

Learn
Generative AI
with **PyTorch**

Mark Liu

Foreword by Sarah Sanders



 MANNING



Aprende IA generativa con PyTorch

Mark Liu

PRÓLOGO DE SARAH SANDERS

Para comentar, ve a [liveBook](#)



Manning

Shelter Island

Para obtener más información sobre este y otros títulos de Manning, visita

www.manning.com

Derechos de autor

Para obtener información en línea y realizar pedidos de estos y otros libros de Manning, visita www.manning.com. La editorial ofrece descuentos en estos libros cuando se piden en grandes cantidades

Para obtener más información, póngase en contacto con

Departamento de Ventas Especiales

Manning Publications Co.

20 Baldwin Road

Apartado postal 761

Shelter Island, NY 11964

Correo electrónico: orders@manning.com

©2025 por Manning Publications Co. Todos los derechos reservados.

Ninguna parte de esta publicación podrá ser reproducida, almacenada en un sistema de recuperación o transmitida en ninguna forma o por medios electrónicos, mecánicos, de fotocopia o de otro tipo, sin el permiso previo por escrito del editor.

Muchas de las designaciones utilizadas por fabricantes y vendedores para distinguir sus productos se reclaman como marcas comerciales. Cuando dichas designaciones aparecen en el libro, y Manning Publications tenía conocimiento de la reclamación de marca comercial, se han impreso con mayúscula inicial o en mayúsculas.

Reconociendo la importancia de preservar lo escrito, la política de Manning es imprimir los libros que publicamos en papel libre de ácido, y nos esforzamos al máximo para lograrlo. Reconociendo también nuestra responsabilidad de conservar los recursos de nuestro planeta, los libros de Manning se impri-

men en papel reciclado al menos en un 15 % y procesado sin cloro elemental.

El autor y la editorial han hecho todo lo posible para garantizar que la información de este libro fuera correcta al momento de la publicación. El autor y la editorial no asumen, ni por la presente, ninguna responsabilidad ante terceros por pérdidas, daños o interrupciones causadas por errores u omisiones, ya sean resultado de negligencia, accidente o cualquier otra causa, o por el uso de la información aquí contenida.



Manning Publications Co.
20 Baldwin Road Technical
Apartado postal 761
Shelter Island, NY 11964

Editor de desarrollo:

Rebecca Johnson

Editores técnicos:

Emmanuel Maggiori y Wee
Hyong Tok

Editor de la reseña:

Dunja Nikitović

Editor de producción:

Andy Marinkovich

Editor de estilo:

Kari Lucke

Corrector:

Jason Everett

Corrector técnico:

Kostas Passadis

Compositor tipográfico:

Tamara Švelić Sabljić

Diseñador de portada:

Marija Tudor

ISBN: 9781633436466

Contenido

Portada

Prólogo

Prefacio

Agradecimientos

Sobre este libro

Sobre el autor

Sobre la ilustración de la portada

Parte 1. Introducción a la IA generativa

1 ¿Qué es la IA generativa y por qué PyTorch?

1.1 Introducción a la IA generativa y PyTorch

1.1.1 ¿Qué es la IA generativa?

1.1.2 El lenguaje de programación Python

1.1.3 Usando PyTorch como nuestro marco de IA

1.2 GAN

1.2.1 Una descripción general de alto nivel de las GAN

1.2.2 Un ejemplo ilustrativo: Generar caras de anime.

1.2.3 ¿Por qué debería importarle las GAN?

1.3 Transformadores

1.3.1 El mecanismo de atención

1.3.2 La arquitectura del transformador.

1.3.3 Transformadores multimodales y LLM preentrenados

1.4 ¿Por qué construir modelos generativos desde cero?

2 Aprendizaje profundo con PyTorch

2.1 Tipos de datos en PyTorch

2.1.1 Creación de tensores de PyTorch

2.1.2 Tensores indexadores y segmentarios de PyTorch

2.1.3 Formas de tensores de PyTorch

2.1.4 Operaciones matemáticas con tensores de PyTorch

2.2 Un proyecto de aprendizaje profundo de extremo a extremo con PyTorch

2.2.1 Aprendizaje profundo en PyTorch: una descripción general de alto nivel

2.2.2 Preprocesamiento de datos

2.3 Clasificación binaria

2.3.1 Creación de lotes

2.3.2 Construcción y entrenamiento de un modelo de clasificación binaria.

2.3.3 Prueba del modelo de clasificación binaria

2.4 Clasificación multicategoría

2.4.1 Conjunto de validación y detención anticipada

2.4.2 Construcción y entrenamiento de un modelo de clasificación multicategoría.

3 Redes generativas antagónicas: generación de formas y números.

3.1 Pasos involucrados en el entrenamiento de GAN

3.2 Preparación de datos de entrenamiento

3.2.1 Un conjunto de datos de entrenamiento que forma una curva de crecimiento exponencial.

3.2.2 Preparación del conjunto de datos de entrenamiento.

3.3 Creación de GAN

3.3.1 La red discriminadora

3.3.2 La red generadora

3.3.3 Funciones de pérdida, optimizadores y parada temprana

3.4 Entrenamiento y uso de GAN para la generación de formas.

3.4.1 El entrenamiento de las GAN

3.4.2 Guardar y usar el generador de entrenamiento

3.5 Generación de números con patrones

3.5.1 ¿Qué son las variables one-hot?

3.5.2 GAN para generar números con patrones

3.5.2 Entrenando las GAN para generar números con patrones

3.5.3 Guardar y usar el modelo entrenador.

Parte 2. Generación de imágenes

4 Generación de imágenes con redes generativas antagónicas

4.1 GAN para generar imágenes en escala de grises de prendas de vestir

4.1.1 Muestras de entrenamiento y el discriminador

4.1.2 Un generador para crear imágenes en escala de grises.

4.1.3 Entrenamiento de GAN para generar imágenes de prendas de vestir

4.2 Capas convolucionales

4.2.1 ¿Cómo funcionan las operaciones convolucionales?

4.2.2 ¿Cómo afecta el paso y el relleno a las operaciones convolucionales?

4.3 Convolución transpuesta y normalización por lotes

4.3.1 ¿Cómo funcionan las capas convolucionales transpuestas?

4.3.2 Normalización por lotes

4.4 Imagenes de colores de caras de anime

4.4.1 Descarga de caras de anime

4.4.2 Imágenes en color con canales primero en PyTorch

4.5 GAN convolucional profunda

4.5.1 Construcción de una DCGAN

4.5.2 Entrenamiento y uso de DCGAN

5 Selección de características en imágenes generadas

5.1 El conjunto de datos de gafas

5.1.1 Descarga del conjunto de datos de gafas

5.1.2 Visualización de imágenes en el conjunto de datos de gafas.

5.2 cGAN y distancia de Wasserstein

5.2.1 WGAN con penalización de gradiente

5.2.2 cGAN

5.3 Crear un cGAN

5.3.1 Un discriminador en cGAN

5.3.2 Un generador en cGAN

5.3.3 Inicialización de peso y función de penalización de gradiente

5.4 Entrenamiento del cGAN

5.4.1 Agregar etiquetas a las entradas

5.4.2 Entrenamiento del cGAN

5.5 Selección de características en imágenes generadas

5.5.1 Seleccionar imágenes con o sin gafas

5.5.2 Aritmética vectorial en el espacio latente

5.5.3 Seleccionar dos características simultáneamente

6 CycleGAN: Convirtiendo el cabello rubio en cabello negro

6.1 CycleGAN y pérdida de consistencia del ciclo

6.1.1 ¿Qué es CycleGAN?

6.1.2 Pérdida de consistencia del ciclo

6.2 El conjunto de datos de rostros de famosos.

6.2.1 Descarga del conjunto de datos de rostros de famosos

6.2.2 Procesar los datos de imágenes de cabello negro y rubio

6.3 Construcción de un modelo CycleGAN

6.3.1 Creación de dos discriminadores

6.3.2 Creación de dos generadores

6.4 Uso de CycleGAN para traducir entre cabello negro y rubio

6.4.1 Entrenando un CycleGAN para traducir entre cabello negro y rubio

6.4.2 Conversiones de ida y vuelta de imágenes de cabello negro e imágenes de cabello rubio

7 Generación de imágenes con autocodificadores variacionales

7.1 Una descripción general de los EA.

7.1.1 ¿Qué es un AE?

7.1.2 Pasos para construir y entrenar un AE

7.2 Construcción y entrenamiento de un AE para generar dígitos

7.2.1 Recopilación de dígitos escritos a mano

7.2.2 Creación y entrenamiento de un EA

7.2.3 Guardado y uso del EA entrenador

7.3 ¿Qué son los VAE?

7.3.1 Diferencias entre EA y VAE

7.3.2 El plan para entrenar un VAE para generar imágenes de rostros humanos

7.4 Un VAE para generar imágenes de rostros humanos.

7.4.1 Construyendo un VAE

7.4.2 Entrenando el VAE

7.4.3 Generando imágenes con el VAE entrenado

7.4.4 Codificación aritmética con el VAE entrenado

8 Generación de texto con redes neuronales recurrentes

8.1 Introducción a las RNN

8.1.1 Desafíos en la generación de texto

8.1.2 ¿Cómo funcionan las RNN?

8.1.3 Pasos para entrenar un modelo LSTM

8.2 Fundamentos de la PNL

8.2.1 Diferentes métodos de tokenización

8.2.2 Incrustaciones de palabras

8.3 Preparación de datos para entrenar el modelo LSTM

8.3.1 Descargar y limpiar el texto

8.3.2 Creación de lotes de datos de entrenamiento

8.4 Construcción y entrenamiento del modelo LSTM

8.4.1 Construyendo un modelo LSTM

8.4.2 Entrenamiento del modelo LSTM

8.5 Generación de texto con el modelo LSTM entrenado

8.5.1 Generar texto prediciendo el siguiente token

8.5.2 Muestreo de temperatura y top-K en la generación de texto

9 Una implementación línea por línea de atención y Transformador

9.1 Introducción a la atención y al Transformador

9.1.1 El mecanismo de atención

9.1.2 La arquitectura del transformador

9.1.3 Diferentes tipos de transformadores

9.2 Construcción de un codificador

9.2.1 El mecanismo de atención

9.2.2 Creación de un codificador

9.3 Construcción de un transformador codificador-decodificador

9.3.1 Creación de una capa de decodificador

9.3.2 Creación de un transformador codificador-decodificador

9.4 Juntando todas las piezas

9.4.1 Definición de un generador

9.4.2 Creación de un modelo para traducir entre dos idiomas

10 Entrenamiento de un transformador para traducir del inglés al francés

10.1 Tokenización de subpalabras

10.1.1 Tokenización de frases en inglés y francés

10.1.2 Relleno de secuencias y creación de lotes

10.2 Incrustación de palabras y codificación posicional.

10.2.1 Incrustaciones de palabras

10.2.2 Codificación posicional

10.3 Entrenamiento del Transformer para la traducción del inglés al francés

10.3.1 Función de pérdida y el optimizador

10.3.2 El bucle de entrenamiento

10.4 Traducción del inglés al francés con el modelo entrenador.

11 Construyendo un Transformer generativo preentrenado desde cero

11.1 Arquitectura GPT-2 y autoatención causal

11.1.1 La arquitectura de GPT-2

11.1.2 Incrustación de palabras y codificación posicional en GPT-2

11.1.3 Autoatención causal en GPT-2

11.2 Construyendo GPT-2XL desde cero

11.2.1 Tokenización BPE

11.2.2 La función de activación unitaria lineal del error gaussiano

11.2.2 Autoatención causal

11.2.3 Construcción del modelo GPT-2XL

11.3 Carga de pesos preentrenados y generación de texto

11.3.1 Carga de parámetros preentrenados en GPT-2XL

11.3.2 Definición de una función generate() para producir texto

11.3.3 Generación de texto con GPT-2XL

12 Entrenamiento de un transformador para generar texto

12.1 Creación y entrenamiento de un GPT desde cero

12.1.1 La arquitectura de un GPT para generar texto

12.1.2 El proceso de entrenamiento del modelo GPT para generar texto

12.2 Tokenización del texto de las novelas de Hemingway

12.2.1 Tokenización del texto

12.2.2 Creación de lotes para entrenamiento

12.3 Creación de un GPT para generar texto

12.3.1 Hiperparámetros del modelo

12.3.2 Modelado del mecanismo causal de autoatención

12.3.3 Construcción del modelo GPT

12.4 Entrenamiento del modelo GPT para generar texto

12.4.1 Entrenamiento del modelo GPT

12.4.2 Una función para generar texto

12.4.3 Generación de texto con diferentes versiones del modelo entrenado

Parte 4. Aplicaciones y nuevos desarrollos

13 Generación de música con MuseGAN

13.1 Representación de música digital

13.1.1 Notas musicales, octava y tono

13.1.2 Una introducción a la música multipista

13.1.3 Representar música digitalmente: Rollos de piano

13.2 Un modelo para la generación de música.

13.2.1 Construyendo música con acordes, estilo, melodía y ritmo

13.2.2 Un plan para entrenar un MuseGAN

13.3 Preparación de los datos de entrenamiento para MuseGAN

13.3.1 Descarga de los datos de entrenamiento

13.3.2 Convertir objetos multidimensionales en piezas musicales

13.4 Construyendo un MuseGAN

13.4.1 Un crítico en MuseGAN

13.4.2 Un generador en MuseGAN

13.4.3 Optimizadores y la función de pérdida

13.5 Entrenamiento de MuseGAN para generar música

13.5.1 Entrenando el MuseGAN

13.5.2 Generando música con el MuseGAN entrenado

14 Construyendo y entrenando un musical Transformer

14.1 Introducción al transformador musical

14.1.1 Representación musical basada en la interpretación.

14.1.2 La arquitectura del transformador musical

14.1.3 Entrenamiento del transformador musical

14.2 Tokenización de piezas musicales

14.2.1 Descarga de datos de entrenamiento

14.2.2 Tokenización de archivos MIDI

14.2.3 Preparación de los datos de entrenamiento

14.3 Construyendo un GPT para generar música

14.3.1 Hiperparámetros en el transformador musical

14.3.2 Construyendo un transformador de música

14.4 Entrenamiento y uso del transformador de música.

14.4.1 Entrenamiento del transformador musical

14.4.2 Generación de música con el Transformer entrenado

15 Modelos de difusión y transformadores de texto a imagen.

15.1 Introducción a los modelos de difusión de eliminación de ruido.

15.1.1 El proceso de difusión hacia adelante

15.1.2 Uso del modelo U-Net para eliminar el ruido de las imágenes

15.1.3 Un plan para entrenar el modelo de eliminación de ruido U-Net

15.2 Preparación de los datos de entrenamiento.

15.2.1 Imágenes de flores como datos de entrenamiento

15.2.2 Visualizando el proceso de difusión hacia adelante

15.3 Construcción de un modelo U-Net de eliminación de ruido.

15.3.1 El mecanismo de atención en el modelo U-Net de eliminación de ruido

15.3.2 El modelo de eliminación de ruido U-Net

15.4 Entrenamiento y uso del modelo de eliminación de ruido U-Net

15.4.1 Entrenamiento del modelo de eliminación de ruido U-Net

15.4.2 Usando el modelo entrenado para generar imágenes de flores

15.5 Transformadores de texto a imagen

15.5.1 CLIP: Un transformador multimodal

15.5.2 Generación de texto a imagen con DALL-E 2

16 Modelos de lenguaje grandes preentrenados y la biblioteca LangChain

16.1 Generación de contenidos con la API de OpenAI

16.1.1 Tareas de generación de texto con la API de OpenAI

16.1.2 Generación de código con la API de OpenAI

16.1.3 Generación de imágenes con OpenAI DALL-E 2

16.1.4 Generación de voz con la API de OpenAI

16.2 Introducción a LangChain

16.2.1 La necesidad de la biblioteca LangChain

16.2.2 Uso de la API de OpenAI en LangChain

16.2.3 Incitación de cero disparos, de un disparo y de pocos disparos

16.3 Un agente sabelotodo de tiro cero en LangChain

16.3.1 Solicitud de una clave API de Wolfram Alpha

16.3.2 Creando un agente en LangChain

16.3.3 Agregar herramientas mediante el uso de GPT de OpenAI

16.3.4 Agregar herramientas para generar códigos e imágenes

16.4 Limitaciones y preocupaciones éticas de los LLM

16.4.1 Limitaciones de los LLM

16.4.2 Preocupaciones éticas para los LLM

Apéndice A. Instalación de Python, Jupyter Notebook y PyTorch

Apéndice B. Lectores mínimamente calificados y fundamentos del aprendizaje profundo

Índice

Portada

Prólogo

Conocí a Mark en la Cumbre de Innovación de PNC en la Universidad de Kentucky, donde ambos fuimos ponentes. Su tema fue "*Cómo aprenden las máquinas*". Desde nuestro primer encuentro, me impresionó su capacidad para explicar conceptos complejos de forma atractiva y fácil de entender. Su habilidad para descomponer ideas complejas en términos fáciles de entender y comprender fue realmente impresionante, un don que ahora comparte a través de su último libro, "*Aprenda IA Generativa con PyTorch*".

En Native AI, donde soy cofundador y director de operaciones, nos encargamos de generar datos sintéticos predictivos altamente precisos y robustos. La exploración de Mark de técnicas como la temperatura y el muestreo top-K para controlar la precisión del texto generado por IA es vanguardista. Estos métodos son esenciales para adaptar los resultados del procesamiento del lenguaje natural a casos de uso específicos, un tema que seguirá adquiriendo mayor importancia y valor comercial.

Aprenda IA Generativa con PyTorch es una guía completa que no solo introduce a los lectores al fascinante mundo de la IA generativa, sino que también les proporciona habilidades prácticas para construir e implementar sus propios modelos. El uso de PyTorch como marco de referencia por parte de Mark demuestra su flexibilidad y potencia para el desarrollo de modelos avanzados de IA. Desde modelos de memoria a corto plazo hasta autocodificadores variacionales, redes generativas antagónicas y Transformers, este libro abarca una impresionante variedad de temas.

El libro de Mark es un recurso invaluable para cualquiera que desee profundizar en la IA generativa, ya sean principiantes que buscan comprender los fundamentos o profesionales experimentados que buscan ampliar sus conocimientos y habilidades. Su capacidad para hacer accesibles y atractivos temas complejos garantiza que los lectores adquieran una sólida comprensión y la confianza para aplicar lo aprendido.

Es un honor para mí escribir el prólogo de este libro excepcional y me entusiasma la idea de que muchos lectores se beneficiarán de la experiencia de Mark. *Aprender IA Generativa con PyTorch* sin duda inspirará y educará, allanando el camino para futuras innovaciones en el campo de la IA Generativa.

— **SARAH SANDERS, COFUNDADORA Y DIRECTORA DE OPERACIONES DE NATIVEAI**

Prefacio

Mi fascinación por la IA generativa comenzó hace unos años, cuando vi por primera vez modelos que convertían imágenes de caballos en imágenes de cebras y Transformers que producían texto realista. Este libro es el resultado de mi experiencia construyendo y comprendiendo estos modelos desde cero. Es el libro que desearía haber tenido durante mis experimentos con diversos modelos generativos. Comienza con modelos sencillos, ayudando a los lectores a desarrollar habilidades fundamentales de aprendizaje profundo antes de avanzar a desafíos más complejos. Elegí PyTorch por su gráfico computacional dinámico y su sintaxis clara después de experimentar con TensorFlow.

Todos los modelos generativos de este libro son redes neuronales profundas. El libro comienza con un proyecto integral de aprendizaje profundo en PyTorch, ideal para quienes se inician en el campo. Cada capítulo está cuidadosamente estructurado para complementar el anterior. Primero, aprenderá a crear contenido básico, como formas, números e imágenes, utilizando redes generativas antagónicas con arquitecturas sencillas. A medida que avance, la complejidad aumentará, culminando en la creación de modelos de vanguardia como Transformers para generar texto y música, y modelos de difusión para generar imágenes de alta resolución.

En principio, este libro ofrece una exploración de diversos modelos de IA generativa. En un nivel más profundo, la evolución tecnológica refleja el funcionamiento de nuestra mente y la esencia de lo que significa ser humano. La importancia de las redes neuronales profundas en estos modelos generativos demuestra nuestro afán por comprender y replicar los complejos procesos del aprendizaje humano. Los modelos de IA generativa, inspirados en las maravillas de la biología evolutiva que moldearon nuestro ce-

rebro, aprenden de la gran cantidad de datos que encuentran, de forma similar a como los humanos aprendemos de los estímulos que nos rodean.

Las implicaciones de la IA generativa van mucho más allá de sus aplicaciones prácticas. Al estar a la vanguardia de esta revolución tecnológica, nos vemos obligados a reevaluar nuestra comprensión de la conciencia, la vida y la naturaleza misma de la existencia humana. Los paralelismos entre el aprendizaje automático y el aprendizaje humano son sorprendentes. Así como la IA generativa opera a través de redes neuronales inspiradas en el cerebro humano, nuestros pensamientos, emociones y comportamientos son el resultado de las redes neuronales dentro de nuestro cuerpo. Por lo tanto, el estudio de la IA generativa trasciende las fronteras tecnológicas, convirtiéndose en una exploración de la condición humana y los mecanismos que subyacen a nuestra conciencia. El estudio de la IA generativa nos lleva a una profunda especulación: ¿Somos los humanos, en esencia, sofisticados modelos de IA generativa?

En ese sentido, la IA generativa no es solo una herramienta: es un espejo que refleja nuestras preguntas existenciales más profundas. A medida que continuamos desarrollando e interactuando con estas tecnologías, no solo estamos configurando el futuro de *la inteligencia artificial*, sino también profundizando nuestra comprensión de *la inteligencia humana*. En definitiva, la exploración de la IA generativa es una exploración de nosotros mismos, un viaje al corazón de la conciencia y la esencia de la vida, que nos desafía a redefinir lo que significa ser consciente, estar vivo y ser humano.

Expresiones de gratitud

Muchas personas han contribuido a hacer realidad este libro. Jonathan Gennick, mi editor de adquisiciones en Manning, desempeñó un papel crucial al identificar los temas que los lectores desean aprender y al estructurar los capítulos para facilitar el aprendizaje. Un agradecimiento especial a mi editora de desarrollo, Rebecca Johnson, cuya incansable búsqueda de la perfección mejoró significativamente el libro. Me animó a explicar conceptos complejos de forma clara y comprensible.

Mi gratitud también se extiende a mi editor técnico, Emmanuel Maggiori, autor de *Smart Until It's Dumb* (Applied Maths Ltd., 2023). Cada vez que me dejaba llevar por mis escritos sobre el maravilloso potencial de la IA, Em-

manuel siempre señalaba rápidamente sus limitaciones. Si bien mi cita favorita es «Cualquier tecnología suficientemente avanzada es indistinguible de la magia» de Arthur C. Clarke, la perspectiva de Emmanuel sobre la IA se puede resumir en el título de su libro. Creo que este choque de puntos de vista proporciona a nuestros lectores una perspectiva más equilibrada.

Gracias a todos los revisores: Abhilash Babu, Ankit Virmani, Arpit Singh, Christopher Kottmyer, David Cronkite, Eduardo Rienzi, Erim Erturk, Francis Osei Annin, Georg Piwonka, Holger Voges, Ian Long, Japneet Singh, Karrtik Iyer, Kollin Trujillo, Michael Petrey, Mirerfan Gheibi, Nathan Crocker, Neeraj Gupta, Neha Shetty, Palak Mathur, Peter Henstock, Piergiorgio Faraglia, Rajat Kant Goel, Ramaa Vissa, Ravi Kiran Bamidi, Richard Tobias, Ruud Gijsen, Slavomir Furman, Sumit Pal, Thiago Britto Borges, Tony Holdroyd, Ursin Stauss, Vamsi Srinivas Parasa, Viju Kothuvatiparambil y Walter Alexander Mata López, sus sugerencias ayudaron a hacer de este un libro mejor.

También deseo agradecer al equipo de producción de Manning Publications por ayudarme a llevar este proyecto a término.

Por último, quiero expresar mi más profundo agradecimiento a mi esposa, Ivey Zhang, y a mi hijo, Andrew Liu, por su constante apoyo durante este viaje.

Acerca de este libro

Aprende IA Generativa con PyTorch te guía en la creación de diversos contenidos (formas, números, imágenes, texto y música) desde cero. Comienza con modelos sencillos, ayudando a los lectores a desarrollar habilidades fundamentales de aprendizaje profundo antes de avanzar a desafíos más complejos. Todos los modelos generativos de este libro son redes neuronales profundas. El libro comienza con un proyecto integral de aprendizaje profundo en PyTorch, ideal para quienes se inician en el campo. Cada capítulo está cuidadosamente estructurado para complementar el anterior. Primero, crearás contenido básico como formas, números e imágenes utilizando redes generativas antagónicas con arquitecturas sencillas. A medida que avances, la complejidad aumentará, culminando en la creación de modelos de vanguardia como transformadores y modelos de difusión.

¿Quién debería leer este libro?

Aprenda IA Generativa con PyTorch está diseñado para entusiastas del aprendizaje automático y científicos de datos de diversos sectores empresariales con conocimientos intermedios de programación en Python. Este libro busca enseñar técnicas de IA generativa para crear contenido novedoso e innovador, como imágenes, texto, patrones, números, formas y audio, para impulsar tanto los negocios de sus empleadores como sus propias carreras profesionales. Si bien existen numerosos materiales de aprendizaje gratuitos disponibles en línea que cubren temas específicos, este libro lo consolida todo en un formato claro, fácil de seguir y actualizado, lo que lo convierte en un recurso invaluable para cualquiera que aspire a convertirse en un experto en IA generativa.

Asumo que los lectores tienen un conocimiento sólido de Python. Deberían estar familiarizados con los tipos de variables, las funciones y clases de Python, y la instalación de bibliotecas y paquetes de Python de terceros. Si necesitan repasar estas habilidades, el tutorial gratuito de Python en línea de W3Schools es un excelente recurso (<https://www.w3schools.com/python/>).

También debe tener conocimientos básicos de aprendizaje automático, en particular de redes neuronales y aprendizaje profundo. De no ser así, un buen libro para este propósito es "*Deep Learning with PyTorch*" de Stevens, Antiga y Viehmann (2020), también publicado por Manning Publications. El Apéndice B de este libro ofrece una revisión de conceptos clave como funciones de pérdida, funciones de activación y optimizadores, esenciales para el desarrollo y entrenamiento de redes neuronales profundas. Sin embargo, este apéndice no pretende ser un tutorial exhaustivo sobre estos temas.

Cómo está organizado este libro: una hoja de ruta

Este libro consta de 16 capítulos, organizados en cuatro partes. La Parte I presenta la IA generativa y el aprendizaje profundo con PyTorch.

- El capítulo 1 explica qué es la IA generativa y el motivo detrás de la selección de PyTorch sobre otros marcos de IA como TensorFlow para construir modelos generativos en este libro.

- El capítulo 2 utiliza PyTorch para crear redes neuronales profundas que permiten realizar clasificaciones binarias y multicategoría, lo que le permitirá familiarizarse con el aprendizaje profundo y las tareas de clasificación. El objetivo es prepararlo para los próximos capítulos, donde utilizará redes neuronales profundas en PyTorch para crear diversos modelos generativos.
- El capítulo 3 presenta las redes generativas antagónicas (GAN). Aprenderá a usarlas para generar formas y secuencias de números con ciertos patrones.

La Parte II cubre la generación de imágenes.

- El capítulo 4 explica cómo construir y entrenar GAN para generar imágenes en color de alta resolución. En particular, aprenderá a usar redes neuronales convolucionales para capturar características espaciales en imágenes. También aprenderá a usar capas convolucionales transpuestas para sobremuestrear y generar mapas de características de alta resolución en imágenes.
- El capítulo 5 detalla dos maneras de seleccionar características en imágenes generadas. El primer método implica seleccionar vectores específicos en el espacio latente. El segundo método utiliza una GAN condicional, donde se construye y entrena una GAN con datos etiquetados.
- El capítulo 6 le enseña cómo utilizar un CycleGAN para traducir imágenes entre dos dominios, como imágenes con cabello negro e imágenes con cabello rubio o imágenes de caballos e imágenes de cebras.
- El capítulo 7 explica cómo generar imágenes de alta resolución utilizando otro modelo generativo: autocodificadores y su variante, los autocodificadores variacionales.

La Parte III se adentra en el procesamiento del lenguaje natural y la generación de texto.

- El capítulo 8 aborda la generación de texto con una red neuronal recurrente. A lo largo del camino, aprenderá cómo funcionan la tokenización y la incrustación de palabras. También aprenderá a generar texto autorregresivamente con el modelo entrenado y a usar la temperatura y el muestreo top-K para controlar la creatividad del texto generado.

- El capítulo 9 construye un Transformer desde cero, basado en el artículo "Atención es todo lo que necesitas", para traducir entre dos idiomas. Implementarás línea por línea el mecanismo de atención multicausal y un Transformer codificador-decodificador.
- El capítulo 10 entrena el Transformer que construiste en el capítulo 9 con más de 47,000 pares de traducciones del inglés al francés. Aprenderás a traducir frases comunes del inglés al francés con el modelo entrenado.
- El capítulo 11 construye GPT-2XL, la versión más grande de GPT-2, desde cero. Después, aprenderá a extraer los pesos preentrenados de Hugging Face y a cargarlos en su propio modelo GPT-2 para generar texto.
- El capítulo 12 construye una versión reducida del modelo GPT con aproximadamente 5 millones de parámetros para que puedas entrenarlo en una computadora convencional. Utilizarás tres novelas de Ernest Hemingway como datos de entrenamiento. El modelo entrenado puede generar texto al estilo de Hemingway.

La Parte IV analiza algunas aplicaciones prácticas de los modelos generativos del libro y los desarrollos más recientes en el campo de la IA generativa.

- El capítulo 13 construye y entrena un MuseGAN para generar música. MuseGAN trata cada pieza musical como un objeto multidimensional similar a una imagen. El generador produce una pieza musical completa y la envía al crítico para su evaluación. Posteriormente, el generador modifica la música basándose en la retroalimentación del crítico hasta que se asemeja a la música real del conjunto de datos de entrenamiento.
- El capítulo 14 adopta un enfoque diferente para la creación musical con IA. En lugar de tratar una pieza musical como un objeto multidimensional, la trata como una secuencia de eventos musicales. Posteriormente, aplicará técnicas de generación de texto para predecir el siguiente elemento de una secuencia.
- El capítulo 15 presenta los modelos de difusión, que constituyen la base de todos los principales transformadores de texto a imagen (como DALL-E 2 o Imagen). Construirá y entrenará un modelo de difusión para generar imágenes de flores de alta resolución.

- El capítulo 16 finaliza el libro con un proyecto en el que se utiliza la biblioteca LangChain para combinar modelos de lenguaje grandes previamente entrenados con las API de Wolfram Alpha y Wikipedia para crear un asistente personal sabelotodo.

El Apéndice A explica cómo instalar PyTorch en su computadora, con o sin una GPU compatible con la arquitectura de dispositivo unificado de cómputo. El Apéndice B proporciona información sobre los conocimientos necesarios para continuar con los proyectos de este libro y algunos conceptos básicos de aprendizaje profundo, como funciones de pérdida, funciones de activación y optimizadores.

Acerca del código

Este libro contiene numerosos ejemplos de código fuente, tanto en listas numeradas como en formato de texto normal. En ambos casos, el código fuente se formatea `fixed-width font like this` para distinguirlo del texto normal. En ocasiones, también se utiliza `in bold` para resaltar el código que ha cambiado con respecto a pasos anteriores del capítulo, como cuando se añade una nueva función a una línea de código existente.

En muchos casos, el código fuente original se ha reformateado; hemos añadido saltos de línea y rediseñado la sangría para adaptarla al espacio disponible en el libro. En raras ocasiones, ni siquiera esto fue suficiente, y los listados incluyen marcadores de continuación de línea (`→`). Además, los comentarios del código fuente se han eliminado con frecuencia de los listados cuando este se describe en el texto. Muchos de los listados se acompañan de anotaciones de código que resaltan conceptos importantes.

Puede obtener fragmentos de código ejecutables de la versión liveBook (en línea) de este libro en <https://livebook.manning.com/book/learn-generative-ai-with-pytorch>. Todos los programas de Python de este libro están disponibles para su descarga desde el sitio web de Manning (www.manning.com) y desde el repositorio de GitHub del libro (<https://github.com/markhliu/DGAI>). Los programas están organizados por capítulos, cada uno en un único archivo de Jupyter Notebook. Consulte el apéndice A para saber cómo instalar Python, PyTorch y Jupyter Notebook en su ordenador.

Foro de discusión de liveBook

La compra de *Learn Generative AI con PyTorch* incluye acceso gratuito a liveBook, la plataforma de lectura en línea de Manning. Con las funciones exclusivas de discusión de liveBook, puedes añadir comentarios al libro globalmente o a secciones o párrafos específicos. Es muy fácil tomar notas, hacer y responder preguntas técnicas, y recibir ayuda del autor y otros usuarios. Para accederPara acceder al foro, visite

<https://livebook.manning.com/book/learn-generative-ai-with-pytorch/discussion>. También puede obtener más información sobre los foros de Manning y las normas de conducta en
<https://livebook.manning.com/discussion>.

El compromiso de Manning con nuestros lectores es brindar un espacio donde se pueda entablar un diálogo significativo entre lectores individuales y entre lectores y el autor. No implica un compromiso de participación específico por parte del autor, cuya contribución al foro es voluntaria (y no remunerada). Le sugerimos que intente hacerle algunas preguntas interesantes al autor para que no pierda el interés. El foro y los archivos de discusiones anteriores estarán disponibles desde el sitio web de la editorial mientras el libro esté impreso.

Sobre el autor



EL DR. MARK LIU es profesor titular de finanzas y director (fundador) del programa de Maestría en Ciencias en Finanzas de la Universidad de Kentucky. Es autor de dos libros: *Make Python Talk* (No Starch Press, 2021) y

Machine Learning, Animated (CRC Press, 2023). Mark cuenta con más de 20 años de experiencia en programación. Obtuvo su doctorado en finanzas en Boston College. Ha publicado sus investigaciones en importantes revistas financieras, como *Journal of Financial Economics*, *Journal of Financial and Quantitative Analysis* y *Journal of Corporate Finance*.

Sobre la ilustración de la portada

La figura de la portada de *Learn Generative AI with PyTorch*, titulada “L'Agent de la rue de Jerusalem” o “El agente de la calle Jerusalén”, está tomada de un libro de Louis Curmer publicado en 1841. Cada ilustración está finamente dibujada y coloreada a mano.

En aquella época, era fácil identificar dónde vivía la gente y cuál era su oficio o posición social simplemente por su vestimenta. Manning celebra la inventiva y la iniciativa de la industria informática con portadas basadas en la rica diversidad de la cultura regional de siglos atrás, revividas con imágenes de colecciones como esta.

Parte 1. Introducción a la IA generativa

¿Qué es la IA generativa? ¿En qué se diferencia de sus contrapartes no generativas, los modelos discriminativos? ¿Por qué elegimos PyTorch como marco de IA en este libro?

En esta parte, respondemos a estas preguntas. Además, todos los modelos de IA generativa de este libro son redes neuronales profundas. Por lo tanto, aprenderá a usar PyTorch para crear redes neuronales profundas y realizar clasificaciones binarias y multicategoría, lo que le permitirá familiarizarse con el aprendizaje profundo y las tareas de clasificación. El objetivo es prepararlo para los próximos capítulos, donde usará redes neuronales profundas en PyTorch para crear diversos modelos generativos. También aprenderá a usar PyTorch para construir y entrenar redes generativas antagonísticas y generar formas y secuencias de números.

1. ¿Qué es la IA generativa y por qué PyTorch?

Este capítulo cubre

- IA generativa vs. IA no generativa
- Por qué PyTorch es ideal para el aprendizaje profundo y la IA generativa
- El concepto de redes generativas antagónicas
- Los beneficios del mecanismo de atención y los Transformers
- Ventajas de crear modelos de IA generativos desde cero

La IA generativa ha impactado significativamente el panorama global, captando la atención de todos y convirtiéndose en un punto focal desde la llegada de ChatGPT en noviembre de 2022. Este avance tecnológico ha revolucionado numerosos aspectos de la vida cotidiana, marcando el comienzo de una nueva era en la tecnología e inspirando a numerosas startups a explorar las amplias posibilidades que ofrecen los diversos modelos generativos.

Considere los avances de Midjourney, una empresa pionera que ahora crea imágenes realistas de alta resolución a partir de breves entradas de texto. De igual manera, Freshworks, una empresa de software, ha acelerado drásticamente el desarrollo de aplicaciones, reduciendo el tiempo requerido de un promedio de 10 semanas a tan solo unos días, una hazaña lograda gracias a las capacidades de ChatGPT (véase el artículo *de Forbes* "10 increíbles ejemplos reales de cómo las empresas están usando ChatGPT en 2023", de Bernard Barr, 2023, <https://mng.bz/Bgx0>). Por ejemplo, algunos elementos de esta misma introducción se han mejorado mediante IA generativa, lo que demuestra su capacidad para refinar el contenido y hacerlo más atractivo.

NOTA: ¿Qué mejor manera de explicar la IA generativa que dejar que se desarrolle sola? Le pedí a ChatGPT que reescribiera un borrador preliminar de esta introducción de una manera más atractiva antes de finalizarla.

Las repercusiones de este avance tecnológico van mucho más allá de estos ejemplos. Las industrias están experimentando una disrupción significativa debido a las capacidades avanzadas de la IA generativa. Esta tecnología ahora produce ensayos comparables a los escritos por humanos, compone música que evoca composiciones clásicas y genera rápidamente documentos legales complejos, tareas que normalmente requieren un esfuerzo y tiempo humanos considerables. Tras el lanzamiento de ChatGPT, la plataforma educativa CheggMate experimentó una caída significativa en el valor de sus acciones. Además, el Gremio de Escritores de Estados Unidos, durante una huelga reciente, llegó a un consenso para poner límites a la intrusión de la IA en la escritura de guiones y la edición (véase el artículo *de WIRED* «Los escritores de Hollywood alcanzaron un acuerdo sobre IA que reescribirá la historia», de Will Bedingfield, 2023, <https://mng.bz/1ajj>).

NOTA: CheggMate cobra a los estudiantes universitarios por las respuestas a sus preguntas por parte de especialistas. Muchos de estos trabajos ahora se pueden realizar con ChatGPT o herramientas similares a un precio mucho menor.

Esto plantea varias preguntas: ¿Qué es la IA generativa y en qué se diferencia de otras tecnologías de IA? ¿Por qué está causando una disrupción tan generalizada en diversos sectores? ¿Cuál es el mecanismo subyacente de la IA generativa y por qué es importante comprenderlo?

Este libro ofrece una exploración a fondo de la IA generativa, una tecnología revolucionaria que está transformando numerosas industrias gracias a su eficiente y rápida capacidad de creación de contenido. En concreto, aprenderá a utilizar modelos generativos de vanguardia para crear diversos tipos de contenido: formas, números, imágenes, texto y audio. Además, en lugar de tratar estos modelos como cajas negras, aprenderá a crearlos desde cero para comprender a fondo el funcionamiento interno de la IA generativa. En palabras del físico Richard Feynman: «Lo que no puedo crear, no lo entiendo».

Todos estos modelos se basan en redes neuronales profundas, y usarás Python y PyTorch para construirlos, entrenarlos y usarlos. Elegimos Python por su sintaxis intuitiva, compatibilidad multiplataforma y amplio soporte de la comunidad. También elegimos PyTorch sobre otros frameworks como TensorFlow por su facilidad de uso y adaptabilidad a diversas arquitectu-

ras de modelos. Python es ampliamente considerado como la herramienta principal para el aprendizaje automático (ML), y PyTorch se ha vuelto cada vez más popular en el campo de la IA. Por lo tanto, usar Python y PyTorch te permite estar al tanto de los nuevos desarrollos en IA generativa. Dado que PyTorch permite la aceleración del entrenamiento de la unidad de procesamiento gráfico (GPU), ¡entrenarás estos modelos en cuestión de minutos u horas y serás testigo de la IA generativa en acción!

1.1 Introducción a la IA generativa y PyTorch

Esta sección explica qué es la IA generativa y en qué se diferencia de sus contrapartes no generativas: los modelos discriminativos. La IA generativa es una categoría de tecnologías con la notable capacidad de producir diversos formatos de contenido nuevo, incluyendo texto, imágenes, audio, video, código fuente y patrones complejos. La IA generativa crea mundos completamente nuevos de contenido novedoso e innovador; ChatGPT es un ejemplo notable. En contraste, el modelado discriminativo se centra principalmente en la tarea de reconocer y categorizar contenido preexistente.

1.1.1 ¿Qué es la IA generativa?

La IA generativa es un tipo de inteligencia artificial que crea contenido nuevo, como texto, imágenes o música, mediante el aprendizaje de patrones a partir de datos existentes. Se diferencia de los modelos discriminativos, que se especializan en discernir disparidades entre distintas instancias de datos y en el aprendizaje de los límites entre clases. La Figura 1.1 ilustra la diferencia entre estos dos métodos de modelado. Por ejemplo, al enfrentarse a una serie de imágenes de perros y gatos, un modelo discriminativo determina si cada imagen representa un perro o un gato capturando algunas características clave que los distinguen (p. ej., los gatos tienen narices pequeñas y orejas puntiagudas). Como muestra la mitad superior de la figura, un modelo discriminativo toma datos como entradas y produce probabilidades de diferentes etiquetas, que denotamos como $\text{Prob}(\text{perro})$ y $\text{Prob}(\text{gato})$. Luego, podemos etiquetar las entradas en función de las probabilidades predichas más altas.

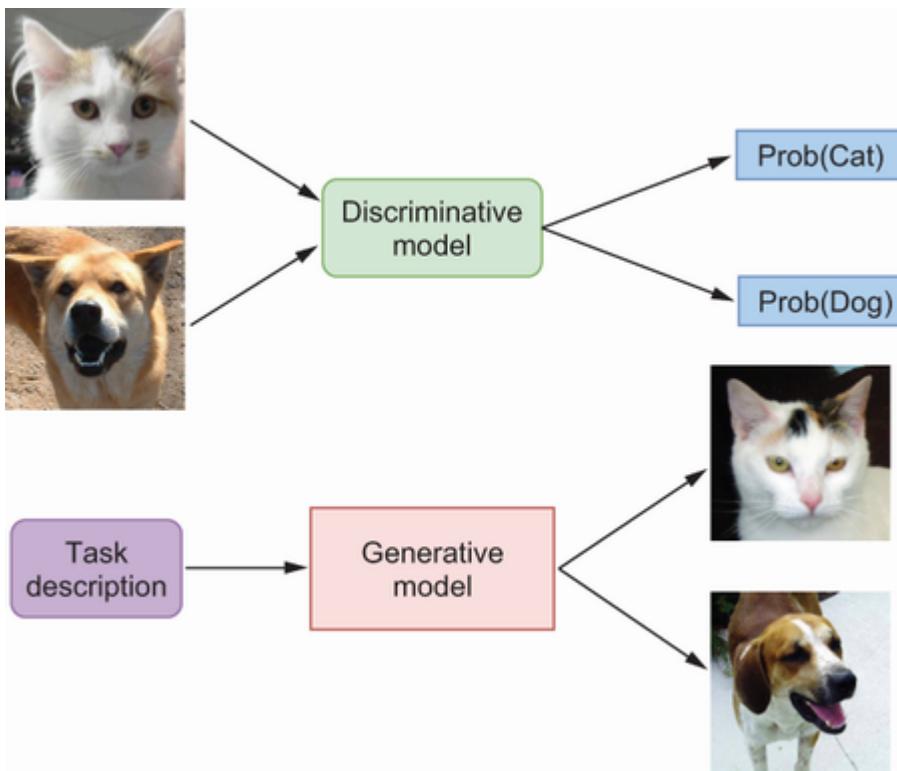


Figura 1.1 Comparación de modelos generativos y discriminativos. Un modelo discriminativo (mitad superior de la figura) toma datos como entrada y genera probabilidades de diferentes etiquetas, que denotamos como $\text{Prob}(\text{perro})$ y $\text{Prob}(\text{gato})$. Por el contrario, un modelo generativo (mitad inferior) adquiere una comprensión profunda de las características definitorias de estas imágenes para sintetizar nuevas imágenes que representan perros y gatos.

En cambio, los modelos generativos exhiben una capacidad única para generar nuevas instancias de datos. En el contexto de nuestro ejemplo de perros y gatos, un modelo generativo adquiere una comprensión profunda de las características definitorias de estas imágenes para sintetizar nuevas imágenes que representan perros y gatos. Como muestra la mitad inferior de la figura 1.1, un modelo generativo toma como entrada descripciones de tareas (como la variación de valores en un espacio latente que resulta en diferentes características en la imagen generada, que analizaremos en detalle en los capítulos 4 a 6) y produce imágenes completamente nuevas de perros y gatos.

Desde una perspectiva estadística, al presentar ejemplos de datos con características X , que describen la entrada y sus correspondientes etiquetas Y , los modelos discriminativos asumen la responsabilidad de predecir probabilidades condicionales, específicamente la probabilidad $\text{prob}(Y|X)$. Por el contrario, los modelos generativos intentan comprender la distribución de probabilidad conjunta de las características de entrada X y la variable objetivo Y , denotada como $\text{prob}(X, Y)$. Con este conocimiento, extraen muestras de la distribución para generar nuevas instancias de X .

Existen diferentes tipos de modelos generativos según las formas específicas de contenido que se desee crear. En este libro, nos centramos principalmente en dos tecnologías destacadas. Redes Generativas Antagónicas (GAN) y Transformadores (aunque también abordaremos los autocodificadores variacionales y los modelos de difusión). El término "antagónico" en las GAN se refiere a que dos redes neuronales compiten entre sí en un juego de suma cero: la red generativa intenta crear instancias de datos indistinguibles de las muestras reales, mientras que la red discriminativa intenta identificar las muestras generadas de las reales. La competencia entre ambas redes conduce a la mejora de ambas, lo que finalmente permite al generador crear datos altamente realistas. Los Transformadores son redes neuronales profundas que pueden resolver eficientemente tareas de predicción secuencia a secuencia, y los explicaremos con más detalle más adelante en este capítulo.

Las GAN, reconocidas por su facilidad de implementación y versatilidad, permiten a personas con conocimientos básicos de aprendizaje profundo construir sus modelos generativos desde cero. Estos versátiles modelos pueden generar una gran variedad de creaciones, desde formas geométricas y patrones intrincados, como se ejemplifica en el capítulo 3 de este libro, hasta imágenes en color de alta calidad, como rostros humanos, que aprenderá a generar en el capítulo 4. Además, las GAN exhiben la capacidad de transformar el contenido de las imágenes, transformando sin problemas la imagen de un rostro humano rubio en una de cabello negro, un fenómeno que se analiza en el capítulo 6. Cabe destacar que extienden su capacidad creativa al campo de la generación musical, produciendo composiciones musicales con un sonido realista, como se demuestra en el capítulo 13.

A diferencia de la generación de formas, números o imágenes, el arte de la generación de texto plantea enormes desafíos, principalmente debido a la naturaleza secuencial de la información textual, donde el orden y la disposición de caracteres y palabras individuales tienen un significado significativo. Para afrontar esta complejidad, recurrimos a los Transformers, redes neuronales profundas diseñadas para abordar con eficacia tareas de predicción secuencia a secuencia. A diferencia de sus predecesores, como las redes neuronales recurrentes (RNN) o las redes neuronales convolucionales (CNN), los Transformers destacan por capturar dependencias complejas y de largo alcance inherentes tanto a las secuencias de entrada como a las

de salida. Cabe destacar que su capacidad de entrenamiento en paralelo (un método de entrenamiento distribuido en el que un modelo se entrena en múltiples dispositivos simultáneamente) ha reducido sustancialmente los tiempos de entrenamiento, lo que nos permite entrenar Transformers con grandes cantidades de datos.

La arquitectura revolucionaria de Transformers sustenta el surgimiento de grandes modelos de lenguaje (LLM; redes neuronales profundas con una gran cantidad de parámetros y entrenadas con grandes conjuntos de datos), como ChatGPT, BERT, DALL-E y T5. Esta arquitectura transformadora constituye la base del reciente auge en los avances de la IA, impulsado por la introducción de ChatGPT y otros modelos generativos preentrenados de Transformers (GPT).

En las secciones siguientes, profundizaremos en el funcionamiento interno de estas dos tecnologías pioneras: sus mecanismos subyacentes y las innumerables posibilidades que abren.

1.1.2 El lenguaje de programación Python

Supongo que tienes conocimientos prácticos de Python. Para seguir el contenido del libro, necesitas conocer los conceptos básicos de Python, como funciones, clases, listas, diccionarios, etc. Si no es así, existen numerosos recursos gratuitos en línea para ayudarte a empezar. Sigue las instrucciones del apéndice A para instalar Python. Después, crea un entorno virtual para este libro e instala Jupyter Notebook como entorno informático para los proyectos que se incluyen en él.

Python se ha consolidado como el lenguaje de programación líder a nivel mundial desde finales de 2018, como documenta *The Economist* (véase el artículo «Python se está convirtiendo en el lenguaje de programación más popular del mundo» del equipo de datos de *The Economist*, 2018, <https://mng.bz/2gj0>). Python no solo es gratuito para todos, sino que también permite a otros usuarios crear y modificar bibliotecas. Python cuenta con un enorme ecosistema impulsado por la comunidad, por lo que es fácil encontrar recursos y ayuda de otros entusiastas de Python. Además, a los programadores de Python les encanta compartir su código, así que, en lugar de reinventar la rueda, puedes importar bibliotecas prediseñadas y compartir las tuyas con la comunidad de Python.

No importa si usas Windows, Mac o Linux, Python te cubre las espaldas. Es un lenguaje multiplataforma, aunque el proceso de instalación de software y bibliotecas puede variar un poco según tus necesidades. sistema operativo r, pero no te preocupes, te mostraré cómo hacerlo en el apéndice A. Una vez que todo está configurado, el código Python se comporta de la misma manera en diferentes sistemas.

Python es un lenguaje expresivo ideal para el desarrollo de aplicaciones generales. Su sintaxis es fácil de comprender, lo que facilita su comprensión y uso para los entusiastas de la IA. Si tiene algún problema con las bibliotecas de Python mencionadas en este libro, puede buscar soluciones en foros de Python o visitar sitios como Stack Overflow (<https://stackoverflow.com/questions/tagged/python>). Y si todo lo demás falla, no dude en contactarme para obtener ayuda.

Por último, Python ofrece una amplia colección de bibliotecas que facilitan la creación de modelos generativos (en comparación con otros lenguajes como C++ o R). En este proyecto, utilizaremos exclusivamente PyTorch como nuestro framework de IA y, en breve, explicaré por qué lo elegimos en lugar de competidores como TensorFlow.

1.1.3 Uso de PyTorch como nuestro marco de IA

Ahora que hemos decidido usar Python como lenguaje de programación para este libro, elegiremos un framework de IA adecuado para el modelo generativo. Los dos frameworks de IA más populares en Python son PyTorch y TensorFlow. En este libro, usamos PyTorch en lugar de TensorFlow por su facilidad de uso, y les recomiendo encarecidamente que hagan lo mismo.

PyTorch es una biblioteca de aprendizaje automático (ML) de código abierto desarrollada por el laboratorio de investigación de IA de Meta. Basada en el lenguaje de programación Python y la biblioteca Torch, PyTorch busca ofrecer una plataforma flexible e intuitiva para la creación y el entrenamiento de modelos de aprendizaje profundo. Torch, predecesora de PyTorch, era una biblioteca de ML para la creación de redes neuronales profundas en C con un contenedor Lua, pero su desarrollo se interrumpió. PyTorch se diseñó para satisfacer las necesidades de investigadores y desa-

rrolladores, proporcionando un marco más intuitivo y adaptable para proyectos de aprendizaje profundo.

Un grafo computacional es un concepto fundamental en el aprendizaje profundo que desempeña un papel crucial en el cálculo eficiente de operaciones matemáticas complejas, especialmente aquellas que involucran matrices multidimensionales o tensores. Un grafo computacional es un grafo dirigido donde los nodos representan operaciones matemáticas y las aristas representan los datos que fluyen entre estas operaciones. Uno de los usos clave de los grafos computacionales es el cálculo de derivadas parciales al implementar algoritmos de retropropagación y descenso de gradiente. La estructura del grafo permite el cálculo eficiente de los gradientes necesarios para actualizar los parámetros del modelo durante el entrenamiento. PyTorch crea y modifica el grafo sobre la marcha, lo que se denomina grafo computacional dinámico. Esto lo hace más adaptable a diversas arquitecturas de modelo y simplifica la depuración. Además, al igual que TensorFlow, PyTorch proporciona computación acelerada mediante entrenamiento en GPU, lo que puede reducir significativamente el tiempo de entrenamiento en comparación con el entrenamiento en la unidad central de procesamiento (CPU).

El diseño de PyTorch se integra perfectamente con el lenguaje de programación Python. Su sintaxis es concisa y fácil de entender, lo que lo hace accesible tanto para principiantes como para desarrolladores experimentados. Tanto investigadores como desarrolladores aprecian PyTorch por su flexibilidad. Les permite experimentar con ideas innovadoras rápidamente gracias a su gráfico computacional dinámico y su interfaz sencilla. Esta flexibilidad es crucial en el campo de la IA generativa, en rápida evolución. PyTorch también cuenta con una comunidad en rápido crecimiento que contribuye activamente a su desarrollo. Esto se traduce en un amplio ecosistema de bibliotecas, herramientas y recursos para desarrolladores.

PyTorch destaca en el aprendizaje por transferencia, una técnica que permite ajustar modelos preentrenados, diseñados para una tarea general, para tareas específicas. Investigadores y profesionales pueden utilizar fácilmente estos modelos, ahorrando tiempo y recursos computacionales. Esta característica es especialmente importante en la era de los LLM preentrenados y nos permite adoptarlos para tareas posteriores como la clasificación, el resumen y la generación de textos.

PyTorch es compatible con otras bibliotecas de Python, como NumPy y Matplotlib. Esta interoperabilidad permite a los científicos de datos e ingenieros integrar PyTorch sin problemas en sus flujos de trabajo, mejorando así la productividad. PyTorch también es conocido por su compromiso con el desarrollo impulsado por la comunidad. Evoluciona rápidamente, con actualizaciones y mejoras periódicas basadas en el uso real y las opiniones de los usuarios, lo que garantiza que se mantenga a la vanguardia de la investigación y el desarrollo de IA.

El Apéndice A proporciona instrucciones detalladas sobre cómo instalar PyTorch en su computadora. Siga las instrucciones para instalar PyTorch en el entorno virtual para este libro. En caso de que no tenga una GPU habilitada para Compute Unified Device Architecture (CUDA) instalada en su computadora, todos los programas de este libro también son compatibles con el entrenamiento de CPU. Mejor aún, proporcionaré los modelos entrenados en el repositorio de GitHub del libro

<https://github.com/markhliu/DGAI> para que pueda ver los modelos entrenados en acción (en caso de que el modelo entrenado sea demasiado grande, los proporcionaré en mi sitio web personal

<https://gattonweb.uky.edu/faculty/lium/>). En el capítulo 2, se sumergirá en PyTorch en profundidad. Primero aprenderá la estructura de datos en PyTorch, Tensor, que contiene números y matrices y proporciona funciones para realizar operaciones. Luego aprenderá a realizar un proyecto de aprendizaje profundo de extremo a extremo usando PyTorch. En concreto, crearás una red neuronal en PyTorch y usarás imágenes de prendas de vestir y sus etiquetas correspondientes para entrenarla. Una vez hecho esto, usarás el modelo entrenado para clasificar las prendas de vestir en 10 tipos de etiquetas diferentes. El proyecto te preparará para usar PyTorch para construir y entrenar diversos modelos generativos en capítulos posteriores.

1.2 GAN

Esta sección ofrece primero una visión general del funcionamiento de las GAN. A continuación, utilizamos la generación de imágenes de caras de anime como ejemplo para mostrar el funcionamiento interno de las GAN. Finalmente, analizaremos sus usos prácticos.

1.2.1 Una descripción general de alto nivel de las GAN

Las GAN representan una categoría de modelos generativos inicialmente propuestos por Ian Goodfellow y sus colaboradores en 2014 (“Generative Adversarial Nets”, <https://arxiv.org/abs/1406.2661>). Las GAN se han vuelto extremadamente populares en los últimos años porque son fáciles de construir y entrenar, y pueden generar una amplia variedad de contenido.

Como verá en el ejemplo ilustrativo de la siguiente subsección, las GAN emplean una arquitectura de red dual que comprende un modelo generativo encargado de capturar la distribución de datos subyacente para generar contenido y un modelo discriminativo que sirve para estimar la probabilidad de que una muestra dada se origine del conjunto de datos de entrenamiento auténtico (considerado como “real”) en lugar de ser un producto del modelo generativo (considerado como “falso”). El objetivo principal del modelo es producir nuevas instancias de datos que se asemejen mucho a las del conjunto de datos de entrenamiento. La naturaleza de los datos generados por las GAN depende de la composición del conjunto de datos de entrenamiento. Por ejemplo, si los datos de entrenamiento consisten en imágenes en escala de grises de prendas de vestir, las imágenes sintetizadas se asemejarán mucho a dichas prendas. Por el contrario, si el conjunto de datos de entrenamiento incluye imágenes en color de rostros humanos, las imágenes generadas también se asemejarán a rostros humanos.

Observe la figura 1.2: la arquitectura de nuestra GAN y sus componentes. Para entrenar el modelo, se presentan al discriminador (centro) tanto muestras reales del conjunto de datos de entrenamiento (como se muestra en la parte superior de la figura 1.2) como muestras falsas creadas por el generador (izquierda). El objetivo principal del generador es crear instancias de datos prácticamente indistinguibles de las del conjunto de datos de entrenamiento. Por otro lado, el discriminador intenta distinguir las muestras falsas generadas por el generador de las reales. Estas dos redes participan en un proceso competitivo continuo, similar al juego del gato y el ratón, intentando superarse mutuamente iterativamente.

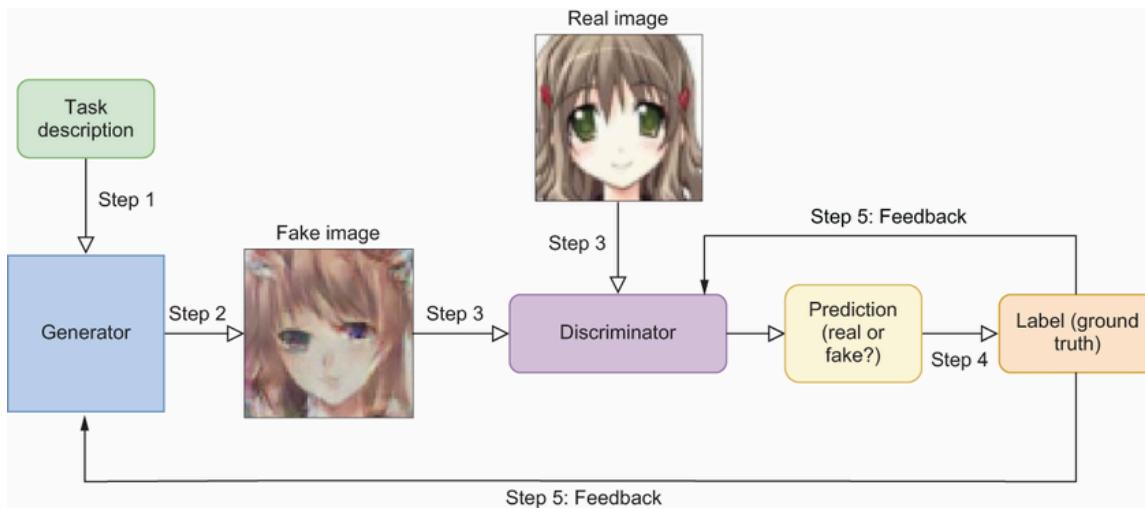


Figura 1.2 Arquitectura de las GAN y sus componentes. Las GAN emplean una arquitectura de red dual que comprende un modelo generativo (izquierda), encargado de capturar la distribución de datos subyacente, y un modelo discriminativo (centro), que sirve para estimar la probabilidad de que una muestra dada provenga del conjunto de datos de entrenamiento auténtico (considerado "real"), en lugar de ser producto del modelo generativo (considerado "falso").

El proceso de entrenamiento del modelo GAN implica múltiples iteraciones. En cada iteración, el generador toma algún tipo de descripción de la tarea (paso 1) y la utiliza para crear imágenes falsas (paso 2). Las imágenes falsas, junto con las imágenes reales del conjunto de entrenamiento, se presentan al discriminador (paso 3). El discriminador intenta clasificar cada muestra como real o falsa. Luego, compara la clasificación con las etiquetas reales, la verdad fundamental (paso 4). Tanto el discriminador como el generador reciben retroalimentación (paso 5) de la clasificación y mejoran sus capacidades: mientras el discriminador adapta su capacidad para identificar muestras falsas, el generador aprende a mejorar su capacidad para generar muestras convincentes que lo engañen. A medida que avanza el entrenamiento, se alcanza un equilibrio en el que ninguna de las redes puede seguir mejorando. En este punto, el generador se vuelve capaz de producir instancias de datos que son prácticamente indistinguibles de las muestras reales.

Para entender exactamente cómo funcionan las GAN, veamos un ejemplo ilustrativo.

1.2.2 Un ejemplo ilustrativo: Generación de caras de anime

Imagínate esto: eres un apasionado del anime y estás en una emocionante búsqueda para crear tus propias caras de anime usando una poderosa herramienta conocida como GAN convolucional profunda (o DCGAN para abreviar; no te preocupes, profundizaremos en esto en el capítulo 4).

Si observa la parte superior central de la figura 1.2, verá una imagen que dice "Imagen real". Usaremos 63 632 imágenes coloridas de caras de anime como conjunto de datos de entrenamiento. Y si pasa a la figura 1.3, verá 32 ejemplos de nuestro conjunto de entrenamiento. Estas imágenes especiales desempeñan un papel crucial, ya que constituyen la mitad de las entradas de nuestra red discriminadora.



Figura 1.3 Ejemplos del conjunto de datos de entrenamiento de caras de anime

La parte izquierda de la figura 1.2 es la red generadora. Para generar imágenes diferentes en cada ocasión, el generador toma como entrada un vector Z del espacio latente. Podríamos considerar este vector como una "descripción de la tarea". Durante el entrenamiento, dibujamos diferentes vectores Z del espacio latente, por lo que la red genera imágenes diferentes en cada ocasión. Estas imágenes falsas constituyen la otra mitad de las entradas de la red discriminadora.

NOTA: Al modificar los valores del vector Z, se generan diferentes resultados. En el capítulo 5, aprenderá a seleccionar el vector Z para generar imágenes con ciertas características (p. ej., rasgos masculinos o femeninos).

Pero aquí está el giro: antes de enseñarles a nuestras dos redes el arte de la creación y la detección, ¡las imágenes producidas por el generador son, bueno, un galimatías! No se parecen en nada a las caras realistas de anime que se ven en la figura 1.3. De hecho, no parecen más que estática en una pantalla de televisión (lo comprobarás de primera mano en el capítulo 4).

Entrenamos el modelo para múltiples iteraciones. En cada iteración, presentamos al discriminador un grupo de imágenes creadas por el generador, junto con un grupo de imágenes de caras de anime de nuestro conjunto de entrenamiento. Le pedimos al discriminador que prediga si cada imagen es creada por el generador (falsa) o del conjunto de entrenamiento (real).

Quizás te preguntes: ¿Cómo aprenden el discriminador y el generador durante cada iteración de entrenamiento? Una vez realizadas las predicciones, el discriminador no se queda de brazos cruzados; aprende de sus errores de predicción para cada imagen. Con este nuevo conocimiento, afina sus parámetros, adaptándose para realizar mejores predicciones en la siguiente ronda. El generador tampoco se queda inactivo. Toma notas de su proceso de generación de imágenes y de los resultados de las predicciones del discriminador. Con este conocimiento, ajusta sus propios parámetros de red, buscando crear imágenes cada vez más realistas en la siguiente iteración. ¿El objetivo? Reducir la probabilidad de que el discriminador detecte sus errores.

A medida que avanzamos en estas iteraciones, se produce una transformación notable. La red generadora evoluciona, produciendo rostros de anime cada vez más realistas, similares a los de nuestra colección de entrenamiento. Mientras tanto, la red discriminadora perfecciona sus habilidades, convirtiéndose en una experta detective en la detección de falsificaciones. Es una fascinante danza entre creación y detección.

Poco a poco, llega un momento mágico. Se alcanza un equilibrio perfecto. Las imágenes creadas por el generador se vuelven tan asombrosamente reales que son indistinguibles de las caras de anime auténticas de nuestros archivos de entrenamiento. En este punto, el discriminador está tan confundido que asigna un 50 % de probabilidad de autenticidad a cada imagen, ya sea de nuestro conjunto de entrenamiento o creada por el generador.

Por último, observe algunos ejemplos del trabajo artístico del generador, como se muestra en la figura 1.4: parecen indistinguibles de los de nuestro conjunto de entrenamiento.



Figura 1.4 Imágenes de rostros de anime generadas por el generador entrenado en DCGAN

1.2.3 ¿Por qué debería importarle las GAN?

Las GAN son fáciles de implementar y versátiles: aprenderá a generar formas geométricas, patrones intrincados, imágenes de alta resolución y música con sonido realista solo en este libro.

El uso práctico de las GAN no se limita a la generación de datos realistas. Las GAN también pueden traducir atributos de un dominio de imagen a otro. Como verá en el capítulo 6, puede entrenar un CycleGAN (un tipo de modelo generativo de la familia GAN) para convertir cabello rubio en cabello negro en imágenes de rostros humanos. El mismo modelo entrenado también puede convertir cabello negro en rubio. La Figura 1.5 muestra cuatro filas de imágenes. La primera fila corresponde a las imágenes originales con cabello rubio. El CycleGAN entrena las convierte en imágenes con cabello negro (segunda fila). Las dos últimas filas corresponden a las imágenes originales con cabello negro y la imagen convertida con cabello rubio, respectivamente.

Original images with blond hair:



Fake images with black hair:



Original images with black hair:



Fake images with blonde hair:



Figura 1.5 Cambio de color de cabello con CycleGAN. Si alimentamos imágenes de cabello rubio (primera fila) a un modelo CycleGAN entrenado, este convierte el cabello rubio en cabello negro en estas imágenes (segunda fila). El mismo modelo entrenado también puede convertir el cabello negro (tercera fila) en cabello rubio (fila inferior).

Piensa en todas las increíbles habilidades que adquirirás al entrenar GAN: ¡no solo son geniales, sino también súper prácticas! Supongamos que gestionas una tienda de ropa en línea con una estrategia de "fabricación por encargo" (que permite a los usuarios personalizar sus compras antes de la fabricación). Tu sitio web muestra una gran cantidad de diseños únicos para que los clientes elijan, pero aquí está la trampa: solo fabricas la ropa cuando alguien realiza un pedido. Crear imágenes de alta calidad de estas prendas puede ser bastante costoso, ya que tienes que producir las prendas y luego fotografiarlas.

¡GAN al rescate! No necesitas una colección enorme de prendas de vestir fabricadas y sus imágenes; en su lugar, puedes usar algo como CycleGAN para transformar las características de un conjunto de imágenes en otro, creando una gama completamente nueva de estilos. Esta es solo una forma ingeniosa de usar GAN. Las posibilidades son infinitas, ya que estos mode-

los son sumamente versátiles y pueden manejar todo tipo de datos, lo que los convierte en una innovación para aplicaciones prácticas.

1.3 Transformadores

Los Transformers son redes neuronales profundas que destacan en problemas de predicción secuencia a secuencia, como tomar una oración de entrada y predecir las palabras más probables que aparecerán a continuación. Esta sección presenta la innovación clave en Transformers: el mecanismo de autoatención. A continuación, analizaremos la arquitectura de Transformers y sus diferentes tipos. Finalmente, analizaremos algunos desarrollos recientes en Transformers, como los modelos multimodales (Transformers cuyas entradas incluyen no solo texto, sino también otros tipos de datos como audio e imágenes) y los LLM preentrenados (modelos entrenados con grandes cantidades de datos textuales que pueden realizar diversas tareas posteriores).

Antes de que un grupo de investigadores de Google inventara la arquitectura Transformer en 2017 (Vaswani et al., “Attention Is All You Need”, <https://arxiv.org/abs/1706.03762>), el procesamiento del lenguaje natural (PLN) y otras tareas de predicción secuencia a secuencia eran gestionadas principalmente por redes neuronales regresivas (RNR). Sin embargo, las RNN tienen dificultades para retener información sobre los elementos anteriores de una secuencia, lo que dificulta su capacidad para capturar dependencias a largo plazo. Incluso variantes avanzadas de RNN, como las redes de memoria a corto plazo (LSTM), que pueden gestionar dependencias de mayor alcance, se quedan cortas cuando se trata de dependencias de alcance extremadamente largo.

Más importante aún, las RNN (incluidas las LSTM) procesan las entradas secuencialmente, lo que significa que estos modelos procesan un elemento a la vez, en secuencia, en lugar de analizar la secuencia completa simultáneamente. El hecho de que las RNN realicen el cálculo a lo largo de las posiciones de los símbolos de las secuencias de entrada y salida impide el entrenamiento en paralelo, lo que lo ralentiza. Esto, a su vez, imposibilita el entrenamiento de los modelos con conjuntos de datos de gran tamaño.

La innovación clave de los Transformers es el mecanismo de autoatención, que destaca por capturar dependencias a largo plazo en una secuencia.

Además, dado que las entradas no se procesan secuencialmente en el modelo, los Transformers pueden entrenarse en paralelo, lo que reduce considerablemente el tiempo de entrenamiento. Aún más importante, el entrenamiento en paralelo permite entrenar a los Transformers con grandes cantidades de datos, lo que hace que los LLM sean inteligentes y estén bien informados (basándose en su capacidad para procesar y generar texto similar al humano, comprender el contexto y realizar diversas tareas lingüísticas). Esto ha impulsado el auge de LLM como ChatGPT y el reciente auge de la IA.

1.3.1 El mecanismo de atención

El mecanismo de atención asigna ponderaciones según la relación de un elemento con todos los elementos de una secuencia (incluido el propio elemento). Cuanto mayor sea la ponderación, mayor será la relación entre ambos elementos. Estas ponderaciones se aprenden a partir de grandes conjuntos de datos de entrenamiento durante el proceso de entrenamiento. Por lo tanto, un LLM entrenado como ChatGPT puede determinar la relación entre dos palabras cualesquiera en una oración, dando así sentido al lenguaje humano.

Quizás te preguntes: ¿cómo asigna el mecanismo de atención puntuaciones a los elementos de una secuencia para capturar las dependencias a largo plazo? Los pesos de la atención se calculan pasando primero las entradas a través de tres capas de red neuronal para obtener la consulta Q , la clave K y el valor V (que explicaremos en detalle en el capítulo 9). El método de usar la consulta, la clave y el valor para calcular la atención proviene de los sistemas de recuperación. Por ejemplo, puedes ir a una biblioteca pública a buscar un libro. Puedes escribir, por ejemplo, "aprendizaje automático en finanzas" en el motor de búsqueda de la biblioteca. En este caso, la consulta Q es "aprendizaje automático en finanzas". Las claves K son los títulos de los libros, las descripciones de los libros, etc. El sistema de recuperación de la biblioteca recomendará una lista de libros (valores V) en función de las similitudes entre la consulta y las claves. Naturalmente, los libros con las frases “aprendizaje automático” o “finanzas”, o ambas, en los títulos o descripciones aparecen en la parte superior, mientras que los libros sin ninguna frase en el título o la descripción aparecerán en la parte inferior de la lista porque a estos libros se les asignará un puntaje de coincidencia bajo.

En los capítulos 9 y 10, aprenderá los detalles del mecanismo de atención; mejor aún, implementará el mecanismo de atención desde cero para construir y entrenar un Transformer para traducir con éxito del inglés al francés.

1.3.2 La arquitectura del Transformer

Los transformadores se propusieron por primera vez al diseñar modelos para la traducción automática de lenguajes (p. ej., del inglés al alemán o del inglés al francés). La figura 1.6 muestra un diagrama de la arquitectura del transformador. El lado izquierdo es el codificador y el lado derecho, el decodificador. En los capítulos 9 y 10, aprenderá a construir un transformador desde cero para entrenar el modelo para traducir del inglés al francés. Posteriormente, explicaremos la figura 1.6 con más detalle.

El codificador del Transformador aprende el significado de la secuencia de entrada (p. ej., la frase en inglés "¿Cómo estás?") y lo convierte en vectores que representan dicho significado antes de pasarlos al decodificador. El decodificador construye la salida (p. ej., la traducción al francés de una frase en inglés) prediciendo palabra por palabra, basándose en las palabras anteriores de la secuencia y la salida del codificador. El modelo entrenado puede traducir frases comunes del inglés al francés.

Hay tres tipos de Transformers: Transformers de solo codificador, Transformers de solo decodificador y Transformers de codificador-decodificador. Un Transformer de solo codificador no tiene decodificador y es capaz de convertir una secuencia en una representación abstracta para varias tareas posteriores, como el análisis de sentimientos, el reconocimiento de entidades con nombre y la generación de texto. Por ejemplo, BERT es un Transformer de solo codificador. Un Transformer de solo decodificador tiene solo un decodificador pero no un codificador, y es muy adecuado para la generación de texto, el modelado de lenguaje y la escritura creativa. GPT-2 (el predecesor de ChatGPT) y ChatGPT son Transformers de solo decodificador. En el capítulo 11, aprenderá a crear GPT-2 desde cero y luego extraer los pesos del modelo entrenado de Hugging Face (una comunidad de IA que aloja y colabora en modelos de ML, conjuntos de datos y aplicaciones). Cargará los pesos en su modelo GPT-2 y comenzará a generar texto coherente.

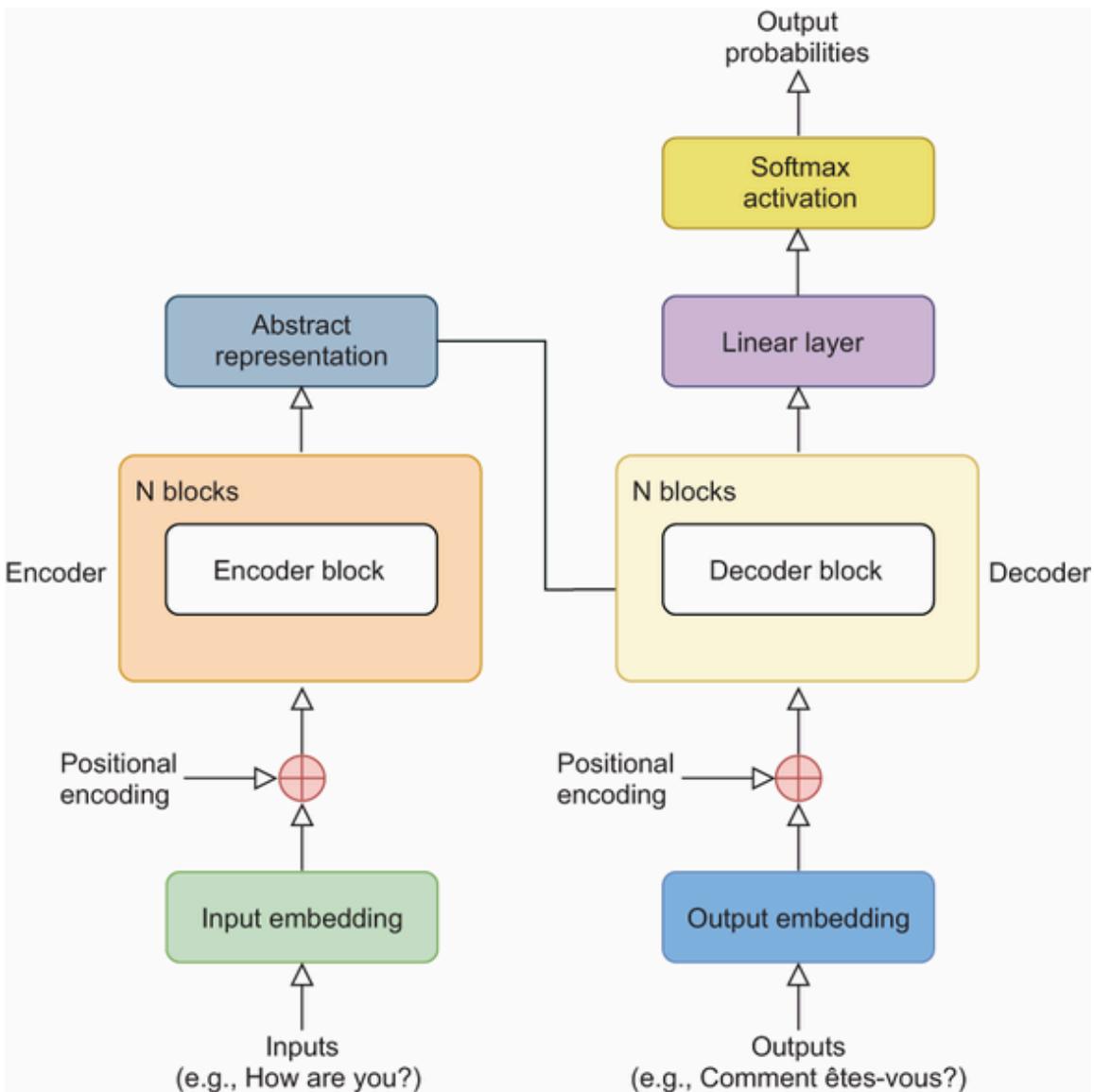


Figura 1.6 Arquitectura del Transformer. El codificador del Transformer (lado izquierdo del diagrama) aprende el significado de la secuencia de entrada (p. ej., la frase en inglés "¿Cómo estás?") y la convierte en una representación abstracta que captura su significado antes de pasarla al decodificador (lado derecho del diagrama). El decodificador construye la salida (p. ej., la traducción al francés de la frase en inglés) prediciendo palabra por palabra, basándose en las palabras anteriores de la secuencia y la representación abstracta del codificador.

Los transformadores codificador-decodificador son necesarios para tareas complejas, como los modelos multimodales que gestionan la generación de texto a imagen o el reconocimiento de voz. Los transformadores codificador-decodificador combinan las ventajas de los codificadores y decodificadores. Los codificadores son eficientes en el procesamiento y la comprensión de los datos de entrada, mientras que los decodificadores destacan en la generación de salidas. Esta combinación permite al modelo comprender eficazmente entradas complejas (como texto o voz) y generar salidas complejas (como imágenes o texto transcritos).

1.3.3 Transformadores multimodales y LLM preentrenados

Los recientes avances en IA generativa han dado lugar a diversos modelos multimodales: transformadores que pueden utilizar no solo texto, sino también otros tipos de datos, como audio e imágenes, como entradas. Los transformadores de texto a imagen son un ejemplo. DALL-E 2, Imagen y Stable Diffusion son modelos de texto a imagen y han atraído mucha atención mediática gracias a su capacidad para generar imágenes de alta resolución a partir de indicaciones textuales. Los transformadores de texto a imagen incorporan los principios de los modelos de difusión, que implican una serie de transformaciones para aumentar gradualmente la complejidad de los datos. Por lo tanto, primero debemos comprender los modelos de difusión antes de analizar los transformadores de texto a imagen.

