementación de Convolutional Capas con Keras, Arquitecturas CNN, Xception**

primer momento (media del gradiente), **Adam**

derivadas parciales de primer orden (jacobianos), **AdamW**

`adaptar()`

- y transformadores personalizados, **Transformadores personalizados, Transformación Tuberías**
- limpieza de datos, **Limpiar los datos**
- versus `part_fit()`, **descenso de gradiente estocástico**
- usar solo con conjunto de entrenamiento, **escalado de funciones y transformación**

función de fitness, aprendizaje basado en modelos y un flujo de trabajo típico de  
aprendizaje automático

`fit_transform()`, **limpieza de datos, escalado y transformación de funciones,**  
**Transformadores personalizados, tuberías de transformación**

Objetivos de valor Q fijo, **Objetivos de valor Q fijo**

conjunto de datos plano, **preparación de los datos para modelos de aprendizaje automático**

Conjunto de datos de flores, [modelos preentrenados para aprendizaje por transferencia: preentrenados](#)  
[Modelos para el aprendizaje por transferencia](#)

FNN (red neuronal de retroalimentación), [el perceptrón multicapa y](#)  
[Retropropagación, neuronas recurrentes y capas](#)

pliegues, [mejor evaluación mediante validación cruzada, MNIST, medición](#)  
[Precisión mediante validación cruzada, medición de la precisión mediante validación cruzada](#)  
[Validación](#)

pronóstico de series de tiempo (ver [datos de series de tiempo](#))

olvidar puerta, LSTM, [celdas LSTM](#)

pase hacia adelante, en propagación hacia atrás, [El Perceptrón Multicapa y](#)  
[Propagación hacia atrás](#)

proceso directo, modelo de difusión, [Modelos de difusión -Modelos de difusión](#)

FPR (tasa de falsos positivos) o caída, [la curva ROC](#)

`from_predictions()`, [Análisis de errores](#)

descenso de gradiente completo, [descenso de gradiente por lotes](#)

capa totalmente conectada, [El Perceptrón, La Arquitectura de lo Visual](#)  
[Corteza, requisitos de memoria](#)

redes totalmente convolucionales (FCN), [redes totalmente convolucionales-](#)  
[Redes Totalmente Convolucionales, Segmentación Semántica](#)

definición de función (`FuncDef`), [funciones TF y funciones concretas](#)

gráfico de funciones (`FuncGraph`), [funciones TF y funciones concretas](#)

API funcional, modelos complejos con, [construcción de modelos complejos](#)  
[Uso de API funcionales : creación de modelos complejos utilizando](#)  
[API funcional](#)

`FunciónTransformador`, [Transformadores personalizados](#)

## Puntuación F , precisión y recuperación

GRAMO

juego (ver aprendizaje por refuerzo)

valor gamma ( $\gamma$ ), núcleo RBF gaussiano

GAN (ver redes generativas adversarias)

controladores de puerta, LSTM, celdas LSTM

unidades de activación cerradas, WaveNet

celda de unidad recurrente cerrada (GRU), celdas GRU- celdas GRU , enmascaramiento

Distribución gaussiana, escalamiento y transformación de características, función de entrenamiento y costo, codificadores automáticos variacionales -variacionales codificadores automáticos

Modelo de mezcla gaussiana (GMM), Mezclas gaussianas-Otras

Algoritmos para la detección de anomalías y novedades

- detección de anomalías, uso de mezclas gaussianas para anomalías  
Detección mediante mezclas gaussianas para la detección de anomalías
- Mezclas bayesianas gaussianas, Mezcla bayesiana gaussiana  
Modelos
- fast-MCD, otros algoritmos para la detección de anomalías y novedades
- inverse\_transform() con PCA, otros algoritmos para detección de anomalías y novedades
- bosque de aislamiento, Otros algoritmos para anomalías y novedades  
Detección
- y limitaciones de k-medias, Límites de k-medias

- factor de valores atípicos locales, otros algoritmos para anomalías y novedades  
Detección
- SVM de una clase, otros algoritmos para anomalías y novedades  
Detección
- seleccionar el número de conglomerados, Seleccionar el número de conglomerados-  
Seleccionar el número de grupos

Proceso gaussiano, ajuste fino de hiperparámetros de redes neuronales

Núcleo RBF gaussiano, Transformadores personalizados, Núcleo RBF gaussiano

Clases de SVM y complejidad computacional, SVM kernelizadas

GBRT (árboles de regresión potenciados por gradiente), aumento de gradiente,  
Aumento de gradiente : aumento de gradiente basado en histograma

GCP (Google Cloud Platform), creación de un servicio de predicción en  
Vertex AI: creación de un servicio de predicción en Vertex AI

GCS (Google Cloud Storage), creación de un servicio de predicción en  
IA de vértice

GD (ver descenso de gradiente)

GELU, GELU, Swish y Mish-GELU, Swish y Mish

error de generalización, pruebas y validación: ajuste de hiperparámetros y selección de  
modelo

redes generativas adversarias (GAN), Generative Adversarial

Redes-EstiloGAN

- convolucional profundo, GAN convolucionales profundos -profundo  
GAN convolucionales
- crecimiento progresivo de, Crecimiento progresivo de GAN-  
Crecimiento progresivo de las GAN
- StyleGAN, StyleGAN-StyleGAN

- dificultades de entrenamiento, [Las dificultades de entrenar GAN: el Dificultades de entrenar GAN](#)

codificadores automáticos generativos, [codificadores automáticos variacionales](#)

modelos generativos, [codificadores automáticos](#), [GAN](#) y modelos de difusión

- (ver también modelo de mezcla gaussiana)

generador, [GAN](#), [codificadores automáticos](#), [GAN](#) y modelos de difusión,

[Redes generativas adversarias: las dificultades de entrenar GAN](#)

algoritmo genético, [búsqueda de políticas](#)

distancia geodésica, [Otras técnicas de reducción de dimensionalidad](#)

datos geográficos, visualización, [Visualización de datos geográficos-](#)

[Visualización de datos geográficos](#)

`get_dummies()`, [Manejo de texto y atributos categóricos](#)

`get_feature_names_out()`, [Transformadores personalizados](#)

`get_params()`, [Transformadores personalizados](#)

[Impureza GINI](#), [Hacer predicciones, ¿ Impureza Gini o entropía?](#)

capa de agrupación promedio global, [Implementación de capas de agrupación con Keras](#)

mínimo global versus local, [descenso de gradiente](#), [descenso de gradiente](#)

[Inicialización de Glorot](#), [Glorot y He](#) [Inicialización-Glorot y He](#)  
[Inicialización](#)

GMM (ver modelo de mezcla gaussiana)

Google Cloud Platform (GCP), [creación de un servicio de predicción en Vertex AI: creación de un servicio de predicción en Vertex AI](#)

Google Cloud Storage (GCS), [creación de un servicio de predicción en IA de vértice](#)

Google Colab, [ejecución de ejemplos de código con Google Colab](#)

[Ejecutar los ejemplos de código usando Google Colab](#)

Google Vertex AI (ver Vertex AI)

GoogLeNet, [GoogLeNet-GoogLeNet](#), Xception

Implementación de GPU, [descenso de gradiente de mini lotes](#), uso de GPU para

Acelere los cálculos: ejecución paralela en varios dispositivos,

Capacitación a escala utilizando la API de estrategias de distribución

- obtener su propia GPU, [obtener su propia GPU: obtener su Propia GPU](#)
- administrar RAM, [Administrar la GPU RAM-Administrar la GPU RAM](#)
- manejo de operaciones, [ejecución paralela en múltiples dispositivos](#)
- ejecución paralela en múltiples dispositivos, [ejecución paralela En varios dispositivos: ejecución paralela en varios Dispositivos](#)
- colocar operaciones y variables en dispositivos, [colocar operaciones y variables en dispositivos-colocar operaciones y variables en Dispositivos](#)

ascenso de gradiente, [búsqueda de políticas](#)

árboles de regresión potenciados por gradiente (GBRT), [aumento de gradiente](#),

Aumento de gradiente : aumento de gradiente basado en histograma

aumento de gradiente, aumento [de gradiente - aumento de gradiente](#)

recorte de degradado, [recorte de degradado](#)

descenso de gradiente (GD), modelos de entrenamiento, descenso de gradiente-mini-Descenso de gradiente por lotes

- comparaciones de algoritmos, [descenso de gradiente de mini lotes](#)
- descenso de gradiente por lotes, [descenso de gradiente por lotes- gradiente por lotes](#)  
[Descenso, regresión de crestas](#)
- mínimo local versus global, [descenso de gradiente](#)
- descenso de gradiente de mini lotes, [Descenso de gradiente de mini lotes-Mini-Descenso de gradiente por lotes](#)
- minimizando la pérdida de bisagra, [bajo el capó de Linear SVM Clasificadores](#)
- versus optimización del impulso, [Momentum](#)
- con optimizadores, [Optimizadores más rápidos-AdamW](#)
- barajar datos, [barajar los datos](#)
- descenso de gradiente estocástico, [descenso de gradiente estocástico-Descenso del gradiente estocástico](#)

aumento del árbol de gradiente, [aumento de gradiente](#)

gradientes

- autodiff para computación, [Computación de gradientes usando Autodiff-Calcular gradientes usando Autodiff](#)
- problema de saturación de ancho de banda, [Saturación de ancho de banda](#)
- Algoritmo PG, [búsqueda de políticas, gradientes de políticas- gradientes de políticas](#)
- [Actualizaciones](#) obsoletas y asincrónicas
- inestable (ver gradientes que desaparecen y explotan)

modo gráfico, [AutoGraph y Tracing](#)

unidades de procesamiento gráfico (ver implementación de GPU)

gráficos y funciones, TensorFlow, [Un recorrido rápido por TensorFlow](#),  
[Funciones y gráficos de TensorFlow: reglas de funciones TF](#), [TensorFlow Gráficos: uso de funciones TF con Keras \(o no\)](#)

[Graphviz, entrenamiento y visualización de un árbol de decisiones](#)

algoritmo codicioso, CART como, [El algoritmo de entrenamiento CART](#)

decodificación codiciosa, [generando texto shakesperiano falso](#)

preentrenamiento codicioso en capas, [preentrenamiento no supervisado](#), [entrenamiento Un codificador automático a la vez, crecimiento progresivo de las GAN](#)

búsqueda de cuadrícula, [Búsqueda de cuadrícula- Búsqueda de cuadrícula](#)

[GridSearchCV](#), [Búsqueda de cuadrícula- Búsqueda de cuadrícula](#)

API de gRPC, consulta a través de, [Consulta de TF Sirviendo a través de la API de gRPC](#)

Celda GRU (unidad recurrente cerrada), [celdas GRU- celdas GRU, enmascaramiento](#)

h

agrupamiento duro, [k-medias](#)

clasificación de margen duro, [clasificación de margen suave, bajo el Campana de clasificadores SVM lineales](#)

Clasificadores de votación dura, [Clasificadores de votación](#)

media armónica, [precisión y recuperación](#)

colisión de hash, [la capa StringLookup](#)

Capa hash, [La capa hash](#)

truco de hash, [la capa StringLookup](#)

HDBSCAN (DBSCAN jerárquico), DBSCAN

He inicialización, Glorot y He Inicialización-Glorot y He Inicialización

Función de paso Heaviside, El Perceptrón

cola pesada, distribución de funciones, escalado y transformación de funciones

La regla de Hebb, el perceptrón.

Aprendizaje hebbiano, el perceptrón

Arpilleras, AdamW

capas ocultas

- neuronas por capa, Número de neuronas por capa oculta
- número de, Número de capas ocultas
- codificadores automáticos apilados, codificadores automáticos apilados-Training One Codificador automático a la vez

agrupamiento jerárquico, aprendizaje no supervisado

DBSCAN jerárquico (HDBSCAN), DBSCAN

alta varianza, con árboles de decisión, los árboles de decisión tienen una alta Diferencia

función de pérdida de bisagra, bajo el capó de los clasificadores lineales SVM

aumento de gradiente basado en histograma (HGB), basado en histograma

Aumento de gradiente : aumento de gradiente basado en histograma

histogramas, eche un vistazo rápido a la estructura de datos

conjuntos de retención, pruebas y validación

validación de exclusión, ajuste de hiperparámetros y selección de modelo

conjunto de datos sobre vivienda, **Trabajar con datos reales: comprobar los supuestos**

Pérdida de Huber, **MLP de regresión, funciones de pérdida personalizadas, personalizadas Métricas y pronósticos mediante un modelo lineal**

Abrazando la cara, **abrazando la cara Biblioteca de Transformers-Abrazando la cara Biblioteca de transformadores**

Algoritmo húngaro, **seguimiento de objetos**

Sintonizador de hiperbanda, **ajuste fino de hiperparámetros de redes neuronales**

**tangente hiperbólica (htan), el perceptrón multicapa y**

**Retropropagación, luchando contra el problema de los gradientes inestables**

**hiperparámetros, sobreajuste de los datos de entrenamiento, ajuste neuronal**

**Hiperparámetros de red: tasa de aprendizaje, tamaño de lote y otros**

**Hiperparámetros**

- función de activación, **tasa de aprendizaje, tamaño de lote y otros Hiperparámetros**
- tamaño de lote, **tasa de aprendizaje, tamaño de lote y otros Hiperparámetros**
- Algoritmo CART, **El algoritmo de entrenamiento CART**
- capas convolucionales, **Implementación de capas convolucionales con Keras**
- en transformaciones personalizadas, **Transformadores personalizados**
- árbol de decisión, **aumento de gradiente**
- reducción de dimensionalidad, **elección del número correcto de Dimensiones**
- valor gamma ( $\gamma$ ), **núcleo RBF gaussiano**
- Desafíos de GAN, **las dificultades de entrenar GAN**

- Keras Tuner, ajuste de hiperparámetros en Vertex AI
- tasa de aprendizaje, descenso de gradiente, tasa de aprendizaje, tamaño de lote y Otros hiperparámetros
- impulso  $\beta$ , impulso
- Muestras de Monte Carlo, abandono de Monte Carlo (MC)
- neuronas por capa oculta, Número de neuronas por capa oculta
- y normalización, escalamiento y transformación de funciones
- número de capas ocultas, Número de capas ocultas
- número de iteraciones, tasa de aprendizaje, tamaño del lote y otros Hiperparámetros
- optimizador, tasa de aprendizaje, tamaño de lote y otros Hiperparámetros
- Algoritmos PG, gradientes de políticas
- interacción entre preprocesador y modelo, búsqueda de cuadrícula
- búsqueda aleatoria, ajuste de la red neuronal  
Hiperparámetros: hiperparámetros de red neuronal de ajuste fino
- guardar junto con el modelo, funciones de activación personalizadas, Inicializadores, regularizadores y restricciones
- SGDClassifier, clases SVM y complejidad computacional
- submuestra, aumento de gradiente
- Clasificadores SVM con kernel polinomial, Polynomial Kernel
- tolerancia ( $\epsilon$ ), clases SVM y complejidad computacional
- ajuste de hiperparámetros y selección de modelo  
Ajuste de hiperparámetros y selección de modelo, cuadrícula de búsqueda

Busque, evalúe su sistema en el conjunto de pruebas, entrene y evalúe el modelo, ajuste de hiperparámetros en Vertex AI-  
Ajuste de hiperparámetros en Vertex AI

hipótesis, seleccione una medida de desempeño

refuerzo de hipótesis (ver refuerzo)

función de hipótesis, [regresión lineal](#)

## I

matriz de identidad, [regresión de crestas](#)

IID (ver independiente y distribuido idénticamente)

generación de imágenes, [Segmentación Semántica](#), [StyleGANs](#), [Difusión Modelos](#)

segmentación de imágenes, [algoritmos de agrupamiento: k-means y DBSCAN](#),

Uso de agrupación en clústeres para la segmentación de imágenes: uso de agrupación en clústeres para imágenes

[Segmentación](#)

imágenes, clasificar y generar, [Ejemplos de Aplicaciones](#)

- codificadores automáticos (ver codificadores automáticos)
- CNN (ver redes neuronales convolucionales)
- modelos de difusión, Modelos [de Difusión -Modelos de Difusión](#)
- generando con GANs, [Generative Adversarial Networks-EstiloGAN](#)
- Implementación de MLP, [creación de un clasificador de imágenes utilizando el API secuencial : uso del modelo para hacer predicciones](#)
- etiquetas, [Clasificación y Localización](#)
- carga y preprocesamiento de datos, [Capas de preprocesamiento de imágenes](#)

- imágenes representativas, uso de agrupación en clústeres para semisupervisado  
Aprendiendo

- segmentación semántica, uso de agrupación en clústeres para imágenes  
Segmentación

- ajuste de hiperparámetros, ajuste fino de la red neuronal  
Hiperparámetros

muestreo de importancia (IS), repetición de experiencia priorizada

medidas de impureza, hacer predicciones, Gini ¿Impureza o entropía?