Imagina que quieras generar imágenes de flores de alta resolución mediante un modelo basado en difusión. Primero, obtendrás un conjunto de entrenamiento de imágenes de flores de alta calidad. A continuación, le pedirás al modelo que añada ruido gradualmente a las imágenes (el llamado proceso de difusión) hasta que se conviertan en ruido completamente aleatorio. Después, entrenarás al modelo para que elimine progresivamente el ruido de estas imágenes ruidosas y genere nuevas muestras de datos. El proceso de difusión se ilustra en la figura 1.7. La columna izquierda contiene cuatro imágenes originales de flores. A medida que nos desplazamos hacia la derecha, se añade ruido a las imágenes en cada paso, hasta que en la columna derecha, las cuatro imágenes son ruido aleatorio puro.



Figura 1.7. El modelo de difusión añade cada vez más ruido a las imágenes y aprende a reconstruirlas. La columna izquierda contiene cuatro imágenes originales de flores. A medida que nos desplazamos hacia la derecha, se añade ruido a las imágenes en cada paso de tiempo, hasta que en la columna derecha, las cuatro imágenes son puro ruido aleatorio. A continuación, utilizamos estas imágenes ruidosas y generemos nuevas muestras de datos.

Quizás se pregunte: ¿Cómo se relacionan los transformadores de texto a imagen con los modelos de difusión? Los transformadores de texto a imagen toman una instrucción de texto como entrada y generan imágenes que corresponden a esa descripción textual. La instrucción de texto actúa como una forma de condicionamiento, y el modelo utiliza una serie de capas de red neuronal para transformar esa descripción textual en una imagen. Al igual que los modelos de difusión, los transformadores de texto a imagen utilizan una arquitectura jerárquica con múltiples capas, cada una de las cuales añade progresivamente más detalle a la imagen generada. El concepto central de refinar iterativamente el resultado es similar tanto en los modelos de difusión como en los transformadores de texto a imagen, como explicaremos en el capítulo 15.

Los modelos de difusión se han vuelto más populares gracias a su capacidad para proporcionar un entrenamiento estable y generar imágenes de alta calidad, y han superado a otros modelos generativos como las GAN y los autocodificadores variacionales. En el capítulo 15, aprenderá a entrenar un modelo de difusión simple utilizando el conjunto de datos Oxford Flower. También aprenderá la idea básica de los transformadores multi-

modales y escribirá un programa en Python para solicitar a DALL-E 2 de OpenAI que genere imágenes mediante una instrucción de texto. Por ejemplo, al introducir "un astronauta con traje espacial montando un unicornio" como instrucción, DALL-E 2 generó la imagen que se muestra en la figura 1.8.



Figura 1.8 Imagen generada por DALL-E 2 con el texto “un astronauta en un traje espacial montando un unicornio”

En el capítulo 16, aprenderá a acceder a LLM preentrenados, como ChatGPT, GPT4 y DALL-E 2. Estos modelos se entrenaon con grandes cantidades de datos textuales y han adquirido conocimientos generales a partir de ellos. Por lo tanto, pueden realizar diversas tareas posteriores, como la generación de texto, el análisis de sentimientos, la respuesta a preguntas y el reconocimiento de entidades con nombre. Dado que los LLM preentrenados se entrenaron con información hace unos meses, no pueden proporcionar información sobre eventos y desarrollos de los últimos uno o dos meses, y mucho menos información en tiempo real como las condiciones meteorológicas, el estado de los vuelos o los precios de las acciones. Utilizaremos LangChain (una biblioteca de Python diseñada para crear aplicaciones con LLM, que proporciona herramientas para la gestión de solicitudes, el encadenamiento de LLM y el análisis de resultados) para encadenar LLM con las API de Wolfram Alpha y Wikipedia y crear un asistente personal inteligente.

1.4 ¿Por qué construir modelos generativos desde cero?

El objetivo de este libro es mostrarte cómo construir y entrenar todos los modelos generativos desde cero. De esta manera, comprenderás a fondo su funcionamiento interno y podrás aprovecharlos mejor. Crear algo desde cero es la mejor manera de comprenderlo. Lograrás este objetivo con las GAN: todos los modelos, incluyendo DCGAN y CycleGAN, se construyen desde cero y se entrena con datos bien seleccionados de dominio público.

Para Transformers, construirás y entrenarás todos los modelos desde cero excepto los LLM. Esta excepción se debe a la gran cantidad de datos y a las instalaciones de supercomputación necesarias para entrenar ciertos LLM. Sin embargo, harás un progreso serio en esta dirección. Específicamente, implementarás en los capítulos 9 y 10 el artículo original e innovador de 2017 "Attention Is All You Need" línea por línea con la traducción del inglés al francés como ejemplo (el mismo Transformer se puede entrenar en otros conjuntos de datos como traducciones del chino al inglés o del inglés al alemán). También construirás un Transformer de solo decodificador de tamaño pequeño y lo entrenarás usando varias de las novelas de Ernest Hemingway, incluyendo *El viejo y el mar*. El modelo entrenado puede generar texto en estilo Hemingway. ChatGPT y GPT-4 son demasiado grandes y complicados para construirlos y entrenarlos desde cero para nuestros propósitos, pero echarás un vistazo a su predecesor, GPT-2, y aprenderás a construirlo desde cero. También extraerás los pesos entrenados de Hugging Face y los cargarás en el modelo GPT-2 que construiste y comenzarás a generar texto realista que puede pasar como escrito por humanos.

En este sentido, el libro adopta un enfoque más fundamental que la mayoría. En lugar de tratar los modelos de IA generativa como una caja negra, los lectores tienen la oportunidad de analizar en profundidad su funcionamiento interno. El objetivo es que los lectores adquieran una comprensión más profunda de los modelos generativos. Esto, a su vez, puede ayudarles a desarrollar una IA generativa mejor y más responsable por las siguientes razones.

En primer lugar, comprender a fondo la arquitectura de los modelos generativos ayuda a los lectores a aprovechar mejor estos modelos. Por ejemplo, en el capítulo 5, aprenderá a seleccionar características en imágenes

generadas, como rasgos masculinos o femeninos, y si se usan o no gafas. Al construir una GAN condicional desde cero, comprenderá que ciertas características de las imágenes generadas están determinadas por el vector de ruido aleatorio, Z, en el espacio latente. Por lo tanto, puede elegir diferentes valores de Z como entradas para el modelo entrenado para generar las características deseadas (como rasgos masculinos o femeninos). Este tipo de selección de atributos es difícil de realizar sin comprender el diseño del modelo.

Para los Transformers, conocer la arquitectura (y lo que hacen los codificadores y decodificadores) le brinda la capacidad de crear y entrenar Transformers para generar los tipos de contenido que necesita. Entretenimiento que te interese (por ejemplo, novelas al estilo de Jane Austin o música al estilo de Mozart). Esta comprensión también te ayuda con los LLM preentrenados. Por ejemplo, si bien es difícil entrenar GPT-2 desde cero con sus 1500 millones de parámetros, puedes añadir una capa adicional al modelo y ajustarlo para otras tareas posteriores, como la clasificación de texto, el análisis de sentimientos y la respuesta a preguntas.

En segundo lugar, una comprensión profunda de la IA generativa ayuda a los lectores a tener una evaluación imparcial de los peligros de la IA. Si bien los extraordinarios poderes de la IA generativa nos han beneficiado en nuestra vida diaria y en el trabajo, también tienen el potencial de causar un gran daño. Elon Musk llegó a decir que "existe la posibilidad de que salga mal y destruya a la humanidad" (véase el artículo de Julia Mueller en *The Hill*, 2023, "Musk: There's a Chance AI 'Goes Wrong and Destroys Humanity'", <https://mng.bz/Aaxz>). Cada vez más personas en el ámbito académico y en la industria tecnológica están preocupadas por los peligros que plantea la IA en general y la IA generativa en particular. La IA generativa, especialmente los LLM, puede tener consecuencias imprevistas, como muchos pionerosLos profesionales de la tecnología han advertido (véase, por ejemplo, Stuart Russell, 2023, "How to Stop Runaway AI", <https://mng.bz/ZVzP>). No es una coincidencia que solo cinco meses después del lanzamiento de ChatGPT, muchos expertos y emprendedores de la industria tecnológica, incluidos Steve Wozniak, Tristan Harris, Yoshua Bengio y Sam Altman, firmaran una carta abierta pidiendo una pausa en el entrenamiento de cualquier sistema de IA que sea más potente que GPT-4 durante al menos seis meses (véase el artículo de Connie Loizos en *TechCrunch*, "1,100+ Notable Signatories Just Signed an Open Letter Asking

'All AI Labs to Pause Immediately for at Least 6 Months'',
<https://mng.bz/RNEK>). Una comprensión profunda de la arquitectura de los modelos generativos nos ayuda a proporcionar una evaluación profunda e imparcial de los beneficios y los peligros potenciales de la IA.

Resumen

- La IA generativa es un tipo de tecnología con la capacidad de producir diversas formas de contenido nuevo, incluidos textos, imágenes, código, música, audio y vídeo.
- Los modelos discriminativos se especializan en asignar etiquetas, mientras que los modelos generativos generan nuevas instancias de datos.
- PyTorch, con sus gráficos computacionales dinámicos y la capacidad de entrenamiento con GPU, es ideal para el aprendizaje profundo y el modelado generativo.
- Las GAN son un tipo de método de modelado generativo que consta de dos redes neuronales: un generador y un discriminador. El objetivo del generador es crear muestras de datos realistas para maximizar la probabilidad de que el discriminador las considere reales. El objetivo del discriminador es identificar correctamente las muestras falsas de las reales.
- Los transformadores son redes neuronales profundas que utilizan el mecanismo de atención para identificar dependencias a largo plazo entre los elementos de una secuencia. El transformador original cuenta con un codificador y un decodificador. Por ejemplo, al traducir del inglés al francés, el codificador convierte la oración en inglés en una representación abstracta antes de pasarla al decodificador. Este genera la traducción al francés palabra por palabra, basándose en la salida del codificador y las palabras generadas previamente.

2. Aprendizaje profundo con PyTorch

Este capítulo cubre

- Tensores de PyTorch y operaciones básicas
- Preparación de datos para el aprendizaje profundo en PyTorch
- Construcción y entrenamiento de redes neuronales profundas con PyTorch
- Realización de binarios y clasificaciones multicategoría con aprendizaje profundo
- Creación de un conjunto de validación para decidir los puntos de parada del entrenamiento

En este libro, utilizaremos redes neuronales profundas para generar una amplia gama de contenido, incluyendo texto, imágenes, formas, música y más. Asumo que ya tienes conocimientos básicos de aprendizaje automático (ML) y, en particular, de redes neuronales artificiales. En este capítulo, te refrescaré la memoria sobre conceptos esenciales como funciones de pérdida, funciones de activación, optimizadores y tasas de aprendizaje, cruciales para el desarrollo y entrenamiento de redes neuronales profundas. Si encuentras alguna laguna en tu comprensión de estos temas, te recomiendo encarecidamente que la abordes antes de continuar con los proyectos de este libro. El Apéndice B ofrece un resumen de las habilidades y conceptos básicos necesarios, incluyendo la arquitectura y el entrenamiento de redes neuronales artificiales.

NOTA: Hay muchos libros excelentes sobre aprendizaje automático disponibles. Algunos ejemplos son "*Aprendizaje automático práctico con Scikit-Learn, Keras y TensorFlow*" (2019, O'Reilly) y "*Aprendizaje automático animado*" (2023, CRC Press). Ambos libros utilizan TensorFlow para crear redes neuronales. Si prefiere un libro que utilice PyTorch, le recomiendo "*Aprendizaje profundo con PyTorch*" (2020, Manning Publications).

Los modelos de IA generativa se enfrentan frecuentemente a tareas de clasificación binaria o multicategoría. Por ejemplo, en las redes generativas antagónicas (GAN), el discriminador desempeña la función esencial de un

clasificador binario, con el propósito de distinguir entre las muestras falsas creadas por el generador y las muestras reales del conjunto de entrenamiento. De igual manera, en el contexto de los modelos de generación de texto, ya sea en redes neuronales recurrentes o en Transformers, el objetivo principal es predecir el carácter o la palabra subsiguiente entre un amplio abanico de posibilidades (en esencia, una tarea de clasificación multicategoría).

En este capítulo, aprenderá a usar PyTorch para crear redes neuronales profundas para realizar clasificaciones binarias y multicategoría para que pueda familiarizarse con las tareas de aprendizaje profundo y clasificación.

En concreto, participarás en un proyecto integral de aprendizaje profundo en PyTorch para clasificar imágenes en escala de grises de prendas de vestir en diferentes categorías, como abrigos, bolsos, zapatillas, camisas, etc. El objetivo es prepararte para la creación de redes neuronales profundas, capaces de realizar tareas de clasificación binaria y multicategoría en PyTorch. Esto, a su vez, te preparará para los próximos capítulos, donde utilizarás redes neuronales profundas en PyTorch para crear diversos modelos generativos.

Para entrenar modelos de IA generativa, utilizamos diversos formatos de datos, como texto sin procesar, archivos de audio, píxeles de imagen y matrices de números. Las redes neuronales profundas creadas en PyTorch no pueden tomar estos datos directamente como entrada. En su lugar, primero debemos convertirlos a un formato que las redes neuronales comprendan y acepten. En concreto, convertirás diversos tipos de datos sin procesar en tensores de PyTorch (estructuras de datos fundamentales para representar y manipular datos) antes de alimentarlos a los modelos de IA generativa. Por lo tanto, en este capítulo, también aprenderás los fundamentos de los tipos de datos, cómo crear diversos tipos de tensores de PyTorch y cómo usarlos en el aprendizaje profundo.

Saber cómo realizar tareas de clasificación tiene muchas aplicaciones prácticas en nuestra sociedad. Las clasificaciones se utilizan ampliamente en la atención médica con fines diagnósticos, como identificar si un paciente padece una enfermedad específica (por ejemplo, si da positivo o negativo para un cáncer específico según los resultados de imágenes o pruebas mé-

dicas). Desempeñan un papel vital en muchas tareas empresariales (recomendaciones de acciones, detección de fraudes con tarjetas de crédito, etc.). Las tareas de clasificación también son parte integral de muchos sistemas y servicios que utilizamos a diario, como la detección de spam y el reconocimiento facial.

2.1 Tipos de datos en PyTorch

En este libro utilizaremos conjuntos de datos de una amplia gama de fuentes y formatos, y el primer paso en el aprendizaje profundo es transformar las entradas en matrices de números.

En esta sección, aprenderá cómo PyTorch convierte diferentes formatos de datos en estructuras algebraicas conocidas como *tensores*. Los tensores se pueden representar como matrices multidimensionales de números, similares a las matrices de NumPy, pero con varias diferencias clave, entre ellas la capacidad de entrenamiento acelerado por GPU. Existen diferentes tipos de tensores según su uso final, y aprenderá a crearlos y cuándo usar cada uno. En esta sección, analizaremos la estructura de datos en PyTorch utilizando las alturas de los 46 presidentes de EE. UU. como ejemplo.

Consulte las instrucciones del apéndice A para crear un entorno virtual e instalar PyTorch y Jupyter Notebook en su computadora. Abra la aplicación Jupyter Notebook dentro del entorno virtual y ejecute la siguiente línea de código en una nueva celda:

```
!pip install matplotlib
```

Este comando instalará la biblioteca Matplotlib en su computadora, lo que le permitirá trazar imágenes en Python.

2.1.1 Creación de tensores de PyTorch

Al entrenar redes neuronales profundas, alimentamos los modelos con matrices de números como entrada. Dependiendo de lo que un modelo generativo intente crear, estos números tienen diferentes tipos. Por ejemplo, al generar imágenes, las entradas son píxeles sin procesar en forma de números enteros entre 0 y 255, pero los convertiremos a números de punto flotante entre -1 y 1; al generar texto, existe un “vocabulario” similar a un

diccionario, y la entrada es una secuencia de números enteros que indica a qué entrada del diccionario corresponde la palabra.

NOTA: El código de este capítulo, así como de otros capítulos de este libro, está disponible en el repositorio de GitHub del libro:

<https://github.com/markhliu/DGAI>.

Imagina que quieres usar PyTorch para calcular la altura promedio de los 46 presidentes de EE. UU. Primero, podemos recopilar las alturas de los 46 presidentes en centímetros y almacenarlas en una lista de Python:

```
heights = [189, 170, 189, 163, 183, 171, 185,
           168, 173, 183, 173, 173, 175, 178,
           183, 193, 178, 173, 174, 183, 183,
           180, 168, 180, 170, 178, 182, 180,
           183, 178, 182, 188, 175, 179, 183,
           193, 182, 183, 177, 185, 188, 188,
           182, 185, 191, 183]
```

Los números están en orden cronológico: el primer valor de la lista, 189, indica que el primer presidente de Estados Unidos, George Washington, media 189 centímetros. El último valor muestra que la altura de Joe Biden es de 183 centímetros. Podemos convertir una lista de Python en un tensor de PyTorch usando el método `tensor()` de PyTorch:

```
import torch
heights_tensor = torch.tensor(heights,               ①
                             dtype=torch.float64)      ②
```

① Convierte una lista de Python en un tensor de PyTorch

② Especifica el tipo de datos en el tensor de PyTorch

Especificamos el tipo de dato mediante el argumento `dtype` del método `tensor()`. El tipo de dato predeterminado en los tensores de PyTorch es `float32`, un número de punto flotante de 32 bits. En la celda de código anterior, convertimos el tipo de dato a `float64`, números de punto flotante de doble precisión. `float64` proporciona resultados más precisos que `float32`, pero su cálculo tarda más. Existe un equilibrio entre la precisión

y el coste computacional. El tipo de dato a utilizar depende de la tarea en cuestión.

La Tabla 2.1 enumera los diferentes tipos de datos y los tipos de tensor correspondientes de PyTorch. Estos incluyen números enteros y de punto flotante con diferentes precisiones. Los números enteros también pueden tener o no signo.

Tabla 2.1 Tipos de datos y tensores en PyTorch

Tipo de tensor de PyTorch	argumento dtype en tensor()	Tipo de datos
FloatTensor	<code>torch.float32</code> or <code>torch.float</code>	punto flotante de 32 bits
HalfTensor	<code>torch.float16</code> or <code>torch.half</code>	punto flotante de 16 bits
DoubleTensor	<code>torch.float64</code> or <code>torch.double</code>	punto flotante de 64 bits
CharTensor	<code>torch.int8</code>	entero de 8 bits (con signo)
ByteTensor	<code>torch.uint8</code>	entero de 8 bits (sin signo)
ShortTensor	<code>torch.int16</code> or <code>torch.short</code>	entero de 16 bits (con signo)
IntTensor	<code>torch.int32</code> or <code>torch.int</code>	entero de 32 bits (con signo)
LongTensor	<code>torch.int64</code> or <code>torch.long</code>	entero de 64 bits (con signo)

Puedes crear un tensor con un tipo de dato específico de dos maneras. La primera es usar la clase PyTorch, como se especifica en la primera columna de la tabla 2.1. La segunda es usar el método `torch.tensor()` y especificar el tipo de dato mediante el argumento `dtype` (el valor del argumento se muestra en la segunda columna de la tabla 2.1). Por ejemplo, para convertir la lista de Python `[1, 2, 3]` en un tensor de PyTorch con enteros de 32 bits, puedes usar los dos métodos de la siguiente lista.

Listado 2.1 Dos formas de especificar tipos de tensores

```
t1=torch.IntTensor([1, 2, 3])      ①  
t2=torch.tensor([1, 2, 3],  
                dtype=torch.int)    ②  
print(t1)  
print(t2)
```

- ① Utiliza `torch.IntTensor()` para especificar el tipo de tensor
- ② Utiliza `dtype=torch.int` para especificar el tipo de tensor

Esto conduce al siguiente resultado:

```
tensor([1, 2, 3], dtype=torch.int32)  
tensor([1, 2, 3], dtype=torch.int32)
```

Ejercicio 2.1

Utilice dos métodos diferentes para convertir la lista de Python `[5, 8, 10]` en un tensor de PyTorch con números de punto flotante de 64 bits. Consulte la tercera fila de la tabla 2.1 para esta pregunta.

Muchas veces, es necesario crear un tensor de PyTorch con valores 0 en todas partes. Por ejemplo, en las GAN, creamos un tensor de ceros como etiquetas para las muestras falsas, como veremos en el capítulo 3. El método `zeros()` de PyTorch genera un tensor de ceros con una forma específica. En PyTorch, un tensor es un array n-dimensional, y su forma es una tupla que representa el tamaño en cada una de sus dimensiones. Las siguientes líneas de código generan un tensor de ceros con dos filas y tres columnas:

```
tensor1 = torch.zeros(2, 3)
print(tensor1)
```

El resultado es el siguiente:

```
tensor([[0., 0., 0.],
       [0., 0., 0.]])
```

El tensor tiene la forma (2, 3), lo que significa que es un array bidimensional; hay dos elementos en la primera dimensión y tres en la segunda. En este caso, no se especificó el tipo de dato, y la salida tiene el tipo de dato predeterminado `float32`.

De vez en cuando, es necesario crear un tensor de PyTorch con valores 1 en todas partes. Por ejemplo, en las GAN, creamos un tensor de unos como etiquetas para muestras reales. Aquí usamos el método `ones()` para crear un tensor 3D con valores 1 en todas partes:

```
tensor2 = torch.ones(1,4,5)
print(tensor2)
```

La salida es

```
tensor([[[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]]])
```

Hemos generado un tensor 3D de PyTorch. Su forma es (1, 4, 5).

Ejercicio 2.2

Crea un tensor 3D de PyTorch con valores 0. Define la forma del tensor (2, 3, 4).

También puedes usar una matriz NumPy en lugar de una lista de Python en el constructor del tensor:

```
import numpy as np

nparr=np.array(range(10))
pt_tensor=torch.tensor(nparr, dtype=torch.int)
print(pt_tensor)
```

La salida es

```
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=torch.int32)
```

2.1.2 Indexar y segmentar tensores de PyTorch

Usamos corchetes ([]) para indexar y segmentar los tensores de PyTorch, como hacemos con las listas de Python. La indexación y la segmentación nos permiten operar con uno o más elementos de un tensor, en lugar de con todos. Para continuar con nuestro ejemplo de las alturas de los 46 presidentes de EE. UU., si queremos calcular la altura del tercer presidente, Thomas Jefferson, podemos hacer lo siguiente:

```
height = heights_tensor[2]
print(height)
```

Esto conduce a una salida de

```
tensor(189., dtype=torch.float64)
```

El resultado muestra que la altura de Thomas Jefferson era de 189 centímetros.

Podemos usar la indexación negativa para contar desde la parte posterior del tensor. Por ejemplo, para hallar la estatura de Donald Trump, el penúltimo presidente de la lista, usamos el índice -2:

```
height = heights_tensor[-2]
print(height)
```

La salida es

```
tensor(191., dtype=torch.float64)
```

El resultado muestra que la altura de Trump es de 191 centímetros.

¿Qué sucede si queremos conocer las alturas de cinco presidentes recientes en el tensor `heights_tensor`? Podemos obtener una porción del tensor:

```
five_heights = heights_tensor[-5:]
print(five_heights)
```

Los dos puntos (`:`) se utilizan para separar el índice inicial del final. Si no se proporciona un índice inicial, el valor predeterminado es 0; si no se proporciona un índice final, se incluye el último elemento del tensor (como hicimos en la celda de código anterior). La indexación negativa significa que se cuenta desde atrás. El resultado es

```
tensor([188., 182., 185., 191., 183.], dtype=torch.float64)
```

Los resultados muestran que los cinco presidentes recientes del tensor (Clinton, Bush, Obama, Trump y Biden) miden 188, 182, 185, 191 y 183 centímetros, respectivamente.

Ejercicio 2.3

Utilice el corte para obtener las alturas de los primeros cinco presidentes de EE. UU. en el tensor `heights_tensor`.

2.1.3 Formas tensoriales de PyTorch

Los tensores de PyTorch tienen un atributo llamado `forma`, que indica sus dimensiones. Es importante conocer las formas de los tensores de PyTorch, ya que las formas no coincidentes generarán errores al operar con ellos. Por ejemplo, si queremos averiguar la forma del tensor `heights_tensor`, podemos hacer lo siguiente:

```
print(heights_tensor.shape)
```

La salida es

```
torch.Size([46])
```

Esto nos dice que `heights_tensor` es un tensor 1D con 46 valores.

También puedes cambiar la forma de un tensor de PyTorch. Para aprender cómo, primero convertiremos las alturas de centímetros a pies. Como un pie equivale aproximadamente a 30,48 centímetros, podemos lograrlo dividiendo el tensor entre 30,48:

```
heights_in_feet = heights_tensor / 30.48
print(heights_in_feet)
```

Esto conduce al siguiente resultado (omití algunos valores para ahorrar espacio; el resultado completo está en el repositorio de GitHub del libro):

```
tensor([6.2008, 5.5774, 6.2008, 5.3478, 6.0039, 5.6102, 6.0696, ...
       6.0039], dtype=torch.float64)
```

El nuevo tensor, `heights_in_feet`, almacena las alturas en pies. Por ejemplo, el último valor del tensor muestra que Joe Biden mide 1,83 metros.

Podemos usar el método `cat()` en PyTorch para concatenar los dos tensores:

```
heights_2_measures = torch.cat(
    [heights_tensor, heights_in_feet], dim=0)
print(heights_2_measures.shape)
```

El argumento `dim` se utiliza en diversas operaciones tensoriales para especificar la dimensión en la que se realizará la operación. En la celda de código anterior, `dim=0` significa que concatenamos los dos tensores a lo largo de la primera dimensión. Esto genera el siguiente resultado:

```
torch.Size([92])
```

El tensor resultante es unidimensional con 92 valores, algunos en centímetros y otros en pies. Por lo tanto, debemos transformarlo en dos filas y 46 columnas para que la primera fila represente las alturas en centímetros y la segunda en pies.

```
heights_reshaped = heights_2_measures.reshape(2, 46)
```

El nuevo tensor, `heights_reshaped`, es bidimensional con forma (2, 46). También podemos indexar y segmentar tensores multidimensionales usando corchetes. Por ejemplo, para imprimir la altura de Trump en pies, podemos hacer lo siguiente:

```
print(heights_reshaped[1, -2])
```

Esto conduce a un resultado de

```
tensor(6.2664, dtype=torch.float64)
```

El comando `heights_reshaped[1, -2]` le dice a Python que busque el valor en la segunda fila y la segunda última columna, que devuelve la altura de Trump en pies, 6.2664.

CONSEJO: El número de índices necesarios para referirse a valores escalares dentro del tensor es igual a su dimensionalidad. Por eso, usamos un solo índice para localizar valores en el tensor unidimensional, `heights_tensor` pero usamos dos para localizar valores en el bidimensional `heights_reshaped`.

Ejercicio 2.4

Utilice la indexación para obtener la altura de Joe Biden en el tensor `heights_reshaped` en centímetros.

2.1.4 Operaciones matemáticas con tensores de PyTorch

Podemos realizar operaciones matemáticas con tensores de PyTorch mediante diferentes métodos, como `mean()`, `median()`, `sum()`, `max()`, etc.

Por ejemplo, para hallar la mediana de la estatura de los 46 presidentes en centímetros, podemos hacer lo siguiente:

```
print(torch.median(heights_reshaped[0, :]))
```

El fragmento de código `heights_reshaped[0, :]` devuelve la primera fila y todas las columnas del tensor `heights_reshaped`. La línea de código anterior devuelve el valor de la mediana en la primera fila, lo que genera una salida de

```
tensor(182., dtype=torch.float64)
```

Esto significa que la altura media de los presidentes estadounidenses es de 182 centímetros.

Para encontrar la altura promedio en ambas filas, podemos usar el argumento `dim=1` en el método `mean()`:

```
print(torch.mean(heights_reshaped, dim=1))
```

El `dim=1` argumento indica que los promedios se calculan contrayendo columnas (la dimensión indexada en 1), obteniendo así promedios a lo largo de la dimensión indexada en 0 (filas). El resultado es

```
tensor([180.0652, 5.9077], dtype=torch.float64)
```

Los resultados muestran que los valores promedio en las dos filas son 180,0652 centímetros y 5,9077 pies.

Para saber quién es el presidente más alto, podemos hacer esto:

```
values, indices = torch.max(heights_reshaped, dim=1)
print(values)
print(indices)
```

La salida es

```
tensor([193.0000,    6.3320], dtype=torch.float64)
tensor([15, 15])
```

El método `torch.max()` devuelve dos tensores: uno `values` con la estatura del presidente más alto (en centímetros y pies) y otro `indices` con los índices del presidente con la estatura máxima. Los resultados muestran que el decimosexto presidente (Lincoln) es el más alto, con 193 centímetros (6,332 pies).

Ejercicio 2.5

Utilice este método `torch.min()` para averiguar el índice y la altura del presidente más bajo de Estados Unidos.

2.2 Un proyecto de aprendizaje profundo de extremo a extremo con PyTorch

En las siguientes secciones, trabajarás en un proyecto de aprendizaje profundo de ejemplo con PyTorch, donde aprenderás a clasificar imágenes en escala de grises de prendas de vestir en uno de los 10 tipos. En esta sección, primero ofreceremos una descripción general de los pasos necesarios. A continuación, explicaremos cómo obtener datos de entrenamiento para este proyecto y cómo preprocesarlos.

2.2.1 Aprendizaje profundo en PyTorch: una descripción general de alto nivel

Nuestro trabajo en este proyecto consiste en crear y entrenar una red neuronal profunda en PyTorch para clasificar imágenes en escala de grises de prendas de vestir. La Figura 2.1 muestra un diagrama de los pasos implicados.

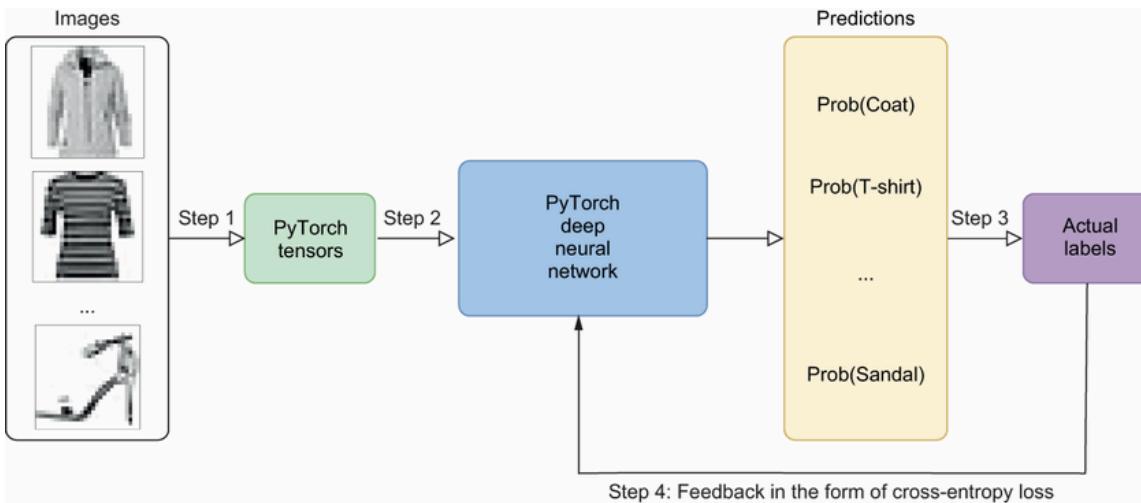


Figura 2.1 Los pasos involucrados en el entrenamiento de un modelo de aprendizaje profundo

Primero, obtendremos un conjunto de datos de imágenes de ropa en escala de grises, como se muestra a la izquierda de la figura 2.1. Las imágenes están en píxeles sin procesar y las convertiremos a tensores de PyTorch en forma de números de coma flotante (paso 1). Cada imagen incluye una etiqueta.

Luego, crearemos una red neuronal profunda en PyTorch, como se muestra en el centro de la figura 2.1. Algunas redes neuronalesLas redes de este libro utilizan redes neuronales convolucionales (CNN). Para este sencillo problema de clasificación, por el momento, solo utilizaremos capas densas.

Seleccionaremos una función de pérdida para la clasificación multicategoría. La pérdida de entropía cruzada se utiliza comúnmente para esta tarea. Esta pérdida mide la diferencia entre la distribución de probabilidad predicha y la distribución real de las etiquetas. Utilizaremos el optimizador Adam (una variante del algoritmo de descenso de gradiente) para actualizar los pesos de la red durante el entrenamiento. Establecemos la tasa de aprendizaje en 0,001. Esta tasa controla cuánto se ajustan los pesos del modelo con respecto al gradiente de pérdida durante el entrenamiento.

Optimizadores en ML

Los optimizadores en ML son algoritmos que actualizan los parámetros del modelo en función de la información del gradiente. Para minimizar la función de pérdida. El Descenso de Gradiente Estocástico (SGD) es el optimizador más fundamental, ya que utiliza actualizaciones sencillas basadas en el gradiente de pérdida. Adam, el optimizador más popular, es co-

nocido por su eficiencia y rendimiento inmediato.s combina las fortalezas del Adaptive GrAlgoritmo adient (AdaGrad) y Propagación de la Raíz Cuadrática Media (RMSProp). A pesar de sus diferencias, todos los optimizadores buscan ajustar iterativamente los parámetros para minimizar la función de pérdida, creando cada uno una ruta de optimización única para alcanzar este objetivo.

Dividiremos los datos de entrenamiento en un conjunto de entrenamiento y un conjunto de validación. En ML, solemos usar el conjunto de validación para proporcionar una evaluación imparcial del modelo y seleccionar los mejores hiperparámetros, como la tasa de aprendizaje, el número de épocas de entrenamiento, etc. El conjunto de validación también puede usarse para evitar el sobreajuste del modelo, en el que este funciona bien en el conjunto de entrenamiento, pero mal con datos no analizados. Una época es cuando todos los datos de entrenamiento se utilizan para entrenar el modelo una sola vez.

Durante el entrenamiento, iterarás los datos de entrenamiento. Durante los pasos hacia adelante, alimentarás imágenes a la red para obtener predicciones (paso 2) y calcularás la pérdida comparando las etiquetas predichas con las reales (paso 3; véase el lado derecho de la figura 2.1). A continuación, retropropagarás el gradiente a través de la red para actualizar los pesos. Aquí es donde ocurre el aprendizaje (paso 4), como se muestra en la parte inferior de la figura 2.1.

Utilizará el conjunto de validación para determinar cuándo debemos detener el entrenamiento. Calculamos la pérdida en el conjunto de validación. Si el modelo deja de mejorar después de un número fijo de épocas, lo consideramos entrenado. A continuación, evaluamos el modelo entrenado en el conjunto de prueba para evaluar su rendimiento al clasificar imágenes en diferentes etiquetas.

Ahora que tienes una descripción general de alto nivel de cómo funciona el aprendizaje profundo en PyTorch, ¡profundicemos en el proyecto de principio a fin!

2.2.2 Preprocesamiento de datos

En este proyecto, utilizaremos el conjunto de datos Fashion Modified del Instituto Nacional de Estándares y Tecnología (MNIST). A lo largo del proyecto, aprenderá a usar los paquetes `datasets` y `transforms` de la biblioteca Torchvision, así como los paquetes `Dataloader` de PyTorch que le serán útiles en el resto del libro. Utilizará estas herramientas para preprocesar datos a lo largo del libro. La biblioteca Torchvision proporciona herramientas para el procesamiento de imágenes, incluyendo conjuntos de datos populares, arquitecturas de modelos y transformaciones de imágenes comunes para aplicaciones de aprendizaje profundo.

Primero importamos las bibliotecas necesarias e instanciamos una clase `Compose()` en el paquete `transforms` para transformar imágenes sin procesar en tensores de PyTorch.

Listado 2.2 Transformación de datos de imágenes sin procesar en tensores de PyTorch

```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as T

torch.manual_seed(42)
transform=T.Compose([
    T.ToTensor(),                      ①
    T.Normalize([0.5],[0.5])])          ②
                                    ③
```

① Compone varias transformaciones juntas

② Convierte los píxeles de la imagen en tensores de PyTorch

③ Normaliza los valores al rango $[-1, 1]$

Utilizamos el método `manual_seed()` en PyTorch para corregir el estado aleatorio y que los resultados sean reproducibles. El paquete *de transformaciones* de Torchvision permite crear una serie de transformaciones para preprocesar imágenes. La clase `ToTensor()` convierte datos de imagen (en formatos de imagen de la Biblioteca de Imágenes de Python

(PIL) o en matrices de NumPy) en tensores de PyTorch. En concreto, los datos de imagen son números enteros de 0 a 255, y la clase `ToTensor()` los convierte en tensores de coma flotante con valores entre 0,0 y 1,0.

La clase `Normalize()` normaliza imágenes tensoriales con media y desviación estándar para n canales. Los datos de Fashion MNIST son imágenes en escala de grises de prendas de vestir, por lo que solo hay un canal de color. Más adelante en este libro, analizaremos imágenes con tres canales de color diferentes (rojo, verde y azul). En la celda de código anterior, `Normalize([0.5],[0.5])` significa que restamos 0,5 a los datos y dividimos la diferencia entre 0,5. Los datos de la imagen resultante varían de -1 a 1. Normalizar los datos de entrada al rango [-1, 1] permite que el descenso de gradiente funcione de forma más eficiente al mantener tamaños de paso más uniformes en todas las dimensiones. Esto facilita una convergencia más rápida durante el entrenamiento, algo que se hará con frecuencia en este libro.

NOTA: El código del listado 2.2 solo define el proceso de transformación de datos. No realiza la transformación propiamente dicha, que se realiza en la siguiente celda.

A continuación, utilizamos el paquete *de conjuntos de datos* de Torchvision para descargar el conjunto de datos a una carpeta en su computadora y realizar la transformación:

```
train_set=torchvision.datasets.FashionMNIST(      ①  
    root=".",  
    train=True,  
    download=True,  
    transform=transform)                            ②  
  
test_set=torchvision.datasets.FashionMNIST(root=".")  
    train=False,download=True,transform=transform)    ③
```

① ¿Qué conjunto de datos descargar?

② Dónde guardar los datos

③ El conjunto de datos de entrenamiento o prueba

④ Si desea o no descargar los datos a su computadora

⑤ Realiza la transformación de datos

Puede imprimir la primera muestra en el conjunto de entrenamiento:

```
print(train_set[0])
```

La primera muestra consiste en un tensor con 784 valores y una etiqueta 9. Los 784 números representan una imagen en escala de grises de 28 por 28 ($28 \times 28 = 784$), y la etiqueta 9 significa que es una bota al tobillo. Te estarás preguntando: ¿Cómo sabes que la etiqueta 9 indica una bota al tobillo? Hay 10 tipos diferentes de prendas de vestir. Las etiquetas en el conjunto de datos están numeradas del 0 al 9. Puedes buscar en línea y encontrar las etiquetas de texto para las 10 categorías (por ejemplo, obtuve las etiquetas de texto aquí <https://github.com/pranay414/Fashion-MNIST-Pytorch>). La lista `text_labels` contiene las 10 etiquetas de texto correspondientes a las etiquetas numéricas del 0 al 9. Por ejemplo, si un artículo tiene una etiqueta numérica de 0 en el conjunto de datos, la etiqueta de texto correspondiente es "camiseta". La lista `text_labels` se define de la siguiente manera:

```
text_labels=['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
            'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
```

Podemos trazar los datos para visualizar las prendas de vestir en el conjunto de datos.

Listado 2.3 Visualización de las prendas de vestir

```
!pip install matplotlib
import matplotlib.pyplot as plt

plt.figure(dpi=300, figsize=(8,4))
for i in range(24):
    ax=plt.subplot(3, 8, i + 1) ①
    img=train_set[i][0] ②
    img=img/2+0.5 ③
    img=img.reshape(28, 28) ④
    plt.imshow(img,
               cmap="binary")
```

```

plt.axis('off')
plt.title(text_labels[train_set[i][1]],      ⑤
          fontsize=8)
plt.show()

```

- ① Dónde colocar la imagen
- ② Obtiene la imagen i-ésima de los datos de entrenamiento
- ③ Convierte los valores de [-1,1] a [0,1]
- ④ Remodela la imagen a 28 por 28
- ⑤ Agrega una etiqueta de texto a cada imagen

La gráfica de la figura 2.2 muestra 24 prendas de vestir, como abrigos, suéteres, sandalias, etc.



Figura 2.2 Imágenes en escala de grises de prendas de vestir en el conjunto de datos Fashion MNIST.

En las próximas dos secciones, aprenderá a crear redes neuronales profundas con PyTorch para realizar problemas de clasificación binaria y multicategoría.

2.3 Clasificación binaria

En esta sección, primero crearemos lotes de datos para el entrenamiento. Después, construiremos una red neuronal profunda en PyTorch para este

propósito y entrenaremos el modelo con los datos. Finalmente, usaremos el modelo entrenado para realizar predicciones y comprobar su precisión. Los pasos para las clasificaciones binarias y multicategoría son similares, con algunas excepciones notables que destacaré más adelante.

2.3.1 Creación de lotes

Crearemos un conjunto de entrenamiento y uno de prueba que contengan solo dos tipos de prendas: camisetas y botines. (Más adelante en este capítulo, cuando analicemos la clasificación multicategoría, también aprenderá a crear un conjunto de validación para determinar cuándo detener el entrenamiento). La siguiente celda de código cumple este objetivo:

```
binary_train_set=[x for x in train_set if x[1] in [0,9]]  
binary_test_set=[x for x in test_set if x[1] in [0,9]]
```

Solo conservamos muestras con etiquetas numéricas 0 y 9 para crear un problema de clasificación binaria con un conjunto de entrenamiento equilibrado. A continuación, creamos lotes para entrenar la red neuronal profunda.

Listado 2.4 Creación de lotes para entrenamiento y pruebas

```
batch_size=64  
binary_train_loader=torch.utils.data.DataLoader(  
    binary_train_set,  
    batch_size=batch_size,  
    shuffle=True)  
binary_test_loader=torch.utils.data.DataLoader(  
    binary_test_set,  
    batch_size=batch_size,shuffle=True)
```

① Crea lotes para el conjunto de entrenamiento binario

② Número de muestras en cada lote

③ Baraja las observaciones al agruparlas

④ Crea lotes para el conjunto de pruebas binarias

La clase `DataLoader` del paquete PyTorch `utils` ayuda a crear iteradores de datos en lotes. Establecemos el tamaño del lote en 64. Creamos dos cargadores de datos en el listado 2.4: un conjunto de entrenamiento y un conjunto de prueba para la clasificación binaria. Al crear lotes, mezclamos las observaciones para evitar correlaciones en el conjunto de datos original: el entrenamiento es más estable si las diferentes etiquetas se distribuyen uniformemente en el cargador de datos.

2.3.2 Construcción y entrenamiento de un modelo de clasificación binaria

Primero crearemos un modelo de clasificación binaria. Después, lo entremos con imágenes de camisetas y botines. Una vez entrenado, veremos si puede distinguir las camisetas de los botines. Usamos PyTorch para crear la siguiente red neuronal con la clase `nn.Sequential` PyTorch (en capítulos posteriores, también aprenderás a usar esta clase `nn.Module` para crear redes neuronales PyTorch).

Listado 2.5 Creación de un modelo de clasificación binaria

```
import torch.nn as nn

device="cuda" if torch.cuda.is_available() else "cpu" ①

binary_model=nn.Sequential( ②
    nn.Linear(28*28,256), ③
    nn.ReLU(), ④
    nn.Linear(256,128),
    nn.ReLU(),
    nn.Linear(128,32),
    nn.ReLU(),
    nn.Linear(32,1),
    nn.Dropout(p=0.25),
    nn.Sigmoid()).to(device) ⑤
```

① PyTorch detecta automáticamente si hay una GPU habilitada para CUDA disponible.

② Crea una red neuronal secuencial en PyTorch

③ Número de neuronas de entrada y salida en una capa lineal

④ Aplica la activación de ReLU a las salidas de la capa

⑤ Aplica activación sigmoidea y mueve el modelo a una GPU si está disponible

La clase `Linear()` en PyTorch crea una transformación lineal de los datos entrantes. Esto crea efectivamente una capa densa en la red neuronal. La forma de entrada es 784 porque luego aplanaremos la imagen 2D a un vector 1D con $28 \times 28 = 784$ valores. Aplanamos la imagen 2D a un tensor 1D porque las capas densas solo aceptan entradas 1D. En capítulos posteriores, verá que no necesita aplanar las imágenes cuando usa capas convolucionales. Hay tres capas ocultas en la red, con 256, 128 y 32 neuronas, respectivamente. Los números 256, 128 y 32 se eligen de forma algo arbitraria: cambiarlos a, por ejemplo, 300, 200 y 50 no afectará el proceso de entrenamiento. Aplicamos la función de activación de la unidad lineal rectificada (ReLU) en las tres capas ocultas. La función de activación ReLU decide si una neurona debe activarse según la suma ponderada. Estas funciones introducen no linealidad en la salida de una neurona para que la red pueda aprender relaciones no lineales entre entradas y salidas. ReLU es la función de activación de referencia, con muy pocas excepciones, y encontrará otras funciones de activación en capítulos posteriores.

La salida de la última capa del modelo contiene un solo valor, y utilizamos la función de activación sigmoidea para reducir el número al rango [0, 1], de modo que pueda interpretarse como la probabilidad de que el objeto sea un botín. Con la probabilidad complementaria, el objeto es una camiseta.

Aquí establecemos la tasa de aprendizaje y definimos el optimizador y la función de pérdida:

```
lr=0.001  
optimizer=torch.optim.Adam(binary_model.parameters(), lr=lr)  
loss_fn=nn.BCELoss()
```

Establecemos la tasa de aprendizaje en 0,001. Determinar qué tasa de aprendizaje establecer es una cuestión empírica, y la respuesta se obtiene

con la experiencia. También se puede determinar mediante el ajuste de hiperparámetros con un conjunto de validación. La mayoría de los optimizadores de PyTorch utilizan una tasa de aprendizaje predeterminada de 0,001. El optimizador Adam es una variante del algoritmo de descenso de gradiente, que se utiliza para determinar cuánto ajustar los parámetros del modelo en cada paso de entrenamiento. El optimizador Adam fue introducido por primera vez en 2014 por Diederik Kingma y Jimmy Ba.¹ En el algoritmo tradicional de descenso de gradiente, solo se consideran los gradientes de la iteración actual. El optimizador Adam, en cambio, también considera los gradientes de iteraciones anteriores.

Utilizamos `nn.BCELoss()`, que es la función de pérdida de entropía cruzada binaria. Las funciones de pérdida miden el rendimiento de un modelo de aprendizaje automático (ML). El entrenamiento de un modelo implica ajustar los parámetros para minimizar la función de pérdida. La función de pérdida de entropía cruzada binaria se utiliza ampliamente en aprendizaje automático, especialmente en problemas de clasificación binaria. Mide el rendimiento de un modelo de clasificación cuyo resultado es un valor de probabilidad entre 0 y 1. La pérdida de entropía cruzada aumenta a medida que la probabilidad predicha se desvía de la etiqueta real.

Entrenamos la red neuronal que acabamos de crear como se muestra en el siguiente listado.

Listado 2.6 Entrenamiento de un modelo de clasificación binaria

```
for i in range(50):  
    tloss=0  
    for n,(imgs,labels) in enumerate(train_loader):  
        imgs=imgs.reshape(-1,28*28)  
        imgs=imgs.to(device)  
        labels=torch.FloatTensor(\n            [x if x==0 else 1 for x in labels])  
        labels=labels.reshape(-1,1).to(device)  
        preds=binary_model(imgs)  
        loss=loss_fn(preds,labels)  
        optimizer.zero_grad()  
        loss.backward()  
        optimizer.step()  
        tloss+=loss.detach()
```

```
tloss=tloss/n  
print(f"at epoch {i}, loss is {tloss}")
```

- ① Trenes para 50 épocas
- ② Itera a través de todos los lotes
- ③ Aplana la imagen antes de mover el tensor a la GPU
- ④ Convierte las etiquetas a 0 y 1
- ⑤ Calcula la pérdida
- ⑥ Retropropagación

Al entrenar modelos de aprendizaje profundo en PyTorch, `loss.backward()` se calcula el gradiente de la pérdida con respecto a cada parámetro del modelo, lo que permite la retropropagación, y `optimizer.step()` se actualizan los parámetros del modelo en función de estos gradientes calculados para minimizar la pérdida. Para simplificar, entrenamos el modelo con 50 épocas (una época es cuando los datos de entrenamiento se utilizan para entrenar el modelo una vez). En la siguiente sección, utilizará un conjunto de validación y una clase de detención temprana para determinar cuántas épocas entrenar. En las clasificaciones binarias, etiquetamos los objetivos como 0 y 1. Dado que solo conservamos las camisetas y los botines con las etiquetas 0 y 9, respectivamente, las convertimos a 0 y 1 en el listado 2.6. Como resultado, las etiquetas para las dos categorías de prendas de vestir son 0 y 1, respectivamente.

Este entrenamiento toma unos minutos si se usa entrenamiento con GPU. Toma más tiempo si se usa entrenamiento con CPU, pero el tiempo de entrenamiento debería ser inferior a una hora.

2.3.3 Prueba del modelo de clasificación binaria

La predicción del modelo de clasificación binaria entrenado es un número entre 0 y 1. Utilizaremos el método `torch.where()` para convertir las predicciones en 0 y 1: si la probabilidad predicha es menor que 0,5, la etiquetamos como 0; de lo contrario, como 1. A continuación, comparamos estas predicciones con las etiquetas reales para calcular su precisión. En la si-

guiente lista, utilizamos el modelo entrenado para realizar predicciones en el conjunto de datos de prueba.

Listado 2.7 Cálculo de la precisión de las predicciones

```
import numpy as np
results=[]
for imgs,labels in binary_test_loader:
    imgs=imgs.reshape(-1,28*28).to(device)
    labels=(labels/9).reshape(-1,1).to(device)
    preds=binary_model(imgs)
    pred10=torch.where(preds>0.5,1,0) ①
    correct=(pred10==labels) ②
    results.append(correct.detach().cpu()\
        .numpy().mean()) ③
accuracy=np.array(results).mean() ④
print(f"the accuracy of the predictions is {accuracy}") ⑤
```

① Itera a través de todos los lotes en el conjunto de prueba

② Hace predicciones utilizando el modelo entrenado

③ Compara predicciones con etiquetas

④ Calcula la precisión en el lote

⑤ Calcula la precisión en el conjunto de prueba

Iteramos todos los lotes de datos del conjunto de prueba. El modelo entrenado genera una probabilidad de que la imagen sea un botín. Luego, convertimos la probabilidad a 0 o 1 con base en el valor de corte de 0,5 mediante el método `torch.where()`. Tras la conversión, las predicciones son 0 (es decir, una camiseta) o 1 (un botín). Comparamos las predicciones con las etiquetas reales y observamos cuántas veces acierta el modelo. Los resultados muestran una precisión del 87,84 % en el conjunto de prueba.

2.4 Clasificación multicategoría

En esta sección, construiremos una red neuronal profunda en PyTorch para clasificar las prendas en una de las 10 categorías. Despues, entrenare-

mos el modelo con el conjunto de datos Fashion MNIST. Finalmente, usaremos el modelo entrenado para realizar predicciones y comprobar su precisión. Primero, crearemos un conjunto de validación y definiremos una clase de detención temprana para determinar cuándo detener el entrenamiento.

2.4.1 Conjunto de validación y detención anticipada

Al construir y entrenar una red neuronal profunda, podemos elegir numerosos hiperparámetros (como la tasa de aprendizaje y el número de épocas de entrenamiento). Estos hiperparámetros afectan el rendimiento del modelo. Para encontrar los mejores hiperparámetros, podemos crear un conjunto de validación para probar el rendimiento del modelo con diferentes hiperparámetros.

Por ejemplo, crearemos un conjunto de validación en la clasificación multi-categoría para determinar el número óptimo de épocas de entrenamiento. Lo hacemos en el conjunto de validación en lugar del conjunto de entrenamiento para evitar el sobreajuste, cuando un modelo tiene un buen rendimiento en el conjunto de entrenamiento, pero un rendimiento deficiente en pruebas fuera de la muestra (es decir, con datos no analizados).

Aquí dividimos 60.000 observaciones del conjunto de datos de entrenamiento en un conjunto de entrenamiento y un conjunto de validación:

```
train_set, val_set = torch.utils.data.random_split(\n    train_set, [50000, 10000])
```

El conjunto de trenes original ahora se convierte en dos conjuntos: el nuevo conjunto de trenes con 50.000 observaciones y un conjunto de validación con las 10.000 observaciones restantes.

Usamos la clase `DataLoader` en el paquete *de utilidades* de PyTorch para convertir los conjuntos de entrenamiento, validación y prueba en tres iteradores de datos en lotes:

```
train_loader = torch.utils.data.DataLoader(\n    train_set,\n    batch_size=batch_size,
```

```
shuffle=True)
val_loader=torch.utils.data.DataLoader(
    val_set,
    batch_size=batch_size,
    shuffle=True)
test_loader=torch.utils.data.DataLoader(
    test_set,
    batch_size=batch_size,
    shuffle=True)
```

A continuación, definimos una clase `EarlyStop()` y creamos una instancia de la clase.

Listado 2.8 La clase `EarlyStop()` para determinar cuándo detener el entrenamiento

```
class EarlyStop:
    def __init__(self, patience=10):          ①
        self.patience = patience
        self.steps = 0
        self.min_loss = float('inf')
    def stop(self, val_loss):                 ②
        if val_loss < self.min_loss:           ③
            self.min_loss = val_loss
            self.steps = 0
        elif val_loss >= self.min_loss:       ④
            self.steps += 1
        if self.steps >= self.patience:
            return True
        else:
            return False
stopper=EarlyStop()
```

① Establece el valor predeterminado de paciencia en 10

② Define el método `stop()`

③ Si se alcanza una nueva pérdida mínima, actualiza el valor de `min_loss`

④ Cuenta cuántas épocas han transcurrido desde la última pérdida mínima

La clase `EarlyStop()` determina si la pérdida en el conjunto de validación ha dejado de mejorar en las últimas épocas `patience=10`. El valor predefinido del argumento `patience` es 10, pero puede elegir un valor diferente al instanciar la clase. El valor de `patience` mide cuántas épocas desea entrenar desde la última vez que el modelo alcanzó la pérdida mínima. El método `stop()` registra la pérdida mínima y el número de épocas desde entonces, y compara este número con el valor de `patience`. El método devuelve el valor de `True` si el número de épocas desde la pérdida mínima es mayor que el valor de `patience`.

2.4.2 Construcción y entrenamiento de un modelo de clasificación multicategoría

El conjunto de datos de Moda MNIST contiene 10 categorías diferentes de prendas de vestir. Por lo tanto, creamos un modelo de clasificación multicategoría para clasificarlas. A continuación, aprenderá a crear dicho modelo y a entrenarlo. También aprenderá a realizar predicciones con el modelo entrenado y a evaluar su precisión. Utilizamos PyTorch para crear la red neuronal para la clasificación multicategoría que se muestra a continuación.

Listado 2.9 Creación de un modelo de clasificación multicategoría

```
model=nn.Sequential(  
    nn.Linear(28*28,256),  
    nn.ReLU(),  
    nn.Linear(256,128),  
    nn.ReLU(),  
    nn.Linear(128,64),  
    nn.ReLU(),  
    nn.Linear(64,10)                                     ①  
).to(device)                                         ②
```

① Hay 10 neuronas en la capa de salida.

② No aplica la activación softmax en la salida

En comparación con el modelo de clasificación binaria que creamos en la sección anterior, hemos realizado algunos cambios. Primero, la salida ahora tiene 10 valores, que representan los 10 tipos diferentes de prendas

de vestir del conjunto de datos. Segundo, hemos cambiado el número de neuronas en la última capa oculta de 32 a 64. Una regla general para crear redes neuronales profundas es aumentar o disminuir gradualmente el número de neuronas de una capa a la siguiente. Dado que el número de neuronas de salida ha aumentado de 1 (en la clasificación binaria) a 10 (en la clasificación multicategoría), cambiamos el número de neuronas de 32 a 64 en la penúltima capa para que coincida con el aumento. Sin embargo, el número 64 no tiene nada de especial: si se utilizan, por ejemplo, 100 neuronas en la penúltima capa, se obtendrán resultados similares.

Usaremos la clase `nn.CrossEntropyLoss()` PyTorch como función de pérdida, que combina `nn.LogSoftmax()` y `nn.NLLLoss()` en una sola clase. Consulte la documentación aquí para obtener más información:

<https://mng.bz/pxd2>. En particular, la documentación indica: "Este criterio calcula la pérdida de entropía cruzada entre los logits de entrada y el objetivo". Esto explica por qué no aplicamos la activación softmax en la lista anterior. En el repositorio de GitHub del libro, he demostrado que si usamos `nn.LogSoftmax()` en el modelo y `nn.NLLLoss()` como función de pérdida, obtenemos resultados idénticos.

Como resultado, la clase `nn.CrossEntropyLoss()` aplicará la función de activación softmax en la salida para comprimir los 10 números en el rango $[0, 1]$ antes de la operación logarítmica. La función de activación preferida en la salida es sigmoidea en clasificaciones binarias y softmax en clasificaciones multicategoría. Además, los 10 números después de la activación softmax suman 1, lo que puede interpretarse como las probabilidades correspondientes a los 10 tipos de prendas de vestir. Utilizaremos la misma tasa de aprendizaje y optimizador que en la clasificación binaria de la sección anterior.

```
lr=0.001
optimizer=torch.optim.Adam(model.parameters(), lr=lr)
loss_fn=nn.CrossEntropyLoss()
```

Definimos `train_epoch()` de la siguiente manera:

```
def train_epoch():
    tloss=0
    for n,(imgs,labels) in enumerate(train_loader):
```

```
    imgs=imgs.reshape(-1,28*28).to(device)
    labels=labels.reshape(-1,).to(device)
    preds=model(imgs)
    loss=loss_fn(preds,labels)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    tloss+=loss.detach()
return tloss/n
```

La función entrena el modelo para una época. El código es similar al que vimos en la clasificación binaria, excepto que las etiquetas van del 0 al 9, en lugar de dos números (0 y 1).

También definimos una función `val_epoch()`:

```
def val_epoch():
    vloss=0
    for n,(imgs,labels) in enumerate(val_loader):
        imgs=imgs.reshape(-1,28*28).to(device)
        labels=labels.reshape(-1,).to(device)
        preds=model(imgs)
        loss=loss_fn(preds,labels)
        vloss+=loss.detach()
    return vloss/n
```

La función utiliza el modelo para realizar predicciones sobre las imágenes en el conjunto de validación y calcular la pérdida promedio por lote de datos.

Ahora entrenamos el clasificador multicategoría:

```
for i in range(1,101):
    tloss=train_epoch()
    vloss=val_epoch()
    print(f"at epoch {i}, tloss is {tloss}, vloss is {vloss}")
    if stopper.stop(vloss)==True:
        break
```

Entrenamos un máximo de 100 épocas. En cada época, entrenamos primero el modelo con el conjunto de entrenamiento. Luego, calculamos la

pérdida promedio por lote en el conjunto de validación. Utilizamos la clase `EarlyStop()` para determinar si el entrenamiento debe detenerse, analizando la pérdida en el conjunto de validación. El entrenamiento se detiene si la pérdida no ha mejorado en las últimas 10 épocas. Después de 19 épocas, el entrenamiento se detiene.

El entrenamiento tarda aproximadamente 5 minutos si se utiliza el entrenamiento GPU, lo cual es más largo que el proceso de entrenamiento en clasificación binaria ya que ahora tenemos más observaciones en el conjunto de entrenamiento (10 prendas de vestir en lugar de solo 2).

El resultado del modelo es un vector de 10 números. Usamos este vector `torch.argmax()` para asignar a cada observación una etiqueta según la probabilidad más alta. Luego, comparamos la etiqueta predicha con la etiqueta real. Para ilustrar cómo funciona la predicción, veamos las predicciones de las primeras cinco imágenes del conjunto de prueba.

Listado 2.10 Prueba del modelo entrenado en cinco imágenes

```
plt.figure(dpi=300, figsize=(5,1))
for i in range(5):
    ax=plt.subplot(1,5, i + 1) (1)
    img=test_set[i][0]
    label=test_set[i][1]
    img=img/2+0.5
    img=img.reshape(28, 28)
    plt.imshow(img, cmap="binary")
    plt.axis('off')
    plt.title(text_labels[label]+f"; {label}", fontsize=8)
    plt.show()
for i in range(5):
    img,label = test_set[i] (2)
    img=img.reshape(-1,28*28).to(device)
    pred=model(img) (3)
    index_pred=torch.argmax(pred, dim=1) (4)
    idx=index_pred.item()
    print(f"the label is {label}; the prediction is {idx}") (5)
```

① Traza las primeras cinco imágenes del conjunto de prueba con sus etiquetas

- ② Obtiene la imagen y la etiqueta i-ésima en el conjunto de prueba
- ③ Predice utilizando el modelo entrenado
- ④ Utiliza el método `torch.argmax()` para obtener la etiqueta prevista
- ⑤ Imprime la etiqueta real y la etiqueta prevista

Graficamos las primeras cinco prendas del conjunto de prueba en una cuadrícula de 1×5 . Luego, usamos el modelo entrenado para hacer una predicción para cada prenda. La predicción es un tensor con 10 valores. El método `torch.argmax()` devuelve la posición de mayor probabilidad en el tensor, y la usamos como etiqueta predicha. Finalmente, imprimimos tanto la etiqueta real como la predicha para comparar y comprobar si las predicciones son correctas. Después de ejecutar el código anterior, debería ver la imagen de la figura 2.3.

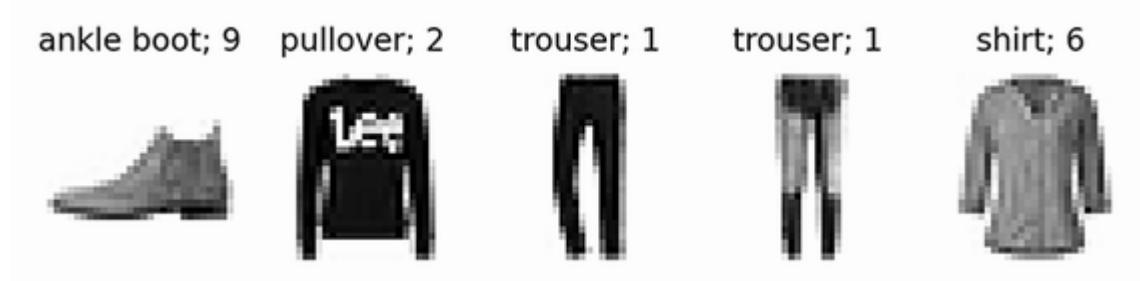


Figura 2.3 Las primeras cinco prendas del conjunto de datos de prueba y sus respectivas etiquetas. Cada prenda tiene una etiqueta de texto y una etiqueta numérica entre 0 y 9.

La figura 2.3 muestra que las primeras cinco prendas de vestir del conjunto de prueba son botín, jersey, pantalón, pantalón y camisa, respectivamente, con etiquetas numéricas 9, 2, 1, 1 y 6.

El resultado después de ejecutar el código del listado 2.10 es el siguiente:

```
the label is 9; the prediction is 9
the label is 2; the prediction is 2
the label is 1; the prediction is 1
the label is 1; the prediction is 1
the label is 6; the prediction is 6
```

El resultado anterior muestra que el modelo ha realizado predicciones correctas en las cinco prendas de vestir.

Arreglando el estado aleatorio en PyTorch

El método `torch.manual_seed()` corrige el estado aleatorio, por lo que los resultados son los mismos al volver a ejecutar los programas. Sin embargo, es posible que obtenga resultados diferentes a los reportados en este capítulo, incluso si usa la misma semilla aleatoria. Esto se debe a que los distintos hardware y versiones de PyTorch manejan las operaciones de punto flotante de forma ligeramente distinta. Consulte, por ejemplo, las explicaciones en <https://mng.bz/RNva>. La diferencia suele ser mínima, así que no se alarme.

A continuación, calculamos la precisión de las predicciones en todo el conjunto de datos de prueba.

Listado 2.11 Prueba del modelo de clasificación multicategoría entrenado

```
results=[]

for imgs,labels in test_loader:
    imgs=imgs.reshape(-1,28*28).to(device) ①
    labels=(labels).reshape(-1,).to(device)
    preds=model(imgs)
    pred10=torch.argmax(preds,dim=1) ②
    correct=(pred10==labels) ③
    results.append(correct.detach().cpu().numpy().mean()) ④

accuracy=np.array(results).mean() ⑤
print(f"the accuracy of the predictions is {accuracy}")
```

① Itera a través de todos los lotes en el conjunto de prueba

② Predice utilizando el modelo entrenado

③ Convierte las probabilidades en una etiqueta predicha

④ Compara la etiqueta prevista con la etiqueta real

⑤ Calcula la precisión en el conjunto de prueba

La salida es

```
the accuracy of the predictions is 0.8819665605095541
```

Iteramos todas las prendas del conjunto de prueba y usamos el modelo entrenado para hacer predicciones. Luego, comparamos las predicciones con las etiquetas reales. La precisión es de aproximadamente el 88 % en la prueba fuera de muestra. Dado que una suposición aleatoria tiene una precisión de aproximadamente el 10 %, una precisión del 88 % es bastante alta. Esto indica que hemos creado y entrenado dos modelos de aprendizaje profundo exitosos en PyTorch. Utilizarás estas habilidades con frecuencia más adelante en este libro. Por ejemplo, en el capítulo 3, la red discriminadora que construirás es esencialmente un modelo de clasificación binaria, similar al que has creado en este capítulo.

Resumen

- En PyTorch, utilizamos tensores para almacenar varias formas de datos de entrada para que podamos alimentarlos a modelos de aprendizaje profundo.
- Puede indexar y cortar tensores de PyTorch, remodelarlos y realizar operaciones matemáticas con ellos.
- El aprendizaje profundo es un tipo de método ML que utiliza redes neuronales artificiales profundas para aprender la relación entre los datos de entrada y salida.
- La función de activación ReLU decide si una neurona debe activarse según la suma ponderada. Introduce no linealidad en la salida de una neurona.
- Las funciones de pérdida miden el rendimiento de un modelo de aprendizaje automático. El entrenamiento de un modelo implica ajustar los parámetros para minimizar la función de pérdida.
- La clasificación binaria es un modelo ML para clasificar observaciones en una de dos categorías.
- La clasificación multicategoría es un modelo ML para clasificar observaciones en una de múltiples categorías.

¹ Diederik Kingma y Jimmy Ba, 2014, “Adam: un método para la optimización estocástica”. <https://arxiv.org/abs/1412.6980>.

3. Redes generativas antagónicas: generación de formas y números

Este capítulo cubre

- Construcción de redes generadoras y discriminadoras en redes generativas adversarias desde cero
- Uso de GAN para generar puntos de datos para formar formas (por ejemplo, una curva de crecimiento exponencial)
- Generar secuencias de números enteros que sean todos múltiplos de 5
- Entrenamiento, guardado, carga y uso de GAN
- Evaluación del rendimiento de GAN y determinación de los puntos de parada del entrenamiento

Cerca de la mitad de los modelos generativos de este libro pertenecen a una categoría denominada redes generativas antagónicas (GAN). El método fue propuesto por primera vez por Ian Goodfellow y sus coautores en 2014.¹ Las GAN, reconocidas por su facilidad de implementación y versatilidad, permiten a las personas, incluso con conocimientos básicos de aprendizaje profundo, construir sus modelos desde cero. El término "antagónico" en GAN se refiere a que dos redes neuronales compiten entre sí en un marco de juego de suma cero. La red generativa intenta crear instancias de datos indistinguibles de las muestras reales. Por el contrario, la red discriminativa intenta identificar las muestras generadas de las reales. Estos modelos versátiles pueden generar diversos formatos de contenido, desde formas geométricas y secuencias de números hasta imágenes en color de alta resolución e incluso composiciones musicales con un sonido realista.

En este capítulo, revisaremos brevemente la teoría de las GAN. Después, te mostraré cómo implementar ese conocimiento en PyTorch. Aprenderás a construir tu primera GAN desde cero para que todos los detalles queden desmitificados. Para que el ejemplo sea comprensible, imagina que depositas \$1 en una cuenta de ahorros que paga un 8% anual. Quieres averiguar el saldo de tu cuenta en función de los años que has invertido. La verdadera relación es una curva de crecimiento exponencial. Aprenderás a usar las GAN para generar muestras de datos: pares de valores (x, y) que forman una curva de crecimiento exponencial, con una relación matemática $y = 1,08^x$. Con esta habilidad, podrás generar datos que imiten cualquier forma: seno, coseno, cuadrática, etc.

En el segundo proyecto de este capítulo, aprenderá a usar GAN para generar una secuencia de números múltiplos de 5. Sin embargo, puede cambiar el patrón a múlti-

plos de 2, 3, 7 u otros patrones. Durante el proceso, aprenderá a crear una red generadora y una red discriminadora desde cero. Aprenderá a entrenar, guardar y usar GAN. Además, aprenderá a evaluar el rendimiento de las GAN, ya sea visualizando las muestras generadas por la red generadora o midiendo la divergencia entre la distribución de las muestras generadas y la distribución de los datos reales.

Imagine que necesita datos para entrenar un modelo de aprendizaje automático (ML) y predecir la relación entre pares de valores (x, y). Sin embargo, preparar manualmente el conjunto de datos de entrenamiento es costoso y requiere mucho tiempo. Las GAN pueden ser muy adecuadas para generar datos en estos casos: si bien los valores generados de x e y generalmente se ajustan a una relación matemática, también presentan ruido. Este ruido puede ser útil para evitar el sobreajuste cuando los datos generados se utilizan para entrenar el modelo de ML.

El objetivo principal de este capítulo no es necesariamente generar contenido novedoso con un uso práctico óptimo. En cambio, mi objetivo es enseñarte a entrenar y usar GAN para crear diversos formatos de contenido desde cero. A lo largo del camino, adquirirás una sólida comprensión del funcionamiento interno de las GAN. Esta base nos permitirá centrarnos en otros aspectos más avanzados de las GAN en capítulos posteriores, al generar otro tipo de contenido, como imágenes de alta resolución o música con sonido realista (por ejemplo, redes neuronales convolucionales o cómo representar una pieza musical como un objeto multidimensional).

3.1 Pasos involucrados en el entrenamiento de GAN

En el capítulo 1, obtuviste una visión general de las teorías que sustentan las GAN. En esta sección, resumiré los pasos para entrenar GAN en general y, en particular, para crear puntos de datos que formen una curva de crecimiento exponencial.

Volvamos a nuestro ejemplo anterior: planeas invertir en una cuenta de ahorros que paga un 8% de interés anual. Depositas \$1 hoy y quieres saber cuánto dinero tendrás en el futuro.

El monto en su cuenta en el futuro, y , depende de cuánto tiempo invierte en la cuenta de ahorros. Denotemos el número de años que invierte por x , que puede ser un número, digamos, entre 0 y 50. Por ejemplo, si invierte por 1 año, el saldo es \$1.08; si invierte por 2 años, el saldo es $1.08^2 = \$1.17$. Para generalizar, la relación entre x e y es $y = 1.08^x$. La función representa una curva de crecimiento exponencial. Tenga en cuenta aquí que x puede ser un número entero como 1 o 2, así como un número decimal como 1.14 o 2.35 y la fórmula sigue funcionando.

Entrenar GAN para generar puntos de datos que se ajusten a una relación matemática específica, como en el ejemplo anterior, es un proceso de varios pasos. En su

caso, desea generar puntos de datos (x, y) tales que $y = 1.08^x$. La Figura 3.1 muestra un diagrama de la arquitectura de las GAN y los pasos necesarios para generar una curva de crecimiento exponencial. Al generar otro contenido, como una secuencia de números enteros, imágenes o música, se siguen pasos similares, como verá en el segundo proyecto de este capítulo, así como en otros modelos de GAN más adelante en este libro.

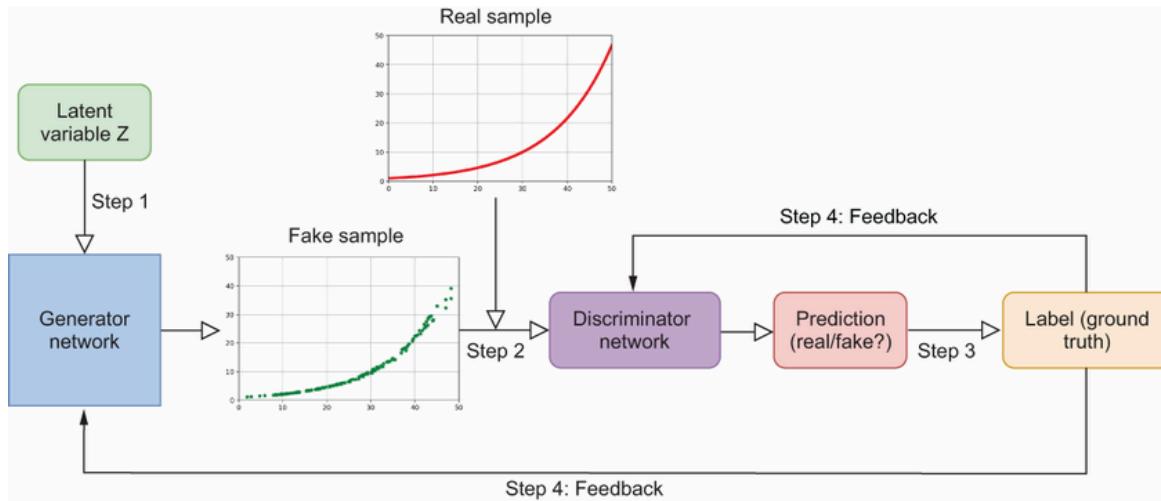


Figura 3.1 Pasos del entrenamiento de GAN para generar una curva de crecimiento exponencial y la arquitectura de red dual en GAN. El generador obtiene un vector de ruido aleatorio Z del espacio latente (arriba a la izquierda) para crear una muestra falsa y la presenta al discriminador (centro). El discriminador clasifica una muestra como real (del conjunto de entrenamiento) o falsa (creada por el generador). Las predicciones se comparan con la realidad fundamental y tanto el discriminador como el generador aprenden de ellas. Tras numerosas iteraciones de entrenamiento, el generador aprende a crear formas indistinguibles de las muestras reales.

Antes de comenzar, necesitamos obtener un conjunto de datos de entrenamiento para entrenar las GAN. En nuestro ejemplo en ejecución, generaremos un conjunto de datos de pares (x, y) usando la relación matemática $y = 1.08^x$. Usamos el ejemplo de la cuenta de ahorros para que los números sean relacionables. Las técnicas que aprendes en este capítulo se pueden aplicar a otras formas: seno, coseno, forma de U, etc. Puedes elegir un rango de valores de x (por ejemplo, de 0 a 50) y calcular los valores de y correspondientes. Dado que generalmente entrenamos modelos en lotes de datos en el aprendizaje profundo, el número de observaciones en tu conjunto de datos de entrenamiento generalmente se establece en un múltiplo del tamaño del lote. Una muestra real se encuentra en la parte superior de la figura 3.1, que tiene una forma de curva de crecimiento exponencial.

Una vez que tenga listo el conjunto de entrenamiento, necesita crear dos redes en las GAN: un generador y un discriminador. El generador, ubicado en la parte inferior izquierda de la figura 3.1, toma un vector de ruido aleatorio Z como entrada y genera puntos de datos (paso 1 de nuestro bucle de entrenamiento). El vector de ruido aleatorio Z que utiliza el generador se obtiene del espacio latente, que representa el rango de posibles salidas que la GAN puede producir y es fundamental para su capacidad de generar diversas muestras de datos. En el capítulo 5, exploraremos el espacio latente para seleccionar los atributos del contenido creado por el generador. El

discriminador, ubicado en el centro de la figura 3.1, evalúa si un punto de datos dado (x, y) es real (del conjunto de datos de entrenamiento) o falso (creado por el generador); este es el paso 2 de nuestro bucle de entrenamiento.

El significado del espacio latente

El espacio latente en una GAN es un espacio conceptual donde el generador puede transformar cada punto en una instancia de datos realista. Este espacio representa el rango de posibles resultados que la GAN puede producir y es fundamental para su capacidad de generar datos variados y complejos. El espacio latente adquiere su importancia exclusivamente cuando se emplea junto con el modelo generativo. En este contexto, se puede interpolar entre puntos del espacio latente para afectar los atributos de los resultados, lo cual se analizará en el capítulo 5.

Para saber cómo ajustar los parámetros del modelo, debemos elegir las funciones de pérdida adecuadas. Necesitamos definir las funciones de pérdida tanto para el generador como para el discriminador. La función de pérdida incentiva al generador a generar puntos de datos que se asemejan a los del conjunto de datos de entrenamiento, lo que hace que el discriminador los clasifique como reales. La función de pérdida incentiva al discriminador a clasificar correctamente los puntos de datos reales y generados.

En cada iteración del bucle de entrenamiento, alternamos entre el entrenamiento del discriminador y el del generador. Durante cada iteración, muestreamos un lote de puntos de datos reales (x, y) del conjunto de datos de entrenamiento y un lote de puntos de datos falsos generados por el generador. Al entrenar el discriminador, comparamos las predicciones del modelo discriminativo (la probabilidad de que la muestra provenga del conjunto de entrenamiento) con la realidad fundamental, que es 1 si la muestra es real y 0 si es falsa (mostrada a la derecha de la figura 3.1); esto constituye la mitad del paso 3 del bucle de entrenamiento. Ajustamos ligeramente los pesos de la red del discriminador para que, en la siguiente iteración, la probabilidad predicha se acerque más a la realidad fundamental (la mitad del paso 4 del bucle de entrenamiento).

Al entrenar el generador, introducimos muestras falsas en el modelo discriminativo y obtenemos la probabilidad de que la muestra sea real (la otra mitad del paso 3). A continuación, ajustamos ligeramente los pesos de la red del generador para que, en la siguiente iteración, la probabilidad predicha se acerque a 1 (ya que el generador busca crear muestras para engañar al discriminador y hacerles creer que son reales); esto constituye la otra mitad del paso 4. Repetimos este proceso durante varias iteraciones, lo que permite que la red del generador genere puntos de datos más realistas.

Una pregunta natural es cuándo detener el entrenamiento de las GAN. Para ello, se evalúa el rendimiento de la GAN generando un conjunto de datos sintéticos y comparándolos con los datos reales del conjunto de datos de entrenamiento. En la mayoría de los casos, utilizamos técnicas de visualización para evaluar la conformidad de los datos generados con la relación deseada. Sin embargo, en nuestro ejemplo de ejecución, dado que conocemos la distribución de los datos de entrenamiento, podemos calcular el error cuadrático medio (EMM) entre los datos generados y la distribución real. Detenemos el entrenamiento de las GAN cuando las muestras generadas dejan de mejorar sus cualidades tras un número determinado de rondas de entrenamiento.

En este punto, el modelo se considera entrenado. Descartamos el discriminador y conservamos el generador. Para crear una curva de crecimiento exponencial, alimentamos el generador entrenado con un vector de ruido aleatorio Z y obtenemos pares de (x, y) para formar la forma deseada.

3.2 Preparación de datos de entrenamiento

En esta sección, creará el conjunto de datos de entrenamiento para usarlo en el entrenamiento del modelo GAN más adelante en este capítulo. En concreto, creará pares de puntos de datos (x, y) que se ajusten a la forma de crecimiento exponencial. Los colocará en lotes para que estén listos para alimentarse a las redes neuronales profundas.

NOTA: El código de este capítulo, así como de otros capítulos de este libro, está disponible en el repositorio de GitHub del libro: <https://github.com/markhliu/DGAI>.

3.2.1 Un conjunto de datos de entrenamiento que forma una curva de crecimiento exponencial

Crearemos un conjunto de datos que contiene muchas observaciones de pares de datos, (x, y), donde x se distribuye uniformemente en el intervalo [0, 50] e y está relacionado con x según la fórmula $y = 1,08^x$, como se muestra en la siguiente lista.

Listado 3.1 Creación de datos de entrenamiento para formar una forma de crecimiento exponencial

```
import torch  
  
torch.manual_seed(0)                                ①  
  
observations = 2048  
  
train_data = torch.zeros((observations, 2))          ②
```

```
train_data[:,0]=50*torch.rand(observations)
```

(3)

```
train_data[:,1]=1.08**train_data[:,0]
```

(4)

① Corrige el estado aleatorio para que los resultados sean reproducibles

② Crea un tensor con 2048 filas y 2 columnas

③ Genera valores de x entre 0 y 50

④ Genera valores de y basados en la relación $y = 1.08^x$

Primero, creamos 2048 valores de x entre 0 y 50 usando el método `torch.rand()`. Usamos el método `manual_seed()` en PyTorch para corregir el estado aleatorio de modo que todos los resultados sean reproducibles. Primero creamos un tensor de PyTorch, `train_data` con 2048 filas y 2 columnas. Los valores de x se colocan en la primera columna del tensor `train_data`. El método `rand()` en PyTorch genera valores aleatorios entre 0.0 y 1.0. Al multiplicar el valor por 50, los valores resultantes de x están entre 0.0 y 50.0. Luego, completamos la segunda columna `train_data` con valores de $y = 1.08^x$.

Ejercicio 3.1

Modifique el listado 3.1 para que la relación entre x e y sea $y = \sin(x)$ mediante la función `torch.sin()`. Establezca el valor de x entre -5 y 5 utilizando esta línea de código: `train_data[:,0]=10*(torch.rand(observations)-0.5)`.

Graficamos la relación entre x e y utilizando la biblioteca Matplotlib.

Listado 3.2 Visualización de la relación entre x e y

```
import matplotlib.pyplot as plt

fig=plt.figure(dpi=100, figsize=(8,6))
plt.plot(train_data[:,0],train_data[:,1],".",c="r")      (1)
plt.xlabel("values of x", fontsize=15)
plt.ylabel("values of $y=1.08^x$ ", fontsize=15)          (2)
plt.title("An exponential growth shape", fontsize=20)    (3)
plt.show()
```

① Traza la relación entre x e y

② Etiquetas del eje y

③ Crea un título para la trama

Después de ejecutar el listado 3.2, verá una curva de crecimiento exponencial que es similar al gráfico superior de la figura 3.1.

Ejercicio 3.2

Modifique el listado 3.2 para representar gráficamente la relación entre x e $y = \text{sen}(x)$ según los cambios realizados en el ejercicio 3.1. Asegúrese de cambiar la etiqueta del eje y y el título del gráfico para reflejar los cambios realizados.

3.2.2 Preparación del conjunto de datos de entrenamiento

Colocaremos las muestras de datos que acabas de crear en lotes para alimentar la red discriminadora. Usamos la clase `DataLoader()` en PyTorch para encapsular un iterable alrededor del conjunto de datos de entrenamiento y así poder acceder fácilmente a las muestras durante el entrenamiento, de la siguiente manera:

```
from torch.utils.data import DataLoader  
  
batch_size=128  
train_loader=DataLoader(  
    train_data,  
    batch_size=batch_size,  
    shuffle=True)
```

Asegúrese de seleccionar el número total de observaciones y el tamaño del lote para que todos los lotes contengan la misma cantidad de muestras. Elegimos 2048 observaciones con un tamaño de lote de 128. Como resultado, tenemos $2048/128 = 16$ lotes. El argumento `shuffle=True` en `DataLoader()` baraja las observaciones aleatoriamente antes de dividirlas en lotes.

NOTA: La reorganización garantiza que las muestras de datos se distribuyan uniformemente y que las observaciones dentro de un lote no estén correlacionadas, lo que, a su vez, estabiliza el entrenamiento. En este ejemplo específico, la reorganización garantiza que los valores de x se ubiquen aleatoriamente entre 0 y 50, en lugar de agruparse en un rango determinado, por ejemplo, entre 0 y 5.

Puede acceder a un lote de datos utilizando los métodos `next()` y `iter()`, de la siguiente manera:

```
batch0=next(iter(train_loader))  
print(batch0)
```

Verá 128 pares de números (x, y) , donde el valor de x cae aleatoriamente entre 0 y 50. Además, los valores de x e y en cada par se ajustan a la relación $y = 1,08^x$.

3.3 Creación de GAN

Ahora que el conjunto de datos de entrenamiento está listo, crearemos una red discriminadora y una red generadora. La red discriminadora es un clasificador binario, muy similar al clasificador binario para prendas de vestir que creamos y entramos en el capítulo 2. En este caso, la función del discriminador es clasificar las muestras como reales o falsas. La red generadora, por otro lado, intenta crear puntos de datos (x, y) indistinguibles de los del conjunto de entrenamiento para que el discriminador los clasifique como reales.

3.3.1 La red discriminadora

Usamos PyTorch para crear una red neuronal discriminadora. Usaremos capas completamente conectadas (densas) con activaciones ReLU . También usaremos capas de abandono para evitar el sobreajuste. Creamos una red neuronal profunda secuencial en PyTorch para representar el discriminador, como se muestra en la siguiente lista.

Listado 3.3 Creación de una red discriminadora

```
importar torch.nn como nn  
  
dispositivo="cuda" si torch.cuda.is_available() de lo contrario "cpu"  
  
D=nn.Secuencial(  
    nn.Lineal(2,256),  
    nn.ReLU(),  
    nn.Abandono(0.3),  
    nn.Lineal(256,128),  
    nn.ReLU(),  
    nn.Abandono(0.3),  
    nn.Lineal(128,64),  
    nn.ReLU(),  
    nn.Abandono(0.3),  
    nn.Linear(64,1),  
    nn.Sigmoid()).to(dispositivo)
```

①

②

③

④

① Comprueba automáticamente si la GPU habilitada para CUDA está disponible.

② La cantidad de características de entrada en la primera capa es 2, lo que coincide con la cantidad de elementos en cada instancia de datos, que tiene dos valores, x e y.

③ La capa de abandono evita el sobreajuste.

④ El número de características de salida en la última capa es 1, de modo que podemos comprimirlo en un valor entre 0 y 1.

Asegúrese de que en la primera capa, la forma de entrada sea 2 porque, en nuestra muestra, cada instancia de datos tiene dos valores: x e y . El número de entradas en la primera capa siempre debe coincidir con el tamaño de los datos de entrada. Además, asegúrese de que el número de características de salida sea 1 en la última capa: la salida de la red discriminadora es un solo valor. Usamos la función de activación sigmoidal para comprimir la salida al rango $[0, 1]$ de modo que pueda interpretarse como la probabilidad, p , de que la muestra sea real. Con la probabilidad complementaria, $1 - p$, la muestra es falsa. Esto es muy similar a lo que hicimos en el capítulo 2 cuando un clasificador binario intenta identificar una prenda de vestir como un botín o una camiseta.

Las capas ocultas contienen 256, 128 y 64 neuronas, respectivamente. Estos números no son mágicos, y se pueden modificar fácilmente para obtener resultados similares siempre que se mantengan dentro de un rango razonable. Si el número de neuronas en las capas ocultas es demasiado grande, puede provocar un sobreajuste del modelo; si es demasiado pequeño, puede provocar un subajuste. El número de neuronas se puede optimizar por separado mediante un conjunto de validación mediante el ajuste de hiperparámetros.

Las capas de abandono desactivan aleatoriamente (o "eliminan") un cierto porcentaje de neuronas en la capa a la que se aplican. Esto significa que estas neuronas no participan en los pasos hacia adelante ni hacia atrás durante el entrenamiento. El sobreajuste ocurre cuando un modelo aprende no solo los patrones subyacentes en los datos de entrenamiento, sino también el ruido y las fluctuaciones aleatorias, lo que resulta en un rendimiento deficiente con datos no detectados. Las capas de abandono son una forma eficaz de prevenir el sobreajuste.²

3.3.2 La red de generadores

La función del generador es crear un par de números (x, y) para que supere la prueba del discriminador. Es decir, el generador intenta crear un par de números para maximizar la probabilidad de que el discriminador considere que los números provienen del conjunto de datos de entrenamiento (es decir, que cumplen la relación $y = 1.08^x$). Creamos la red neuronal en la siguiente lista para representar al generador.

Listado 3.4 Creación de una red de generadores

```

G=nn.Sequential(
    nn.Linear(2,16),          ①
    nn.ReLU(),
    nn.Linear(16,32),
    nn.ReLU(),
    nn.Linear(32,2)).to(device) ②

```

① El número de características de entrada en la primera capa es 2, lo mismo que la dimensión del vector de ruido aleatorio del espacio latente.

② El número de características de salida en la última capa es 2, lo mismo que la dimensión de la muestra de datos, que contiene dos valores (x, y).

Alimentamos al generador con un vector de ruido aleatorio desde un espacio latente 2D (z_1, z_2). El generador genera un par de valores (x, y) basándose en la entrada del espacio latente. En este caso, usamos un espacio latente 2D, pero cambiar la dimensión a otros números, como 5 o 10, no afectaría nuestros resultados.

3.3.3 Funciones de pérdida, optimizadores y detención temprana

Dado que la red discriminadora realiza esencialmente una tarea de clasificación binaria (identificando una muestra de datos como real o falsa), utilizamos la pérdida de entropía cruzada binaria, la función de pérdida preferida en clasificaciones binarias, para la red discriminadora. El discriminador intenta maximizar la precisión de la clasificación binaria: identifica una muestra real como real y una muestra falsa como falsa. Los pesos en la red discriminadora se actualizan según el gradiente de la función de pérdida con respecto a los pesos.

El generador intenta minimizar la probabilidad de que la muestra falsa se identifique como falsa. Por lo tanto, también utilizaremos la pérdida de entropía cruzada binaria para la red del generador: el generador actualiza los pesos de su red para que el discriminador clasifique las muestras generadas como reales en un problema de clasificación binaria.

Como hicimos en el capítulo 2, usamos el optimizador Adam como algoritmo de descenso de gradiente. Establecemos la tasa de aprendizaje en 0,0005. Codifiquemos estos pasos con PyTorch:

```

loss_fn=nn.BCELoss()
lr=0.0005
optimD=torch.optim.Adam(D.parameters(),lr=lr)
optimG=torch.optim.Adam(G.parameters(),lr=lr)

```

Queda una pregunta antes de llegar al entrenamiento real: ¿Cuántas épocas debemos entrenar las GAN? ¿Cómo sabemos que el modelo está bien entrenado para que el generador esté listo para crear muestras que puedan imitar la forma de la curva de crecimiento exponencial? Si recuerda, en el capítulo 2, dividimos el conjunto de entrenamiento en un conjunto de entrenamiento y un conjunto de validación.

Luego, usamos la pérdida en el conjunto de validación para determinar si los parámetros habían convergido para que pudiéramos detener el entrenamiento. Sin embargo, las GAN se entran utilizando un enfoque diferente en comparación con los modelos tradicionales de aprendizaje supervisado (como los modelos de clasificación que ha visto en el capítulo 2). Dado que la calidad de las muestras generadas mejora a lo largo del entrenamiento, la tarea del discriminador se vuelve cada vez más difícil (de alguna manera, el discriminador en las GAN está haciendo predicciones sobre un objetivo en movimiento). La pérdida de la red discriminadora no es un buen indicador de la calidad del modelo.

Un método común para medir el rendimiento de las GAN es la inspección visual. Se puede evaluar la calidad y el realismo de las instancias de datos generadas con solo observarlas. Este es un enfoque cualitativo, pero puede ser muy informativo. En nuestro caso simple, dado que conocemos la distribución exacta del conjunto de datos de entrenamiento, analizaremos el MSE de las muestras generadas en relación con las muestras del conjunto de entrenamiento y lo usaremos como medida del rendimiento del generador. Codifiquemos esto:

```
mse=nn.MSELoss()                                     ①

def performance(fake_samples):
    real=1.08**fake_samples[:,0]                      ②
    mseloss=mse(fake_samples[:,1],real)               ③
    return mseloss
```

① Utiliza MSE como criterio para medir el desempeño

② Descubre la distribución verdadera

③ Compara la distribución generada con la distribución real y calcula MSE

Dejaremos de entrenar el modelo si el rendimiento del generador no mejora en, digamos, 1000 épocas. Por lo tanto, definimos una clase de detención temprana, como hicimos en el capítulo 2, para decidir cuándo detener el entrenamiento del modelo.

Listado 3.5 Una clase de parada temprana para decidir cuándo parar el entrenamiento

```

class EarlyStop:
    def __init__(self, patience=1000):          ①
        self.patience = patience
        self.steps = 0
        self.min_gdif = float('inf')
    def stop(self, gdif):                      ②
        if gdif < self.min_gdif:                ③
            self.min_gdif = gdif
            self.steps = 0
        elif gdif >= self.min_gdif:
            self.steps += 1
        if self.steps >= self.patience:        ④
            return True
        else:
            return False

stopper=EarlyStop()

```

① Establece el valor predeterminado de paciencia en 1000

② Define el método stop()

③ Si se alcanza una nueva diferencia mínima entre la distribución generada y la distribución verdadera, actualiza el valor de min_gdif.

④ Detiene el entrenamiento si el modelo deja de mejorar durante 1000 épocas

Con esto, tenemos todos los componentes que necesitamos para entrenar nuestras GAN, lo que haremos en la siguiente sección.

3.4 Entrenamiento y uso de GAN para la generación de formas

Ahora que tenemos los datos de entrenamiento y dos redes, entrenaremos el modelo. Después, descartaremos el discriminador y usaremos el generador para generar puntos de datos y formar una curva de crecimiento exponencial.

3.4.1 El entrenamiento de las GAN

Primero, creamos etiquetas para las muestras reales y falsas, respectivamente. Específicamente, etiquetaremos todas las muestras reales como 1 y todas las falsas como 0. Durante el proceso de entrenamiento, el discriminador compara sus propias predicciones con las etiquetas para recibir retroalimentación y ajustar los parámetros del modelo para obtener mejores predicciones en la siguiente iteración.

Aquí definimos dos tensores `real_labels` y `fake_labels`:

```
real_labels=torch.ones((batch_size,1))
real_labels=real_labels.to(device)

fake_labels=torch.zeros((batch_size,1))
fake_labels=fake_labels.to(device)
```

El tensor `real_labels` es bidimensional con una forma de `(batch_size, 1)`, es decir, 128 filas y 1 columna. Usamos 128 filas porque alimentaremos un lote de 128 muestras reales a la red discriminadora para obtener 128 predicciones. De forma similar, el tensor `fake_labels` es bidimensional con una forma de `(batch_size, 1)`. Alimentaremos un lote de 128 muestras falsas a la red discriminadora para obtener 128 predicciones y compararlas con la realidad fundamental: 128 etiquetas de 0. Transferimos los dos tensores a la GPU para un entrenamiento rápido si su computadora tiene una GPU compatible con CUDA.

Para entrenar las GAN, definimos algunas funciones para que el bucle de entrenamiento se vea organizado. La primera función, `train_D_on_real()`, entrena la red discriminadora con un lote de muestras reales.

Listado 3.6 Definición de una función `train_D_on_real()`

```
def train_D_on_real(real_samples):
    real_samples=real_samples.to(device)
    optimD.zero_grad()
    out_D=D(real_samples)                      ①
    loss_D=loss_fn(out_D,real_labels)            ②
    loss_D.backward()
    optimD.step()                                ③
    return loss_D
```

① Hace predicciones sobre muestras reales

② Calcula la pérdida

③ Retropropagación (es decir, actualiza los pesos del modelo en la red discriminadora para que las predicciones sean más precisas en la siguiente iteración)

La función `train_D_on_real()` primero transfiere las muestras reales a la GPU si el ordenador tiene una GPU compatible con CUDA. La red discriminadora, D, realiza predicciones sobre el lote de muestras. A continuación, el modelo compara las predicciones del discriminador, `out_D`, con la realidad fundamental, `real_labels`, y calcula la pérdida de las predicciones en consecuencia. El método `backward()` cal-

cula los gradientes de la función de pérdida con respecto a los parámetros del modelo. El método `step()` ajusta los parámetros del modelo (es decir, retropropagación). El método `zero_grad()` implica que establecemos explícitamente los gradientes en 0 antes de la retropropagación. De lo contrario, se utilizan los gradientes acumulados en lugar de los gradientes incrementales en cada llamada `backward()`.

CONSEJO: Llamamos al método `zero_grad()` antes de actualizar los pesos del modelo al entrenar cada lote de datos. Establecemos explícitamente los gradientes a 0 antes de la retropropagación para usar gradientes incrementales en lugar de los gradientes acumulados en cada llamada `backward()`.

La segunda función, `train_D_on_fake()`, entrena la red discriminadora con un lote de muestras falsas.

Listado 3.7 Definición de la función `train_D_on_fake()`

```
def train_D_on_fake():
    noise=torch.randn((batch_size,2))
    noise=noise.to(device)
    fake_samples=G(noise)          ①
    optimD.zero_grad()
    out_D=D(fake_samples)          ②
    loss_D=loss_fn(out_D,fake_labels) ③
    loss_D.backward()
    optimD.step()                  ④
    return loss_D
```

① Genera un lote de muestras falsas

② Hace predicciones sobre las muestras falsas

③ Calcula la pérdida

④ Retropropagación

La función `train_D_on_fake()` primero alimenta un lote de vectores de ruido aleatorio del espacio latente al generador para obtener un lote de muestras falsas. A continuación, presenta las muestras falsas al discriminador para obtener predicciones. La función compara las predicciones del discriminador, `out_D`, con la realidad fundamental, `fake_labels`, y calcula la pérdida de las predicciones en consecuencia. Finalmente, ajusta los parámetros del modelo en función de los gradientes de la función de pérdida con respecto a los pesos del modelo.

NOTA: Usamos los términos *ponderaciones* y *parámetros* indistintamente. En sentido estricto, los parámetros del modelo también incluyen términos de sesgo, pero usa-

mos el término *ponderaciones del modelo* de forma imprecisa para incluir los sesgos del modelo. De igual forma, usamos los términos *ponderaciones de ajuste, parámetros de ajuste y retropropagación* indistintamente.

La tercera función, `train_G()`, entrena la red generadora con un lote de muestras falsas.

Listado 3.8 Definición de la función `train_G()`

```
def train_G():
    noise=torch.randn((batch_size,2))
    noise=noise.to(device)
    optimG.zero_grad()
    fake_samples=G(noise)          ①
    out_G=D(fake_samples)          ②
    loss_G=loss_fn(out_G,real_labels) ③
    loss_G.backward()
    optimG.step()                  ④
    return loss_G, fake_samples
```

- ① Crea un lote de muestras falsas
- ② Presenta las muestras falsas al discriminador para obtener predicciones
- ③ Calcula la pérdida en función de si G ha tenido éxito
- ④ Retropropagación (es decir, actualiza los pesos en la red del generador para que las muestras generadas sean más realistas en la siguiente iteración)

Para entrenar el generador, primero le introducimos un lote de vectores de ruido aleatorio del espacio latente para obtener un lote de muestras falsas. Luego, presentamos las muestras falsas a la red discriminadora para obtener un lote de predicciones. Comparamos las predicciones del discriminador con `real_labels`, un tensor de unos, y calculamos la pérdida. Es importante usar un tensor de unos, no de ceros, como etiquetas, ya que el objetivo del generador es engañar al discriminador para que piense que las muestras falsas son reales. Finalmente, ajustamos los parámetros del modelo en función de los gradientes de la función de pérdida con respecto a los pesos del modelo para que, en la siguiente iteración, el generador pueda crear muestras más realistas.

NOTA: Utilizamos el tensor `real_labels` (un tensor de 1s) en lugar de `fake_labels` (un tensor de 0s) al calcular la pérdida y evaluar la red del generador porque el generador quiere que el discriminador prediga muestras falsas como reales.

Finalmente, definimos una función, `test_epoch()` que imprime periódicamente las pérdidas del discriminador y del generador. Además, grafica los puntos de datos generados por el generador y los compara con los del conjunto de entrenamiento. La función `test_epoch()` se muestra en la siguiente lista.

Listado 3.9 Definición de la función `test_epoch()`

```
import os  
os.makedirs("files", exist_ok=True) ①  
  
def test_epoch(epoch,gloss,dloss,n,fake_samples):  
    if epoch==0 or (epoch+1)%25==0:  
        g=gloss.item()/n  
        d=dloss.item()/n  
        print(f"at epoch {epoch+1}, G loss: {g}, D loss {d}") ②  
        fake=fake_samples.detach().cpu().numpy()  
        plt.figure(dpi=200)  
        plt.plot(fake[:,0],fake[:,1],"*",c="g",  
                 label="generated samples") ③  
        plt.plot(train_data[:,0],train_data[:,1],".",c="r",  
                 alpha=0.1,label="real samples") ④  
        plt.title(f"epoch {epoch+1}")  
        plt.xlim(0,50)  
        plt.ylim(0,50)  
        plt.legend()  
        plt.savefig(f"files/p{epoch+1}.png")  
        plt.show()
```

① Crea una carpeta para guardar archivos

② Imprime periódicamente las pérdidas

③ Traza los puntos generados como asteriscos (*)

④ Traza los datos de entrenamiento como puntos (.)

Después de cada 25 épocas, la función imprime las pérdidas promedio del generador y el discriminador en la época. Además, grafica un lote de puntos de datos falsos generados por el generador (en asteriscos) y los compara con los puntos de datos del conjunto de entrenamiento (en puntos). El gráfico se guarda como imagen en la carpeta local `/files/`.

Ahora estamos listos para entrenar el modelo. Iteramos todos los lotes del conjunto de datos de entrenamiento. Para cada lote de datos, primero entrenamos el discriminador con las muestras reales. Después, el generador crea un lote de muestras falsas y las usamos para entrenar el discriminador de nuevo. Finalmente, dejamos que el

generador cree un lote de muestras falsas de nuevo, pero esta vez las usamos para entrenarlo. Entrenamos el modelo hasta que se cumpla la condición de parada anticipada, como se muestra en la siguiente lista.

Listado 3.10 Entrenamiento de GAN para generar una curva de crecimiento exponencial

```
for epoch in range(10000):                                ①
    gloss=0
    dloss=0
    for n, real_samples in enumerate(train_loader):        ②
        loss_D=train_D_on_real(real_samples)
        dloss+=loss_D
        loss_D=train_D_on_fake()
        dloss+=loss_D
        loss_G,fake_samples=train_G()
        gloss+=loss_G
    test_epoch(epoch,gloss,dloss,n,fake_samples)           ③
    gdif=performance(fake_samples).item()
    if stopper.stop(gdif)==True:                            ④
        break
```

① Inicia bucles de entrenamiento

② Itera a través de todos los lotes en el conjunto de datos de entrenamiento

③ Muestra muestras generadas periódicamente

④ Determina si el entrenamiento debe detenerse

El entrenamiento toma unos minutos si usas entrenamiento con GPU. De lo contrario, puede tardar entre 20 y 30 minutos, dependiendo de la configuración de hardware de tu computadora. También puedes descargar el modelo entrenado del repositorio de GitHub del libro: <https://github.com/markhliu/DGAI>.

Tras 25 épocas de entrenamiento, los datos generados se dispersan alrededor del punto (0,0) y no forman una forma significativa (una época es cuando todos los datos de entrenamiento se utilizan una sola vez). Tras 200 épocas de entrenamiento, los puntos de datos empiezan a formar una curva de crecimiento exponencial, aunque muchos puntos están lejos de la curva de puntos, formada por puntos del conjunto de entrenamiento. Tras 1025 épocas, los puntos generados se ajustan perfectamente a la curva de crecimiento exponencial. La Figura 3.2 muestra subgráficos de los resultados de seis épocas diferentes. Nuestras GAN funcionan de maravilla: el generador puede generar puntos de datos para formar la forma deseada.

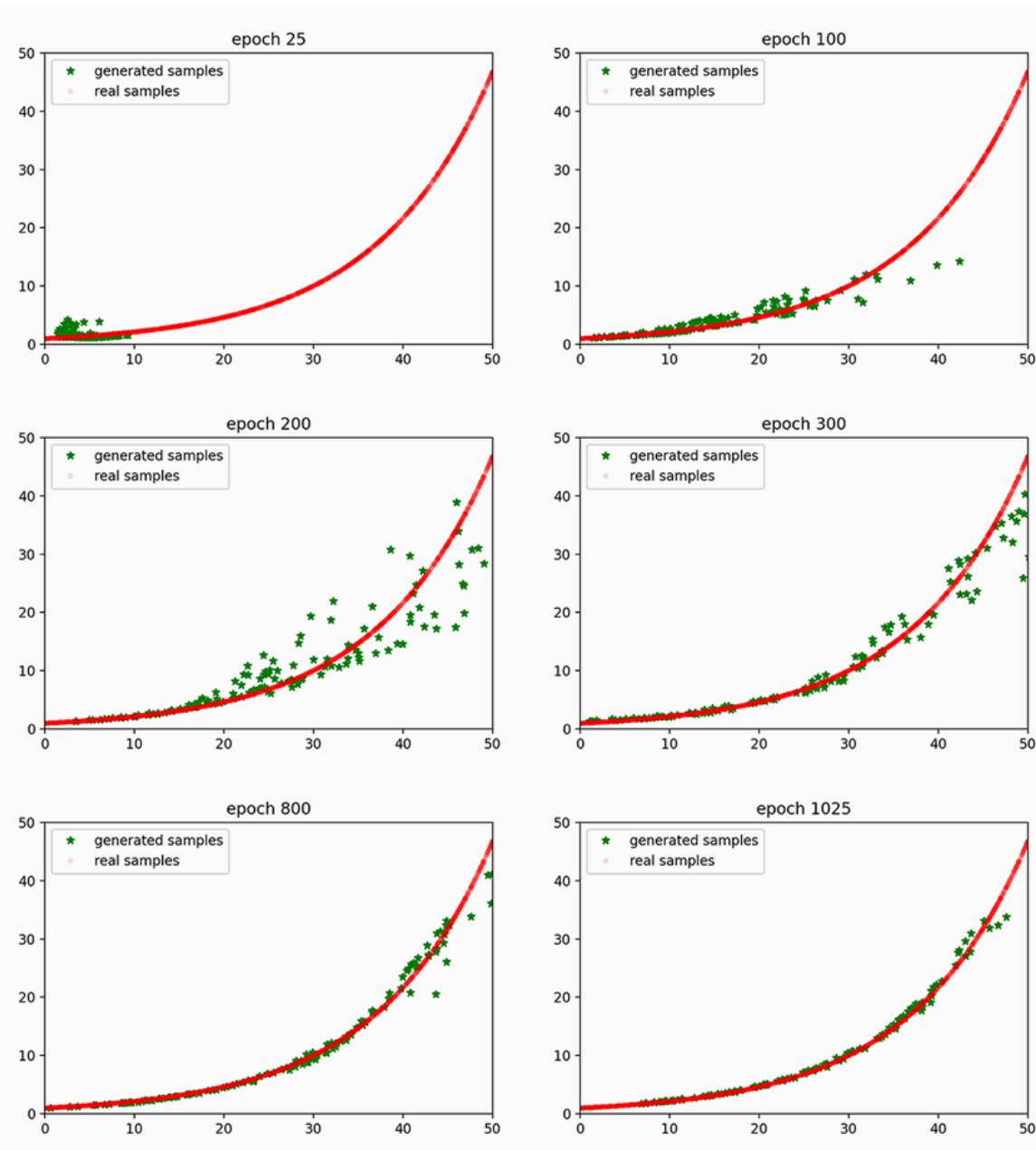


Figura 3.2 Subgráficos de la comparación de la forma generada (asteriscos en el gráfico) con la forma real de la curva de crecimiento exponencial (puntos en el gráfico) en diferentes etapas del proceso de entrenamiento. En la época 25, las muestras generadas no presentan una forma significativa. En la época 200, las muestras empiezan a asemejarse a una curva de crecimiento exponencial. En la época 1025, las muestras generadas se alinean estrechamente con la curva de crecimiento exponencial.

3.4.2 Guardar y usar el generador entrenado

Ahora que las GAN están entrenadas, descartaremos la red discriminadora, como siempre hacemos en las GAN, y guardaremos la red generadora entrenada en la carpeta local, de la siguiente manera:

```
import os
os.makedirs("files", exist_ok=True)
scripted = torch.jit.script(G)
scripted.save('files/exponential.pt')
```

El método `torch.jit.script()` crea una función o clase `nn.Module` como código TorchScript mediante el compilador de TorchScript. Usamos este método para crear

la red de generadores entrenada y guardarla como archivo `exponential.pt` en su computadora.

Para usar el generador, ni siquiera necesitamos definir el modelo. Simplemente cargamos el archivo guardado y lo usamos para generar puntos de datos como se indica a continuación:

```
new_G=torch.jit.load('archivos/exponential.pt',
                     map_location=dispositivo)
nuevo_G.eval()
```

El generador entrenado ya está cargado en tu dispositivo, que puede ser `CPU` o `CUDA` depende de si tienes una GPU compatible con CUDA en tu ordenador. El argumento `map_location=device` en `torch.jit.load()` especifica dónde cargar el generador. Ahora podemos usar el generador entrenado para generar un lote de puntos de datos:

```
noise=torch.randn((batch_size,2)).to(device)
new_data=new_G(noise)
```

Aquí, primero obtenemos un lote de vectores de ruido aleatorio del espacio latente. Luego, los alimentamos al generador para generar los datos falsos. Podemos graficar los datos generados:

```
fig=plt.figure(dpi=100)
plt.plot(new_data.detach().cpu().numpy()[:,0],
         new_data.detach().cpu().numpy()[:,1],"*",c="g",
         label="generated samples")                               ①
plt.plot(train_data[:,0],train_data[:,1],".",c="r",
         alpha=0.1,label="real samples")                      ②
plt.title("Inverted-U Shape Generated by GANs")
plt.xlim(0,50)
plt.ylim(0,50)
plt.legend()
plt.show()
```

① Traza las muestras de datos generadas como asteriscos

② Traza los datos de entrenamiento como puntos

Debería ver un gráfico similar al último subgráfico de la figura 3.2: las muestras de datos generadas se parecen mucho a una curva de crecimiento exponencial.

¡Felicitaciones! Has creado y entrenado tus primeras GAN. Con esta habilidad, puedes modificar fácilmente el código para que los datos generados coincidan con otras formas, como seno, coseno, forma de U, etc.

Ejercicio 3.3

Modifique los programas del primer proyecto para que el generador genere muestras de datos que formen una forma sinusoidal entre $x = -5$ y $x = 5$. Al graficar las muestras de datos, establezca el valor de y entre $-1,2$ y $1,2$.

3.5 Generación de números con patrones

En este segundo proyecto, construirás y entrenarás GAN para generar una secuencia de 10 enteros entre 0 y 99, todos múltiplos de 5. Los pasos principales son similares a los de generar una curva de crecimiento exponencial, con la excepción de que el conjunto de entrenamiento no son puntos de datos con dos valores (x, y). En su lugar, el conjunto de datos de entrenamiento es una secuencia de enteros, todos múltiplos de 5, entre 0 y 99.

En esta sección, primero aprenderá a convertir los datos de entrenamiento a un formato que las redes neuronales comprendan: variables one-hot. Además, las convertirá de nuevo a un número entero entre 0 y 99, de fácil comprensión para las personas. Por lo tanto, básicamente, estará traduciendo datos entre formatos legibles para humanos y compatibles con modelos. Después, creará un discriminador y un generador, y entrenará las GAN. También usará la detención temprana para determinar cuándo finaliza el entrenamiento. A continuación, descartará el discriminador y usará el generador entrenado para crear una secuencia de números enteros con el patrón deseado.

3.5.1 ¿Qué son las variables one-hot?

La codificación one-hot es una técnica utilizada en aprendizaje automático y preprocesamiento de datos para representar datos categóricos como vectores binarios. Los datos categóricos consisten en categorías o etiquetas, como colores, tipos de animales o ciudades, que no son inherentemente numéricas. Los algoritmos de aprendizaje automático suelen trabajar con datos numéricos, por lo que es necesario convertirlos a un formato numérico.

Imagina que trabajas con una característica categórica; por ejemplo, el color de una casa que puede tomar los valores "rojo", "verde" y "azul". Con la codificación one-hot, cada categoría se representa como un vector binario. Crearás tres columnas binarias, una para cada categoría. El color "rojo" se codifica one-hot como [1, 0, 0], el "verde" como [0, 1, 0] y el "azul" como [0, 0, 1]. De esta forma, se conserva la informa-

ción categórica sin introducir ninguna relación ordinal entre las categorías. Cada categoría se trata como independiente.

Aquí definimos una función `onehot_encoder()` para convertir un entero en una variable one-hot:

```
import torch
def onehot_encoder(position,depth):
    onehot=torch.zeros((depth,))
    onehot[position]=1
    return onehot
```

La función acepta dos argumentos: el primero, `position`, es el índice en el que el valor se activa como 1, y el segundo, `depth`, es la longitud de la variable one-hot. Por ejemplo, si imprimimos el valor de `onehot_encoder(1,5)`, se verá así:

```
print(onehot_encoder(1,5))
```

El resultado es

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

El resultado muestra un tensor de cinco valores con el segundo lugar (cuyo valor de índice es 1) activado como 1 y el resto desactivado como 0.

Ahora que comprende cómo funciona la codificación one-hot, puede convertir cualquier número entero entre 0 y 99 en una variable one-hot:

```
def int_to_onehot(number):
    onehot=onehot_encoder(number,100)
    return onehot
```

Utilicemos la función para convertir el número 75 en un tensor de valor 100:

```
onehot75=int_to_onehot(75)  
print(onehot75)
```

La salida es

```
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
0., 0., 1., 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.,
```

El resultado es un tensor de valor 100 con el lugar 76 (cuyo valor de índice es 75) activado como 1 y todas las demás posiciones desactivadas como 0.

La función `int_to_onehot()` convierte un entero en una variable one-hot. En cierto modo, traduce un lenguaje legible para humanos a un lenguaje compatible con modelos.

A continuación, queremos traducir el lenguaje compatible con modelos a un lenguaje legible para humanos. Supongamos que tenemos una variable one-hot: ¿Cómo podemos convertirla en un entero comprensible para humanos? La siguiente función `onehot_to_int()` logra este objetivo:

```
def onehot_to_int(onehot):  
    num=torch.argmax(onehot)  
    return num.item()
```

La función `onehot_to_int()` toma el argumento `onehot` y lo convierte en un entero según qué posición tenga el valor más alto.

Probemos la función para ver qué sucede si usamos el tensor `onehot75` que acabamos de crear como entrada:

```
print(onehot_to_int(onehot75))
```

La salida es

```
75
```

El resultado muestra que la función convierte la variable one-hot en un entero 75, que es la respuesta correcta. Por lo tanto, sabemos que las funciones están definidas correctamente.

A continuación, construiremos y entrenaremos GAN para generar múltiplos de 5.

3.5.2 GAN para generar números con patrones

Nuestro objetivo es construir y entrenar un modelo para que el generador pueda generar una secuencia de 10 enteros, todos múltiplos de 5. Primero, preparamos los datos de entrenamiento y luego los convertimos en números listos para el modelo

por lotes. Finalmente, usamos el generador entrenado para generar los patrones deseados.

Para simplificar, generaremos una secuencia de 10 números enteros entre 0 y 99. Luego, convertiremos la secuencia en 10 números listos para el modelo.

La siguiente función genera una secuencia de 10 números enteros, todos múltiplos de 5:

```
def gen_sequence():
    indices = torch.randint(0, 20, (10,))
    values = indices*5
    return values
```

Primero usamos el método `randint()` en PyTorch para generar 10 números entre 0 y 19. Luego los multiplicamos por 5 y los convertimos en tensores de PyTorch. Esto crea 10 enteros que son múltiplos de 5.

Intentemos generar una secuencia de datos de entrenamiento:

```
sequence=gen_sequence()
print(sequence)
```

La salida es

```
tensor([60, 95, 50, 55, 25, 40, 70, 5, 0, 55])
```

Los valores en la salida anterior son todos múltiplos de 5.

A continuación, convertimos cada número en una variable one-hot para que podamos alimentarlos a la red neuronal más tarde:

```
import numpy as np

def gen_batch():
    sequence=gen_sequence()                               ①
    batch=[int_to_onehot(i).numpy() for i in sequence]  ②
    batch=np.array(batch)
    return torch.tensor(batch)
batch=gen_batch()
```

① Crea una secuencia de 10 números, todos múltiplos de 5

② Convierte cada entero en una variable one-hot de 100 valores

La función anterior `gen_batch()` crea un lote de datos para que podamos alimentar a la red neuronal con fines de entrenamiento.

También definimos una función `data_to_num()` para convertir variables one-hot en una secuencia de números enteros para que los humanos puedan entender el resultado:

```
def data_to_num(data):
    num=torch.argmax(data,dim=-1) ①
    return num
numbers=data_to_num(batch) ②
```

① Convierte vectores en números enteros según los valores más grandes en un vector de 100 valores

② Aplica la función en un ejemplo

El argumento `dim=-1` en la función `torch.argmax()` significa que estamos tratando de encontrar la posición (es decir, el índice) del valor más grande en la última dimensión: es decir, entre el vector one-hot de 100 valores, qué posición tiene el valor más alto.

A continuación, crearemos dos redes neuronales: una para el discriminador D y otra para el generador G. Construiremos redes neuronales generativas (GAN) para generar el patrón de números deseado. De forma similar a lo que hicimos anteriormente en este capítulo, crearemos una red discriminadora, que es un clasificador binario que distingue las muestras falsas de las reales. También crearemos una red generadora para generar una secuencia de 10 números. Aquí está la red neuronal discriminadora:

```
from torch import nn
D=nn.Sequential(
    nn.Linear(100,1),
    nn.Sigmoid()).to(device)
```

Dado que convertiremos enteros en variables one-hot de 100 valores, usamos 100 como tamaño de entrada en la primera capa `Linear` del modelo. La última capa `Linear` tiene solo una característica de salida, y usamos la función de activación sigmoida para comprimir la salida al rango [0, 1], de modo que pueda interpretarse como la probabilidad, p, de que la muestra sea real. Con la probabilidad complementaria 1 – p, la muestra es falsa.

El trabajo del generador es crear una secuencia de números para que puedan pasar como reales frente al discriminador D. Es decir, G está intentando crear una secuencia de números para maximizar la probabilidad de que D piense que los números son del conjunto de datos de entrenamiento.

Creamos la siguiente red neuronal para representar el generador G:

```
G=nn.Sequential(  
    nn.Linear(100,100),  
    nn.ReLU()).to(device)
```

Introduciremos vectores de ruido aleatorio desde un espacio latente de 100 dimensiones al generador. El generador crea un tensor de 100 valores basado en la entrada. Observe que usamos la función `ReLU` de activación en la última capa para que la salida sea no negativa. Dado que intentamos generar 100 valores de 0 o 1, los valores no negativos son apropiados en este caso.

Al igual que en el primer proyecto, utilizamos el optimizador Adam tanto para el discriminador como para el generador, con una tasa de aprendizaje de 0,0005:

```
loss_fn=nn.BCELoss()  
lr=0.0005  
optimD=torch.optim.Adam(D.parameters(),lr=lr)  
optimG=torch.optim.Adam(G.parameters(),lr=lr)
```

Ahora que tenemos los datos de entrenamiento y dos redes, entrenaremos el modelo. Después, descartaremos el discriminador y usaremos el generador para generar una secuencia de 10 enteros.

3.5.3 Entrenamiento de las GAN para generar números con patrones

El proceso de entrenamiento para este proyecto es muy similar al de nuestro primer proyecto en el que se generó una forma de crecimiento exponencial.

Definimos una función `train_D_G()`, que combina las tres funciones `train_D_on_real()`, `train_D_on_fake()` y `train_G()` que definimos para el primer proyecto. Esta función `train_D_G()` se encuentra en el Jupyter Notebook de este capítulo, en el repositorio de GitHub del libro:

<https://github.com/markhliu/DGAI>. Echa un vistazo a la función `train_D_G()` para ver los pequeños cambios que hemos realizado en comparación con las tres funciones que definimos para el primer proyecto.

Usamos la misma clase de detención temprana que definimos para el primer proyecto para saber cuándo detener el entrenamiento. Sin embargo, modificamos el argumento `patience` a 800 al instanciar la clase, como se muestra en la siguiente lista.

Listado 3.11 Entrenamiento de GAN para generar múltiplos de 5

```
stopper=EarlyStop(800)                                ①

mse=nn.MSELoss()
real_labels=torch.ones((10,1)).to(device)
fake_labels=torch.zeros((10,1)).to(device)
def distance(generated_data):                         ②
    nums=data_to_num(generated_data)
    remainders=nums%5
    ten_zeros=torch.zeros((10,1)).to(device)
    mseloss=mse(remainders,ten_zeros)
    return mseloss

for i in range(10000):
    gloss=0
    dloss=0
    generated_data=train_D_G(D,G,loss_fn,optimD,optimG) ③
    dis=distance(generated_data)
    if stopper.stop(dis)==True:
        break
    if i % 50 == 0:
        print(data_to_num(generated_data))                ④
```

① Crea una instancia de la clase de detención temprana

② Define una función de `distancia()` para calcular la pérdida en los números generados

③ Entrena las GAN para una época

④ Imprime la secuencia generada de números enteros después de cada 50 épocas

También hemos definido una función `distance()` para medir la diferencia entre el conjunto de entrenamiento y las muestras de datos generadas: calcula el MSE del resto de cada número generado cuando se divide por 5. La medida es 0 cuando todos los números generados son múltiplos de 5.

Si ejecuta la celda de código anterior, verá el siguiente resultado:

```
tensor([14, 34, 19, 89, 44, 5, 58, 6, 41, 87], device='cuda:0')
...  
...
```

```
tensor([ 0, 80, 65, 0, 0, 10, 80, 75, 75, 75], device='cuda:0')
tensor([25, 30, 0, 0, 65, 20, 80, 20, 80, 20], device='cuda:0')
tensor([65, 95, 10, 65, 75, 20, 20, 20, 65, 75], device='cuda:0')
```

En cada iteración, generamos un lote de 10 números. Primero, entrenamos el discriminador D con muestras reales. Después, el generador crea un lote de muestras falsas y las usamos para entrenar de nuevo el discriminador D. Finalmente, dejamos que el generador cree de nuevo un lote de muestras falsas, pero las usamos para entrenar el generador G. Detenemos el entrenamiento si la red del generador deja de mejorar después de 800 épocas desde la última vez que se alcanzó la pérdida mínima. Después de cada 50 épocas, imprimimos la secuencia de 10 números creada por el generador para que se pueda determinar si todos son múltiplos de 5.

El resultado durante el proceso de entrenamiento es el mismo que se mostró anteriormente. En los primeros cientos de épocas, el generador aún produce números que no son múltiplos de 5. Sin embargo, después de 900 épocas, todos los números generados son múltiplos de 5. El proceso de entrenamiento toma aproximadamente un minuto con entrenamiento en GPU. Toma menos de 10 minutos con entrenamiento en CPU. Como alternativa, puede descargar el modelo entrenado del repositorio de GitHub del libro: <https://github.com/markliu/DGAI>.

3.5.4 Guardar y usar el modelo entrenado

Descartaremos el discriminador y guardaremos el generador entrenado en la carpeta local:

```
import os
os.makedirs("files", exist_ok=True)
scripted = torch.jit.script(G)
scripted.save('files/num_gen.pt')
```

Hemos guardado el generador en la carpeta local. Para usarlo, simplemente cargamos el modelo y lo usamos para generar una secuencia de enteros:

```
new_G=torch.jit.load('files/num_gen.pt',
                     map_location=device) ①
new_G.eval()
noise=torch.randn((10,100)).to(device) ②
new_data=new_G(noise)
print(data_to_num(new_data)) ③
```

① Carga el generador guardado

② Obtiene vectores de ruido aleatorio

- ③ Alimenta los vectores de ruido aleatorio al modelo entrenado para generar una secuencia de números enteros

El resultado es el siguiente:

```
tensor([40, 25, 65, 25, 20, 25, 95, 10, 10, 65], device='cuda:0')
```

Los números generados son todos múltiplos de 5.

Puede cambiar fácilmente el código para generar otros patrones, como números impares, números pares, múltiplos de 3, etc.

Ejercicio 3.4

Modifique los programas en el segundo proyecto para que el generador genere una secuencia de diez números enteros que sean todos múltiplos de 3.

Ahora que sabe cómo funcionan las GAN, podrá ampliar la idea detrás de las GAN a otros formatos en capítulos posteriores, incluidas imágenes de alta resolución y música con sonido realista.

Resumen

- Las GAN constan de dos redes: un discriminador para distinguir muestras falsas de muestras reales y un generador para crear muestras que sean indistinguibles de las del conjunto de entrenamiento.
- Los pasos involucrados en las GAN son preparar datos de entrenamiento, crear un discriminador y un generador, entrenar el modelo y decidir cuándo detener el entrenamiento y, finalmente, descartar el discriminador y usar el generador entrenado para crear nuevas muestras.
- El contenido generado por las GAN depende de los datos de entrenamiento. Cuando el conjunto de datos de entrenamiento contiene pares de datos (x, y) que forman una curva de crecimiento exponencial, las muestras generadas también son pares de datos que imitan dicha forma. Cuando el conjunto de datos de entrenamiento contiene secuencias de números que son múltiplos de 5, las muestras generadas también son secuencias de números con múltiplos de 5 en ellas.
- Las GAN son versátiles y capaces de generar muchos formatos diferentes de contenido.

¹ Goodfellow et al, 2014, “Redes generativas antagónicas”.

<https://arxiv.org/abs/1406.2661>.

² Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever y Ruslan Salakhutdinov, 2014, “Abandono: una forma sencilla de evitar el sobreajuste de las redes neuronales”. *Journal of Machine Learning Research* 15 (56): 1929–1958.

Parte 2. Generación de imágenes

La segunda parte se adentra profundamente en la generación de imágenes.

En el capítulo 4, aprenderá a construir y entrenar redes generativas antagónicas para generar imágenes en color de alta resolución. En particular, aprenderá a usar redes neuronales convolucionales para capturar características espaciales en imágenes. También aprenderá a usar capas convolucionales transpuestas para sobremuestrear y generar mapas de características de alta resolución en imágenes. En el capítulo 5, aprenderá dos maneras de seleccionar características en las imágenes generadas. En el capítulo 6, aprenderá a construir y entrenar un CycleGAN para traducir imágenes entre dos dominios, como imágenes con cabello negro e imágenes con cabello rubio, o imágenes de caballos e imágenes de cebras. En el capítulo 7, aprenderá a crear imágenes utilizando otro modelo generativo: autocodificadores y su variante, los autocodificadores variacionales.

4. Generación de imágenes con redes generativas antagónicas

Este capítulo cubre

- Diseño de un generador reflejando los pasos en la red discriminadora
- Cómo funciona una operación convolucional 2D en una imagen
- Cómo una operación convolucional transpuesta 2D inserta espacios entre los valores de salida y genera mapas de características de mayor resolución
- Construcción y entrenamiento de redes generativas adversarias para generar imágenes en escala de grises y en color

Has generado con éxito una curva de crecimiento exponencial y una secuencia de números enteros múltiplos de 5 en el capítulo 3. Ahora que comprendes el funcionamiento de las redes generativas antagónicas (GAN), estás listo para aplicar las mismas habilidades para generar muchos otros tipos de contenido, como imágenes a color de alta resolución y música con sonido realista. Sin embargo, puede que sea más fácil decirlo que hacerlo (ya sabes lo que dicen: la clave está en los detalles). Por ejemplo, ¿cómo podemos lograr que el generador cree imágenes realistas de la nada? Esa es la pregunta que abordaremos en este capítulo.

Un enfoque común para que el generador cree imágenes desde cero consiste en replicar los pasos de la red discriminadora. En el primer proyecto de este capítulo, el objetivo es crear imágenes en escala de grises de prendas como abrigos, camisas, sandalias, etc. Aprenderá a replicar las capas de la red discriminadora al diseñar una red generadora. En este proyecto, solo se utilizan capas densas tanto en la red generadora como en la discriminadora. Cada neurona de una capa densa está conectada a todas las neuronas de la capa anterior y la siguiente. Por esta razón, las capas densas también se denominan capas completamente conectadas.

En el segundo proyecto de este capítulo, el objetivo es crear imágenes a color de alta resolución de rostros de anime. Al igual que en el primer pro-

yecto, el generador replica los pasos de la red discriminadora para generar imágenes. Sin embargo, las imágenes a color de alta resolución de este proyecto contienen muchos más píxeles que las imágenes en escala de grises de baja resolución del primer proyecto. Si solo usamos capas densas, el número de parámetros del modelo aumenta enormemente. Esto, a su vez, hace que el aprendizaje sea lento e ineficaz. Por lo tanto, recurrimos a las redes neuronales convolucionales (CNN). En las CNN, cada neurona de una capa está conectada solo a una pequeña región de la entrada. Esta conectividad local reduce el número de parámetros, lo que aumenta la eficiencia de la red. Las CNN requieren menos parámetros que las redes completamente conectadas de tamaño similar, lo que se traduce en tiempos de entrenamiento más rápidos y menores costos computacionales. Las CNN también suelen ser más eficaces para capturar jerarquías espaciales en datos de imágenes, ya que las tratan como objetos multidimensionales en lugar de vectores unidimensionales.

Para prepararte para el segundo proyecto, te mostraremos cómo funcionan las operaciones convolucionales y cómo submuestrean las imágenes de entrada y extraen características espaciales. También aprenderás conceptos como el tamaño del filtro, el paso y el relleno de ceros, y cómo afectan el grado de submuestreo en las CNN. Mientras que la red discriminadora utiliza capas convolucionales, el generador las refleja mediante capas convolucionales transpuestas (también conocidas como capas de deconvolución o sobremuestreo). Aprenderás cómo se utilizan las capas convolucionales transpuestas para el sobremuestreo y así generar mapas de características de alta resolución.

En resumen, en este capítulo aprenderá a replicar los pasos de la red discriminadora para crear imágenes desde cero. Además, aprenderá cómo funcionan las capas convolucionales y las capas convolucionales transpuestas. Después de este capítulo, utilizará capas convolucionales y capas convolucionales transpuestas para crear imágenes de alta resolución en otros entornos más adelante en este libro (como en transferencias de características al entrenar un CycleGAN para convertir cabello rubio en cabello negro o...un autocodificador variacional [VAE] para generar imágenes de rostros humanos de alta resolución).

4.1 GAN para generar imágenes en escala de grises de prendas de vestir

Nuestro objetivo en el primer proyecto es entrenar un modelo para generar imágenes en escala de grises de prendas de vestir como sandalias, camisetas, abrigos y bolsos.

Al usar GAN para generar imágenes, siempre se comienza obteniendo los datos de entrenamiento. Luego, se crea una red discriminadora desde cero. Se replican los pasos de la red discriminadora al crear una red generadora. Finalmente, se entranan las GAN y se usa el modelo entrenado para generar imágenes. Veamos cómo funciona esto con un proyecto sencillo que crea imágenes en escala de grises de prendas de vestir.

4.1.1 Muestras de entrenamiento y el discriminador

Los pasos para preparar los datos de entrenamiento son similares a los del capítulo 2, con algunas excepciones que destacaré más adelante. Para ahorrar tiempo, omitiré los pasos que ya vimos en el capítulo 2 y los remitiré al repositorio de GitHub del libro. Siga los pasos en el Jupyter Notebook de este capítulo en el repositorio de GitHub del libro (<https://github.com/markhliu/DGAI>) para que pueda crear un iterador de datos con lotes.

Hay 60.000 imágenes en el conjunto de entrenamiento. En el capítulo 2, dividimos el conjunto de entrenamiento en un conjunto de entrenamiento y un conjunto de validación. Utilizamos la pérdida en el conjunto de validación para determinar si los parámetros habían convergido para poder detener el entrenamiento. Sin embargo, las GAN se entranan con un enfoque diferente al de los modelos tradicionales de aprendizaje supervisado (como los modelos de clasificación que se vieron en el capítulo 2). Dado que la calidad de las muestras generadas mejora durante el entrenamiento, la tarea del discriminador se vuelve cada vez más difícil. La pérdida de la red discriminadora no es un buen indicador de la calidad del modelo. La forma habitual de medir el rendimiento de las GAN es mediante inspección visual para evaluar la calidad y el realismo de las imágenes generadas. Potencialmente, podemos comparar la calidad de las muestras generadas con las muestras de entrenamiento y usar métodos como el Inception Score para evaluar el rendimiento de las GAN (ver, por ejemplo, “Pros and Cons of

GAN Evaluation Measures”, de Ali Borji, 2018, para una encuesta sobre varios métodos de evaluación de GAN; <https://arxiv.org/abs/1802.03446>). Sin embargo, los investigadores han documentado las debilidades de estas medidas (“A Note on the Inception Score”, de Shane Barratt y Rishi Sharma, 2018, demuestra que el inception score no proporciona una guía útil al comparar modelos; <https://arxiv.org/abs/1801.01973>). En este capítulo, usaremos inspecciones visuales para verificar la calidad de las muestras generadas periódicamente y determinar cuándo detener el entrenamiento.

La red discriminadora es un clasificador binario, similar al clasificador binario para prendas de vestir que analizamos en el capítulo 2. Aquí, el trabajo del discriminador es clasificar las muestras en reales o falsas.

Usamos PyTorch para crear la siguiente red neuronal discriminadora D:

```
import torch
import torch.nn as nn

device="cuda" if torch.cuda.is_available() else "cpu"
D=nn.Sequential(
    nn.Linear(784, 1024),           ①
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(1024, 512),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(512, 256),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(256, 1),
    nn.Sigmoid()).to(device)        ②
```

① La primera capa completamente conectada tiene 784 entradas y 1024 salidas.

② La última capa completamente conectada tiene 256 entradas y 1 salida.

El tamaño de entrada es 784 porque cada imagen en escala de grises tiene un tamaño de 28×28 píxeles en el conjunto de entrenamiento. Dado que las capas densas solo aceptan entradas unidimensionales, aplanamos las imágenes antes de alimentarlas al modelo. La capa de salida contiene una

sola neurona: la salida del discriminador D es un valor único. Utilizamos la función de activación sigmoidea para comprimir la salida al rango [0, 1], de modo que pueda interpretarse como la probabilidad, p, de que la muestra sea real. Con probabilidad complementariabilidad $1 - p$, la muestra es falsa.

Ejercicio 4.1

Modifique el discriminador D para que los números de salidas en las primeras tres capas sean 1000, 500 y 200 en lugar de 1024, 512 y 256. Asegúrese de que el número de salidas en una capa coincida con el número de entradas en la siguiente capa.

4.1.2 Un generador para crear imágenes en escala de grises

Si bien la red discriminadora es bastante fácil de crear, crear un generador que pueda generar imágenes realistas es un asunto diferente. Un enfoque común consiste en reflejar las capas utilizadas en la red discriminadora para crear un generador, como se muestra en la siguiente lista.

Listado 4.1 Diseño de un generador mediante la duplicación de capas en el discriminador

```
G=nn.Sequential(  
    nn.Linear(100, 256),          ①  
    nn.ReLU(),  
    nn.Linear(256, 512),          ②  
    nn.ReLU(),  
    nn.Linear(512, 1024),         ③  
    nn.ReLU(),  
    nn.Linear(1024, 784),         ④  
    nn.Tanh()).to(device)        ⑤
```

① La primera capa del generador es simétrica a la última capa del discriminador.

② La segunda capa del generador es simétrica a la segunda última capa del discriminador (los números de entradas y salidas tienen posiciones cambiadas).

- ③ La tercera capa del generador es simétrica a la tercera última capa del discriminador.
- ④ La última capa del generador es simétrica a la primera capa del discriminador.
- ⑤ Utiliza la activación Tanh() para que la salida esté entre -1 y 1 , lo mismo que los valores en las imágenes.

La Figura 4.1 muestra un diagrama de la arquitectura de las redes generadoras y discriminadoras en la GAN para generar imágenes en escala de grises de prendas de vestir. Como se muestra en la esquina superior derecha de la figura 4.1, una imagen aplanada en escala de grises del conjunto de entrenamiento, que contiene $28 \times 28 = 784$ píxeles, recorre secuencialmente cuatro capas densas en la red discriminadora, y la salida es la probabilidad de que la imagen sea real. Para crear una imagen, el generador utiliza las mismas cuatro capas densas, pero en orden inverso: obtiene un vector de ruido aleatorio de 100 valores del espacio latente (abajo a la izquierda en la figura 4.1) y lo alimenta a través de las cuatro capas densas. En cada capa, se invierten los números de *entradas* y *salidas* del discriminador y se utilizan como números de *entradas* y *salidas* en el generador. Finalmente, el generador genera un tensor de 784 valores, que puede transformarse en una imagen en escala de grises de 28×28 (arriba a la izquierda).

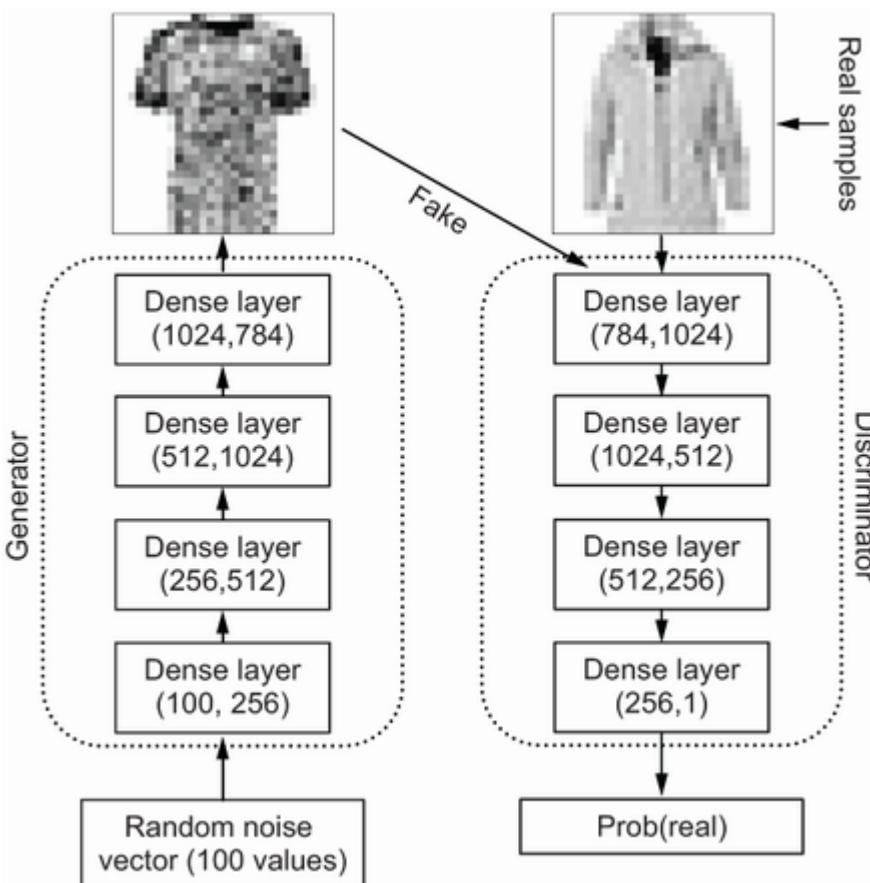


Figura 4.1 Diseño de una red generadora para crear prendas de vestir mediante la duplicación de las capas de la red discriminadora. El lado derecho del diagrama muestra la red discriminadora, que contiene cuatro capas densas. Para diseñar un generador capaz de crear prendas de vestir de la nada, duplicamos las capas de la red discriminadora. Específicamente, como se muestra en la mitad izquierda de la figura, el generador tiene cuatro capas densas similares, pero en orden inverso: la primera capa del generador duplica la última capa del discriminador, la segunda capa del generador duplica la penúltima capa del discriminador, y así sucesivamente. Además, en cada una de las tres capas superiores, el número de entradas y salidas del discriminador se invierte y se utiliza como el número de entradas y salidas del generador.

El lado izquierdo de la figura 4.1 corresponde a la red del generador, mientras que el lado derecho corresponde a la red del discriminador. Si compara ambas redes, observará cómo el generador refleja las capas utilizadas en el discriminador. En concreto, el generador contiene cuatro capas densas similares, pero en orden inverso: la primera capa del generador refleja la última capa del discriminador, la segunda capa del generador refleja la penúltima capa del discriminador, y así sucesivamente. El número de salidas del generador es 784, con valores entre -1 y 1 tras la activación $\text{Tanh}()$, lo que coincide con la entrada de la red del discriminador.

Ejercicio 4.2

Modifique el generador G para que los números de salidas en las primeras tres capas sean 1000, 500 y 200 en lugar de 1024, 512 y 256. Asegúrese de

que el generador modificado refleje las capas utilizadas en el discriminador modificado en el ejercicio 4.1.

Como en los modelos GAN que vimos en el capítulo 3, la función de pérdida es la pérdida de entropía cruzada binaria, ya que el discriminador D realiza un problema de clasificación binaria. Utilizaremos el optimizador Adam tanto para el discriminador como para el generador, con una tasa de aprendizaje de 0,0001:

```
loss_fn=nn.BCELoss()  
lr=0.0001  
optimD=torch.optim.Adam(D.parameters(), lr=lr)  
optimG=torch.optim.Adam(G.parameters(), lr=lr)
```

A continuación, entrenaremos las GAN que acabamos de crear utilizando las imágenes de las prendas de vestir en el conjunto de datos de entrenamiento.

4.1.3 Entrenamiento de GAN para generar imágenes de prendas de vestir

El proceso de entrenamiento es similar a lo que hicimos en el capítulo 3 cuando entrenamos GAN para generar una curva de crecimiento exponencial o para generar una secuencia de números que son todos múltiplos de 5.

A diferencia del capítulo 3, nos basaremos únicamente en inspecciones visuales para determinar si el modelo está bien entrenado. Para ello, definimos una función `see_output()` que visualiza periódicamente las imágenes falsas creadas por el generador.

NOTA: Los lectores interesados pueden consultar este repositorio de GitHub para aprender a implementar la puntuación de inicio en PyTorch para evaluar GAN: <https://github.com/sbarratt/inception-score-pytorch>. Sin embargo, el repositorio no recomienda usar la puntuación de inicio para evaluar modelos generativos debido a su ineeficacia.

Listado 4.2 Definición de una función para visualizar las prendas generadas

```
import matplotlib.pyplot as plt

def see_output():
    noise=torch.randn(32,100).to(device=device)
    fake_samples=G(noise).cpu().detach()                                ①
    plt.figure(dpi=100,figsize=(20,10))
    for i in range(32):
        ax=plt.subplot(4, 8, i + 1)                                     ②
        img=(fake_samples[i]/2+0.5).reshape(28, 28)
        plt.imshow(img)                                                 ③
        plt.xticks([])
        plt.yticks([])
    plt.show()

see_output()                                                        ④
```

① Genera 32 imágenes falsas

② Los traza en una cuadrícula de 4×8

③ Muestra la i -ésima imagen

④ Llama a la función `see_output()` para visualizar las imágenes generadas antes del entrenamiento

Si ejecuta la celda de código anterior, verá 32 imágenes que parecen copos de nieve en una pantalla de televisión, como se muestra en la figura 4.2. No parecen prendas de vestir en absoluto, ya que aún no hemos entrenado el generador.

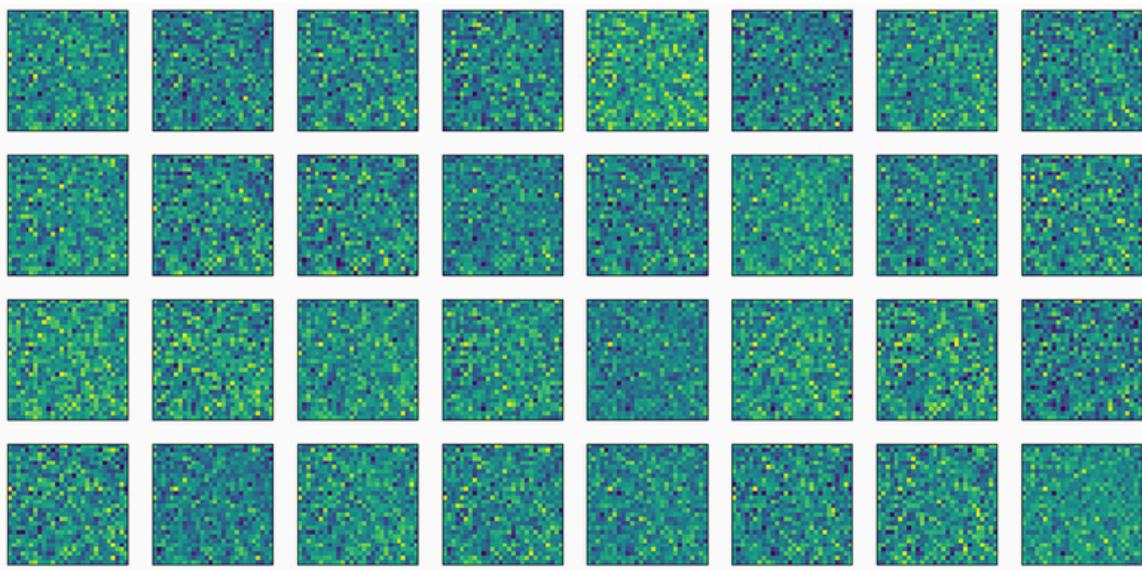


Figura 4.2 Salida del modelo GAN para generar prendas de vestir antes del entrenamiento. Dado que el modelo no está entrenado, las imágenes generadas no se parecen en nada a las del conjunto de entrenamiento.

Para entrenar el modelo GAN, definimos algunas funciones:

`train_D_on_real()`, `train_D_on_fake()`, y `train_G()`. Son similares a las definidas en el capítulo 3. Visite el Jupyter Notebook de este capítulo en el repositorio de GitHub del libro y vea las pequeñas modificaciones que hemos realizado.

Ahora estamos listos para entrenar el modelo. Iteramos todos los lotes del conjunto de datos de entrenamiento. Para cada lote de datos, primero entrenamos el discriminador con las muestras reales. Después, el generador crea un lote de muestras falsas y las usamos para entrenar el discriminador de nuevo. Finalmente, dejamos que el generador cree un lote de muestras falsas de nuevo, pero esta vez las usamos para entrenarlo. Entrenamos el modelo durante 50 épocas, como se muestra en la siguiente lista.

Listado 4.3 GAN de entrenamiento para la generación de prendas de vestir

```
for i in range(50):
    gloss=0
    dloss=0
    for n, (real_samples,_) in enumerate(train_loader):
        loss_D=train_D_on_real(real_samples)          ①
        dloss+=loss_D
        loss_D=train_D_on_fake()                     ②
        dloss+=loss_D
        loss_G=train_G()                           ③
        gloss+=loss_G
```

```
gloss=gloss/n  
dloss=dloss/n  
if i % 10 == 9:  
    print(f"at epoch {i+1}, dloss: {dloss}, gloss {gloss}")  
see_output()  
④
```

- ① Entrena el discriminador usando muestras reales
- ② Entrena al discriminador usando muestras falsas
- ③ Entrena el generador
- ④ Visualiza muestras generadas después de cada 10 épocas

El entrenamiento dura unos 10 minutos si usas entrenamiento con GPU. De lo contrario, puede tardar aproximadamente una hora, dependiendo de la configuración del hardware de tu computadora. También puedes descargar el entrenamiento del Modelo de mi sitio web:

https://gattonweb.uky.edu/faculty/lium/gai/fashion_gen.zip. Descomprímelo después de descargarlo.

Tras cada 10 épocas de entrenamiento, se pueden visualizar las prendas generadas, como se muestra en la figura 4.3. Tras tan solo 10 épocas de entrenamiento, el modelo ya puede generar prendas que claramente pueden considerarse reales: se puede identificar su identidad. Las tres primeras prendas de la primera fila de la figura 4.3 son, por ejemplo, un abrigo, un vestido y un pantalón. A medida que avanza el entrenamiento, la calidad de las imágenes generadas mejora cada vez más.

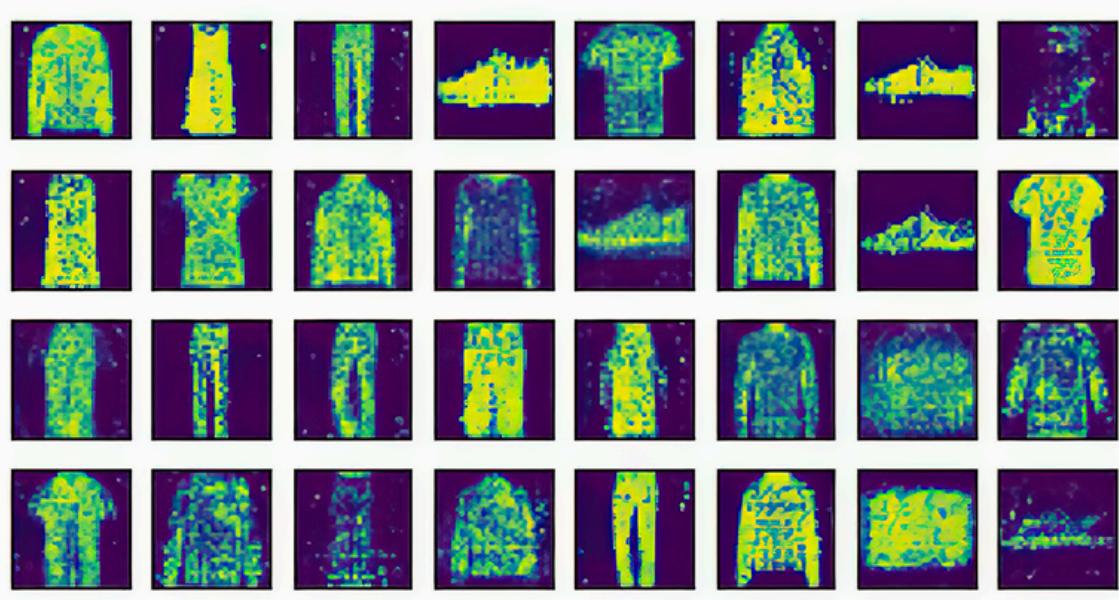


Figura 4.3 Prendas de vestir generadas por un modelo GAN de imágenes después de 10 épocas de entrenamiento

Como hacemos en todas las GAN, descartamos el discriminador y guardamos el generador entrenado para generar muestras más tarde:

```
scripted = torch.jit.script(G)
scripted.save('files/fashion_gen.pt')
```

Hemos guardado el generador en la carpeta local. Para usarlo, cargamos el modelo:

```
new_G=torch.jit.load('files/fashion_gen.pt',
                     map_location=device)
new_G.eval()
```

El generador ya está cargado. Podemos usarlo para generar prendas de vestir:

```
noise=torch.randn(32,100).to(device=device)
fake_samples=new_G(noise).cpu().detach()
for i in range(32):
    ax = plt.subplot(4, 8, i + 1)
    plt.imshow((fake_samples[i]/2+0.5).reshape(28, 28))
    plt.xticks([])
    plt.yticks([])
```

```
plt.subplots_adjust(hspace=-0.6)
plt.show()
```

Las prendas generadas se muestran en la figura 4.4. Como puede observar, son bastante similares a las del conjunto de entrenamiento.

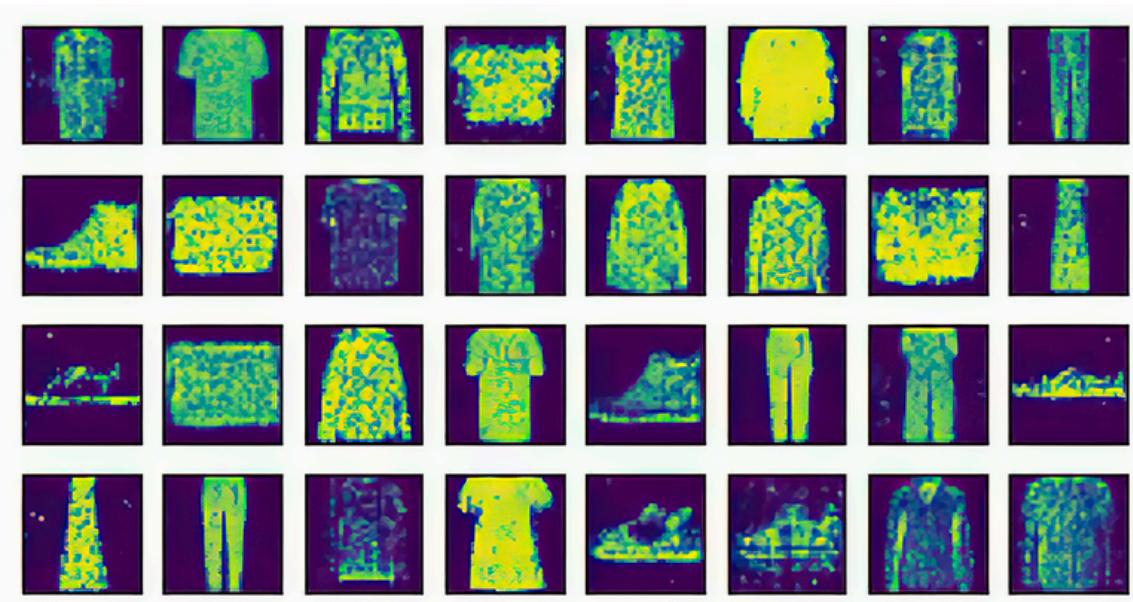


Figura 4.4 Prendas de vestir generadas por un modelo GAN de imágenes entrenado (después de 50 épocas)

Ahora que ha aprendido a crear imágenes en escala de grises mediante el uso de GAN, aprenderá a generar imágenes de alta resolución.imágenes en color mediante el uso de GAN convolucional profunda (DCGAN) en las secciones restantes de este capítulo.

4.2 Capas convolucionales

Para crear imágenes en color de alta resolución, necesitamos técnicas más sofisticadas que las simples redes neuronales completamente conectadas. Específicamente, usaremos CNN, que son particularmente efectivas para procesar datos con una topología similar a una cuadrícula, como imágenes. Se diferencian de las capas completamente conectadas (densas) en un par de maneras. Primero, en las CNN, cada neurona en una capa está conectada solo a una pequeña región de la entrada. Esto se basa en la comprensión de que en los datos de imagen, los grupos locales de píxeles tienen más probabilidades de estar relacionados entre sí. Esta conectividad local reduce el número de parámetros, lo que hace que la red sea más eficiente. Segundo, las CNN usan el concepto de pesos compartidos: se usan los mis-

mos pesos en diferentes regiones de la entrada. Esto es similar a deslizar un filtro por todo el espacio de entrada. Este filtro detecta características específicas (por ejemplo, bordes o texturas) independientemente de su posición en la entrada, lo que lleva a la propiedad de invariancia de la traducción.

Debido a su estructura, las CNN son más eficientes para el procesamiento de imágenes. Requieren menos parámetros que las redes completamente conectadas de tamaño similar, lo que se traduce en tiempos de entrenamiento más rápidos y menores costos computacionales. También suelen ser más eficaces para capturar jerarquías espaciales en datos de imágenes.

Las capas convolucionales y las capas convolucionales transpuestas son dos componentes fundamentales de las CNN, comúnmente utilizadas en el procesamiento de imágenes y la visión artificial. Tienen diferentes propósitos y características: las capas convolucionales se utilizan para la extracción de características. Aplican un conjunto de filtros aprendibles (también conocidos como kernels) a los datos de entrada para detectar patrones y características a diferentes escalas espaciales. Estas capas son esenciales para capturar representaciones jerárquicas de los datos de entrada. Por el contrario, las capas convolucionales transpuestas se utilizan para el sobre-muestreo o la generación de mapas de características de alta resolución.

En esta sección, aprenderá cómo funcionan las operaciones convolucionales y cómo el tamaño del kernel, el paso y el relleno de ceros afectan operaciones convolucionales ct.

4.2.1 ¿Cómo funcionan las operaciones convolucionales?

Las capas convolucionales utilizan filtros para extraer patrones espaciales de los datos de entrada. Una capa convolucional es capaz de detectar automáticamente un gran número de patrones y asociarlos con la etiqueta de destino. Por lo tanto, se utilizan comúnmente en tareas de clasificación de imágenes.

Las operaciones convolucionales implican la aplicación de un filtro a una imagen de entrada para generar un mapa de características. Este proceso implica la multiplicación del filtro por elementos de la imagen de entrada y la suma de los resultados. Los pesos del filtro son los mismos a medida que

este se desplaza sobre la imagen de entrada para escanear diferentes áreas. La Figura 4.5 muestra un ejemplo numérico del funcionamiento de las operaciones convolucionales. La columna izquierda representa la imagen de entrada y la segunda columna es un filtro (una matriz de 2×2). Las operaciones convolucionales (la tercera columna) implican deslizar el filtro sobre la imagen de entrada, multiplicar los elementos correspondientes y sumar los resultados (la última columna).