imputación, limpiar los datos

aprendizaje incremental, aprendizaje en línea

PCA incremental (IPCA), PCA incremental -PCA incremental

instancias de entrenamiento independientes e idénticamente distribuidas (IID) como,  
Descenso del gradiente estocástico

polarización inductiva, transformadores de visión

inerzia, modelo, métodos de inicialización de centroide, k-medias aceleradas y k-medias de mini lotes

inferencia, aprendizaje basado en modelos y un flujo de trabajo típico de aprendizaje automático

info(), echa un vistazo rápido a la estructura de datos

teoría de la información, regresión Softmax, ¿ impureza de Gini o entropía?

inliers, Técnicas de aprendizaje no supervisado

secuencias de entrada y salida, RNN, secuencias de entrada y salida-  
Secuencias de entrada y salida

puerta de entrada, LSTM, celdas LSTM

capa de entrada, red neuronal, **el perceptrón**, creación del modelo utilizando la API secuencial, construcción de modelos complejos utilizando el API funcional

- (ver también capas ocultas)

firma de entrada, **funciones TF** y funciones concretas

segmentación de instancias, uso de agrupaciones en clústeres para la segmentación de imágenes, Segmentación semántica

aprendizaje basado en instancias, aprendizaje basado en instancias, aprendizaje basado en modelos y un flujo de trabajo típico de aprendizaje automático

grupo de subprocessos interoperativos, ejecución paralela en varios dispositivos

término de intersección constante, **regresión lineal**

entrelazar líneas de múltiples archivos, Intercalar líneas de múltiples Archivos: líneas entrelazadas de varios archivos

ML interpretable, **hacer predicciones**

invariancia, capa de agrupación máxima, **capas de agrupación**

inverse\_transform(), Escalado y transformación de funciones, personalizado Transformadores, PCA para compresión, otros algoritmos para detección de anomalías y novedades

IPCA (PCA incremental), PCA incremental -PCA incremental

conjunto de datos de iris, **límites de decisión**

error irreducible, **Curvas de aprendizaje**

IS (muestreo de importancia), repetición de experiencia priorizada

bosque de aislamiento, Otros algoritmos para la detección de anomalías y novedades

Isomap, otras técnicas de reducción de dimensionalidad

ruido isotrópico, [Modelos de Difusión](#)

[IterativeImputer](#), [limpia los datos](#)

j

[Jacobianos](#), [AdamW](#)

biblioteca joblib, [iniciar, monitorear y mantener su sistema](#): iniciar,  
[Supervise y mantenga su sistema](#)

Líneas JSON, [ejecución de trabajos de predicción por lotes en Vertex AI](#), ejecución  
[Trabajos de predicción por lotes en Vertex AI](#)

Jupyter, [ejecución de ejemplos de código con Google Colab](#)

k

validación cruzada k veces mayor, [mejor evaluación mediante validación cruzada](#),  
[Medición de la precisión mediante validación cruzada](#), medición de la precisión  
[Usando validación cruzada](#)

Algoritmo k-means, [Transformadores personalizados](#), [k-means-Límites de k-medio](#)

- k-medias aceleradas, [k-medias aceleradas y k-mini-lotes medio](#)
- Métodos de inicialización de centroide, [Métodos de inicialización de centroide](#).  
[Métodos de inicialización de centroide](#)
- encontrar el número óptimo de conglomerados, [Encontrar el número óptimo de conglomerados](#)-Encontrar [el número óptimo de conglomerados](#)
- limitaciones de, [límites de k-medias](#)
- k-medias mini-lote, [k-medias aceleradas y mini-lote k-medio](#)

- funcionamiento de, [El algoritmo k-means](#)-[El algoritmo k-means](#)

[k-means++](#), [métodos de inicialización de centroide](#)

Regresión de k vecinos más cercanos, [aprendizaje basado en modelos](#) y un flujo de trabajo típico de aprendizaje automático

[Inicialización de Kaiming](#), [Inicialización de Glorot y He](#)

Filtros Kalman, [seguimiento de objetos](#)

API de Keras, [objetivo](#) y [enfoque](#), un recorrido rápido por TensorFlow

- (ver también redes neuronales artificiales)
- y acceder a la API de TensorFlow directamente, [preprocesamiento de imágenes](#)  
[Capas](#)
- soporte de función de activación, [Leaky ReLU](#), [ELU](#) y [SELU](#),  
[GELU](#), [Swish](#) y [Mish](#)
- implementación de capa convolucional, [Implementación de convolucional](#)  
[Capas con capas convolucionales que implementan Keras con](#)  
[Keras](#)
- funciones personalizadas en, [Funciones y gráficos de TensorFlow](#)
- recorte de degradado, [recorte de degradado](#)
- capas de preprocesamiento de imágenes, [Capas de preprocesamiento de imágenes](#)
- implementar MLP con, [implementar MLP con Keras](#)-Using  
[TensorBoard para visualización](#)
- manejo de inicialización, [Glorot y He Inicialización](#)
- inicializadores, [Creando el modelo usando la API secuencial](#)
- preprocesamiento de capas, [Keras Preprocesamiento de capas](#)-Uso  
[Componentes del modelo de lenguaje previamente entrenado](#)

- programación de tasa de aprendizaje, [programación de tasa de aprendizaje-aprendizaje](#)  
[Programación de tarifas](#)
- cargar un conjunto de datos, [construir un clasificador de imágenes usando el](#)  
[API secuencial](#)
- Algoritmo PG, [gradientes de políticas- gradientes de políticas](#)
- implementación de capa de pool, [Implementación de capas de pool con](#)  
[Implementación de capas de agrupación de Keras con Keras](#)
- modelos CNN previamente entrenados, [utilizando modelos previamente entrenados de Keras-](#)  
[Uso de modelos previamente entrenados de Keras](#)
- ResNet-34 CNN con, [implementación de ResNet-34 CNN usando](#)  
[Keras](#)
- guardar modelos, [exportar modelos guardados, implementar un modelo en un](#)  
[Dispositivo móvil o integrado](#)
- implementación de codificador apilado, [Implementación de un codificador apilado](#)  
[Codificador automático usando Keras](#)
- Conjunto de datos API tf.data, [Uso del conjunto de datos con Keras-Uso del conjunto](#)  
[de datos con Keras](#)
- Biblioteca tf.keras, [uso de Keras para cargar el conjunto de datos](#)
- tf.keras.activations.get(), [Capas personalizadas](#)
- tf.keras.activations.relu(), [Creando el modelo usando la API secuencial](#)
- Módulo tf.keras.applications, [uso de modelos previamente entrenados de Keras](#)
- tf.keras.applications.xception.preprocess\_input(), [Modelos previamente entrenados](#)  
[para el aprendizaje por transferencia](#)
- Módulo tf.keras.backend, [tensores y operaciones](#)

- `tf.keras.callbacks.EarlyStopping`, uso de devoluciones de llamada
- `tf.keras.callbacks.LearningRateScheduler`, programación de la tasa de aprendizaje
- `tf.keras.callbacks.ModelCheckpoint`, uso de devoluciones de llamada
- `tf.keras.callbacks.TensorBoard`, uso de TensorBoard para visualización y enmascaramiento
- `tf.keras.datasets.imdb.load_data()`, Análisis de sentimiento
- `tf.keras.initializers.VarianceScaling`, Glorot y He Inicialización
- `tf.keras.layers.ActivityRegularization`, codificadores automáticos dispersos
- `tf.keras.layers.AdditiveAttention`, Mecanismos de atención
- `tf.keras.layers.Attention`, Mecanismos de atención
- `tf.keras.layers.AvgPool2D`, Implementación de capas de agrupación con Keras
  
- `tf.keras.layers.BatchNormalization`, Normalización por lotes, Implementación de la normalización por lotes con Keras-Implementación de la normalización por lotes con Keras, Lucha contra el problema de los gradientes inestables
  
- `tf.keras.layers.Bidirectional`, RNN bidireccionales
- `tf.keras.layers.CategoryEncoding`, la capa CategoryEncoding
- `tf.keras.layers.CenterCrop`, Capas de preprocessamiento de imágenes
- `tf.keras.layers.Concatenate`, Creación de modelos complejos mediante la API funcional, Creación de modelos complejos mediante la API funcional, GoogLeNet
  
- `tf.keras.layers.Conv1D`, segmentación semántica, enmascaramiento

- `tf.keras.layers.Conv2D`, [Implementación de capas convolucionales con Keras](#)
- `tf.keras.layers.Conv2DTranspose`, [segmentación semántica](#)
- `tf.keras.layers.Conv3D`, [Segmentación Semántica](#)
- `tf.keras.layers.Dense`, [creación de modelos complejos utilizando la API funcional](#), [creación de modelos complejos utilizando la API funcional](#), [capas personalizadas - capas personalizadas, codificación de características categóricas mediante incrustaciones, pesos de vinculación, codificadores automáticos variacionales](#)
- `tf.keras.layers.Discretization`, [la capa de discretización](#)
- `tf.keras.layers.Abandono`, [Abandono](#)
- `tf.keras.layers.Embedding`, [codificación de características categóricas mediante incrustaciones, construcción y entrenamiento del modelo Char-RNN, análisis de sentimiento, codificaciones posicionales](#)
- `tf.keras.layers.GlobalAvgPool2D`, [Implementación de capas de agrupación con Keras](#)
- `tf.keras.layers.GRU`, [celdas GRU](#)
- `tf.keras.layers.GRUCell`, [células GRU](#)
- `tf.keras.layers.Hashing`, [la capa de hash](#)
- `tf.keras.layers.Input`, [creación de modelos complejos utilizando la API funcional](#)
- `tf.keras.layers.Lambda`, [capas personalizadas, construcción y entrenamiento del modelo Char-RNN](#)
- `tf.keras.layers.LayerNormalization`, [luchando contra el problema de los gradientes inestables](#)
- `tf.keras.layers.LeakyReLU`, [Leaky ReLU](#)

- `tf.keras.layers.LSTM`, [celdas LSTM](#)
- `tf.keras.layers.Enmascaramiento`, [Enmascaramiento](#)
- `tf.keras.layers.MaxPool2D`, [Implementación de capas de agrupación con Keras](#)
- `tf.keras.layers.MultiHeadAttention`, [Atención multicabezal](#)
- `tf.keras.layers.Normalization`, [creación de un MLP de regresión utilizando la API secuencial](#), [creación de modelos complejos utilizando la API funcional](#), [la capa de normalización: la capa de normalización](#)
- `tf.keras.layers.PReLU`, [Leaky ReLU](#)
- `tf.keras.layers.Rescaling`, [capas de preprocessamiento de imágenes](#)
- `tf.keras.layers.Resizing`, [capas de preprocessamiento de imágenes](#), [uso de modelos previamente entrenados de Keras](#)
- `tf.keras.layers.RNN`, [células LSTM](#)
- `tf.keras.layers.SeparableConv2D`, [Xception](#)
- `tf.keras.layers.StringLookup`, [la capa StringLookup](#)
- `tf.keras.layers.TextVectorization`, [preprocessamiento de texto](#)-[preprocessamiento de texto](#), [creación del conjunto de datos de entrenamiento](#), [construcción y entrenamiento del modelo Char-RNN](#), [análisis de sentimiento](#), [análisis de sentimiento: una red de codificador-decodificador para traducción automática neuronal](#)
- `tf.keras.layers.TimeDistributed`, [pronóstico utilizando un modelo de secuencia a secuencia](#)
- `tf.keras.losses.Huber`, [funciones de pérdida personalizadas](#)
- `tf.keras.losses.kullback_leibler_divergence()`, [codificadores automáticos dispersos](#)

- `tf.keras.losses.Pérdida`, [guardado y carga de modelos que contienen componentes personalizados](#)
- `tf.keras.losses.sparse_categorical_crossentropy()`, [Compilación del modelo, Arquitecturas CNN, Construcción y entrenamiento del modelo Char-RNN, Una red codificadora-decodificadora para traducción automática neuronal, Biblioteca Transformers de Hugging Face](#)
- `tf.keras.metrics.MeanIoU`, [Clasificación y Localización](#)
- `tf.keras.metrics.Metric`, [Métricas personalizadas](#)
- `tf.keras.metrics.Precisión`, [Métricas personalizadas](#)
- `tf.keras.Model`, [creación de modelos complejos utilizando la API funcional](#)
- `tf.keras.models.clone_model()`, [Uso de la API de subclases para crear modelos dinámicos y transferir aprendizaje con Keras](#)
- `tf.keras.models.load_model()`, [Guardar y restaurar un modelo, Guardar y cargar modelos que contienen componentes personalizados: Guardar y cargar modelos que contienen componentes personalizados, Modelos personalizados, Capacitación a escala utilizando la API de estrategias de distribución](#)
- `tf.keras.optimizers.Adam`, [Creación de un MLP de regresión utilizando la API secuencial, AdamW](#)
- `tf.keras.optimizadores.Adamax`, [AdamW](#)
- `tf.keras.optimizadores.experimental.AdamW`, [AdamW](#)
- `tf.keras.optimizadores.Nadam`, [AdamW](#)
- `tf.keras.optimizers.schedules`, [programación de la tasa de aprendizaje](#)
- `tf.keras.optimizers.SGD`, [impulso](#)

- `tf.keras.preprocessing.image.ImageDataGenerator`, [modelos previamente entrenados para el aprendizaje por transferencia](#)
  - `tf.keras.regularizers.l1_l2()`, [Regularización  \$\ell\_1\$  y  \$\ell\_2\$](#)
  - `tf.keras.Sequential`, [Creación del modelo utilizando la API secuencial](#), [Creación de un MLP de regresión utilizando la API secuencial](#), [Arquitecturas CNN](#)
- 
- `tf.keras.utils.get_file()`, [Creando el conjunto de datos de entrenamiento](#)
  - `tf.keras.utils.set_random_seed()`, [Creando el modelo usando la API secuencial](#)
- 
- `tf.keras.utils.timeseries_dataset_from_array()`, [Preparación de los datos para modelos de aprendizaje automático](#), [Preparación de los datos para modelos de aprendizaje automático](#)
  - `tf.keras.utils.to_categorical()`, [Compilando el modelo](#)
  - [pronóstico de series de tiempo para RNN](#), [pronóstico usando un simple Pronóstico RNN utilizando un RNN profundo](#)
  - [transferir aprendizaje con](#), [Transferir aprendizaje con Keras-Transfer Aprendiendo con Keras](#)
  - [usando funciones TF \(o no\)](#), [usando funciones TF con Keras \(o No\)](#)

Sesión de Keras, [Creando el modelo usando la API secuencial](#).

Keras Tuner, [ajuste de hiperparámetros en Vertex AI](#)

truco del kernel, clases [polinomiales Kernel-SVM y computación Complejidad](#), SVM kernelizadas -SVM kernelizadas

SVM kernelizadas, SVM [kernelizadas -SVM kernelizadas](#)

kernels (kernels de convolución), [filtros, arquitecturas CNN](#)

kernels (tiempos de ejecución), Ejecución de ejemplos de código con Google Colab

Divergencia de KL (Kullback-Leibler), Regresión Softmax, Sparse codificadores automáticos

KLDivergenceRegularizer, codificadores automáticos dispersos

KMeans, Transformadores personalizados

KNeighborsClassifier, Clasificación multietiqueta, Multisalida Clasificación

KNNImputer, limpia los datos

Divergencia de Kullback-Leibler (KL), regresión Softmax, escasa codificadores automáticos

## I

propagación de etiquetas, uso de agrupación en clústeres para el aprendizaje semisupervisado

Uso de agrupación en clústeres para el aprendizaje semisupervisado

etiquetas, encuadre el problema

- en agrupamiento, k-medias
- clasificación de imágenes, Clasificación y Localización
- aprendizaje supervisado, aprendizaje supervisado
- problema de datos sin etiquetar, preentrenamiento no supervisado usando apilados codificadores automáticos

puntos de referencia, características de similitud

modelos de lenguaje, Generando texto de Shakespeare usando un Personaje RNN

- (ver también procesamiento del lenguaje natural)

clasificación de gran margen, clasificación lineal SVM

Lazo, regresión de lazo

regresión de lazo, regresión de lazo- regresión de lazo

modelos de difusión latente, Modelos de Difusión

pérdida latente, codificadores automáticos variacionales

representación latente de entradas, transformadores de visión, codificadores automáticos, GAN y modelos de difusión, representaciones de datos eficientes, profundidad GAN convolucionales

espacio latente, Autocodificadores variacionales

ley de los grandes números, Clasificadores de Votación

normalización de capas, ejercicios, secuencias de procesamiento utilizando RNN y CNN, lucha contra el problema de los gradientes inestables