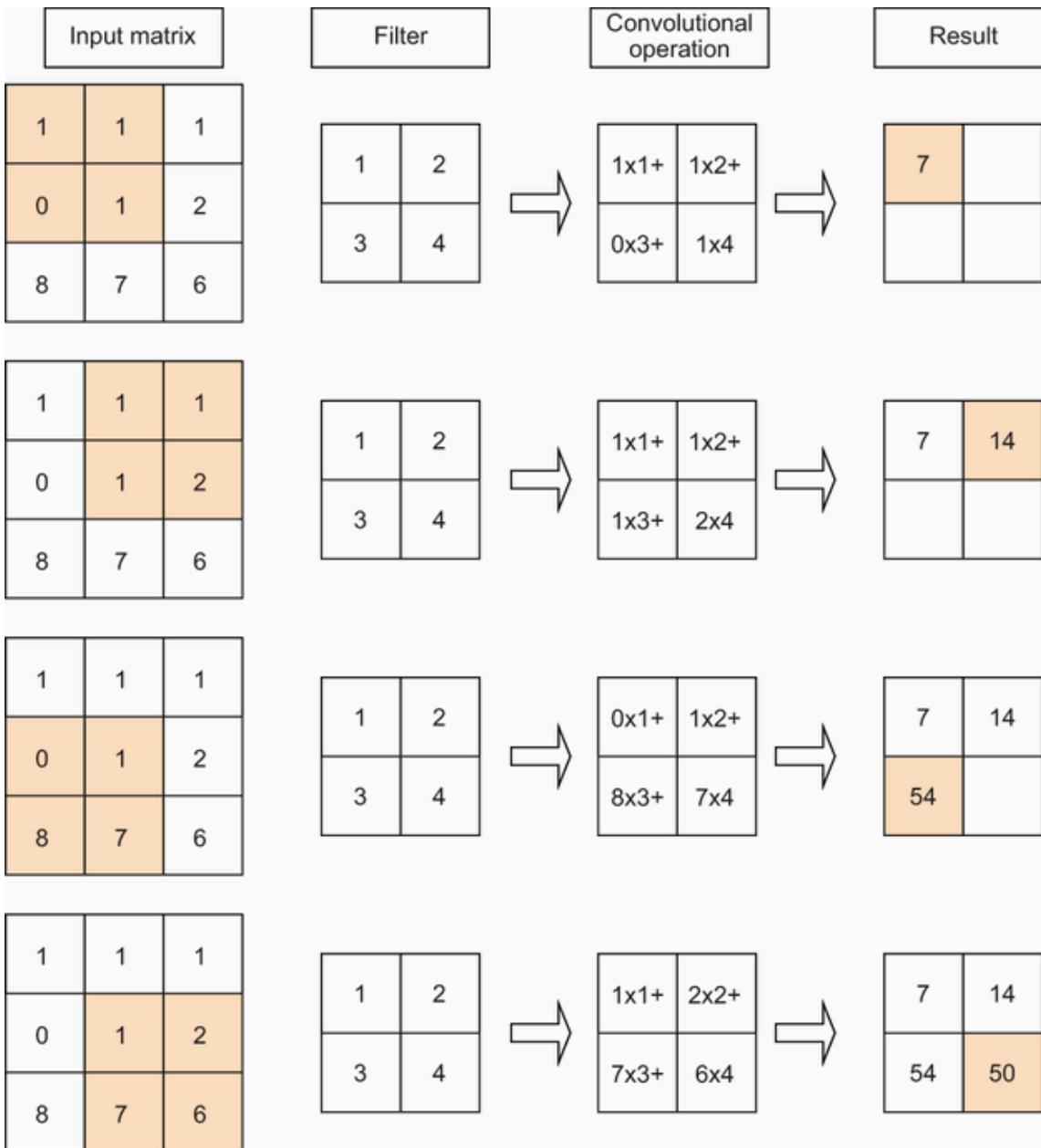


Figura 4.5 Un ejemplo numérico de cómo funcionan las operaciones convolucionales, con paso igual a 1 y sin relleno

Para comprender a fondo cómo funcionan las operaciones convolucionales, implementémoslas en PyTorch en paralelo para que pueda verificar los números, como se muestra en la figura 4.5. Primero, creamos un tensor de PyTorch para representar la imagen de entrada en la figura:

```
img = torch.Tensor([[1,1,1],  
                   [0,1,2],  
                   [8,7,6]]).reshape(1,1,3,3)      ①
```

- ① Los cuatro valores en la forma de la imagen, $(1, 1, 3, 3)$, son la cantidad de imágenes en el lote, la cantidad de canales de color, la altura de la imagen y el ancho de la imagen, respectivamente.

La imagen se redimensiona para que tenga una dimensión de $(1, 1, 3, 3)$, lo que indica que solo hay una observación en el lote y que la imagen tiene un solo canal de color. La altura y el ancho de la imagen son de 3 píxeles.

Representemos el filtro 2×2 , como se muestra en la segunda columna de la figura 4.5, creando una capa convolucional 2D en PyTorch:

```
conv=nn.Conv2d(in_channels=1,  
              out_channels=1,  
              kernel_size=2,  
              stride=1)                      ①  
sd=conv.state_dict()                  ②  
print(sd)
```

- ① Inicia una capa convolucional 2D

- ② Extrae los pesos y el sesgo inicializados aleatoriamente en la capa

Una capa convolucional 2D acepta varios argumentos. El argumento `in_channels` es el número de canales en la imagen de entrada. Este valor es 1 para imágenes en escala de grises y 3 para imágenes en color, ya que las imágenes en color tienen tres canales de color (rojo, verde y azul [RGB]). El argumento `out_channels` es el número de canales después de la capa convolucional, que puede ser cualquier número según la cantidad de características que se deseen extraer de la imagen. El argumento `kernel_size` controla el tamaño del kernel; por ejemplo, `kernel_size=3` significa que el filtro tiene una forma de 3×3 y `kernel_size=4` significa que el filtro tiene una forma de 4×4 . Establecemos el tamaño del kernel en 2 para que el filtro tenga una forma de 2×2 .

Una capa convolucional 2D también tiene varios argumentos opcionales. El argumento `stride` especifica cuántos píxeles se moverán hacia la derecha o hacia abajo cada vez que el filtro recorra la imagen de entrada. El valor `stride` predeterminado es 1. Un valor más alto de `stride` implica un mayor submuestreo de la imagen. El argumento `padding` indica cuántas filas de ceros se añadirán a los cuatro lados de la imagen de entrada, con un valor predeterminado de 0. El argumento `bias` indica si se añadirá un sesgo aprendible como parámetro, con un valor predeterminado de `True`.

La capa convolucional 2D anterior tiene un canal de entrada y un canal de salida, con un tamaño de kernel de 2×2 y un paso de 1. Al crear la capa convolucional, los pesos y el sesgo se inicializan aleatoriamente. Verá la siguiente salida como pesos y sesgo de esta capa convolucional:

```
OrderedDict([('weight', tensor([[[[ 0.3823,  0.4150],  
[-0.1171,  0.4593]]]])), ('bias', tensor([-0.1096]))])
```

Para que nuestro ejemplo sea más fácil de seguir, reemplazaremos los pesos y el sesgo con números enteros:

```
weights={'weight':torch.tensor([[[[1,2],  
[3,4]]]]), 'bias':torch.tensor([0])} ①  
for k in sd:  
    with torch.no_grad():  
        sd[k].copy_(weights[k]) ②  
print(conv.state_dict()) ③
```

① Selecciona manualmente pesos y sesgos

② Reemplaza los pesos y el sesgo en la capa convolucional con nuestros números seleccionados manualmente

③ Imprime los nuevos pesos y sesgos en la capa convolucional

Dado que no estamos aprendiendo los parámetros en la capa convolucional, `torch.no_grad()` se usa para deshabilitar el cálculo de gradiente, lo que reduce el consumo de memoria y acelera los cálculos. Ahora, la capa convolucional tiene los pesos y el sesgo que hemos elegido. Estos también

coinciden con los números de la figura 4.5. El resultado de la celda de código anterior es:

```
OrderedDict([('weight', tensor([[[[1., 2.],  
[3., 4.]]]])), ('bias', tensor([0.]))])
```

Si aplicamos la capa convolucional anterior a la imagen de 3×3 que mencionamos, ¿cuál es el resultado? Averigüémoslo:

```
output = conv(img)  
print(output)
```

La salida es

```
tensor([[[[ 7., 14.],  
[54., 50.]]]], grad_fn=<ConvolutionBackward0>)
```

La salida tiene una forma de $(1, 1, 2, 2)$, con cuatro valores: 7, 14, 54 y 50. Estos números coinciden con los de la figura 4.5.

Pero ¿cómo genera exactamente la capa convolucional esta salida a través del filtro? Lo explicaremos en detalle a continuación.

La imagen de entrada es una matriz de 3×3 y el filtro es una matriz de 2×2 . Cuando el filtro escanea la imagen, primero cubre los cuatro píxeles de la esquina superior izquierda, que tienen valores $[[1, 1], [0, 1]]$, como se muestra en la primera fila de la Figura 4.5. El filtro tiene valores $[[1, 2], [3, 4]]$. La operación de convolución calcula la suma de la multiplicación elemento a elemento de los dos tensores (en este caso, un tensor es el filtro y el otro es el área cubierta). En otras palabras, la operación de convolución realiza la multiplicación elemento a elemento en cada una de las cuatro celdas y luego suma los valores de las cuatro celdas. Por lo tanto, el resultado del escaneo de la esquina superior izquierda es

$$1 \times 1 \times 1 \times 2 + 0 \times 3 + 1 \times 4 = 7.$$

Esto explica por qué la esquina superior izquierda de la salida tiene un valor de 7. De igual manera, al aplicar el filtro a la esquina superior derecha

de la imagen, el área cubierta es `[[1,1],[1,2]]`. Por lo tanto, la salida es:

$$1 \times 1 + 1 \times 2 + 1 \times 3 + 2 \times 4 = 14.$$

Esto explica por qué la esquina superior derecha de la salida tiene un valor de 14.

Ejercicio 4.3

¿Cuáles son los valores del área cubierta cuando se aplica el filtro a la esquina inferior derecha de la imagen? Explique por qué la plataforma inferior...La esquina superior derecha de la salida tiene un valor de 50.

4.2.2 ¿Cómo afectan el paso y el relleno a las operaciones convolucionales?

El paso y el relleno de ceros son dos conceptos importantes en el contexto de las operaciones convolucionales. Desempeñan un papel crucial en la determinación de las dimensiones del mapa de características de salida y la forma en que el filtro interactúa con los datos de entrada.

El paso se refiere al número de píxeles que el filtro recorre la imagen de entrada. Cuando el paso es 1, el filtro recorre un píxel a la vez. Un paso mayor significa que el filtro salta sobre más píxeles al deslizarse por la imagen. Aumentar el paso reduce las dimensiones espaciales del mapa de características de salida.

El relleno de ceros implica añadir capas de ceros alrededor del borde de la imagen de entrada antes de aplicar la operación convolucional. Este relleno permite controlar las dimensiones espaciales del mapa de características de salida. Sin relleno, las dimensiones de la salida serán menores que las de la entrada. Al añadir relleno, se pueden conservar las dimensiones de la entrada.

Usemos un ejemplo para mostrar cómo funcionan la zancada y el relleno. La siguiente celda de código redefine la capa convolucional 2D:

```
conv=nn.Conv2d(in_channels=1,  
              out_channels=1,  
              kernel_size=2,
```

```

        stride=2,                                ①
        padding=1)
sd=conv.state_dict()
for k in sd:
    with torch.no_grad():
        sd[k].copy_(weights[k])
output = conv(img)
print(output)

```

① Cambia la zancada de 1 a 2

② Cambia el relleno de 0 a 1

La salida es

```

tensor([[[[ 4.,  7.],
          [32., 50.]]]], grad_fn=<ConvolutionBackward0>)

```

El argumento `padding=1` agrega una fila de 0 alrededor de la imagen de entrada, por lo que la imagen rellena ahora tiene un tamaño de 5×5 en lugar de 3×3 .

Cuando el filtro escanea la imagen rellena, primero cubre la esquina superior izquierda, que tiene valores $[[0, 0], [0, 1]]$. El filtro tiene valores $[[1, 2], [3, 4]]$. Por lo tanto, el resultado del escaneo de la esquina superior izquierda es:

$$0 \times 1 + 0 \times 2 + 0 \times 3 + 1 \times 4 = 4$$

Esto explica por qué la esquina superior izquierda de la salida tiene un valor de 4. De igual forma, cuando el filtro se desliza dos píxeles hacia abajo, hasta la esquina inferior izquierda de la imagen, el área cubierta es $[[0, 0], [0, 8]]$. Por lo tanto, la salida es:

$$0 \times 1 + 0 \times 2 + 0 \times 3 + 8 \times 4 = 32$$

Esto explica por qué la esquina inferior izquierda de la salida tiene un valor de 32.

4.3 Convolución transpuesta y normalización por lotes

Las capas convolucionales transpuestas también se conocen como capas de deconvolución o de sobremuestreo. Se utilizan para sobremuestrear o generar mapas de características de alta resolución. Suelen emplearse en modelos generativos como las GAN y las VAE.

Las capas convolucionales transpuestas aplican un filtro a los datos de entrada, pero a diferencia de la convolución estándar, aumentan las dimensiones espaciales insertando espacios entre los valores de salida, lo que aumenta la escala de los mapas de características. Este proceso genera mapas de características de mayor resolución. Las capas convolucionales transpuestas ayudan a aumentar la resolución espacial, lo cual resulta útil en la generación de imágenes.

Las zancadas se pueden usar en capas de convolución transpuestas para controlar el sobremuestreo. Cuanto mayor sea el valor de la zancada, mayor será el sobremuestreo que la capa de convolución transpuesta aplicará a los datos de entrada.

La normalización por lotes bidimensional es una técnica utilizada en redes neuronales, en particular en las CNN, para estabilizar y acelerar el proceso de entrenamiento. Aborda diversos problemas, como la saturación, los gradientes evanescentes y los gradientes explosivos, que son desafíos comunes en el aprendizaje profundo. En esta sección, analizará algunos ejemplos para comprender mejor su funcionamiento. En la siguiente sección, la utilizará al crear GAN para generar imágenes en color de alta resolución.

Gradientes evanescentes y explosivos en el aprendizaje profundo

El problema del gradiente evanesciente ocurre en redes neuronales profundas cuando los gradientes de la función de pérdida con respecto a los parámetros de la red se vuelven excesivamente pequeños durante la retropropagación. Esto resulta en actualizaciones muy lentas de los parámetros, lo que dificulta el proceso de aprendizaje, especialmente en las primeras capas de la red. Por el contrario, el problema del gradiente explosivo ocurre cuando estos gradientes se vuelven excesivamente grandes, lo que genera actualizaciones inestables y provoca que los parámetros del

modelo oscilen o diverjan a valores muy altos. Ambos problemas dificultan el entrenamiento eficaz de redes neuronales profundas.

4.3.1 ¿Cómo funcionan las capas convolucionales transpuestas?

A diferencia de las capas convolucionales, las capas convolucionales transpuestas sobremuestrean y llenan los huecos de una imagen para generar características y aumentar la resolución mediante el uso de kernels (es decir, filtros). La salida suele ser mayor que la entrada en una capa convolucional transpuesta. Por lo tanto, las capas convolucionales transpuestas son herramientas esenciales para generar imágenes de alta resolución. Para mostrar exactamente cómo funcionan las operaciones convolucionales transpuestas 2D, utilicemos un ejemplo sencillo y una figura. Supongamos que tiene una imagen de entrada muy pequeña de 2×2 , como se muestra en la columna izquierda de la figura 4.6.

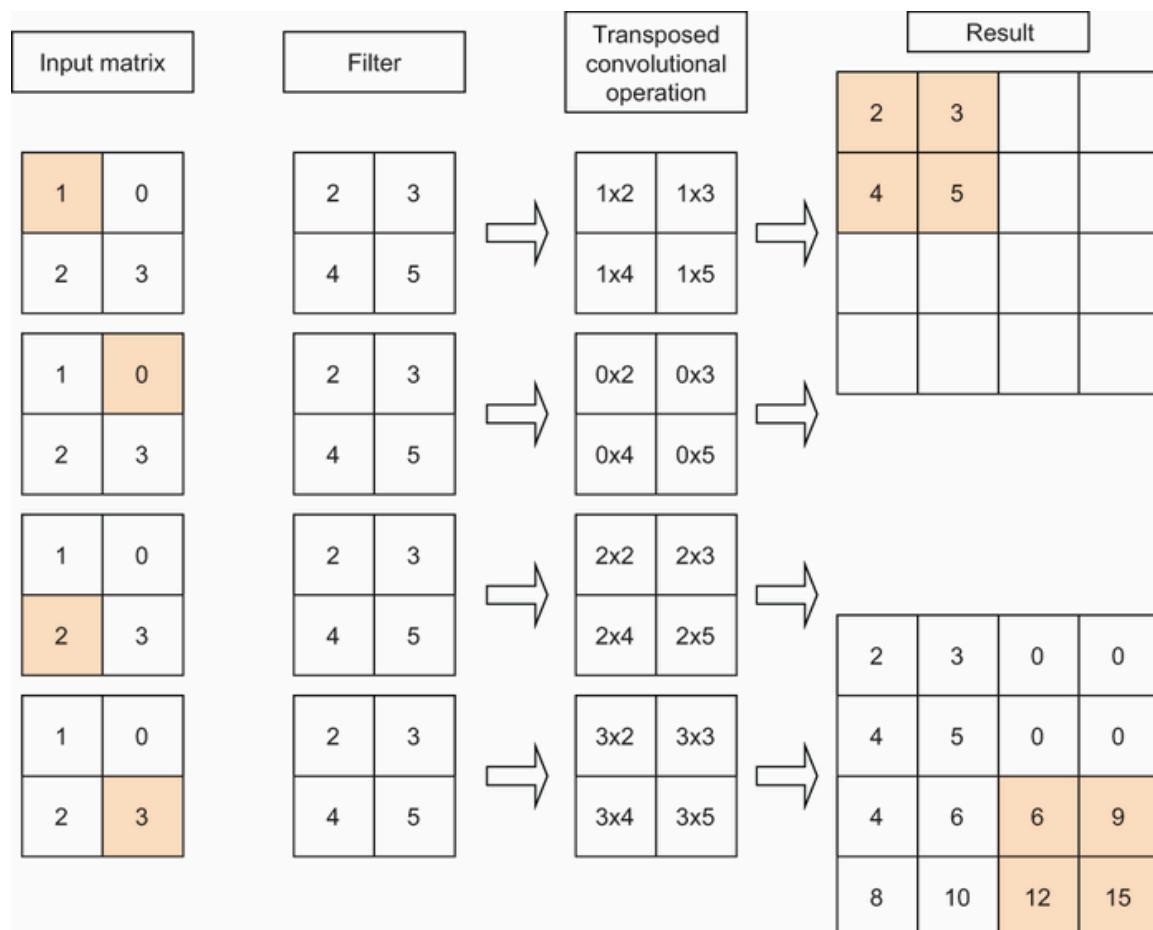


Figura 4.6 Un ejemplo numérico de cómo funcionan las operaciones convolucionales transpuestas

La imagen de entrada tiene los siguientes valores:

```
img = torch.Tensor([[1,0],  
[2,3]]).reshape(1,1,2,2)
```

Quiere sobremuestrear la imagen para que tenga mayor resolución. Puede crear una capa convolucional transpuesta 2D en PyTorch:

```
transconv=nn.ConvTranspose2d(in_channels=1,  
                           out_channels=1,  
                           kernel_size=2,  
                           stride=2) ①  
sd=transconv.state_dict()  
weights={'weight':torch.tensor([[[[2,3],  
[4,5]]]]), 'bias':torch.tensor([0])}  
for k in sd:  
    with torch.no_grad():  
        sd[k].copy_(weights[k]) ②
```

- ① Una capa convolucional transpuesta con un canal de entrada, un canal de salida, un tamaño de kernel de 2 y un paso de 2
- ② Reemplaza los pesos y el sesgo en la capa convolucional transpuesta con valores seleccionados manualmente

Esta capa convolucional transpuesta 2D tiene un canal de entrada y un canal de salida, con un tamaño de kernel de 2×2 y un paso de 2. El filtro 2×2 se muestra en la segunda columna de la figura 4.6. Reemplazamos los pesos inicializados aleatoriamente y el sesgo en la capa con números enteros seleccionados manualmente para facilitar el seguimiento de los cálculos. El método `state_dict()` del código anterior devuelve los parámetros en una red neuronal profunda.

Al aplicar la capa convolucional transpuesta a la imagen 2×2 que mencionamos antes, ¿cuál es el resultado? Averigüémoslo:

```
transoutput = transconv(img)  
print(transoutput)
```

La salida es

```
tensor([[[[ 2.,  3.,  0.,  0.],
          [ 4.,  5.,  0.,  0.],
          [ 4.,  6.,  6.,  9.],
          [ 8., 10., 12., 15.]]]], grad_fn=<ConvolutionBackward0>)
```

La salida tiene la forma $(1, 1, 4, 4)$, lo que significa que hemos sobremuestreado una imagen de 2×2 a una de 4×4 . ¿Cómo genera la capa convolucional transpuesta la salida anterior a través del filtro? Lo explicaremos en detalle a continuación.

La imagen es una matriz de 2×2 y el filtro también lo es . Al aplicar el filtro a la imagen, cada elemento se multiplica por él y se muestra en la salida. El valor superior izquierdo de la imagen es 1, y lo multiplicamos por los valores del filtro, , lo que da como resultado los cuatro valores en el bloque superior izquierdo de la matriz de salida, con valores, como se muestra en la esquina superior derecha de la figura 4.6. De forma similar, el valor inferior izquierdo de la imagen es 2, y lo multiplicamos por los valores del filtro, , lo que da como resultado los cuatro valores en el bloque inferior izquierdo de la matriz de salida, .
[[2, 3], [4, 5]] transoutput [[2, 3], [4, 5]]
[[2, 3], [4, 5]] transoutput [[4, 6], [8, 10]]

Ejercicio 4.4

Si una imagen contiene valores [[10, 10], [15, 20]], ¿cuál es el resultado tras aplicarle la capa convolucional transpuesta 2D transconv ? Supongamos transconv que contiene valores [[2, 3], [4, 5]]. Supongamos un tamaño de kernel de 2 y un tamaño de paso de 2.

4.3.2 Normalización de lotes

La normalización bidimensional por lotes es una técnica estándar en los marcos modernos de aprendizaje profundo y se ha convertido en un componente crucial para el entrenamiento eficaz de redes neuronales profundas. La verás con frecuencia más adelante en este libro.

En la normalización por lotes 2D, la normalización se realiza de forma independiente para cada canal de características ajustando y escalando valores en el canal, por lo que tienen una media de 0 y una varianza de 1. Un canal de características se refiere a una de las dimensiones de un ten-

sor multidimensional en las CNN que se utiliza para representar diferentes aspectos o características de los datos de entrada. Por ejemplo, pueden representar canales de color como rojo, verde o azul. La normalización garantiza que la distribución de las entradas a las capas profundas de la red se mantenga más estable durante el entrenamiento. Esta estabilidad surge porque el proceso de normalización reduce el desplazamiento interno de las covariables, que es el cambio en la distribución de las activaciones de la red debido a la actualización de los pesos en las capas inferiores. También ayuda a abordar los problemas de gradientes de desaparición o explosión al mantener las entradas en un rango adecuado para evitar que los gradientes se vuelvan demasiado pequeños (desaparición) o demasiado grandes (explosión).¹

Así funciona la normalización por lotes 2D: para cada canal de características, primero calculamos la media y la varianza de todas las observaciones dentro del canal. A continuación, normalizamos los valores de cada canal utilizando la media y la varianza obtenidas previamente (restando la media de cada observación y dividiendo la diferencia entre la desviación estándar). Esto garantiza que los valores de cada canal tengan una media de 0 y una desviación estándar de 1 después de la normalización, lo que ayuda a estabilizar y acelerar el entrenamiento. También ayuda a mantener gradientes estables durante la retropropagación, lo que facilita aún más el entrenamiento de redes neuronales profundas.

Utilicemos un ejemplo concreto para mostrar cómo funciona la normalización de lotes 2D.

Supongamos que tiene una entrada de tres canales con un tamaño de 64×64 . Pasa la entrada a través de una capa convolucional 2D con tres canales de salida de la siguiente manera:

```
torch.manual_seed(42)                                ①  
img = torch.rand(1,3,64,64)                          ②  
conv = nn.Conv2d(in_channels=3,  
                out_channels=3,  
                kernel_size=3,  
                stride=1,  
                padding=1)                                ③
```

```
out=conv(img)  
print(out.shape)
```

(4)

- ① Corrige el estado aleatorio para que los resultados sean reproducibles
- ② Crea una entrada de 3 canales
- ③ Crea una capa convolucional 2D
- ④ Pasa la entrada a través de la capa convolucional

La salida de la celda de código anterior es

```
torch.Size([1, 3, 64, 64])
```

Creamos una entrada de tres canales y la pasamos a través de una capa convolucional 2D con tres canales de salida. La entrada procesada tiene tres canales con un tamaño de 64×64 píxeles.

Veamos la media y la desviación estándar de los píxeles en cada uno de los tres canales de salida:

```
for i in range(3):  
    print(f"mean in channel {i} is", out[:,i,:,:].mean().item())  
    print(f"std in channel {i} is", out[:,i,:,:].std().item())
```

La salida es

```
mean in channel 0 is -0.3766776919364929  
std in channel 0 is 0.17841289937496185  
mean in channel 1 is -0.3910464942455292  
std in channel 1 is 0.16061744093894958  
mean in channel 2 is 0.39275866746902466  
std in channel 2 is 0.18207983672618866
```

Los valores promedio de los píxeles en cada canal de salida no son 0; las desviaciones estándar de los píxeles en cada canal de salida no son 1.

Ahora, realizamos una normalización por lotes 2D:

```
norm=nn.BatchNorm2d(3)
out2=norm(out)
print(out2.shape)
for i in range(3):
    print(f"mean in channel {i} is", out2[:,i,:,:].mean().item())
    print(f"std in channel {i} is", out2[:,i,:,:].std().item())
```

Entonces tenemos el siguiente resultado:

```
torch.Size([1, 3, 64, 64])
mean in channel 0 is 6.984919309616089e-09
std in channel 0 is 0.9999650120735168
mean in channel 1 is -5.3085386753082275e-08
std in channel 1 is 0.9999282956123352
mean in channel 2 is 9.872019290924072e-08
std in channel 2 is 0.9999712705612183
```

Los valores promedio de los píxeles en cada canal de salida ahora son prácticamente 0 (o un número muy pequeño que está cerca de 0); las desviaciones estándar de los píxeles en cada canal de salida ahora son un número cercano a 1. Eso es lo que hace la normalización por lotes: normaliza las observaciones en cada canal de características para que los valores en cada canal de características tengan una media de 0 y una desviación estándar unitaria.

4.4 Imágenes en color de caras de anime

En este segundo proyecto, aprenderás a crear imágenes a color de alta resolución. Los pasos de entrenamiento son similares a los del primero, con la excepción de que los datos de entrenamiento son imágenes a color de caras de anime. Además, las redes neuronales discriminadoras y generadoras son más sofisticadas. Utilizaremos capas convolucionales 2D y convolucionales transpuestas 2D en ambas redes.

4.4.1 Descargar caras de anime

Puedes descargar los datos de entrenamiento de Kaggle (<https://mng.bz/1a9R>), que contiene 63,632 imágenes a color de caras de anime. Primero, debes crear una cuenta gratuita de Kaggle para iniciar se-

sión. Extrae los datos del archivo zip y guárdalos en una carpeta de tu computadora. Por ejemplo, yo guardé todo el contenido del archivo zip en /files/anime/ en mi computadora. Como resultado, todas las imágenes de caras de anime están en /files/anime/images/.

Define el nombre de la ruta para que puedas usarlo más tarde para cargar las imágenes en Pytorch:

```
anime_path = r"files/anime"
```

Cambie el nombre de la ruta según la ubicación de las imágenes en su ordenador. Tenga en cuenta que la clase `ImageFolder()` utiliza el nombre del directorio de las imágenes para identificar la clase a la que pertenecen. Por lo tanto, el directorio final `/images/` no se incluye en la definición anterior `anime_path`.

A continuación, usamos la clase `ImageFolder()` del paquete `datasets` Torchvision para cargar el conjunto de datos:

```
from torchvision import transforms as T
from torchvision.datasets import ImageFolder

transform = T.Compose([T.Resize((64, 64)),                      ①
                      T.ToTensor(),                         ②
                      T.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])])    ③
train_data = ImageFolder(root=anime_path,
                        transform=transform)                           ④
```

- ① Cambia el tamaño de la imagen a 64×64
- ② Convierte imágenes en tensores de PyTorch
- ③ Normaliza los valores de la imagen a $[-1, 1]$ en los tres canales de color
- ④ Carga los datos y transforma las imágenes

Realizamos tres transformaciones diferentes al cargar las imágenes desde la carpeta local. Primero, redimensionamos todas las imágenes a 64 píxeles de alto y 64 píxeles de ancho. Segundo, convertimos las imágenes a tenso-

res de PyTorch con valores en el rango [0, 1] mediante la clase `ToTensor()`. Finalmente, usamos la clase `Normalize()` para restar 0,5 del valor y dividir la diferencia entre 0,5. Como resultado, los datos de la imagen ahora están entre -1 y 1.

Ahora podemos poner los datos de entrenamiento en lotes:

```
from torch.utils.data import DataLoader  
  
batch_size = 128  
train_loader = DataLoader(dataset=train_data,  
                          batch_size=batch_size, shuffle=True)
```

El conjunto de datos de entrenamiento ahora está en lotes, con un tamaño de lote de 128.

4.4.2 Imágenes en color de canales primero en PyTorch

PyTorch utiliza un enfoque de "canales primero" al procesar imágenes a color. Esto significa que la forma de las imágenes en PyTorch es (número_canales, altura, anchura). En cambio, en otras bibliotecas de Python, como TensorFlow o Matplotlib, se utiliza un enfoque de "canales al final": una imagen a color tiene una forma de (altura, anchura, número_canales).

Veamos una imagen de ejemplo en nuestro conjunto de datos e imprimamos la forma de la imagen:

```
image0, _ = train_data[0]  
print(image0.shape)
```

La salida es

```
torch.Size([3, 64, 64])
```

La primera imagen tiene unas dimensiones de $3 \times 64 \times 64$. Esto significa que tiene tres canales de color (RGB). Tanto la altura como el ancho de la imagen son de 64 píxeles.

Cuando graficamos las imágenes en Matplotlib, necesitamos convertirlas a canales-last usando el método `permute()` en PyTorch:

```
import matplotlib.pyplot as plt

plt.imshow(image0.permute(1,2,0)*0.5+0.5)
plt.show()
```

Ten en cuenta que debemos multiplicar el tensor de PyTorch que representa la imagen por 0,5 y luego sumarle 0,5 para convertir los valores del rango [-1, 1] al rango [0, 1]. Verás un gráfico de una cara de anime después de ejecutar la celda de código anterior.

A continuación, definimos una función `plot_images()` para visualizar 32 imágenes en cuatro filas y ocho columnas:

```
def plot_images(imgs):                                ①
    for i in range(32):
        ax = plt.subplot(4, 8, i + 1)                  ②
        plt.imshow(imgs[i].permute(1,2,0)/2+0.5)
        plt.xticks([])
        plt.yticks([])
    plt.subplots_adjust(hspace=-0.6)
    plt.show()

imgs, _ = next(iter(train_loader))                  ③
plot_images(imgs)                                    ④
```

- ① Define una función para visualizar 32 imágenes
- ② Los coloca en una cuadrícula de 4×8
- ③ Obtiene un lote de imágenes
- ④ Llama a la función para visualizar las imágenes

Verás una trama de 32 caras de anime en una cuadrícula de 4×8 después de ejecutar la celda de código anterior, como se muestra en la figura 4.7.



Figura 4.7 Ejemplos del conjunto de datos de entrenamiento de caras de anime

4.5 GAN convolucional profunda

En esta sección, crearás un modelo DCGAN para que podamos entrenarlo y generar imágenes de caras de anime. Como es habitual, el modelo GAN consta de una red discriminadora y una red generadora. Sin embargo, estas redes son más sofisticadas que las que hemos visto anteriormente: utilizaremos capas convolucionales, capas convolucionales transpuestas y capas de normalización por lotes.

Comenzaremos con la red discriminadora. Después, explicaré cómo la red generadora refleja las capas de la red discriminadora para crear imágenes en color realistas. Después, entrenarás el modelo con los datos que preparamos anteriormente en este capítulo y lo usarás para generar nuevas imágenes de rostros de anime.

4.5.1 Construyendo un DCGAN

Al igual que en modelos GAN anteriores, el discriminador es un clasificador binario que clasifica las muestras como reales o falsas. Sin embargo, a diferencia de las redes que hemos utilizado hasta ahora, utilizaremos capas convolucionales y normalizaciones por lotes. Las imágenes en color de alta resolución de este proyecto tienen demasiados parámetros, y si solo usamos capas densas, es difícil entrenar el modelo eficazmente. La estructura de la red neuronal discriminadora se muestra en la siguiente lista.

Listado 4.4 Un discriminador en DCGAN

```

import torch.nn as nn
import torch

device = "cuda" if torch.cuda.is_available() else "cpu"

D = nn.Sequential(
    nn.Conv2d(3, 64, 4, 2, 1, bias=False), ①
    nn.LeakyReLU(0.2, inplace=True), ②
    nn.Conv2d(64, 128, 4, 2, 1, bias=False),
    nn.BatchNorm2d(128), ③
    nn.LeakyReLU(0.2, inplace=True),
    nn.Conv2d(128, 256, 4, 2, 1, bias=False),
    nn.BatchNorm2d(256),
    nn.LeakyReLU(0.2, inplace=True),
    nn.Conv2d(256, 512, 4, 2, 1, bias=False),
    nn.BatchNorm2d(512),
    nn.LeakyReLU(0.2, inplace=True),
    nn.Conv2d(512, 1, 4, 1, 0, bias=False),
    nn.Sigmoid(),
    nn.Flatten()).to(device) ④

```

- ① Pasa la imagen a través de una capa convolucional 2D
- ② Aplica la activación de LeakyReLU en las salidas de la primera capa convolucional
- ③ Realiza la normalización por lotes 2D en las salidas de la segunda capa convolucional
- ④ La salida es un valor único entre 0 y 1, que puede interpretarse como la probabilidad de que una imagen sea real.

La entrada a la red discriminadora es una imagen en color con tres canales. La primera capa convolucional 2D es `Conv2d(3, 64, 4, 2, 1, bias=False)`: esto significa que la entrada tiene tres canales y la salida 64; el tamaño del kernel es 4; el paso es 2; y el relleno es 1. Cada una de las capas convolucionales 2D de la red toma una imagen y aplica filtros para extraer características espaciales.

A partir de la segunda capa convolucional 2D, aplicamos la normalización por lotes 2D (que expliqué en la sección anterior) y la activación Leaky-

ReLU (que explicaré más adelante) a la salida. La función de activación LeakyReLU es una versión modificada de ReLU. Permite que la salida tenga una pendiente para valores inferiores a cero. En concreto, la función LeakyReLU se define de la siguiente manera:

$$\text{LeakyReLU}(x) = \begin{cases} x, & \text{for } x > 0 \\ -\beta x, & \text{for } x \leq 0 \end{cases}$$

Donde β es una constante entre 0 y 1. La función de activación LeakyReLU se utiliza comúnmente para abordar el problema de gradientes dispersos (cuando la mayoría de los gradientes se vuelven cero o casi cero). El entrenamiento de DCGAN es un ejemplo de ello. Cuando la entrada a una neurona es negativa, la salida de ReLU es cero y la neurona se vuelve inactiva. LeakyReLU devuelve un valor negativo pequeño, no cero, para entradas negativas. Esto ayuda a mantener las neuronas activas y en proceso de aprendizaje, manteniendo un mejor flujo de gradiente y acelerando la convergencia de los parámetros del modelo.

Utilizaremos el mismo enfoque al crear el generador para la generación de prendas de vestir. Duplicaremos las capas utilizadas en el discriminador en DCGAN para crear un generador, como se muestra en la siguiente lista.

Listado 4.5 Diseño de un generador en DCGAN

```
G=nn.Sequential(  
    nn.ConvTranspose2d(100, 512, 4, 1, 0, bias=False),      (1)  
    nn.BatchNorm2d(512),  
    nn.ReLU(inplace=True),  
    nn.ConvTranspose2d(512, 256, 4, 2, 1, bias=False),     (2)  
    nn.BatchNorm2d(256),  
    nn.ReLU(inplace=True),  
    nn.ConvTranspose2d(256, 128, 4, 2, 1, bias=False),  
    nn.BatchNorm2d(128),  
    nn.ReLU(inplace=True),  
    nn.ConvTranspose2d(128, 64, 4, 2, 1, bias=False),  
    nn.BatchNorm2d(64),  
    nn.ReLU(inplace=True),  
    nn.ConvTranspose2d(64, 3, 4, 2, 1, bias=False),        (3)  
    nn.Tanh()).to(device)                                (4)
```

- ① La primera capa del generador se modela a partir de la última capa del discriminador.
- ② La segunda capa del generador es simétrica a la segunda última capa del discriminador (los números de entradas y salidas tienen posiciones cambiadas).
- ③ La última capa del generador es simétrica a la primera capa del discriminador.
- ④ Utiliza la activación Tanh() para comprimir los valores en la capa de salida al rango $[-1, 1]$ porque las imágenes en el conjunto de entrenamiento tienen valores entre -1 y 1

Como se muestra en la figura 4.8, para crear una imagen, el generador utiliza cinco capas convolucionales transpuestas 2D: son simétricas a las cinco capas convolucionales 2D del discriminador. Por ejemplo, la última capa, `ConvTranspose2d(64, 3, 4, 2, 1, bias=False)`, se modela a partir de la primera capa del discriminador, `Conv2d(3, 64, 4, 2, 1, bias=False)`. Los números de canales *de entrada y salida* `Conv2d` en se invierten y se utilizan como los números de canales *de entrada y salida* en `ConvTranspose2d`.

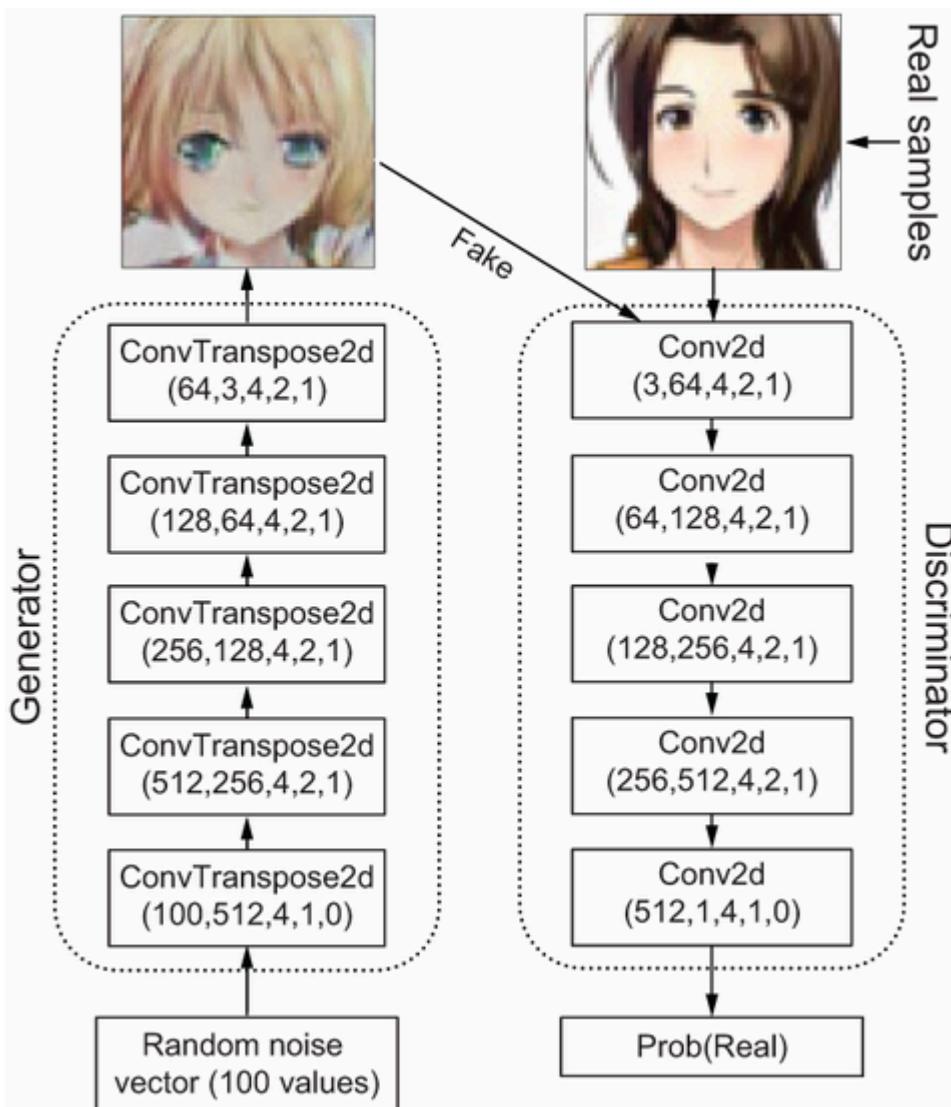


Figura 4.8 Diseño de una red generadora en DCGAN para crear caras de anime mediante la duplicación de las capas de la red discriminadora. El lado derecho del diagrama muestra la red discriminadora, que contiene cinco capas convolucionales 2D. Para diseñar un generador capaz de crear caras de anime de la nada, duplicamos las capas de la red discriminadora. Específicamente, como se muestra en la mitad izquierda de la figura, el generador tiene cinco capas convolucionales 2D transpuestas, simétricas a las capas convolucionales 2D del discriminador. Además, en cada una de las cuatro capas superiores, los números de `input` y `output` de los canales del discriminador se invierten y se utilizan como los números de `output` y de `input` los canales del generador.

El número de canales de entrada en la primera capa convolucional transpuesta 2D es 100. Esto se debe a que el generador obtiene un vector de ruido aleatorio de 100 valores del espacio latente (parte inferior izquierda de la figura 4.8) y lo alimenta. El número de canales de salida en la última capa convolucional transpuesta 2D del generador es 3, ya que la salida es una imagen con tres canales de color (RGB). Aplicamos la función de activación Tanh a la salida del generador para comprimir todos los valores al rango $[-1, 1]$, ya que todas las imágenes de entrenamiento tienen valores entre -1 y 1 .

Como es habitual, la función de pérdida es la pérdida de entropía cruzada binaria. El discriminador intenta maximizar la precisión de la clasificación

binaria: identificar una muestra real como real y una muestra falsa como falsa. El generador, por otro lado, intenta minimizar la probabilidad de que la muestra falsa se identifique como falsa.

Usaremos el optimizador Adam tanto para el discriminador como para el generador y estableceremos la tasa de aprendizaje en 0,0002:

```
loss_fn=nn.BCELoss()
lr = 0.0002
optimG = torch.optim.Adam(G.parameters(),
                         lr = lr, betas=(0.5, 0.999))
optimD = torch.optim.Adam(D.parameters(),
                         lr = lr, betas=(0.5, 0.999))
```

Ya viste el optimizador Adam en el capítulo 2, pero con valores beta predeterminados. Aquí, seleccionamos betas diferentes a los predeterminados. Las betas del optimizador Adam desempeñan un papel crucial en la estabilización y aceleración de la convergencia del proceso de entrenamiento. Esto se logra controlando el énfasis en la información de gradiente reciente frente a la pasada (beta1) y adaptando la tasa de aprendizaje en función de la certeza de la información de gradiente (beta2). Estos parámetros suelen ajustarse con precisión según las características específicas del problema que se está resolviendo.

4.5.2 Entrenamiento y uso de DCGAN

El proceso de entrenamiento para DCGAN es similar al que hemos realizado para otros modelos GAN, como los utilizados en el capítulo 3 y en secciones anteriores de este capítulo. Dado que desconocemos la distribución real de las imágenes de rostros de anime, utilizaremos técnicas de visualización para determinar cuándo finaliza el entrenamiento. En concreto, definimos una función `test_epoch()` para visualizar los rostros de anime creados por el generador después de cada época de entrenamiento:

```
def test_epoch():
    noise=torch.randn(32,100,1,1).\
        to(device=device)                                ①
    fake_samples=G(noise).cpu().detach()               ②
    for i in range(32):
        ax = plt.subplot(4, 8, i + 1)
```

```

    img=(fake_samples.cpu().detach()[i]/2+0.5).\
        permute(1,2,0)
    plt.imshow(img)
    plt.xticks([])
    plt.yticks([])
    plt.subplots_adjust(hspace=-0.6)
    plt.show()
test_epoch()
```

④

- ① Obtiene 32 vectores de ruido aleatorio del espacio latente
- ② Genera 32 imágenes de caras de anime
- ③ Traza las imágenes generadas en una cuadrícula de 4×8
- ④ Llama a la función para generar imágenes antes de entrenar el modelo

Si ejecutas la celda de código anterior, verás 32 imágenes que parecen copos de nieve en la pantalla de un televisor. No parecen caras de anime en absoluto, ya que aún no hemos entrenado el generador.

Definimos tres funciones, `train_D_on_real()`, `train_D_on_fake()`, y `train_G()`, similares a las que utilizamos para entrenar las GAN y generar imágenes en escala de grises de prendas de vestir anteriormente en este capítulo. Visite el Jupyter Notebook de este capítulo en el repositorio de GitHub del libro y familiarícese con las funciones. Estas funciones entranan al discriminador con imágenes reales. Luego, lo entrenaan con imágenes falsas y, finalmente, entrenaan al generador.

A continuación, entrenaam os el modelo durante 20 épocas:

```

for i in range(20):
    gloss=0
    dloss=0
    for n, (real_samples,_) in enumerate(train_loader):
        loss_D=train_D_on_real(real_samples)
        dloss+=loss_D
        loss_D=train_D_on_fake()
        dloss+=loss_D
        loss_G=train_G()
        gloss+=loss_G
```

```
gloss=gloss/n  
dloss=dloss/n  
print(f"epoch {i+1}, dloss: {dloss}, gloss {gloss}")  
test_epoch()
```

El entrenamiento dura unos 20 minutos si se utiliza entrenamiento con GPU. De lo contrario, puede tardar de 2 a 3 horas, dependiendo del hardware. Configuración en tu computadora. También puedes descargar el modelo entrenado desde mi sitio web:

https://gattonweb.uky.edu/faculty/lium/gai/anime_gen.zip.

Tras cada época de entrenamiento, se pueden visualizar las caras de anime generadas. Tras una sola época de entrenamiento, el modelo ya puede generar imágenes en color que se asemejan a caras de anime, como se muestra en la figura 4.9. A medida que avanza el entrenamiento, la calidad de las imágenes generadas mejora cada vez más.



Figura 4.9 Imágenes generadas en DCGAN después de una época de entrenamiento

Descartaremos el discriminador y guardaremos el generador entrenado en la carpeta local:

```
scripted = torch.jit.script(G)  
scripted.save('files/anime_gen.pt')
```

Para utilizar el generador entrenado, cargamos el modelo y lo usamos para generar 32 imágenes:

```
new_G=torch.jit.load('files/anime_gen.pt',
                      map_location=device)
new_G.eval()
noise=torch.randn(32,100,1,1).to(device)
fake_samples=new_G(noise).cpu().detach()
for i in range(32):
    ax = plt.subplot(4, 8, i + 1)
    img=(fake_samples.cpu().detach()[i]/2+0.5).permute(1,2,0)
    plt.imshow(img)
    plt.xticks([])
    plt.yticks([])
plt.subplots_adjust(hspace=-0.6)
plt.show()
```

Las caras de anime generadas se muestran en la figura 4.10. Las imágenes generadas se parecen mucho a las del conjunto de entrenamiento que se muestra en la figura 4.7.



Figura 4.10 Imágenes de rostros de anime generadas por el generador entrenado en DCGAN

Quizás hayas notado que los colores de pelo de las imágenes generadas son diferentes: algunos son negros, otros pelirrojos y otros rubios. Quizás te preguntes: ¿Podemos indicarle al generador que cree imágenes con una característica específica, como pelo negro o pelirrojo? La respuesta es sí. En el capítulo 5, aprenderás un par de métodos diferentes para seleccionar características en imágenes generadas en GAN.

Resumen

- Para crear imágenes de apariencia realista de la nada, el generador refleja capas utilizadas en la red discriminadora.
- Si bien es posible generar imágenes en escala de grises utilizando solo capas completamente conectadas, para generar imágenes en color de alta resolución, necesitamos utilizar CNN.
- Las capas convolucionales bidimensionales se utilizan para la extracción de características. Aplican un conjunto de filtros aprendibles (también conocidos como kernels) a los datos de entrada para detectar patrones y características a diferentes escalas espaciales. Estas capas son esenciales para capturar representaciones jerárquicas de los datos de entrada.
- Las capas convolucionales transpuestas bidimensionales (también conocidas como capas de deconvolución o de sobremuestreo) se utilizan para sobremuestrear o generar mapas de características de alta resolución. Aplican un filtro a los datos de entrada. Sin embargo, a diferencia de la convolución estándar, aumentan las dimensiones espaciales insertando espacios entre los valores de salida, lo que aumenta la escala de los mapas de características. Este proceso genera mapas de características de mayor resolución.
- La normalización por lotes bidimensional es una técnica comúnmente utilizada en aprendizaje profundo y redes neuronales para mejorar el entrenamiento y el rendimiento de las CNN y otros modelos que trabajan con datos 2D, como imágenes. Normaliza los valores de cada canal de características, de modo que tengan una media de 0 y una desviación estándar de 1, lo que ayuda a estabilizar y acelerar el entrenamiento.

¹ Sergey Ioffe, Christian Szegedy, 2015, “Normalización por lotes: aceleración del entrenamiento de redes profundas mediante la reducción del desplazamiento interno de covariables”. <https://arxiv.org/abs/1502.03167>.

5. Selección de características en imágenes generadas

Este capítulo cubre

- Construcción de una red generativa adversarial condicional para generar imágenes con ciertos atributos (rostros humanos con o sin gafas, por ejemplo)
- Implementación de la distancia de Wasserstein y la penalización de gradiente para mejorar la calidad de la imagen
- Seleccionar vectores asociados a diferentes características para que el modelo GAN entrenado genere imágenes con determinadas características (rostros masculinos o femeninos, por ejemplo)
- Combinando GAN condicional con selección vectorial para especificar dos atributos simultáneamente (caras femeninas sin gafas o caras masculinas con gafas, por ejemplo)

Los rostros de anime que generamos con GAN convolucional profundo (DC-GAN) en el capítulo 4 parecen realistas. Sin embargo, habrás notado que cada imagen generada tiene diferentes atributos, como el color del pelo, el color de los ojos y si la cabeza se inclina hacia la izquierda o la derecha. Quizás te preguntes si hay alguna forma de ajustar el modelo para que las imágenes generadas tengan ciertas características (como el pelo negro y la inclinación hacia la izquierda). Resulta que sí es posible.

En este capítulo, aprenderá dos maneras diferentes de seleccionar características en las imágenes generadas y sus respectivas ventajas y desventajas. El primer método implica seleccionar vectores específicos en el espacio latente. Diferentes vectores corresponden a diferentes características.cs: por ejemplo, un vector podría generar un rostro masculino y otro uno femenino. El segundo método utiliza una GAN condicional (cGAN), que implica entrenar el modelo con datos etiquetados. Esto nos permite indicar al modelo que genere imágenes con una etiqueta específica, cada una representando una característica distinta, como rostros con o sin gafas.

Además, aprenderá a combinar ambos métodos para seleccionar simultáneamente dos atributos independientes de las imágenes. Como resultado, podrá generar cuatro grupos de imágenes diferentes: hombres con gafas, hombres sin gafas, mujeres con gafas y mujeres sin gafas. Para hacerlo más interesante, puede usar una media ponderada de las etiquetas o de los vectores de entrada para generar imágenes que cambien de un atributo a otro. Por ejemplo, puede generar una serie de imágenes en las que las gafas se desvanezcan gradualmente en el rostro de la misma persona (aritmética de etiquetas). O bien, puede generar una serie de imágenes en las que los rasgos masculinos se desvanezcan gradualmente y el rostro masculino se transforme en el rostro femenino (aritmética de vectores).

Poder realizar aritmética vectorial o de etiquetas por separado parece ciencia ficción, y más aún si se realizan ambas simultáneamente. Toda la experiencia nos recuerda la cita de Arthur C. Clarke (autor de *2001: Odisea del Espacio*): «Cualquier tecnología suficientemente avanzada es indistinguible de la magia».

A pesar del realismo de los rostros de anime generados en el capítulo 4, estos se vieron limitados por la baja resolución. El entrenamiento de modelos GAN puede ser complejo y suele verse obstaculizado por problemas como muestras pequeñas o imágenes de baja calidad. Estos desafíos pueden impedir la convergencia de los modelos, lo que resulta en una mala calidad de imagen. Para solucionar esto, analizaremos e implementaremos una técnica de entrenamiento mejorada utilizando la distancia de Wasserstein con penalización de gradiente en nuestro cGAN. Esta mejora da como resultado rostros humanos más realistas y una calidad de imagen notablemente mejor en comparación con el capítulo anterior.

5.1 El conjunto de datos de gafas

En este capítulo, usaremos el conjunto de datos de gafas para entrenar un modelo cGAN. En el siguiente capítulo, también usaremos este conjunto de datos para entrenar un modelo CycleGAN en uno de los ejercicios: convertir una imagen con gafas en una imagen sin gafas y viceversa. En esta sección, aprenderá a descargar el conjunto de datos y preprocesar las imágenes que contiene.

Los programas Python de este capítulo y el siguiente están adaptados de dos excelentes proyectos de código abierto en línea: el proyecto Kaggle de YashikaJain <https://mng.bz/JNVQ> y un repositorio de GitHub de Aladdin Persson <https://mng.bz/w5yg>. Les recomiendo que examinen estos dos proyectos mientras lean este capítulo y el siguiente.

5.1.1 Descarga del conjunto de datos de gafas

El conjunto de datos de gafas que usamos es de Kaggle. Inicia sesión en Kaggle y ve aEnlace <https://mng.bz/q0oz> para descargar la carpeta de imágenes y los dos archivos CSV a la derecha: `train.csv` y `test.csv`. Hay 5000 imágenes en la carpeta `/faces-spring-2020/`. Una vez que tenga los datos, coloque la carpeta de imágenes y los dos archivos CSV dentro de la carpeta `/files/` en su computadora.

A continuación, ordenaremos las fotos en dos subcarpetas: una conteniendo sólo imágenes con gafas y otra con imágenes sin gafas.

Primero, veamos el archivo `train.csv`:

```
!pip install pandas
import pandas as pd

train=pd.read_csv('files/train.csv')          ①
train.set_index('id', inplace=True)            ②
```

① Carga los datos en el archivo `train.csv` como un DataFrame de pandas

② Establece los valores en la columna `id` como índices de observaciones

La celda de código anterior importa el archivo `train.csv` y establece la variable `id` como índice de cada observación. La columna `glasses` del archivo tiene dos valores: 0 o 1, que indican si la imagen contiene gafas (0 significa sin gafas; 1 significa con gafas).

A continuación, separamos las imágenes en dos carpetas diferentes: una que contiene imágenes con gafas y otra que contiene imágenes sin gafas.

Listado 5.1 Ordenación de imágenes con y sin gafas

```
import os, shutil

G='files/glasses/G/'
NoG='files/glasses/NoG/'
os.makedirs(G, exist_ok=True)                                ①
os.makedirs(NoG, exist_ok=True)                             ②
folder='files/faces-spring-2020/faces-spring-2020/'
for i in range(1,4501):
    oldpath=f"{folder}face-{i}.png"
    if train.loc[i]['glasses']==0:                         ③
        newpath=f"{NoG}face-{i}.png"
    elif train.loc[i]['glasses']==1:                        ④
        newpath=f"{G}face-{i}.png"
    shutil.move(oldpath, newpath)
```

① Crea una subcarpeta /archivos/gafas/G/ para contener imágenes con gafas

② Crea una subcarpeta /archivos/gafas/NoG/ para contener imágenes sin gafas

③ Mueve las imágenes etiquetadas como 0 a la carpeta NoG

④ Mueve las imágenes etiquetadas con el número 1 a la carpeta G

En la celda de código anterior, primero usamos la biblioteca `os` para crear dos subcarpetas, `/glasses/G/` y `/glasses/NoG/`, dentro de la carpeta `/files/` de su computadora. Luego, usamos la biblioteca `shutil` para mover las imágenes a ambas carpetas según la etiqueta `glasses` del archivo `train.csv`. Las que tienen la etiqueta 1 se mueven a la carpeta G y las que tienen la etiqueta 0 a la carpeta NoG.

5.1.2 Visualización de imágenes en el conjunto de datos de gafas

La columna de clasificación `glasses` del archivo `train.csv` no es perfecta. Si, por ejemplo, accedes a la subcarpeta G de tu ordenador, verás que la mayoría de las imágenes tienen gafas, pero aproximadamente el 10 % no. De igual forma, si accedes a la subcarpeta NoG, verás que aproximadamente el 10 % sí las tiene. Debes corregir esto manualmente moviendo las imágenes de una carpeta a otra. Esto es importante para nuestra formación posterior, así que debes mover manualmente las imágenes en ambas carpetas para

que una contenga solo imágenes con gafas y la otra, imágenes sin gafas. Bienvenido a la vida de un científico de datos: ¡resolver problemas de datos es parte de la rutina diaria! Primero, visualicemos algunos ejemplos de imágenes con gafas.

Listado 5.2 Visualización de imágenes con gafas

```
import random
import matplotlib.pyplot as plt
from PIL import Image

imgs=os.listdir(G)
random.seed(42)
samples=random.sample(imgs,16)          ①
fig=plt.figure(dpi=200, figsize=(8,2))
for i in range(16):                    ②
    ax = plt.subplot(2, 8, i + 1)
    img=Image.open(f"{G}{samples[i]}")
    plt.imshow(img)
    plt.xticks([])
    plt.yticks([])
plt.subplots_adjust(wspace=-0.01, hspace=-0.01)
plt.show()
```

① Selecciona aleatoriamente 16 imágenes de la carpeta G

② Muestra las 16 imágenes en una cuadrícula de 2×8

Si corrigió manualmente el etiquetado incorrecto de las imágenes en la carpeta G, verá 16 imágenes con gafas después de ejecutar el código del listado 5.2. El resultado se muestra en la figura 5.1.



Figura 5.1 Imágenes de muestra con gafas en el conjunto de datos de entrenamiento

Puede cambiar G a NoG en el listado 5.2 para visualizar 16 imágenes de muestra sin gafas en el conjunto de datos. El código completo se encuentra en el repositorio de GitHub del libro (<https://github.com/markliu/DGAI>). El resultado se muestra en la figura 5.2.



Figura 5.2 Imágenes de muestra sin gafas en el conjunto de datos de entrenamiento

5.2 cGAN y distancia de Wasserstein

Una cGAN es similar a los modelos GAN que se han visto en los capítulos 3 y 4, con la excepción de que se asigna una etiqueta a los datos de entrada. Las etiquetas corresponden a diferentes características en los datos de entrada. Una vez que el modelo GAN entrenado "aprende" a asociar una etiqueta con una característica, se puede introducir un vector de ruido aleatorio con una etiqueta para generar una salida con la característica deseada. ¹

Los modelos GAN suelen presentar problemas como el colapso de modo (el generador encuentra un tipo de salida específico que engaña al discriminador y reduce sus salidas a estos pocos modos, ignorando otras variaciones), gradientes evanescentes y convergencia lenta. Wasserstein GAN (WGAN) introduce la distancia de la máquina de movimiento de tierras (o Wasserstein-1) como función de pérdida, lo que ofrece un flujo de gradiente más suave y un entrenamiento más estable. Mitiga problemas como el colapso de modo. ² Lo implementaremos en el entrenamiento de cGAN en este capítulo. Cabe destacar que WGAN es un concepto independiente de cGAN: utiliza la distancia de Wasserstein para mejorar el proceso de entrenamiento y se puede aplicar a cualquier modelo GAN (como los que creamos en los capítulos 3 y 4). Combinaremos ambos conceptos en una sola configuración para ahorrar espacio.

Otras formas de estabilizar el entrenamiento GAN

Los problemas con el entrenamiento de modelos GAN son más comunes al generar imágenes de alta resolución. La arquitectura del modelo suele ser compleja, con numerosas capas neuronales. Además de las WGAN, las GAN progresivas son otra forma de estabilizar el entrenamiento. Las GAN progresivas mejoran la estabilidad del entrenamiento al descomponer la compleja tarea de generar imágenes de alta resolución en pasos manejables, lo que permite un aprendizaje más controlado y efectivo. Para más detalles, consulte «Crecimiento progresivo de GAN para mejorar la calidad, la estabilidad y la variación» de Karas et al., <https://arxiv.org/abs/1710.10196>.

5.2.1 WGAN con penalización de gradiente

WGAN es una técnica utilizada para mejorar la estabilidad y el rendimiento del entrenamiento de los modelos GAN. Las GAN convencionales (como las que se han visto en los capítulos 3 y 4) tienen dos componentes: un generador y un discriminador. El generador crea datos falsos, mientras que el discriminador evalúa si los datos son reales o falsos. El entrenamiento implica un juego competitivo de suma cero en el que el generador intenta engañar al discriminador, y este intenta clasificar con precisión las instancias de datos reales y falsos.

Los investigadores han propuesto utilizar la distancia de Wasserstein (una medida de disimilitud entre dos distribuciones) en lugar de la entropía cruzada binaria como función de pérdida para estabilizar el entrenamiento con un término de penalización de gradiente.³ Esta técnica ofrece un flujo de gradiente más suave y mitiga problemas como el colapso modal. La Figura 5.3 muestra un diagrama de WGAN. Como se puede observar a la derecha de la figura, las pérdidas asociadas con las imágenes reales y falsas son pérdidas de Wasserstein en lugar de la pérdida de entropía cruzada binaria habitual.

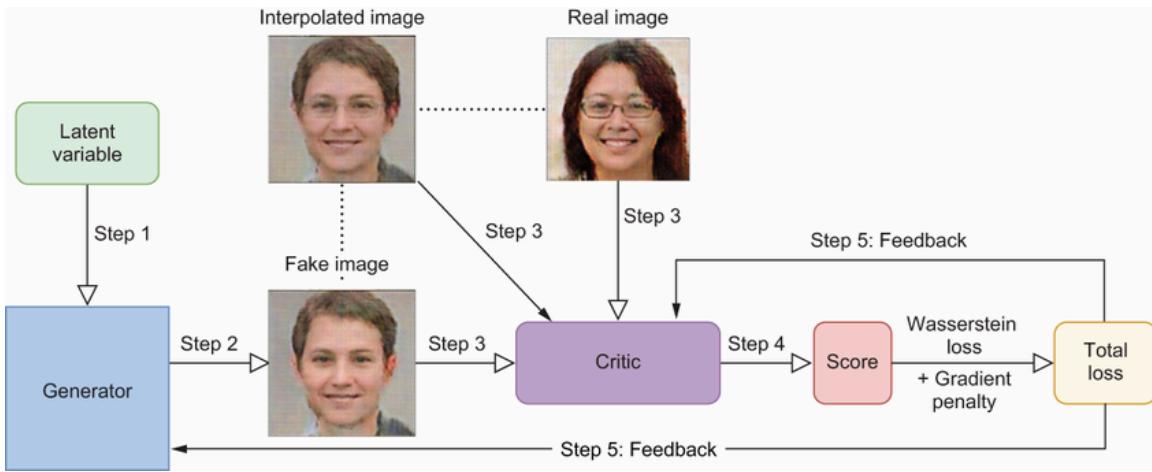


Figura 5.3 WGAN con penalización de gradiente. La red discriminadora en WGAN (que llamamos crítico) clasifica la imagen de entrada.s: Intenta asignar una puntuación de $-\infty$ a una imagen falsa (abajo a la izquierda) y una puntuación de ∞ a la imagen real (arriba en el centro). Además, se presenta al crítico una imagen interpolada de las imágenes real y falsa (arriba a la izquierda), y la penalización de gradiente con respecto a la calificación del crítico en la imagen interpolada se suma a la pérdida total en el proceso de entrenamiento.

Además, para que la distancia de Wasserstein funcione correctamente, el discriminador (denominado crítico en las WGAN) debe ser continuo 1-Lipschitz, lo que significa que las normas de gradiente de la función del crítico deben ser como máximo 1 en todas partes. El artículo original de la WGAN propuso el recorte de pesos para aplicar la restricción de Lipschitz.

Para abordar los problemas de recorte de peso, se añade la penalización de gradiente a la función de pérdida para aplicar la restricción de Lipschitz de forma más eficaz. Para implementar WGAN con penalización de gradiente, primero muestreamos aleatoriamente puntos a lo largo de la línea recta entre los puntos de datos reales y generados (como se indica en la imagen interpolada en la parte superior izquierda de la figura 5.3). Dado que tanto las imágenes reales como las falsas tienen etiquetas adjuntas, la imagen interpolada también tiene una etiqueta adjunta, que es el valor interpolado de las dos etiquetas originales. A continuación, calculamos el gradiente de la salida del crítico con respecto a estos puntos muestreados. Finalmente, añadimos una penalización a la función de pérdida proporcional a la desviación de estas normas de gradiente de 1 (el término de penalización se denomina penalización de gradiente). Es decir, la penalización de gradiente en las WGAN es una técnica para mejorar la estabilidad del entrenamiento y la calidad de la muestra al aplicar la restricción de Lipschitz de forma más eficaz, abordando las limitaciones del modelo WGAN original.

5.2.2 cGAN

cGAN es una extensión del marco básico de GAN. En una cGAN, tanto el generador como el discriminador (o el crítico, ya que implementamos WGAN y cGAN en el mismo entorno) están condicionados a información adicional. Esta puede ser cualquier cosa, como etiquetas de clase, datos de otras modalidades o incluso descripciones textuales. Este condicionamiento se logra típicamente introduciendo esta información adicional tanto en el generador como en el discriminador. En nuestro entorno, añadiremos etiquetas de clase a las entradas tanto del generador como del crítico: asignamos una etiqueta a las imágenes con gafas y otra a las imágenes sin gafas. La Figura 5.4 muestra un diagrama del proceso de entrenamiento de las cGAN.

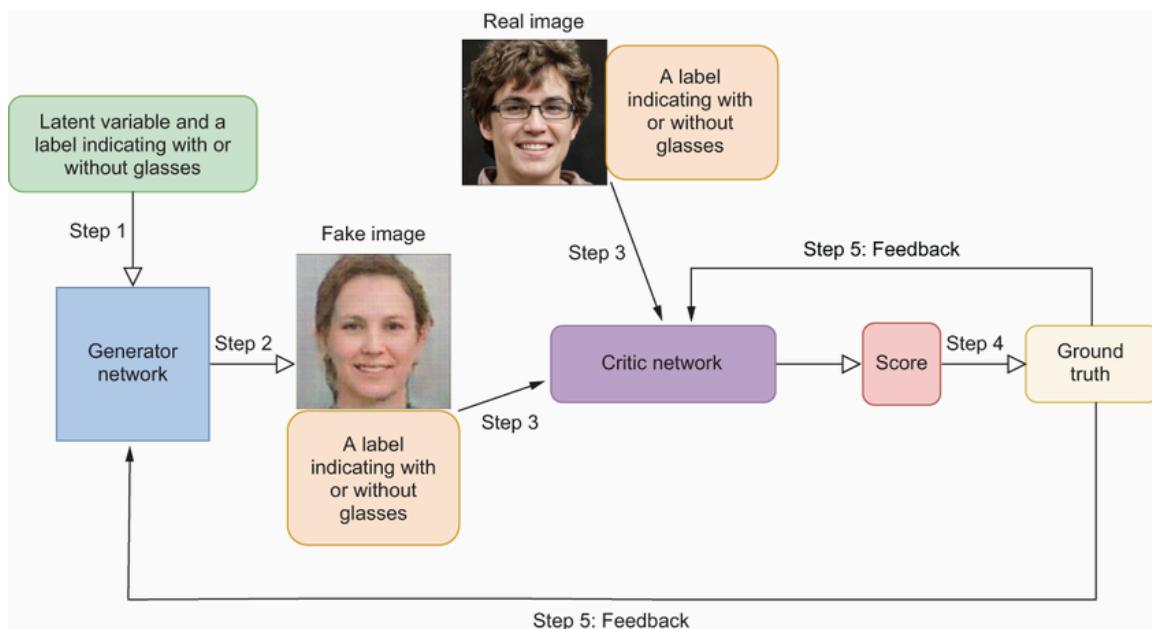


Figura 5.4 El proceso de entrenamiento para cGAN

Como se puede ver en la parte superior izquierda de la figura 5.4, en una cGAN, el generador recibe como entrada un vector de ruido aleatorio y la información condicional (una etiqueta que indica si la imagen tiene gafas o no). Utiliza esta información para generar datos que no solo parecen reales, sino que también se alinean con la entrada condicional.

El crítico recibe datos reales del conjunto de entrenamiento o datos falsos generados por el generador, junto con la información condicional (una etiqueta que indica si la imagen tiene gafas o no en nuestro entorno). Su tarea es determinar si los datos dados son reales o falsos, considerando la información condicional (¿la imagen generada tiene gafas?). En la figura 5.4, utilizamos la red crítica en lugar de la red discriminadora, ya que implementa-

mos cGAN y WGAN simultáneamente. Sin embargo, el concepto de cGAN también se aplica a las GAN tradicionales.

La principal ventaja de las cGAN es su capacidad para seleccionar aspectos de los datos generados, lo que las hace más versátiles y aplicables en escenarios donde la salida debe dirigirse o condicionarse a ciertos parámetros de entrada. En nuestro caso, entrenaremos la cGAN para que podamos seleccionar si las imágenes generadas tienen gafas o no.

En resumen, las cGAN son una poderosa extensión de la arquitectura GAN básica, que permite la generación específica de datos sintéticos basados en entradas condicionales.

5.3 Crear una cGAN

En esta sección, aprenderá a crear una cGAN para generar rostros humanos con o sin gafas. También aprenderá a implementar la WGAN con penalización de gradiente para estabilizar el entrenamiento.

El generador en las cGAN utiliza no solo vectores de ruido aleatorio, sino también información condicional, como etiquetas, como entrada para crear imágenes con o sin gafas. Además, una red crítica en las WGAN es diferente de la red discriminadora en las GAN tradicionales. En esta sección, también aprenderá a calcular la distancia de Wasserstein y la penalización de gradiente.

5.3.1 Una crítica en cGAN

En las cGAN, el discriminador es un clasificador binario que identifica la entrada como real o falsa, según la etiqueta. En las WGAN, la red discriminadora se denomina crítico. Este evalúa la entrada y otorga una puntuación entre $-\infty$ e ∞ . Cuanto mayor sea la puntuación, mayor será la probabilidad de que la entrada provenga del conjunto de entrenamiento (es decir, real).

El Listado 5.3 crea la red crítica. La arquitectura es similar a la de la red discriminadora que usamos en el capítulo 4 al generar imágenes a color de rostros de anime. En particular, usamos siete capas `Conv2d` en PyTorch para reducir gradualmente el tamaño de la entrada, de modo que la salida sea un único valor entre $-\infty$ e ∞ .

```

class Critic(nn.Module):
    def __init__(self, img_channels, features):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(img_channels, features,
                     kernel_size=4, stride=2, padding=1), (1)
            nn.LeakyReLU(0.2),
            self.block(features, features * 2, 4, 2, 1),
            self.block(features * 2, features * 4, 4, 2, 1),
            self.block(features * 4, features * 8, 4, 2, 1),
            self.block(features * 8, features * 16, 4, 2, 1),
            self.block(features * 16, features * 32, 4, 2, 1),
            nn.Conv2d(features * 32, 1, kernel_size=4,
                     stride=2, padding=0)) (2)
    def block(self, in_channels, out_channels,
              kernel_size, stride, padding):
        return nn.Sequential((3)
            nn.Conv2d(in_channels,out_channels,
                     kernel_size,stride,padding,bias=False,),
            nn.InstanceNorm2d(out_channels, affine=True),
            nn.LeakyReLU(0.2))
    def forward(self, x):
        return self.net(x)

```

① La red crítica tiene dos capas Conv2d más cinco bloques.

② La salida tiene una función, sin activación.

③ Cada bloque contiene una capa Conv2d, una capa InstanceNorm2d, con activación LeakyReLU.

La entrada a la red crítica es una imagen a color con una forma de $5 \times 256 \times 256$. Los tres primeros canales son de color (rojo, verde y azul). Los dos últimos (cuarto y quinto) son canales de etiqueta que indican al crítico si la imagen es con o sin gafas. Analizaremos el mecanismo exacto para lograr esto en la siguiente sección.

La red crítica consta de siete capas Conv2d. En el capítulo 4, analizamos a fondo su funcionamiento. Se utilizan para la extracción de características mediante la aplicación de un conjunto de filtros aprendibles en las imáge-

nes de entrada para detectar patrones y características a diferentes escalas espaciales, capturando eficazmente representaciones jerárquicas de los datos de entrada. El crítico evalúa las imágenes de entrada basándose en estas representaciones. Las cinco capas `Conv2d` centrales van seguidas de una capa `InstanceNorm2d` y una activación `LeakyReLU`; por lo tanto, definimos un método `block()` para optimizar la red crítica. Esta capa `InstanceNorm2d` es similar a la `BatchNorm2d` que analizamos en el capítulo 4, salvo que normalizamos cada instancia individual del lote de forma independiente.

Otro punto clave es que la salida ya no es un valor entre 0 y 1 ya que no utilizamos la activación sigmoidea en la última capa de la red crítica. En cambio, la salida es un valor entre $-\infty$ y ∞ , ya que utilizamos la distancia de Wasserstein con penalización de gradiente en nuestro cGAN.

5.3.2 Un generador en cGAN

En las WGAN, la función del generador es crear instancias de datos para que el crítico pueda evaluarlas con una puntuación alta. En las cGAN, el generador debe generar instancias de datos con información condicional (con o sin gafas en nuestro entorno). Dado que estamos implementando una cGAN con distancia de Wasserstein, le indicaremos al generador el tipo de imágenes que queremos generar asignando una etiqueta al vector de ruido aleatorio. Analizaremos el mecanismo exacto en la siguiente sección.

Creamos la red neuronal que se muestra en el siguiente listado para representar el generador.

Listado 5.4 Un generador en cGAN

```
class Generator(nn.Module):
    def __init__(self, noise_channels, img_channels, features):
        super(Generator, self).__init__()
        self.net = nn.Sequential(
            self.block(noise_channels, features * 64, 4, 1, 0),
            self.block(features * 64, features * 32, 4, 2, 1),
            self.block(features * 32, features * 16, 4, 2, 1),
            self.block(features * 16, features * 8, 4, 2, 1),
            self.block(features * 8, features * 4, 4, 2, 1),
            self.block(features * 4, features * 2, 4, 2, 1),
            nn.ConvTranspose2d(
```

```

        features * 2, img_channels, kernel_size=4,
        stride=2, padding=1),
    nn.Tanh())
def block(self, in_channels, out_channels,
          kernel_size, stride, padding):
    return nn.Sequential(
        nn.ConvTranspose2d(in_channels,out_channels,
                          kernel_size,stride,padding,bias=False,),
        nn.BatchNorm2d(out_channels),
        nn.ReLU())
def forward(self, x):
    return self.net(x)

```

① El generador consta de siete capas ConvTranspose2d.

② Utiliza la activación de Tanh para comprimir los valores en el rango [-1, 1], igual que las imágenes en el conjunto de entrenamiento

③ Cada bloque consta de una capa ConvTranspose2d, una capa BatchNorm2d y una activación ReLU.

Introduciremos un vector de ruido aleatorio de un espacio latente de 100 dimensiones como entrada al generador. También introduciremos una etiqueta de imagen codificada one-hot de dos valores para indicarle que genere una imagen con o sin gafas. Concatenaremos ambos datos para formar una variable de entrada de 102 dimensiones para el generador. El generador genera una imagen en color basándose en la entrada del espacio latente y la información de etiquetado.

La red generadora consta de siete capas ConvTranspose2d, y la idea es replicar los pasos de la red crítica para generar imágenes, como se explicó en el capítulo 4. Las primeras seis capas ConvTranspose2d van seguidas de una capa BatchNorm2d y una activación ReLU; por lo tanto, definimos un método block() en la red generadora para simplificar la arquitectura.

Como se hizo en el capítulo 4, utilizamos la función de activación Tanh en la capa de salida, de modo que los píxeles de salida estén todos en el rango de -1 y 1, igual que las imágenes del conjunto de entrenamiento.

5.3.3 Inicialización de peso y función de penalización de gradiente

En el aprendizaje profundo, los pesos en las redes neuronales se inicializan aleatoriamente. Cuando la arquitectura de la red es compleja y existen muchas capas ocultas (como en nuestro caso), la inicialización de los pesos es crucial.

Por lo tanto, definimos la siguiente función `weights_init()` para inicializar pesos tanto en la red generadora como en la crítica:

```
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)
```

La función inicializa pesos en las capas `Conv2d` y `ConvTranspose2d` con valores de una distribución normal con una media de 0 y una desviación típica de 0,02. También inicializa pesos en las capas `BatchNorm2d` con valores de una distribución normal con una media de 1 y una desviación típica de 0,02. Se elige una desviación típica pequeña en la inicialización de pesos para evitar gradientes explosivos.

A continuación, creamos un generador y un crítico basados en las clases `Generator()` y `Critic()` que definimos en la subsección anterior. Inicializamos sus pesos según la función `weights_init()` definida anteriormente:

```
z_dim=100
img_channels=3
features=16
gen=Generator(z_dim+2,img_channels,features).to(device)
critic=Critic(img_channels+2,features).to(device)
gen.apply(weights_init)
critic.apply(weights_init)
```

Como de costumbre, utilizaremos el optimizador Adam tanto para el crítico como para el generador:

```

lr = 0.0001
opt_gen = torch.optim.Adam(gen.parameters(),
                           lr = lr, betas=(0.0, 0.9))
opt_critic = torch.optim.Adam(critic.parameters(),
                             lr = lr, betas=(0.0, 0.9))

```

El generador intenta crear imágenes indistinguibles de las del conjunto de entrenamiento con la etiqueta dada. Presenta las imágenes al crítico para obtener altas calificaciones en las imágenes generadas. El crítico, por otro lado, intenta asignar altas calificaciones a las imágenes reales y bajas a las falsas, dependiendo de la etiqueta dada. Específicamente, la función de pérdida del crítico tiene tres componentes:

$$\text{valor_crítico(falso)} - \text{valor_crítico(real)} + \text{peso} \times \text{Penalización_de_gradiente}$$

El primer término, *critic_value(fake)*, indica que si una imagen es falsa, el objetivo del crítico es identificarla como falsa y asignarle una calificación baja. El segundo término, *- critic_value(real)*, indica que si la imagen es real, el objetivo del crítico es identificarla como real y asignarle una calificación alta. Además, el crítico busca minimizar el término de penalización de gradiente, *weight × GradientPenalty*, donde *weight* es una constante que determina la penalización que se asigna a las desviaciones de las normas de gradiente con respecto al valor 1. La penalización de gradiente se calcula como se muestra en la siguiente lista.

Listado 5.5 Cálculo de la penalización de gradiente

```

def GP(critic, real, fake):
    B, C, H, W = real.shape
    alpha=torch.rand((B,1,1,1)).repeat(1,C,H,W).to(device)
    interpolated_images = real*alpha+fake*(1-alpha)          ①
    critic_scores = critic(interpolated_images)             ②
    gradient = torch.autograd.grad(
        inputs=interpolated_images,
        outputs=critic_scores,
        grad_outputs=torch.ones_like(critic_scores),
        create_graph=True,
        retain_graph=True)[0]                                ③
    gradient = gradient.view(gradient.shape[0], -1)
    gradient_norm = gradient.norm(2, dim=1)

```

```
gp = torch.mean((gradient_norm - 1) ** 2)  
return gp
```

(4)

- ① Crea una imagen interpolada de lo real y lo falso.
- ② Obtiene el valor crítico con respecto a la imagen interpolada
- ③ Calcula el gradiente del valor crítico
- ④ La penalización de gradiente es la desviación al cuadrado de la norma de gradiente con respecto al valor 1.

En la función `GP()`, primero creamos imágenes interpoladas de imágenes reales y falsas. Esto se realiza mediante un muestreo aleatorio de puntos a lo largo de la línea recta entre las imágenes reales y las generadas. Imagine un control deslizante: en un extremo está la imagen real y en el otro, la imagen falsa. Al mover el control deslizante, se observa una fusión continua entre la imagen real y la falsa, donde las imágenes interpoladas representan las etapas intermedias.

A continuación, presentamos imágenes interpoladas a la red crítica para obtener calificaciones y calcular el gradiente de la salida del crítico con respecto a las imágenes interpoladas. Finalmente, la penalización de gradiente se calcula como la desviación al cuadrado de las normas de gradiente con respecto al valor objetivo de 1.

5.4 Entrenamiento del cGAN

Como mencionamos en la última sección, necesitamos encontrar una manera de decirle tanto al crítico como al generador cuál es la etiqueta de la imagen para que sepan si la imagen tiene anteojos o no.

En esta sección, aprenderá primero a agregar etiquetas a las entradas de la red crítica y a las entradas de la red generadora para que el generador sepa qué tipo de imágenes crear, mientras que el crítico puede evaluar las imágenes según las etiquetas. Después, aprenderá a entrenar la cGAN con la distancia de Wasserstein.

5.4.1 Agregar etiquetas a las entradas

Primero preprocesamos los datos y convertimos las imágenes en tensores de torch:

```
import torchvision.transforms as T
import torchvision

batch_size=16
imgsz=256
transform=T.Compose([
    T.Resize((imgsz,imgsz)),
    T.ToTensor(),
    T.Normalize([0.5,0.5,0.5],[0.5,0.5,0.5]))]
data_set=torchvision.datasets.ImageFolder(
    root=r"files/glasses",
    transform=transform)
```

Establecemos el tamaño del lote en 16 y el tamaño de la imagen en 256 x 256 píxeles. Los valores de píxeles se eligen para que las imágenes generadas tengan resoluciones más altas que las del capítulo anterior (64 x 64 píxeles). Elegimos un tamaño de lote de 16, menor que el del capítulo 3, debido al mayor tamaño de la imagen. Si el tamaño del lote es demasiado grande, la GPU (o incluso la CPU) se quedará sin memoria.

CONSEJO: Si usas entrenamiento de GPU y esta tiene poca memoria (por ejemplo, 6 GB), considera reducir el tamaño del lote a un número menor a 16, como 10 u 8, para que tu GPU no se quede sin memoria. Como alternativa, puedes mantener el tamaño del lote en 16, pero cambiar al entrenamiento de CPU para solucionar el problema de memoria de la GPU.

A continuación, añadiremos etiquetas a los datos de entrenamiento. Dado que existen dos tipos de imágenes (con gafas y sin gafas), crearemos dos etiquetas de imagen one-hot. Las imágenes con gafas tendrán una etiqueta one-hot de [1, 0], y las imágenes sin gafas, de [0, 1].

La entrada al generador es un vector de ruido aleatorio de 100 valores. Concatenamos la etiqueta one-hot con el vector de ruido aleatorio y alimentamos la entrada de 102 valores al generador. La entrada a la red crítica es una imagen a color de tres canales con una forma de 3 por 256 por 256 (Py-

Torch usa tensores de canal primero para representar imágenes). ¿Cómo adjuntamos una etiqueta con una forma de 1 por 2 a una imagen con una forma de 3 por 256 por 256? La solución es agregar dos canales a la imagen de entrada para que la forma de la imagen cambie de (3, 256, 256) a (5, 256, 256): los dos canales adicionales son las etiquetas one-hot. Específicamente, si una imagen tiene gafas, el cuarto canal se llena con 1 y el quinto canal con 0; si la imagen no tiene gafas, el cuarto canal se llena con 0 y el quinto canal con 1.

Creación de etiquetas si hay más de dos valores en una característica

El modelo cGAN se puede extender fácilmente a características con más de dos valores. Por ejemplo, si se crea un modelo para generar imágenes con diferentes colores de pelo: negro, rubio y blanco, las etiquetas de imagen que se introducen en el generador pueden tener los valores [1, 0, 0], [0, 1, 0] y [0, 0, 1], respectivamente. Se pueden asociar tres canales a la imagen de entrada antes de introducirla en el discriminador o crítico. Por ejemplo, si una imagen tiene pelo negro, el cuarto canal se llena con unos y los canales quinto y sexto con ceros.

Además, en el ejemplo de las gafas, dado que la etiqueta solo tiene dos valores, se pueden usar los valores 0 y 1 para indicar imágenes con y sin gafas al introducir la etiqueta en el generador. Se puede asignar un canal a la imagen de entrada antes de introducirla en el crítico: si una imagen tiene gafas, el cuarto canal se llena con unos; si la imagen no tiene gafas, el cuarto canal se llena con ceros. Les dejo esto como ejercicio. La solución se encuentra en el repositorio de GitHub del libro:

<https://github.com/markhliu/DGAI>

Implementamos este cambio como se muestra en el siguiente listado.

Listado 5.6 Adjuntar etiquetas a las imágenes de entrada

```
newdata=[]
for i,(img,label) in enumerate(data_set):
    onehot=torch.zeros((2))
    onehot[label]=1
    channels=torch.zeros((2,imgsz,imgsz))           ①
    if label==0:
        channels[0,:,:,:]=1                         ②
```

```
else:  
    channels[1,:,:]=1  
    img_and_label=torch.cat([img,channels],dim=0)  
    newdata.append((img,label,onehot,img_and_label))
```

① Crea dos canales adicionales llenos de 0, cada canal con una forma de 256 por 256, la misma que la dimensión de cada canal en la imagen de entrada

② Si la etiqueta de la imagen original es 0, llena el cuarto canal con 1s

③ Si la etiqueta de la imagen original es 1, llena el quinto canal con 1s

④ Agrega el cuarto y quinto canal a la imagen original para formar una imagen etiquetada de cinco canales

CONSEJO : Al cargar las imágenes con el método

`torchvision.datasets.ImageFolder()` de la carpeta /files/glasses, PyTorch asigna etiquetas a las imágenes de cada subcarpeta en orden alfabético. Por lo tanto, a las imágenes en /files/glasses/G/ se les asigna la etiqueta 0, y a las de /files/glasses/NoG/, la etiqueta 1.

Primero creamos una lista vacía `newdata` para almacenar imágenes con etiquetas. Creamos un tensor de PyTorch con forma (2, 256, 256) que se adjuntará a la imagen de entrada original para formar una nueva imagen con forma (5, 256, 256). Si la etiqueta de la imagen original es 0 (esto significa que las imágenes provienen de la carpeta /files/glasses/G/), llenamos el cuarto canal con 1 y el quinto con 0 para que el crítico sepa que es una imagen con gafas. Por otro lado, si la etiqueta de la imagen original es 1 (esto significa que las imágenes provienen de la carpeta /files/glasses/NoG/), llenamos el cuarto canal con 0 y el quinto con 1 para que el crítico sepa que es una imagen sin gafas.

Creamos un iterador de datos con lotes (para mejorar la eficiencia computacional, el uso de memoria y la dinámica de optimización en el proceso de entrenamiento) de la siguiente manera:

```
data_loader=torch.utils.data.DataLoader(  
    newdata,batch_size=batch_size,shuffle=True)
```

5.4.2 Entrenamiento del cGAN

Ahora que tenemos los datos de entrenamiento y dos redes, entrenaremos la cGAN. Utilizaremos inspecciones visuales para determinar cuándo debe finalizar el entrenamiento.

Una vez entrenado el modelo, descartaremos la red crítica y usaremos el generador para crear imágenes con una determinada característica (con o sin gafas, en nuestro caso).

Crearemos una función para probar periódicamente cómo se ven las imágenes generadas.

Listado 5.7 Inspección de imágenes generadas

```
def plot_epoch(epoch):
    noise = torch.randn(32, z_dim, 1, 1)
    labels = torch.zeros(32, 2, 1, 1)
    labels[:,0,:,:]=1
    noise_and_labels=torch.cat([noise,labels],dim=1).to(device)①
    fake=gen(noise_and_labels).cpu().detach()②
    fig=plt.figure(figsize=(20,10),dpi=100)
    for i in range(32):
        ax = plt.subplot(4, 8, i + 1)
        img=(fake.cpu().detach()[i]/2+0.5).permute(1,2,0)
        plt.imshow(img)
        plt.xticks([])
        plt.yticks([])
        plt.subplots_adjust(hspace=-0.6)
        plt.savefig(f"files/glasses/G{epoch}.png")③
        plt.show()
    noise = torch.randn(32, z_dim, 1, 1)
    labels = torch.zeros(32, 2, 1, 1)
    labels[:,1,:,:]=1
    ... (code omitted)④
```

① Crea una etiqueta one-hot para imágenes con gafas

② Alimenta el vector de ruido concatenado y la etiqueta al generador para crear imágenes con gafas

③ Traza las imágenes generadas con gafas

④ Crea una etiqueta one-hot para imágenes sin gafas

Después de cada época de entrenamiento, le pediremos al generador que cree un conjunto de imágenes con gafas y un conjunto de imágenes sin gafas. Luego, graficamos las imágenes para poder inspeccionarlas visualmente. Para crear imágenes con gafas, primero creamos etiquetas one-hot [1, 0] y las asociamos a los vectores de ruido aleatorio antes de alimentar el vector concatenado a la red del generador. El generador crea imágenes con gafas, ya que la etiqueta es [1, 0] en lugar de [0, 1]. Luego, graficamos las imágenes generadas en cuatro filas y ocho columnas y guardamos los subgráficos en su computadora. El proceso de creación de imágenes sin gafas es similar, excepto que usamos la etiqueta one-hot [0, 1] en lugar de [1, 0]. Omití parte del código en el listado 5.7, pero puede encontrarlo en el repositorio de GitHub del libro: <https://github.com/markliu/DGAI>.

Definimos una función `train_batch()` para entrenar el modelo con un lote de datos.

Listado 5.8 Entrenamiento del modelo con un lote de datos

```
def train_batch(onehots,img_and_labels,epoch):
    real = img_and_labels.to(device)
    B = real.shape[0]
    for _ in range(5):
        noise = torch.randn(B, z_dim, 1, 1)
        onehots=onehots.reshape(B,2,1,1)
        noise_and_labels=torch.cat([noise,onehots],dim=1).to(device)
        fake_img = gen(noise_and_labels).to(device)
        fakelabels=img_and_labels[:,3:,:,:].to(device)
        fake=torch.cat([fake_img,fakelabels],dim=1).to(device) ②
        critic_real = critic(real).reshape(-1)
        critic_fake = critic(fake).reshape(-1)
        gp = GP(critic, real, fake)
        loss_critic=(-(torch.mean(critic_real) -
                      torch.mean(critic_fake)) + 10 * gp) ③
        critic.zero_grad()
        loss_critic.backward(retain_graph=True)
        opt_critic.step()
        gen_fake = critic(fake).reshape(-1)
        loss_gen = -torch.mean(gen_fake) ④
        gen.zero_grad()
        loss_gen.backward()
```

```
    opt_gen.step()  
    return loss_critic, loss_gen
```

- ① Un lote de imágenes reales con etiquetas
- ② Un lote de imágenes generadas con etiquetas
- ③ La pérdida total para el crítico tiene tres componentes: pérdida por evaluar imágenes reales, pérdida por evaluar imágenes falsas y pérdida por penalización de gradiente.
- ④ Entrena el generador con la pérdida de Wasserstein

En la función `train_batch()`, primero entrenamos al crítico con imágenes reales. También le pedimos al generador que cree un lote de datos falsos con la etiqueta dada. Luego, entrenamos al crítico con imágenes falsas. En la función `train_batch()`, también entrenamos al generador con un lote de datos falsos.

NOTA La pérdida para el crítico tiene tres componentes: pérdida por evaluar imágenes reales, pérdida por evaluar imágenes falsas y pérdida por penalización de gradiente.

Ahora entrenamos el modelo para 100 épocas:

```
for epoch in range(1,101):  
    closs=0  
    gloss=0  
    for _,_,onehots,img_and_labels in data_loader:  
        loss_critic, loss_gen = train_batch(onehots,\n                                              img_and_labels,epoch)  
        closs+=loss_critic.detach()/len(data_loader)  
        gloss+=loss_gen.detach()/len(data_loader)  
    print(f"at epoch {epoch},\n        critic loss: {closs}, generator loss {gloss}")  
    plot_epoch(epoch)  
torch.save(gen.state_dict(),'files/cgan.pth')
```

- ①
- ②
- ③

- ① Itera a través de todos los lotes en el conjunto de datos de entrenamiento
- ② Entrena el modelo con un lote de datos

③ Guarda los pesos en el generador entrenado

Después de cada época de entrenamiento, imprimimos la pérdida crítica y la pérdida del generador para asegurar que se mantengan dentro de un rango razonable. También generamos 32 imágenes de rostros con gafas y 32 imágenes sin gafas mediante la función `plot_epoch()` definida anteriormente. Guardamos los pesos del generador entrenado en la carpeta local una vez finalizado el entrenamiento para poder generar imágenes posteriormente con el modelo entrenado.

Este entrenamiento dura unos 30 minutos si usas entrenamiento con GPU. De lo contrario, podría tardar varias horas, dependiendo de la configuración de hardware de tu ordenador. También puedes descargar el modelo entrenado desde mi sitio web:

<https://gattonweb.uky.edu/faculty/lium/gai/cgan.zip>. Descomprime el archivo después de descargarlo.

5.5 Selección de características en imágenes generadas

Hay al menos dos maneras de generar imágenes con una característica específica. La primera es etiquetar un vector de ruido aleatorio antes de alimentarlo al modelo cGAN entrenado. Diferentes etiquetas generan diferentes características en la imagen generada (en nuestro caso, si la imagen tiene gafas). La segunda manera es seleccionar el vector de ruido que se alimenta al modelo entrenado: un vector genera una imagen con un rostro masculino, mientras que otro genera una imagen con un rostro femenino. Tenga en cuenta que este segundo método funciona incluso en una GAN tradicional, como las que entrenamos en el capítulo 4. También funciona en una cGAN.

Mejor aún, en esta sección, aprenderá a combinar estos dos métodos para poder seleccionar dos características simultáneamente: una imagen de un rostro masculino con gafas o un rostro femenino sin gafas, etc.

Existen ventajas y desventajas para cada uno de estos dos métodos a la hora de seleccionar una característica específica en las imágenes generadas. El primer método, la cGAN, requiere datos etiquetados para entrenar el modelo. En ocasiones, la selección de datos etiquetados es costosa. Sin em-

bargo, una vez que se ha entrenado correctamente una cGAN, se puede generar una amplia gama de imágenes con una característica específica. En nuestro caso, se pueden generar muchas imágenes diferentes con gafas (o sin gafas); cada una es diferente de la otra. El segundo método, la selección manual de un vector de ruido, no requiere datos etiquetados para entrenar el modelo. Sin embargo, cada vector de ruido seleccionado manualmente solo puede generar una imagen. Si se desea generar muchas imágenes diferentes con la misma característica que la cGAN, se deberán seleccionar manualmente muchos vectores de ruido diferentes ex ante.

5.5.1 Selección de imágenes con o sin gafas

Al adjuntar una etiqueta de [1, 0] o [0, 1] a un vector de ruido aleatorio antes de introducirlo en el modelo cGAN entrenado, puede seleccionar si la imagen generada tiene anteojos.

Primero, usaremos el modelo entrenado para generar 32 imágenes con gafas y las representaremos en una cuadrícula de 4×8 . Para que los resultados sean reproducibles, corregiremos el estado aleatorio en PyTorch. Además, usaremos el mismo conjunto de vectores de ruido aleatorio para observar el mismo conjunto de caras.

Fijamos el estado aleatorio en la semilla 0 y generamos 32 imágenes de caras con gafas.

Listado 5.9 Generación de imágenes de rostros humanos con gafas

```
torch.manual_seed(0)                                     ①

generator=Generator(z_dim+2,img_channels,features).to(device)
generator.load_state_dict(torch.load("files/cgan.pth",
    map_location=device))                            ②
generator.eval()

noise_g=torch.randn(32, z_dim, 1, 1)                  ③
labels_g=torch.zeros(32, 2, 1, 1)
labels_g[:,0,:,:]=1
noise_and_labels=torch.cat([noise_g,labels_g],dim=1).to(device)
fake=generator(noise_and_labels)
plt.figure(figsize=(20,10),dpi=50)
for i in range(32):
```

```
ax = plt.subplot(4, 8, i + 1)
img=(fake.cpu().detach()[i]/2+0.5).permute(1,2,0)
plt.imshow(img.numpy())
plt.xticks([])
plt.yticks([])
plt.subplots_adjust(wspace=-0.08, hspace=-0.01)
plt.show()
```

- ① Corrige el estado aleatorio para que los resultados sean reproducibles
- ② Carga las pesas entrenadas
- ③ Genera un conjunto de vectores de ruido aleatorio y lo guarda para que podamos seleccionar ciertos vectores para realizar aritmética vectorial.
- ④ Crea una etiqueta para generar imágenes con gafas

Creamos otra instancia de la clase `Generator()` y la llamamos `generator`. Luego, cargamos los pesos entrenados que guardamos en la carpeta local en la sección anterior (o puedes descargarlos de mi sitio web:

<https://mng.bz/75Z4>). Para generar 32 imágenes de rostros humanos con gafas, primero dibujamos 32 vectores de ruido aleatorio en el espacio latente. También crearemos un conjunto de etiquetas y las llamaremos `labels_g`, que le indican al generador que produzca 32 imágenes con gafas.

Si ejecuta el programa del listado 5.9, verá 32 imágenes como se muestra en la figura 5.5.

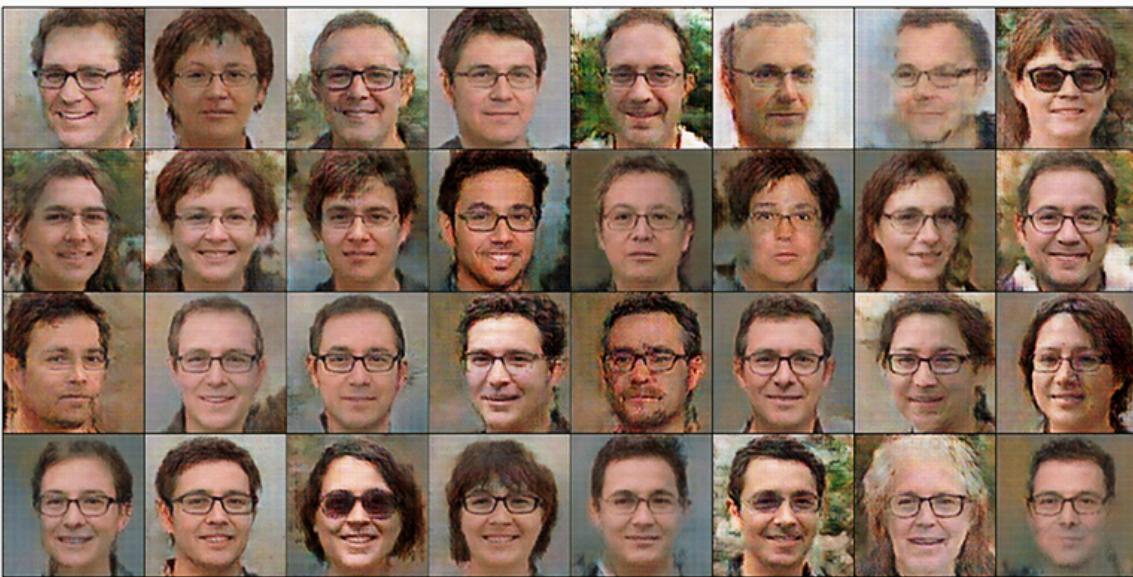


Figura 5.5 Imágenes de rostros humanos con gafas generadas por el modelo cGAN entrenado

En primer lugar, las 32 imágenes sí contienen gafas. Esto indica que el modelo cGAN entrenado puede generar imágenes según las etiquetas proporcionadas. Habrá notado que algunas imágenes tienen rasgos masculinos, mientras que otras tienen rasgos femeninos. Para prepararnos para la aritmética vectorial en la siguiente subsección, seleccionaremos un vector de ruido aleatorio que genere una imagen con rasgos masculinos y otro con rasgos femeninos. Tras inspeccionar las 32 imágenes de la figura 5.5, seleccionamos imágenes con valores de índice 0 y 14, de la siguiente manera:

```
z_male_g=noise_g[0]  
z_female_g=noise_g[14]
```

Para generar 32 imágenes sin gafas, primero producimos otro conjunto de vectores de ruido aleatorio y etiquetas:

```
noise_ng = torch.randn(32, z_dim, 1, 1)  
labels_ng = torch.zeros(32, 2, 1, 1)  
labels_ng[:,1,:,:]=1
```

El nuevo conjunto de vectores de ruido aleatorio se llama `noise_ng`, y el nuevo conjunto de etiquetas `labels_ng`. Introdúzcalos en el generador y debería ver 32 imágenes sin gafas, como se muestra en la figura 5.6.

Ninguno de los 32 rostros de la figura 5.6 tiene gafas: el modelo cGAN entrenado puede generar imágenes según la etiqueta dada. Seleccionamos imáge-

nes con índices 8 (masculino) y 31 (femenino) para preparar la aritmética vectorial en la siguiente subsección:

```
z_male_ng=noise_ng[8]
z_female_ng=noise_ng[31]
```

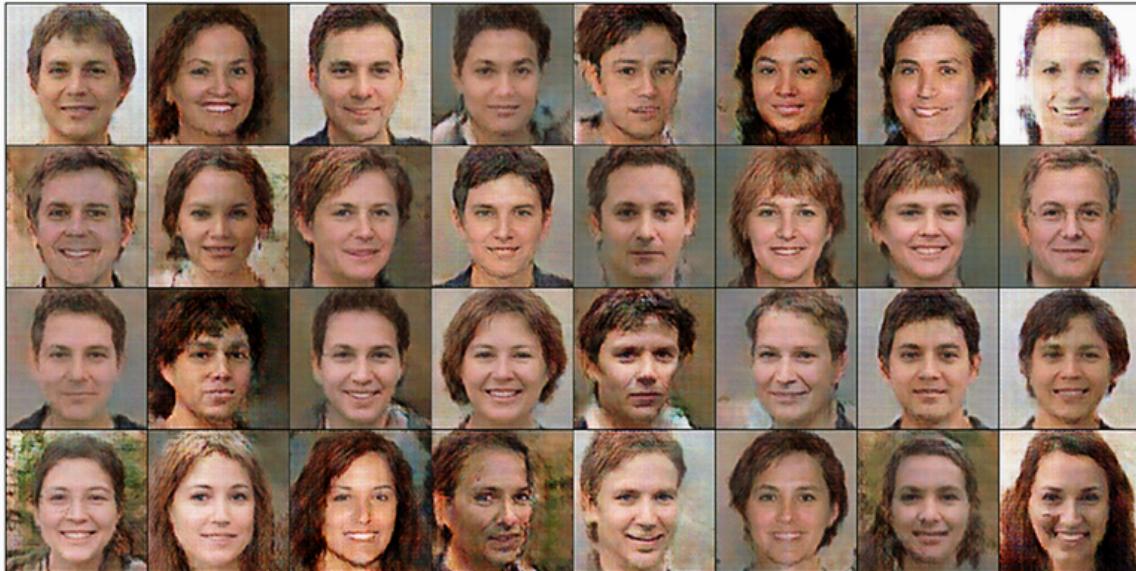


Figura 5.6 Imágenes de rostros humanos sin gafas generadas por el modelo cGAN entrenado

A continuación, usaremos la interpolación de etiquetas para realizar cálculos aritméticos. Recordemos que las dos etiquetas, `noise_g` y `noise_ng`, indican al modelo cGAN entrenado que cree imágenes con y sin gafas, respectivamente. ¿Qué sucede si introducimos una etiqueta interpolada (un promedio ponderado de las dos etiquetas [1, 0] y [0, 1]) en el modelo? ¿Qué tipo de imágenes producirá el generador entrenado? Averigüémoslo.

Listado 5.10 Aritmética de etiquetas en cGAN

```
weights=[0,0.25,0.5,0.75,1]①
plt.figure(figsize=(20,4),dpi=300)
for i in range(5):
    ax = plt.subplot(1, 5, i + 1)
    # change the value of z
    label=weights[i]*labels_ng[0]+(1-weights[i])*labels_g[0]②
    noise_and_labels=torch.cat(
        [z_female_g.reshape(1, z_dim, 1, 1),
         label.reshape(1, 2, 1, 1)],dim=1).to(device)
    fake=generator(noise_and_labels).cpu().detach()
    img=(fake[0]/2+0.5).permute(1,2,0)③
```

```

plt.imshow(img)
plt.xticks([])
plt.yticks([])
plt.subplots_adjust(wspace=-0.08, hspace=-0.01)
plt.show()

```

- ① Crea cinco pesos
- ② Crea un promedio ponderado de las dos etiquetas
- ③ Le da la nueva etiqueta al modelo entrenado para crear una imagen

Primero creamos cinco pesos (w): 0, 0,25, 0,5, 0,75 y 1, equidistantes entre 0 y 1. Cada uno de estos cinco valores de w corresponde al peso asignado a la etiqueta "sin gafas" `labels_ng`. El peso complementario se asigna a la etiqueta "gafas" `labels_g`. Por lo tanto, la etiqueta interpolada tiene un valor de $w*labels_ng+(1-w)*labels_g$. A continuación, introducimos la etiqueta interpolada en el modelo entrenado, junto con el vector de ruido aleatorio `z_female_g` que guardamos previamente. Las cinco imágenes generadas, basadas en los cinco valores de w , se representan en una cuadrícula de 1×5 , como se muestra en la figura 5.7.



Figura 5.7 Aritmética de etiquetas en cGAN. Primero creamos dos etiquetas: la etiqueta "sin gafas" `labels_ng` y la etiqueta "gafas" `labels_g`. Estas dos etiquetas indican al generador entrenado que produzca imágenes con y sin gafas, respectivamente. Luego creamos cinco etiquetas intercaladas, cada una como promedio ponderado de las dos etiquetas originales: $w*labels_ng+(1-w)*labels_g$, donde el peso w toma cinco valores diferentes: 0, 0,25, 0,5, 0,75 y 1. Las cinco imágenes generadas con base en las cinco etiquetas intercaladas se muestran en la figura. La imagen del extremo izquierdo tiene gafas. A medida que nos movemos de izquierda a derecha, las gafas se desvanece gradualmente, hasta que la imagen del extremo derecho no tiene gafas.

Al observar las cinco imágenes generadas en la figura 5.7, de izquierda a derecha, se observa que las gafas se desvanece gradualmente. La imagen de la izquierda tiene gafas, mientras que la de la derecha no. Las tres imágenes del centro muestran algunos indicios de gafas, pero estas no son tan visibles como las de la primera imagen.

Ejercicio 5.1

Dado que usamos el vector de ruido aleatorio `z_female_g` en el listado 5.10, las imágenes de la figura 5.7 tienen un rostro femenino. Cambie el vector de ruido a [`z_male_g` nombre del listado 5.10] y vuelva a ejecutar el programa; observe el aspecto de las imágenes.

5.5.2 Aritmética vectorial en el espacio latente

Quizás hayas notado que algunas imágenes generadas de rostros humanos tienen rasgos masculinos, mientras que otras tienen rasgos femeninos. Quizás te preguntes: ¿Podemos seleccionar rasgos masculinos o femeninos en las imágenes generadas? La respuesta es sí. Podemos lograrlo seleccionando vectores de ruido en el espacio latente.

En la última subsección, guardamos dos vectores de ruido aleatorio, `z_male_ng` y `z_female_ng`, que generan imágenes de un rostro masculino y uno femenino, respectivamente. A continuación, introducimos un promedio ponderado de los dos vectores (es decir, un vector interpolado) en el modelo entrenado y observamos el aspecto de las imágenes generadas.

Listado 5.11 Aritmética vectorial para seleccionar características de la imagen

```
weights=[0,0.25,0.5,0.75,1]                                     ①
plt.figure(figsize=(20,4),dpi=50)
for i in range(5):
    ax = plt.subplot(1, 5, i + 1)
    # change the value of z
    z=weights[i]*z_female_ng+(1-weights[i])*z_male_ng          ②
    noise_and_labels=torch.cat(
        [z.reshape(1, z_dim, 1, 1),
         labels_ng[0].reshape(1, 2, 1, 1)],dim=1).to(device)
    fake=generator(noise_and_labels).cpu().detach()               ③
    img=(fake[0]/2+0.5).permute(1,2,0)
    plt.imshow(img)
    plt.xticks([])
    plt.yticks([])
plt.subplots_adjust(wspace=-0.08,hspace=-0.01)
plt.show()
```

① Crea cinco pesos

② Crea un promedio ponderado de los dos vectores de ruido aleatorio

③ Alimenta el nuevo vector de ruido aleatorio al modelo entrenado para crear una imagen

Hemos creado cinco pesos: 0, 0,25, 0,5, 0,75 y 1. Iteramos sobre los cinco pesos y creamos cinco promedios ponderados de los dos vectores de ruido aleatorio, $w*z_{\text{female_ng}} + (1-w)*z_{\text{male_ng}}$. Luego, introducimos los cinco vectores, junto con la etiqueta, `labels_ng`, en el modelo entrenado para obtener cinco imágenes, como se muestra en la figura 5.8.



Figura 5.8 Aritmética vectorial en GAN. Primero, guardamos dos vectores de ruido aleatorio `z_female_ng` y `z_male_ng`. Estos dos vectores generan imágenes de rostros femeninos y masculinos, respectivamente. A continuación, creamos cinco vectores interpolados, cada uno como promedio ponderado de los dos vectores originales: $w*z_{\text{female_ng}} + (1-w)*z_{\text{male_ng}}$, donde el peso `w` toma cinco valores diferentes: 0, 0,25, 0,5, 0,75 y 1. Las cinco imágenes generadas a partir de los cinco vectores interpolados se muestran en la figura. La imagen del extremo izquierdo presenta rasgos masculinos. A medida que nos desplazamos de izquierda a derecha, los rasgos masculinos se desvanecen gradualmente y los femeninos aparecen gradualmente, hasta que la imagen del extremo derecho muestra un rostro femenino.

La aritmética vectorial permite la transición de una imagen a otra. Dado que hemos seleccionado una imagen masculina y otra femenina, al observar las cinco imágenes generadas en la figura 5.8 de izquierda a derecha, observará que los rasgos masculinos se difuminan gradualmente y los femeninos aparecen gradualmente. La primera imagen muestra un rostro masculino, mientras que la última muestra uno femenino.

Ejercicio 5.2

Dado que usamos la etiqueta `labels_ng` del listado 5.11, las imágenes de la figura 5.8 no tienen gafas. Cambie la etiqueta a `labels_g` la del listado 5.11 y vuelva a ejecutar el programa para ver cómo se ven las imágenes.

5.5.3 Selección de dos características simultáneamente

Hasta ahora, hemos seleccionado una característica a la vez. Al seleccionar la etiqueta, aprendiste a generar imágenes con o sin gafas. Al seleccionar un vector de ruido específico, aprendiste a seleccionar una instancia específica de la imagen generada.

¿Qué sucede si desea seleccionar dos características (por ejemplo, gafas y género) simultáneamente? Hay cuatro combinaciones posibles de las dos características independientes: rostros masculinos con gafas, rostros masculinos sin gafas, rostros femeninos con gafas y rostros femeninos sin gafas. A continuación, generaremos una imagen de cada tipo.

Listado 5.12 Selección de dos características simultáneamente

```
plt.figure(figsize=(20,5),dpi=50)
for i in range(4):
    ax = plt.subplot(1, 4, i + 1) ①
    p=i//2
    q=i%2
    z=z_female_g*p+z_male_g*(1-p) ②
    label=labels_ng[0]*q+labels_g[0]*(1-q) ③
    noise_and_labels=torch.cat(
        [z.reshape(1, z_dim, 1, 1),
         label.reshape(1, 2, 1, 1)],dim=1).to(device) ④
    fake=generator(noise_and_labels)
    img=(fake.cpu().detach()[0]/2+0.5).permute(1,2,0)
    plt.imshow(img.numpy())
    plt.xticks([])
    plt.yticks([])
    plt.subplots_adjust(wspace=-0.08,hspace=-0.01)
    plt.show()
```

① Itera de 0 a 3

② El valor de p, que puede ser 0 o 1, selecciona el vector de ruido aleatorio para generar una cara masculina o femenina.

③ El valor de q, que puede ser 0 o 1, selecciona la etiqueta para determinar si la imagen generada tiene anteojos o no.

- ④ Combina el vector de ruido aleatorio con la etiqueta para seleccionar dos características

Para generar cuatro imágenes que cubran los cuatro casos diferentes, necesitamos usar uno de los vectores de ruido como entrada: `z_female_g` o `z_male_g`. También necesitamos adjuntar a la entrada una etiqueta, que puede ser `labels_ng` o `labels_g`. Para usar un solo programa para cubrir los cuatro casos, iteramos a través de cuatro valores de i , 0 a 3, y creamos dos valores, p y q , que son el cociente entero y el resto del valor i dividido por 2. Por lo tanto, los valores de p y q pueden ser 0 o 1. Al establecer el valor del vector de ruido aleatorio en `z_female_g*p+z_male_g*(1-p)`, podemos seleccionar un vector de ruido aleatorio para generar una cara masculina o femenina. De manera similar, al establecer el valor de la etiqueta en `labels_ng[0]*q+labels_g[0]*(1-q)`, podemos seleccionar una etiqueta para determinar si la imagen generada tiene anteojos o no. Una vez que combinamos el vector de ruido aleatorio con la etiqueta y los alimentamos al modelo entrenado, podemos seleccionar dos características simultáneamente.

Si ejecuta el programa del listado 5.12, verá cuatro imágenes como se muestra en la figura 5.9.



Figura 5.9 Selección simultánea de dos características en la imagen generada. Seleccionamos un vector de ruido entre las dos opciones siguientes: `z_female_ng` y `z_male_ng`. También seleccionamos una etiqueta entre las dos opciones siguientes: `labels_ng` y `labels_g`. A continuación, introducimos el vector de ruido y la etiqueta en el generador entrenado para crear una imagen. Con base en los valores del vector de ruido y la etiqueta, el modelo entrenado puede crear cuatro tipos de imágenes. De esta manera, seleccionamos dos características independientes en la imagen generada: un rostro masculino o femenino y si la imagen contiene gafas o no.

Las cuatro imágenes generadas en la figura 5.9 tienen dos características independientes: rostro masculino o femenino y si la imagen incluye gafas. La primera imagen muestra un rostro masculino con gafas; la segunda, un rostro masculino sin gafas. La tercera imagen muestra un rostro femenino con gafas, mientras que la última muestra un rostro femenino sin gafas.

Ejercicio 5.3

Usamos los dos vectores de ruido aleatorio `z_female_g` y `z_male_g` en el listado 5.12. Cambie los dos vectores de ruido aleatorio a `z_female_ng` y `z_male_ng` y vuelva a ejecutar el programa para ver el aspecto de las imágenes.

Finalmente, podemos realizar aritmética de etiquetas y aritmética de vectores simultáneamente. Es decir, podemos introducir un vector de ruido interpolado y una etiqueta interpolada en el modelo cGAN entrenado y observar el aspecto de la imagen generada. Para ello, ejecute el siguiente bloque de código:

```
plt.figure(figsize=(20,20),dpi=50)
for i in range(36):
    ax = plt.subplot(6,6, i + 1)
    p=i//6
    q=i%6
    z=z_female_ng*p/5+z_male_ng*(1-p/5)
    label=labels_ng[0]*q/5+labels_g[0]*(1-q/5)
    noise_and_labels=torch.cat(
        [z.reshape(1, z_dim, 1, 1),
         label.reshape(1, 2, 1, 1)],dim=1).to(device)
    fake=generator(noise_and_labels)
    img=(fake.cpu().detach()[0]/2+0.5).permute(1,2,0)
    plt.imshow(img.numpy())
    plt.xticks([])
    plt.yticks([])
plt.subplots_adjust(wspace=-0.08,hspace=-0.01)
plt.show()
```

El código es similar al del listado 5.12, excepto que p y q pueden tomar seis valores diferentes: 0, 1, 2, 3, 4 y 5. El vector de ruido aleatorio,

`z_female_ng*p/5+z_male_ng*(1-p/5)`, toma seis valores diferentes según el valor de p . La etiqueta, `labels_ng[0]*q/5+labels_g[0]*(1-q/5)`, toma seis valores diferentes según el valor de q . Por lo tanto, tenemos 36 combinaciones diferentes de imágenes basadas en el vector de ruido interpolado y la etiqueta interpolada. Si ejecuta el programa anterior, verá 36 imágenes, como se muestra en la figura 5.10.

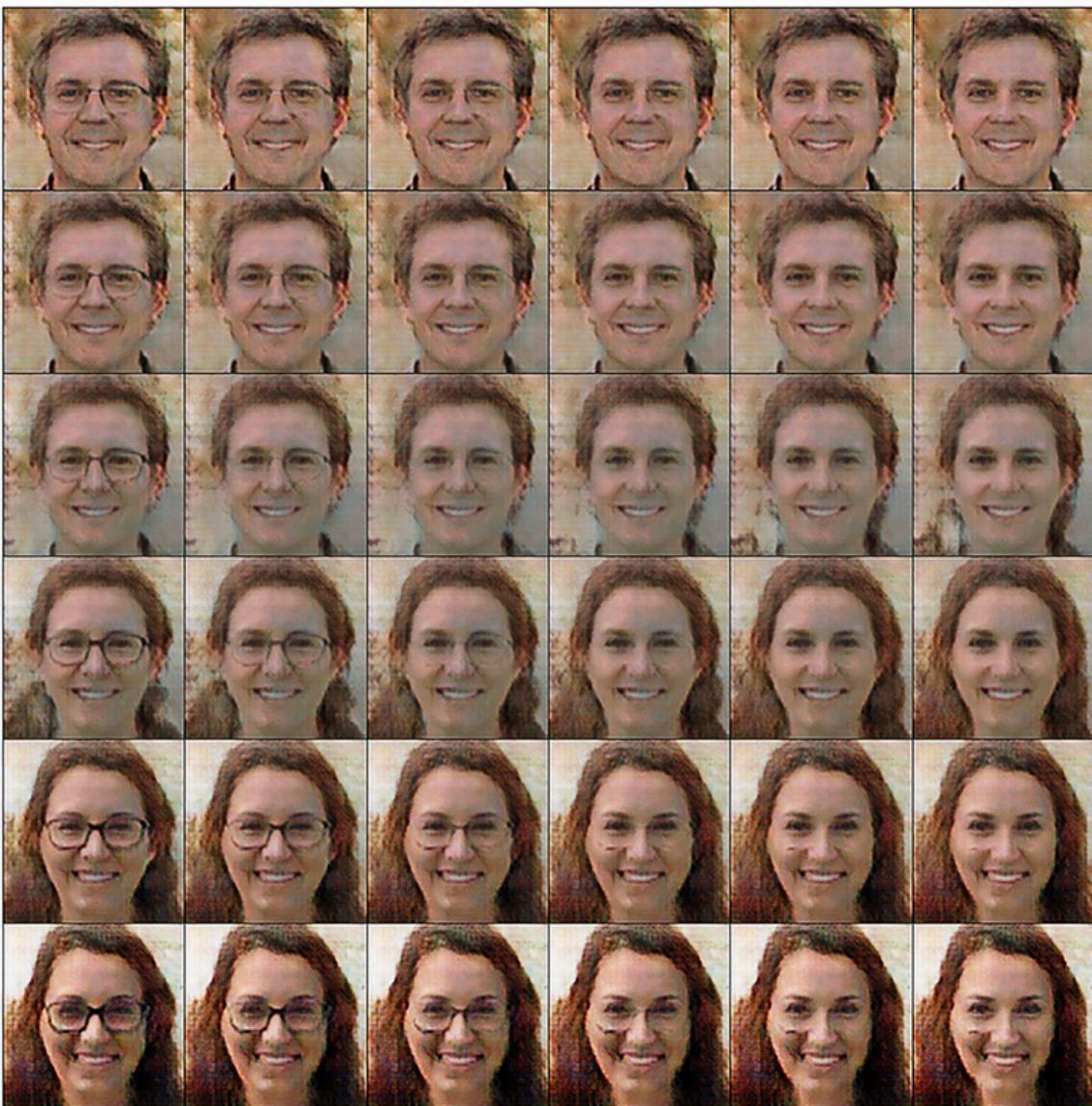


Figura 5.10 Realización simultánea de aritmética vectorial y de etiquetas. El valor de i cambia de 0 a 35; p y q son el cociente y el residuo enteros, respectivamente, al dividir i entre 6. Por lo tanto, p y q pueden tomar seis valores diferentes: 0, 1, 2, 3, 4 y 5. El vector de ruido interpolado.

`z_female_ng*p/5+z_male_ng*(1-p/5)`, y la etiqueta interpolada, `labels_ng[0]*q/5+labels_g[0]*(1-q/5)`, pueden tomar seis valores diferentes. En cada fila, de izquierda a derecha, las gafas se desvanecen gradualmente. En cada columna, de arriba a abajo, la imagen cambia gradualmente de un rostro masculino a uno femenino.

Hay 36 imágenes en la figura 5.10. El vector de ruido interpolado es un promedio ponderado de los dos vectores de ruido aleatorio, `z_female_ng` y `z_male_ng`, que generan un rostro femenino y uno masculino, respectivamente. La etiqueta es un promedio ponderado de las dos etiquetas, `labels_ng` y `labels_g`, que determinan si la imagen generada tiene gafas o no. El modelo entrenado genera 36 imágenes diferentes basándose en el vector de ruido interpolado y la etiqueta interpolada. En cada fila, cuando se va de izquierda a derecha, las gafas se desvanecen gradualmente. Es decir, realizamos aritmética de etiquetas en cada fila. En cada columna, cuando se va de arriba a abajo, la imagen cambia gradualmente de un rostro masculino a uno femenino. Es decir, realizamos aritmética de vectores en cada columna.

Ejercicio 5.4

En este proyecto, hay dos valores en la etiqueta: uno indica gafas y otro indica que no las hay. Por lo tanto, podemos usar un valor binario en lugar de variables one-hot como etiquetas. Modifique los programas de este capítulo y use los valores 1 y 0 (en lugar de [1, 0] y [0, 1]) para representar imágenes con y sin gafas. Añada 1 o 0 al vector de ruido aleatorio para alimentar un vector de 101 valores al generador. Añada un canal a la imagen de entrada antes de alimentarla al crítico: si una imagen tiene gafas, el cuarto canal se llena con 0; si la imagen no tiene gafas, el cuarto canal se llena con 1. Luego, cree un generador y un crítico; use el conjunto de datos de entrenamiento para entrenarlos. La solución se proporciona en el repositorio de GitHub del libro, junto con las soluciones a los otros tres ejercicios de este capítulo.

Ahora que has visto las capacidades de los modelos GAN, profundizarás en el siguiente capítulo realizando transferencias de estilo con GAN. Por ejemplo, aprenderás a construir un modelo CycleGAN y a entrenarlo con imágenes de rostros de famosos para convertir cabello rubio en negro o cabello negro en rubio en estas imágenes. Este mismo modelo se puede entrenar con otros conjuntos de datos: por ejemplo, puedes entrenarlo con el conjunto de datos de rostros humanos que usaste en este capítulo para añadir o quitar gafas en imágenes de rostros humanos.

Resumen

- Al seleccionar un determinado vector de ruido en el espacio latente y alimentarlo al modelo GAN entrenado, podemos seleccionar una determinada característica en la imagen generada, como por ejemplo si la imagen tiene un rostro masculino o femenino.
- Una cGAN es diferente de una GAN tradicional. Entrenamos el modelo con datos etiquetados y le pedimos que genere datos con un atributo específico. Por ejemplo, una etiqueta indica al modelo que genere imágenes de rostros humanos con gafas, mientras que otra indica que cree rostros humanos sin gafas.

- Tras el entrenamiento de una cGAN, podemos usar una serie de promedios ponderados de las etiquetas para generar imágenes que transitan de una imagen representada por una etiqueta a otra representada por otra; por ejemplo, una serie de imágenes en las que las gafas se desvanecen gradualmente en el rostro de la misma persona. A esto lo llamamos aritmética de etiquetas.
- También podemos usar una serie de promedios ponderados de dos vectores de ruido diferentes para crear imágenes que cambian de un atributo a otro; por ejemplo, una serie de imágenes en la que los rasgos masculinos se desvanecen gradualmente y los femeninos aparecen gradualmente. A esto lo llamamos aritmética vectorial.
- La GAN de Wasserstein (WGAN) es una técnica que mejora la estabilidad y el rendimiento del entrenamiento de los modelos GAN mediante el uso de la distancia de Wasserstein en lugar de la entropía cruzada binaria como función de pérdida. Además, para que la distancia de Wasserstein funcione correctamente, el crítico en las WGAN debe ser continuo 1-Lipschitz, lo que significa que las normas de gradiente de la función del crítico deben ser como máximo 1 en todas partes. La penalización de gradiente en las WGAN añade un término de regularización a la función de pérdida para aplicar la restricción de Lipschitz de forma más eficaz.

¹ Mehdi Mirza, Simon Osindero, 2014, “Redes generativas adversarias condicionales”. <https://arxiv.org/abs/1411.1784>.

² Martin Arjovsky, Soumith Chintala y Léon Bottou, 2017, “Wasserstein GAN”. <https://arxiv.org/abs/1701.07875>.

³ Martin Arjovsky, Soumith Chintala y Leon Bottou, 2017, “Wasserstein GAN”. <https://arxiv.org/abs/1701.07875>; e Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin y Aaron Courville, 2017, “Entrenamiento mejorado de las GAN de Wasserstein”.

<https://arxiv.org/abs/1704.00028>.

6. CycleGAN: Convirtiendo el cabello rubio en cabello negro

Este capítulo cubre

- La idea detrás de CycleGAN y la pérdida de consistencia del ciclo
- Construcción de un modelo CycleGAN para traducir imágenes de un dominio a otro
- Entrenamiento de un CycleGAN utilizando cualquier conjunto de datos con dos dominios de imágenes
- Convertir cabello negro a rubio y viceversa

Todos los modelos de redes generativas antagónicas (GAN) que hemos analizado en los últimos tres capítulos intentan producir imágenes que sean indistinguibles de las del conjunto de entrenamiento.

Quizás te preguntes: ¿Podemos traducir imágenes de un dominio a otro, como transformar caballos en cebras, convertir pelo negro en rubio o pelo rubio en negro, añadir o quitar gafas en las imágenes, convertir fotografías en pinturas o convertir escenas invernales en escenas estivales? ¡Resulta que sí es posible, y adquirirás estas habilidades en este capítulo a través de CycleGAN!

CycleGAN se introdujo en 2017.¹ Su principal innovación reside en su capacidad de aprender a traducir entre dominios sin ejemplos emparejados. CycleGAN ofrece diversas aplicaciones interesantes y útiles, como la simulación del proceso de envejecimiento o rejuvenecimiento facial para facilitar la verificación de identidad digital o la visualización de prendas de vestir en diferentes colores o estampados sin necesidad de crear físicamente cada variante para agilizar el proceso de diseño.

CycleGAN utiliza una función de pérdida de consistencia cíclica para garantizar que la imagen original se pueda reconstruir a partir de la imagen transformada, lo que fomenta la conservación de las características clave. La idea detrás de la pérdida de consistencia cíclica es realmente ingeniosa y merece ser destacada aquí. El CycleGAN de este capítulo cuenta con dos generadores: el generador de cabello negro y el generador de cabello rubio, respectivamente. El generador de cabello negro toma una imagen con cabello rubio (en lugar de un

vector de ruido aleatorio como se ha visto anteriormente) y la convierte en una con cabello negro, mientras que el generador de cabello rubio toma una imagen con cabello negro y la convierte en una con cabello rubio.

Para entrenar el modelo, le daremos una imagen real con cabello negro al generador de cabello rubio para producir una imagen falsa con cabello rubio. Luego, le daremos la imagen falsa de cabello rubio al generador de cabello negro para convertirla nuevamente a una imagen con cabello negro. Si ambos generadores funcionan bien, hay poca diferencia entre la imagen original con cabello negro y la falsa después de una conversión de ida y vuelta. Para entrenar CycleGAN, ajustamos los parámetros del modelo para minimizar la suma de las pérdidas adversarias y las pérdidas de consistencia del ciclo. Al igual que en los capítulos 3 y 4, las pérdidas adversarias se utilizan para cuantificar qué tan bien el generador puede engañar al discriminador y qué tan bien el discriminador puede diferenciar entre muestras reales y falsas. La pérdida de consistencia del ciclo, un concepto único en CycleGANs, mide la diferencia entre la imagen original y la imagen falsa después de una conversión de ida y vuelta. La inclusión de la pérdida de consistencia del ciclo en la función de pérdida total es la innovación clave en CycleGANs.

Usaremos imágenes de cabello negro y rubio como ejemplos de dos dominios al entrenar CycleGAN. Sin embargo, el modelo puede aplicarse a dos dominios de imágenes cualesquiera. Para recalcar el mensaje, les pediré que entrenen el mismo modelo CycleGAN usando imágenes con y sin gafas que usaron en el capítulo 5. La solución se encuentra en el repositorio de GitHub del libro (<https://github.com/markliu/DGAI>), y verán que el modelo entrenado puede añadir o quitar gafas de imágenes de rostros humanos.

6.1 CycleGAN y pérdida de consistencia del ciclo

CycleGAN amplía la arquitectura básica de GAN para incluir dos generadores y dos discriminadores. Cada par generador-discriminador se encarga de aprender la correspondencia entre dos dominios distintos. Su objetivo es traducir imágenes de un dominio a otro (p. ej., caballos a cebras, escenas de verano a invierno, etc.) conservando las características clave de las imágenes originales. Utiliza una pérdida de consistencia de ciclo que garantiza la reconstrucción de la imagen original a partir de la imagen transformada, lo que fomenta la conservación de las características clave.

En esta sección, analizaremos primero la arquitectura de CycleGAN. Haremos hincapié en su innovación clave: la pérdida de consistencia del ciclo.

6.1.1 ¿Qué es CycleGAN?

CycleGAN consta de dos generadores y dos discriminadores. Los generadores traducen imágenes de un dominio a otro, mientras que los discriminadores determinan la autenticidad de las imágenes en sus respectivos dominios. Estas redes son capaces de transformar fotografías en obras de arte que imitan el estilo de pintores famosos o movimientos artísticos específicos, acortando así la distancia entre el arte y la tecnología. También se pueden utilizar en el ámbito sanitario para tareas como la conversión de imágenes de resonancia magnética a tomografías computarizadas o viceversa, lo que puede ser útil cuando un tipo de imagen no está disponible o es demasiado costoso.

Para nuestro proyecto de este capítulo, realizaremos la conversión entre imágenes con cabello negro y rubio. Por lo tanto, las usaremos como ejemplo al explicar el funcionamiento de CycleGAN. La Figura 6.1 muestra un diagrama de la arquitectura de CycleGAN.

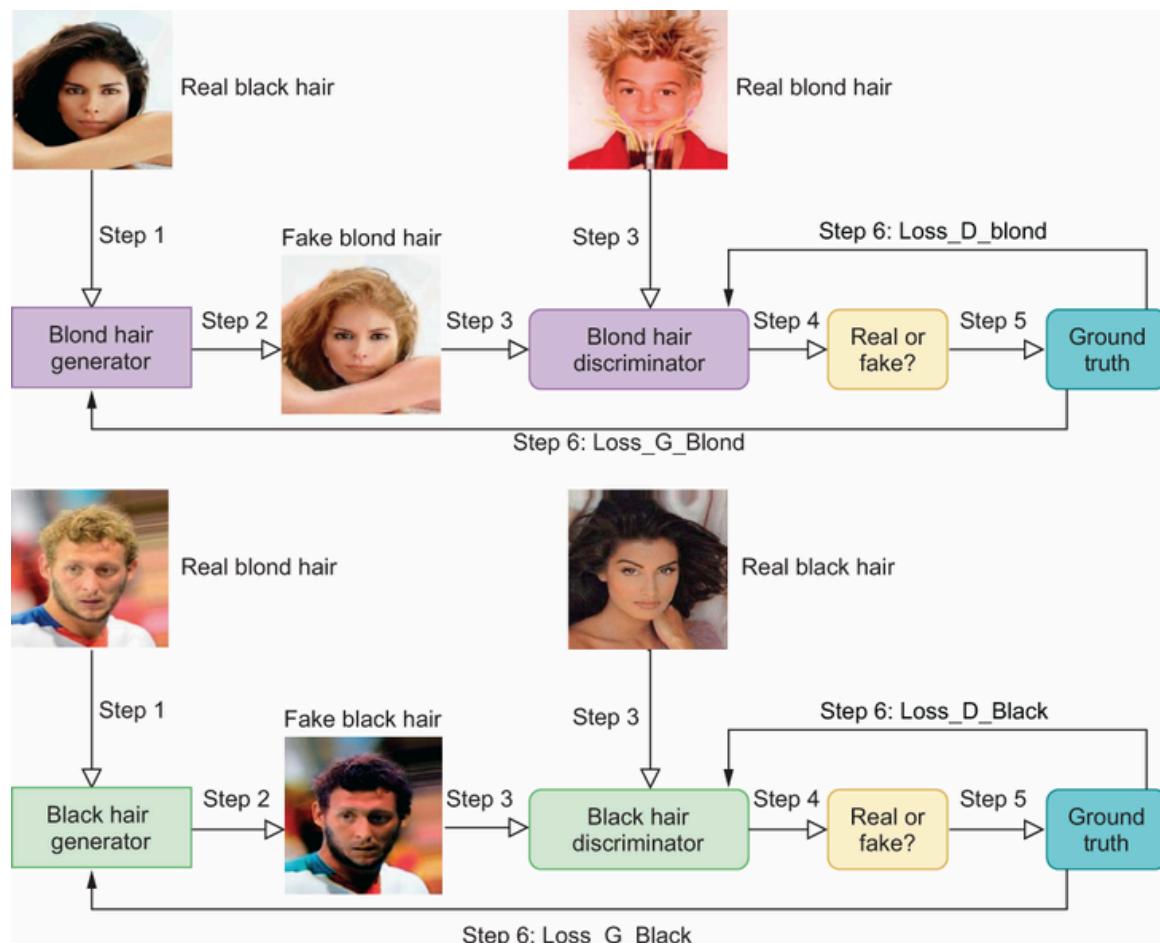


Figura 6.1 Arquitectura de un CycleGAN para convertir imágenes con cabello negro en rubio y para convertir imágenes con cabello rubio en negro. El diagrama también describe los pasos de entrenamiento para minimizar las pérdidas adversarias. La figura 6.2 explica cómo el modelo minimiza las pérdidas de consistencia del ciclo.

Para entrenar CycleGAN, utilizamos conjuntos de datos no apareados de los dos dominios entre los que deseamos realizar la traducción. Utilizaremos 48 472 imágenes de rostros de famosos con cabello negro y 29 980 imágenes con cabello rubio. Ajustamos los parámetros del modelo para minimizar la suma de las pérdidas adversarias y las pérdidas de consistencia del ciclo. Para facilitar la explicación, en la figura 6.1 solo explicaremos las pérdidas adversarias. En la siguiente subsección, explicaremos cómo el modelo minimiza las pérdidas de consistencia del ciclo.

En cada iteración del entrenamiento, alimentamos imágenes reales de cabello negro (arriba a la izquierda en la figura 6.1) al generador de cabello rubio para obtener imágenes falsas de cabello rubio. Posteriormente, alimentamos las imágenes falsas de cabello rubio, junto con las imágenes de cabello rubio real, al discriminador de cabello rubio (arriba en el centro). El discriminador de cabello rubio genera una probabilidad de que cada una sea una imagen de cabello rubio real. A continuación, comparamos las predicciones con la realidad (si una imagen es una imagen real con cabello rubio) y calculamos la pérdida para el discriminador (`Loss_D_Blond`), así como la pérdida para el generador (`Loss_G_Blond`).

Al mismo tiempo, en cada iteración del entrenamiento, alimentamos imágenes reales de cabello rubio (centro izquierda) al generador de cabello negro (abajo izquierda) para crear imágenes falsas de cabello negro. Presentamos las imágenes falsas de cabello negro, junto con las reales, al discriminador de cabello negro (centro abajo) para obtener predicciones de que son reales. Comparamos las predicciones del discriminador de cabello negro con la verdad de campo y calculamos la pérdida para el discriminador (`Loss_D_Black`) y la pérdida para el generador (`Loss_G_Black`). Entrenamos los generadores y los discriminadores simultáneamente. Para entrenar los dos discriminadores, ajustamos los parámetros del modelo para minimizar la pérdida del discriminador, que es la suma de `Loss_D_Black` y `Loss_D_Blond`.

6.1.2 Pérdida de consistencia del ciclo

Para entrenar los dos generadores, ajustamos los parámetros del modelo para minimizar la suma de la pérdida adversarial y la pérdida de consistencia del ciclo. La pérdida adversarial es la suma de `Loss_G_Black` y `Loss_G_Blond` como se explicó en la subsección anterior. Para explicar la pérdida de consistencia del ciclo, veamos la figura 6.2.

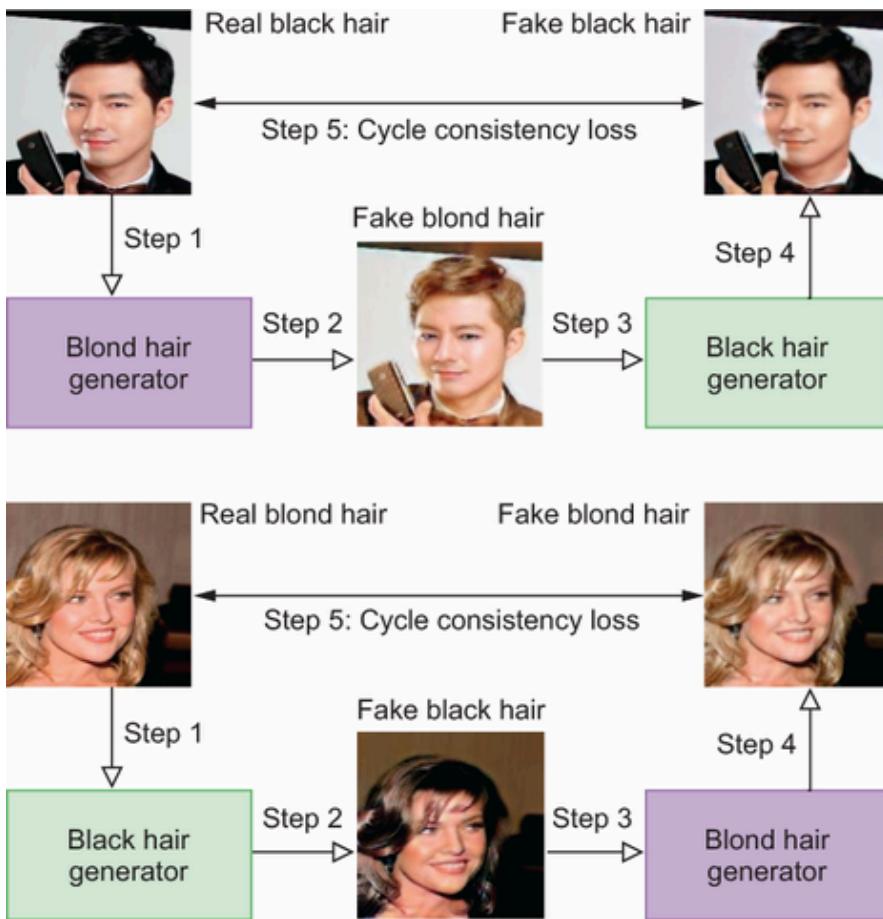


Figura 6.2 Cómo CycleGAN minimiza las pérdidas de consistencia del ciclo entre las imágenes originales de cabello negro y las falsas después de los viajes de ida y vuelta y las pérdidas de consistencia del ciclo entre las imágenes originales de cabello rubio y las falsas después de los viajes de ida y vuelta

La función de pérdida de los generadores en CycleGAN consta de dos partes. La primera, la pérdida adversarial, garantiza que las imágenes generadas sean indistinguibles de las imágenes reales en el dominio objetivo. Por ejemplo,

`Loss_G_Blond` (definido en la subsección anterior) garantiza que las imágenes rubias falsas generadas por el generador de cabello rubio se asemejen a las imágenes reales con cabello rubio en el conjunto de entrenamiento. La segunda parte, la pérdida de consistencia del ciclo, garantiza que una imagen traducida de un dominio a otro pueda traducirse de vuelta al dominio original.

La pérdida de consistencia del ciclo es un componente crucial de CycleGANs, ya que garantiza la recuperación de la imagen de entrada original tras una traducción de ida y vuelta. La idea es que, si traduces una imagen de cabello negro real (arriba a la izquierda en la figura 6.2) a una imagen de cabello rubio falso y la conviertes de nuevo a una imagen de cabello negro falso (arriba a la derecha), deberías obtener una imagen similar a la original. La pérdida de consistencia del ciclo para imágenes de cabello negro es el error absoluto medio, a nivel de píxel, entre la imagen falsa y la original. Llamemos a esta pérdida `Loss_Cycle_Black`. Lo mismo ocurre con la traducción de cabello rubio a cabello negro y luego de vuelta a cabello rubio, y la llamamos pérdida

`Loss_Cycle_Blond`. La pérdida total de consistencia del ciclo es la suma de `Loss_Cycle_Black` y `Loss_Cycle_Blon`.

6.2 El conjunto de datos de rostros de celebridades

Usaremos imágenes de rostros de famosos con cabello negro y rubio como dos dominios. Primero descargarás los datos en esta sección. Luego, procesarás las imágenes para prepararlas para el entrenamiento más adelante en este capítulo.

En este capítulo, usarás dos nuevas bibliotecas de Python: `pandas` y `albumentations`. Para instalarlas, ejecuta la siguiente línea de código en una nueva celda de tu aplicación Jupyter Notebook en tu computadora:

```
!pip install pandas albumentations
```

Siga las instrucciones en pantalla para finalizar la instalación.

6.2.1 Descarga del conjunto de datos de rostros de famosos

Para descargar el conjunto de datos de rostros de famosos, inicia sesión en Kaggle y accede al enlace <https://mng.bz/Ompo>. Descomprime el conjunto de datos después de descargarlo y guarda todos los archivos de imagen en la carpeta `/files/img_align_celeba/img_align_celeba/` de tu ordenador (ten en cuenta que hay una subcarpeta con el mismo nombre dentro de la carpeta). Esta carpeta contiene aproximadamente 200 000 imágenes. También descarga el archivo `list_attr_celeba.csv` de Kaggle y colócalo en la carpeta `/files/` de tu ordenador. El archivo CSV especifica varios atributos de cada imagen.

El conjunto de datos de rostros de famosos contiene imágenes con diferentes colores de cabello: castaño, gris, negro, rubio, etc. Seleccionaremos imágenes con cabello negro o rubio como conjunto de entrenamiento, ya que estos dos tipos son los más abundantes en el conjunto de datos. Ejecute el código de la siguiente lista para seleccionar todas las imágenes con cabello negro o rubio.

Listado 6.1 Selección de imágenes con cabello negro o rubio

```
import pandas as pd
import os, shutil

df=pd.read_csv("files/list_attr_celeba.csv")
```

(1)

```

os.makedirs("files/black", exist_ok=True)
os.makedirs("files/blond", exist_ok=True)          ②
folder="files/img_align_celeba/img_align_celeba"
for i in range(len(df)):
    dfi=df.iloc[i]
    if dfi['Black_Hair']==1:                      ③
        try:
            oldpath=f"{folder}/{dfi['image_id']}"
            newpath=f"files/black/{dfi['image_id']}"
            shutil.move(oldpath, newpath)
        except:
            pass
    elif dfi['Blond_Hair']==1:                     ④
        try:
            oldpath=f"{folder}/{dfi['image_id']}"
            newpath=f"files/blond/{dfi['image_id']}"
            shutil.move(oldpath, newpath)
        except:
            pass

```

① Carga el archivo CSV que contiene los atributos de la imagen

② Crea dos carpetas para almacenar imágenes con cabello negro y rubio

③ Si el atributo Black_Hair es 1, mueve la imagen a la carpeta negra.

④ Si el atributo Blond_Hair es 1, mueve la imagen a la carpeta rubia.

Primero usamos la biblioteca `pandas` para cargar el archivo

`list_attr_celeba.csv` y así saber si cada imagen tiene cabello negro o rubio. Luego, creamos dos carpetas locales, `/files/black/` y `/files/blond/`, para almacenar las imágenes con cabello negro y rubio, respectivamente. El Listado 6.1 itera a través de todas las imágenes del conjunto de datos. Si el atributo de una imagen `Black_Hair` es 1, la movemos a la carpeta `/files/black/`; si el atributo de una imagen `Blond_Hair` es 1, la movemos a la carpeta `/files/blond/`. Verá 48,472 imágenes con cabello negro y 29,980 imágenes con cabello rubio. La Figura 6.3 muestra algunos ejemplos de las imágenes.



Figura 6.3 Imágenes de muestra de rostros de celebridades con cabello negro o rubio

Las imágenes de la fila superior de la figura 6.3 muestran cabello negro, mientras que las de la fila inferior muestran cabello rubio. Además, la calidad de la imagen es alta: todos los rostros aparecen al frente y centrados, y los colores del cabello son fáciles de identificar. La cantidad y calidad de los datos de entrenamiento facilitarán el entrenamiento del modelo CycleGAN.

6.2.2 Procesar los datos de imágenes de cabello negro y rubio

Generalizaremos el modelo CycleGAN para que pueda entrenarse con cualquier conjunto de datos con dos dominios de imágenes. También definiremos una clase `LoadData()` para procesar el conjunto de datos de entrenamiento del modelo CycleGAN. La función se puede aplicar a cualquier conjunto de datos con dos dominios, ya sean imágenes de rostros humanos con diferentes colores de pelo, imágenes con o sin gafas, o imágenes con escenas de verano e invierno.

Para ello, hemos creado un módulo local `ch06util`. Descarga los archivos `ch06util.py` del repositorio `__init__.py` de GitHub del libro (<https://github.com/markhliu/DGAI>) y colócalos en la carpeta `/utils/` de tu ordenador. En el módulo local, hemos definido la siguiente clase `LoadData()`.

Listado 6.2 La clase `LoadData()` para procesar los datos de entrenamiento en CycleGAN

```
class LoadData(Dataset):
    def __init__(self, root_A, root_B, transform=None):      ①
        super().__init__()
        self.root_A = root_A
        self.root_B = root_B
        self.transform = transform
        self.A_images = []
        for r in root_A:
            files=os.listdir(r)
```

```

        self.A_images += [r+i for i in files]
    self.B_images = []
    for r in root_B:
        files=os.listdir(r)
        self.B_images += [r+i for i in files]
    self.len_data = max(len(self.A_images),
                        len(self.B_images))
    self.A_len = len(self.A_images)
    self.B_len = len(self.B_images)
def __len__(self):
    return self.len_data
def __getitem__(self, index):
    A_img = self.A_images[index % self.A_len]
    B_img = self.B_images[index % self.B_len]
    A_img = np.array(Image.open(A_img).convert("RGB"))
    B_img = np.array(Image.open(B_img).convert("RGB"))
    if self.transform:
        augmentations = self.transform(image=B_img,
                                         image0=A_img)
        B_img = augmentations["image"]
        A_img = augmentations["image0"]
    return A_img, B_img

```

① Las dos carpetas `root_A` y `root_B` son donde se almacenan las imágenes en los dos dominios

② Carga todas las imágenes en cada dominio

③ Define un método para contar la longitud del conjunto de datos

④ Define un método para acceder a elementos individuales en cada dominio

La clase `LoadData()` se hereda de la clase `Dataset` de PyTorch. Ambas listas `root_A` contienen carpetas `root_B` de imágenes en los dominios A y B, respectivamente. La clase carga imágenes en ambos dominios y genera un par de imágenes, una del dominio A y otra del dominio B, para que podamos usarlas para entrenar el modelo CycleGAN posteriormente.

Como hicimos en capítulos anteriores, creamos un iterador de datos con lotes para mejorar la eficiencia computacional, el uso de memoria y la dinámica de optimización en el proceso de entrenamiento.

Listado 6.3 Procesamiento de imágenes de cabello negro y rubio para entrenamiento

```

transforms = albumentations.Compose(
    [albumentations.Resize(width=256, height=256), ①
     albumentations.HorizontalFlip(p=0.5),
     albumentations.Normalize(mean=[0.5, 0.5, 0.5],
     std=[0.5, 0.5, 0.5], max_pixel_value=255), ②
     ToTensorV2()],
     additional_targets={"image0": "image"})
dataset = LoadData(root_A=["files/black/"],
    root_B=["files/blond/"],
    transform=transforms) ③
loader=DataLoader(dataset,batch_size=1,
    shuffle=True, pin_memory=True) ④

```

① Cambia el tamaño de las imágenes a 256 por 256 píxeles

② Normaliza las imágenes en el rango de -1 a 1

③ Aplica la clase LoadData() en las imágenes

④ Crea un iterador de datos para el entrenamiento

Primero definimos una instancia de la clase `Compose()` en la biblioteca `albumentations` (famosa por sus ampliaciones de imágenes rápidas y flexibles) y la llamamos `transforms`. La clase transforma las imágenes de varias maneras: redimensiona las imágenes a 256 x 256 píxeles y normaliza...Los valores están en el rango de -1 a 1. El argumento `HorizontalFlip()` del listado 6.3 crea una imagen reflejada de la imagen original en el conjunto de entrenamiento. La inversión horizontal es una técnica de aumento simple pero potente que mejora la diversidad de los datos de entrenamiento, lo que ayuda a los modelos a generalizar mejor y a ser más robustos. Los aumentos y el aumento de tamaño mejoran el rendimiento del modelo CycleGAN y hacen que las imágenes generadas sean realistas.

Luego aplicamos la clase `LoadData()` a las imágenes de cabello negro y rubio. Establecemos el tamaño del lote en 1, ya que las imágenes tienen un tamaño de archivo grande, y usamos un par de imágenes para entrenar el modelo en cada iteración. Si el tamaño del lote es mayor que 1, el equipo podría quedarse sin memoria.

6.3 Construcción de un modelo CycleGAN

En esta sección, construiremos un modelo CycleGAN desde cero. Nos aseguraremos de que nuestro modelo CycleGAN sea general para que pueda entrenarse con cualquier conjunto de datos con dos dominios de imágenes. Por lo tanto, usaremos A y B para representar ambos dominios en lugar de, por ejemplo, imágenes de cabello negro y rubio. Como ejercicio, entrenarás el mismo modelo CycleGAN con el conjunto de datos de gafas que utilizaste en el capítulo 5. Esto te ayudará a aplicar las habilidades aprendidas en este capítulo a otras aplicaciones del mundo real con un conjunto de datos diferente.

6.3.1 Creación de dos discriminadores

Aunque CycleGAN tiene dos discriminadores, son idénticos ex ante. Por lo tanto, crearemos una sola clase `Discriminator()` y la instanciaremos dos veces: una instancia será el discriminador A y la otra, el discriminador B. Los dos dominios en CycleGAN son simétricos, y no importa cuál denominemos dominio A: imágenes con cabello negro o imágenes con cabello rubio.

Abre el archivo `ch06util.py` que acabas de descargar. En él, he definido la clase `Discriminator()`.

Listado 6.4 Definición de la clase `Discriminator()` en CycleGAN

```
class Discriminator(nn.Module):
    def __init__(self, in_channels=3, features=[64,128,256,512]):
        super().__init__()
        self.initial = nn.Sequential(
            nn.Conv2d(in_channels,features[0],①
                      kernel_size=4,stride=2,padding=1,
                      padding_mode="reflect"),
            nn.LeakyReLU(0.2, inplace=True))
        layers = []
        in_channels = features[0]
        for feature in features[1:]:②
            layers.append(Block(in_channels, feature,
                                stride=1 if feature == features[-1] else 2))
            in_channels = feature
        layers.append(nn.Conv2d(in_channels,1,kernel_size=4,③
                              stride=1,padding=1,padding_mode="reflect"))
        self.model = nn.Sequential(*layers)
    def forward(self, x):
```

```
out = self.model(self.initial(x))  
return torch.sigmoid(out)
```

(4)

- ① La primera capa Conv2d tiene 3 canales de entrada y 64 canales de salida.
- ② Tres capas Conv2d más con 126, 256 y 512 canales de salida, respectivamente
- ③ La última capa Conv2d tiene 512 canales de entrada y 1 canal de salida.
- ④ Aplica la función de activación sigmoidea en la salida para que pueda interpretarse como una probabilidad

El código anterior define la red discriminadora. Su arquitectura es similar a la de la red discriminadora del capítulo 4 y a la de la red crítica del capítulo 5. Los componentes principales son cinco capas `Conv2d`. Aplicamos la función de activación sigmoidea en la última capa, ya que el discriminador realiza un problema de clasificación binaria. El discriminador toma como entrada una imagen a color de tres canales y genera un número entre 0 y 1, que puede interpretarse como la probabilidad de que la imagen de entrada sea una imagen real en el dominio.

El argumento `padding_mode="reflect"` que usamos en el listado 6.4 significa que el relleno añadido al tensor de entrada refleja el propio tensor de entrada. El relleno reflejado ayuda a preservar la información de los bordes al no introducir ceros artificiales en ellos. Crea transiciones más suaves en los límites del tensor de entrada, lo cual resulta beneficioso para diferenciar imágenes en diferentes dominios de nuestro entorno.

Luego creamos dos instancias de la clase y las llamamos `disc_A` y `disc_B`, respectivamente:

```
from utils.ch06util import Discriminator, weights_init      ①  
import torch  
  
device = "cuda" if torch.cuda.is_available() else "cpu"  
disc_A = Discriminator().to(device)  
disc_B = Discriminator().to(device)                          ②  
weights_init(disc_A)  
weights_init(disc_B)                                       ③
```

- ① Importa la clase Discriminador del módulo local

② Crea dos instancias de la clase Discriminador

③ Inicializa pesos

En el módulo local `ch06util`, también definimos una función `weights_init()` para inicializar los pesos del modelo. Esta función se define de forma similar a la del capítulo 5. A continuación, inicializamos los pesos en los dos discriminadores recién creados, `disc_A` y `disc_B`.

Ahora que tenemos dos discriminadores, a continuación crearemos dos generadores.

6.3.2 Creación de dos generadores

De manera similar, definimos una sola clase `Generator()` en el módulo local e instanciamos la clase dos veces: una instancia es el generador A y la otra es el generador B. En el archivo `ch06util.py` que acaba de descargar, hemos definido la clase `Generator()`.

Listado 6.5 La clase `Generator()` en CycleGAN

```
class Generator(nn.Module):
    def __init__(self, img_channels, num_features=64,
                 num_residuals=9):
        super().__init__()
        self.initial = nn.Sequential(
            nn.Conv2d(img_channels, num_features, kernel_size=7,
                     stride=1, padding=3, padding_mode="reflect", ),
            nn.InstanceNorm2d(num_features),
            nn.ReLU(inplace=True))
        self.down_blocks = nn.ModuleList(
            [ConvBlock(num_features, num_features*2, kernel_size=3,
                      stride=2, padding=1),
             ConvBlock(num_features*2, num_features*4, kernel_size=3,
                      stride=2, padding=1)])
        self.res_blocks = nn.Sequential(
            *[ResidualBlock(num_features * 4)
              for _ in range(num_residuals)])
        self.up_blocks = nn.ModuleList(
            [ConvBlock(num_features * 4, num_features * 2,
                      down=False, kernel_size=3, stride=2,
                      padding=1, output_padding=1),
             ConvBlock(num_features * 2, num_features * 1,
                      down=False, kernel_size=3, stride=2,
                      padding=1, output_padding=1)])
```

(1) (2) (3)

```

        padding=1, output_padding=1)])
self.last = nn.Conv2d(num_features * 1, img_channels,
                     kernel_size=7, stride=1,
                     padding=3, padding_mode="reflect")

def forward(self, x):
    x = self.initial(x)
    for layer in self.down_blocks:
        x = layer(x)
    x = self.res_blocks(x)
    for layer in self.up_blocks:
        x = layer(x)
    return torch.tanh(self.last(x))

```

(4)

① Tres capas Conv2d

② Nueve bloques residuales

③ Dos bloques de sobremuestreo

④ Aplica la activación tanh en la salida

La red generadora consta de varias capas Conv2d, seguidas de nueve bloques residuales (que explicaré en detalle más adelante). A continuación, la red tiene dos bloques de sobremuestreo: una capa ConvTranspose2d, una capa InstanceNorm2d y una activación ReLU. Como en capítulos anteriores, utilizamos la función de activación tanh en la capa de salida, de modo que todos los píxeles de salida se encuentran en el rango de -1 a 1, igual que las imágenes del conjunto de entrenamiento.

El bloque residual en el generador se define en el módulo local de la siguiente manera:

```

class ConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels,
                 down=True, use_act=True, **kwargs):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels,
                     padding_mode="reflect", **kwargs)
        if down
        else nn.ConvTranspose2d(in_channels,
                              out_channels, **kwargs),
            nn.InstanceNorm2d(out_channels),

```

```
nn.ReLU(inplace=True) if use_act else nn.Identity())
def forward(self, x):
    return self.conv(x)

class ResidualBlock(nn.Module):
    def __init__(self, channels):
        super().__init__()
        self.block = nn.Sequential(
            ConvBlock(channels, channels, kernel_size=3, padding=1),
            ConvBlock(channels, channels,
                      use_act=False, kernel_size=3, padding=1))
    def forward(self, x):
        return x + self.block(x)
```

Una conexión residual es un concepto del aprendizaje profundo, en particular en el diseño de redes neuronales profundas. Lo verás con frecuencia más adelante en este libro. Es una técnica utilizada para abordar el problema de los gradientes evanescentes, que suele ocurrir en redes muy profundas. En un bloque residual, que es la unidad básica de una red con conexiones residuales, la entrada se somete a una serie de transformaciones (como convolución, activación y normalización por lotes o instancias) y luego se vuelve a sumar a la salida de estas transformaciones. La Figura 6.4 muestra un diagrama de la arquitectura del bloque residual definido previamente.