LDA (análisis discriminante lineal), reducción de otras dimensionalidades

Técnicas

nodo hoja, árbol de decisión, hacer predicciones, estimar clases

Probabilidades, hiperparámetros de regularización

LeakyReLU, Leaky ReLU-Leaky ReLU

curvas de aprendizaje, análisis de sobreajuste o desajuste, Curvas de aprendizaje-Curvas de aprendizaje

tasa de aprendizaje, aprendizaje en línea, descenso de gradiente, gradiente por lotes Descenso, descenso de gradiente estocástico, tasa de aprendizaje, tamaño de lote y otros hiperparámetros, Q-Learning

horarios de tasa de aprendizaje, Tasa de aprendizaje Programación- Tasa de aprendizaje Planificación

horarios de aprendizaje, descenso de gradiente estocástico

Inicialización de LeCun, inicialización de Glorot y He

LeNet-5, **La arquitectura de la corteza visual, LeNet-5**

Distancia de Levenshtein, **núcleo RBF gaussiano**

Biblioteca liblinear, **clases SVM y complejidad computacional**

Biblioteca libsvm, **clases SVM y complejidad computacional**

Conjunto de datos de satisfacción con la vida, **aprendizaje basado en modelos y un flujo de trabajo típico de aprendizaje automático.**

función de probabilidad, **Seleccionar el número de conglomerados-Seleccionar el Número de grupos**

análisis discriminante lineal (LDA), **reducción de otras dimensionalidades**

**Técnicas**

modelos lineales

- pronóstico de series de tiempo, **pronóstico utilizando un modelo lineal**
- regresión lineal (ver regresión lineal)
- Modelos **lineales** regularizados y regularizados : **parada anticipada**
- SVM, **clasificación SVM lineal: clasificación de margen blando**
- ejemplo de entrenamiento y ejecución, **aprendizaje basado en modelos y un flujo de trabajo típico de aprendizaje automático**

Regresión lineal, **aprendizaje basado en modelos y un flujo de trabajo típico de aprendizaje automático . Aprendizaje basado en modelos y un flujo de trabajo típico de aprendizaje automático. Regresión **lineal** , descenso de gradiente en mini lotes.**

- comparación de algoritmos, **descenso de gradiente mini-batch**
- complejidad computacional, **Complejidad Computacional**
- descenso de gradiente en, gradiente de descenso **de gradiente -mini-batch Descendencia**

- curvas de aprendizaje en, Curvas de Aprendizaje -Curvas de Aprendizaje
- Ecuación normal, La ecuación normal-La ecuación normal
- modelos de regularización (ver regularización)
- regresión de crestas, regresión de crestas- regresión de crestas, elástica Regresión neta
- evaluación del conjunto de entrenamiento, entrenar y evaluar en el conjunto de entrenamiento
- usando descenso de gradiente estocástico, Descenso de gradiente estocástico

clasificación SVM lineal, clasificación SVM lineal- margen suave

Clasificación, bajo el capó de clasificadores SVM lineales kernelizados SVM

unidades de umbral lineal (LTU), el perceptrón

linealmente separable, clases SVM, Clasificación SVM lineal-Lineal

Clasificación SVM, características de similitud

Regresión lineal, limpieza de datos, escalado de funciones y  
Transformar, entrenar y evaluar en el conjunto de entrenamiento, lo normal  
Ecuación, complejidad computacional, regresión polinómica

LinearSVC, clasificación de margen suave, clases SVM y  
Complejidad computacional, bajo el capó de SVM lineal  
Clasificadores

LinearSVR, regresión SVM- regresión SVM

Lipschitz continuo, derivado como, descenso de gradiente

LLE (incrustación localmente lineal), LLE-LLE

cargar y preprocesar datos, cargar y preprocesar datos con TensorFlow: el  
 proyecto de conjuntos de datos de TensorFlow

- capas de preprocesamiento de imágenes, Capas de preprocesamiento de imágenes

- preprocesamiento de capas en Keras, carga y preprocesamiento de datos con TensorFlow, preprocesamiento de capas de Keras : uso de preentrenamiento Componentes del modelo de lenguaje
- API tf.data, carga y preprocesamiento de datos con TensorFlow: uso del conjunto de datos con Keras
- Proyecto TFDS, el proyecto de conjuntos de datos TensorFlow: el Proyecto de conjuntos de datos TensorFlow
- Formato TFRecord, carga y preprocesamiento de datos con TensorFlow, el formato TFRecord que maneja listas de listas utilizando el Protobuf SequenceExample

factor de valores atípicos locales (LOF), otros algoritmos para anomalías y novedades Detección

campo receptivo local, La arquitectura de la corteza visual

normalización de respuesta local (LRN), AlexNet

mínimo local versus global, descenso de gradiente, descenso de gradiente

hash sensible a la localidad (LSH), proyección aleatoria

localización, CNN, clasificación y localización- detección de objetos

incrustación localmente lineal (LLE), LLE-LLE

LOF (factor atípico local), otros algoritmos para anomalías y novedades Detección

pérdida de registros, capacitación y función de costos

función logarítmica de probabilidades, estimación de probabilidades

transformador-log, Transformadores personalizados

dispositivo GPU lógico, Gestión de la RAM de la GPU

función logística, estimación de probabilidades

regresión logística, aprendizaje supervisado, escalamiento de funciones y  
Transformación, regresión logística- regresión Softmax

- Ilustración de límites de decisión, Límites de decisión-Decisión Límites
- estimar probabilidades, Estimar probabilidades-Estimar Probabilidades
- modelo de regresión softmax, Regresión Softmax-Softmax Regresión
- función de capacitación y costos, Función de capacitación y costos - Función de capacitación y costos

Regresión logística, límites de decisión, regresión Softmax

función logit, estimación de probabilidades

secuencias largas, entrenamiento de RNN, manejo de secuencias largas-  
OndaNet

- Problema de memoria a corto plazo, Abordar la memoria a corto plazo Problema-WaveNet
- problema de gradientes inestables, Luchando contra los gradientes inestables Cómo combatir el problema de los gradientes inestables

célula de memoria a corto plazo (LSTM), células LSTM - células LSTM,  
Enmascaramiento, una red codificador-decodificador para máquinas neuronales  
Traducción, RNN bidireccionales

funciones de pérdida

- basado en los aspectos internos del modelo, pérdidas y métricas basadas en el modelo Internos: Pérdidas y métricas basadas en modelos internos
- personalizado, Funciones de pérdida personalizadas
- versus métricas, métricas personalizadas

- salida, Creación de modelos complejos utilizando la API funcional

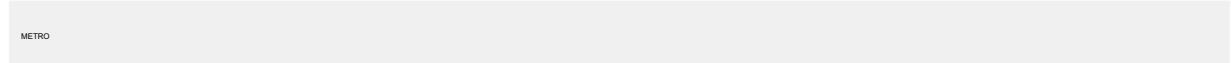
LRN (normalización de respuesta local), AlexNet

LSH (hash sensible a la localidad), proyección aleatoria

Célula LSTM (memoria a largo plazo), células LSTM - células LSTM,  
Enmascaramiento, una red codificador-decodificador para máquinas neuronales  
Traducción, RNN bidireccionales

LTU (unidades de umbral lineal), el perceptrón

Atención Luong, Mecanismos de Atención



aprendizaje automático (ML), ¿Qué es el aprendizaje automático?

- ejemplos de aplicaciones/técnicas, ejemplos de aplicaciones-  
Ejemplos de aplicaciones
- desafíos de, Principales desafíos del aprendizaje automático-paso a paso  
Atrás
- notaciones, Seleccione una medida de desempeño-Seleccione una actuación  
Medida
- lista de verificación del proyecto, Mira el panorama general
  - (consulte también el ejercicio del proyecto ML de un extremo a otro)
- Razones para su uso, ¿ Por qué utilizar el aprendizaje automático? - ¿Por qué utilizar  
el aprendizaje automático?
- recursos en, Otros recursos-Otros recursos
- Ejemplo de filtro de spam, El panorama del aprendizaje automático: ¿por qué  
utilizar el aprendizaje automático?
- prueba y validación, Prueba y validación: discrepancia de datos

- tipos de sistemas, tipos de sistemas de aprendizaje automático: aprendizaje basado en modelos y un flujo de trabajo típico de aprendizaje automático

MAE (error absoluto medio), seleccione una medida de rendimiento

predicciones de voto mayoritario, Ejercicios

make\_column\_selector(), Canalizaciones de transformación

make\_column\_transformer(), Canalizaciones de transformación

make\_pipeline(), Canalizaciones de transformación, Curvas de aprendizaje

Norma de Manhattan, seleccione una medida de desempeño

hipótesis múltiple, aprendizaje múltiple

aprendizaje múltiple, reducción de dimensiones, aprendizaje múltiple -múltiple

Aprendiendo

Estimación MAP (máximo a posteriori), selección del número de Clústeres

mAP (precisión media media), sólo miras una vez

MAPE (error porcentual absoluto medio), Predicción de una serie temporal

red de mapeo, StyleGAN, StyleGAN

MapReduce, enmarca el problema

violaciones de margen, Clasificación de margen suave, Bajo el capó de Clasificadores SVM lineales

Cadenas de Markov, Procesos de Decisión de Markov

Procesos de decisión de Markov (MDP), procesos de decisión de Markov-Procesos de decisión de Markov, políticas de exploración

máscara R-CNN, segmentación semántica

tensor de máscara, enmascaramiento

modelo de lenguaje enmascarado (MLM), una avalancha de transformadores  
Modelos

capa de atención enmascarada de múltiples cabezas, La atención es todo lo que necesita: el  
Arquitectura de transformador original

enmascaramiento, enmascaramiento-enmascaramiento, atención multicabezal

Servicio Matching Engine, Vertex AI, creación de un servicio de predicción en  
IA de vértice

Matplotlib, ejecución de ejemplos de código con Google Colab

capa de agrupación máxima, Capas de agrupación, Arquitecturas CNN

regularización de norma máxima, regularización de norma máxima

paso de maximización, mezclas gaussianas, mezclas gaussianas

estimación máxima a posteriori (MAP), seleccionando el número de  
Clústeres

estimación de máxima verosimilitud (MLE), seleccionando el número de  
Clústeres

Regularización de abandono escolar de MC (Monte Carlo), Monte Carlo (MC)  
Abandono-Monte Carlo (MC) Abandono

MCTS (búsqueda de árbol de Monte Carlo), descripción general de algunas RL populares  
Algoritmos

MDP (procesos de decisión de Markov), procesos de decisión de Markov-  
Procesos de decisión de Markov, políticas de exploración

MDS (escalado multidimensional), reducción de otras dimensiones  
Técnicas

error absoluto medio (MAE), seleccione una medida de rendimiento

error porcentual absoluto medio (MAPE), pronóstico de una serie temporal

Precisión promedio media (mAP), **solo miras una vez**  
error cuadrático medio (MSE), **regresión lineal, variacional**  
**codificadores automáticos**  
  
cambio medio, algoritmos de agrupamiento, **otros algoritmos de agrupamiento**  
  
**mean\_squared\_error()**, **entrenar y evaluar en el conjunto de entrenamiento**  
  
medida de similitud, **aprendizaje basado en instancias**  
  
ancho de banda de memoria, tarjeta GPU, **captación previa**  
  
células de memoria (células), RNN, **células de memoria, abordando el corto plazo**  
**Problema de memoria -WaveNet**  
  
**requisitos de memoria, capas convolucionales, requisitos de memoria**  
  
**Teorema de Mercer, SVM kernelizadas**  
  
**metaaprendiz, apilamiento**  
  
**metagrafías, SavedModel, Exportación de SavedModels**  
  
Escalado mínimo-máximo, **escalado de funciones y transformación.**  
  
discriminación de mini lotes, **Las dificultades de entrenar GAN**  
  
descenso de gradiente de mini lotes, **Descenso de gradiente de mini lotes-Mini-Descenso de gradiente por lotes, parada anticipada**  
  
k-medias mini-lote, k-medias aceleradas y k-medias mini-lote  
  
minilotes, **aprendizaje en línea, crecimiento progresivo de GAN**  
  
**MinMaxScaler, escalado y transformación de funciones**  
  
estrategia reflejada, paralelismo de datos, **paralelismo de datos utilizando la estrategia reflejada, entrenamiento a escala utilizando las estrategias de distribución API, entrenamiento de un modelo en un clúster de TensorFlow**  
  
Función de activación de Mish, **GELU, Swish y Mish**

regularización de mezcla, StyleGAN, StyleGANs

ML (ver aprendizaje automático)

Operaciones de aprendizaje automático (MLOps), **inicie, supervise y mantenga su Sistema**

MLE (estimación de máxima verosimilitud), **Selección del número de Clústeres**

MLM (modelo de lenguaje enmascarado), **una avalancha de transformadores Modelos**

MLP (ver perceptrones multicapa)

Conjunto de datos MNIST, MNIST-MNIST

dispositivo móvil, implementar modelo en, **implementar un modelo en un dispositivo móvil o Dispositivo integrado : implementación de un modelo en un dispositivo móvil o integrado Dispositivo**

colapso del modo, Vision Transformers, Las dificultades del entrenamiento GAN

paralelismo de modelos, **Paralelismo de modelos- Paralelismo de modelos**

parámetros del modelo, **aprendizaje basado en modelos y un flujo de trabajo típico de aprendizaje automático**

- regularización de parada anticipada, **función de formación y costes**
- en descenso de gradiente, **descenso de gradiente, descenso de gradiente por lotes**
- Mecánica del clasificador SVM lineal, **bajo el capó de SVM lineal Clasificadores**
- y actualización de variables, **Variables**
- Forma de matriz de peso, **creación del modelo utilizando el secuencial. API**

pudrición del modelo, [aprendizaje por lotes](#)

selección de modelo, [aprendizaje basado en modelos y un flujo de trabajo típico de aprendizaje automático, ajuste de hiperparámetros y selección de modelo](#)  
[Ajuste de hiperparámetros y selección de modelo](#)

servidor modelo (ver TensorFlow Serving)

calentamiento del modelo, [Implementación de una nueva versión del modelo](#)

Aprendizaje basado en modelos, [Aprendizaje basado en modelos y un flujo de trabajo típico de aprendizaje automático. Aprendizaje basado en modelos y un flujo de trabajo típico de aprendizaje automático.](#)

modos, [escalado de características y transformación](#)

optimización del impulso, [Momentum](#)

impulso  $\beta$ , [impulso](#)

Abandono de Monte Carlo (MC), [Abandono de Monte Carlo \(MC\)-Monte Carlo \(MC\) Abandono](#)

Búsqueda de árbol de Monte Carlo (MCTS), [descripción general de algunas RL populares Algoritmos](#)

MSE (error cuadrático medio), [Regresión lineal, Variacional codificadores automáticos](#)

capa de atención multicabezal, [La atención es todo lo que necesitas: el original Arquitectura transformadora, atención de múltiples cabezas -atención de múltiples cabezas](#)

codificación multi-hot, [la capa de codificación de categorías](#)

clasificación multiclase (multinomial), [clasificación multiclase-Clasificación multiclase, Clasificación MLP-Clasificación MLP](#)

escalamiento multidimensional (MDS), [reducción de otras dimensionalidades Técnicas](#)

clasificadores multilabel, [Clasificación Multilabel-Multilabel](#)

## Clasificación

perceptrones multicapa (MLP), [Introducción a los sistemas neuronales artificiales](#)

[Redes con Keras, el TensorBoard que utiliza perceptrones para](#)

[Visualización](#)

- y codificadores automáticos, [representaciones de datos eficientes, rendimiento PCA con un codificador automático lineal incompleto](#)
- y retropropagación, [el perceptrón multicapa y Propagación hacia atrás: el perceptrón multicapa y Propagación hacia atrás](#)
- devoluciones de llamada, [Uso de devoluciones de llamada-Uso de devoluciones de llamada](#)
- clasificación MLP, [Clasificación MLP-Clasificación MLP](#)
- modelos complejos, [Construcción de modelos complejos utilizando el funcional Creación de modelos complejos de API utilizando la API funcional](#)
- modelos dinámicos, [Uso de la API de subclases para crear modelos dinámicos Modelos: uso de la API de subclases para crear modelos dinámicos](#)
- clasificador de imágenes, [Creación de un clasificador de imágenes utilizando el API secuencial : uso del modelo para hacer predicciones](#)
- MLP de regresión, [MLP de regresión -MLP de regresión, Creación de un MLP de regresión utilizando la API secuencial](#)
- guardar y restaurar un modelo, [Guardar y restaurar un modelo-Guardar y restaurar un modelo](#)
- visualización con TensorBoard, [Uso de TensorBoard para Visualización: uso de TensorBoard para visualización](#)

Distribución multimodal, [escalamiento de características y transformación.](#)

transformadores multimodales, [transformadores de visión](#)

regresión logística multinomial, [Regresión Softmax-Softmax](#)

[Regresión](#)

clasificadores multisalida, [Clasificación multisalida-Multisalida](#)

[Clasificación](#)

regresión múltiple, [encuadre el problema](#)

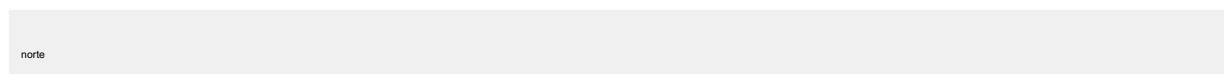
atención multiplicativa, [Mecanismos de Atención](#)

clasificación multitarea, [construcción de modelos complejos utilizando API funcional](#)

regresión multivariada, [encuadre el problema](#)

series de tiempo multivariadas, [Pronóstico de una serie de tiempo](#), [Pronóstico](#)

[Series de tiempo multivariadas : pronóstico de series de tiempo multivariadas](#)



Nadam, [Nadam](#)

NAG (gradiente acelerado de Nesterov), [acelerado de Nesterov](#)

[gradiente, nadam](#)

pronóstico ingenuo, [pronóstico de una serie temporal](#)

Equilibrio de Nash, [las dificultades de entrenar GAN](#)

procesamiento del lenguaje natural (NLP), procesamiento del lenguaje natural con RNN y biblioteca Transformers de Attention-Hugging Face

- Modelo char-RNN para generar texto, [Generando Shakespeare Texto usando un carácter RNN-RNN con estado](#)
- red codificador-decodificador para traducción automática, [An Encoder–Red decodificadora para traducción automática neuronal: un codificador Red decodificadora para traducción automática neuronal](#)
- ejemplos de aprendizaje automático, [ejemplos de aplicaciones](#)

- análisis de sentimiento, Análisis de sentimiento : reutilización de material preentrenado  
Incorporaciones y modelos de lenguaje
- clasificación de texto, análisis de sentimiento : reutilización de material previamente entrenado  
Incorporaciones y modelos de lenguaje
- codificación de texto, preprocesamiento de texto mediante lenguaje previamente entrenado  
Componentes del modelo
- resumen de texto, ejemplos de aplicaciones
- modelos de transformadores (ver modelos de transformadores)
- incrustaciones de palabras, codificación de características categóricas mediante  
Incrustaciones

comprensión del lenguaje natural (NLU), ejemplos de aplicaciones

NCCL (biblioteca de comunicaciones colectivas de Nvidia), Capacitación a escala

Uso de la API de estrategias de distribución

NEAT (neuroevolución de topologías aumentadas), Búsqueda de políticas

clase negativa, Matrices de confusión, Regresión logística

conjunto de datos anidado, preparación de los datos para modelos de aprendizaje automático

gradiente acelerado de Nesterov (NAG), acelerado de Nesterov

gradiente, nadam

Optimización del impulso de Nesterov, gradiente acelerado de Nesterov,

nadam

Traducción automática neuronal (NMT), reutilización de incrustaciones y modelos de lenguaje  
previamente entrenados: atención de múltiples cabezas

- y mecanismos de atención, Mecanismos de atención - Atención de múltiples cabezas

- con transformadores, Una avalancha de modelos de transformadores-Un  
Avalancha de modelos de transformadores

redes neuronales (ver redes neuronales artificiales)

neuroevolución de topologías aumentantes (NEAT), [búsqueda de políticas](#)

Predictión de la siguiente oración (NSP), [An Avalanche of Transformer Modelos](#)

NLU (comprensión del lenguaje natural), [ejemplos de aplicaciones](#)

NMT (ver traducción automática neuronal)

Teorema sin almuerzo gratis, [discrepancia de datos](#)

supresión no máxima, cuadros delimitadores, [detección de objetos](#)

reducción de dimensionalidad no lineal (NLDR), LLE-LLE

Clasificadores SVM no lineales, [Clasificación SVM no lineal-SVM](#)

[Clases y complejidad computacional](#)

modelos no paramétricos, [hiperparámetros de regularización](#)

datos de entrenamiento no representativos, [Datos de entrenamiento no representativos](#)

sesgo de falta de respuesta, [datos de capacitación no representativos](#)

distribución normal, [Los problemas de gradientes que desaparecen/explotan, Inicialización de Glorot y He](#)

Ecuación normal, [La ecuación normal-La ecuación normal](#)

normalización, [escalado y transformación de funciones, lotes](#)

Normalización: implementación de [la normalización por lotes con Keras, lucha contra el problema de los gradientes inestables](#)

Capa de normalización, [La capa de normalización-La normalización](#)

[Capa](#)

exponencial normalizado (función softmax), **regresión Softmax**

notaciones, **Seleccione una medida de desempeño**-**Seleccione una actuación**  
**Medida**, **regresión lineal**

detección de novedades, **aprendizaje no supervisado**, uso de mezclas gaussianas para la  
detección de anomalías

Problema NP completo, CART como, **el algoritmo de entrenamiento CART**

NSP (predicción de la siguiente oración), **An Avalanche of Transformer**  
**Modelos**

muestreo de núcleos, **generación de texto shakesperiano falso**

hipótesis nula, **hiperparámetros de regularización**

número de entradas, **creación del modelo utilizando la API secuencial**,  
**Inicialización de Glorot y He**

número de neuronas por capa oculta, **Número de neuronas por capa oculta**  
**Capa**

NumPy, **ejecución de ejemplos de código con Google Colab**

Matrices NumPy, **limpieza de datos**, transformadores personalizados, uso  
**TensorFlow como** NumPy-Otras **estructuras de datos**

Biblioteca de comunicaciones colectivas de Nvidia (NCCL), **capacitación a escala**  
**Uso de la API de estrategias de distribución**

Tarjeta GPU Nvidia, **Obtener su propia GPU**: Obtener su propia GPU

oh

OAuth 2.0, **creación de un servicio de predicción en Vertex AI**

detección de objetos, CNN, **detección de objetos**: solo miras una vez

seguimiento de objetos, CNN, **seguimiento de objetos**

puntuación de objetividad, [detección de objetos](#)  
espacio de observación, aprendizaje por refuerzo, [aprender a optimizar](#)  
[Recompensas, políticas de redes neuronales](#)  
observaciones, [Introducción a OpenAI Gym-Introducción a OpenAI](#)  
[Gimnasia](#)  
OCR (reconocimiento óptico de caracteres), [El Aprendizaje Automático](#)  
[Paisaje](#)  
OEL (aprendizaje abierto), [descripción general de algunas RL populares](#)  
[Algoritmos](#)  
algoritmo fuera de política, [Q-Learning](#)  
aprendizaje fuera de línea, [aprendizaje por lotes](#)  
algoritmo de política, [Q-Learning](#)  
SVM de una clase, otros algoritmos para la detección de anomalías y novedades  
codificación one-hot, [manejo de texto y manejo de atributos categóricos](#)  
[Atributos categóricos y de texto, la capa de codificación de categorías,](#)  
[Codificación de características categóricas mediante incrustaciones](#)  
estrategia uno contra todos (OvA), [clasificación multiclas-múlticlas](#)  
[Clasificación](#)  
estrategia uno contra uno (OvO), [Clasificación multiclas-Múlticlas](#)  
[Clasificación](#)  
estrategia uno contra el resto (OvR), [clase multiclas-](#)  
[Clasificación multiclas](#)  
Programación de 1 ciclo, [Programación de tasa de aprendizaje, Tasa de aprendizaje](#)  
[Planificación](#)  
Capas convolucionales 1D, segmentación semántica, uso de capas  
convolucionales 1D para procesar secuencias

OneHotEncoder, manejo de texto y atributos categóricos: manejo  
Atributos categóricos y de texto, escalado y transformación de funciones,  
Tuberías de transformación

SVM kernelizadas en línea, SVM kernelizadas

aprendizaje en línea, aprendizaje en línea- aprendizaje en línea

modelo en línea, DQN, objetivos de valor Q fijo

Evaluación OOB (fuera de bolsa), Evaluación fuera de bolsa -Fuera de bolsa  
Evaluación

aprendizaje abierto (OEL), descripción general de algunas RL populares  
Algoritmos

OpenAI Gym, Introducción a OpenAI Gym-Introducción a OpenAI

Gimnasia

operaciones (ops) y tensores, un recorrido rápido por TensorFlow, tensores y operaciones-  
Tensores y operaciones

reconocimiento óptico de caracteres (OCR), el aprendizaje automático  
Paisaje

valor de estado óptimo, MDP, procesos de decisión de Markov

optimizadores, optimizadores más rápidos-AdamW

- AdaGrad, AdaGrad
- Optimización de Adam, Adam
- AdaMax, AdaMax
- AdamW, AdamW
- hiperparámetros, tasa de aprendizaje, tamaño de lote y otros  
Hiperparámetros
- optimización del impulso, Momentum

- Nadam, [Nadam](#)

- gradiente acelerado de Nesterov, [gradiente acelerado de Nesterov](#)

- capa de salida, [una red codificadora-decodificadora para máquinas neuronales](#)  
[Traducción](#)

- RMSProp, [RMSProp](#)

Oracle, [ajuste fino de los hiperparámetros de la red neuronal](#)

orden de integración (d) hiperparámetro, [la familia de modelos ARMA](#)

OrdinalEncoder, [manejo de texto y atributos categóricos](#)

matriz ortogonal, [una red codificador-decodificador para máquinas neuronales](#)  
[Traducción](#)

evaluación fuera de bolsa (OOB), [evaluación fuera de bolsa](#) -fuera de bolsa  
[Evaluación](#)

aprendizaje fuera del núcleo, [aprendizaje en línea](#)

error fuera de muestra, [Pruebas y Validaciones](#)

detección de valores atípicos (ver detección de anomalías)

valores atípicos, [técnicas de aprendizaje no supervisadas](#)

puerta de salida, LSTM, [celdas LSTM](#)

capa de salida, red neuronal, [una red codificadora-decodificadora para](#)  
[Traducción automática neuronal](#)

Estrategia OvA (uno contra todos), [Clasificación Multiclas-Multiclas](#)  
[Clasificación](#)

codificador automático sobrecompleto, [codificadores automáticos convolucionales](#)

sobreajuste de datos, [sobreajuste de los datos de entrenamiento: sobreajuste de los datos](#)

[Datos de entrenamiento, crear un conjunto de prueba](#)

- evitando mediante la regularización, [evitando el sobreajuste mediante Regularización-Max-Norma Regularización](#)
- y árboles de decisión, [hiperparámetros de regularización, Regresión](#)
- y regularización de deserción, [Abandono](#)
- Hiperparámetro gamma ( $\gamma$ ) para ajustar, [kernel RBF gaussiano](#)
- Clasificación de imágenes, [entrenamiento y evaluación del modelo.](#)
- curvas de aprendizaje para evaluar, [Curvas de Aprendizaje -Curvas de Aprendizaje](#)
- número de neuronas por capa oculta, [número de neuronas por Capa oculta](#)
- regresión polinómica, [modelos de entrenamiento](#)
- Modelo SVM, [clasificación de margen suave](#)

Estrategia OvO (uno contra uno), [Clasificación Multiclas-Multiclas Clasificación](#)

Estrategia OvR (uno contra el resto), [clasificación multiclas. Clasificación multiclas](#)

PAG

Valor p, [hiperparámetros de regularización](#)

PACF (función de autocorrelación parcial), [la familia de modelos ARMA](#)

opciones de relleno, capa convolucional, [Implementación de convolucional Capas con Keras que implementan capas convolucionales con Keras](#)

PaLM (modelo de lenguaje Pathways), [una avalancha de transformadores Modelos](#)

Pandas, ejecutando los ejemplos de código usando Google Colab, busque Correlaciones: búsqueda de correlaciones, manejo de texto y categórico Atributos, escalamiento de características y transformación paralelismo, modelos de entrenamiento entre dispositivos, modelos de entrenamiento entre dispositivos Múltiples dispositivos: ajuste de hiperparámetros en Vertex AI

- paralelismo de datos, Paralelismo de datos- saturación de ancho de banda
- API de estrategias de distribución, capacitación a escala utilizando la API de estrategias de distribución
- con GPU, ejecución paralela en varios dispositivos: en paralelo Ejecución en múltiples dispositivos
- ajuste de hiperparámetros, Ajuste de hiperparámetros en Vertex AI- Ajuste de hiperparámetros en Vertex AI
- paralelismo de modelos, Paralelismo de modelos- Paralelismo de modelos
- en el clúster de TensorFlow, Entrenamiento de un modelo en TensorFlow Entrenamiento de clústeres de un modelo en un clúster de TensorFlow
- Vertex AI para ejecutar trabajos grandes, ejecutar trabajos de capacitación grandes en Vertex AI ejecuta grandes trabajos de capacitación en Vertex AI

eficiencia de parámetros, Número de capas ocultas matriz de parámetros, regresión Softmax servidores de parámetros, Paralelismo de datos con parámetros centralizados espacio de parámetros, descenso de gradiente vector de parámetros, regresión lineal, descenso de gradiente, función de costo y entrenamiento, regresión Softmax ReLU con fugas paramétricas (PReLU), ReLU con fugas modelos paramétricos, hiperparámetros de regularización

función de autocorrelación parcial (PACF), la familia de modelos ARMA

derivada parcial, descenso de gradiente por lotes

parcial\_fit(), descenso de gradiente estocástico

Modelo de lenguaje Pathways (PaLM), una avalancha de transformadores Modelos

PCA (ver análisis de componentes principales)

PDF (función de densidad de probabilidad), aprendizaje no supervisado

Técnicas, selección del número de conglomerados

r de Pearson, buscar correlaciones

sanciones, aprendizaje por refuerzo, aprendizaje por refuerzo

PER (repetición de experiencia priorizada), repetición de experiencia priorizada

Perceptor, transformadores de visión.

percentiles, eche un vistazo rápido a la estructura de datos

perceptrones, Introducción a las redes neuronales artificiales con Keras,

Los MLP de clasificación de perceptrones

- (ver también perceptrones multicapa)

medidas de desempeño, Medidas de desempeño: la curva ROC

- matriz de confusión, Matrices de confusión -Matrices de confusión

- validación cruzada para medir la precisión, Medición de la precisión

Uso de la precisión de medición de validación cruzada

Validación

- precisión y recuperación, Precisión y recuperación: la precisión/recuperación Compensación

- Curva ROC, La curva ROC-La curva ROC

- seleccionar, **Seleccionar una medida de desempeño**-**Seleccionar un desempeño Medida**

programación de rendimiento, **programación de tasa de aprendizaje**, **tasa de aprendizaje Planificación**

permutación(), **crear un conjunto de prueba**

Algoritmo PG (gradientes de políticas), búsqueda de políticas, gradientes de políticas **gradientes de políticas**

programación constante por partes, **programación de tasa de aprendizaje**, **aprendizaje Programación de tarifas**

PipeDream, **saturación de ancho de banda**

Clase de canalización, **canalizaciones de transformación**

Constructor de oleoductos, **Transformación Pipelines**-**Transformación Tuberías**

oleoductos, **enmarcar el problema**, **oleoductos de transformación-**

Canales de transformación, curvas de aprendizaje, margen blando  
**Clasificación**

capa de normalización por píxeles, **crecimiento progresivo de GAN**

marcadores de posición, definiciones de funciones, **exploración de definiciones de funciones y Graficos**

Algoritmo POET, **descripción general de algunos algoritmos RL populares**

algoritmo de gradientes de políticas (PG), **búsqueda de políticas**, **gradientes de políticas-**  
**gradientes de políticas**

parámetros de política, **búsqueda de políticas**

espacio de políticas, **Búsqueda de políticas**

política, aprendizaje por refuerzo, aprendizaje por refuerzo, política

Búsqueda, políticas de redes neuronales

características polinomiales, clasificadores SVM, kernel polinómico

kernel polinomial, Kernel polinomial, SVM kernelizadas

regresión polinómica, modelos de entrenamiento, regresión polinómica-

Regresión polinomial

tiempo polinomial, el algoritmo de entrenamiento CART

Funciones polinómicas, regresión polinómica, SVM no lineal

Clasificación

núcleo de agrupación, capas de agrupación

capas de agrupación, Capas de agrupación : implementación de capas de agrupación con Keras

codificaciones posicionales, Codificaciones posicionales -Codificaciones posicionales

clase positiva, Matrices de confusión, Regresión logística

Cuantización posterior al entrenamiento, implementación de un modelo en un dispositivo móvil o

Dispositivo integrado

distribución posterior, Autocodificadores variacionales

Distribución de ley de potencia, escalamiento de características y transformación.

programación de energía, programación de tasa de aprendizaje

PPO (optimización de políticas próximas), descripción general de algunas RL populares

Algoritmos

precisión y recuperación, métricas clasificadoras, precisión y recuperación:

Compensación entre precisión y recuperación

curva de precisión/recuperación (PR), la compensación precisión/recuperación, la República de China

Curva

compensación precisión/recuperación, [La compensación precisión/recuperación: la Compensación entre precisión y recuperación](#)

[predecir\(\)](#), [Limpiar los datos](#), [Transformadores personalizados](#), [Transformación Tuberías](#)

clase predicha, [matrices de confusión](#)

servicio de predicción, en Vertex AI, [Creación de un servicio de predicción en Trabajos de predicción por lotes ejecutados por Vertex AI en Vertex AI](#)

predicciones

- retropropagación, [el perceptrón multicapa y Propagación hacia atrás](#)
- matriz de confusión, [Matrices de confusión -Matrices de confusión](#)
- validación cruzada para medir la precisión, [Medición de la precisión Uso de la precisión de medición de validación cruzada Validación](#)
- árboles de decisión, [Haciendo Predicciones-El Entrenamiento CART Algoritmo](#)
- con clasificador SVM lineal, [bajo el capó de SVM lineal Clasificadores](#)

predictores, [aprendizaje supervisado](#)