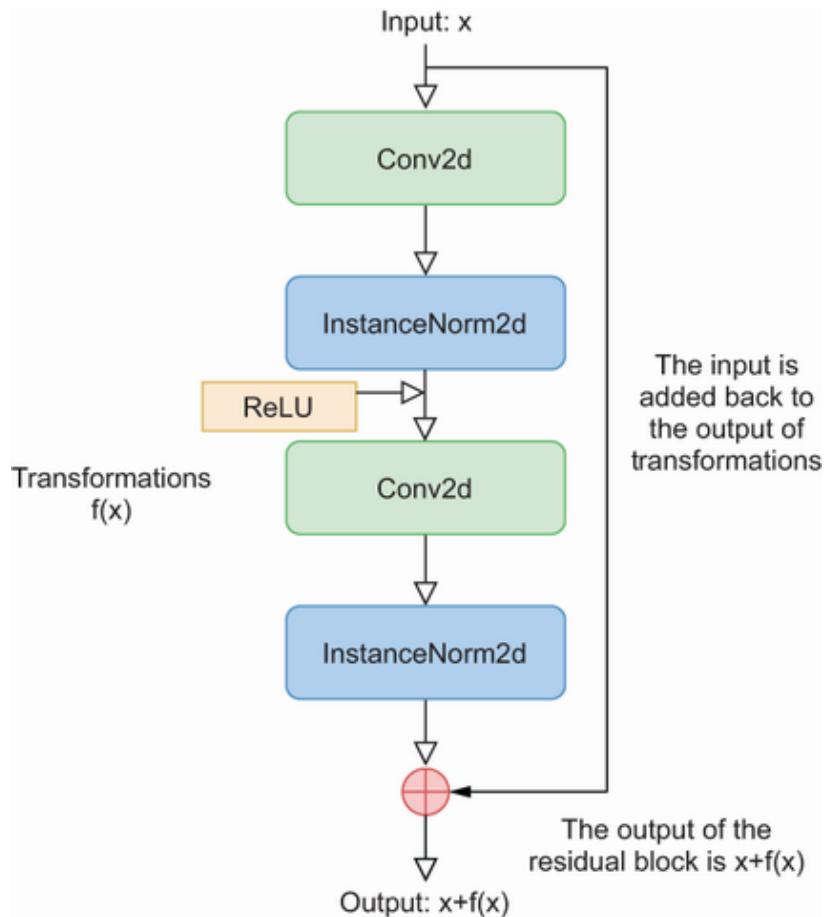


Figura 6.4 Arquitectura de un bloque residual. La entrada x se somete a una serie de transformaciones (dos conjuntos de capas Conv2d e InstanceNorm2d, y una activación ReLU). La entrada x se suma a la salida de estas transformaciones, $f(x)$. Por lo tanto, la salida del bloque residual es $x + f(x)$.

Las transformaciones en cada bloque residual son diferentes. En este ejemplo, la entrada x pasa por dos conjuntos de capas Conv2d y InstanceNorm2d y una activación ReLU entre ambos. La entrada x se suma a la salida de estas transformaciones, $f(x)$, para formar la salida final, $x + f(x)$ —de ahí el nombre de conexión residual.

A continuación, creamos dos instancias de la clase Generator() y llamamos a una de ellas gen_A y a la otra gen_B :

```
from utils.ch06util import Generator

gen_A = Generator(img_channels=3, num_residuals=9).to(device)
gen_B = Generator(img_channels=3, num_residuals=9).to(device)
weights_init(gen_A)
weights_init(gen_B)
```

Al entrenar el modelo, utilizaremos el error absoluto medio (es decir, la pérdida L1) para medir la pérdida de consistencia del ciclo. Utilizaremos el error cuadrático medio (es decir, la pérdida L2) para medir la pérdida adversarial. La pérdida L1 se utiliza a menudo si los datos son ruidosos y presentan muchos

valores atípicos, ya que penaliza los valores extremos menos que la pérdida L2. Por lo tanto, importamos las siguientes funciones de pérdida:

```
import torch.nn as nn

l1 = nn.L1Loss()
mse = nn.MSELoss()
g_scaler = torch.cuda.amp.GradScaler()
d_scaler = torch.cuda.amp.GradScaler()
```

Las pérdidas L1 y L2 se calculan a nivel de píxel. La imagen original tiene una forma de (3, 256, 256), al igual que la imagen falsa. Para calcular las pérdidas, primero calculamos la diferencia (valor absoluto de esta diferencia para la pérdida L1 y su valor al cuadrado para la pérdida L2) entre los valores de píxel correspondientes de dos imágenes en cada una de las $3 \times 256 \times 256 = 196608$ posiciones y calculamos el promedio de las posiciones.

Utilizaremos el paquete de precisión mixta automática de PyTorch `torch.cuda.amp` para acelerar el entrenamiento. El tipo de dato predeterminado en los tensores de PyTorch es `float32`, un número de punto flotante de 32 bits que ocupa el doble de memoria que un número flotante de 16 bits `float16`. Las operaciones con el primero son más lentas que con el segundo. Existe un equilibrio entre la precisión y el coste computacional. El tipo de dato a utilizar depende de la tarea en cuestión. `torch.cuda.amp` proporciona una precisión mixta automática, donde algunas operaciones usan `float32` y otras `float16`. La precisión mixta intenta asociar cada operación con su tipo de dato adecuado para acelerar el entrenamiento.

Como hicimos en el capítulo 4, utilizaremos el optimizador Adam tanto para los discriminadores como para los generadores:

```
lr = 0.00001
opt_disc = torch.optim.Adam(list(disc_A.parameters()) +
    list(disc_B.parameters()), lr=lr, betas=(0.5, 0.999))
opt_gen = torch.optim.Adam(list(gen_A.parameters()) +
    list(gen_B.parameters()), lr=lr, betas=(0.5, 0.999))
```

A continuación, entrenaremos el modelo CycleGAN utilizando imágenes con cabello negro o rubio.

6.4 Uso de CycleGAN para traducir entre cabello negro y rubio

Ahora que tenemos los datos de entrenamiento y el modelo CycleGAN, entramos el modelo usando imágenes con cabello negro o rubio. Como con todos los modelos GAN, descartaremos los discriminadores después del entrenamiento. Usaremos los dos generadores entrenados para convertir imágenes de cabello negro en rubio y para convertir imágenes de cabello rubio en negro.

6.4.1 Entrenando un CycleGAN para traducir entre cabello negro y rubio

Como explicamos en el capítulo 4, usaremos inspecciones visuales para determinar cuándo detener el entrenamiento. Para ello, creamos una función que prueba el aspecto de las imágenes reales y las generadas correspondientes, de modo que podamos compararlas e inspeccionar visualmente la eficacia del modelo. En el módulo local `ch06util`, definimos una función `test()`:

```
def test(i,A,B,fake_A,fake_B):
    save_image(A*0.5+0.5,f"files/A{i}.png")
    save_image(B*0.5+0.5,f"files/B{i}.png") ①
    save_image(fake_A*0.5+0.5,f"files/fakeA{i}.png")
    save_image(fake_B*0.5+0.5,f"files/fakeB{i}.png") ②
```

① Imágenes reales en los dominios A y B, guardadas en una carpeta local

② Las imágenes falsas correspondientes en los dominios A y B, creadas por los generadores en el lote i

Guardamos cuatro imágenes después de cada 100 lotes de entrenamiento. Guardamos las imágenes reales y las imágenes falsas correspondientes de ambos dominios en la carpeta local para poder revisar periódicamente las imágenes generadas y compararlas con las reales y evaluar el progreso del entrenamiento. Esta función es general para que pueda aplicarse a imágenes de dos dominios cualesquiera.

Además, definimos una función `train_epoch()` en el módulo local `ch06util` para entrenar los discriminadores y los generadores de una época. La siguiente lista muestra el código que usamos para entrenar ambos discriminadores.

Listado 6.6 Entrenamiento de los dos discriminadores en CycleGAN

```

def train_epoch(disc_A, disc_B, gen_A, gen_B, loader, opt_disc,
               opt_gen, l1, mse, d_scaler, g_scaler, device):
    loop = tqdm(loader, leave=True)
    for i, (A,B) in enumerate(loop):
        A=A.to(device)
        B=B.to(device)
        with torch.cuda.amp.autocast():
            fake_A = gen_A(B)
            D_A_real = disc_A(A)
            D_A_fake = disc_A(fake_A.detach())
            D_A_real_loss = mse(D_A_real,
                                torch.ones_like(D_A_real))
            D_A_fake_loss = mse(D_A_fake,
                                torch.zeros_like(D_A_fake))
            D_A_loss = D_A_real_loss + D_A_fake_loss
            fake_B = gen_B(A)
            D_B_real = disc_B(B)
            D_B_fake = disc_B(fake_B.detach())
            D_B_real_loss = mse(D_B_real,
                                torch.ones_like(D_B_real))
            D_B_fake_loss = mse(D_B_fake,
                                torch.zeros_like(D_B_fake))
            D_B_loss = D_B_real_loss + D_B_fake_loss
            D_loss = (D_A_loss + D_B_loss) / 2
            opt_disc.zero_grad()
            d_scaler.scale(D_loss).backward()
            d_scaler.step(opt_disc)
            d_scaler.update()
            ...

```

① Itera a través de todos los pares de imágenes en los dos dominios

② Utiliza el paquete de precisión mixta automática PyTorch para acelerar el entrenamiento

③ La pérdida total de los dos discriminadores es el promedio simple de las pérdidas adversarias de los dos discriminadores.

Utilizamos este método `detach()` para eliminar gradientes en tensores `fake_A`, `fake_B` reducir la memoria y acelerar los cálculos. El entrenamiento de los dos discriminadores es similar al del capítulo 4, con un par de diferencias. En primer lugar, en lugar de tener un solo discriminador, tenemos dos: uno para las imágenes del dominio A y otro para las del dominio B. La pérdida total de los dos discriminadores es el promedio simple de sus pérdidas adversaria-

rias. En segundo lugar, utilizamos el paquete automático de precisión mixta PyTorch para acelerar el entrenamiento, reduciendo el tiempo de entrenamiento en más del 50 %.

Entrenamos simultáneamente los dos generadores en la misma iteración. La siguiente lista destaca el código que usamos para entrenarlos.

Listado 6.7 Entrenamiento de los dos generadores en CycleGAN

```
def train_epoch(disc_A, disc_B, gen_A, gen_B, loader, opt_disc,
                opt_gen, l1, mse, d_scaler, g_scaler, device):
    ...
    with torch.cuda.amp.autocast():
        D_A_fake = disc_A(fake_A)
        D_B_fake = disc_B(fake_B)
        loss_G_A = mse(D_A_fake, torch.ones_like(D_A_fake)) ①
        loss_G_B = mse(D_B_fake, torch.ones_like(D_B_fake))
        cycle_B = gen_B(fake_A)
        cycle_A = gen_A(fake_B)
        cycle_B_loss = l1(B, cycle_B) ②
        cycle_A_loss = l1(A, cycle_A)
        G_loss=loss_G_A+loss_G_B+cycle_A_loss*10+cycle_B_loss*10 ③
        opt_gen.zero_grad()
        g_scaler.scale(G_loss).backward()
        g_scaler.step(opt_gen)
        g_scaler.update()
        if i % 100 == 0:
            test(i,A,B,fake_A,fake_B) ④
        loop.set_postfix(D_loss=D_loss.item(),G_loss=G_loss.item())
```

① Pérdidas adversas para los dos generadores

② Pérdidas de consistencia del ciclo para los dos generadores

③ La pérdida total de los dos generadores es la suma ponderada de las pérdidas adversarias y las pérdidas de consistencia del ciclo.

④ Genera imágenes para inspección visual después de cada 100 lotes de entrenamiento

El entrenamiento de los dos generadores difiere de lo realizado en el capítulo 4 en dos aspectos importantes. Primero, en lugar de tener un solo generador, entramos dos simultáneamente. Segundo, la pérdida total de los dos generado-

res es la suma ponderada de las pérdidas adversarias y las pérdidas de consistencia del ciclo, y ponderaremos estas últimas 10 veces más que la primera. Sin embargo, si se cambia el valor 10 por otros números como 9 o 12, se obtendrán resultados similares.

La pérdida de consistencia del ciclo es el error absoluto medio entre la imagen original y la imagen falsa que se traduce de nuevo al dominio original.

Ahora que tenemos todo listo, comenzaremos el bucle de entrenamiento:

```
from utils.ch06util import train_epoch

for epoch in range(1):
    train_epoch(disc_A, disc_B, gen_A, gen_B, loader, opt_disc,
                opt_gen, l1, mse, d_scaler, g_scaler, device)
    torch.save(gen_A.state_dict(), "files/gen_black.pth")  
①
    torch.save(gen_B.state_dict(), "files/gen_blond.pth")  
②
```

① Entrena el CycleGAN durante una época usando las imágenes de cabello negro y rubio

② Guarda los pesos del modelo entrenado

El entrenamiento anterior toma un par de horas si usas entrenamiento con GPU. De lo contrario, podría tomar un día entero. Si no tienes los recursos informáticos necesarios para entrenar el modelo, descarga los generadores preentrenados de mi sitio web:

<https://gattonweb.uky.edu/faculty/lium/ml/hair.zip> gen_black.pth . Descomprime el archivo y colóca gen_blond.pth en la carpeta /files/ de tu computadora. Podrás convertir imágenes de cabello negro a rubio en la siguiente subsección.

Ejercicio 6.1

Al entrenar el modelo CycleGAN, asumimos que el dominio A contiene imágenes con cabello negro y el dominio B, imágenes con cabello rubio. Modifique el código del listado 6.2 para que el dominio A contenga imágenes con cabello rubio y el dominio B, imágenes con cabello negro.

6.4.2 Conversiones de ida y vuelta de imágenes de cabello negro e imágenes de cabello rubio

Gracias a la alta calidad y la gran cantidad del conjunto de datos de entrenamiento, hemos entrenado CycleGAN con gran éxito. No solo convertiremos imágenes con cabello negro a rubio, sino que también realizaremos conversiones de ida y vuelta. Por ejemplo, convertiremos imágenes con cabello negro a rubio y luego las volveremos a convertir a negro. De esta forma, podemos comparar las imágenes originales con las generadas en el mismo dominio tras una conversión de ida y vuelta y observar la diferencia.

La siguiente lista realiza conversiones de imágenes entre los dos dominios, así como conversiones de ida y vuelta de imágenes en cada dominio.

Listado 6.8 Conversiones de ida y vuelta de imágenes con cabello negro o rubio

```
gen_A.load_state_dict(torch.load("files/gen_black.pth",
    map_location=device))
gen_B.load_state_dict(torch.load("files/gen_blond.pth",
    map_location=device))
i=1
for black,blond in loader:
    fake_blond=gen_B(black.to(device))
    save_image(black*0.5+0.5,f"files/black{i}.png")          ①
    save_image(fake_blond*0.5+0.5,f"files/fakeblond{i}.png")
    fake2black=gen_A(fake_blond)
    save_image(fake2black*0.5+0.5,
        f"files/fake2black{i}.png")                         ②
    fake_black=gen_A(blond.to(device))
    save_image(blond*0.5+0.5,f"files/blond{i}.png")          ③
    save_image(fake_black*0.5+0.5,f"files/fakeblack{i}.png")
    fake2blond=gen_B(fake_black)
    save_image(fake2blond*0.5+0.5,
        f"files/fake2blond{i}.png")                         ④
    i=i+1
    if i>10:
        break
```

① Imagen original con cabello negro

② Una imagen falsa con cabello negro después de un viaje de ida y vuelta.

③ Imagen original con cabello rubio

④ Una imagen falsa con cabello rubio después de un viaje de ida y vuelta.

Hemos guardado seis conjuntos de imágenes en su carpeta local /files/. El primer conjunto contiene las imágenes originales con cabello negro. El segundo conjunto contiene las imágenes rubias falsas generadas por el generador de cabello rubio entrenado: las imágenes se guardan como `fakeblond0.png`, `fakeblond1.png`, y así sucesivamente. El tercer conjunto contiene las imágenes falsas con cabello negro tras un proceso de ida y vuelta: alimentamos las imágenes falsas que acabamos de crear al generador de cabello negro entrenado para obtener imágenes falsas con cabello negro. Se guardan como `fake2black0.png`, `fake2black1.png`, y así sucesivamente. La Figura 6.5 muestra los tres conjuntos de imágenes.

Original images with black hair:



Fake images with blond hair:



Fake images translated to black hair:



Figura 6.5 Conversión de ida y vuelta de imágenes con cabello negro. Las imágenes de la fila superior son las imágenes originales con cabello negro del conjunto de entrenamiento. Las imágenes de la fila central son las imágenes falsas correspondientes con cabello rubio, generadas por el generador de cabello rubio entrenado. Las imágenes de la fila inferior son imágenes falsas con cabello negro tras una conversión de ida y vuelta: alimentamos las imágenes de la fila central al generador de cabello negro entrenado para crear imágenes falsas con cabello negro.

La figura 6.5 muestra tres filas de imágenes. La fila superior muestra imágenes originales con cabello negro del conjunto de entrenamiento. La fila central muestra imágenes de cabello rubio falso generadas a partir del cabello rubio entrenado. La fila inferior contiene imágenes de cabello negro falso tras una conversión de ida y vuelta: ¡las imágenes son casi idénticas a las de la fila superior! Nuestro modelo CycleGAN entrenado funciona de maravilla.

El cuarto conjunto de imágenes en la carpeta local /files/ son las imágenes originales con cabello rubio. El quinto conjunto es la imagen falsa generada por el generador de cabello negro entrenado. Finalmente, el sexto conjunto contiene imágenes falsas con cabello rubio tras un ciclo de ida y vuelta. La Figura 6.6 compara estos tres conjuntos de imágenes.

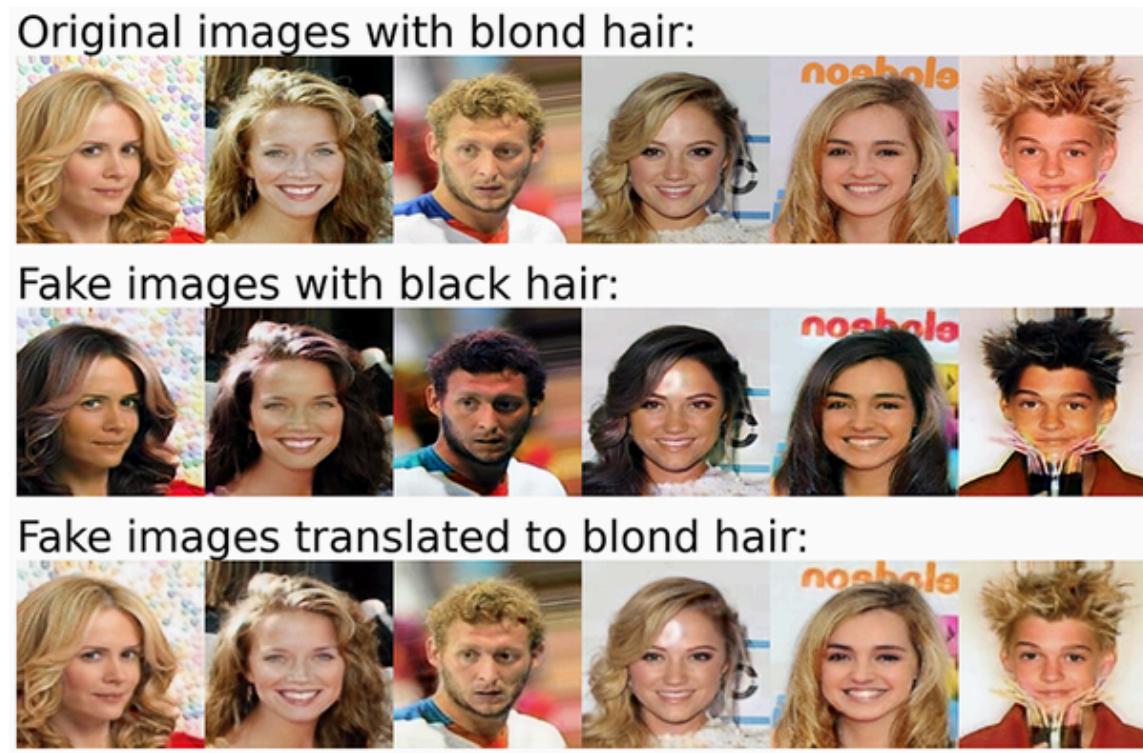


Figura 6.6 Conversión de ida y vuelta de imágenes con cabello rubio. Las imágenes de la fila superior son las imágenes originales con cabello rubio del conjunto de entrenamiento. Las imágenes de la fila central son las imágenes falsas correspondientes con cabello negro, generadas por el generador de cabello negro entrenado. Las imágenes de la fila inferior son imágenes falsas con cabello rubio tras una conversión de ida y vuelta: alimentamos las imágenes de la fila central al generador de cabello rubio entrenado para crear imágenes falsas con cabello rubio.

En la figura 6.6, las imágenes de cabello negro falso generadas por el generador de cabello negro entrenado se muestran en la fila central: presentan cabello negro en los mismos rostros humanos que la fila superior. Las imágenes de cabello rubio falso tras un ciclo de ida y vuelta se muestran en la fila inferior: son casi idénticas a las imágenes originales de cabello rubio de la fila superior.

Ejercicio 6.2

El modelo CycleGAN es general y se puede aplicar a cualquier conjunto de datos de entrenamiento con dos dominios de imágenes. Entrene el modelo CycleGAN con las imágenes de gafas que descargó en el capítulo 5. Use las imágenes con gafas como dominio A y las imágenes sin gafas como dominio B. Luego, use el CycleGAN entrenado para añadir y eliminar gafas de las imágenes (es

decir, traducir imágenes entre los dos dominios). Un ejemplo de implementación y los resultados se encuentran en el repositorio de GitHub del libro.

Hasta ahora, nos hemos centrado en un tipo de modelo generativo: las GAN. En el siguiente capítulo, aprenderá a usar otro tipo de modelo generativo, los auto-codificadores variacionales (VAE), para generar imágenes de alta resolución. Aprenderá las ventajas y desventajas de los VAE en comparación con las GAN. Y, aún más importante, aprenderá la arquitectura del codificador-decodificador en los VAE. Esta arquitectura se usa ampliamente en modelos generativos, incluyendo los Transformers, que estudiaremos más adelante en el libro.

Resumen

- CycleGAN puede traducir imágenes entre dos dominios sin ejemplos emparejados. Consta de dos discriminadores y dos generadores. Un generador convierte las imágenes del dominio A al dominio B, mientras que el otro las convierte del dominio B al dominio A. Los dos discriminadores clasifican si una imagen dada pertenece a un dominio específico.
- CycleGAN utiliza una función de pérdida de consistencia de ciclo para garantizar que la imagen original pueda reconstruirse a partir de la imagen transformada, lo que fomenta la preservación de las características clave.
- Un modelo CycleGAN correctamente construido puede aplicarse a cualquier conjunto de datos con imágenes de dos dominios. El mismo modelo puede entrenarse con diferentes conjuntos de datos y utilizarse para traducir imágenes en distintos dominios.
- Cuando disponemos de abundantes datos de entrenamiento de alta calidad, el CycleGAN entrenado puede convertir imágenes de un dominio a otro y luego devolverlas al dominio original. Tras una conversión de ida y vuelta, las imágenes pueden parecer casi idénticas a las originales.

¹ Jun-Yan Zhu, Taesung Park, Phillip Isola y Alexie Efros, 2017, “Traducción de imagen a imagen no apareada utilizando redes adversarias consistentes en ciclos”. <https://arxiv.org/abs/1703.10593>.

7. Generación de imágenes con autocodificadores variacionales

Este capítulo cubre

- Codificadores automáticos frente a codificadores automáticos variacionales
- Construcción y entrenamiento de un autocodificador para reconstruir dígitos escritos a mano
- Construcción y entrenamiento de un autocodificador variacional para generar imágenes de rostros humanos
- Realización de aritmética de codificación e interpolación con un autocodificador variacional entrenado

Hasta ahora, has aprendido a generar formas, números e imágenes mediante redes generativas antagónicas (GAN). En este capítulo, aprenderás a crear imágenes mediante otro modelo generativo: los autocodificadores variacionales (VAE). También aprenderás los usos prácticos de los VAE mediante la realización de operaciones aritméticas de codificación y la interpolación de codificación.

Para comprender el funcionamiento de los autocodificadores VAE, primero debemos comprender los autocodificadores (AE). Los AE tienen una estructura de dos componentes: un codificador y un decodificador. El codificador comprime los datos en una representación abstracta en un espacio de menor dimensión (el espacio latente), y el decodificador descomprime la información codificada y reconstruye los datos. El objetivo principal de un AE es aprender una representación comprimida de los datos de entrada, centrándose en minimizar la reconstrucción.error—la diferencia entre la entrada original y su reconstrucción (a nivel de píxel, como hemos visto en el capítulo 6 al calcular la pérdida de consistencia del ciclo). La arquitectura del codificador-decodificador es una piedra angular en varios modelos generativos, incluyendo Transformers, que explorarás en detalle en la segunda mitad de este libro. Por ejemplo, en el capítulo 9, construirás un Transformer para la traducción del lenguaje de máquina: el codificador convierte una frase en inglés en una representación abstracta mientras que el decodificador construye la traducción al francés basándose en la representación comprimida generada por el codificador. Los Transformers de texto a imagen como DALL-E 2 e Imagen también utilizan una arquitectura

AE en su diseño. Esto implica primero codificar una imagen en una distribución de probabilidad compacta y de baja dimensión. Luego, decodifican a partir de esta distribución. Por supuesto, lo que constituye un codificador y un decodificador es diferente en diferentes modelos.

Su primer proyecto en este capítulo implica construir y entrenar un AE desde cero para generar dígitos manuscritos. Utilizará 60.000 imágenes en escala de grises de dígitos manuscritos (del 0 al 9), cada una con un tamaño de $28 \times 28 = 784$ píxeles, como datos de entrenamiento. El codificador del AE comprime cada imagen en una representación vectorial determinista con solo 20 valores. El decodificador del AE reconstruye la imagen con el objetivo de minimizar la diferencia entre la imagen original y la reconstruida. Esto se logra minimizando el error absoluto medio entre las dos imágenes a nivel de píxel. El resultado final es un AE capaz de generar dígitos manuscritos casi idénticos a los del conjunto de entrenamiento.

Si bien los EA son eficaces para replicar los datos de entrada, a menudo fallan al generar nuevas muestras que no están presentes en el conjunto de entrenamiento. Más importante aún, los EA no son eficaces para la interpolación de entrada: a menudo no generan representaciones intermedias entre dos puntos de datos de entrada. Esto nos lleva a los VAE. Los VAE se diferencian de los EA en dos aspectos fundamentales. En primer lugar, mientras que un EA codifica cada entrada en un punto específico del espacio latente, un VAE la codifica en una distribución de probabilidad dentro de este espacio. En segundo lugar, un EA se centra únicamente en minimizar el error de reconstrucción, mientras que un VAE aprende los parámetros de la distribución de probabilidad de las variables latentes, minimizando una función de pérdida que incluye tanto la pérdida de reconstrucción como un término de regularización, el Kullback–Liebler (KL).

La divergencia KL facilita que el espacio latente se aproxime a una distribución determinada (una distribución normal en nuestro ejemplo) y garantiza que las variables latentes no solo memoricen los datos de entrenamiento, sino que capturen la distribución subyacente. Esto ayuda a lograr un espacio latente bien estructurado donde puntos de datos similares se mapean estrechamente, lo que hace que el espacio sea continuo e interpretable. Como resultado, podemos manipular las codificaciones para obtener nuevos resultados, lo que posibilita la aritmética de codificación y la interpolación de entrada en las VAE.

En el segundo proyecto de este capítulo, construirás y entrenarás un VAE desde cero para generar imágenes de rostros humanos. En este caso, tu conjunto de

entrenamiento incluye imágenes de gafas que descargaste en el capítulo 5. El codificador del VAE comprime una imagen de $3 \times 256 \times 256 = 196\,608$ píxeles en un vector probabilístico de 100 valores, cada uno con una distribución normal. El decodificador reconstruye la imagen basándose en este vector probabilístico. El VAE entrenado no solo puede replicar rostros humanos del conjunto de entrenamiento, sino también generar rostros nuevos.

Aprenderá cómo realizar aritmética de codificación e interpolación de entrada en VAE. Manipulará las representaciones codificadas (vectores latentes) de diferentes entradas para lograr resultados específicos (es decir, con o sin ciertas características en las imágenes) cuando se decodifiquen. Los vectores latentes controlan diferentes características en las imágenes decodificadas, como el género, si hay anteojos en una imagen, etc. Por ejemplo, primero puede obtener los vectores latentes para hombres con anteojos (z_1), mujeres con anteojos (z_2) y mujeres sin anteojos (z_3). Luego, calcula un nuevo vector latente, $z_4 = z_1 - z_2 + z_3$. Dado que tanto z_1 como z_2 conducen a anteojos en las imágenes cuando se decodifican, $z_1 - z_2$ cancela la característica de anteojos en la imagen resultante. De manera similar, dado que tanto z_2 como z_3 conducen a un rostro femenino, $z_3 - z_2$ cancela la característica femenina en la imagen resultante. Por lo tanto, si decodificas $z_4 = z_1 - z_2 + z_3$ con el VAE entrenado, obtendrás una imagen de un hombre sin gafas.

También crearás una serie de imágenes que cambian de una mujer con gafas a una mujer sin gafas variando el peso asignado a los vectores latentes z_1 y z_2 . Estos ejercicios ejemplifican la versatilidad y el potencial creativo de los VAE en el campo de los modelos generativos.

En comparación con las GAN, que estudiamos en los capítulos anteriores, las EA y las VAE tienen una arquitectura sencilla y son fáciles de construir. Además, suelen ser más fáciles y estables de entrenar que las GAN. Sin embargo, las imágenes generadas por EA y VAE tienden a ser más borrosas que las generadas por las GAN. Las GAN son excelentes para generar imágenes realistas de alta calidad, pero presentan dificultades de entrenamiento y un consumo intensivo de recursos. La elección entre GAN y VAE depende en gran medida de los requisitos específicos de la tarea en cuestión, incluyendo la calidad deseada del resultado, los recursos computacionales disponibles y la importancia de contar con un proceso de entrenamiento estable.

Las VAE tienen una amplia gama de aplicaciones prácticas en el mundo real. Imagine, por ejemplo, que dirige una tienda de gafas y ha comercializado con

éxito un nuevo estilo de gafas para hombre en línea. Ahora, desea dirigirse al mercado femenino con el mismo estilo, pero carece de imágenes de mujeres usándolas, y enfrenta altos costos para una sesión fotográfica profesional. Aquí es donde entran en juego las VAE: puede combinar imágenes existentes de hombres con las gafas con fotos de hombres y mujeres sin gafas. De esta manera, puede crear imágenes realistas de mujeres luciendo el mismo estilo de gafas, como se ilustra en la figura 7.1, mediante la codificación aritmética, una técnica que aprenderá en este capítulo.

woman without glasses - man without glasses + man with glasses = woman with glasses



Figura 7.1 Generación de imágenes de mujeres con gafas mediante la realización de operaciones aritméticas de codificación

En otro escenario, supongamos que su tienda ofrece gafas con monturas oscuras y claras, ambas muy populares. Quiere ofrecer una opción intermedia con monturas de un tono intermedio. Con las VAE, mediante un método llamado interpolación de codificación, puede generar fácilmente una serie de imágenes con una transición suave, como se muestra en la figura 7.2. Estas imágenes variarían de gafas con montura oscura a clara, ofreciendo a los clientes un espectro visual de opciones.



Figura 7.2 Generación de una serie de imágenes que pasan de gafas con marcos oscuros a gafas con marcos claros

El uso de VAE no se limita a las gafas; se extiende a prácticamente cualquier categoría de producto, ya sea ropa, muebles o alimentos. Esta tecnología ofrece una solución creativa y rentable para visualizar y comercializar una amplia gama de productos. Además, aunque la generación de imágenes es un ejemplo

destacado, los VAE pueden aplicarse a muchos otros tipos de datos, como música y texto. Su versatilidad abre un sinfín de posibilidades prácticas.

7.1 Una descripción general de los EA

Esta sección explica qué es un AE y su estructura básica. Para comprender a fondo su funcionamiento interno, en este capítulo, construirás y entregarás un AE para generar dígitos escritos a mano. Esta sección ofrece una descripción general de la arquitectura de un AE y un plan para completar el primer proyecto.

7.1.1 ¿Qué es un AE?

Los AE son un tipo de red neuronal utilizada en el aprendizaje no supervisado, especialmente eficaz para tareas como la generación, compresión y eliminación de ruido de imágenes. Un AE consta de dos partes principales: un codificador y un decodificador. El codificador comprime la entrada en una representación de menor dimensión (espacio latente) y el decodificador reconstruye la entrada a partir de esta representación.

La representación comprimida, o espacio latente, captura las características más importantes de los datos de entrada. En la generación de imágenes, este espacio codifica aspectos cruciales de las imágenes con las que se ha entrenado la red. Los AE son útiles por su eficiencia en el aprendizaje de representaciones de datos y su capacidad para trabajar con datos sin etiquetar, lo que los hace adecuados para tareas como la reducción de dimensionalidad y el aprendizaje de características. Un desafío con los AE es el riesgo de perder información en el proceso de codificación, lo que puede resultar en reconstrucciones menos precisas. El uso de arquitecturas más profundas con múltiples capas ocultas puede facilitar el aprendizaje de representaciones más complejas y abstractas, mitigando potencialmente la pérdida de información en los AE. Además, el entrenamiento de AE para generar imágenes de alta calidad puede ser computacionalmente intensivo y requiere grandes conjuntos de datos.

Como mencionamos en el capítulo 1, la mejor manera de aprender algo es crearlo desde cero. Para ello, en el primer proyecto de este capítulo aprenderás a crear un AE para generar dígitos escritos a mano. La siguiente subsección proporciona un plan para hacerlo.

7.1.2 Pasos para la construcción y entrenamiento de un AE

Imagina que debes construir y entrenar un AE desde cero para generar imágenes en escala de grises de dígitos escritos a mano, de modo que adquieras las habilidades necesarias para usar AE en tareas más complejas, como la generación de imágenes en color o la reducción de dimensionalidad. ¿Cómo deberías abordar esta tarea?

La Figura 7.3 proporciona un diagrama de la arquitectura de un AE y los pasos involucrados en el entrenamiento de un AE para generar dígitos escritos a mano.

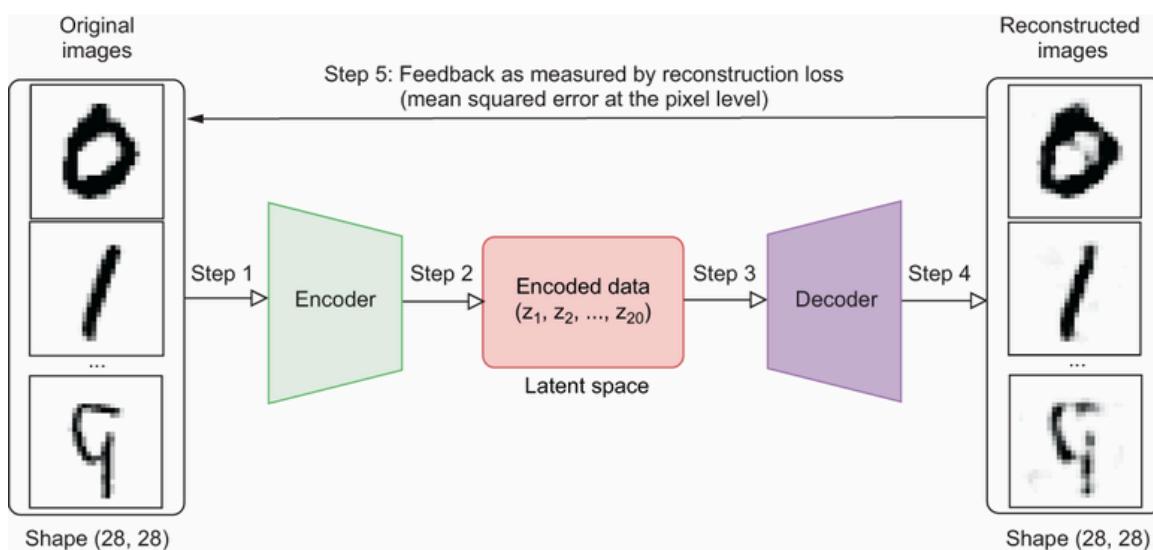


Figura 7.3 Arquitectura de un EA y los pasos para entrenarlo para generar dígitos manuscritos. Un EA consta de un codificador (centro izquierda) y un decodificador (centro derecha). En cada iteración de entrenamiento, se introducen imágenes de dígitos manuscritos en el codificador (paso 1). El codificador comprime las imágenes a puntos deterministas en el espacio latente (paso 2). El decodificador toma los vectores codificados (paso 3) del espacio latente y reconstruye las imágenes (paso 4). El EA ajusta sus parámetros para minimizar la pérdida de reconstrucción, la diferencia entre los originales y las reconstrucciones (paso 5).

Como se puede ver en la figura, el AE consta de dos partes principales: un codificador (centro izquierda) que comprime imágenes de dígitos manuscritos en vectores en el espacio latente y un decodificador (centro derecha) que reconstruye estas imágenes basándose en los vectores codificados. Tanto el codificador como el decodificador son redes neuronales profundas que pueden incluir diferentes tipos de capas, como capas densas, capas convolucionales, capas convolucionales transpuestas, etc. Dado que nuestro ejemplo incluye imágenes en escala de grises de dígitos manuscritos, solo utilizaremos capas densas. Sin embargo, los AE también pueden utilizarse para generar imágenes en color de mayor resolución; para estas tareas, las redes neuronales convolucionales (CNN) suelen incluirse en los codificadores y decodificadores. El uso de CNN en los AE depende de la resolución de las imágenes que se deseen generar.

Al construir un AE, sus parámetros se inicializan aleatoriamente. Necesitamos obtener un conjunto de entrenamiento para entrenar el modelo: PyTorch proporciona 60 000 imágenes en escala de grises de dígitos manuscritos, distribuidas uniformemente entre los 10 dígitos del 0 al 9. El lado izquierdo de la figura 7.3 muestra tres ejemplos, que corresponden a los dígitos 0, 1 y 9, respectivamente. En el primer paso del bucle de entrenamiento, alimentamos al codificador con las imágenes del conjunto de entrenamiento. El codificador comprime las imágenes en vectores de 20 valores en el espacio latente (paso 2). El número 20 no tiene nada de mágico. Si se utilizan vectores de 25 valores en el espacio latente, se obtendrán resultados similares. A continuación, alimentamos las representaciones vectoriales al decodificador (paso 3) y le solicitamos que reconstruya las imágenes (paso 4). Calculamos la pérdida de reconstrucción, que es el error cuadrático medio, sobre todos los píxeles, entre la imagen original y la imagen reconstruida. Luego, propagamos esta pérdida a través de la red para actualizar los parámetros del codificador y el decodificador y minimizar la pérdida de reconstrucción (paso 5). De esta manera, en la siguiente iteración, el EA puede reconstruir imágenes más cercanas a las originales. Este proceso se repite para varias épocas en el conjunto de datos.

Una vez entrenado el modelo, introducirá imágenes inéditas de dígitos manuscritos en el codificador y obtendrá las codificaciones. A continuación, introducirá las codificaciones en el decodificador para obtener imágenes reconstruidas. Observará que las imágenes reconstruidas son casi idénticas a las originales. El lado derecho de la figura 7.3 muestra tres ejemplos de imágenes reconstruidas: son similares a las originales correspondientes en el lado izquierdo de la figura.

7.2 Construcción y entrenamiento de un AE para generar dígitos

Ahora que tiene un plan para construir y entrenar un AE para generar dígitos escritos a mano, profundicemos en el proyecto e implementemos los pasos descritos en la última sección.

En esta sección, aprenderás primero a obtener un conjunto de entrenamiento y uno de prueba con imágenes de dígitos manuscritos. Después, construirás un codificador y un decodificador con capas densas. Entrenarás el AE con el conjunto de datos de entrenamiento y usarás el codificador entrenado para codificar imágenes en el conjunto de prueba. Finalmente, aprenderás a usar el deco-

dificador entrenado para reconstruir imágenes y compararlas con las originales.

7.2.1 Recopilación de dígitos escritos a mano

Puede descargar imágenes en escala de grises de imágenes escritas a mano utilizando el paquete *de conjuntos de datos* en la biblioteca Torchvision, de manera similar a cómo descargó imágenes de prendas de vestir en el capítulo 2.

Primero, descarguemos un conjunto de entrenamiento y un conjunto de prueba:

```
import torchvision
import torchvision.transforms as T

transform=T.Compose([
    T.ToTensor()])
train_set=torchvision.datasets.MNIST(root=".",
    train=True,download=True,transform=transform)      ①
test_set=torchvision.datasets.MNIST(root=".",
    train=False,download=True,transform=transform)     ②
                                                    ③
```

① Descarga dígitos escritos a mano mediante la clase `MNIST()` en `torchvision.datasets`

② El argumento `train=True` significa que descarga el conjunto de entrenamiento.

③ El argumento `train=False` significa que descarga el conjunto de prueba.

En lugar de usar la clase `FashionMNIST()` como hicimos en el capítulo 2, usamos `MNIST()` aquí. El argumento `train` de la clase indica a PyTorch si debe descargar el conjunto de entrenamiento (cuando el argumento se establece en `True`) o el conjunto de prueba (cuando el argumento se establece en `False`). Antes de la transformación, los píxeles de la imagen son números enteros de 0 a 255. La clase `ToTensor()` del bloque de código anterior los convierte en tensores flotantes de PyTorch con valores entre 0 y 1. Hay 60 000 imágenes en el conjunto de entrenamiento y 10 000 en el conjunto de prueba, distribuidas uniformemente entre 10 dígitos, del 0 al 9, en cada conjunto.

Crearemos lotes de datos para entrenamiento y pruebas, con 32 imágenes en cada lote:

```

import torch

batch_size=32
train_loader=torch.utils.data.DataLoader(
    train_set,batch_size=batch_size,shuffle=True)
test_loader=torch.utils.data.DataLoader(
    test_set,batch_size=batch_size,shuffle=True)

```

Ahora que tenemos los datos listos, construiremos y entrenaremos un AE.

7.2.2 Construcción y entrenamiento de un AE

Un AE consta de dos partes: el codificador y el decodificador. Definiremos una `AE()` clase, como se muestra en la siguiente lista, para representar el AE.

Listado 7.1 Creación de un AE para generar dígitos escritos a mano

```

import torch.nn.functional as F
from torch import nn

device="cuda" if torch.cuda.is_available() else "cpu"
input_dim = 784
z_dim = 20
h_dim = 200
class AE(nn.Module):
    def __init__(self,input_dim,z_dim,h_dim):
        super().__init__()
        self.common = nn.Linear(input_dim, h_dim)
        self.encoded = nn.Linear(h_dim, z_dim)
        self.l1 = nn.Linear(z_dim, h_dim)
        self.decode = nn.Linear(h_dim, input_dim)
    def encoder(self, x):
        common = F.relu(self.common(x))
        mu = self.encoded(common)
        return mu
    def decoder(self, z):
        out=F.relu(self.l1(z))
        out=torch.sigmoid(self.decode(out))
        return out
    def forward(self, x):
        mu=self.encoder(x)
        out=self.decoder(mu)
        return out, mu

```

- ① La entrada al AE tiene $28 \times 28 = 784$ valores.
- ② La variable latente (codificación) tiene 20 valores.
- ③ El codificador comprime las imágenes en variables latentes.

- ④ El decodificador reconstruye las imágenes basándose en codificaciones.

5 El codificador y el decodificador forman el AE.

El tamaño de entrada es 784, ya que las imágenes en escala de grises de los dígitos manuscritos tienen un tamaño de 28 x 28 píxeles. Aplanamos las imágenes a tensores unidimensionales y las introducimos en el AE. Las imágenes pasan primero por el codificador: se comprimen en codificaciones en un espacio de menor dimensión. Cada imagen se representa ahora mediante una variable latente de 20 valores. El decodificador reconstruye las imágenes basándose en las variables latentes. La salida del AE tiene dos tensores: `out`, las imágenes reconstruidas, y `mu`, las variables latentes (es decir, las codificaciones).

A continuación, instanciamos la clase `AE()` definida anteriormente para crear un AE. También usamos el optimizador Adam durante el entrenamiento, como hicimos en capítulos anteriores:

```
model = AE(input_dim,z_dim,h_dim).to(device)
lr=0.00025
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
```

Definimos una función `plot_digits()` para inspeccionar visualmente los dígitos escritos a mano reconstruidos después de cada época de entrenamiento, como se muestra en la siguiente lista.

Listado 7.2 La función `plot_digits()` para inspeccionar imágenes reconstruidas

```
import matplotlib.pyplot as plt

originals = []  
idx = 0  
for img,label in test_set:  
    if label == idx:  
        originals.append(img)  
    idx += 1
```

```

    if idx == 10:
        break
def plot_digits():
    reconstructed=[]
    for idx in range(10):
        with torch.no_grad():
            img = originals[idx].reshape((1,input_dim))
            out,mu = model(img.to(device))          (2)
            reconstructed.append(out)
    imgs=originals+reconstructed
    plt.figure(figsize=(10,2),dpi=50)
    for i in range(20):
        ax = plt.subplot(2,10, i + 1)
        img=(imgs[i]).detach().cpu().numpy()
        plt.imshow(img.reshape(28,28),           (4)
                   cmap="binary")
        plt.xticks([])
        plt.yticks([])
    plt.show()

```

- ① Recopila una imagen de muestra de cada dígito en el conjunto de prueba
- ② Alimenta la imagen al AE para obtener una imagen reconstruida
- ③ Recopila la imagen reconstruida de cada imagen original
- ④ Compara visualmente los dígitos originales con los reconstruidos

Primero recopilamos 10 imágenes de muestra, cada una representando un dígito diferente, y las colocamos en una lista `originals`. Alimentamos las imágenes al AE para obtener las imágenes reconstruidas. Finalmente, graficamos tanto las imágenes originales como las reconstruidas para compararlas y evaluar periódicamente el rendimiento del AE.

Antes de que comience el entrenamiento, llamamos a la función `plot_digits()` para visualizar la salida:

```
plot_digits()
```

Verá el resultado como se muestra en la figura 7.4.

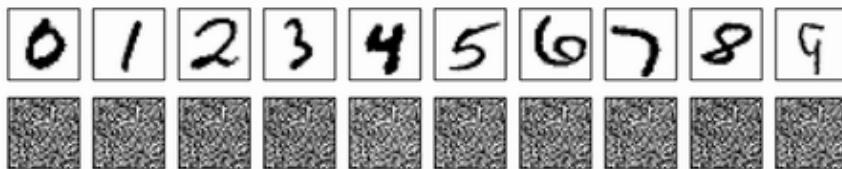


Figura 7.4 Comparación de las imágenes reconstruidas por el EA con las originales antes del inicio del entrenamiento. La fila superior muestra 10 imágenes originales de dígitos manuscritos del conjunto de prueba. La fila inferior muestra las imágenes reconstruidas por el EA antes del entrenamiento. Las reconstrucciones son puramente ruido.

Aunque podríamos dividir nuestros datos en conjuntos de entrenamiento y validación y entrenar el modelo hasta que no se observen más mejoras en el conjunto de validación (como hicimos en el capítulo 2), nuestro objetivo principal es comprender el funcionamiento de los EA, no necesariamente lograr el mejor ajuste de parámetros. Por lo tanto, entrenaremos el EA durante 10 épocas.

Listado 7.3 Entrenamiento del AE para generar dígitos escritos a mano

```

for epoch in range(10):
    tloss=0
    for imgs, labels in train_loader:          ①
        imgs=imgs.to(device).view(-1, input_dim)
        out, mu=model(imgs)                   ②
        loss=((out-imgs)**2).sum()            ③
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        tloss+=loss.item()
    print(f"at epoch {epoch} total loss = {tloss/len(train_loader)}") ④
    plot_digits()

```

① Itera a través de lotes en el conjunto de entrenamiento

② Utiliza el AE para reconstruir imágenes

③ Calcula la pérdida de reconstrucción medida por el error cuadrático medio

④ Inspecciona visualmente el rendimiento del AE

En cada época de entrenamiento, iteramos todos los lotes de datos del conjunto de entrenamiento. Alimentamos las imágenes originales al EA para obtener las imágenes reconstruidas. A continuación, calculamos la pérdida de reconstrucción, que es el error cuadrático medio entre las imágenes originales y las reconstruidas. Específicamente, la pérdida de reconstrucción se obtiene calcu-

lando primero la diferencia entre las dos imágenes, píxel a píxel, elevando al cuadrado los valores y promediando la diferencia al cuadrado. Ajustamos los parámetros del modelo para minimizar la pérdida de reconstrucción utilizando el optimizador Adam, una variante del método de descenso de gradiente.

El modelo tarda unos 2 minutos en entrenarse si se utiliza entrenamiento con GPU. También puedes descargar el modelo entrenado desde mi sitio web:
<https://mng.bz/YV6K>.

7.2.3 Guardar y usar el AE entrenado

Guardaremos el modelo en la carpeta local de tu computadora:

```
scripted = torch.jit.script(model)
scripted.save('files/AEdigits.pt')
```

Para usarlo para reconstruir una imagen de dígitos escritos a mano, cargamos el modelo:

```
model=torch.jit.load('files/AEdigits.pt',map_location=device)
model.eval()
```

Podemos usarlo para generar dígitos escritos a mano llamando a la función `plot_digits()` que definimos anteriormente:

```
plot_digits()
```

La salida se muestra en la figura 7.5.



Figura 7.5 Comparación de las imágenes reconstruidas por el EA entrenado con las originales. La fila superior muestra 10 imágenes originales de dígitos manuscritos del conjunto de prueba. La fila inferior muestra las imágenes reconstruidas por el EA entrenado. Las imágenes reconstruidas son similares a las originales.

Los dígitos manuscritos reconstruidos se parecen a los originales, aunque la reconstrucción no es perfecta. Parte de la información se pierde durante el pro-

ceso de codificación y decodificación. Sin embargo, en comparación con las GAN, los AE son fáciles de construir y requieren menos tiempo de entrenamiento. Además, muchos modelos generativos emplean la arquitectura codificador-decodificador. Este proyecto te ayudará a comprender los capítulos posteriores, especialmente cuando exploremos los Transformers.

7.3 ¿Qué son los VAE?

Si bien los EA son eficaces para reconstruir imágenes originales, no logran generar imágenes nuevas que no se observan en el conjunto de entrenamiento. Además, los EA no suelen mapear entradas similares a puntos cercanos en el espacio latente. Como resultado, el espacio latente asociado a un EA no es continuo ni fácilmente interpretable. Por ejemplo, no se pueden interpolar dos puntos de datos de entrada para generar representaciones intermedias significativas. Por estas razones, estudiaremos una mejora en los EA: los VAE.

En esta sección, aprenderá primero las diferencias clave entre los AE y los VAE y por qué estas diferencias permiten a estos últimos generar imágenes realistas invisibles en el conjunto de entrenamiento. A continuación, aprenderá los pasos para entrenar los VAE en general y, en particular, para entrenar uno para generar imágenes de rostros humanos de alta resolución.

7.3.1 Diferencias entre AE y VAE

Los VAE fueron propuestos por primera vez por Diederik Kingma y Max Welling en 2013.¹ Son una variante de los AE. Al igual que un AE, un VAE también consta de dos partes principales: un codificador y un decodificador.

Sin embargo, existen dos diferencias clave entre los AE y los VAE. En primer lugar, el espacio latente en un EA es determinista. Cada entrada se asigna a un punto fijo en el espacio latente. En cambio, el espacio latente en un VAE es probabilístico. En lugar de codificar una entrada como un único vector en el espacio latente, un VAE codifica una entrada como una distribución de posibles valores. En nuestro segundo proyecto, por ejemplo, codificaremos una imagen a color en un vector probabilístico de 100 valores. Además, asumiremos que cada elemento de este vector sigue una distribución normal independiente. Dado que definir una distribución normal solo requiere la media (μ) y la desviación estándar (σ), cada elemento de nuestro vector probabilístico de 100 elementos se caracterizará por estos dos parámetros. Para reconstruir la imagen, muestreamos un vector de esta distribución y lo decodificamos. La singulari-

dad de los VAE se destaca por el hecho de que cada muestreo de la distribución da como resultado una salida ligeramente variable.

En términos estadísticos, el codificador en un VAE está intentando aprender la verdadera distribución de los datos de entrenamiento $x, p(x|\Theta)$, donde Θ es el parámetro éteres que definen la distribución. Para mayor manejabilidad, generalmente asumimos que la distribución de la variable latente es normal. Dado que solo necesitamos la media, μ , y la desviación estándar, σ , para definir una distribución normal, podemos reescribir la distribución verdadera como $p(x|\Theta) = p(x|\mu, \sigma)$. El decodificador en el VAE genera una muestra basada en la distribución aprendida por el codificador. Es decir, el decodificador genera una instancia probabilísticamente a partir de la distribución $p(x|\mu, \sigma)$.

La segunda diferencia clave entre los AE y los VAE reside en la función de pérdida. Al entrenar un AE, minimizamos la pérdida de reconstrucción para que las imágenes reconstruidas sean lo más parecidas posible a las originales. En cambio, en los VAE, la función de pérdida consta de dos partes: la pérdida de reconstrucción y la divergencia KL. La divergencia KL mide cómo una distribución de probabilidad diverge de una segunda probabilidad esperada distribución de la incertidumbre. En las VAE, la divergencia KL se utiliza para regularizar el codificador penalizando las desviaciones de la distribución aprendida (la salida del codificador) con respecto a una distribución previa (una distribución normal estándar). Esto fomenta que el codificador aprenda representaciones latentes significativas y generalizables. Al penalizar las distribuciones demasiado alejadas de la previa, la divergencia KL ayuda a evitar el sobreajuste.

La divergencia KL se calcula de la siguiente manera en nuestro entorno, ya que asumimos una distribución normal (la fórmula es diferente si se asume una distribución no normal):

$$\text{KL Divergence} = \sum_{n=1}^{100} \left(\frac{\sigma_n^2}{2} + \frac{\mu_n^2}{2} - \log(\sigma_n^2) - \frac{1}{2} \right) \quad (7.1)$$

La suma se realiza sobre las 100 dimensiones del espacio latente. Cuando el codificador comprime las imágenes en distribuciones normales estándar en el espacio latente, de modo que $\mu = 0$ y $\sigma = 1$, la divergencia KL se convierte en 0. En cualquier otro caso, el valor es mayor que 0. Por lo tanto, la divergencia KL se

minimiza cuando el codificador comprime correctamente las imágenes en distribuciones normales estándar dentro del espacio latente.

7.3.2 El plan para entrenar un VAE para generar imágenes de rostros humanos

En el segundo proyecto de este capítulo, construirás y entrenarás un VAE desde cero para generar imágenes en color de rostros humanos. El modelo entrenado puede generar imágenes no vistas en el conjunto de entrenamiento. Además, puedes interpolar entradas para generar imágenes novedosas que sean representaciones intermedias entre dos puntos de datos de entrada. A continuación, se presenta un plan para este segundo proyecto.

La figura 7.6 proporciona un diagrama de la arquitectura de un VAE y los pasos para entrenar un VAE para generar imágenes de rostros humanos.

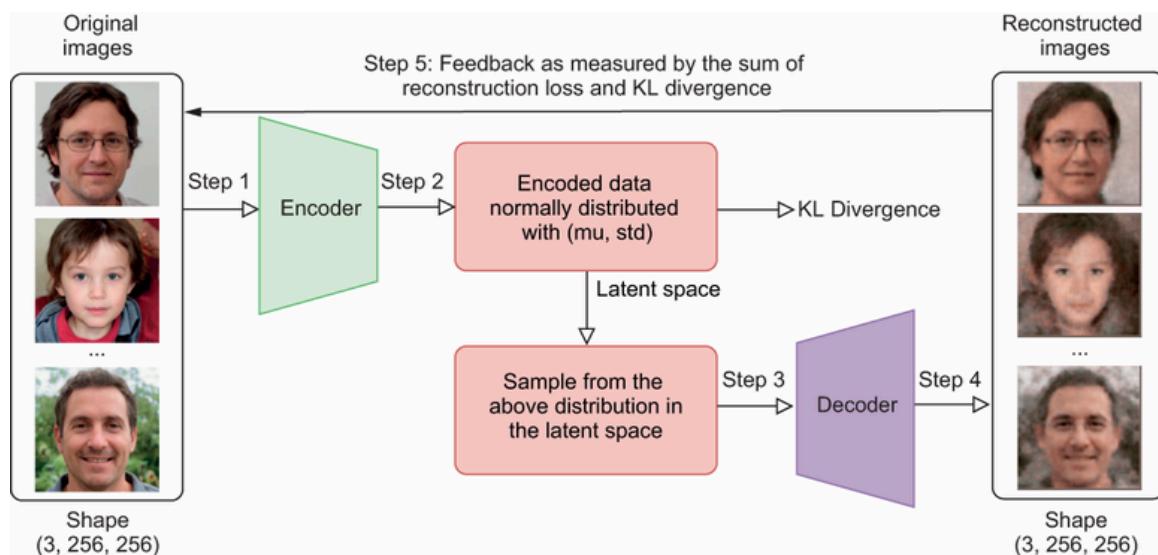


Figura 7.6 Arquitectura de un VAE y pasos para entrenarlo para generar imágenes de rostros humanos. Un VAE consta de un codificador (centro superior izquierdo) y un decodificador (centro inferior derecho). En cada iteración de entrenamiento, se alimentan imágenes de rostros humanos al codificador (paso 1). El codificador comprime las imágenes a puntos probabilísticos en el espacio latente (paso 2; dado que asumimos distribuciones normales, cada punto de probabilidad se caracteriza por un vector de medias y un vector de desviaciones estándar). Luego, muestreamos codificaciones de la distribución y las presentamos al decodificador. El decodificador toma las codificaciones muestreadas (paso 3) y reconstruye las imágenes (paso 4). El VAE ajusta sus parámetros para minimizar la suma de la pérdida de reconstrucción y la divergencia KL. La divergencia KL mide la diferencia entre la salida del codificador y una distribución normal estándar.

La Figura 7.6 muestra que una VAE también consta de dos partes: un codificador (centro superior izquierdo) y un decodificador (centro inferior derecho). Dado que el segundo proyecto implica imágenes en color de alta resolución, utilizaremos CNN para crear la VAE. Como se explicó en el capítulo 4, las imágenes en color de alta resolución contienen muchos más píxeles que las imágenes en escala de grises de baja resolución. Si solo utilizamos capas completamente conectadas (densas), el número de parámetros del modelo es demasiado

elevado, lo que hace que el aprendizaje sea lento e ineficaz. Las CNN requieren menos parámetros que las redes completamente conectadas de tamaño similar, lo que permite un aprendizaje más rápido y eficaz.

Una vez creado el VAE, usará el conjunto de datos de anteojos que descargó en el capítulo 5 para entrenar el modelo. El lado izquierdo de la figura 7.6 muestra tres ejemplos de las imágenes originales de rostros humanos en el conjunto de entrenamiento. En el primer paso del bucle de entrenamiento, alimentamos imágenes en el conjunto de entrenamiento, con un tamaño de $3 \times 256 \times 256 = 196,608$ píxeles , al codificador. El codificador comprime las imágenes a vectores probabilísticos de 100 valores en el espacio latente (paso 2; vectores de medias y desviaciones estándar debido al supuesto de distribución normal). Luego, tomamos muestras de la distribución y alimentamos las representaciones vectoriales muestreadas al decodificador (paso 3) y le pedimos que reconstruya las imágenes (paso 4). Calculamos la pérdida total como la suma de la pérdida de reconstrucción a nivel de píxel y la divergencia KL como se especifica en la ecuación 7.1. Propagamos esta pérdida a través de la red para actualizar los parámetros del codificador y el decodificador y minimizar la pérdida total (paso 5). Esta pérdida total incentiva al VAE a codificar las entradas en representaciones latentes más significativas y generalizables, y a reconstruir imágenes más cercanas a las originales.

Una vez entrenado el modelo, se introducen imágenes de rostros humanos en el codificador y se obtienen las codificaciones. A continuación, se introducen las codificaciones en el decodificador para obtener imágenes reconstruidas. Observará que las imágenes reconstruidas se parecen bastante a las originales. El lado derecho de la figura 7.6 muestra tres ejemplos de imágenes reconstruidas: se parecen a las originales correspondientes en el lado izquierdo de la figura, aunque no son perfectas.

Más importante aún, puede descartar el codificador y extraer codificaciones aleatorias del espacio latente para alimentarlas al decodificador entrenado en VAE para generar nuevas imágenes de rostros humanos que no se ven en el conjunto de entrenamiento. Además, puede manipular las representaciones codificadas de diferentes entradas para obtener resultados específicos al decodificarlas. También puede crear una serie de imágenes que transicionen de una instancia a otra variando el peso asignado a dos codificaciones cualesquiera.

7.4 Un VAE para generar imágenes de rostros humanos

Esta sección crea y entrena un VAE desde cero para generar imágenes de rostros humanos siguiendo los pasos descritos en la última sección.

En comparación con lo que hemos hecho para construir y entrenar AE, nuestro enfoque para el segundo proyecto incorpora varias modificaciones. En primer lugar, planeamos usar CNN tanto en los codificadores como en los decodificadores de VAE, particularmente porque las imágenes en color de alta resolución poseen un mayor número de píxeles. Confiar únicamente en capas completamente conectadas (densas) resultaría en un número excesivo de parámetros, lo que resultaría en un aprendizaje lento e ineficiente. En segundo lugar, como parte de nuestro proceso para comprimir imágenes en vectores que siguen una distribución normal en el espacio latente, generaremos un vector de media y un vector de desviación estándar durante la codificación de cada imagen. Esto difiere del vector de valor fijo utilizado en AE. A partir de la distribución normal codificada, muestrearemos para obtener codificaciones, que posteriormente se decodificarán para producir imágenes. Cabe destacar que cada imagen reconstruida variará ligeramente cada vez que muestreemos de esta distribución, lo que da lugar a la capacidad de los VAE para generar imágenes novedosas.

7.4.1 Construcción de un VAE

Si recuerdas, el conjunto de datos de gafas que descargaste en el capítulo 5 se guarda en la carpeta `/files/glasses/` de tu ordenador tras corregir manualmente algunas etiquetas. Ajustaremos el tamaño de las imágenes a 256 x 256 píxeles con valores entre 0 y 1. A continuación, crearemos un iterador por lotes con 16 imágenes en cada lote:

```
transform = T.Compose([
    T.Resize(256),                                     ①
    T.ToTensor(),                                      ②
])
data = torchvision.datasets.ImageFolder(
    root="files/glasses",
    transform=transform)                                ③
batch_size=16
loader = torch.utils.data.DataLoader(data,
    batch_size=batch_size, shuffle=True)                ④
```

① Cambia el tamaño de las imágenes a 256 por 256 píxeles

② Convierte imágenes en tensores con valores entre 0 y 1

③ Carga imágenes de la carpeta y aplica las transformaciones

④ Coloca los datos en un iterador por lotes

A continuación, crearemos un VAE que incluya capas convolucionales y transpuestas. Primero, definiremos una clase `Encoder()` como se indica a continuación.

Listado 7.4 El codificador en el VAE

```
latent_dims=100  
class Encoder(nn.Module):  
    def __init__(self, latent_dims=100):  
        super().__init__()  
        self.conv1 = nn.Conv2d(3, 8, 3, stride=2, padding=1)  
        self.conv2 = nn.Conv2d(8, 16, 3, stride=2, padding=1)  
        self.batch2 = nn.BatchNorm2d(16)  
        self.conv3 = nn.Conv2d(16, 32, 3, stride=2, padding=0)  
        self.linear1 = nn.Linear(31*31*32, 1024)  
        self.linear2 = nn.Linear(1024, latent_dims)  
        self.linear3 = nn.Linear(1024, latent_dims)  
        self.N = torch.distributions.Normal(0, 1)  
        self.N.loc = self.N.loc.cuda()  
        self.N.scale = self.N.scale.cuda()  
    def forward(self, x):  
        x = x.to(device)  
        x = F.relu(self.conv1(x))  
        x = F.relu(self.batch2(self.conv2(x)))  
        x = F.relu(self.conv3(x))  
        x = torch.flatten(x, start_dim=1)  
        x = F.relu(self.linear1(x))  
        mu = self.linear2(x)          ②  
        std = torch.exp(self.linear3(x)) ③  
        z = mu + std*self.N.sample(mu.shape) ④  
        return mu, std, z
```

① La dimensión del espacio latente es 100.

② La media de la distribución de las codificaciones

③ La desviación estándar de las codificaciones

④ La representación vectorial codificada

La red del codificador consta de varias capas convolucionales que extraen las características espaciales de las imágenes de entrada. El codificador comprime las entradas en representaciones vectoriales `z`, que se distribuyen normalmente con medias `mu`, y desviaciones típicas `std`. La salida del codificador consta de tres tensores: `mu`, `std`, y `z`. Mientras que `mu` y `std` son la media y la desviación típica del vector probabilístico, respectivamente, `z` es una instancia muestreada de esta distribución.

Especificamente, la imagen de entrada, con un tamaño de (3, 256, 256), pasa primero por una capa Conv2d con un valor de paso de 2. Como explicamos en el capítulo 4, esto significa que el filtro omite dos píxeles cada vez que se mueve sobre la imagen de entrada, lo que provoca un submuestreo de la imagen. La salida tiene un tamaño de (8, 128, 128). Luego pasa por dos capas Conv2d más, y el tamaño se convierte en (32, 31, 31). Se aplana y se pasa por capas lineales para obtener valores de `mu` y `std`.

Definimos una `Decoder()` clase para representar el decodificador en el VAE.

Listado 7.5 El decodificador en el VAE

```
class Decoder(nn.Module):
    def __init__(self, latent_dims=100):
        super().__init__()
        self.decoder_lin = nn.Sequential(
            nn.Linear(latent_dims, 1024),
            nn.ReLU(True),
            nn.Linear(1024, 31*31*32),
            nn.ReLU(True))
        self.unflatten = nn.Unflatten(dim=1,
                                      unflattened_size=(32,31,31))
        self.decoder_conv = nn.Sequential(
            nn.ConvTranspose2d(32,16,3,stride=2,
                             output_padding=1),
            nn.BatchNorm2d(16),
            nn.ReLU(True),
            nn.ConvTranspose2d(16, 8, 3, stride=2,
                             padding=1, output_padding=1),
            nn.BatchNorm2d(8),
            nn.ReLU(True),
```

```

        nn.ConvTranspose2d(8, 3, 3, stride=2,
                           padding=1, output_padding=1))

    def forward(self, x):
        x = self.decoder_lin(x)
        x = self.unflatten(x)
        x = self.decoder_conv(x)
        x = torch.sigmoid(x) ④
        return x

```

- ① Las codificaciones primero pasan por dos capas densas.
- ② Reestructura las codificaciones en objetos multidimensionales para que podamos realizar operaciones convolucionales transpuestas en ellos.
- ③ Pasa las codificaciones a través de tres capas convolucionales transpuestas
- ④ Comprime la salida a valores entre 0 y 1, los mismos que los valores en las imágenes de entrada

El decodificador es una imagen especular del codificador: en lugar de realizar operaciones convolucionales, realiza operaciones convolucionales transpuestas sobre las codificaciones para generar mapas de características. Convierte gradualmente las codificaciones del espacio latente en imágenes en color de alta resolución.

Especificamente, la codificación primero pasa por dos capas lineales. Luego, se desacopla a una forma (32, 31, 31), reflejando el tamaño de la imagen después de la última capa Conv2d en el codificador. Después, pasa por tres capas ConvTranspose2d, reflejando las capas Conv2d en el codificador. La salida del decodificador tiene una forma de (3, 256, 256), igual que la de la imagen de entrenamiento.

Combinaremos el codificador con el decodificador para crear un VAE:

```

class VAE(nn.Module):
    def __init__(self, latent_dims=100):
        super().__init__()
        self.encoder = Encoder(latent_dims) ①
        self.decoder = Decoder(latent_dims) ②
    def forward(self, x):
        x = x.to(device)

```

```
    mu, std, z = self.encoder(x)
    return mu, std, self.decoder(z)
```

(3)

(4)

- ① Crea un codificador instanciando la clase Encoder()
- ② Crea un decodificador instanciando la clase Decoder()
- ③ Pasa la entrada a través del codificador para obtener la codificación
- ④ La salida del VAE es la media y la desviación estándar de las codificaciones, así como las imágenes reconstruidas.

El VAE consta de un codificador y un decodificador, según lo definido por las clases `Encoder()` y `Decoder()`. Al pasar imágenes a través del VAE, la salida consta de tres tensores: la media y la desviación estándar de las codificaciones y las imágenes reconstruidas.

A continuación, creamos un VAE instanciando la clase `VAE()` y definimos el optimizador para el modelo:

```
vae=VAE().to(device)
lr=1e-4
optimizer=torch.optim.Adam(vae.parameters(),
                           lr=lr,weight_decay=1e-5)
```

Calcularemos manualmente la pérdida de reconstrucción y la pérdida de divergencia KL durante el entrenamiento. Por lo tanto, no definimos una función de pérdida aquí.

7.4.2 Entrenamiento del VAE

Para entrenar el modelo, primero definimos una función `train_epoch()` para entrenar el modelo para una época.

Listado 7.6 Definición de la función `train_epoch()`

```
def train_epoch(epoch):
    vae.train()
    epoch_loss = 0.0
    for imgs, _ in loader:
        imgs = imgs.to(device)
        mu, std, out = vae(imgs)
```

(1)

```

reconstruction_loss = ((imgs-out)**2).sum()
(2)
kl = ((std**2)/2 + (mu**2)/2 - torch.log(std) - 0.5).sum()
(3)
loss = reconstruction_loss + kl
(4)
optimizer.zero_grad()
loss.backward()
optimizer.step()
epoch_loss+=loss.item()
print(f'at epoch {epoch}, loss is {epoch_loss}')

```

① Obtiene las imágenes reconstruidas

② Calcula la pérdida de reconstrucción

③ Calcula la divergencia KL

④ Suma de la pérdida de reconstrucción y la divergencia KL.

Iteramos todos los lotes del conjunto de entrenamiento. Pasamos las imágenes por el VAE para obtener imágenes reconstruidas. La pérdida total es la suma de la pérdida de reconstrucción y la divergencia KL. Los parámetros del modelo se ajustan en cada iteración para minimizar la pérdida total.

También definimos una función `plot_epoch()` para inspeccionar visualmente las imágenes generadas por el VAE:

```

import numpy as np
import matplotlib.pyplot as plt

def plot_epoch():
    with torch.no_grad():
        noise = torch.randn(18,latent_dims).to(device)
        imgs = vae.decoder(noise).cpu()
        imgs = torchvision.utils.make_grid(imgs,6,3).numpy()
        fig, ax = plt.subplots(figsize=(6,3),dpi=100)
        plt.imshow(np.transpose(imgs, (1, 2, 0)))
        plt.axis("off")
        plt.show()

```

Un VAE bien entrenado puede mapear entradas similares a puntos cercanos en el espacio latente, lo que resulta en un espacio latente más continuo e interpretable. Como resultado, podemos extraer vectores aleatoriamente del espacio latente, y el VAE puede decodificarlos en salidas significativas. Por lo tanto, en la función anterior `plot_epoch()`, extraemos aleatoriamente 18 vectores del es-

pacio latente y los usamos para generar 18 imágenes después de cada época de entrenamiento. Los trazamos en una cuadrícula de 3×6 y los inspeccionamos visualmente para ver el rendimiento del VAE durante el proceso de entrenamiento.

A continuación, entrenamos el VAE durante 10 épocas:

```
for epoch in range(1,11):
    train_epoch(epoch)
    plot_epoch()
    torch.save(vae.state_dict(),"files/VAEglasses.pth")
```

Este entrenamiento dura aproximadamente media hora si usas entrenamiento con GPU, o varias horas si no. Los pesos del modelo entrenado se guardan en tu computadora. También puedes descargarlos de mi sitio web:

<https://mng.bz/GNRR>. Asegúrate de descomprimir el archivo después de descargarlo.

7.4.3 Generación de imágenes con el VAE entrenado

Ahora que el VAE está entrenado, podemos usarlo para generar imágenes. Primero, cargamos los pesos del modelo entrenado que guardamos en la carpeta local:

```
vae.eval()
vae.load_state_dict(torch.load('files/VAEglasses.pth',
    map_location=device))
```

Luego comprobamos la capacidad del VAE para reconstruir imágenes y vemos qué tan parecidas son a las originales:

```
imgs,_=next(iter(loader))
imgs = imgs.to(device)
mu, std, out = vae(imgs)
images=torch.cat([imgs[:8],out[:8],imgs[8:16],out[8:16]],
                 dim=0).detach().cpu()
images = torchvision.utils.make_grid(images,8,4)
fig, ax = plt.subplots(figsize=(8,4),dpi=100)
plt.imshow(np.transpose(images, (1, 2, 0)))
plt.axis("off")
plt.show()
```

Si ejecuta el bloque de código anterior, verá una salida similar a la figura 7.7.

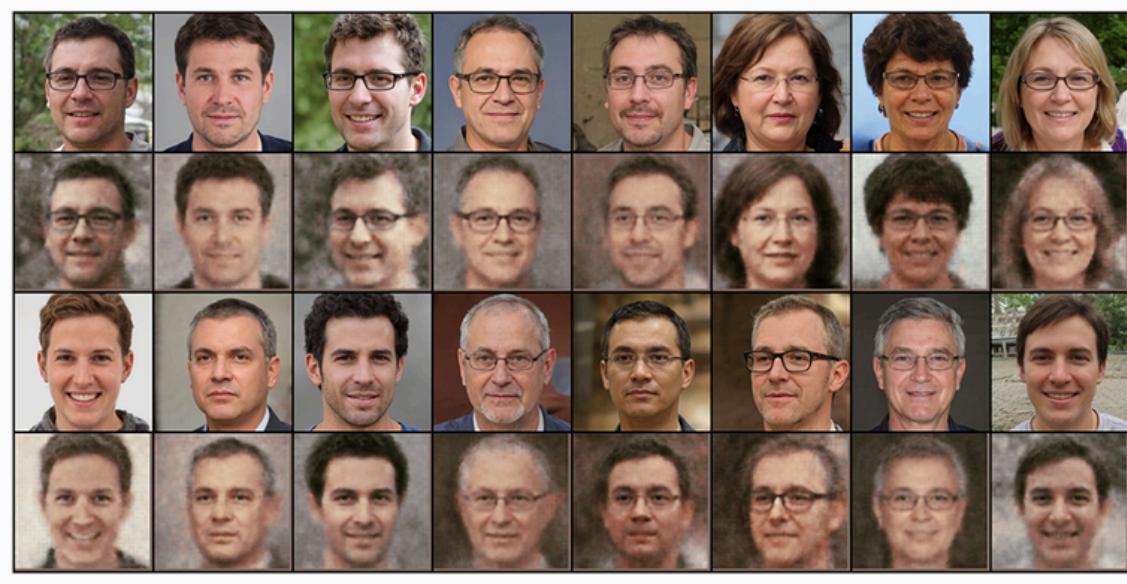


Figura 7.7 Comparación de las imágenes reconstruidas por un VAE entrenado con las originales. La primera y la tercera fila corresponden a las imágenes originales. Las introducimos en el VAE entrenado para obtener las imágenes reconstruidas, que se muestran debajo de las originales.

Las imágenes originales se muestran en la primera y tercera filas, mientras que las imágenes reconstruidas se muestran debajo. Las imágenes reconstruidas se asemejan a las originales, como se muestra en la figura 7.7. Sin embargo, se pierde información durante el proceso de reconstrucción: no se ven tan realistas como las originales.

A continuación, probamos la capacidad del VAE para generar imágenes nuevas que no se ven en el conjunto de entrenamiento, llamando a la función `plot_epoch()` que definimos antes:

```
plot_epoch()
```

La función extrae aleatoriamente 18 vectores del espacio latente y los pasa al VAE entrenado para generar 18 imágenes. El resultado se muestra en la figura 7.8.



Figura 7.8 Imágenes nuevas generadas por el VAE entrenado. Extraemos aleatoriamente representaciones vectoriales en el espacio latente y las introducimos en el decodificador del VAE entrenado. Las imágenes decodificadas se muestran en esta figura. Dado que las representaciones vectoriales se extraen aleatoriamente, las imágenes no corresponden a ninguna imagen original del conjunto de entrenamiento.

Estas imágenes no están presentes en el conjunto de entrenamiento: las codificaciones se extraen aleatoriamente del espacio latente, no de los vectores codificados tras pasar las imágenes del conjunto de entrenamiento por el codificador. Esto se debe a que el espacio latente en los VAE es continuo e interpretable. Las codificaciones nuevas e inéditas en el espacio latente pueden decodificarse significativamente en imágenes que se asemejan, pero difieren, de las del conjunto de entrenamiento.

7.4.4 Codificación aritmética con el VAE entrenado

Las VAE incluyen un término de regularización (divergencia KL) en su función de pérdida, lo que facilita que el espacio latente se aproxime a una distribución normal. Esta regularización garantiza que las variables latentes no solo memoricen los datos de entrenamiento, sino que capturen la distribución subyacente. Esto ayuda a lograr un espacio latente bien estructurado donde los puntos de datos similares se mapean estrechamente, lo que hace que el espacio sea continuo e interpretable. Como resultado, podemos manipular las codificaciones para obtener nuevos resultados.

Para que los resultados sean reproducibles, te animo a que... Para descargar los pesos entrenados de mi sitio web (<https://mng.bz/GNRR>) y usar los mismos bloques de código para el resto del capítulo. Como explicamos en la introducción, la codificación aritmética nos permite generar imágenes con ciertas características. Para ilustrar cómo funciona la codificación aritmética en VAE, primero recopilemos manualmente tres imágenes de cada uno de los siguien-

tes cuatro grupos: hombres con gafas, hombres sin gafas, mujeres con gafas y mujeres sin gafas.

Listado 7.7 Recopilación de imágenes con diferentes características

```
torch.manual_seed(0)
glasses=[]
for i in range(25):
    img,label=data[i]
    glasses.append(img)
    plt.subplot(5,5,i+1)
    plt.imshow(img.numpy().transpose((1,2,0)))
    plt.axis("off")
plt.show()
men_g=[glasses[0],glasses[3],glasses[14]]          ①
women_g=[glasses[9],glasses[15],glasses[21]]        ②
③

noglasses=[]
for i in range(25):
    img,label=data[-i-1]
    noglasses.append(img)
    plt.subplot(5,5,i+1)
    plt.imshow(img.numpy().transpose((1,2,0)))
    plt.axis("off")
plt.show()
men_ng=[noglasses[1],noglasses[7],noglasses[22]]      ④
women_ng=[noglasses[4],noglasses[9],noglasses[19]])    ⑤
⑥
```

① Muestra 25 imágenes con gafas

② Selecciona tres imágenes de hombres con gafas

③ Selecciona tres imágenes de mujeres con gafas

④ Muestra 25 imágenes sin gafas

⑤ Selecciona tres imágenes de hombres sin gafas

⑥ Selecciona tres imágenes de mujeres sin gafas

Seleccionamos tres imágenes en cada grupo en lugar de una sola para calcular el promedio de múltiples codificaciones del mismo grupo al realizar posteriormente cálculos de codificación. Los VAE están diseñados para aprender la distribución de los datos de entrada en el espacio latente. Al promediar múltiples

codificaciones, suavizamos eficazmente la representación en este espacio. Esto nos ayuda a encontrar una representación promedio que capture las características comunes entre las diferentes muestras de un grupo.