- (ver también aprendizaje en conjunto)

[predict\\_log\\_proba\(\)](#), [Clasificación de margen suave](#)

[predict\\_proba\(\)](#), [Clasificación de margen suave](#)

precarga de datos, [Prefetching-Prefetching](#)

[PReLU \(ReLU paramétrico con fugas\)](#), [Leaky ReLU](#)

atributos preprocesados, [eche un vistazo rápido a la estructura de datos](#)

preprocesamiento de datos (ver carga y preprocesamiento de datos)

discrepancia en el preprocesamiento, la capa de normalización

preentrenamiento y capas preentrenadas

- en tarea auxiliar, Preentrenamiento en una tarea auxiliar
- CNN, uso de modelos previamente entrenados a partir de modelos previamente entrenados por Keras para el aprendizaje por transferencia
- preentrenamiento codicioso en capas, preentrenamiento no supervisado, Entrenamiento de un codificador automático a la vez, crecimiento progresivo de GAN
- Componentes del modelo de lenguaje, Uso de lenguaje previamente entrenado Componentes del modelo
- reutilizar incrustaciones, Reutilizar incrustaciones previamente entrenadas y Modelos de lenguaje : reutilización de incrustaciones previamente entrenadas y Modelos de lenguaje
- reutilizar capas, Reutilizar capas previamente entrenadas: preentrenamiento en un Tarea auxiliar
- en aprendizaje no supervisado, Preentrenamiento no supervisado, Reutilización Incorporaciones y modelos de lenguaje previamente entrenados, una avalancha de Modelos de transformadores, preentrenamiento no supervisado utilizando apilados codificadores automáticos

problema primordial, el problema dual

componente principal (PC), Componentes Principales -Principal Componentes

Análisis de componentes principales (PCA), PCA-PCA incremental

- elegir el número de dimensiones, elegir el número correcto de Dimensiones: elegir el número correcto de dimensiones

- para compresión, [PCA para compresión](#)-[PCA para compresión](#)
- relación de varianza explicada, [relación de varianza explicada](#)
- encontrar componentes principales, [Componentes principales](#)
- PCA incremental, PCA [incremental](#) -[PCA incremental](#)
- preservando la variación, [Preservando la variación](#)
- Proyectando hasta d dimensiones, [Proyectando hasta d Dimensiones](#)
- PCA aleatorizado, [PCA aleatorizado](#)
- para escalar datos en árboles de decisión, [sensibilidad a la orientación del eje](#)
- con codificador automático lineal incompleto, [Realización de PCA con un PCA de rendimiento de codificador automático lineal incompleto](#) con un [Codificador automático lineal subcompleto](#)
- usando Scikit\_Learn para, [Usando Scikit-Learn](#)

distribución previa, [Autocodificadores variacionales](#)

repetición de experiencia priorizada (PER), [repetición de experiencia priorizada](#)

Autocodificadores probabilísticos, [Autocodificadores variacionales](#)

probabilidades, estimación, [Estimación de probabilidades](#)-estimación

Probabilidades, estimación de probabilidades de clase, clasificadores de votación

función de densidad de probabilidad (PDF), aprendizaje no supervisado

Técnicas, selección del número de conglomerados

probabilidad versus verosimilitud, [selección del número de grupos](#)

Seleccionar el número de grupos

perfilar la red, con TensorBoard, [Usar TensorBoard para](#)

Visualización

aplicación web progresiva (PWA), ejecución de un modelo en una página web

proyección, reducción de dimensionalidad, Proyección-Proyección

Lógica proposicional, de las neuronas biológicas a las artificiales

buffers de protocolo (protobuf), una breve introducción a los buffers de protocolo-TensorFlow Protobufs, manejo de listas de listas utilizando el SequenceExample Protobuf, consulta de servicio TF a través de gRPC API

Optimización de políticas próximas (PPO), descripción general de algunas RL populares Algoritmos

poda de nodos del árbol de decisión, hiperparámetros de regularización

pseudoinversa, la ecuación normal

PWA (aplicación web progresiva), Ejecución de un modelo en una página web

API de Python, obtenga los datos

q

Algoritmo Q-learning, Q-Learning-Dueling DQN

- Q-learning aproximado, Q-Learning aproximado y Q-profundo Aprendiendo
- políticas de exploración, Políticas de Exploración
- Implementación de Q-Learning profundo, Implementación de Q-Learning profundo-Implementación de Q-Learning profundo
- variantes en Q-learning profundo, Duelo de variantes de Q-Learning profundo DQN

Algoritmo de iteración del valor Q, procesos de decisión de Markov-Markov Procesos de decisión

Valores Q, procesos de decisión de Markov- Procesos de decisión de Markov

ecuación cuadrática, regresión polinómica

Problemas de programación cuadrática (QP), bajo el capó de la linealidad

Clasificadores SVM

capacitación consciente de la cuantificación, implementación de un modelo en un dispositivo móvil o

Dispositivo integrado

cuartiles, eche un vistazo rápido a la estructura de datos

consultas por segundo (QPS), entrenamiento e implementación de TensorFlow

Modelos a escala, creación de un servicio de predicción en Vertex AI

módulos de preguntas y respuestas, ejemplos de aplicaciones

colas, Otras estructuras de datos, Colas

R

función de base radial (RBF), escalado y transformación de características

dimensiones irregulares, otras estructuras de datos

Agente arcoíris, duelo contra DQN

bosques aleatorios, mejor evaluación mediante validación cruzada, conjunto

Aprendizaje y bosques aleatorios, bosques aleatorios : importancia de las características

- análisis de modelos y sus errores, Análisis de los mejores modelos y sus errores
- árboles de decisión (ver árboles de decisión)
- árboles adicionales, árboles adicionales
- medición de la importancia de la característica, Importancia de la característica

inicialización aleatoria, descenso de gradiente, descenso de gradiente

parches aleatorios, parches aleatorios y subespacios aleatorios

algoritmo de proyección aleatoria, Proyección aleatoria- Proyección aleatoria

subespacios aleatorios, parches aleatorios y subespacios aleatorios

RandomForestClassifier, la curva ROC-La curva ROC

RandomForestRegressor, mejor evaluación mediante validación cruzada

ReLU con fugas aleatorias (RReLU), ReLU con fugas

PCA aleatorizado, PCA aleatorizado

búsqueda aleatoria, búsqueda aleatoria -búsqueda aleatoria, fina-

Ajuste de los hiperparámetros de la red neuronal: ajuste fino de los nervios

Hiperparámetros de red

RBF (función de base radial), escalado y transformación de características

rbf\_kernel(), Escalado y transformación de funciones, personalizado  
transformadores

métrica de recuperación, precisión y recuperación

curva de característica operativa del receptor (ROC), la curva ROC: la  
Curva ROC

red de reconocimiento, representaciones de datos eficientes

- (ver también codificadores automáticos)

sistema de recomendación, ejemplos de aplicaciones

error de reconstrucción, PCA para compresión, atado de pesas

pérdida de reconstrucción, pérdidas y métricas basadas en elementos internos del modelo,  
Representaciones de datos eficientes

unidades lineales rectificadas (ReLU) (ver ReLU)

abandono recurrente, procesamiento de secuencias utilizando RNN y CNN

normalización de capas recurrentes, procesamiento de secuencias utilizando RNN y CNN, lucha contra el problema de los gradientes inestables

redes neuronales recurrentes (RNN), procesamiento de secuencias utilizando Búsqueda por haz de RNN y CNN

- bidireccional, RNN bidireccionales
- RNN profundo, pronóstico utilizando un RNN profundo
- pronóstico de series de tiempo (ver datos de series de tiempo)
- recorte de degradado, recorte de degradado
- Manejo de secuencias largas, Manejo de secuencias largas-WaveNet
- secuencias de entrada y salida, Secuencias de entrada y salida- Secuencias de entrada y salida
- células de memoria, Células de memoria, Abordar la memoria a corto plazo Problema-WaveNet
- PNL (ver procesamiento del lenguaje natural)
- y Perceptor, Transformadores de Visión
- división entre dispositivos, paralelismo de modelos
- Procesamiento del lenguaje natural con estado con RNN y atención, RNN con estado -RNN con estado
- Procesamiento de lenguaje natural sin estado con RNN y Atención, RNN con estado
- formación, Formación RNN
- y transformadores de visión, Transformadores de visión

neuronas recurrentes, Neuronas Recurrentes y Capas

reducir la operación, el paralelismo de datos utilizando la estrategia reflejada

red de propuesta de región (RPN), solo miras una vez

MLP de regresión, MLP de regresión -MLP de regresión

modelos de regresión

- y clasificación, Aprendizaje supervisado, Multisalida Clasificación
- Tareas de árbol de decisión, Regresión-Regresión.
- ejemplo de previsión, ejemplos de aplicaciones
- regresión de lazo, regresión de lazo- regresión de lazo
- regresión lineal (ver regresión lineal)
- regresión logística (ver regresión logística)
- regresión múltiple, encuadre el problema
- regresión multivariada, encuadre el problema
- regresión polinómica, modelos de entrenamiento, regresión polinómica- Regresión polinomial
- MLP de regresión, Creación de un MLP de regresión utilizando el API secuencial
- regresión de crestas, regresión de crestas- regresión de crestas, elástica Regresión neta
- regresión softmax, regresión softmax- regresión softmax
- SVM, regresión SVM- regresión SVM
- regresión univariada, encuadre el problema

regresión a la media, aprendizaje supervisado

regularización, sobreajuste de los datos de entrenamiento, evitar el sobreajuste

Mediante regularización- regularización máxima-norma

- regularizadores personalizados, funciones de activación personalizadas, inicializadores,  
**Regularizadores y restricciones**

- árboles de decisión, **hiperparámetros de regularización**
- abandono, abandono-abandono
- parada temprana, **parada temprana- parada temprana**, gradiente  
**Impulsar y pronosticar utilizando un modelo lineal**
- red elástica, **regresión neta elástica**
- hiperparámetros, **Hiperparámetros de regularización-**  
**Hiperparámetros de regularización**
- regresión de lazo, **regresión de lazo- regresión** de lazo
- modelos lineales, Modelos **lineales regularizados** : parada temprana
- norma máxima, **regularización de norma máxima**
- Abandono de MC, **Montecarlo (MC)** Abandono-Monte Carlo (MC)  
**Abandonar**
- cresta, **regresión de cresta**
- contracción, **aumento de gradiente**
- subpalabra, **Análisis de sentimiento**
- Tikhonov, **Regresión de crestas- Regresión** de crestas
- Decaimiento de peso, **AdamW**
- Regularización  $\ell_1$  y  $\ell_2$ , **Regularización**  $\ell_1$  y  $\ell_2$

Algoritmos REINFORCE, **gradientes de políticas**

aprendizaje por refuerzo (RL), **aprendizaje por refuerzo, refuerzo**

Descripción general del aprendizaje **de algunos algoritmos RL populares**

- acciones, [Evaluación de acciones: el problema de la asignación de crédito](#)  
[Evaluación de acciones: el problema de la asignación de crédito](#)
- problema de asignación de crédito, [Evaluación de acciones: el crédito](#)  
[Acciones de evaluación de problemas de asignación : la asignación de crédito](#)  
[Problema](#)
- ejemplos de, [Ejemplos de Aplicaciones, Aprender a Optimizar Recompensas](#)
- aprender para optimizar las recompensas, [aprender a optimizar Recompensas](#)
- Procesos de decisión de Markov, [Procesos de decisión de Markov](#)  
[Procesos de decisión de Markov](#)
- políticas de redes neuronales, [Políticas de redes neuronales](#)
- OpenAI Gym, [Introducción a OpenAI Gym-Introducción a Gimnasio OpenAI](#)
- Algoritmos PG, [gradientes de políticas- gradientes de políticas](#)
- búsqueda de políticas, [búsqueda de políticas](#)
- Q-learning, [Q-Learning-Duelo DQN](#)
- Aprendizaje TD, [aprendizaje por diferencia temporal](#)

### ReLU (unidades lineales rectificadas)

- y retropropagación, [el perceptrón multicapa y Propagación hacia atrás](#)
- en arquitecturas CNN, [Arquitecturas CNN](#)
- por defecto para tareas simples, [GELU, Swish y Mish](#)
- LeakyReLU, [Leaky ReLU-Leaky ReLU](#)
- y MLPs, [MLP de regresión](#)

- Problema de gradientes inestables de RNN, [luchando contra lo inestable](#)  
[Problema de gradientes](#)

- RReLU, [ReLU con fugas](#)

- ajuste de hiperparámetros, [ajuste fino de la red neuronal](#)  
[Hiperparámetros](#)

búfer de reproducción, [Implementación de Q-Learning profundo](#)

Aprendizaje de representación, [manejo de texto y atributos categóricos.](#)

- (ver también codificadores automáticos)

bloque residual, Modelos [personalizados - Modelos personalizados](#)

errores residuales, aumento de gradiente - [aumento de gradiente](#)

aprendizaje residual, [ResNet](#)

red residual (ResNet), ResNet-ResNet

unidades residuales, [ResNet](#)

ResNet-152, [ResNet](#)

ResNet-34, [ResNet](#), implementación de una CNN ResNet-34 usando Keras

ResNet-50, utilizando modelos previamente entrenados de Keras

ResNeXt, otras arquitecturas destacadas

API REST, consulta a través de, [Consulta TF Sirviendo a través de](#)  
[API DESCANSO](#)

retorno, en aprendizaje por refuerzo, [gradientes de políticas](#)

proceso inverso, modelo de difusión, Modelos de Difusión -Modelos de Difusión

autodiff en modo inverso, [el perceptrón multicapa y](#)

Propagación hacia atrás, cálculo de gradientes mediante Autodiff

recompensas, aprendizaje por refuerzo, [aprendizaje por refuerzo](#), aprender a Optimizar recompensas

Cresta, [regresión de cresta](#)

regresión de crestas, [regresión de crestas](#)- regresión de crestas, red elástica Regresión

regularización de crestas, [regresión de crestas](#)

RidgeCV, [regresión de crestas](#)

RL (ver aprendizaje por refuerzo)

RMSProp, [RMSProp](#)

Curva ROC (características operativas del receptor), [la curva ROC](#): la Curva ROC

error cuadrático medio (RMSE), [seleccione una medida de rendimiento](#)

Seleccione una medida de desempeño, capacítese y evalúe la capacitación

Conjunto, regresión lineal, parada anticipada

nodo raíz, árbol de decisión, [Hacer predicciones](#), Hacer predicciones

RPN (red de propuesta de región), [Sólo miras una vez](#)

RReLU (ReLU con fugas aleatorizado), [Leaky ReLU](#)

S

SAC (actor-crítico suave), descripción general de algunos algoritmos RL populares

“mismo” relleno, visión por computadora, [implementación convolucional](#)

Capas con Keras

SAMME, AdaBoost

muestra de ineficiencia, [gradientes de políticas](#)

softmax muestreado, una red codificadora-decodificadora para máquinas neuronales

## Traducción

sesgo de muestreo, datos de entrenamiento no representativos, creación de un conjunto de pruebas

ruido de muestreo, datos de entrenamiento no representativos

Modelo SARIMA, La familia de modelos ARMA-La familia de modelos ARMA

SavedModel, Exportación de SavedModels-Exportación de SavedModels

guardar, cargar y restaurar modelos, guardar y restaurar un

Guardar y restaurar un modelo, guardar y cargar modelos

Que contienen componentes personalizados: guardar y cargar modelos que

Contiene componentes personalizados

capa de atención de producto escalado, atención de múltiples cabezales

matriz de dispersión, buscar correlaciones-buscar correlaciones

Scikit-Learn, objetivo y enfoque

- embolsar y pegar, Embolsar y pegar en Scikit-Learn
- Algoritmo CART, Algoritmo de entrenamiento CART, Regresión
- validación cruzada, mejor evaluación mediante validación cruzada: mejor  
Evaluación mediante validación cruzada
- principios de diseño, Limpiar los datos-Limpiar los datos
- Implementación de PCA, uso de Scikit-Learn
- Constructor de oleoductos, Transformación Pipelines-Transformación  
Tuberías
- sklearn.base.BaseEstimator, Transformadores personalizados
- sklearn.base.clone(), parada anticipada
- sklearn.base.TransformerMixin, Transformadores personalizados