A continuación, introducimos las tres imágenes de hombres con gafas en el VAE entrenado para obtener sus codificaciones en el espacio latente. Calculamos la codificación promedio de las tres imágenes y la utilizamos para obtener una imagen reconstruida de un hombre con gafas. Repetimos este proceso para los otros tres grupos.

Listado 7.8 Codificación y decodificación de imágenes en cuatro grupos diferentes

```
# create a batch of images of men with glasses
men_g_batch = torch.cat((men_g[0].unsqueeze(0),
                         men_g[1].unsqueeze(0),
                         men_g[2].unsqueeze(0)), dim=0).to(device) ①

# Obtain the three encodings
_, _, men_g_encodings=vae.encoder(men_g_batch)
# Average over the three images to obtain the encoding for the group
men_g_encoding=men_g_encodings.mean(dim=0) ②

# Decode the average encoding to create an image of a man with glasses
men_g_recon=vae.decoder(men_g_encoding.unsqueeze(0)) ③

# Do the same for the other three groups
# group 2, women with glasses
women_g_batch = torch.cat((women_g[0].unsqueeze(0),
                           women_g[1].unsqueeze(0),
                           women_g[2].unsqueeze(0)), dim=0).to(device)

# group 3, men without glasses
men_ng_batch = torch.cat((men_ng[0].unsqueeze(0),
                          men_ng[1].unsqueeze(0),
                          men_ng[2].unsqueeze(0)), dim=0).to(device)

# group 4, women without glasses
women_ng_batch = torch.cat((women_ng[0].unsqueeze(0),
                            women_ng[1].unsqueeze(0),
                            women_ng[2].unsqueeze(0)), dim=0).to(device)

# obtain average encoding for each group
_, _, women_g_encodings=vae.encoder(women_g_batch)
women_g_encoding=women_g_encodings.mean(dim=0)
_, _, men_ng_encodings=vae.encoder(men_ng_batch)
men_ng_encoding=men_ng_encodings.mean(dim=0)
_, _, women_ng_encodings=vae.encoder(women_ng_batch)
women_ng_encoding=women_ng_encodings.mean(dim=0) ④

# decode for each group
```

```
women_g_recon=vae.decoder(women_g_encoding.unsqueeze(0))  
men_ng_recon=vae.decoder(men_ng_encoding.unsqueeze(0))  
women_ng_recon=vae.decoder(women_ng_encoding.unsqueeze(0))
```

(5)

- ① Crea un lote de imágenes de hombres con gafas.
- ② Obtiene la codificación promedio para hombres con gafas
- ③ Descodifica la codificación promedio para hombres con gafas
- ④ Obtiene las codificaciones promedio para los otros tres grupos
- ⑤ Decodifica las codificaciones promedio de los otros tres grupos

Las codificaciones promedio para los cuatro grupos son `men_g_encoding`, `women_g_encoding`, `men_ng_encoding`, y `women_ng_encoding`, respectivamente, donde `g` representa con y `ng` sin gafas. Las imágenes decodificadas para los cuatro grupos son `men_g_recon`, `women_g_recon`, `men_ng_recon`, y `women_ng_recon`, respectivamente. Graficamos las cuatro imágenes:

```
imgs=torch.cat((men_g_recon,  
                women_g_recon,  
                men_ng_recon,  
                women_ng_recon),dim=0)  
imgs=torchvision.utils.make_grid(imgs,4,1).cpu().numpy()  
imgs=np.transpose(imgs,(1,2,0))  
fig, ax = plt.subplots(figsize=(8,2),dpi=100)  
plt.imshow(imgs)  
plt.axis("off")  
plt.show()
```

Verá el resultado como se muestra en la figura 7.9.



Figura 7.9 Imágenes decodificadas según las codificaciones promedio. Primero, obtenemos tres imágenes de cada uno de los siguientes cuatro grupos: hombres con gafas, mujeres con gafas, hombres sin gafas y mujeres sin gafas. Introducimos las 12 imágenes en el codificador del VAE entrenado para obtener sus codificaciones en el espacio latente. A continuación, calculamos la codificación promedio de las tres imágenes de cada grupo. Las cuatro codificaciones promedio se introducen en el decodificador del VAE entrenado para obtener cuatro imágenes, que se muestran en esta figura.

Las cuatro imágenes decodificadas se muestran en la figura 7.9. Son las imágenes compuestas que representan los cuatro grupos. Observe que son diferentes de las 12 imágenes originales. Al mismo tiempo, conservan las características definitorias de cada grupo.

A continuación, manipularemos las codificaciones para crear una nueva y usaremos el decodificador entrenado en el VAE para decodificarla y ver qué sucede. Por ejemplo, podemos restar la codificación promedio de las mujeres con gafas de la codificación promedio de los hombres con gafas y sumar la codificación promedio de las mujeres sin gafas. Luego, enviamos el resultado al decodificador y vemos la salida.

Listado 7.9 Un ejemplo de codificación aritmética

```
z=men_g_encoding-women_g_encoding+women_ng_encoding          ①
out=vae.decoder(z.unsqueeze(0))                                ②
imgs=torch.cat((men_g_recon,
                women_g_recon,
                women_ng_recon,out),dim=0)
imgs=torchvision.utils.make_grid(imgs,4,1).cpu().numpy()
imgs=np.transpose(imgs,(1,2,0))
fig, ax = plt.subplots(figsize=(8,2),dpi=100)                  ③
plt.imshow(imgs)
plt.title("man with glasses - woman \
with glasses + woman without \
glasses = man without glasses ",fontsize=10,c="r")           ④
plt.axis("off")
plt.show()
```

① Define z como la codificación de hombres con gafas – mujeres con gafas + mujeres sin gafas

② Decodifica z para generar una imagen

③ Muestra las cuatro imágenes

④ Muestra un título en la parte superior de las imágenes

Si ejecuta el bloque de código del listado 7.9, verá una salida como la que se muestra en la figura 7.10.

man with glasses - woman with glasses + woman without glasses = man without glasses



Figura 7.10. Ejemplo de codificación aritmética con el VAE entrenado. Primero, obtenemos las codificaciones promedio de los tres grupos siguientes: hombres con gafas (z_1), mujeres con gafas (z_2) y mujeres sin gafas (z_3). Definimos una nueva codificación $z = z_1 - z_2 + z_3$. A continuación, introducimos z en el decodificador del VAE entrenado y obtenemos la imagen decodificada, como se muestra en el extremo derecho de esta figura.

Las tres primeras imágenes de la figura 7.10 son las imágenes compuestas que representan los tres grupos de entrada. La imagen de salida, en el extremo derecho, es la de un hombre sin gafas.

Dado que tanto "`men_g_encoding`" y "`women_g_encoding`" como conducen a la presencia de gafas en las imágenes al decodificarse, `men_g_encoding` "-" `women_g_encoding` anula las características de las gafas en la imagen resultante. De igual manera, dado que tanto "`women_ng_encoding`" como `women_g_encoding`" conducen a un rostro femenino, `women_ng_encoding` "-" `women_g_encoding` anula las características femeninas en la imagen resultante. Por lo tanto, si se decodifica "`men_g_encoding` + `women_g_encoding` - `women_ng_encoding`" con el VAE entrenado, se obtendrá la imagen de un hombre sin gafas. La aritmética de codificación en este ejemplo muestra que se puede obtener una codificación para hombres sin gafas manipulando las codificaciones promedio de los otros tres grupos.

Realice las siguientes operaciones aritméticas de codificación modificando el listado de código 7.9:

1. Reste la codificación promedio de los hombres sin gafas de la codificación promedio de los hombres con gafas y sume la codificación promedio de las mujeres sin gafas. Introduzca el resultado en el decodificador y observe qué sucede.
2. Reste la codificación promedio de las mujeres sin gafas de la codificación promedio de los hombres sin gafas y sume la codificación promedio de las mujeres con gafas. Introduzca el resultado en el decodificador y observe qué sucede.
3. Reste la codificación promedio de los hombres sin gafas de la codificación promedio de las mujeres sin gafas y sume la codificación promedio de los hombres con gafas. Pase el resultado al decodificador y observe qué sucede. Asegúrese de modificar los títulos de las imágenes para reflejar los cambios. Las soluciones se encuentran en el repositorio de GitHub del libro: <https://github.com/markhliu/DGAI>.

Además, podemos interpolar dos codificaciones cualesquiera en el espacio latente asignándoles pesos diferentes y creando una nueva codificación. A continuación, podemos decodificar la nueva codificación y crear una imagen compuesta. Al elegir pesos diferentes, podemos crear una serie de imágenes intermedias que transicionan entre sí.

Tomemos como ejemplo las codificaciones de mujeres con y sin gafas. Definiremos una nueva codificación `z` como `w*women_ng_encoding+(1-w)*women_g_encoding`, donde `w` es el peso que asignamos a `women_ng_encoding`. Cambiaremos el valor de `w` de 0 a 1 con un incremento de 0,2 en cada paso. Luego, las decodificamos y mostramos las seis imágenes resultantes.

Listado 7.10 Interpolación de dos codificaciones para crear una serie de imágenes

```
results=[]
for w in [0, 0.2, 0.4, 0.6, 0.8, 1.0]:          ①
    z=w*women_ng_encoding+(1-w)*women_g_encoding ②
    out=vae.decoder(z.unsqueeze(0))                  ③
    results.append(out)
imgs=torch.cat((results[0],results[1],results[2],
               results[3],results[4],results[5]),dim=0)
```

```



```

(4)

① Itera a través de seis valores diferentes de w

② Interpola entre dos codificaciones

③ Decodifica la codificación interpolada

④ Muestra las seis imágenes resultantes

Después de ejecutar el código del listado 7.10, verá una salida como la que se muestra en la figura 7.11.



Figura 7.11 Interpolación de codificaciones para crear una serie de imágenes intermedias. Primero, obtenemos las codificaciones promedio para mujeres con gafas (`women_g_encoding`) y mujeres sin gafas (`women_ng_encoding`). La codificación interpolada z se define como $w * \text{women_ng_encoding} + (1-w) * \text{women_g_encoding}$, donde w es el peso de `women_ng_encoding`. Cambiamos el valor de w de 0 a 1 con un incremento de 0,2 para crear seis codificaciones interpoladas. Luego, las decodificamos y mostramos las seis imágenes resultantes en la figura.

Como se puede observar en la figura 7.11, al desplazarse de izquierda a derecha, la imagen cambia gradualmente de una mujer con gafas a una mujer sin gafas. Esto demuestra que las codificaciones en el espacio latente son continuas, significativas e interpolables.

Ejercicio 7.2

Modifique el listado 7.10 para crear una serie de imágenes intermedias utilizando los siguientes pares de codificaciones: (i) `men_ng_encoding` y `men_g_encoding`; (ii) `men_ng_encoding` y `women_ng_encoding`; (iii) `men_g_encoding` y `women_g_encoding`. Las soluciones se encuentran en el repositorio de GitHub del libro: <https://github.com/markhliu/DGAI>.

A partir del siguiente capítulo, te embarcarás en un viaje por el procesamiento del lenguaje natural. Esto te permitirá generar otro tipo de contenido: texto. Sin embargo, muchas de las herramientas que has utilizado hasta ahora se reutilizarán en capítulos posteriores, como las redes neuronales profundas y la arquitectura de codificador-decodificador.

Resumen

- Los AE tienen una estructura de dos componentes: un codificador y un decodificador. El codificador comprime los datos en una representación abstracta en un espacio de menor dimensión (el espacio latente), y el decodificador descomprime la información codificada y reconstruye los datos.
- Los VAE también constan de un codificador y un decodificador. Se diferencian de los AE en dos aspectos fundamentales. En primer lugar, mientras que un AE codifica cada entrada en un punto específico del espacio latente, un VAE la codifica en una distribución de probabilidad dentro de este espacio. En segundo lugar, un AE se centra exclusivamente en minimizar el error de reconstrucción, mientras que un VAE aprende los parámetros de la distribución de probabilidad de las variables latentes, minimizando así una función de pérdida que incluye tanto la pérdida de reconstrucción como un término de regularización: la divergencia KL.
- La divergencia KL en la función de pérdida durante el entrenamiento de las VAE garantiza que la distribución de las variables latentes se asemeje a una distribución normal. Esto fomenta el aprendizaje de representaciones latentes continuas, significativas y generalizables por parte del codificador.
- Un VAE bien entrenado puede mapear entradas similares a puntos cercanos en el espacio latente, lo que resulta en un espacio latente más continuo e interpretable. Como resultado, los VAE pueden decodificar vectores aleatorios en el espacio latente en salidas significativas, generando imágenes invisibles en el conjunto de entrenamiento.
- El espacio latente en un VAE es continuo e interpretable, a diferencia del de un AE. Como resultado, podemos manipular las codificaciones para lograr nuevos resultados. También podemos crear una serie de imágenes intermedias que transicionan de una instancia a otra variando los pesos de dos codificaciones en el espacio latente.

¹ Diederik P Kingma y Max Welling, 2013, “Bayes variacional de codificación automática”. <https://arxiv.org/abs/1312.6114>.

Parte 3. Procesamiento del lenguaje natural y transformadores

La Parte III se centra en la generación de texto.

En el capítulo 8, aprenderá a construir y entrenar una red neuronal recurrente para generar texto. A lo largo del camino, aprenderá cómo funcionan la tokenización y la incrustación de palabras. También aprenderá a generar texto autorregresivamente y cómo usar la temperatura y el muestreo top-K para controlar la creatividad del texto generado. En los capítulos 9 y 10, construirá un Transformer desde cero, basado en el artículo "Attention Is All You Need", para traducir del inglés al francés. En el capítulo 11, aprenderá a construir GPT-2XL, la versión más grande de GPT-2, desde cero. Después de eso, aprenderá a extraer los pesos preentrenados de Hugging Face y cargarlos en su propio modelo GPT-2. Usará su GPT-2 para generar texto al alimentar un mensaje al modelo. En el capítulo 12, construirá y entrenará un modelo GPT para generar texto al estilo Hemingway.

8. Generación de texto con redes neuronales recurrentes

Este capítulo cubre

- La idea detrás de las RNN y por qué pueden manejar datos secuenciales
- Tokenización de caracteres, tokenización de palabras y tokenización de subpalabras
- Cómo funciona la incrustación de palabras
- Construcción y entrenamiento de una RNN para generar texto
- Uso de la temperatura y el muestreo top-K para controlar la creatividad en la generación de texto

Hasta ahora en este libro, hemos analizado cómo generar formas, números e imágenes. A partir de este capítulo, nos centraremos principalmente en la generación de texto. La generación de texto se considera a menudo el santo grial de la IA generativa por varias razones convincentes. El lenguaje humano es increíblemente complejo y lleno de matices. Implica comprender no solo la gramática y el vocabulario, sino también el contexto, el tono y las referencias culturales. Generar texto coherente y contextualizado con éxito es un reto importante que requiere una profunda comprensión y procesamiento del lenguaje.

Como humanos, nos comunicamos principalmente a través del lenguaje. La IA capaz de generar texto con un lenguaje similar al humano puede interactuar de forma más natural con los usuarios, haciendo que la tecnología sea más accesible e intuitiva. La generación de texto tiene múltiples aplicaciones, desde la automatización de las respuestas de atención al cliente hasta la creación de artículos completos, la creación de guiones para juegos y películas, la asistencia en la escritura creativa e incluso el desarrollo de asistentes personales. El impacto potencial en diferentes sectores es enorme.

En este capítulo, haremos nuestro primer intento de construir y entrenar modelos para generar texto. Aprenderá a abordar tres desafíos principales

en el modelado de la generación de texto. Primero, el texto son datos secuenciales, que consisten en puntos de datos organizados en una secuencia específica, donde cada punto se ordena sucesivamente para reflejar el orden inherente y las interdependencias dentro de los datos. Predecir los resultados de las secuencias es un desafío debido a su ordenamiento sensible. Alterar la secuencia de elementos cambia su significado. Segundo, el texto exhibe dependencias de largo alcance: el significado de cierta parte del texto depende de elementos que aparecieron mucho antes en el texto (por ejemplo, hace 100 palabras). Comprender y modelar estas dependencias de largo alcance es esencial para generar texto coherente. Por último, el lenguaje humano es ambiguo y depende del contexto. Entrenar un modelo para comprender matices, sarcasmo, modismos y referencias culturales para generar texto contextualmente preciso es un desafío.

Explorarás una red neuronal específica diseñada para gestionar datos secuenciales, como texto o series temporales: la red neuronal recurrente (RNN). Las redes neuronales tradicionales, como las redes neuronales de propagación hacia adelante o las redes totalmente conectadas, tratan cada entrada de forma independiente. Esto significa que la red procesa cada entrada por separado, sin considerar ninguna relación ni orden entre las diferentes entradas. Por el contrario, las RNN están diseñadas específicamente para gestionar datos secuenciales. En una RNN, la salida en un paso de tiempo determinado depende no solo de la entrada actual, sino también de las entradas anteriores. Esto permite que las RNN mantengan una forma de memoria, capturando información de pasos de tiempo anteriores para influir en el procesamiento de la entrada actual.

Este procesamiento secuencial hace que las RNN sean adecuadas para tareas donde el orden de las entradas es importante, como el modelado lingüístico, donde el objetivo es predecir la siguiente palabra de una oración basándose en palabras anteriores. Nos centraremos en una variante de las RNN, las redes de memoria a corto y largo plazo (LSTM), que pueden reconocer patrones de datos tanto a corto como a largo plazo en datos secuenciales, como el texto. Los modelos LSTM utilizan un estado oculto para capturar información en intervalos de tiempo anteriores. Por lo tanto, un modelo LSTM entrenado puede producir texto coherente basado en el contexto.

El estilo del texto generado depende de los datos de entrenamiento. Además, como planeamos entrenar un modelo desde cero para la generación de texto, la longitud del texto de entrenamiento es un factor crucial. Debe ser lo suficientemente extenso para que el modelo aprenda e imite eficazmente un estilo de escritura particular, pero lo suficientemente conciso para evitar demandas computacionales excesivas durante el entrenamiento. Como resultado, usaremos el texto de la novela *Anna Karenina*, que parece tener la longitud adecuada para nuestros propósitos, para entrenar un modelo LSTM. Dado que las redes neuronales como una LSTM no pueden aceptar texto como entrada directamente, aprenderá a descomponer el texto en tokens (palabras individuales en este capítulo, pero que pueden ser partes de palabras, como verá en capítulos posteriores), un proceso conocido como *tokenización*. Luego, creará un diccionario para mapear cada token único en un entero (es decir, un índice). Con base en este diccionario, convertirá el texto en una larga secuencia de enteros, lista para ser alimentada a una red neuronal.

Utilizarás secuencias de índices de cierta longitud como entrada para entrenar el modelo LSTM. Desplazas la secuencia de entradas un token a la derecha y la usas como salida: estás entrenando al modelo para predecir el siguiente token en una oración. Este es el llamado problema de predicción *secuencia a secuencia* en el procesamiento del lenguaje natural (PLN), y lo verás de nuevo en capítulos posteriores.

Una vez entrenado el LSTM, lo usará para generar texto, un token a la vez, basándose en tokens anteriores en la secuencia, de la siguiente manera: introduce una instrucción (parte de una oración como "Anna y el") en el modelo entrenado. El modelo predice el siguiente token más probable y lo añade a su instrucción. La instrucción actualizada sirve de nuevo como entrada, y el modelo se utiliza una vez más para predecir el siguiente token. El proceso iterativo continúa hasta que la instrucción alcanza una longitud determinada. Este enfoque es similar al mecanismo empleado por modelos generativos más avanzados como ChatGPT (aunque ChatGPT no es un LSTM). Verá cómo el modelo LSTM entrenado genera texto gramaticalmente correcto y coherente, con un estilo que coincide con el de la novela original.

Finalmente, también aprenderá a controlar la creatividad del texto generado mediante la temperatura y el muestreo top-K. La temperatura con-

trola la aleatoriedad de las predicciones del modelo entrenado. Una temperatura alta hace que el texto generado sea más creativo, mientras que una temperatura baja lo hace más seguro y predecible. El muestreo top-K es un método que consiste en seleccionar el siguiente token de entre los K tokens más probables, en lugar de seleccionarlo de todo el vocabulario. Un valor bajo de K lleva a la selección de tokens altamente probables en cada paso, lo que, a su vez, hace que el texto generado sea menos creativo y más coherente.

El objetivo principal de este capítulo no es necesariamente generar el texto más coherente posible, lo cual, como se mencionó anteriormente, presenta desafíos sustanciales. En cambio, nuestro objetivo es demostrar las limitaciones de las RNN, sentando así las bases para la introducción de los Transformers en capítulos posteriores. Más importante aún, este capítulo establece los principios básicos de la generación de texto, incluyendo la tokenización, la incrustación de palabras, la predicción de secuencias, la configuración de la temperatura y el muestreo top-K. Por consiguiente, en capítulos posteriores, adquirirá una sólida comprensión de los fundamentos del PLN. Esta base nos permitirá centrarnos en otros aspectos más avanzados del PLN, como el funcionamiento del mecanismo de atención y la arquitectura de los Transformers.

8.1 Introducción a las RNN

Al principio de este capítulo, abordamos las complejidades de la generación de texto, en particular al buscar coherencia y relevancia contextual. Esta sección profundiza en estos desafíos y explora la arquitectura de las RNN. Explicaremos por qué las RNN son adecuadas para esta tarea y sus limitaciones (que son las razones por las que han sido superadas por los Transformers).

Las RNN están diseñadas específicamente para gestionar datos secuenciales, lo que les permite generar texto, una tarea inherentemente secuencial. Utilizan un tipo de memoria, conocida como estados ocultos, para capturar y retener información de partes anteriores de la secuencia. Esta capacidad es crucial para mantener el contexto y comprender las dependencias a medida que la secuencia progresá.

En este capítulo, utilizaremos específicamente redes LSTM, versiones avanzadas de RNN, para la generación de texto, usando sus capacidades avanzadas para abordar los desafíos de esta tarea.

8.1.1 Desafíos en la generación de texto

El texto representa un ejemplo por excelencia de *datos secuenciales*, definidos como cualquier conjunto de datos donde el orden de los elementos es crucial. Esta estructuración implica que la posición relativa de los elementos individuales tiene un significado significativo, a menudo transmitiendo información esencial para la comprensión de los datos. Ejemplos de datos secuenciales incluyen series temporales (como precios de acciones), contenido textual (como oraciones) y composiciones musicales (una sucesión de notas).

Este libro se centra principalmente en la generación de texto, aunque también se adentra en la generación musical en los capítulos 13 y 14. El proceso de generación de texto es complejo. Un desafío principal reside en modelar la secuencia de palabras dentro de las oraciones, donde alterar el orden puede cambiar drásticamente el significado. Por ejemplo, en la oración "Kentucky derrotó a Vanderbilt en el partido de fútbol americano de anoche", intercambiar "Kentucky" por "Vanderbilt" invierte por completo la implicación de la oración, a pesar de usar las mismas palabras. Además, como se mencionó en la introducción, la generación de texto enfrenta desafíos al gestionar dependencias de largo alcance y abordar el problema de la ambigüedad.

En este capítulo, exploraremos un enfoque para abordar estos desafíos, concretamente mediante el uso de RNN. Si bien este método no es perfecto, sienta las bases para técnicas más avanzadas que encontrará en capítulos posteriores. Este enfoque le brindará información sobre cómo gestionar el orden de las palabras, abordar las dependencias de largo alcance y abordar la ambigüedad inherente al texto, dotándolo de habilidades fundamentales para la generación de texto. El recorrido por este capítulo le servirá como punto de partida para métodos más sofisticados y una comprensión más profunda en las siguientes partes del libro. A lo largo del camino, adquirirá muchas habilidades valiosas en PNL, como la tokenización de texto, la incrustación de palabras y las predicciones secuencia a secuencia.

8.1.2 ¿Cómo funcionan las RNN?

Las RNN son un tipo especializado de red neuronal artificial, diseñada para reconocer patrones en secuencias de datos, como texto, música o precios de acciones. A diferencia de las redes neuronales tradicionales, que procesan las entradas de forma independiente, las RNN tienen bucles que permiten la persistencia de la información.

Uno de los desafíos en la generación de texto es cómo predecir la siguiente palabra basándose en todas las palabras anteriores para que la predicción capture tanto las dependencias de largo alcance como el significado contextual. Las RNN toman la entrada no solo como un elemento independiente, sino como una secuencia (como las palabras en una oración, por ejemplo). En cada paso de tiempo, la predicción se basa no solo en la entrada actual, sino también en todas las entradas anteriores en forma de resumen a través de un estado oculto. Consideremos la frase "una rana tiene cuatro patas" como ejemplo. En el primer paso de tiempo, usamos la palabra "a" para predecir la segunda palabra "rana". En el segundo paso de tiempo, predecimos la siguiente palabra usando tanto "a" como "rana". Para cuando predigamos la última palabra, necesitamos usar las cuatro palabras anteriores "una rana tiene cuatro".

Una característica clave de las RNN es el llamado estado oculto, que capta información de todos los elementos previos de una secuencia. Esta característica es crucial para que la red pueda procesar y generar datos secuenciales eficazmente. El funcionamiento de las RNN y este procesamiento secuencial se muestra en la figura 8.1, que ilustra cómo se desarrolla una capa de neuronas recurrentes a lo largo del tiempo.

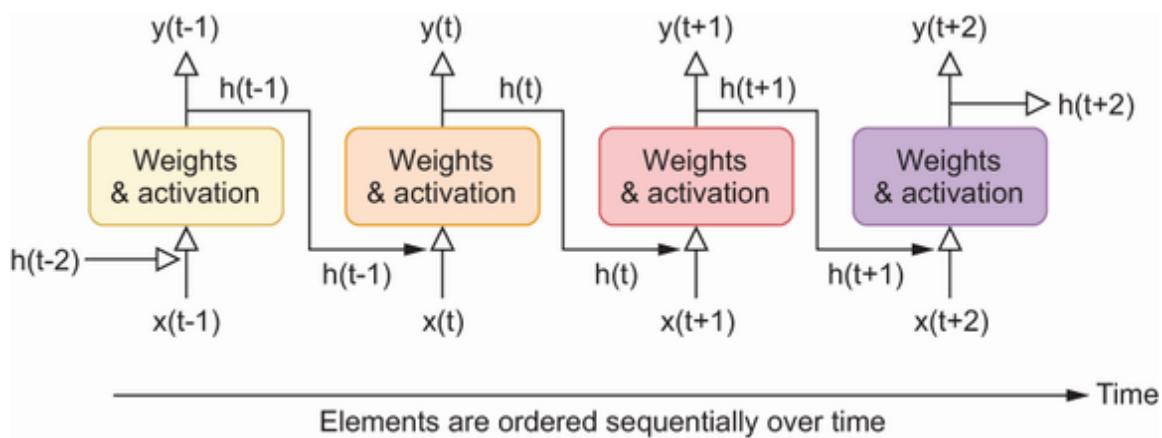


Figura 8.1 Cómo se despliega una capa de neuronas recurrentes a lo largo del tiempo. Cuando una red neuronal recurrente realiza una predicción a partir de datos secuenciales, toma el estado oculto del momento anterior, paso $h(t-1)$, junto con la entrada en el paso de tiempo actual, $x(t)$, genera la salida, $y(t)$, y el estado oculto actualizado, $h(t)$. El estado oculto en el paso de tiempo t captura la información de todos los pasos de tiempo anteriores, $x(0), x(1), \dots, x(t)$.

El estado oculto en las RNN desempeña un papel fundamental en la captura de información a lo largo de todos los intervalos de tiempo. Esto permite a las RNN realizar predicciones basadas no solo en la entrada actual, $x(t)$, sino también en el conocimiento acumulado de todas las entradas anteriores, $x(0), x(1), \dots, x(t-1)$. Esta característica permite a las RNN comprender las dependencias temporales. Pueden captar el contexto de una secuencia de entrada, lo cual es indispensable para tareas como el modelado del lenguaje, donde las palabras anteriores de una oración preparan el terreno para predecir la siguiente palabra.

Sin embargo, las RNN no están exentas de inconvenientes. Si bien las RNN estándar son capaces de gestionar dependencias a corto plazo, presentan dificultades con las dependencias de largo alcance dentro del texto. Esta dificultad se debe al problema del gradiente evanescente, que se produce en secuencias largas donde los gradientes (esenciales para el entrenamiento de la red) disminuyen, lo que dificulta la capacidad del modelo para aprender relaciones a largas distancias. Para mitigar esto, se han desarrollado versiones avanzadas de las RNN, como las redes LSTM.

Las redes LSTM fueron introducidas por Hochreiter y Schmidhuber en 1997.¹ Una red LSTM está compuesta de unidades LSTM (o células), cada una de las cuales tiene una estructura más compleja que una neurona RNN estándar. El estado de la célula es la innovación clave de las LSTM: actúa como una especie de cinta transportadora, que recorre toda la cadena de unidades LSTM. Tiene la capacidad de llevar información relevante a través de la red. La capacidad de agregar o eliminar información del estado

de la célula permite a las LSTM capturar dependencias a largo plazo y recordar información durante largos períodos. Esto las hace más efectivas para tareas como el modelado del lenguaje y la generación de texto. En este capítulo, aprovecharemos el modelo LSTM para emprender un proyecto sobre generación de texto, con el objetivo de imitar el estilo de la novela *Anna Karenina*.

Sin embargo, cabe destacar que incluso las variantes avanzadas de RNN, como las LSTM, enfrentan dificultades para capturar dependencias de alcance extremadamente amplio en datos de secuencias. Analizaremos estos desafíos y ofreceremos soluciones en el siguiente capítulo, continuando nuestra exploración de modelos sofisticados para el procesamiento y la generación eficaces de datos de secuencias.

8.1.3 Pasos para entrenar un modelo LSTM

A continuación, analizaremos los pasos necesarios para entrenar un modelo LSTM para generar texto. Esta descripción general busca proporcionar una comprensión básica del proceso de entrenamiento antes de comenzar el proyecto.

La elección del texto para el entrenamiento depende del resultado deseado. Una novela extensa es un buen punto de partida. Su amplio contenido permite al modelo aprender y replicar eficazmente un estilo de escritura específico. Una cantidad abundante de datos textuales mejora la competencia del modelo en este estilo. Al mismo tiempo, las novelas no suelen ser excesivamente largas, lo que facilita la gestión del tiempo de entrenamiento. Para el entrenamiento de nuestro modelo LSTM, utilizaremos el texto de *Ana Karenina*, en consonancia con los criterios de datos de entrenamiento descritos anteriormente.

Al igual que otras redes neuronales profundas, los modelos LSTM no pueden procesar texto sin procesar directamente. En su lugar, convertiremos el texto a formato numérico. Esto comienza descomponiendo el texto en fragmentos más pequeños, un proceso conocido como tokenización, donde cada fragmento es un token. Los tokens pueden ser palabras completas, signos de puntuación (como un signo de exclamación o una coma) o caracteres especiales (como & o %). En este capítulo, cada uno de estos elementos se tratará como tokens separados. Aunque este método de tokenización

puede no ser el más eficiente, es fácil de implementar, ya que solo necesitamos asignar palabras a tokens. Utilizaremos la tokenización de subpalabras en capítulos posteriores, donde algunas palabras poco frecuentes se descomponen en fragmentos más pequeños, como silabas. Después de la tokenización, asignamos un entero único a cada token, creando una representación numérica del texto como una secuencia de enteros.

Para preparar los datos de entrenamiento, dividimos esta secuencia larga en secuencias más cortas de igual longitud. Para nuestro proyecto, utilizaremos secuencias de 100 enteros cada una. Estas secuencias forman las características (la variable x) de nuestro modelo. Luego, generamos la salida y desplazando la secuencia de entrada un token a la derecha. Esta configuración permite al modelo LSTM predecir el siguiente token de una secuencia. Los pares de entrada y salida sirven como datos de entrenamiento.

Nuestro modelo incluye capas LSTM para comprender patrones a largo plazo en el texto y una capa de incrustación para comprender significados semánticos.

Repasemos el ejemplo de predicción de la frase "una rana tiene cuatro patas" que mencionamos anteriormente. La Figura 8.2 muestra un diagrama de cómo funciona el entrenamiento del modelo LSTM.

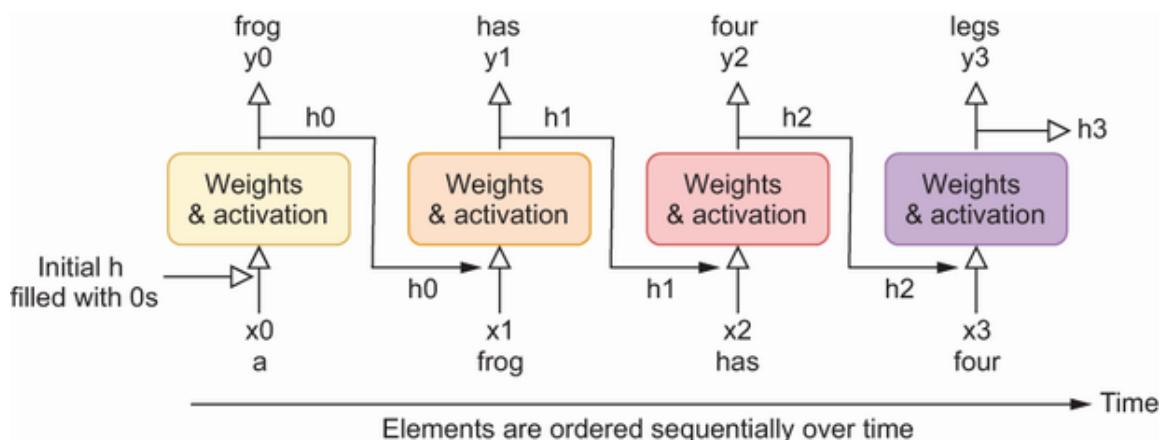


Figura 8.2. Ejemplo de entrenamiento de un modelo LSTM. Primero, descomponemos el texto de entrenamiento en tokens y asignamos un entero único a cada token, creando una representación numérica del texto como una secuencia de índices. A continuación, dividimos esta secuencia larga en secuencias más cortas de igual longitud. Estas secuencias forman las características (la variable x) de nuestro modelo. Finalmente, generamos la salida y desplazando la secuencia de entrada un token a la derecha. Esta configuración permite al modelo LSTM predecir el siguiente token basándose en los tokens anteriores de la secuencia.

En el primer paso de tiempo, el modelo usa la palabra "a" para predecir la palabra "rana". Dado que "a" no tiene una palabra precedente, inicializamos el estado oculto con ceros. El modelo LSTM recibe como entrada tanto

el índice de "a" como este estado oculto inicial y genera la siguiente palabra predicha junto con un estado oculto actualizado, h_0 . En el siguiente paso de tiempo, la palabra "rana" y el estado actualizado h_0 se usan para predecir "has" y generar un nuevo estado oculto, h_1 . Esta secuencia de predicción de la siguiente palabra y actualización del estado oculto continúa hasta que el modelo pronostica la última palabra de la oración, "piernas".

Las predicciones se comparan con la siguiente palabra de la oración. Dado que el modelo predice el siguiente token de entre todos los tokens posibles del vocabulario, existe un problema de clasificación multicategoría. Ajustamos los parámetros del modelo en cada iteración para minimizar la pérdida de entropía cruzada, de modo que en la siguiente iteración, las predicciones del modelo se acerquen más a los resultados reales en los datos de entrenamiento.

Una vez entrenado el modelo, la generación de texto comienza con la introducción de una secuencia de semillas. El modelo predice el siguiente token, que se añade a la secuencia. Este proceso iterativo de predicción y actualización de la secuencia se repite para generar texto durante el tiempo que se desee.

8.2 Fundamentos de la PNL

Los modelos de aprendizaje profundo, incluyendo los modelos LSTM que analizamos anteriormente y los Transformers, que aprenderá en capítulos posteriores, no pueden procesar texto sin procesar directamente porque están diseñados para trabajar con datos numéricos, generalmente en forma de vectores o matrices. Las capacidades de procesamiento y aprendizaje de las redes neuronales se basan en operaciones matemáticas como la suma, la multiplicación y las funciones de activación, que requieren información numérica. Por lo tanto, es esencial primero descomponer el texto en elementos más pequeños y manejables, conocidos como tokens. Estos tokens pueden variar desde caracteres y palabras individuales hasta subpalabras.

El siguiente paso crucial en las tareas de PLN es transformar estos tokens en representaciones numéricas. Esta conversión es necesaria para alimentarlos a redes neuronales profundas, lo cual es fundamental para el entrenamiento de nuestros modelos.

En esta sección, analizaremos diferentes métodos de tokenización, junto con sus ventajas y desventajas. Además, comprenderá el proceso de conversión de tokens en representaciones vectoriales densas, un método conocido como incrustación de palabras. Esta técnica es crucial para capturar el significado del lenguaje en un formato que los modelos de aprendizaje profundo puedan utilizar eficazmente.

8.2.1 Diferentes métodos de tokenización

La tokenización implica dividir el texto en fragmentos más pequeños, conocidos como tokens, que pueden ser palabras, caracteres, símbolos u otras unidades significativas. El objetivo principal de la tokenización es optimizar el proceso de análisis y procesamiento de datos textuales.

En términos generales, existen tres enfoques para la tokenización. El primero es la tokenización de caracteres, donde el texto se divide en sus caracteres constituyentes. Este método se utiliza en idiomas con estructuras morfológicas complejas, como el turco o el finlandés, en los que el significado de las palabras puede cambiar significativamente con ligeras variaciones en los caracteres. Tomemos como ejemplo la frase en inglés "¡Es increíblemente bueno!"; se desglosa en caracteres individuales de la siguiente manera: `['I', 't', ' ', 'i', 's', ' ', 'u', 'n', 'b', 'e', 'l', 'i', 'e', 'v', 'a', 'b', 'l', 'y', ' ', 'g', 'o', 'o', 'd', '!']`. Una ventaja clave de la tokenización de caracteres es el número limitado de tokens únicos. Esta limitación reduce significativamente los parámetros en los modelos de aprendizaje profundo, lo que lleva a un entrenamiento más rápido y eficiente. Sin embargo, el principal inconveniente es que los caracteres individuales a menudo carecen de un significado significativo, lo que dificulta que los modelos de aprendizaje automático obtengan información significativa de una secuencia de caracteres.

Ejercicio 8.1

Utilice la tokenización de caracteres para dividir la frase "¡Hola!" en tokens individuales.

El segundo enfoque es la tokenización de palabras, donde el texto se divide en palabras individuales y signos de puntuación. Se utiliza a menudo en situaciones donde el número de palabras únicas no es demasiado elevado.

Por ejemplo, la misma frase "¡Es increíblemente bueno!" se convierte en cinco tokens: ['It', 'is', 'unbelievably', 'good', '!']. La principal ventaja de este método es que cada palabra tiene un significado semántico inherente, lo que facilita la interpretación del texto por parte de los modelos. Sin embargo, la desventaja radica en el aumento sustancial de tokens únicos, lo que incrementa el número de parámetros en los modelos de aprendizaje profundo. Este aumento puede resultar en procesos de entrenamiento más lentos y menos eficientes.

Ejercicio 8.2

Utilice la tokenización de palabras para dividir la frase “Hola, ¿cómo estás?” en tokens individuales.

El tercer enfoque es la tokenización de subpalabras. Este método, un concepto clave en PNL, divide el texto en componentes más pequeños y significativos llamados subpalabras. Por ejemplo, la frase "¡Es increíblemente bueno!" se dividiría en tokens como ['It', 'is', 'un', 'believ', 'ably', 'good', '!']. La mayoría de los modelos de lenguaje avanzados, como ChatGPT, utilizan la tokenización de subpalabras, método que utilizarás en los próximos capítulos. La tokenización de subpalabras logra un equilibrio entre las técnicas de tokenización más tradicionales, que suelen dividir el texto en palabras o caracteres individuales. La tokenización basada en palabras, si bien captura más significado, genera un vocabulario más amplio. Por el contrario, la tokenización basada en caracteres genera un vocabulario más reducido, pero cada token tiene menos valor semántico.

La tokenización de subpalabras mitiga eficazmente estos problemas al mantener las palabras de uso frecuente en su totalidad en el vocabulario, mientras que divide las palabras menos comunes o más complejas en subcomponentes. Esta técnica es especialmente ventajosa para idiomas con vocabularios extensos o con una alta variación en la forma de las palabras. Al adoptar la tokenización de subpalabras, el tamaño total del vocabulario se reduce sustancialmente. Esta reducción mejora la eficiencia y la eficacia de las tareas de procesamiento del lenguaje, especialmente al trabajar con una amplia gama de estructuras lingüísticas.

En este capítulo, nos centraremos en la tokenización de palabras, ya que ofrece una base sencilla para principiantes. A medida que avancemos en los capítulos posteriores, nos centraremos en la tokenización de subpalabras, utilizando modelos ya entrenados con esta técnica. Este enfoque nos permite centrarnos en temas más avanzados, como comprender la arquitectura de Transformer y explorar el funcionamiento interno del mecanismo de atención.

8.2.2 Incrustación de palabras

La incrustación de palabras es un método que transforma tokens en representaciones vectoriales compactas, capturando su información semántica e interrelaciones. Esta técnica es vital en el procesamiento del lenguaje natural (PLN), especialmente porque las redes neuronales profundas, incluyendo modelos como LSTM y Transformers, requieren entrada numérica.

Tradicionalmente, los tokens se convierten en números mediante codificación one-hot antes de introducirse en los modelos de PLN. En la codificación one-hot, cada token se representa mediante un vector donde solo un elemento es '1' y el resto son '0'. Por ejemplo, en este capítulo, hay 12.778 tokens únicos basados en palabras en el texto de la novela *Anna Karenina*. Cada token se representa mediante un vector de 12.778 dimensiones. En consecuencia, una frase como "las familias felices son todas iguales" se representa como una matriz de 5×12.778 , donde 5 representa el número de tokens. Sin embargo, esta representación es altamente ineficiente debido a su gran dimensionalidad, lo que genera un mayor número de parámetros, lo que puede dificultar la velocidad y la eficiencia del entrenamiento.

Los LSTM, los transformadores y otros modelos avanzados de PLN abordan esta ineficiencia mediante la incrustación de palabras. En lugar de voluminosos vectores one-hot, la incrustación de palabras utiliza vectores continuos de menor dimensión (por ejemplo, los vectores de 128 valores que utilizamos en este capítulo). Como resultado, la frase «las familias felices son todas iguales» se representa mediante una matriz más compacta de 5×128 tras la incrustación de palabras. Esta representación simplificada reduce drásticamente la complejidad del modelo y mejora la eficiencia del entrenamiento.

La incrustación de palabras no solo reduce la complejidad de las palabras al condensarlas en un espacio de menor dimensión, sino que también captura eficazmente el contexto y las relaciones semánticas matizadas entre ellas, una característica de la que carecen las representaciones más simples, como la codificación one-hot, por las siguientes razones. En la codificación one-hot, todos los tokens tienen la misma distancia entre sí en el espacio vectorial. Sin embargo, en las incrustaciones de palabras, los tokens con significados similares se representan mediante vectores cercanos entre sí en el espacio de incrustación. Las incrustaciones de palabras se aprenden del texto en los datos de entrenamiento; los vectores resultantes capturan información contextual. Los tokens que aparecen en contextos similares tendrán incrustaciones similares, incluso si no están explícitamente relacionados.

Incrustación de palabras en PNL

Las incrustaciones de palabras son un método poderoso para representar tokens en PNL que ofrecen ventajas significativas sobre la codificación one-hot tradicional al capturar el contexto y las relaciones semánticas entre palabras.

La codificación one-hot representa los tokens como vectores dispersos con una dimensión igual al tamaño del vocabulario, donde cada token se representa mediante un vector compuesto exclusivamente por ceros, excepto un uno en el índice correspondiente. Por el contrario, las incrustaciones de palabras representan los tokens como vectores densos con dimensiones mucho menores (p. ej., 128 dimensiones en este capítulo y 256 dimensiones en el capítulo 12). Esta representación densa es más eficiente y permite capturar más información.

Específicamente, en la codificación one-hot, todos los tokens tienen la misma distancia entre sí en el espacio vectorial, lo que significa que no existe la noción de similitud entre ellos. Sin embargo, en las incrustaciones de palabras, los tokens similares se representan mediante vectores cercanos en el espacio de incrustación. Por ejemplo, las palabras "rey" y "reina" tendrían incrustaciones similares, lo que refleja su relación semántica.

Las incrustaciones de palabras se aprenden del texto en los datos de entrenamiento. El proceso de incrustación utiliza el contexto en el que aparecen los tokens para aprender sus incrustaciones, lo que significa que los vectores resultantes capturan información contextual. Los tokens que aparecen en contextos similares tendrán incrustaciones similares, incluso si no están explícitamente relacionados.

En general, las incrustaciones de palabras proporcionan una representación más matizada y eficiente de las palabras que capture las relaciones semánticas y la información contextual, lo que las hace más adecuadas para las tareas de PNL en comparación con la codificación one-hot.

En la práctica, especialmente en frameworks como PyTorch, la incrustación de palabras se implementa pasando índices a través de una capa lineal, que los comprime en un espacio de menor dimensión. Es decir, al pasar un índice a la capa `nn.Embedding()`, esta busca la fila correspondiente en la matriz de incrustación y devuelve el vector de incrustación para ese índice, evitando la necesidad de crear vectores one-hot potencialmente muy grandes. Los pesos de esta capa de incrustación no están predefinidos, sino que se aprenden durante el proceso de entrenamiento. Este aspecto del aprendizaje permite al modelo refinar su comprensión de la semántica de las palabras basándose en los datos de entrenamiento, lo que resulta en una representación del lenguaje más matizada y contextualizada en la red neuronal. Este enfoque mejora significativamente la capacidad del modelo para procesar e interpretar datos lingüísticos de forma eficiente y significativa.

8.3 Preparación de datos para entrenar el modelo LSTM

En esta sección, procesaremos datos de texto y los prepararemos para el entrenamiento. Primero, descompondremos el texto en tokens individuales. El siguiente paso consiste en crear un diccionario que asigne un índice a cada token, asignándolos básicamente a números enteros. Tras esta configuración, organizaremos estos tokens en lotes de datos de entrenamiento, lo cual será crucial para entrenar un modelo LSTM en la sección siguiente.

Explicaremos el proceso de tokenización paso a paso y en detalle, asegurándonos de que comprenda a fondo su funcionamiento. Utilizaremos la

tokenización de palabras, debido a su simplicidad para dividir el texto en palabras, a diferencia de la tokenización de subpalabras, más compleja, que requiere un conocimiento profundo de la estructura lingüística. En capítulos posteriores, emplearemos tokenizadores preentrenados para la tokenización de subpalabras mediante métodos más sofisticados. Esto nos permitirá centrarnos en temas avanzados, como el mecanismo de atención y la arquitectura Transformer, sin atascarnos en las etapas iniciales del procesamiento de texto.

8.3.1 Descarga y limpieza del texto

Usaremos el texto de la novela *Ana Karenina* para entrenar nuestro modelo. Vaya a <https://mng.bz/znmX> para descargar el archivo de texto y guárdelo como anna.txt en la carpeta /files/ de su computadora. Después, abra el archivo y elimine todo después de la línea 39888, que dice " "END OF THIS PROJECT GUTENBERG EBOOK ANNA KARENINA . " También puede descargar el archivo anna.txt del repositorio de GitHub del libro: <https://github.com/markhliu/DGAI>.

Primero, cargamos los datos e imprimimos algunos pasajes para tener una idea del conjunto de datos:

```
with open("files/anna.txt","r") as f:  
    text=f.read()  
words=text.split(" ")  
print(words[:20])
```

La salida es

```
[ 'Chapter', '1\n\n\nHappy', 'families', 'are', 'all',  
'alike;', 'every', 'unhappy', 'family', 'is', 'unhappy',  
'in', 'its', 'own\nway.\n\nEverything', 'was', 'in',  
'confusion', 'in', 'the', "Oblonskys'" ]
```

Como puede ver, los saltos de línea (representados por \n) se consideran parte del texto. Por lo tanto, debemos reemplazarlos con espacios para que no se incluyan en el vocabulario. Además, convertir todas las palabras a minúsculas es útil en nuestro contexto, ya que garantiza que palabras como "The" y "the" se reconozcan como el mismo token. Este paso es vital

para reducir la variedad de tokens únicos, lo que hace que el proceso de entrenamiento sea más eficiente. Además, los signos de puntuación deben estar separados de las palabras que los siguen. Sin esta separación, combinaciones como "way." y "way" se tratarían erróneamente como tokens diferentes. Para solucionar estos problemas, limpiaremos el texto:

```
clean_text=text.lower().replace("\n", " ")  
①  
clean_text=clean_text.replace("-", " ")  
②  
for x in ",.:;?!$()/_&%*@'`":  
    clean_text=clean_text.replace(f"\"{x}\\"", f" {x} ")  
clean_text=clean_text.replace("'", ' ' ')  
③  
text=clean_text.split()
```

- ① Reemplaza el salto de línea con un espacio
- ② Reemplaza un guion por un espacio
- ③ Agrega un espacio alrededor de los signos de puntuación y caracteres especiales

A continuación, obtenemos tokens únicos:

```
from collections import Counter  
word_counts = Counter(text)  
words=sorted(word_counts, key=word_counts.get,  
             reverse=True)  
print(words[:10])
```

La lista `words` contiene todos los tokens únicos del texto, con el más frecuente primero y el menos frecuente último. La salida del bloque de código anterior es

```
[',', '.', 'the', "'", 'and', 'to', 'of', 'he', "", 'a']
```

La salida anterior muestra los 10 tokens más frecuentes. La coma (,) y el punto (.) son el token más frecuente y el segundo más frecuente, respectivamente. La palabra "the" es el tercer token más frecuente, y así sucesivamente.

Ahora creamos dos diccionarios: uno que asigna tokens a índices y el otro que asigna índices a tokens.

Listado 8.1 Diccionarios para asignar tokens a índices e índices a tokens

```
text_length=len(text)                                ①
num_unique_words=len(words)                         ②
print(f"the text contains {text_length} words")
print(f"there are {num_unique_words} unique tokens")
word_to_int={v:k for k,v in enumerate(words)}      ③
int_to_word={k:v for k,v in enumerate(words)}        ④
print({k:v for k,v in word_to_int.items() if k in words[:10]}) 
print({k:v for k,v in int_to_word.items() if v in words[:10]})
```

① La longitud del texto (cuántos tokens hay en el texto)

② La longitud de los tokens únicos

③ Asigna tokens a índices

④ Asigna índices a tokens

La salida del bloque de código anterior es

```
the text contains 437098 words
there are 12778 unique tokens
{',': 0, '.': 1, 'the': 2, "'": 3, 'and': 4,
 'to': 5, 'of': 6, 'he': 7, """": 8, 'a': 9}
{0: ',', 1: '.', 2: 'the', 3: "'", 4: 'and',
 5: 'to', 6: 'of', 7: 'he', 8: """", 9: 'a'}
```

El texto de la novela *Ana Karenina* tiene un total de 437.098 tokens. Hay 12.778 tokens únicos. El diccionario `word_to_int` asigna un índice a cada token único. Por ejemplo, a la coma (,) se le asigna un índice de 0 y al punto (.) se le asigna un índice de 1. El diccionario `int_to_word` traduce un índice de vuelta a un token. Por ejemplo, el índice 2 se traduce de vuelta al token " the ". El índice 4 se traduce de vuelta al token " and ", y así sucesivamente.

Finalmente, convertimos todo el texto en índices:

```
print(text[0:20])
wordidx=[word_to_int[w] for w in text]
print([word_to_int[w] for w in text[0:20]])
```

La salida es

```
['chapter', '1', 'happy', 'families', 'are', 'all',
 'alike', ';', 'every', 'unhappy', 'family', 'is',
 'unhappy', 'in', 'its', 'own', 'way', '.',
 'everything', 'was']
[208, 670, 283, 3024, 82, 31, 2461, 35, 202, 690,
 365, 38, 690, 10, 234, 147, 166, 1, 149, 12]
```

Convertimos todos los tokens del texto en sus índices correspondientes y los guardamos en una lista `wordidx`. La salida anterior muestra los primeros 20 tokens del texto, así como sus índices correspondientes. Por ejemplo, el primer token del texto es `chapter`, con un valor de índice de 208.

Ejercicio 8.3

Descubra el valor del índice del token `anna` en el diccionario `word_to_int`.

8.3.2 Creación de lotes de datos de entrenamiento

A continuación, creamos pares de (x, y) para el entrenamiento. Cada x es una secuencia con 100 índices. El número 100 no tiene nada de mágico, y se puede cambiar fácilmente a 90 o 110 y obtener resultados similares. Un número demasiado grande puede ralentizar el entrenamiento, mientras que un número demasiado pequeño puede provocar que el modelo no capture las dependencias de largo alcance. A continuación, deslizamos la ventana un token a la derecha y la usamos como el valor objetivo y . Desplazar la secuencia un token a la derecha y usarla como salida durante la generación de la secuencia es una técnica común en el entrenamiento de modelos de lenguaje, incluyendo los transformadores. El bloque de código de la siguiente lista crea los datos de entrenamiento.

Listado 8.2 Creación de datos de entrenamiento

```
import torch
seq_len=100
xys=[]
for n in range(0, len(wordidx)-seq_len-1):
    x = wordidx[n:n+seq_len]
    y = wordidx[n+1:n+seq_len+1]
    xys.append((torch.tensor(x), (torch.tensor(y))))
```

① Cada entrada contiene 100 índices.

② A partir del primer token en el texto, se desliza hacia la derecha uno a la vez

③ Define la entrada x

④ Desplaza la entrada x hacia la derecha un token y la utiliza como salida y

Al desplazar la secuencia un token a la derecha y usarla como salida, el modelo se entrena para predecir el siguiente token dados los tokens anteriores. Por ejemplo, si la secuencia de entrada es "how are you", la secuencia desplazada sería "are you today". Durante el entrenamiento, el modelo aprende a predecir 'are' tras ver 'how', 'you' tras ver 'are', y así sucesivamente. Esto le ayuda a comprender la distribución de probabilidad del siguiente token en una secuencia. Verá esta práctica repetidamente más adelante en este libro.

Crearemos lotes de datos para el entrenamiento, con 32 pares de (x, y) en cada lote:

```
from torch.utils.data import DataLoader

torch.manual_seed(42)
batch_size=32
loader = DataLoader(xys, batch_size=batch_size,
shuffle=True)
```

Ya tenemos el conjunto de datos de entrenamiento. A continuación, crearemos un modelo LSTM y lo entrenaremos con los datos que acabamos de

procesar.

8.4 Construcción y entrenamiento del modelo LSTM

En esta sección, comenzarás construyendo un modelo LSTM utilizando la capa LSTM integrada de PyTorch. Este modelo comienza con una capa de incrustación de palabras, que transforma cada índice en un vector denso de 128 dimensiones. Tus datos de entrenamiento pasarán por esta capa de incrustación antes de ser alimentados a la capa LSTM. Esta capa LSTM está diseñada para procesar elementos de una secuencia de forma secuencial. Después de la capa LSTM, los datos pasarán a una capa lineal, cuyo tamaño de salida se ajusta al tamaño de tu vocabulario. Las salidas generadas por el modelo LSTM son esencialmente logits, que sirven como entradas para que la función softmax calcule probabilidades.

Una vez creado el modelo LSTM, el siguiente paso consistirá en usar los datos de entrenamiento para entrenarlo. Esta fase de entrenamiento es crucial para perfeccionar la capacidad del modelo para comprender y generar patrones consistentes con los datos que se le han suministrado.

8.4.1 Construcción de un modelo LSTM

En el listado 8.3, definimos una clase `WordLSTM()` que servirá como modelo LSTM y se entrenará para generar texto al estilo de *Ana Karenina*. La clase se define como se muestra en el siguiente listado.

Listado 8.3 Definición de la clase `WordLSTM()`

```
from torch import nn
device="cuda" if torch.cuda.is_available() else "cpu"
class WordLSTM(nn.Module):
    def __init__(self, input_size=128, n_embed=128,
                 n_layers=3, drop_prob=0.2):
        super().__init__()
        self.input_size=input_size
        self.drop_prob = drop_prob
        self.n_layers = n_layers
        self.n_embed = n_embed
        vocab_size=len(word_to_int)
        self.embedding=nn.Embedding(vocab_size,n_embed)      ①
        self.lstm = nn.LSTM(input_size=self.input_size,
```

```

        hidden_size=self.n_embed,
        num_layers=self.n_layers,
        dropout=self.drop_prob,batch_first=True)      ②
    self.fc = nn.Linear(input_size, vocab_size)

    def forward(self, x, hc):
        embed=self.embedding(x)
        x, hc = self.lstm(embed, hc)                ③
        x = self.fc(x)
        return x, hc

    def init_hidden(self, n_seqs):
        weight = next(self.parameters()).data
        return (weight.new(self.n_layers,
                           n_seqs, self.n_embed).zero_(),
                weight.new(self.n_layers,
                           n_seqs, self.n_embed).zero_())

```

- ① Los datos de entrenamiento primero pasan por una capa de incrustación.
- ② Crea una capa LSTM con la clase LSTM() de PyTorch
- ③ En cada paso de tiempo, la capa LSTM utiliza el token anterior y el estado oculto para predecir el siguiente token y el siguiente estado oculto.
- ④ Inicia el estado oculto para el primer token en la secuencia de entrada

La clase `WordLSTM()` definida previamente tiene tres capas: la capa de incrustación de palabras, la capa LSTM y una capa lineal final. Establecemos el valor del argumento `n_layers` en 3, lo que significa que la capa LSTM apila tres LSTM para formar un LSTM apilado, donde los dos últimos LSTM toman como entrada la salida del LSTM anterior. El método `init_hidden()` rellena el estado oculto con ceros cuando el modelo utiliza el primer elemento de la secuencia para realizar predicciones. En cada paso de tiempo, la entrada es el token actual y el estado oculto anterior, mientras que la salida es el siguiente token y el siguiente estado oculto.

Cómo funciona la clase `torch.nn.Embedding()`

La clase `torch.nn.Embedding()` de PyTorch se utiliza para crear una capa de incrustación en una red neuronal. Una capa de incrustación es una tabla de búsqueda entrenable que asigna índices enteros a representaciones vectoriales densas y continuas (incrustaciones).

Cuando crea una instancia de `torch.nn.Embedding()`, debe especificar dos parámetros principales: `num_embeddings`, el tamaño del vocabulario (número total de tokens únicos) y `embedding_dim`, el tamaño de cada vector de incrustación (la dimensionalidad de las incrustaciones de salida).

Internamente, la clase crea una matriz (o tabla de consulta) de forma (`num_embeddings`, `embedding_dim`), donde cada fila corresponde al vector de incrustación de un índice específico. Inicialmente, estas incrustaciones se inicializan aleatoriamente, pero se aprenden y actualizan durante el entrenamiento mediante retropropagación.

Al pasar un tensor de índices a la capa de incrustación (durante el paso hacia adelante de la red), esta busca los vectores de incrustación correspondientes en la tabla de búsqueda y los devuelve. PyTorch proporciona más información sobre la clase en <https://mng.bz/n0Zd>.

Creamos una instancia de la clase `WordLSTM()` y la usamos como nuestro modelo LSTM, de la siguiente manera:

```
model=WordLSTM().to(device)
```

Al crear el modelo LSTM, los pesos se inicializan aleatoriamente. Al usar pares de (x, y) para entrenar el modelo, LSTM aprende a predecir el siguiente token basándose en todos los tokens anteriores de la secuencia mediante el ajuste de los parámetros del modelo. Como se ilustra en la figura 8.2, LSTM aprende a predecir el siguiente token y el siguiente estado oculto basándose en el token y el estado oculto actuales, que es un resumen de la información de todos los tokens anteriores.

Utilizamos el optimizador Adam con una tasa de aprendizaje de 0,0001. La función de pérdida es la pérdida de entropía cruzada, ya que se trata esencialmente de un problema de clasificación multicategoría: el modelo intenta predecir el siguiente token de un diccionario con 12 778 opciones:

```
lr=0.0001
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
loss_func = nn.CrossEntropyLoss()
```

Ahora que el modelo LSTM está construido, entrenaremos el modelo con los lotes de datos de entrenamiento que preparamos anteriormente.

8.4.2 Entrenamiento del modelo LSTM

Durante cada época de entrenamiento, revisamos todos los lotes de datos (x, y) del conjunto de entrenamiento. El modelo LSTM recibe la secuencia de entrada, x , y genera una secuencia de salida predicha, \hat{y} . Esta predicción se compara con la secuencia de salida real, y , para calcular la pérdida de entropía cruzada, ya que básicamente realizamos una clasificación multicategoría. A continuación, ajustamos los parámetros del modelo para reducir esta pérdida, como hicimos en el capítulo 2 al clasificar prendas de vestir.

Aunque podríamos dividir nuestros datos en conjuntos de entrenamiento y validación, y entrenar el modelo hasta que no se observen más mejoras en el conjunto de validación (como hicimos en el capítulo 2), nuestro objetivo principal es comprender el funcionamiento de los modelos LSTM, no necesariamente lograr el mejor ajuste de parámetros. Por lo tanto, entrenaremos el modelo durante 50 épocas.

Listado 8.4 Entrenamiento del modelo LSTM para generar texto

```
model.train()

for epoch in range(50):
    tloss=0
    sh,sc = model.init_hidden(batch_size)
    for i, (x,y) in enumerate(loader):
        if x.shape[0]==batch_size:
            inputs, targets = x.to(device), y.to(device)
            optimizer.zero_grad()
            output, (sh,sc) = model(inputs, (sh,sc))          ①
            loss = loss_func(output.transpose(1,2),targets)   ②
            sh,sc=sh.detach(),sc.detach()
            loss.backward()
```

```
nn.utils.clip_grad_norm_(model.parameters(), 5)
optimizer.step()
tloss+=loss.item()
if (i+1)%1000==0:
    print(f"at epoch {epoch} iteration {i+1}\\
average loss = {tloss/(i+1)}")
```

(4)

- ① Itera a través de todos los lotes de (x,y) en los datos de entrenamiento
- ② Utiliza el modelo para predecir la secuencia de salida
- ③ Compara las predicciones con el resultado real y calcula la pérdida
- ④ Ajusta los parámetros del modelo para minimizar la pérdida

En el código anterior, `sh` y `sc` juntos forman el estado oculto. En particular, el estado de la celda `sc` actúa como una cinta transportadora, transportando información a lo largo de múltiples intervalos de tiempo, con información añadida o eliminada en cada uno. El componente `sh` es la salida de la celda LSTM en un intervalo de tiempo determinado. Contiene información sobre la entrada actual y se utiliza para pasar información a la siguiente celda LSTM de la secuencia.

Si tienes una GPU compatible con CUDA, este entrenamiento dura unas 6 horas. Si solo usas CPU, puede tardar uno o dos días, dependiendo de tu hardware. También puedes descargar los pesos preentrenados de mi sitio web: <https://mng.bz/vJZa>.

A continuación, guardamos los pesos del modelo entrenado en la carpeta local:

```
import pickle

torch.save(model.state_dict(),"files/wordLSTM.pth")
with open("files/word_to_int.p","wb") as fb:
    pickle.dump(word_to_int, fb)
```

El diccionario `word_to_int` también se guarda en su computadora, lo que es un paso práctico que garantiza que pueda generar texto utilizando el modelo entrenado sin necesidad de repetir el proceso de tokenización.

8.5 Generación de texto con el modelo LSTM entrenado

Ahora que ya tienes un modelo LSTM entrenado, en esta sección aprenderás a usarlo para generar texto. El objetivo es comprobar si el modelo entrenado puede generar texto gramaticalmente correcto y coherente mediante la predicción iterativa del siguiente token con base en los tokens anteriores. También aprenderás a usar la temperatura y el muestreo top-K para controlar la creatividad del texto generado.

Al generar texto con el modelo LSTM entrenado, comenzamos con una solicitud como entrada inicial. Usamos el modelo entrenado para predecir el siguiente token más probable. Tras añadir el siguiente token a la solicitud, alimentamos el modelo entrenado con la nueva secuencia para predecirlo de nuevo. Repetimos este proceso hasta que la secuencia alcance una longitud determinada.

8.5.1 Generación de texto mediante la predicción del siguiente token

Primero, cargamos los pesos del modelo entrenado y el diccionario `word_to_int` desde la carpeta local:

```
model.load_state_dict(torch.load("files/wordLSTM.pth",
                                 map_location=device))
with open("files/word_to_int.p", "rb") as fb:
    word_to_int = pickle.load(fb)
int_to_word={v:k for k,v in word_to_int.items()}
```

El archivo también está disponible en el repositorio de GitHub del libro. Para crear el diccionario, `word_to_int.p` intercambiamos las posiciones de las claves y los valores en el diccionario `.word_to_int int_to_word`

Para generar texto con el modelo LSTM entrenado, necesitamos un mensaje inicial. Estableceremos el mensaje predeterminado como "Anna y el". Una forma sencilla de determinar cuándo detenerse es limitar el texto generado a una longitud determinada, por ejemplo, 200 tokens: una vez alcanzado el tamaño deseado, le pedimos al modelo que deje de generar.

La siguiente lista define una función `sample()` para generar texto según una solicitud.

Listado 8.5 Una función `sample()` para generar texto

```
import numpy as np
def sample(model, prompt, length=200):
    model.eval()
    text = prompt.lower().split(' ')
    hc = model.init_hidden(1)
    length = length - len(text) (1)
    for i in range(0, length):
        if len(text)<= seq_len:
            x = torch.tensor([[word_to_int[w] for w in text]])
        else:
            x = torch.tensor([[word_to_int[w] for w \
                in text[-seq_len:]]]) (2)
        inputs = x.to(device)
        output, hc = model(inputs, hc) (3)
        logits = output[0][-1]
        p = nn.functional.softmax(logits, dim=0).detach()\
            .cpu().numpy()
        idx = np.random.choice(len(logits), p=p) (4)
        text.append(int_to_word[idx]) (5)
        text=" ".join(text)
        for m in ",.;?!$()/_&%*@`":
            text=text.replace(f" {m}", f"{m} ")
        text=text.replace(' ', ' ')
        text=text.replace('` ', '`')
        text=text.replace('` ', '`')
        text=text.replace('` ', '`')
        return text
```

- ① Determina cuántos tokens deben generarse
- ② La entrada es la secuencia actual; la recorta si tiene más de 100 tokens
- ③ Hace una predicción utilizando el modelo entrenado
- ④ Selecciona el siguiente token según las probabilidades previstas
- ⑤ Agrega el siguiente token previsto a la secuencia y lo repite

La función `sample()` acepta tres argumentos. El primero es el modelo LSTM entrenado que se utilizará. El segundo es el indicador inicial para la generación de texto, que puede ser una frase de cualquier longitud, entre comillas. El tercer parámetro especifica la longitud del texto que se generará, medida en tokens, con un valor predeterminado de 200 tokens.

Dentro de la función, primero restamos el número de tokens en la solicitud de la longitud total deseada para determinar la cantidad de tokens que deben generarse. Al generar el siguiente token, consideramos la longitud de la secuencia actual. Si es menor de 100 tokens, introducimos la secuencia completa en el modelo; si es mayor de 100 tokens, solo se utilizan como entrada los últimos 100 tokens de la secuencia. Esta entrada se introduce en el modelo LSTM entrenado para predecir el token siguiente, que posteriormente añadimos a la secuencia actual. Continuamos este proceso hasta que la secuencia alcanza la longitud deseada.

Al generar el siguiente token, el modelo emplea el método `random.choice(len(logits), p = p)` de NumPy. El primer parámetro del método indica el rango de opciones, que en este caso es `len(logits) = 12778`. Esto significa que el modelo seleccionará aleatoriamente un número entero entre 0 y 12777, y cada número entero corresponde a un token diferente del vocabulario. El segundo parámetro, `p`, es una matriz de 12778 elementos, donde cada elemento representa la probabilidad de seleccionar un token correspondiente del vocabulario. Los tokens con mayor probabilidad en esta matriz tienen mayor probabilidad de ser seleccionados.

Generemos un pasaje con el modelo usando “Ana y el príncipe” como indicación (asegúrese de poner un espacio antes de los signos de puntuación cuando use su propia indicación):

```
torch.manual_seed(42)
np.random.seed(42)
print(sample(model, prompt='Anna and the prince'))
```

Aquí, fijé el número de semilla aleatoria a 42 tanto en PyTorch como en NumPy por si se desea reproducir los resultados. El pasaje generado es

```
anna and the prince did not forget what he had not spoken.
when the softening barrier was not so long as he had
```

talked to his brother, all the hopelessness of the impression. "official tail, a man who had tried him, though he had been able to get across his charge and locked close, and the light round the snow was in the light of the altar villa. the article in law levin was first more precious than it was to him so that if it was most easy as it would be as the same. this was now perfectly interested. when he had got up close out into the sledge, but it was locked in the light window with their one grass, and in the band of the leaves of his projects, and all the same stupid woman, and really, and i swung his arms round that thinking of bed. a little box with the two boys were with the point of a gleam of filling the boy, noiselessly signed the bottom of his mouth, and answering them took the red

Quizás hayas notado que el texto generado está completamente en minúsculas. Esto se debe a que, durante el procesamiento del texto, convertimos todas las mayúsculas a minúsculas para minimizar la cantidad de tokens únicos.

El texto generado tras 6 horas de entrenamiento es impresionante. La mayoría de las oraciones se ajustan a las normas gramaticales. Si bien no alcanza el nivel de sofisticación que se observa en textos generados por sistemas avanzados como ChatGPT, es un logro significativo. Con las habilidades adquiridas en este ejercicio, estará listo para entrenar modelos de generación de texto más avanzados en capítulos posteriores.

8.5.2 Muestreo de temperatura y top-K en la generación de texto

La creatividad del texto generado se puede controlar mediante técnicas como la temperatura y el muestreo top-K.

La temperatura ajusta la distribución de probabilidades asignada a cada token potencial antes de seleccionar el siguiente. Escala eficazmente los logits, que son las entradas de la función softmax que calcula estas probabilidades, según el valor de la temperatura. Los logits son las salidas del modelo LSTM antes de la aplicación de la función softmax.

En la función `sample()` que acabamos de definir, no ajustamos los logits, lo que implica una temperatura predeterminada de 1. Una temperatura más baja (inferior a 1; p. ej., 0,8) genera menos variaciones, lo que hace que el modelo sea más determinista y conservador, favoreciendo las opciones más probables. Por el contrario, una temperatura más alta (superior a 1; p. ej., 1,5) aumenta la probabilidad de elegir palabras improbables en la generación de texto, lo que genera resultados más variados e ingeniosos. Sin embargo, esto también podría hacer que el texto sea menos coherente o relevante, ya que el modelo podría optar por palabras menos probables.

El muestreo Top-K es otro método para influir en el resultado. Este enfoque implica seleccionar la siguiente palabra de las K opciones más probables, según lo predicho por el modelo. La distribución de probabilidad se trunca para incluir solo las K palabras más probables. Con un valor de K pequeño, como 5, las opciones del modelo se limitan a unas pocas palabras altamente probables, lo que resulta en resultados más predecibles y coherentes, pero potencialmente menos diversos e interesantes. En la función `sample()` definida anteriormente, no aplicamos el muestreo Top-K, por lo que el valor de K fue, en efecto, el tamaño del vocabulario (12.778 en nuestro caso).

A continuación, presentamos una nueva función, `generate()`, para la generación de texto. Esta función es similar a la función `sample()`, pero incluye dos parámetros adicionales: `temperature` y `top_k`, lo que permite un mayor control sobre la creatividad y la aleatoriedad del texto generado. La función `generate()` se define en la siguiente lista.

Listado 8.6 Generación de texto con muestreo de temperatura y top-K

```
def generate(model, prompt , top_k=None,
            length=200, temperature=1):
    model.eval()
    text = prompt.lower().split(' ')
    hc = model.init_hidden(1)
    length = length - len(text)
    for i in range(0, length):
        if len(text)<= seq_len:
            x = torch.tensor([[word_to_int[w] for w in text]])
        else:
            x = torch.tensor([[word_to_int[w] for w in
```

```

        text[-seq_len:]]])
inputs = x.to(device)
output, hc = model(inputs, hc)
logits = output[0][-1]
logits = logits/temperature
p = nn.functional.softmax(logits, dim=0).detach().cpu()
if top_k is None:
    idx = np.random.choice(len(logits), p=p.numpy())
else:
    ps, tops = p.topk(top_k) ①
    ps=ps/ps.sum()
    idx = np.random.choice(tops, p=ps.numpy()) ②
text.append(int_to_word[idx]) ③

text=".join(text)
for m in ",.;?!$()/_&%*@`":
    text=text.replace(f" {m}", f"{m} ")
text=text.replace(' ', ' ')
text=text.replace(' ', ' ')
text=text.replace(' ', ' ')
text=text.replace(' ', ' ')
return text

```

① Escala los logits con la temperatura

② Mantiene solo los K candidatos más probables

③ Selecciona el siguiente token de los K principales candidatos

En comparación con la función `sample()`, la nueva función `generate()` tiene dos argumentos opcionales adicionales: `top_k` y `temperature`. De forma predeterminada `top_k`, `None`, y `temperature` tiene el valor 1. Por lo tanto, si se llama a la función `generate()` sin especificar estos dos argumentos, el resultado será el mismo que el de la función `sample()`.

Ilustremos las variaciones en el texto generado centrándonos en la creación de un único token. Para ello, usaremos "No voy a ver" como mensaje (nótese el espacio antes del apóstrofo, como ya hicimos en este capítulo). Llamamos a la función `generate()` 10 veces, estableciendo su argumento `length` en uno más que la longitud del mensaje. Este enfoque garantiza que la función añada solo un token adicional al mensaje:

```
prompt="I ' m not going to see"
torch.manual_seed(42)
np.random.seed(42)
for _ in range(10):
    print(generate(model, prompt, top_k=None,
                   length=len(prompt.split(" "))+1, temperature=1))
```

La salida es

```
i'm not going to see you
i'm not going to see those
i'm not going to see me
i'm not going to see you
i'm not going to see her
i'm not going to see her
i'm not going to see the
i'm not going to see my
i'm not going to see you
i'm not going to see me
```

Con la configuración predeterminada de `top_k = None` y `temperatura = 1`, existe cierto grado de repetición en la salida. Por ejemplo, la palabra "tú" se repitió tres veces. Hay un total de seis tokens únicos.

Sin embargo, la funcionalidad de [se necesita contexto para "se necesita contexto"] `generate()` se amplía al ajustar estos dos argumentos. Por ejemplo, establecer una temperatura baja, como 0,5, y un valor pequeño `top_k`, como 3, genera un texto más predecible y menos creativo.

Repitamos el ejemplo del token único. Esta vez, fijamos la temperatura en 0,5 y el valor `top_k` en 3:

```
prompt="I ' m not going to see"
torch.manual_seed(42)
np.random.seed(42)
for _ in range(10):
    print(generate(model, prompt, top_k=3,
                   length=len(prompt.split(" "))+1, temperature=0.5))
```

La salida es

```
i'm not going to see you  
i'm not going to see the  
i'm not going to see her  
i'm not going to see you  
i'm not going to see her  
i'm not going to see you  
i'm not going to see her
```

El resultado tiene menos variaciones: solo hay 3 tokens únicos en 10 intentos: “tú”, “el” y “ella”.

Veamos esto en acción usando “Ana y el príncipe” como nuestro mensaje inicial cuando establecemos la temperatura en 0,5 y el valor `top_k` en 3:

```
torch.manual_seed(42)  
np.random.seed(42)  
print(generate(model, prompt='Anna and the prince',  
               top_k=3,  
               temperature=0.5))
```

La salida es

```
anna and the prince had no milk. but, "answered levin,  
and he stopped. "i've been skating to look at you all  
the harrows, and i'm glad. . . ""no, i'm going to  
the country. ""no, it's not a nice fellow. ""yes, sir.  
""well, what do you think about it? ""why, what's the  
matter? ""yes, yes, "answered levin, smiling, and he  
went into the hall. "yes, i'll come for him and go away,  
"he said, looking at the crumpled front of his shirt.  
"i have not come to see him, "she said, and she went  
out. "i'm very glad, "she said, with a slight bow to  
the ambassador's hand. "i'll go to the door. "she looked  
at her watch, and she did not know what to say
```

Ejercicio 8.4

Genere texto estableciendo la temperatura en 0.6 y `top_k` en 10, y usando "Ana y la enfermera" como mensaje inicial. Establezca el número de semilla aleatorio en 0 tanto en PyTorch como en NumPy.

Por el contrario, optar por un valor `temperature` más alto, como 1,5, junto con un valor `top_k` superior, por ejemplo `None` (que permite la selección de todo el conjunto de 12 778 tokens), genera resultados más creativos y menos predecibles. Esto se demuestra a continuación, en el ejemplo de un solo token. En esta ocasión, establecemos la temperatura en 2 y el valor `top_k` en `None`:

```
prompt="I ' m not going to see"
torch.manual_seed(42)
np.random.seed(42)
for _ in range(10):
    print(generate(model, prompt, top_k=None,
                    length=len(prompt.split(" "))+1, temperature=2))
```

La salida es

```
i'm not going to see them
i'm not going to see scarlatina
i'm not going to see behind
i'm not going to see us
i'm not going to see it
i'm not going to see it
i'm not going to see a
i'm not going to see misery
i'm not going to see another
i'm not going to see seryozha
```

El resultado casi no tiene repetición: hay 9 tokens únicos en 10 intentos; solo se repitió la palabra "it".

Usemos nuevamente "Ana y el príncipe" como mensaje inicial, pero establezcamos la temperatura en 2 y el valor `top_k` en Ninguno y veamos qué sucede:

```
torch.manual_seed(42)
np.random.seed(42)
```

```
print(generate(model, prompt='Anna and the prince',  
              top_k=None,  
              temperature=2))
```

El texto generado es

anna and the prince took sheaves covered suddenly people.
"pyotr marya borissovna, propped mihail though her son
will seen how much evening her husband; if tomorrow she
liked great time too. "adopted heavens details for it women
from this terrible, admitting this touching all everything
ill with flirtation shame consolation altogether: ivan only
all the circle with her honorable carriage in its house dress,
beethoven ashamed had the conversations raised mihailov stay
of close i taste work? "on new farming show ivan nothing.
hat yesterday if interested understand every hundred of two
with six thousand roubles according to women living over a
thousand: snetkov possibly try disagreeable schools with
stake old glory mysterious one have people some moral
conclusion, got down and then their wreath. darya alexandrovna
thought inwardly peaceful with varenka out of the listen from
and understand presented she was impossible anguish. simply
satisfied with staying after presence came where he pushed up
his hand as marya her pretty hands into their quarters.
waltz was about the rider gathered; sviazhsky further alone
have an hand paused riding towards an exquisite

El resultado generado no es repetitivo, aunque carece de coherencia en muchos lugares.

Ejercicio 8.5

Genere texto estableciendo la temperatura en 2 y `top_k` en 10000, y usando "Ana y la enfermera" como mensaje inicial. Establezca el número de semilla aleatorio en 0 tanto en PyTorch como en NumPy.