- `sklearn.cluster.DBSCAN`, DBSCAN
- `sklearn.cluster.KMeans`, transformadores personalizados, k-means
- `sklearn.cluster.MiniBatchKMeans`, k-medias aceleradas y k-medias mini-batch
- `sklearn.compose.ColumnTransformer`, canalizaciones de transformación
- `sklearn.compose.TransformedTargetRegressor`, escalado y transformación de funciones
- `sklearn.datasets.load_iris()`, Límites de decisión
- `sklearn.datasets.make_moons()`, Clasificación SVM no lineal
- `sklearn.decomposition.PCA incremental`, PCA incremental
- `sklearn.decomposition.PCA`, uso de Scikit-Learn
- `sklearn.ensemble.AdaBoostClassifier`, AdaBoost
- `sklearn.ensemble.BaggingClassifier`, embolsado y pegado en Scikit-Learn
- `sklearn.ensemble.GradientBoostingRegressor`, aumento de gradiente - aumento de gradiente
- `sklearn.ensemble.HistGradientBoostingClassifier`, aumento de gradiente basado en histograma
- `sklearn.ensemble.HistGradientBoostingRegressor`, aumento de gradiente basado en histograma
- `sklearn.ensemble.RandomForestClassifier`, La curva ROC-La curva ROC, Clasificadores de votación, Bosques aleatorios, Importancia de las características, Elección del número correcto de dimensiones
- `sklearn.ensemble.RandomForestRegressor`, mejor evaluación mediante validación cruzada, bosques aleatorios

- `sklearn.ensemble.StackingClassifier`, Apilamiento
- `sklearn.ensemble.StackingRegressor`, Apilamiento
- `sklearn.ensemble.VotingClassifier`, Clasificadores de votación
- `sklearn.externals.joblib`, Inicie, supervise y mantenga su sistema:  
Inicie, supervise y mantenga su sistema
- `sklearn.feature_selection.SelectFromModel`, analizando los mejores modelos y sus errores
- `sklearn.impute.IterativeImputer`, limpiar los datos
- `sklearn.impute.KNNImputer`, Limpiar los datos
- `sklearn.impute.SimpleImputer`, limpiar los datos
- `sklearn.linear_model.ElasticNet`, regresión neta elástica
- `sklearn.linear_model.Lasso`, regresión de lazo
- `sklearn.linear_model.LinearRegression`, aprendizaje basado en modelos y un flujo de trabajo típico de aprendizaje automático, limpieza de datos, escalamiento y transformación de características, la ecuación normal, complejidad computacional, regresión polinómica
- `sklearn.linear_model.LogisticRegression`, límites de decisión, regresión Softmax, uso de agrupación en clústeres para el aprendizaje semisupervisado
- `sklearn.linear_model.Perceptron`, El Perceptrón
- `sklearn.linear_model.Ridge`, regresión de crestas
- `sklearn.linear_model.RidgeCV`, regresión de crestas
- `sklearn.linear_model.SGDClassifier`, Entrenamiento de un clasificador binario, Compensación precisión/recuperación, Compensación precisión/recuperación, Curva ROC- Curva ROC, Clasificación multiclase, SVM

Clases y complejidad computacional, bajo el capó de  
Clasificadores lineales SVM, el perceptrón

- `sklearn.linear_model.SGDRegressor`, **descenso de gradiente estocástico, parada anticipada**
- `sklearn.manifold.LocallyLinearEmbedding`, **LLE**
- `sklearn.metrics.ConfusionMatrixDisplay`, **Análisis de errores**
- `sklearn.metrics.confusion_matrix()`, **Matrices de confusión, Análisis de errores**
- `sklearn.metrics.f1_score()`, **Precisión y recuperación, Clasificación multietiqueta**
- `sklearn.metrics.mean_squared_error()`, **entrenar y evaluar en el conjunto de entrenamiento**
- `sklearn.metrics.precision_recall_curve()`, **La compensación entre precisión y recuperación, La curva ROC**
- `sklearn.metrics.precision_score()`, **Precisión y recuperación**
- `sklearn.metrics.recall_score()`, **Precisión y recuperación**
- `sklearn.metrics.roc_auc_score()`, **La curva ROC**
- `sklearn.metrics.roc_curve()`, **La curva ROC**
- `sklearn.metrics.silhouette_score()`, **Encontrar el número óptimo de clústeres**
- `sklearn.mixture.BayesianGaussianMixture`, **modelos de mezcla bayesiana gaussiana**
- `sklearn.mixture.GaussianMixture`, **Mezclas gaussianas**
- `sklearn.model_selection.cross_val_predict()`, **Matrices de confusión, El equilibrio precisión/recuperación, La curva ROC, Error**

## Análisis, apilamiento

- `sklearn.model_selection.cross_val_score()`, Mejor evaluación mediante validación cruzada, medición de la precisión mediante validación cruzada
- `sklearn.model_selection.GridSearchCV`, Búsqueda de cuadrícula- Búsqueda de cuadrícula
- `sklearn.model_selection.learning_curve()`, Curvas de aprendizaje
- `sklearn.model_selection.RandomizedSearchCV`, elección del número correcto de dimensiones
- `sklearn.model_selection.StratifiedKFold`, medición de la precisión mediante validación cruzada
- `sklearn.model_selection.StratifiedShuffleSplit`, crear un conjunto de prueba
- `sklearn.model_selection.train_test_split()`, Crear un conjunto de pruebas, Crear un conjunto de pruebas, Mejor evaluación mediante validación cruzada
- `sklearn.multiclass.OneVsOneClassifier`, clasificación multiclas
- `sklearn.multiclass.OneVsRestClassifier`, clasificación multiclas
- `sklearn.multioutput.ChainClassifier`, clasificación de etiquetas múltiples
- `sklearn.neighbors.KNeighborsClassifier`, clasificación multietiqueta, clasificación multisalida, DBSCAN
- `sklearn.neighbors.KNeighborsRegressor`, aprendizaje basado en modelos y un flujo de trabajo típico de aprendizaje automático
- `sklearn.neural_network.MLPClassifier`, Clasificación de MLP
- `sklearn.neural_network.MLPRegressor`, MLP de regresión
- `sklearn.pipeline.make_pipeline()`, Curvas de aprendizaje
- `sklearn.pipeline.Pipeline`, Canalizaciones de transformación

- `sklearn.preprocessing.FunctionTransformer`, Transformadores personalizados
- `sklearn.preprocessing.MinMaxScaler`, escalado y transformación de funciones
- `sklearn.preprocessing.OneHotEncoder`, Manejo de texto y atributos categóricos-Manejo de texto y atributos categóricos, Escalado y transformación de funciones, Canales de transformación
- `sklearn.preprocessing.OrdinalEncoder`, manejo de texto y atributos categóricos, aumento de gradiente basado en histograma
- `sklearn.preprocessing.PolynomialFeatures`, regresión polinómica, clasificación SVM no lineal
- `sklearn.preprocessing.StandardScaler`, escalado y transformación de características, descenso de gradiente, regresión de crestas, clasificación SVM no lineal
- `sklearn.random_projection.GaussianRandomProjection`, proyección aleatoria
- `sklearn.random_projection.SparseRandomProjection`, proyección aleatoria
- `sklearn.semi_supervised.LabelPropagation`, uso de agrupaciones en clústeres para el aprendizaje semisupervisado
- `sklearn.semi_supervised.LabelSpreading`, uso de agrupaciones en clústeres para el aprendizaje semisupervisado
- `sklearn.semi_supervised.SelfTrainingClassifier`, uso de agrupación en clústeres para el aprendizaje semisupervisado
- `sklearn.svm.LinearSVC`, clasificación de margen suave, clases SVM y complejidad computacional, bajo el capó de clasificadores SVM lineales

- `sklearn.svm.SVC`, **clasificación multiclas, núcleo polinomial, clases SVM y complejidad computacional, bajo el capó de clasificadores SVM lineales**
- `sklearn.svm.SVR`, **regresión SVM**
- `sklearn.tree.DecisionTreeClassifier`, **entrenamiento y visualización de un árbol de decisión, ¿ impureza o entropía de Gini?, hiperparámetros de regularización, sensibilidad a la orientación del eje, bosques aleatorios**
- `sklearn.tree.DecisionTreeRegressor`, **entrenar y evaluar en el conjunto de entrenamiento, árboles de decisión, regresión, aumento de gradiente**
- `sklearn.tree.export_graphviz()`, **Entrenamiento y visualización de un árbol de decisiones**
- `sklearn.tree.ExtraTreesClassifier`, **árboles adicionales**
- `sklearn.utils.estimator_checks`, **Transformadores personalizados**
- Módulo `sklearn.utils.validation`, **Transformadores personalizados**
- Clases de clasificación SVM, **Clases SVM y Computacionales**  
**Complejidad**

`puntuación()`, **Limpiar los datos**

motores de búsqueda, agrupamiento para, **Algoritmos de agrupamiento: k-medias y DBSCAN**

espacio de búsqueda, **búsqueda aleatoria**

estacionalidad, modelado de series temporales, **Predicción de una serie temporal**

segundo momento (variación del gradiente), **Adam**

derivadas parciales de segundo orden (Hessianas), **AdamW**

incrustación de segmentos, **una avalancha de modelos de transformadores**

SelectFromModel, analizando los mejores modelos y sus errores

capas de autoatención, La atención es todo lo que necesitas: el original  
Arquitectura transformadora, Atención multicabezal, Transformadores de visión

autodestilación, Vision Transformers

autonormalización, ELU y SELU, Abandono, Resumen y Práctico  
Pautas

aprendizaje autosupervisado, aprendizaje autosupervisado -aprendizaje autosupervisado

Función de activación SELU (ELU escalada), ELU y SELU, abandono

interpolación semántica, generación de imágenes MNIST de moda

segmentación semántica, Ejemplos de aplicaciones, Uso de clustering para la  
segmentación de imágenes, Segmentación semántica-Semántica  
Segmentación

Aprendizaje semisupervisado, Aprendizaje semisupervisado, Agrupación  
Algoritmos: k-means y DBSCAN, uso de agrupación en clústeres para semi-  
Aprendizaje supervisado: uso de agrupaciones para el aprendizaje semisupervisado

SENet, SENet-SENet

sensibilidad (recuerdo), curva ROC, La curva ROC

métrica de sensibilidad, matrices de confusión

sensores, Aprendiendo a optimizar las recompensas

codificador de oraciones, uso de componentes del modelo de lenguaje previamente entrenado,  
Reutilización de incrustaciones y modelos de lenguaje previamente entrenados

Proyecto SentencePieza, Análisis de Sentimiento

análisis de sentimiento, Análisis de sentimiento : reutilización de material preentrenado  
Incorporaciones y modelos de lenguaje

neurona de sentimiento, RNN con estado

capa de convolución separable, Xception

longitud de secuencia, uso de capas convolucionales 1D para procesar secuencias, codificaciones posicionales

Red secuencia a secuencia (seq2seq), entrada y salida

Secuencias, pronóstico utilizando un modelo de secuencia a secuencia.

Pronóstico utilizando un modelo de secuencia a secuencia

red de secuencia a vector, secuencias de entrada y salida

SequenceExample protobuf, Manejo de listas de listas usando el SecuenciaEjemplo Protobuf

API secuencial, clasificador de imágenes con, creación de un clasificador de imágenes

Uso de la API secuencial: uso del modelo para hacer predicciones

cuenta de servicio, GCP, creación de un servicio de predicción en Vertex AI

trabajador de servicio, ejecutando un modelo en una página web

conjuntos, conjuntos

set\_params(), Transformadores personalizados

SGD (ver descenso de gradiente estocástico)

SGDClassifier, entrenamiento de un clasificador binario, precisión/recuperación

Compensación, la compensación precisión/recuperación, la curva ROC-La República de China

Curva, Clasificación Multiclasa, Clases SVM y Computacional

Complejidad, bajo el capó de los clasificadores SVM lineales, bajo el Campana de clasificadores SVM lineales

SGDRegressor, descenso de gradiente estocástico, parada anticipada

afilado, transformadores de PNL, transformadores de visión

contracción, aumento de gradiente

shuffle\_and\_split\_data(), Crear un conjunto de prueba, Crear un conjunto de prueba

barajar datos, descenso de gradiente estocástico, barajar los datos

Mezclando los datos

Red neuronal siamesa, ejecución paralela en múltiples dispositivos

función de activación sigmoidea, estimación de probabilidades, multicapa

Perceptrón y retropropagación, la desaparición/explosión

Problemas de gradientes, codificadores automáticos dispersos

señales, neuronas biológicas

coeficiente de silueta, encontrar el número óptimo de grupos

diagrama de silueta, Encontrar el número óptimo de grupos

Función de activación SiLU, GELU, Swish y Mish

características de similitud, SVM, características de similitud

SimpleImputer, limpia los datos

recocido simulado, descenso de gradiente estocástico

entornos simulados, Introducción a OpenAI Gym-Introducción a Gimnasio OpenAI

programa único, datos múltiples (SPMD), paralelismo de datos- saturación de ancho de banda

aprendizaje de un solo disparo, segmentación semántica

descomposición de valores singulares (SVD), ecuación normal, principio Componentes, PCA aleatorio

conjuntos de datos sesgados, medición de la precisión mediante validación cruzada

sesgado hacia la izquierda/derecha, eche un vistazo rápido a la estructura de datos

omitar conexiones, ELU y SELU, ResNet

variable de holgura, **bajo el capó de los clasificadores SVM lineales**  
términos de suavizado, **normalización por lotes**, **AdaGrad**  
**SMOTE** (técnica de sobremuestreo de minorías sintéticas), **AlexNet**  
actor-crítico suave (SAC), **descripción general de algunos algoritmos RL populares**  
agrupamiento suave, **k-medias**  
clasificación de margen suave, **Clasificación de margen suave- Margen suave**  
**Clasificación**, **bajo el capó de los clasificadores SVM lineales**  
Voto suave, **Clasificadores de votación.**  
función de activación softmax, **regresión Softmax**, **clasificación**  
**MLP**, creación del modelo utilizando la API secuencial, un codificador–  
Red decodificadora para traducción automática neuronal  
regresión softmax, **regresión softmax- regresión softmax**  
función de activación softplus, **MLP de regresión**  
filtros de spam, **El panorama del aprendizaje automático: ¿por qué utilizar el aprendizaje**  
**automático?**, **Tipos de sistemas de aprendizaje automático : aprendizaje supervisado**  
codificadores automáticos dispersos, **codificadores automáticos dispersos- codificadores automáticos dispersos**  
características dispersas, clase SVC, **clases SVM y computacional**  
**Complejidad**  
matriz dispersa, **Manejo de texto y atributos categóricos**, **Característica**  
**Escalamiento y Transformación**, **Canales de Transformación**  
modelos dispersos, **regresión de lazo**, **AdamW**  
tensores dispersos, **Tensores dispersos**  
pérdida de escasez, **codificadores automáticos dispersos**  
especificidad, curva ROC, **La curva ROC**

agrupamiento espectral, otros algoritmos de agrupamiento

reconocimiento de voz, ¿Por qué utilizar el aprendizaje automático?, Ejemplos de aplicaciones

nodo dividido, árbol de decisión, hacer predicciones

SPMD (programa único, datos múltiples), paralelismo de datos- saturación de ancho de banda

pérdida de bisagra cuadrada, bajo el capó de los clasificadores lineales SVM

Difusión estable, modelos de difusión

codificadores automáticos apilados, codificadores automáticos apilados-Training One

Codificador automático a la vez

apilamiento (generalización apilada), apilamiento-apilamiento

gradiéntes obsoletos, actualizaciones asincrónicas

coeficiente de correlación estándar, buscar correlaciones

desviación estándar, eche un vistazo rápido a la estructura de datos

estandarización, escalamiento de características y transformación

StandardScaler, escalado y transformación de funciones, degradado

Descenso, regresión de crestas, clasificación SVM no lineal

valores de acción de estado (valores Q), procesos de decisión de Markov-Markov

Procesos de decisión

métrica con estado, métricas personalizadas

RNN con estado, procesamiento del lenguaje natural con RNN y

Atención, RNN con estado -RNN con estado

RNN sin estado, procesamiento del lenguaje natural con RNN y

Atención, RNN con estado, RNN con estado

series de tiempo estacionarias, [Predicción de una serie de tiempo](#)

modo estadístico, [embolsado y pegado](#)

significación estadística, [hiperparámetros de regularización](#)

Biblioteca statsmodels, [la familia de modelos ARMA](#)

derivación, [ejercicios](#)

funciones escalonadas, TLU, [El Perceptrón](#)

aumento de gradiente estocástico, [aumento de gradiente](#)

descenso de gradiente estocástico (SGD), [descenso de gradiente estocástico-](#)

[Descenso de gradiente estocástico, clases SVM y computacional](#)

Complejidad

- parada anticipada, [parada anticipada](#)
- clasificación de imágenes, [Compilación del modelo](#)
- y algoritmo de aprendizaje del perceptrón, [The Perceptron](#)
- regularización de crestas, [regresión de crestas](#)
- y aprendizaje TD, [aprendizaje por diferencia temporal](#)
- entrenar clasificador binario, [entrenar un clasificador binario](#)

muestreo estratificado, [Crear un conjunto de pruebas-Crear un conjunto de pruebas, Medición](#)

[Precisión mediante validación cruzada](#)

`strat_train_set`, [preparar los datos para algoritmos de aprendizaje automático](#)

métrica de transmisión, [métricas personalizadas](#)

avances, [Capas convolucionales, Segmentación semántica, Uso de capas convolucionales](#)

1D para procesar secuencias

núcleos de cadena, núcleo RBF gaussiano

tensores de cuerdas, [Otras estructuras de datos](#)

Capa StringLookup, [La capa StringLookup](#)

cuerdas, cuerdas-cuerdas

estudiantes fuertes, [clasificadores de votación](#)

mezcla de estilos, StyleGAN, [StyleGANs](#)

transferencia de estilo, GAN, [StyleGAN](#)

StyleGAN, StyleGAN-StyleGAN

API de subclasiﬁcación, [Uso de la API de subclasiﬁcación para crear dinámica](#)

Modelos: uso [de la API de subclases para crear modelos dinámicos](#)

vector subgradiente, [regresión de lazo](#)

submuestreo, capa de agrupación, [capas de agrupación](#)

regularización de subpalabras, [análisis de sentimiento](#)

superconvergencia, [programación de tasas de aprendizaje](#)

superresolución, [segmentación semántica](#)

aprendizaje supervisado, [aprendizaje supervisado](#)

máquinas de vectores de soporte (SVM), [máquinas de vectores de soporte-SVM kernelizadas](#)

- función de decisión, [bajo el capó de los clasificadores SVM lineales](#)  
[Bajo el capó de los clasificadores SVM lineales](#)

- problema dual, [Las SVM kernelizadas de problema dual](#)
- SVM kernelizadas, [SVM kernelizadas -SVM kernelizadas](#)
- clasificación lineal, [Clasificación SVM lineal- Margen suave](#)  
[Clasificación](#)

- Mecánica de, **bajo el capó** de los clasificadores SVM lineales.  
SVM kernelizadas
  - en clasificación multiclas, **Clasificación multiclas**
  - clasificadores no lineales, **Clasificación SVM no lineal**
  - SVM de una clase, **otros algoritmos para anomalías y novedades Detección**
  - Regresión SVM, **Regresión SVM- Regresión SVM**

vectores de soporte, **clasificación SVM lineal**

Clase SVC, **Kernel polinomial**, Clases SVM y Computacional  
**Complejidad**

SVD (descomposición en valores singulares), **ecuación normal**, principal  
**Componentes**, PCA aleatorio

SVM (ver máquinas de vectores de soporte)

Clase SVR, **regresión SVM**

Función de activación Swish, **GELU**, Swish y Mish

Conjunto de datos de rollo suizo, aprendizaje **múltiple - aprendizaje múltiple**

diferenciación simbólica, **Autodiff modo directo**

tensores simbólicos, **AutoGraph y Tracing**, Funciones TF y  
**Funciones concretas**

actualizaciones sincrónicas, con parámetros centralizados, **actualizaciones sincrónicas**

técnica de sobremuestreo de minorías sintéticas (SMOTE), **AlexNet**

t

Incrustación de vecinos estocásticos distribuidos en t (t-SNE), otros

Técnicas de reducción de dimensionalidad, visualizando la moda.

Conjunto de datos MNIST

atributos de destino, eche un vistazo rápido a la estructura de datos

distribución objetivo, transformación, escalado de funciones y transformación

modelo objetivo, DQN, objetivos de valor Q fijo

Error TD, aprendizaje por diferencia temporal

Objetivo TD, aprendizaje de diferencia temporal

TD-Gammon, Aprendizaje por refuerzo

Forzamiento docente, una red codificadora-decodificadora para máquinas neuronales

Traducción

temperatura, modelo Char-RNN, generación de Shakespeare falso

Texto

aprendizaje de diferencia temporal (TD), aprendizaje de diferencia temporal,

Repetición de experiencia priorizada

matrices tensoriales, matrices tensoriales

unidades de procesamiento de tensores (TPU), un recorrido rápido por TensorFlow,

Implementar una nueva versión del modelo, Implementar un modelo en un dispositivo móvil o

Dispositivo integrado, entrenamiento de un modelo en un clúster de TensorFlow

TensorBoard, uso de TensorBoard para visualización

TensorBoard para visualización, un recorrido rápido por TensorFlow, enmascaramiento,

Ejecutar grandes trabajos de capacitación en Vertex AI

TensorFlow, Objetivo y Enfoque, Objetivo y Enfoque,

Modelos personalizados y entrenamiento con TensorFlow-The TensorFlow

Proyecto de conjuntos de datos, capacitación e implementación de modelos TensorFlow en

Ajuste de hiperparámetros de escala en Vertex AI

- (ver también API de Keras)
  - arquitectura, un recorrido rápido por TensorFlow
  - creando una función de entrenamiento, [usando el conjunto de datos con Keras](#)
  - modelos personalizados (ver modelos personalizados y algoritmos de entrenamiento)
  - implementar un modelo en un dispositivo móvil, [implementar un modelo en un Dispositivo móvil o integrado : implementación de un modelo en un dispositivo móvil o Dispositivo integrado](#)
  - funciones y gráficos, [Gráficos de TensorFlow : uso de funciones TF con Keras \(o no\)](#)
- 
- Gestión de GPU con, [Gestión de la RAM de la GPU: Gestión de la RAM de GPU, ejecución en paralelo en varios dispositivos: en paralelo Ejecución en múltiples dispositivos](#)
  - gráficos y funciones, [Un recorrido rápido por TensorFlow, TensorFlow Funciones y gráficos: reglas de funciones TF, gráficos de TensorFlow Usar funciones TF con Keras \(o no\)](#)
  - hub.KerasLayer, [uso de componentes del modelo de lenguaje previamente entrenado](#)
  - operaciones matemáticas, [Tensores y Operaciones](#)
  - con NumPy, [usando TensorFlow como NumPy-Otros datos Estructuras](#)
  - operaciones (ops) y tensores, [un recorrido rápido por TensorFlow, Tensores y operaciones: tensores y NumPy](#)
  - paralelismo para entrenar modelos (ver paralelismo)
  - plataformas y API disponibles, [un recorrido rápido por TensorFlow](#)
  - servir un modelo (ver Servicio de TensorFlow)

- estructuras de datos especiales, [Estructuras de datos especiales-colas](#)
- tf.add(), [Tensores y Operaciones](#)
- tf.autograph.to\_code(), [AutoGraph y Seguimiento](#)
- tf.cast(), [Conversiones de tipos](#)
- tf.config.set\_soft\_device\_placement, [Colocación de operaciones y variables en dispositivos](#)
- tf.config.threading.set\_inter\_op\_parallelism\_threads(), [ejecución paralela en varios dispositivos](#)
- tf.config.threading.set\_intra\_op\_parallelism\_threads(), [ejecución paralela en varios dispositivos](#)
- tf.constant(), [Tensores y Operaciones](#)
- API tf.data (ver API tf.data)
- tf.device(), [colocación de operaciones y variables en dispositivos](#)
- tf.distribute.experimental.CentralStorageStrategy, [capacitación a escala utilizando la API de estrategias de distribución](#)
- tf.distribute.experimental.TPUStrategy, [entrenamiento de un modelo en un clúster de TensorFlow](#)
- tf.distribute.MirroredStrategy, [capacitación a escala utilizando la API de estrategias de distribución, ajuste de hiperparámetros en Vertex AI](#)
- tf.distribute.MultiWorkerMirroredStrategy, [entrenamiento de un modelo en un clúster de TensorFlow](#)
- tf.float32, [tensores y NumPy](#)
- Tipo de datos tf.float32, [métricas personalizadas, preprocesamiento de los datos](#)
- tf.function(), [Funciones y gráficos de TensorFlow, Reglas de funciones TF](#)

- tipo de datos `tf.int32`, [otras estructuras de datos](#)
- `tf.io.decode_base64()`, [Ejecución de trabajos de predicción por lotes en Vertex AI](#)
- `tf.io.decode_csv()`, [Preprocesamiento de los datos](#)
- `tf.io.decode_image()`, [Ejecución de trabajos de predicción por lotes en Vertex AI](#)
- `tf.io.decode_png()`, [Ejecución de trabajos de predicción por lotes en Vertex AI](#)
- `tf.io.decode_proto()`, [una breve introducción a los buffers de protocolo](#)
- `tf.io.FixedLenFeature`, [ejemplos de carga y análisis](#)
- `tf.io.parse_example()`, [Ejemplos de carga y análisis](#)
- `tf.io.parse_sequence_example()`, [Manejo de listas de listas usando el Protobuf SequenceExample](#)
- `tf.io.parse_single_example()`, [Ejemplos de carga y análisis](#)
- `tf.io.parse_single_sequence_example()`, [Manejo de listas de listas usando el Protobuf SequenceExample](#)
- `tf.io.parse_tensor()`, [ejemplos de carga y análisis](#)
- `tf.io.serialize_tensor()`, [ejemplos de carga y análisis](#)
- `tf.io.TFRecordOptions`, [archivos TFRecord comprimidos](#)
- `tf.io.TFRecordWriter`, [el formato TFRecord](#)
- `tf.io.VarLenFeature`, [Ejemplos de características, carga y análisis](#)
- `tf.linalg.band_part()`, [atención multicabezal](#)
- `tf.lite.TFLiteConverter.from_keras_model()`, [Implementación de un modelo en un dispositivo móvil o integrado](#)

- `tf.make_tensor_proto()`, [consulta de TF Serving a través de la API gRPC](#)
- `tf.matmul()`, [Tensores y Operaciones](#), [Atención multicabezal](#)
- `tf.nn.conv2d()`, [Implementación de capas convolucionales con Keras](#)
- `tf.nn.embedding_lookup()`, [Capas de preprocesamiento de imágenes](#)
- `tf.nn.local_response_normalization()`, [AlexNet](#)
- `tf.nn.moments()`, [Capas de preprocesamiento de imágenes](#)
- `tf.nn.sampled_softmax_loss()`, [una red de codificador-decodificador para traducción automática neuronal](#)
- `tf.py_function()`, [Reglas de función TF](#), [Exportación de modelos guardados](#)
- Módulo `tf.queue`, [Otras estructuras de datos](#)
- `tf.queue.FIFOQueue`, [Colas](#)
- `tf.RaggedTensor`, [otras estructuras de datos](#)
- `tf.random.categorical()`, [generando texto shakesperiano falso](#)
- `tf.reduce_max()`, [Implementación de capas de agrupación con Keras](#)
- `tf.reduce_mean()`, [bucles de entrenamiento personalizados](#)
- `tf.reduce_sum()`, [funciones y gráficos de TensorFlow](#)
- Comando `tf.saved_model_cli`, [Exportación de modelos guardados](#)
- Módulo `tf.sets`, [Otras estructuras de datos](#)
- `tf.sort()`, [Reglas de función TF](#)
- `tf.SparseTensor`, [otras estructuras de datos](#)
- `tf.stack()`, [preprocesamiento de datos](#), [RNN con estado](#)
- tipo de datos `tf.string`, [otras estructuras de datos](#)

- Módulo `tf.strings`, [Otras estructuras de datos](#)
- `tf.Tensor`, [Tensores y Operaciones](#), [Variables](#)
- `tf.TensorArray`, [otras estructuras de datos](#)
- `tf.transpose()`, [Tensores y Operaciones](#)
- `tf.Variable`, [Variables](#)
- `tf.Variable.assign()`, [Variables](#)
- conversiones de tipo, [Conversiones de tipo](#)
- variables, [variables](#)
- página web, ejecutar un modelo en, [ejecutar un modelo en una página web](#)

Clúster de TensorFlow, [entrenamiento de un modelo en un clúster de TensorFlow](#)

[Entrenamiento de un modelo en un clúster de TensorFlow](#)

Proyecto TensorFlow Datasets (TFDS), [los conjuntos de datos TensorFlow](#)

[Proyecto: el proyecto de conjuntos de datos de TensorFlow](#)

TensorFlow Extended (TFX), [un recorrido rápido por TensorFlow](#)

TensorFlow Hub, [un recorrido rápido por TensorFlow](#), utilizando la capacitación previa

[Componentes del modelo de lenguaje](#), reutilización de incrustaciones previamente entrenadas y

[Modelos de lenguaje](#)

TensorFlow Lite, [un recorrido rápido por TensorFlow](#)

Zona de juegos TensorFlow, [clasificación MLP](#)

Servicio TensorFlow (servicio TF), [servicio de un modelo TensorFlow](#)

[Ejecución de trabajos de predicción por lotes en Vertex AI](#)

- trabajos de predicción por lotes en Vertex AI, [ejecución de predicción por lotes](#)
- [Empleos en Vertex AI](#)

- creando un servicio de predicción, [Creando un servicio de predicción en Vertex AI: creación de un servicio de predicción en Vertex AI](#)
- Implementando una nueva versión del modelo, [Implementando una nueva versión del modelo. Implementación de una nueva versión del modelo](#)
- Contenedor Docker, [Instalación e inicio de TensorFlow Serving](#)
- exportando modelos guardados, [exportando modelos guardados-exportando Modelos guardados](#)
- API de gRPC, consulta a través de, [Consulta de TF Sirviendo a través de la API de gRPC](#)
- instalar e iniciar, [Instalar e iniciar TensorFlow Servicio: instalación e inicio de TensorFlow Serving](#)
- API REST, consulta a través de, [Consulta TF Sirviendo a través de API DESCANSO](#)

Texto TensorFlow, [preprocesamiento de texto, análisis de sentimiento](#)

Biblioteca JavaScript TensorFlow.js (TFJS), [ejecución de un modelo en una página web](#)

tensores, [Tensores y Operaciones-Tensores y NumPy](#)

frecuencia-término x frecuencia-documento-inversa (TF-IDF), [Texto Preprocesamiento](#)

estado terminal, cadena de Markov, [procesos de decisión de Markov](#)

conjunto de pruebas, [Pruebas y validación, Crear un conjunto de pruebas-Crear un conjunto de pruebas](#)

atributos de texto, [manejo de texto y atributos categóricos](#)

procesamiento de texto (ver procesamiento del lenguaje natural)

Servicio TF (consulte Servicio TensorFlow)

tf.data API, [La API tf.data : uso del conjunto de datos con Keras](#)

- Encadenamiento de transformaciones, [Encadenamiento de transformaciones-Encadenamiento Transformaciones](#)
- entrelazar líneas de múltiples archivos, [Intercalar líneas de Múltiples archivos: líneas entrelazadas de múltiples archivos](#)
- y capas de preprocesamiento de Keras, [la capa de normalización](#)
- captación previa, captación previa-captación previa
- preprocesar los datos, [Preprocesar los datos-Preprocesar los datos](#)
- barajar datos, [barajar datos-barajar datos](#)
- tf.data.AUTOTUNE, [Encadenamiento de transformaciones](#)
- tf.data.Dataset.from\_tensor\_slices(), [Las transformaciones de encadenamiento de API de tf.data](#)
- tf.data.TFRecordDataset, [el formato TFRecord, el formato TFRecord, ejemplos de carga y análisis](#)
- usando el conjunto de datos con Keras, [Usando el conjunto de datos con Keras-Usando el conjunto de datos con Keras](#)

Proyecto TFDS (TensorFlow Datasets), [Los conjuntos de datos de TensorFlow Proyecto: el proyecto de conjuntos de datos de TensorFlow](#)

Biblioteca JavaScript TFJS (TensorFlow.js), [ejecución de un modelo en una página web](#)

TFLite, [implementación de un modelo en un dispositivo móvil o integrado](#)  
[Implementación de un modelo en un dispositivo móvil o integrado](#)

Formato TFRecord, [carga y preprocesamiento de datos con TensorFlow, El formato TFRecord: manejo de listas de listas utilizando el SecuenciaEjemplo Protobuf](#)

TFX (TensorFlow Extended), [un recorrido rápido por TensorFlow](#)

Criterio de información teórica, Selección del número de conglomerados.

Capas convolucionales 3D, Segmentación Semántica

unidades lógicas de umbral (TLU), el perceptrón, la multicapa

Perceptrón y retropropagación

Regularización de Tikhonov, Regresión de crestas- Regresión de crestas,

Regresión neta elástica

datos de series de tiempo, pronóstico, procesamiento de secuencias usando RNN y

CNN, pronóstico de una serie de tiempo: pronóstico usando un

Modelo secuencia a secuencia

- Familia de modelos ARMA, La familia de modelos ARMA-EI ARMA
  - Familia de modelos
- preparación de datos para modelos ML, preparación de datos para máquina
  - Modelos de aprendizaje : preparación de datos para el aprendizaje automático
  - Modelos
- con RNN profundo, pronóstico utilizando un RNN profundo
- con modelo lineal, Pronósticos usando un modelo lineal
- series de tiempo multivariadas, Pronóstico de una serie de tiempo, Pronóstico Series de tiempo multivariadas : pronóstico de series de tiempo multivariadas
- con modelo secuencia a secuencia, Entrada y Salida
  - Secuencias, pronóstico utilizando una secuencia a secuencia
  - Pronóstico de modelos utilizando un modelo de secuencia a secuencia
- varios pasos de tiempo por delante, pronosticando varios pasos de tiempo
  - Previsión anticipada de varios pasos de tiempo por delante
- con RNN simple, pronóstico usando un pronóstico RNN simple
  - Usando un RNN simple

TLU (unidades lógicas de umbral), El Perceptrón, La Multicapa

Perceptrón y retropropagación

TNR (tasa negativa verdadera), la curva ROC

biblioteca de tokenizadores, análisis de sentimiento

tolerancia ( $\epsilon$ ), descenso de gradiente por lotes, clases SVM y

Complejidad computacional

TPR (tasa de verdaderos positivos), matrices de confusión, curva ROC

TPU (unidades de procesamiento de tensores), Un recorrido rápido por TensorFlow,

Implementar una nueva versión del modelo, Implementar un modelo en un dispositivo móvil o

Dispositivo integrado, entrenamiento de un modelo en un clúster de TensorFlow

conjunto de desarrollo de tren, discrepancia de datos

capacitación, aprendizaje basado en modelos y un flujo de trabajo típico de aprendizaje automático

instancia de capacitación, ¿Qué es el aprendizaje automático?

bucles de entrenamiento, bucles de entrenamiento personalizados -bucles de entrenamiento personalizados,

Usando el conjunto de datos con Keras

modelos de entrenamiento, Modelos de entrenamiento -Regresión Softmax

- curvas de aprendizaje en, Curvas de Aprendizaje -Curvas de Aprendizaje
- regresión lineal, Modelos de entrenamiento, Regresión lineal-Mini-Descenso de gradiente por lotes
- regresión logística, Regresión logística- Regresión Softmax
- perceptrones, El Perceptrón-El Perceptrón
- regresión polinómica, modelos de entrenamiento, regresión polinómica-Regresión polinomial

conjunto de capacitación, ¿Qué es el aprendizaje automático?, Pruebas y validación

- función de costo de, **Capacitación y función de costo-Capacitación y costo Función**
- cantidades insuficientes, **cantidad insuficiente de datos de entrenamiento**
- características irrelevantes, **características irrelevantes**
- Escalado mínimo-máximo, **escalado de funciones y transformación.**
- **Datos de capacitación no representativos, no representativos**
- sobreajuste, **sobreajuste de los datos de entrenamiento-sobreajuste del entrenamiento Datos**
- preparación para algoritmos de ML, **preparación de datos para la máquina Algoritmos de aprendizaje**
- capacitar y evaluar, **Capacitar y Evaluar en la Capacitación Establecer, entrenar y evaluar en el conjunto de entrenamiento**
- transformación de datos, **escalamiento de funciones y transformación**
- desadaptación, **Desadaptación de los datos de entrenamiento**
- visualizar datos, **explorar y visualizar los datos para obtener información**

expansión del conjunto de entrenamiento, **ejercicios, AlexNet, modelos preentrenados para Transferir aprendizaje**

**sesgo de entrenamiento/servicio, carga y preprocessamiento de datos con TensorFlow**

**train\_test\_split(), creación de un conjunto de pruebas, mejor evaluación mediante el uso cruzado Validación**

transferencia de aprendizaje, aprendizaje autosupervisado, reutilización de formación previa Capas, **Transferir Aprendizaje con Keras-Transferir Aprendizaje con Keras,**  
**Modelos previamente entrenados para transferir aprendizaje: modelos previamente entrenados para Transferir aprendizaje**

transform(), limpieza de datos, manejo de texto y categórico

Atributos, escalado y transformación de funciones, personalizado  
transformadores

transformación de datos

- transformadores personalizados, [Transformadores personalizados-Personalizados transformadores](#)
- transformadores estimadores, [Limpieza los datos](#)
- y escalado de funciones, [escalado y transformación de funciones.](#)  
[Escalado y transformación de funciones](#)
- modelos de transformadores (ver modelos de transformadores)

pipelines de transformación, [Pipelines de Transformación -Transformación](#)

Tuberías

TransformedTargetRegressor, [escalado y transformación de funciones](#)

transformador, [atención es todo lo que necesita: el transformador original](#)  
[Arquitectura](#)

modelos de transformadores

- mecanismos de atención, [La atención es todo lo que necesitas: el original Arquitectura transformadora: atención de múltiples cabezas , visión transformadores](#)
- BERT, [una avalancha de modelos de transformadores](#)
- DistilBERT, [una avalancha de modelos de transformadores, abrazos](#)  
[Biblioteca de Face's Transformers : abrazando a Face's Transformers](#)  
[Biblioteca](#)
- Biblioteca de Hugging Face, [Biblioteca de Transformers de Hugging Face-Biblioteca de Transformers de Hugging Face](#)

- Modelo de lenguaje Pathways, una avalancha de transformadores Modelos
  - transformadores de visión, **Transformadores de visión- Transformadores** de visión

TransformerMixin, **Transformadores personalizados**

biblioteca de transformadores, **Abrazando** la biblioteca de transformadores de Face-Hugging  
**Biblioteca de Transformers de Face**

traducción, con RNN, **procesamiento del lenguaje natural con RNN y atención, una red codificadora-decodificadora para máquinas neuronales**

Búsqueda por haz de traducción

- (ver también modelos de transformadores)

operador de transposición, **seleccione una medida de rendimiento**

capa convolucional transpuesta, **segmentación semántica-semántica**

**Segmentación**

tasa negativa verdadera (TNR), **la curva ROC**

verdaderos negativos, matriz de confusión, **matrices de confusión**

tasa de verdaderos positivos (TPR), **matrices de confusión, curva ROC**

verdaderos positivos, matriz de confusión, **Matrices de confusión**

optimización de políticas de región de confianza (TRPO), **descripción general de algunos populares Algoritmos RL**

Capas convolucionales 2D, **Implementación de capas convolucionales con Keras**

atar pesas, **atar pesas**

errores tipo I, matriz de confusión, **Matrices de confusión**

errores tipo II, matriz de confusión, **Matrices de confusión**

Ud.

muestreo de incertidumbre, uso de agrupamiento para semisupervisado

Aprendiendo

subcompleto, codificador automático como, representaciones de datos eficientes.

Realización de PCA con un codificador automático lineal incompleto

desadaptación de los datos, desadaptación de los datos de entrenamiento, capacitación y

Evaluar en el conjunto de entrenamiento, Curvas de aprendizaje -Curvas de aprendizaje,

Núcleo RBF gaussiano

regresión univariada, encuadre el problema

series temporales univariadas, Predicción de una serie temporal

Codificador de oraciones universal, reutilización de incrustaciones previamente entrenadas y

Modelos de lenguaje : reutilización de incrustaciones y lenguaje previamente entrenados

Modelos

efectividad irrazonable de los datos, cantidad insuficiente de capacitación

Datos

desenrollando la red a través del tiempo, Neuronas y Capas Recurrentes

problema de gradientes inestables, Los gradientes que desaparecen/explotan

Problemas, luchando contra el problema de los gradientes inestables

- (ver también gradientes que desaparecen y explotan)

aprendizaje no supervisado, Aprendizaje no supervisado -Aprendizaje no supervisado,

Técnicas de aprendizaje no supervisado-Otros algoritmos para

Detección de anomalías y novedades

- detección de anomalías, aprendizaje no supervisado
- aprendizaje de reglas de asociación, aprendizaje no supervisado
- codificadores automáticos (ver codificadores automáticos)

- agrupación (ver algoritmos de agrupación)
- estimación de densidad, **Técnicas de aprendizaje no supervisado**
- modelos de difusión, Modelos **de Difusión -Modelos de Difusión**
- reducción de dimensionalidad (ver reducción de dimensionalidad)
- GAN (ver redes generativas adversarias)
- GMM, **mezclas gaussianas: otros algoritmos para anomalías y Detección de novedades**
- k-medias (ver algoritmo k-medias)
- detección de novedades, **aprendizaje no supervisado**
- preentrenamiento, **preentrenamiento no supervisado, reutilización de preentrenado Incorporaciones y modelos de lenguaje, una avalancha de Modelos de transformadores, preentrenamiento no supervisado utilizando apilados codificadores automáticos**
- codificadores automáticos apilados, **preentrenamiento no supervisado usando apilados codificadores automáticos**
- Modelos de transformadores, **Una avalancha de modelos de transformadores.**
- algoritmos de visualización, **Aprendizaje no supervisado -Aprendizaje no supervisado**

capa de muestreo ascendente, **segmentación semántica**

función de utilidad, **aprendizaje basado en modelos y un flujo de trabajo típico de aprendizaje automático**

V

VAE (codificadores automáticos variacionales), **codificadores automáticos variacionales-Autocodificadores variacionales**

relleno "válido", visión por computadora, [implementación de capas convolucionales con Keras](#)

conjunto de validación, [ajuste de hiperparámetros y selección de modelo, entrenamiento y evaluación del modelo](#)-entrenamiento y evaluación del modelo

`value_counts()`, [echa un vistazo rápido a la estructura de datos](#)

gradientes que desaparecen y explotan, [The Vanishing/Exploding](#)  
Problemas [de gradientes : recorte de gradientes](#)

- mejoras en la función de activación, [mejores funciones de activación-GELU, Swish y Mish](#)
- normalización por lotes, Normalización [por lotes : implementación de la normalización por lotes con Keras](#)
- Inicialización de Glorot y He, [Inicialización de Glorot y He- Inicialización de Glorot y He](#)
- recorte de degradado, [recorte de degradado](#)
- problema de gradientes inestables, [Luchando contra los gradientes inestables](#)  
[Cómo combatir el problema de los gradientes inestables](#)

variables

- manejo en funciones TF, [Manejo de Variables y Otros Recursos en funciones TF: manejo de variables y otros Recursos en funciones TF](#)
- persistencia de [métricas personalizadas](#)
- colocación en GPU, [colocación de operaciones y variables en dispositivos](#)
- en TensorFlow, [Variables](#)

diferencia

- Compensación entre sesgo y varianza, [Curvas de aprendizaje](#)

- explicado, Relación de varianza explicada : elegir el correcto Número de dimensiones
- alta varianza con los árboles de decisión, los árboles de decisión tienen una alta Diferencia
- preservando, preservando la variación

codificadores automáticos variacionales (VAE), codificadores automáticos variacionales- Autocodificadores variacionales

red de vector a secuencia, secuencias de entrada y salida

vectores, normas para medir distancias, Seleccione una actuación Medida

Vertex AI, inicie, monitoree y mantenga su sistema, creando una Servicio de predicción en Vertex AI: creación de un servicio de predicción en Vertex AI, ejecutando grandes trabajos de capacitación en Vertex AI-Running Large Trabajos de formación en Vertex AI

VGGNet, VGGNet

dispositivo GPU virtual, Gestión de la RAM de la GPU

Transformadores de visión (ViTs), Transformadores de visión- Transformadores de visión arquitectura de la corteza visual, La arquitectura de la corteza visual

visualización de datos, Ejemplos de aplicaciones, Aprendizaje no supervisado -Aprendizaje no supervisado, Explorar y visualizar los datos para Obtenga conocimientos y experimente con combinaciones de atributos

- árboles de decisión, Entrenamiento y visualización de un árbol de decisión- Entrenamiento y visualización de un árbol de decisión
- reducción de dimensionalidad, Reducción de dimensionalidad, Elección del número correcto de dimensiones

- Ejercicio de principio a fin, [explorar y visualizar los datos para obtener Insights](#)-Experimento con combinaciones de atributos
- MLP con TensorBoard, [uso de TensorBoard para visualización](#)  
[Usando TensorBoard para visualización](#)
- codificadores automáticos apilados, [visualizando las reconstrucciones](#)-  
[Visualizando las reconstrucciones](#)
- t-SNE, [otras técnicas de reducción de dimensionalidad](#)

ViTs (transformadores de visión), [Vision Transformers](#)-[Vision Transformers](#)

clasificadores de votación, [Clasificadores de votación](#)- Clasificadores de votación

W.

tiempo de pared, [normalización por lotes](#)

fase de preparación, actualizaciones de modelo asincrónicas, [actualizaciones asincrónicas](#)

WaveNet, procesamiento de secuencias utilizando RNN y CNN, [WaveNet](#)-  
[OndaNet](#)

estudiantes débiles, [clasificadores de votación](#)

página web, ejecutar un modelo en, [ejecutar un modelo en una página web](#)

Decaimiento de peso, [AdamW](#)

ocultación de peso, [saturación de ancho de banda](#)

atar pesas, [Atar pesas](#)

pesas

- impulsando, AdaBoost-AdaBoost
- capas convolucionales, [Implementación de capas convolucionales con Keras](#)

- congelar capas reutilizadas, [reutilizar capas previamente entrenadas](#)
- de capas ocultas, [Creando el modelo usando la API secuencial](#)
- en repetición de experiencia priorizada, [repetición de experiencia priorizada](#)
- guardar en lugar del modelo completo, [Guardar y restaurar un modelo](#)

modelos de caja blanca, [Haciendo predicciones](#)

Red neuronal amplia y profunda, [construcción de modelos complejos utilizando API funcional](#) : creación de modelos complejos utilizando la API funcional

longitud de la ventana, [Creación del conjunto de datos de entrenamiento](#)

sabiduría de la multitud, [Ensemble Learning y Random Forests](#)

incrustaciones de palabras, [codificación de características categóricas mediante Incrustaciones](#)

- traducción automática neuronal, una red codificadora-decodificadora para [Traducción automática neuronal: búsqueda por haz](#)
- análisis de sentimiento, [Análisis de sentimiento](#) : reutilización de material preentrenado [Incorporaciones y modelos de lenguaje](#)

tipo de tarea de trabajador, [Entrenamiento de un modelo en un clúster de TensorFlow](#)

trabajadores, [Paralelismo de datos con parámetros centralizados](#)

X

Inicialización de Xavier, [Glorot y He Inicialización](#)

Xception (Extreme Inception), [Xception-Xception](#), modelos preentrenados para el aprendizaje por transferencia-Modelos preentrenados para el aprendizaje por transferencia

XLA (álgebra lineal acelerada), [funciones y gráficos de TensorFlow](#)

Problema XOR (o exclusivo), [El Perceptrón](#)

Y

Solo miras una vez (YOLO), Solo miras una vez- Solo miras  
Una vez

Z

relleno cero, Capas convolucionales, Implementación de convolucional  
Capas con Keras

aprendizaje de disparo cero (ZSL), una avalancha de modelos de transformadores,  
Transformadores de visión

ZFNet, AlexNet

## Sobre el Autor

Aurélien Géron es consultor y conferenciante sobre aprendizaje automático. Ex empleado de Google, dirigió el equipo de clasificación de vídeos de YouTube de 2013 a 2016. Ha sido fundador y director de tecnología de varias empresas diferentes: Wifirst, un ISP inalámbrico líder en Francia; Polyconseil, una consultora centrada en telecomunicaciones, medios y estrategia; y Kiwisoft, una consultora centrada en aprendizaje automático y privacidad de datos.

Antes de todo eso, Aurélien trabajó como ingeniero en diversos ámbitos: finanzas (JP Morgan y Société Générale), defensa (DOD de Canadá) y atención médica (transfusión de sangre). También publicó algunos libros técnicos (sobre C++, WiFi y arquitecturas de Internet) y dio conferencias sobre informática en una escuela de ingeniería francesa.

Algunas curiosidades: enseñó a sus tres hijos a contar en binario con los dedos (hasta 1.023), estudió microbiología y genética evolutiva antes de dedicarse a la ingeniería de software y su paracaídas no se abrió en el segundo salto.

## Colofón El

animal de la portada de Hands-On Machine Learning con Scikit-Learn, Keras y TensorFlow es la salamandra (*Salamandra salamandra*), un anfibio que se encuentra en la mayor parte de Europa. Su piel negra y brillante presenta grandes manchas amarillas en la cabeza y la espalda, lo que indica la presencia de toxinas alcaloides. Ésta es una posible fuente del nombre común de este anfibio: el contacto con estas toxinas (que también pueden rociar distancias cortas) provoca convulsiones e hiperventilación. Los dolorosos venenos o la humedad de la piel de la salamandra (o ambos) llevaron a la creencia equivocada de que estas criaturas no sólo podrían sobrevivir al fuego sino que también podrían extinguirlo.

Las salamandras vivas viven en bosques sombreados, escondiéndose en grietas húmedas y debajo de troncos cerca de estanques u otros cuerpos de agua dulce que facilitan su reproducción. Aunque pasan la mayor parte de su vida en la tierra, dan a luz a sus crías en el agua. Subsisten principalmente con una dieta de insectos, arañas, babosas y gusanos. Las salamandras de fuego pueden crecer hasta un pie de largo y en cautiverio pueden vivir hasta 50 años.

El número de salamandras rojas se ha reducido debido a la destrucción de su hábitat forestal y su captura para el comercio de mascotas, pero la mayor amenaza que enfrentan es la susceptibilidad de su piel permeable a la humedad a contaminantes y microbios. Desde 2014, se han extinguido en algunas partes de los Países Bajos y Bélgica debido a la introducción de un hongo.

Muchos de los animales que aparecen en las portadas de O'Reilly están en peligro de extinción; Todos ellos son importantes para el mundo. La ilustración de la portada es de Karen Montgomery, basada en un grabado de Wood's Illustrated Natural History. Las fuentes de portada son URW Typewriter y Guardian Sans. La fuente del texto es Adobe Minion Pro; la fuente del título es Adobe Myriad Condensed; y la fuente del código es Ubuntu Mono de Dalton Maag.