En este capítulo, ha adquirido habilidades fundamentales en PNL, incluyendo tokenización a nivel de palabra, incrustación de palabras y predicción de secuencias. Mediante estos ejercicios, ha aprendido a construir un modelo de lenguaje basado en tokenización a nivel de palabra y lo ha entrenado con LSTM para la generación de texto. En los siguientes capítulos,

le presentaremos el entrenamiento de Transformers, el tipo de modelos utilizados en sistemas como ChatGPT. Esto le proporcionará una comprensión más profunda de las técnicas avanzadas de generación de texto.

Resumen

- Las RNN son una forma especializada de red neuronal artificial diseñada para reconocer patrones en secuencias de datos, como texto, música o precios de acciones. A diferencia de las redes neuronales tradicionales, que procesan las entradas de forma independiente, las RNN tienen bucles que permiten la persistencia de la información. Las redes LSTM son versiones mejoradas de las RNN.
- Existen tres enfoques para la tokenización. El primero es la tokenización de caracteres, donde el texto se divide en sus caracteres constituyentes. El segundo es la tokenización de palabras, donde el texto se divide en palabras individuales. El tercer enfoque es la tokenización de subpalabras, que divide las palabras en componentes más pequeños y significativos llamados subpalabras.
- La incrustación de palabras es un método que transforma las palabras en representaciones vectoriales compactas, capturando su información semántica e interrelaciones. Esta técnica es vital en el procesamiento del lenguaje natural (PLN), especialmente porque las redes neuronales profundas, incluyendo modelos como LSTM y Transformers, requieren entrada numérica.
- La temperatura es un parámetro que influye en el comportamiento de los modelos de generación de texto. Controla la aleatoriedad de las predicciones escalando los logits (las entradas de la función softmax para el cálculo de probabilidad) antes de aplicar softmax. Una temperatura baja hace que el modelo sea más conservador en sus predicciones, pero también más repetitivo. A temperaturas más altas, el modelo se vuelve menos repetitivo y más innovador, lo que aumenta la diversidad del texto generado.

- El muestreo de Top-K es otra forma de influir en el comportamiento de los modelos de generación de texto. Consiste en seleccionar la siguiente palabra entre las K candidatas más probables, según lo determine el modelo. La distribución de probabilidad se trunca para conservar solo las K palabras principales. Valores bajos de K hacen que el resultado sea más predecible y coherente, pero potencialmente menos diverso e interesante.

¹ Sepp Hochreiter y Jurgen Schmidhuber, 1997, “Long Short-Term Memory”, *Neural Computation* 9(8): 1735-1780.

9. Una implementación línea por línea de atención y Transformador

Este capítulo cubre

- La arquitectura y funcionalidades de los codificadores y decodificadores en Transformers
- Cómo el mecanismo de atención utiliza la consulta, la clave y el valor para asignar pesos a los elementos de una secuencia
- Diferentes tipos de transformadores
- Construyendo un Transformer desde cero para la traducción de idiomas

Los transformadores son modelos avanzados de aprendizaje profundo que destacan en la gestión de desafíos de predicción secuencia a secuencia, superando a modelos más antiguos como las redes neuronales recurrentes (RNN) y las redes neuronales convolucionales (CNN). Su fortaleza reside en la comprensión eficaz de las relaciones entre los elementos de las secuencias de entrada y salida a grandes distancias, como dos palabras muy separadas en el texto. A diferencia de las RNN, los transformadores son capaces de entrenarse en paralelo, lo que reduce significativamente los tiempos de entrenamiento y permite gestionar grandes conjuntos de datos. Esta arquitectura transformadora ha sido fundamental en el desarrollo de grandes modelos de lenguaje (LLM) como ChatGPT, BERT y T5, marcando un hito significativo en el progreso de la IA.

Antes de la introducción de Transformers en el innovador artículo de 2017 "Attention Is All You Need" (La atención es todo lo que necesitas), elaborado por un grupo de investigadores de Google,¹ el procesamiento del lenguaje natural (PLN) y tareas similares dependían principalmente de las redes neuronales recesivas (RNN), incluyendo los modelos de memoria a largo plazo (MLPA). Sin embargo, las RNN procesan la información secuencialmente, lo que limita su velocidad debido a la imposibilidad de entrenarse en paralelo y a la dificultad para mantener la información sobre las partes anteriores de una secuencia, lo que impide capturar las dependencias a largo plazo.

El aspecto revolucionario de la arquitectura Transformer reside en su mecanismo de atención. Este mecanismo evalúa la relación entre las palabras de una secuencia mediante la asignación de pesos, determinando el grado de relación de significado entre las palabras según los datos de entrenamiento. Esto permite que modelos como ChatGPT comprendan las relaciones entre palabras, lo que mejora la comprensión del lenguaje humano. El procesamiento no secuencial de las entradas permite el entrenamiento en paralelo, lo que reduce el tiempo de entrenamiento y facilita el uso de grandes conjuntos de datos, impulsando así el auge de los LLM especializados y el auge actual de los avances en IA.

En este capítulo, implementaremos, línea por línea, la creación de un Transformador desde cero, basándonos en el artículo "Attention Is All You Need". Una vez entrenado, el Transformador puede gestionar traducciones entre dos idiomas (como del alemán al inglés o del inglés al chino). En el siguiente capítulo, nos centraremos en entrenar el Transformador desarrollado aquí para realizar traducciones del inglés al francés.

Para construir el Transformer desde cero, exploraremos el funcionamiento interno del mecanismo de autoatención, incluyendo las funciones de los vectores de consulta, clave y valor, y el cálculo de la atención de producto escalable (SDPA). Construiremos una capa de codificador integrando la normalización de capas y la conexión residual en una capa de atención multicabezal y combinándola con una capa de avance. A continuación, apilaremos seis de estas capas de codificador para formar el codificador. De igual forma, desarrollaremos un decodificador en el Transformer capaz de generar la traducción token por token, basándose en los tokens previos de la traducción y la salida del codificador.

Este trabajo preliminar te capacitará para entrenar el Transformer para traducciones entre dos idiomas cualesquiera. En el siguiente capítulo, aprenderás a entrenar el Transformer usando un conjunto de datos con más de 47,000 traducciones del inglés al francés. Verás cómo el modelo entrenado traduce frases comunes del inglés al francés con una precisión comparable a la del Traductor de Google.

9.1 Introducción a la atención y al Transformador

Para comprender el concepto de Transformers en el aprendizaje automático, es fundamental comprender primero el mecanismo de atención. Este mecanismo permite a los Transformers reconocer dependencias de largo alcance entre elementos de una secuencia, una característica que los distingue de modelos de predicción de secuencias anteriores, como las RNN. Con este mecanismo, los Transformers pueden centrarse simultáneamente en cada elemento de una secuencia, comprendiendo el contexto de cada palabra.

Considere la palabra "bank" para ilustrar cómo el mecanismo de atención interpreta las palabras según el contexto. En la oración "Ayer fui a pescar al río y estuve cerca de la orilla toda la tarde", la palabra "bank" se relaciona con "fishing" porque se refiere a la zona junto a un río. Aquí, un Transformer entiende "bank" como parte del terreno del río.

Por el contrario, en «Kate fue al banco después del trabajo ayer y depositó un cheque», «banco» se conecta con «cheque», lo que lleva al Transformer a identificar «banco» como una institución financiera. Este ejemplo muestra cómo los Transformers distinguen el significado de las palabras según el contexto.

En esta sección, profundizarás en el mecanismo de atención y explorarás su funcionamiento. Este proceso es crucial para determinar la importancia, o el peso, de varias palabras dentro de una oración. Después, examinaremos la estructura de diferentes modelos de Transformer, incluyendo uno que puede traducir entre dos idiomas.

9.1.1 El mecanismo de atención

El mecanismo de atención es un método utilizado para determinar las interconexiones entre los elementos de una secuencia. Calcula puntuaciones que indican cómo se relaciona un elemento con los demás en la secuencia; las puntuaciones más altas indican una relación más sólida. En PNL, este mecanismo es fundamental para conectar las palabras dentro de una oración de forma significativa. Este capítulo le guiará en la implementación del mecanismo de atención para la traducción de idiomas.

Para ello, construiremos un Transformador compuesto por un codificador y un decodificador. En el siguiente capítulo, entrenaremos al Transformador para traducir del inglés al francés. El codificador transforma una oración en inglés, como "¿Cómo estás?", en representaciones vectoriales que capturan su significado. El decodificador utiliza estas representaciones vectoriales para generar la traducción al francés.

Para transformar la frase "¿Cómo estás?" en representaciones vectoriales, el modelo primero la descompone en tokens [how, are, you, ?], un proceso similar al que ha realizado en el capítulo 8. Cada uno de estos tokens está representado por un vector de 256 dimensiones conocido como incrustaciones de palabras, que captura el significado de cada token. El codificador también emplea codificación posicional, un método para determinar las posiciones de los tokens en la secuencia. Esta codificación posicional se agrega a las incrustaciones de palabras para crear incrustaciones de entrada, que luego se utilizan para calcular la autoatención. La incrustación de entrada para "¿Cómo estás?" forma un tensor con dimensiones (4, 256), donde 4 representa el número de tokens y 256 es la dimensionalidad de cada incrustación.

Aunque existen diferentes maneras de calcular la atención, utilizaremos el método más común, SDPA. Este mecanismo también se denomina autoatención porque el algoritmo calcula cómo una palabra presta atención a todas las palabras de la secuencia, incluida la propia palabra. La Figura 9.1 muestra un diagrama que muestra cómo calcular SDPA.

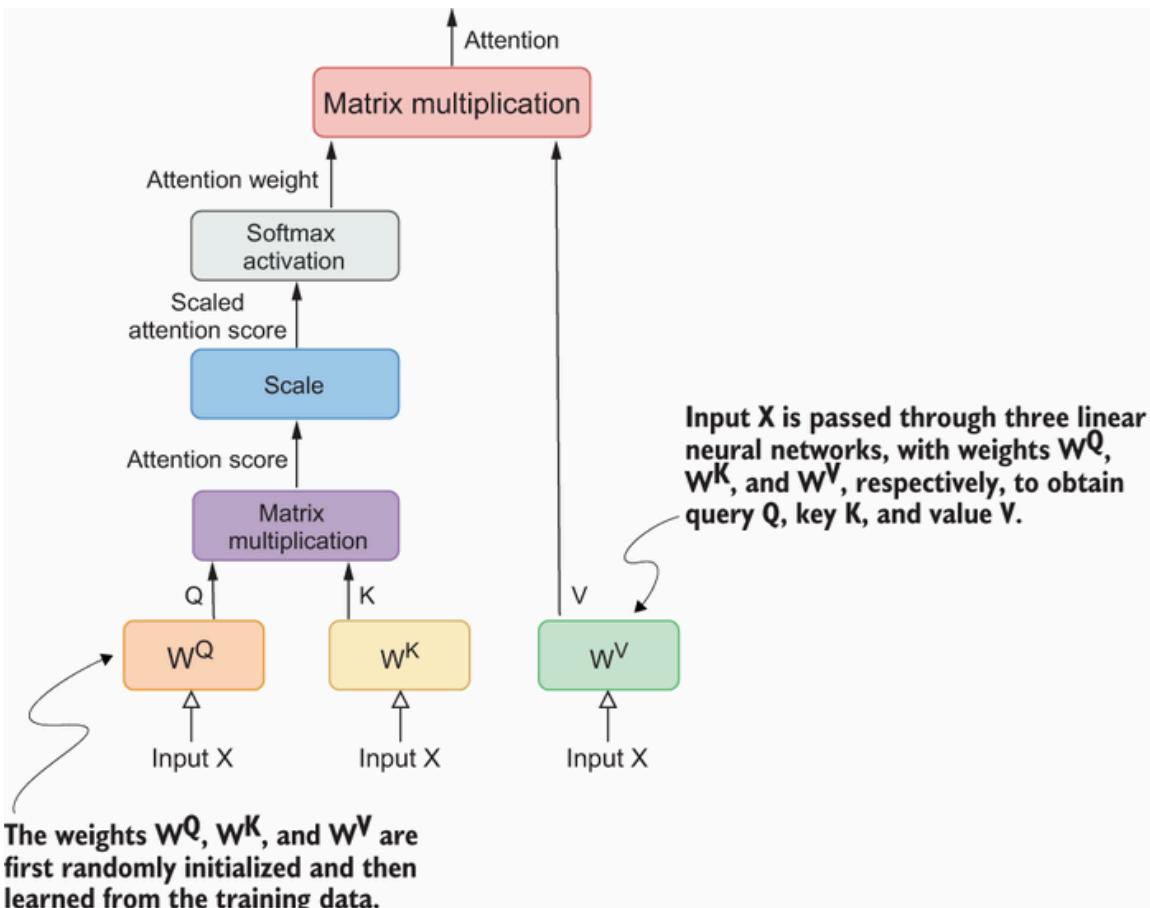


Figura 9.1 Diagrama del mecanismo de autoatención. Para calcular la atención, la incrustación de entrada X se pasa primero a través de tres capas neuronales con pesos W^Q , W^K y W^V , respectivamente. Las salidas son la consulta Q , la clave K y el valor V . La puntuación de atención escalada es el producto de Q y K dividido por la raíz cuadrada de la dimensión de K , d_k . Aplicamos la función softmax a la puntuación de atención escalada para obtener el peso de la atención. La atención es el producto del peso de la atención y el valor V .

El uso de la consulta, la clave y el valor para calcular la atención se inspira en los sistemas de recuperación. Considere visitar una biblioteca pública para buscar un libro. Si busca "aprendizaje automático en finanzas" en el motor de búsqueda de la biblioteca, esta frase se convierte en su consulta. Los títulos y las descripciones de los libros en la biblioteca sirven como claves. Según la similitud entre su consulta y estas claves, el sistema de recuperación de la biblioteca sugiere una lista de libros (valores). Los libros que contienen "aprendizaje automático", "finanzas" o ambos en sus títulos o descripciones probablemente tendrán una mejor clasificación. Por el contrario, los libros no relacionados con estos términos tendrán una puntuación de coincidencia más baja y, por lo tanto, es menos probable que se recomiendan.

Para calcular SDPA, la incrustación de entrada X se procesa a través de tres capas de red neuronal distintas. Los pesos correspondientes a estas capas son W^Q , W^K y W^V ; cada uno tiene una dimensión de 256×256 . Estos pesos se aprenden de los datos durante la fase de entrenamiento. Por lo tanto, po-

demos calcular la consulta Q, la clave K y el valor V como $Q = X * W^Q$, $K = X * W^K$ y $V = X * W^V$. Las dimensiones de Q, K y V coinciden con las de la incrustación de entrada X, que son 4×256 .

Similar al ejemplo del sistema de recuperación mencionado anteriormente, en el mecanismo de atención, evaluamos las similitudes entre los vectores de consulta y clave mediante el enfoque SDPA. SDPA implica calcular el producto escalar de los vectores de consulta (Q) y clave (K). Un producto escalar alto indica una gran similitud entre ambos vectores y viceversa. Por ejemplo, en la oración "¿Cómo estás?", la puntuación de atención escalada se calcula de la siguiente manera:

$$\text{AttentionScore}(Q, K) = \frac{Q * K^T}{\sqrt{d_k}} \quad (9.1)$$

Donde d_k representa la dimensión del vector clave K, que en nuestro caso es 256. Escalamos el producto escalar de Q y K por la raíz cuadrada de d_k para estabilizar el entrenamiento. Este escalado se realiza para evitar que el producto escalar crezca demasiado. El producto escalar entre los vectores de consulta y clave puede ser muy grande cuando la dimensión de estos vectores (es decir, la profundidad de la incrustación) es alta. Esto se debe a que cada elemento del vector de consulta se multiplica por cada elemento del vector clave, y estos productos se suman.

El siguiente paso es aplicar la función softmax a estas puntuaciones de atención, convirtiéndolas en ponderaciones de atención. Esto garantiza que la atención total que una palabra otorga a todas las palabras de la oración sume el 100 %.

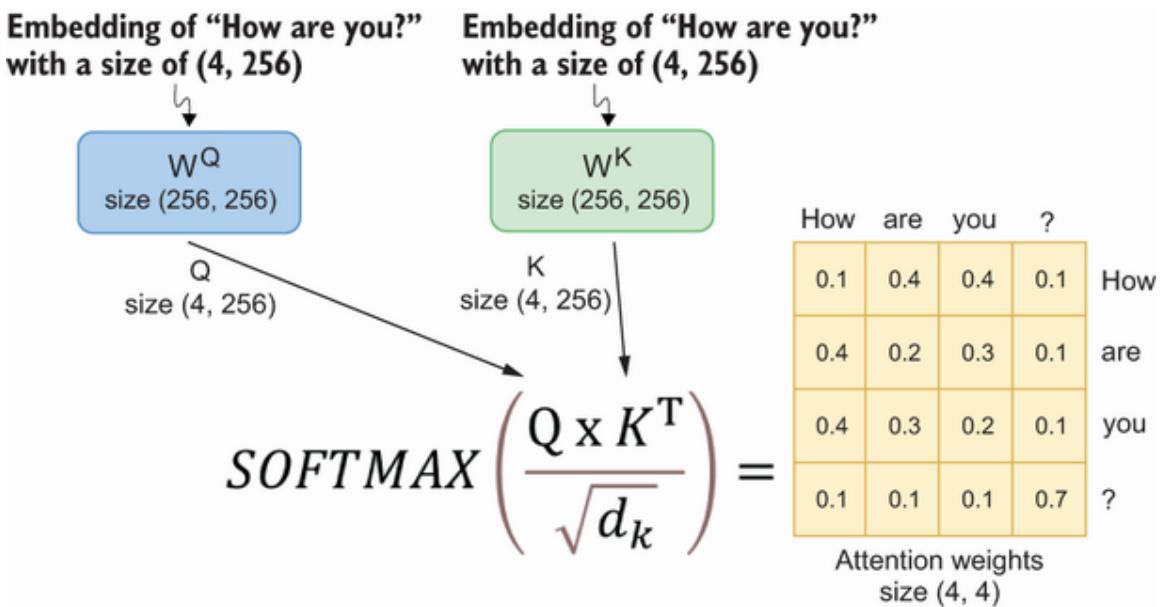


Figura 9.2 Pasos para calcular los pesos de atención. La incrustación de entrada se pasa a través de dos redes neuronales para obtener la consulta Q y la clave K. Las puntuaciones de atención escaladas se calculan como el producto escalar de Q y K dividido por la raíz cuadrada de la dimensión de K. Finalmente, aplicamos la función softmax a las puntuaciones de atención escaladas para obtener los pesos de atención, que muestran la relación entre cada elemento y los demás elementos de la secuencia.

La Figura 9.2 muestra cómo se hace esto. Para la oración "¿Cómo estás?", los pesos de atención forman una matriz de 4×4 , que muestra cómo cada token `["How", "are", "you", "?"]` se relaciona con todos los demás tokens (incluido él mismo). Los números en la Figura 9.2 son números inventados para ilustrar este punto. Por ejemplo, la primera fila de los pesos de atención muestra que el token `"How"` se dedica el 10 % de su atención a sí mismo y el 40 %, 40 % y 10 % a los otros tres tokens, respectivamente.

La atención final se calcula entonces como el producto escalar de estos pesos de atención y el vector de valores V (también ilustrado en la figura 9.3):

$$\text{Attention}(Q, K, V) = \text{softmax} \left(\frac{Q * K^T}{\sqrt{d_k}} \right) * V \quad (9.2)$$

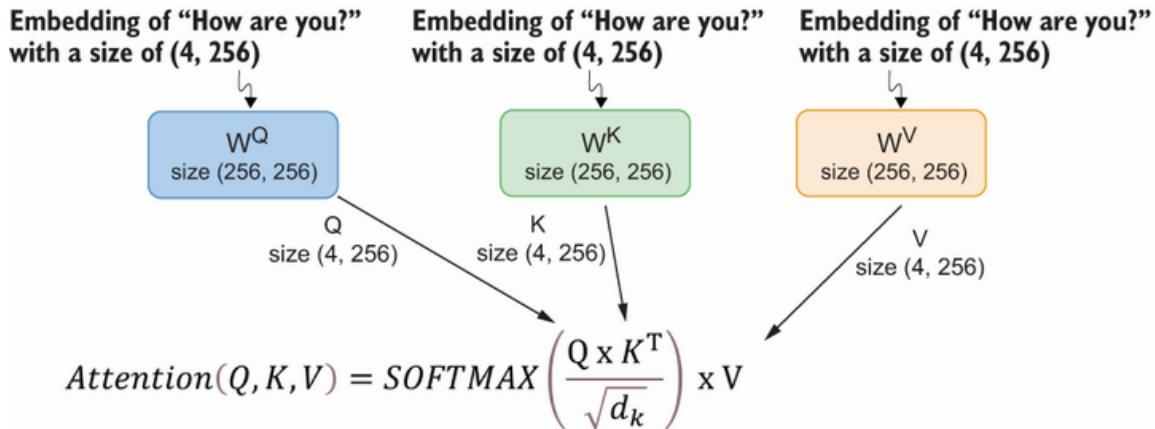


Figura 9.3 . Utilice los pesos de atención y el vector de valores para calcular el vector de atención. La incrustación de entrada se pasa a través de una red neuronal para obtener el valor V. La atención final es el producto escalar de los pesos de atención calculados anteriormente y el vector de valores V.

Esta salida también mantiene una dimensión de 4×256 , consistente con nuestras dimensiones de entrada.

Para resumir, el proceso comienza con la incrustación de entrada X de la oración "¿Cómo estás?", que tiene una dimensión de 4×256 . Esta incrustación captura los significados de los cuatro tokens individuales, pero carece de comprensión contextualizada. El mecanismo de atención termina con la salida `attention(Q, K, V)` , que mantiene la misma dimensión de 4×256 .

Esta salida puede verse como una combinación contextualmente Enriquecida de los cuatro tokens originales. La ponderación de los tokens originales varía en función de la relevancia contextual de cada token, otorgando más importancia a las palabras que son más importantes dentro del contexto de la oración. A través de este procedimiento, el mecanismo de atención transforma vectores que representan tokens aislados en vectores imbuidos de significados contextualizados, extrayendo así una comprensión más rica y matizada de la oración.

Además, en lugar de usar un conjunto de vectores de consulta, clave y valor, los modelos Transformer usan un concepto llamado atención multicaheada. Por ejemplo, los vectores de consulta, clave y valor de 256 dimensiones se pueden dividir en, digamos, 8, cabezas, y cada cabeza tiene un con-

junto de vectores de consulta, clave y valor con dimensiones de 32 (porque $256/8 = 32$). Cada cabeza presta atención a diferentes partes o aspectos de la entrada, lo que permite al modelo capturar una gama más amplia de información y formar una comprensión más detallada y contextual de los datos de entrada. La atención multicabeza es especialmente útil cuando una palabra tiene múltiples significados en una oración, como en un juego de palabras. Continuemos con el ejemplo de "banco" que mencionamos antes. Considere el chiste del juego de palabras, "¿Por qué es tan rico el río? Porque tiene dos orillas". En el proyecto de traducción del inglés al francés en el siguiente capítulo, implementará de primera mano la división de Q, K y V en múltiples cabezas para calcular la atención en cada cabeza antes de concatenarlas de nuevo en un solo vector de atención.

9.1.2 La arquitectura del Transformer

El concepto del mecanismo de atención fue introducido por Bahdanau, Cho y Bengio en 2014.² Su uso se generalizó tras el innovador artículo "Attention Is All You Need", que se centró en la creación de un modelo para la traducción automática de lenguajes. La arquitectura de este modelo, conocido como Transformer, se muestra en la figura 9.4. Presenta una estructura de codificador-decodificador que se basa en gran medida en el mecanismo de atención. En este capítulo, construirás este modelo desde cero, codificándolo línea por línea, con la intención de entrenarlo para la traducción entre dos idiomas cualesquiera.

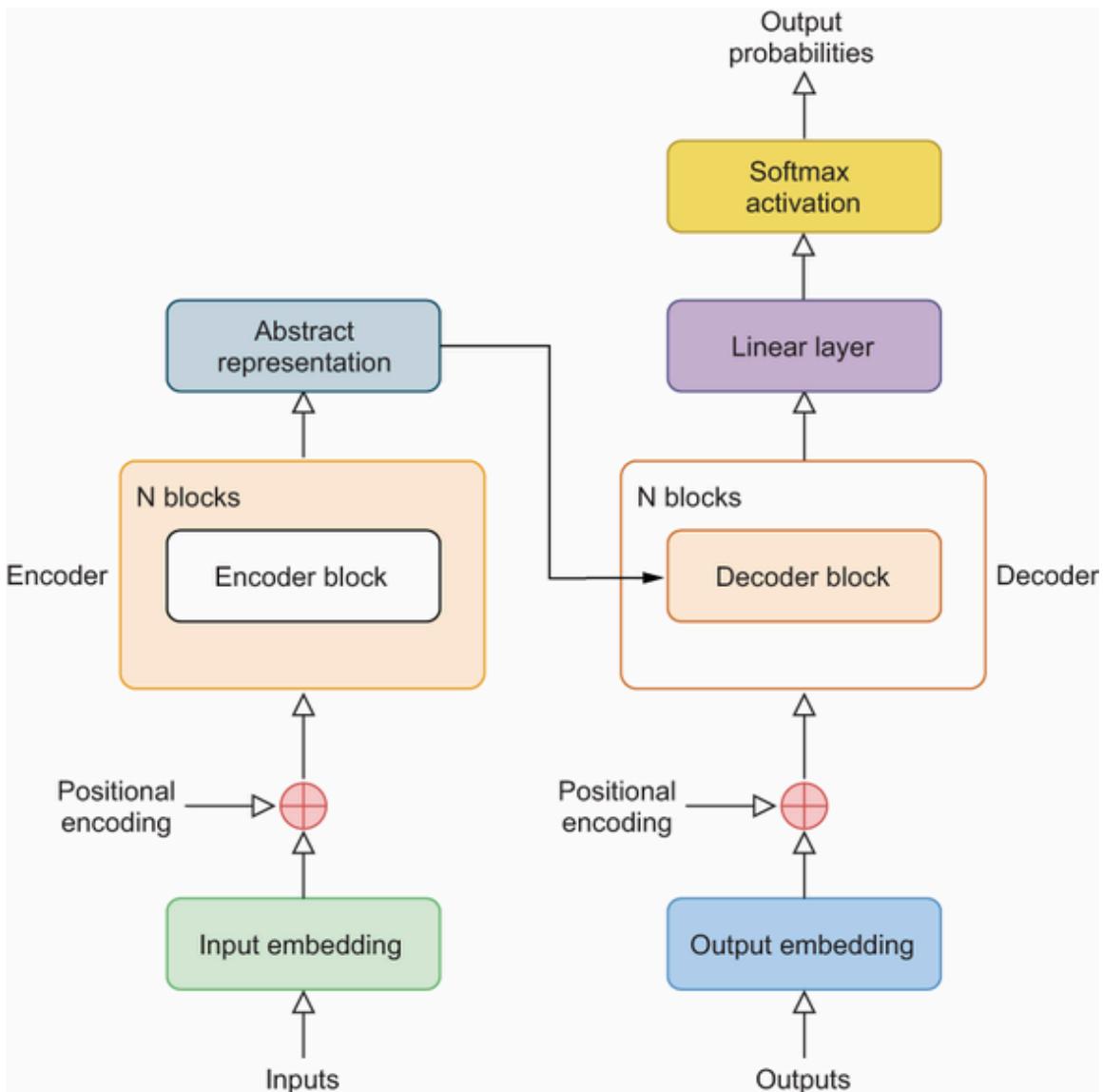


Figura 9.4 Arquitectura del Transformador. El codificador del Transformador (lado izquierdo del diagrama), compuesto por N capas idénticas, aprende el significado de la secuencia de entrada y lo convierte en vectores que lo representan. Posteriormente, pasa estos vectores al decodificador (lado derecho del diagrama), compuesto por N capas idénticas. El decodificador construye la salida (p. ej., la traducción al francés de una frase en inglés) prediciendo un token a la vez, basándose en tokens previos de la secuencia y las representaciones vectoriales del codificador. El generador, en la esquina superior derecha, es el cabezal conectado a la salida del decodificador, de modo que la salida es la distribución de probabilidad de todos los tokens del idioma de destino (p. ej., el vocabulario francés).

Tomemos como ejemplo la traducción del inglés al francés. El codificador del Transformer transforma una frase en inglés como "No hablo francés" en representaciones vectoriales que almacenan su significado. El decodificador del Transformer las procesa para producir la traducción al francés "Je ne parle pas français". La función del codificador es capturar la esencia de la frase original en inglés. Por ejemplo, si el codificador es eficaz, debería traducir "No hablo francés" y "No hablo francés" en representaciones vectoriales similares. En consecuencia, el decodificador interpretará estos vectores y generará traducciones similares. Curiosamente, al usar ChatGPT, estas dos frases en inglés resultan en la misma traducción al francés.

El codificador del Transformer aborda la tarea tokenizando primero las oraciones en inglés y francés. Este proceso es similar al descrito en el capítulo 8, pero con una diferencia clave: emplea tokenización de subpalabras. La tokenización de subpalabras es una técnica utilizada en PNL para dividir las palabras en componentes más pequeños, o subpalabras, lo que permite un procesamiento más eficiente y preciso. Por ejemplo, como verá en el siguiente capítulo, la frase en inglés «I do not speak French» se divide en seis tokens: (i, do, not, speak, fr, ench). De igual manera, su contraparte en francés «Je ne parle pas français» se tokeniza en seis partes: (je, ne, parle, pas, franc, ais). Este método de tokenización mejora la capacidad del Transformer para gestionar las variaciones y complejidades del lenguaje.

Los modelos de aprendizaje profundo, incluyendo los Transformers, no pueden procesar texto directamente, por lo que los tokens se indexan con enteros antes de introducirse en el modelo. Estos tokens suelen representarse primero mediante codificación one-hot, como se explicó en el capítulo 8. Después, los pasamos por una capa de incrustación de palabras para comprimirlos en vectores con valores continuos de un tamaño mucho menor, como una longitud de 256. Por lo tanto, tras aplicar la incrustación de palabras, la oración "No hablo francés" se representa mediante una matriz de 6×256 .

Los transformadores procesan datos de entrada, como oraciones, en paralelo, a diferencia de las RNN, que manejan los datos secuencialmente. Este paralelismo mejora su eficiencia, pero no les permite reconocer inherentemente el orden de la secuencia de entrada. Para solucionar esto, los transformadores añaden codificaciones posicionales a las incrustaciones de entrada. Estas codificaciones posicionales son vectores únicos asignados a cada posición en la secuencia de entrada y se alinean en dimensión con las incrustaciones de entrada. Los valores de los vectores están determinados por una función posicional específica, que involucra en particular funciones seno y coseno de frecuencias variables, definidas como

$$\text{PositionalEncoding}(\text{pos}, 2i) = \sin\left(\frac{\text{pos}}{n^{2i/d}}\right)$$

$$\text{PositionalEncoding}(\text{pos}, 2i + 1) = \cos\left(\frac{\text{pos}}{n^{2i/d}}\right) \quad (9.3)$$

En estas ecuaciones, los vectores se calculan utilizando la función seno para índices pares y la función coseno para índices impares. Los parámetros pos e i representan la posición de un token dentro de la secuencia y el índice dentro del vector, respectivamente. A modo de ejemplo, considere la codificación posicional de la frase "No hablo francés". Esta se representa como una matriz de 6×256 , del mismo tamaño que la incrustación de palabras de la oración. Aquí, pos varía de 0 a 5, y los índices $2i$ y $2i + 1$ abarcan colectivamente 256 valores distintos (de 0 a 255). Una ventaja de este enfoque de codificación posicional es que todos los valores están restringidos en el rango de -1 a 1 .

Es importante tener en cuenta que cada posición de token se identifica de forma única mediante un vector de 256 dimensiones, cuyos valores se mantienen constantes durante el entrenamiento. Antes de introducirse en las capas de atención, estas codificaciones posicionales se añaden a las incrustaciones de palabras de la secuencia. En el ejemplo de la oración "No hablo francés", el codificador genera tanto la incrustación de palabras como la codificación posicional, cada una con dimensiones de 6×256 , antes de combinarlas en una única representación de 6×256 dimensiones. Posteriormente, el codificador aplica el mecanismo de atención para refinar esta incrustación en representaciones vectoriales más sofisticadas que capturan el significado general de la frase, antes de pasarlas al decodificador.

El codificador del Transformer, como se muestra en la figura 9.5, está compuesto por seis capas idénticas ($N = 6$). Cada una de estas capas comprende dos subcapas distintas. La primera subcapa es una capa de autoatención multicabezal, similar a lo que se discutió anteriormente. La segunda subcapa es una red de alimentación hacia adelante básica, completamente conectada por posición. Esta red trata cada posición en la secuencia de forma

independiente en lugar de como elementos secuenciales. En la arquitectura del modelo, cada subcapa incorpora normalización de capa y una conexión residual. La normalización de capa normaliza las observaciones para que tengan media cero y desviación estándar unitaria. Dicha normalización ayuda a estabilizar el proceso de entrenamiento. Después de la capa de normalización, realizamos la conexión residual. Esto significa que la entrada a cada subcapa se agrega a su salida, mejorando el flujo de información a través de la red.

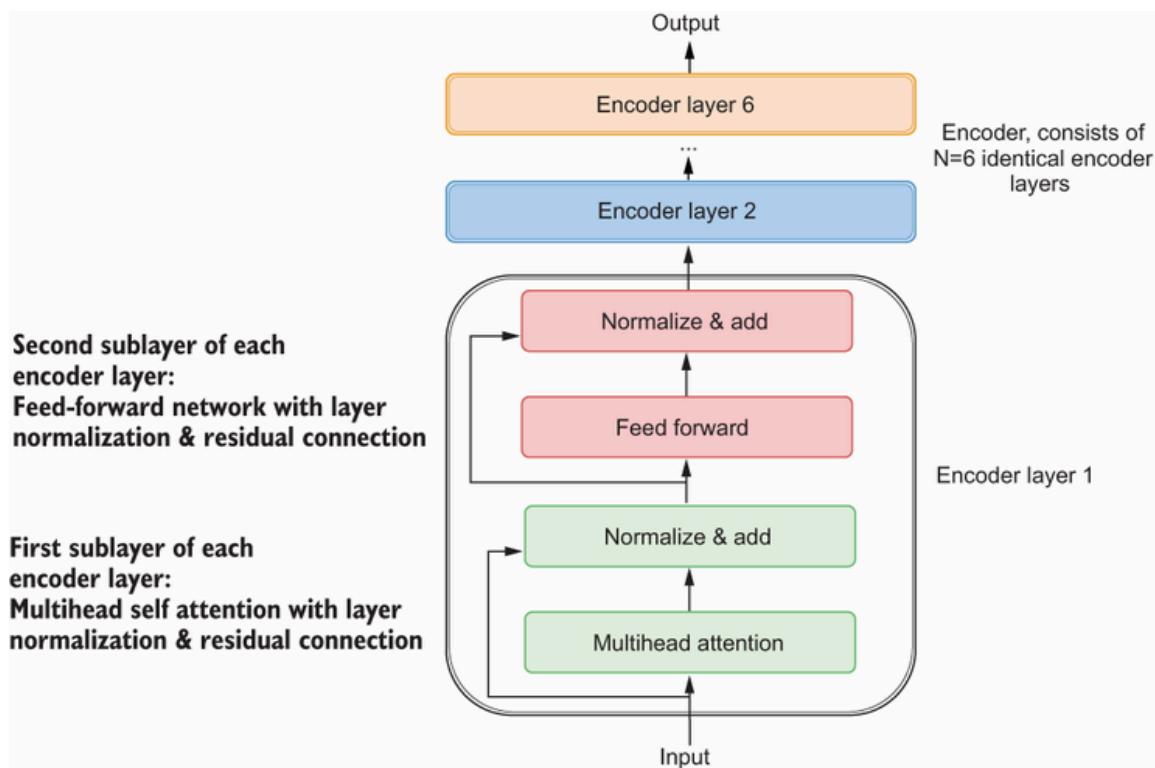


Figura 9.5 Estructura del codificador en el Transformador. El codificador consta de $N = 6$ capas idénticas. Cada capa contiene dos subcapas. La primera es una capa de autoatención multicabezal y la segunda es una red de retroalimentación. Cada subcapa utiliza normalización de capas y conexión residual.

El decodificador del modelo Transformer, como se muestra en la figura 9.6, consta de seis capas idénticas ($N = 6$). Cada una de estas capas cuenta con tres subcapas: una subcapa de autoatención multicabezal, una subcapa que realiza la atención cruzada multicabezal entre la salida de la primera subcapa y la salida del codificador, y una subcapa de avance. Cabe destacar que la entrada de cada subcapa es la salida de la subcapa anterior. Además, la segunda subcapa de la capa del decodificador también recibe la salida del codificador como entrada. Este diseño es crucial para integrar la información del codificador: así es como el decodificador genera traducciones basadas en la salida del codificador.

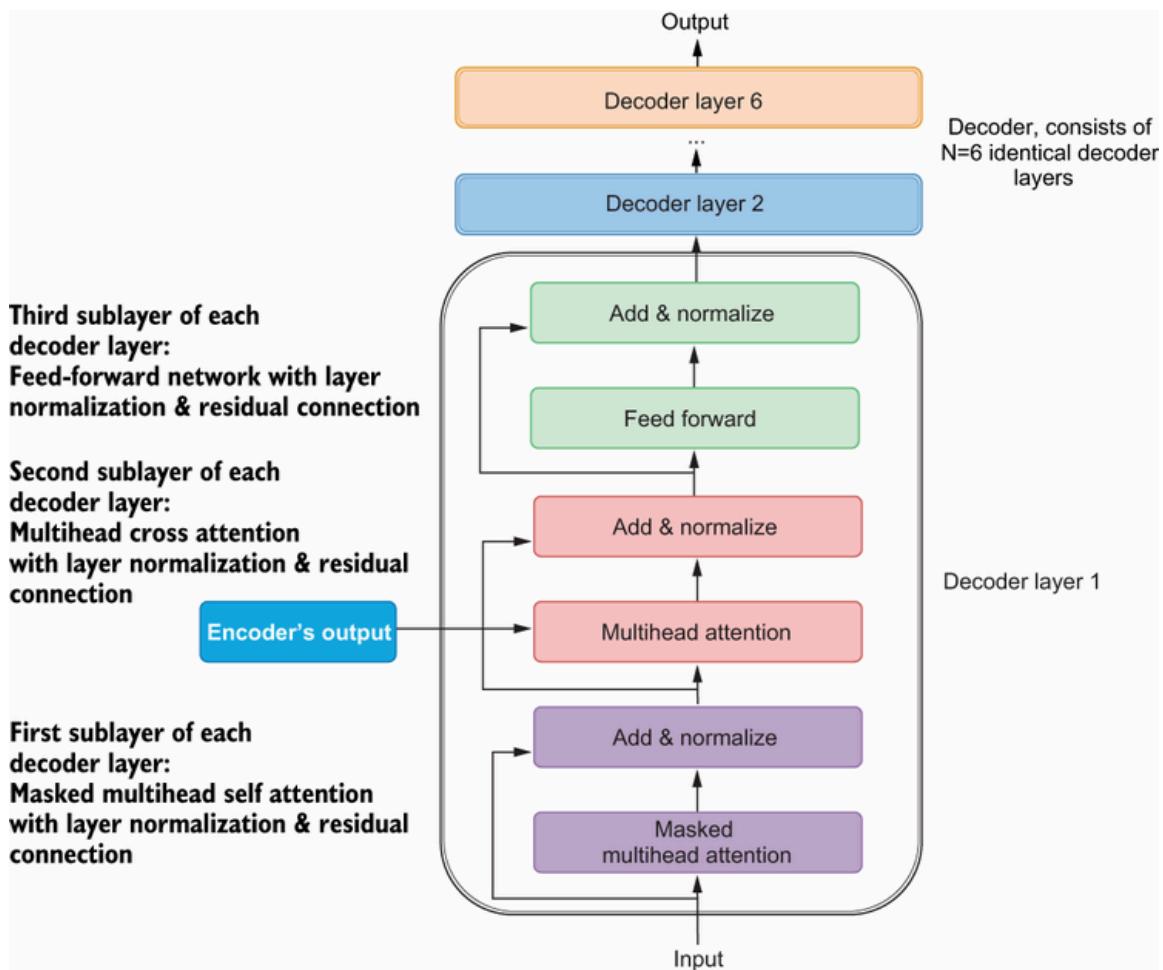


Figura 9.6 Estructura del decodificador en el Transformador. El decodificador consta de $N = 6$ capas idénticas. Cada capa contiene tres subcapas. La primera es una capa de autoatención multicabezal enmascarada. La segunda es una capa de atención cruzada multicabezal para calcular la atención cruzada entre la salida de la primera subcapa y la salida del codificador. La tercera subcapa es una red de avance. Cada subcapa utiliza normalización de capas y conexión residual.

Un aspecto clave de la subcapa de autoatención del decodificador es el mecanismo de enmascaramiento. Este enmascaramiento impide que el modelo acceda a posiciones futuras en la secuencia, garantizando que las predicciones para una posición específica solo dependan de elementos previamente conocidos. Esta dependencia secuencial es vital para tareas como la traducción de idiomas o la generación de texto.

El proceso de decodificación comienza cuando el decodificador recibe una frase de entrada en francés. El decodificador transforma los tokens franceses en incrustaciones de palabras y codificaciones posicionales antes de combinarlas en una única incrustación. Este paso garantiza que el modelo no solo comprenda el contenido semántico de la frase, sino que también mantenga el contexto secuencial, crucial para la precisión de las tareas de traducción o generación.

El decodificador funciona de forma autorregresiva, generando la secuencia de salida un token a la vez. En el primer paso de tiempo, comienza con el token "BOS", que indica el comienzo de una oración. Utilizando este token de inicio como entrada inicial, el decodificador examina representaciones vectoriales de la frase en inglés "I do not speak French" e intenta predecir el primer token después de "BOS". Supongamos que la primera predicción del decodificador es "Je". En el siguiente paso de tiempo, utiliza la secuencia "BOS Je" como nueva entrada para predecir el siguiente token. Este proceso continúa iterativamente, y el decodificador añade cada nuevo token predicho a su secuencia de entrada para la predicción posterior.

El proceso de traducción está diseñado para concluir cuando el decodificador predice el token "EOS", lo que indica el final de la oración. Al preparar los datos de entrenamiento, añadimos EOS al final de cada frase, de modo que el modelo haya aprendido que significa el final de una oración. Al alcanzar este token, el decodificador reconoce la finalización de la tarea de traducción y detiene su operación. Este enfoque autorregresivo garantiza que cada paso del proceso de decodificación se base en todos los tokens predichos previamente, lo que permite traducciones coherentes y contextualmente adecuadas.

9.1.3 Diferentes tipos de transformadores

Existen tres tipos de transformadores: transformadores solo codificadores, transformadores solo decodificadores y transformadores codificadores-decodificadores. En este capítulo y en el siguiente, utilizaremos un transformador codificador-decodificador, pero más adelante en el libro, tendrás la oportunidad de explorar directamente los transformadores solo decodificadores.

Un transformador solo codificador consta de N capas de codificador idénticas, como se muestra en el lado izquierdo de la figura 9.4, y es capaz de convertir una secuencia en representaciones vectoriales abstractas y continuas. Por ejemplo, BERT es un transformador solo codificador que contiene 12 capas. Un transformador solo codificador puede utilizarse para la clasificación de texto, por ejemplo. Si dos oraciones tienen representaciones vectoriales similares, podemos clasificarlas en una categoría. Por otro lado, si dos secuencias tienen representaciones vectoriales muy diferentes, podemos clasificarlas en categorías diferentes.

Un Transformer solo decodificador también consta de N capas idénticas, y cada capa es una capa decodificadora, como se muestra a la derecha de la figura 9.4. Por ejemplo, ChatGPT es un Transformer solo decodificador que contiene varias capas decodificadoras. Este Transformer puede generar texto basándose en una instrucción, por ejemplo. Extrae el significado semántico de las palabras de la instrucción y predice el siguiente token más probable. Luego, añade el token al final de la instrucción y repite el proceso hasta que el texto alcanza una longitud determinada.

El transformador de traducción de lenguaje máquina que analizamos anteriormente es un ejemplo de transformador codificador-decodificador. Son necesarios para gestionar tareas complejas, como la generación de texto a imagen o el reconocimiento de voz. Los transformadores codificador-decodificador combinan las ventajas de ambos. Los codificadores son eficientes en el procesamiento y la comprensión de los datos de entrada, mientras que los decodificadores destacan en la generación de salidas. Esta combinación permite al modelo comprender eficazmente entradas complejas (como texto o voz) y generar salidas complejas (como imágenes o texto transcritos).

9.2 Construcción de un codificador

Desarrollaremos y entrenaremos un transformador codificador-decodificador diseñado para la traducción de lenguajes de máquina. El código de este proyecto se adapta del trabajo de Chris Cui en la traducción del chino al inglés (<https://mng.bz/9o1o>) y del proyecto de traducción del alemán al inglés de Alexander Rush (<https://mng.bz/j0mp>).

Esta sección explica cómo construir un codificador en el Transformador. En concreto, profundizaremos en el proceso de construcción de varias subcapas dentro de cada capa del codificador y en la implementación del mecanismo de autoatención multicabezal.

9.2.1 El mecanismo de atención

Si bien existen diferentes mecanismos de atención, utilizaremos el SDPA por su amplio uso y eficacia. Este mecanismo utiliza consultas, claves y valores para calcular las relaciones entre los elementos de una secuencia.

Asigna puntuaciones para mostrar la relación de un elemento con todos los elementos de la secuencia (incluido el propio elemento).

En lugar de usar un conjunto de vectores de consulta, clave y valor, el modelo Transformer utiliza un concepto llamado atención multicabezal. Nuestros vectores de consulta, clave y valor de 256 dimensiones se dividen en 8 cabezas, y cada cabeza tiene un conjunto de vectores de consulta, clave y valor con dimensiones de 32 (ya que $256/8 = 32$). Cada cabeza presta atención a diferentes partes o aspectos de la entrada, lo que permite al modelo capturar una gama más amplia de información y formar una comprensión más detallada y contextualizada de los datos de entrada. Por ejemplo, la atención multicabezal permite al modelo capturar los múltiples significados de la palabra "banco" en el chiste "¿Por qué es tan rico el río? Porque tiene dos orillas".

Para implementar esto, definimos una función `attention()` en el módulo local `ch09util`. Descarga el archivo `ch09util.py` del repositorio de GitHub del libro (<https://github.com/markhliu/DGAI>) y guárdalo en el directorio `/utils/` de tu computadora. La función `attention()` se define como se muestra en la siguiente lista.

Listado 9.1 Cálculo de la atención según la consulta, la clave y el valor

```
def attention(query, key, value, mask=None, dropout=None):
    d_k = query.size(-1)
    scores = torch.matmul(query,
                          key.transpose(-2, -1)) / math.sqrt(d_k)          ①
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)           ②
    p_attn = nn.functional.softmax(scores, dim=-1)            ③
    if dropout is not None:
        p_attn = dropout(p_attn)
    return torch.matmul(p_attn, value), p_attn                ④
```

① La puntuación de atención escalada es el producto escalar de la consulta y la clave, escalado por la raíz cuadrada de d_k .

② Si hay una máscara, oculta elementos futuros en la secuencia

③ Calcula los pesos de atención

④ Devuelve tanto la atención como los pesos de atención

La función `attention()` toma la consulta, la clave y el valor como entradas y calcula la atención y sus ponderaciones, como se explicó anteriormente en este capítulo. La puntuación de atención escalada es el producto escalar de la consulta y la clave, escalada por la raíz cuadrada de la dimensión de la clave, d_k . Aplicamos la función softmax a la puntuación de atención escalada para obtener las ponderaciones. Finalmente, la atención se calcula como el producto escalar de las ponderaciones de atención y el valor.

Usemos nuestro ejemplo en ejecución para mostrar cómo funciona la atención multicabezal (ver figura 9.7). La incrustación para "¿Cómo estás?" es un tensor con un tamaño de (1, 6, 256), como explicamos en la última sección (después de agregar codificación posicional a la incrustación de palabras). Tenga en cuenta que 1 significa que hay una oración en el lote y hay seis tokens en la oración en lugar de cuatro porque agregamos BOS y EOS al principio y al final de la secuencia. Esta incrustación se pasa a través de tres capas lineales para obtener la consulta Q, la clave K y el valor V, cada uno del mismo tamaño (1, 6, 256). Estos se dividen en ocho cabezas, lo que resulta en ocho conjuntos distintos de Q, K y V, ahora con un tamaño de (1, 6, 256/8 = 32) cada uno. La función de atención, como se definió anteriormente, se aplica a cada uno de estos conjuntos, lo que produce ocho salidas de atención, cada una también con un tamaño de (1, 6, 32). Luego concatenamos las ocho salidas de atención en una sola, y el resultado es un tensor con un tamaño de (1, 6, 32 × 8 = 256). Finalmente, esta atención combinada pasa por otra capa lineal de 256 × 256, lo que genera la salida de la clase `MultiHeadAttention()`. Esta salida mantiene las dimensiones de la entrada original, que son (1, 6, 256).

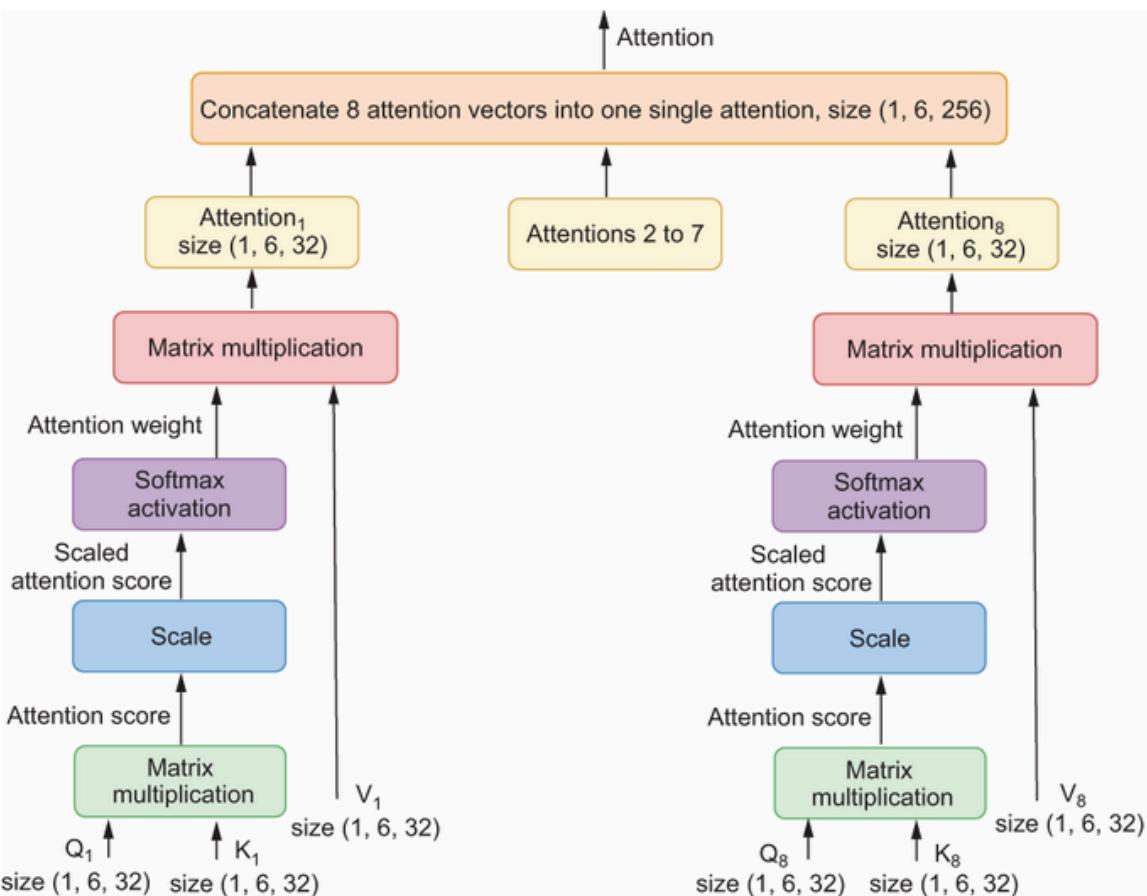


Figura 9.7 Ejemplo de atención multicabezal. Este diagrama utiliza el cálculo de la autoatención multicabezal para la frase "¿Cómo estás?" como ejemplo. Primero, pasamos la incrustación a través de tres redes neuronales para obtener la consulta Q , la clave K y el valor V , cada uno con un tamaño de $(1, 6, 256)$. Las dividimos en ocho cabezas, cada una con un conjunto de Q , k y V , con un tamaño de $(1, 6, 32)$. Calculamos la atención en cada cabeza. Los vectores de atención de las ocho cabezas se unen nuevamente en un solo vector de atención, con un tamaño de $(1, 6, 256)$.

Esto se implementa en la siguiente lista de código en el módulo local.

Listado 9.2 Cálculo de la atención multicabezal

```
from copy import deepcopy
class MultiHeadedAttention(nn.Module):
    def __init__(self, h, d_model, dropout=0.1):
        super().__init__()
        assert d_model % h == 0
        self.d_k = d_model // h
        self.h = h
        self.linears = nn.ModuleList([deepcopy(
            nn.Linear(d_model, d_model)) for i in range(4)])
        self.attn = None
        self.dropout = nn.Dropout(p=dropout)

    def forward(self, query, key, value, mask=None):
        if mask is not None:
            mask = mask.unsqueeze(1)
```

```

nbatches = query.size(0)
query, key, value = [l(x).view(nbatches, -1, self.h,
    self.d_k).transpose(1, 2)
    for l, x in zip(self.linears, (query, key, value))]
x, self.attn = attention(
    query, key, value, mask=mask, dropout=self.dropout) ②
x = x.transpose(1, 2).contiguous().view(
    nbatches, -1, self.h * self.d_k) ③
output = self.linears[-1](x)
return output ④

```

① Pasa la entrada a través de tres capas lineales para obtener Q, K, V y las divide en múltiples cabezales

② Calcula la atención y los pesos de atención para cada cabeza

③ Concatena vectores de atención de múltiples cabezas en un único vector de atención

④ Pasa la salida a través de una capa lineal

Cada capa de codificador y decodificador también contiene una subcapa de propagación hacia adelante, que consiste en una red neuronal completamente conectada de dos capas, cuyo objetivo es mejorar la capacidad del modelo para capturar y aprender características complejas en el conjunto de datos de entrenamiento. Además, la red neuronal procesa cada incrustación de forma independiente. No trata la secuencia de incrustaciones como un único vector. Por lo tanto, a menudo la denominamos red de propagación hacia adelante de toda la posición (o red convolucional unidimensional). Para ello, definimos una clase `PositionwiseFeedForward()` en el módulo local de la siguiente manera:

```

class PositionwiseFeedForward(nn.Module):
    def __init__(self, d_model, d_ff, dropout=0.1):
        super().__init__()
        self.w_1 = nn.Linear(d_model, d_ff)
        self.w_2 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)
    def forward(self, x):
        h1 = self.w_1(x)

```

```
    h2 = self.dropout(h1)
    return self.w_2(h2)
```

La clase `PositionwiseFeedForward()` se define con dos parámetros clave: `d_ff`, la dimensionalidad de la capa de avance, y `d_model`, que representa el tamaño de la dimensión del modelo. Normalmente, `d_ff` se elige un tamaño cuatro veces mayor que `d_model`. En nuestro ejemplo, `d_model` es 256, por lo que se establece `d_ff` en $256 * 4 = 1024$. Esta práctica de ampliar la capa oculta en comparación con el tamaño del modelo es un enfoque estándar en las arquitecturas Transformer. Mejora la capacidad de la red para capturar y aprender características complejas en el conjunto de datos de entrenamiento.

9.2.2 Creación de un codificador

Para crear una capa de codificador, primero definimos la siguiente clase `EncoderLayer()` y clase `SublayerConnection()`.

Listado 9.3 Una clase para definir una capa de codificador

```
class EncoderLayer(nn.Module):
    def __init__(self, size, self_attn, feed_forward, dropout):
        super().__init__()
        self.self_attn = self_attn
        self.feed_forward = feed_forward
        self.sublayer = nn.ModuleList([deepcopy(
            SublayerConnection(size, dropout)) for i in range(2)])
        self.size = size
    def forward(self, x, mask):
        x = self.sublayer[0](
            x, lambda x: self.self_attn(x, x, x, mask))      ①
        output = self.sublayer[1](x, self.feed_forward)       ②
        return output
class SublayerConnection(nn.Module):
    def __init__(self, size, dropout):
        super().__init__()
        self.norm = LayerNorm(size)
        self.dropout = nn.Dropout(dropout)
    def forward(self, x, sublayer):
        output = x + self.dropout(sublayer(self.norm(x)))   ③
        return output
```

- ① La primera subcapa de cada capa de codificador es una red de autoatención de múltiples cabezales.
- ② La segunda subcapa en cada capa de codificador es una red de avance.
- ③ Cada subcapa pasa por una conexión residual y una normalización de capa.

Cada capa de codificador se compone de dos subcapas distintas: una es una capa de autoatención multicabezal, como se describe en la clase `MultiHeadAttention()`, y la otra es una red de avance posicional, directa y totalmente conectada, como se especifica en la clase `PositionwiseFeedForward()`. Además, ambas subcapas incorporan normalización de capas y conexiones residuales. Como se explica en el capítulo 6, una conexión residual implica pasar la entrada a través de una secuencia de transformaciones (ya sea la capa de atención o la de avance en este contexto) y luego agregar la entrada de nuevo a la salida de estas transformaciones. El método de conexión residual se emplea para combatir el problema de los gradientes de desaparición, un problema común en redes muy profundas. Otra ventaja de las conexiones residuales en Transformers es proporcionar un paso para pasar las codificaciones posicionales (que se calculan solo antes de la primera capa) a las capas posteriores.

La normalización de capas es similar a la normalización por lotes que implementamos en el capítulo 4. Estandariza las observaciones de una capa para que tengan una media cero y una desviación estándar unitaria. Para lograr esto, dentro del módulo local, definimos la clase `LayerNorm()` que ejecuta la normalización de capas de la siguiente manera:

```
class LayerNorm(nn.Module):  
    def __init__(self, features, eps=1e-6):  
        super().__init__()  
        self.a_2 = nn.Parameter(torch.ones(features))  
        self.b_2 = nn.Parameter(torch.zeros(features))  
        self.eps = eps  
    def forward(self, x):  
        mean = x.mean(-1, keepdim=True)  
        std = x.std(-1, keepdim=True)  
        x_zscore = (x - mean) / torch.sqrt(std ** 2 + self.eps)
```

```
        output = self.a_2*x_zscore+self.b_2
    return output
```

Los valores `mean` y `std` de la clase anterior son la media y la desviación estandar de las entradas de cada capa. Las capas y de la clase se expanden de nuevo hasta alcanzar la forma de la entrada

```
.std LayerNorm() a_2 b_2 LayerNorm() x_zscore x
```

Ahora podemos crear un codificador apilando seis capas. Para ello, definimos la clase `Encoder()` en el módulo local:

```
from copy import deepcopy
class Encoder(nn.Module):
    def __init__(self, layer, N):
        super().__init__()
        self.layers = nn.ModuleList(
            [deepcopy(layer) for i in range(N)])
        self.norm = LayerNorm(layer.size)
    def forward(self, x, mask):
        for layer in self.layers:
            x = layer(x, mask)
        output = self.norm(x)
        return output
```

Aquí, la clase `Encoder()` se define con dos argumentos: `layer`, que corresponde a una capa de codificador, como se define en la clase `EncoderLayer()` del listado 9.3, y `N`, el número de capas de codificador en el codificador. La clase `Encoder()` toma la entrada `x` (por ejemplo, un lote de frases en inglés) y la máscara (para ocultar el relleno de secuencia, como se explicará en el capítulo 10) para generar la salida (representaciones vectoriales que capturan el significado de las frases en inglés).

Con esto, has creado un codificador. A continuación, aprenderás a crear un decodificador.

9.3 Construcción de un transformador codificador-decodificador

Ahora que entiendes cómo crear un codificador en el Transformer, pasemos al decodificador. En esta sección, aprenderás a crear una capa de de-

codificador. Luego, apilarás $N = 6$ capas de decodificador idénticas para formar un decodificador.

Luego creamos un transformador codificador-decodificador con cinco componentes: `encoder`, `decoder`, `src_embed`, `tgt_embed`, y `generator`, que explicaré en esta sección.

9.3.1 Creación de una capa de decodificador

Cada capa del decodificador consta de tres subcapas: (1) una capa de autoatención multicabezal, (2) la atención cruzada entre la salida de la primera subcapa y la salida del codificador, y (3) una red de retroalimentación.

Cada una de estas tres subcapas incorpora una normalización de capa y la conexión residual, similar a lo que hemos hecho en las capas del codificador. Además, la subcapa de autoatención multicabezal de la pila del decodificador está enmascarada para evitar que las posiciones atiendan a las posiciones posteriores. La máscara obliga al modelo a usar elementos anteriores en una secuencia para predecir elementos posteriores. Explicaré cómo funciona la autoatención multicabezal enmascarada en breve. Para implementarla, definimos la clase `DecoderLayer()` en el módulo local.

Listado 9.4 Creación de una capa de decodificador

```
class DecoderLayer(nn.Module):
    def __init__(self, size, self_attn, src_attn,
                 feed_forward, dropout):
        super().__init__()
        self.size = size
        self.self_attn = self_attn
        self.src_attn = src_attn
        self.feed_forward = feed_forward
        self.sublayer = nn.ModuleList([deepcopy(
            SublayerConnection(size, dropout)) for i in range(3)])
    def forward(self, x, memory, src_mask, tgt_mask):
        x = self.sublayer[0](x, lambda x:
                            self.self_attn(x, x, x, tgt_mask))
        x = self.sublayer[1](x, lambda x:
                            self.src_attn(x, memory, memory, src_mask))
        output = self.sublayer[2](x, self.feed_forward)
        return output
```

(1)

(2)

(3)

- ① La primera subcapa es una capa de autoatención multicabezal enmascarada.
- ② La segunda subcapa es una capa de atención cruzada entre el idioma de destino y el idioma de origen.
- ③ La tercera subcapa es una red de propagación hacia adelante.

Para ilustrar el funcionamiento de una capa de decodificador, consideremos nuestro ejemplo actual. El decodificador toma tokens `['BOS', 'comment', 'et', 'es-vous', '?']`, junto con la salida del codificador (denominado `memory` en el bloque de código anterior), para predecir la secuencia `['comment', 'et', 'es-vous', '?', 'EOS']`. La incrustación de `['BOS', 'comment', 'et', 'es-vous', '?']` es un tensor de tamaño $(1, 5, 256)$: 1 es el número de secuencias en el lote, 5 es el número de tokens en la secuencia y 256 significa que cada token está representado por un vector de 256 valores. Pasamos esta incrustación a través de la primera subcapa, una capa de autoatención multicabezal enmascarada. Este proceso es similar al cálculo de autoatención multicabezal que se vio anteriormente en la capa del codificador. Sin embargo, el proceso utiliza una máscara, designada en el bloque `tgt_mask` de código anterior, que es un tensor de 5×5 con los siguientes valores en el ejemplo actual:

```
tensor([[ True, False, False, False, False],
        [ True,  True, False, False, False],
        [ True,  True,  True, False, False],
        [ True,  True,  True,  True, False],
        [ True,  True,  True,  True,  True]], device='cuda:0')
```

Como habrás notado, la mitad inferior de la máscara (valores por debajo de la diagonal principal en el tensor) se activa como `True`, y la mitad superior (valores por encima de la diagonal principal) se desactiva como `False`. Cuando esta máscara se aplica a las puntuaciones de atención, el primer token se presta solo a sí mismo durante el primer paso de tiempo. En el segundo paso de tiempo, las puntuaciones de atención se calculan exclusivamente entre los dos primeros tokens. A medida que el proceso continúa, por ejemplo, en el tercer paso de tiempo, el decodificador usa tokens `['BOS', 'comment', 'et']` para predecir el token `'es-vous'`,

y los puntajes de atención se calculan solo entre estos tres tokens, ocultando efectivamente los tokens futuros ['es-vous', '?']

Tras este proceso, la salida generada desde la primera subcapa, que es un tensor de tamaño (1, 5, 256), coincide con el tamaño de la entrada. Esta salida, denominada x , se introduce en la segunda subcapa. Aquí, se calcula la atención cruzada entre x y la salida de la pila del codificador, denominada `memory`. Recordará que `memory` tiene una dimensión de (1, 6, 256), ya que la frase en inglés «How are you?» se convierte a seis tokens ['BOS', 'how', 'are', 'you', '?', 'EOS'].

La Figura 9.8 muestra cómo se calculan los pesos de atención cruzada. Para calcular la atención cruzada entre x y `memory`, primero pasamos x a través de una red neuronal para obtener la consulta, que tiene una dimensión de (1, 5, 256). Luego pasamos `memory` a través de dos redes neuronales para obtener la clave y el valor, cada uno con una dimensión de (1, 6, 256). La puntuación de atención escalada se calcula utilizando la fórmula especificada en la ecuación 9.1. Esta puntuación de atención escalada tiene una dimensión de (1, 5, 6): la consulta Q tiene una dimensión de (1, 5, 256) y la clave transpuesta K tiene una dimensión de (1, 256, 6). Por lo tanto, la puntuación de atención escalada, que es el producto escalar de los dos, escalado por $\sqrt{d_k}$, tiene un tamaño de (1, 5, 6). Después de aplicar la función softmax a la puntuación de atención escalada, obtenemos los pesos de atención, que es una matriz de 5×6 . Esta matriz nos indica cómo los cinco tokens de la entrada en francés ['BOS', 'comment', 'et', 'es-vous', '?'] corresponden a los seis tokens de la frase en inglés ['BOS', 'how', 'are', 'you', '?', 'EOS']. Así es como el decodificador capta el significado de la frase en inglés al traducir.

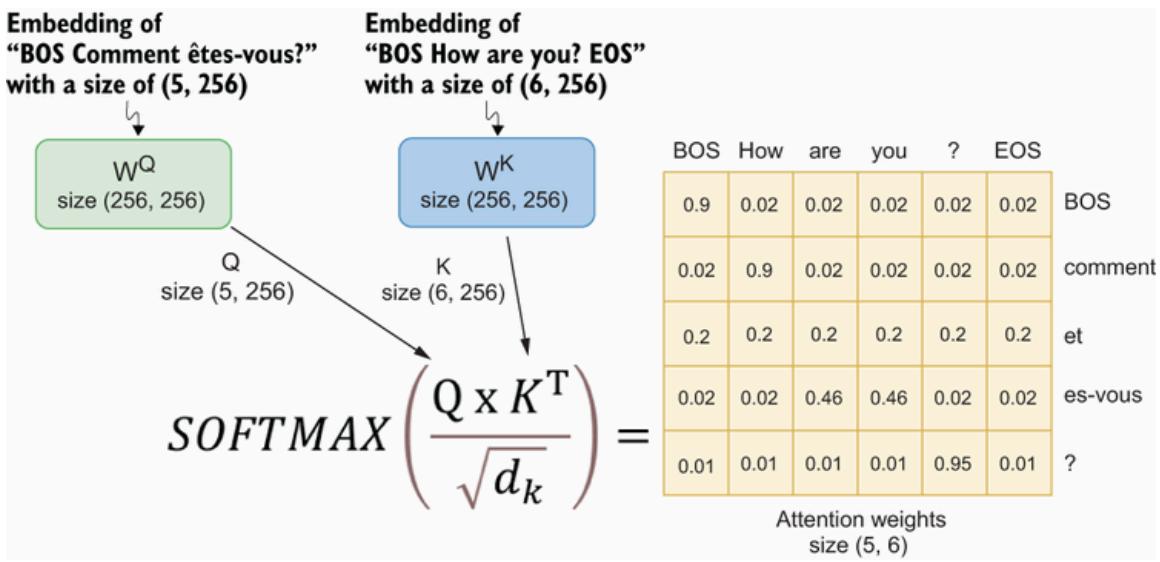


Figura 9.8. Ejemplo de cómo se calculan los pesos de atención cruzada entre la entrada del decodificador y la salida del codificador. La entrada del decodificador se pasa a través de una red neuronal para obtener la consulta Q. La salida del codificador se pasa a través de otra red neuronal para obtener la clave K. Las puntuaciones de atención cruzada escaladas se calculan como el producto escalar de Q y K dividido por la raíz cuadrada de la dimensión de K. Finalmente, aplicamos la función softmax a las puntuaciones de atención cruzada escaladas para obtener los pesos de atención cruzada, que demuestran cómo cada elemento de Q se relaciona con todos los elementos de K.

La atención cruzada final en la segunda subcapa se calcula entonces como el producto escalar de los pesos de atención y el vector de valores V. Los pesos de atención tienen una dimensión de (1, 5, 6) y el vector de valores tiene una dimensión de (1, 6, 256), por lo que la atención cruzada final, que es el producto escalar de los dos, tiene un tamaño de (1, 5, 256). Por lo tanto, la entrada y la salida de la segunda subcapa tienen la misma dimensión de (1, 5, 256). Despues del procesamiento a través de esta segunda subcapa, la salida se dirige a la tercera subcapa, que es una red de propagación hacia adelante.

9.3.2 Creación de un transformador codificador-decodificador

El decodificador consta de $N = 6$ capas de decodificador idénticas.

La clase `Decoder()` se define en el módulo local de la siguiente manera:

```
class Decoder(nn.Module):
    def __init__(self, layer, N):
        super().__init__()
        self.layers = nn.ModuleList(
            [deepcopy(layer) for i in range(N)])
        self.norm = LayerNorm(layer.size)
    def forward(self, x, memory, src_mask, tgt_mask):
        for layer in self.layers:
```

```
    x = layer(x, memory, src_mask, tgt_mask)
    output = self.norm(x)
    return output
```

Para crear un transformador codificador-decodificador, primero definimos una clase `Transformer()` en el módulo local. Abra el archivo ch09util.py y verá la definición de la clase, como se muestra en la siguiente lista.

Listado 9.5 Una clase para representar un codificador-decodificador Transformador

```
class Transformer(nn.Module):
    def __init__(self, encoder, decoder,
                 src_embed, tgt_embed, generator):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.src_embed = src_embed
        self.tgt_embed = tgt_embed
        self.generator = generator
    def encode(self, src, src_mask):
        return self.encoder(self.src_embed(src), src_mask)
    def decode(self, memory, src_mask, tgt, tgt_mask):
        return self.decoder(self.tgt_embed(tgt),
                           memory, src_mask, tgt_mask)
    def forward(self, src, tgt, src_mask, tgt_mask):
        memory = self.encode(src, src_mask) ③
        output = self.decode(memory, src_mask, tgt, tgt_mask) ④
        return output
```

① Define el codificador en el Transformador

② Define el decodificador en el Transformador

③ El idioma fuente se codifica en representaciones vectoriales abstractas.

④ El decodificador utiliza estas representaciones vectoriales para generar la traducción en el idioma de destino.

La clase `Transformer()` se construye con cinco componentes clave: `encoder`, `decoder`, `src_embed`, `tgt_embed` y `generator`. El codificador y

el decodificador están representados por las clases `Encoder()` y `Decoder()` definidas previamente. En el siguiente capítulo, aprenderá a generar la incrustación en el idioma de origen: procesaremos representaciones numéricas de frases en inglés mediante incrustación de palabras y codificación posicional, combinando los resultados para formar el componente `src_embed`. De igual manera, para el idioma de destino, procesaremos representaciones numéricas de frases en francés de la misma manera, utilizando la salida combinada como componente `tgt_embed`. El generador produce probabilidades predichas para cada índice que corresponde a los tokens en el idioma de destino. En la siguiente sección, definiremos una clase `Generator()` para este propósito.

9.4 Juntando todas las piezas

En esta sección, juntaremos todas las piezas para crear un modelo que pueda traducir entre dos idiomas cualesquiera.

9.4.1 Definición de un generador

Primero, definimos una clase `Generator()` en el módulo local para generar la distribución de probabilidad del siguiente token (véase la figura 9.9). La idea es conectar un cabezal al decodificador para las tareas posteriores. En nuestro ejemplo del siguiente capítulo, la tarea posterior consiste en predecir el siguiente token en la traducción al francés.

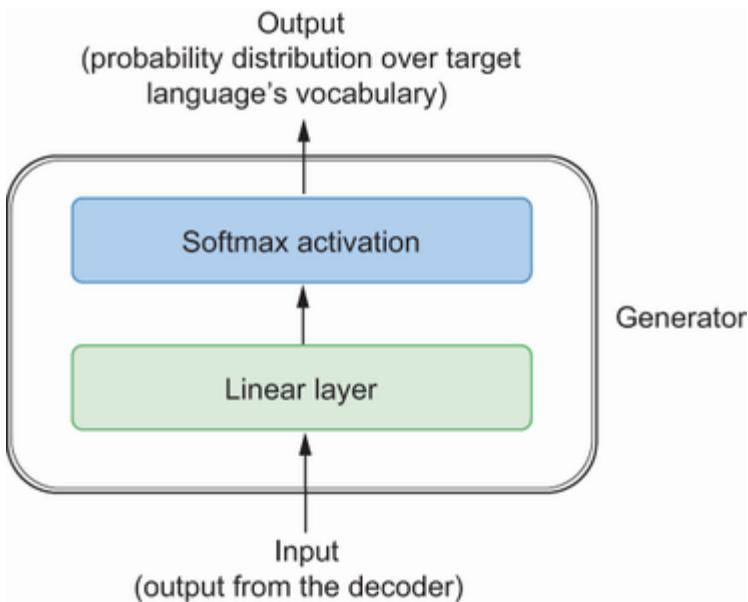


Figura 9.9 Estructura del generador en el Transformador. El generador convierte la salida de la pila del decodificador en una distribución de probabilidad sobre el vocabulario del idioma de destino, de modo que el Transformador pueda usar esta distribución para predecir el siguiente token en la traducción al francés de una frase en inglés. El generador contiene una capa lineal para que el número de salidas sea igual al número de tokens en el vocabulario francés. El generador también aplica una activación softmax a la salida para que esta sea una distribución de probabilidad.

La clase se define de la siguiente manera:

```

class Generator(nn.Module):
    def __init__(self, d_model, vocab):
        super().__init__()
        self.proj = nn.Linear(d_model, vocab)

    def forward(self, x):
        out = self.proj(x)
        probs = nn.functional.log_softmax(out, dim=-1)
        return probs

```

La clase `Generator()` genera probabilidades predichas para cada índice correspondiente a los tokens en el idioma de destino. Esto permite al modelo predecir tokens secuencialmente de forma autorregresiva, utilizando tokens generados previamente y la salida del codificador.

9.4.2 Creación de un modelo para traducir entre dos idiomas

Ahora estamos listos para crear un modelo de Transformador para traducir entre dos idiomas cualesquiera (por ejemplo, de inglés a francés o de chino a inglés). La función `create_model()` definida en el módulo local lo logra.

Listado 9.6 Creación de un transformador para traducir entre dos idiomas

```
def create_model(src_vocab, tgt_vocab, N, d_model,
                 d_ff, h, dropout=0.1):
    attn=MultiHeadedAttention(h, d_model).to(DEVICE)
    ff=PositionwiseFeedForward(d_model, d_ff, dropout).to(DEVICE)
    pos=PositionalEncoding(d_model, dropout).to(DEVICE)
    model = Transformer(
        Encoder(EncoderLayer(d_model,deepcopy(attn),deepcopy(ff),
                           dropout).to(DEVICE),N).to(DEVICE), ①
        Decoder(DecoderLayer(d_model,deepcopy(attn),
                           deepcopy(attn),deepcopy(ff), dropout).to(DEVICE),
                           N).to(DEVICE), ②
        nn.Sequential(Embeddings(d_model, src_vocab).to(DEVICE),
                           deepcopy(pos)), ③
        nn.Sequential(Embeddings(d_model, tgt_vocab).to(DEVICE),
                           deepcopy(pos)), ④
        Generator(d_model, tgt_vocab)).to(DEVICE)
    for p in model.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform_(p)
    return model.to(DEVICE)
```

① Crea un codificador instanciando la clase Encoder()

② Crea un decodificador instanciando la clase Decoder()

③ Crea src_embed pasando el idioma de origen a través de la incrustación de palabras y la codificación posicional

④ Crea tgt_embed pasando el idioma de destino a través de la incrustación de palabras y la codificación posicional

⑤ Crea un generador instanciando la clase Generator()

El elemento principal de la función `create_model()` es la clase `Transformer()`, definida previamente. Recordemos que la clase `Transformer()` se construye con cinco elementos esenciales: `encoder`, `decoder`, `src_embed`, `tgt_embed` y `generator`. Dentro de la función `create_model()`, construimos secuencialmente estos cinco componentes, utilizando las clases `Encoder()` y `Decoder()`, definidas recientemente

`Generator()`. En el siguiente capítulo, analizaremos en detalle cómo generar las incrustaciones de los idiomas de origen y destino, `src_embed` y `tgt_embed`.

En el siguiente capítulo, aplicarás el Transformador que creaste aquí a la traducción del inglés al francés. Entrenarás el modelo con más de 47 000 pares de traducciones del inglés al francés. Después, usarás el modelo entrenado para traducir frases comunes del inglés al francés.

Resumen

- Los transformadores son modelos avanzados de aprendizaje profundo que destacan en la gestión de desafíos de predicción secuencia a secuencia. Su fortaleza reside en la comprensión eficaz de las relaciones entre los elementos de las secuencias de entrada y salida a largas distancias.
- El aspecto revolucionario de la arquitectura Transformer reside en su mecanismo de atención. Este mecanismo evalúa la relación entre las palabras de una secuencia mediante la asignación de pesos, determinando su grado de relación con base en los datos de entrenamiento. Esto permite que los modelos Transformer como ChatGPT comprendan las relaciones entre palabras, lo que permite una comprensión más eficaz del lenguaje humano.
- Para calcular SDPA, la incrustación de entrada X se procesa a través de tres capas de red neuronal distintas: consulta (Q), clave (K) y valor (V). Los pesos correspondientes a estas capas son W^Q , W^K y W^V . Podemos calcular Q, K y V como $Q = X * W^Q$, $K = X * W^K$ y $V = X * W^V$. SDPA se calcula de la siguiente manera:

$$\text{Attention}(Q, K, V) = \text{softmax} \left(\frac{Q * K^T}{\sqrt{d_k}} \right) * V$$

Donde d_k representa la dimensión del vector clave K. La función softmax se aplica a las puntuaciones de atención, convirtiéndolas en ponderaciones de atención. Esto garantiza que la atención total que una palabra otorga a todas las palabras de la oración sume el 100 %. La atención final es el producto escalar de estas ponderaciones de atención y el vector de valores V.

- En lugar de usar un conjunto único de vectores de consulta, clave y valor, los modelos Transformer emplean un concepto denominado atención multicabezal. Los vectores de consulta, clave y valor se dividen en múltiples cabezas. Cada cabeza presta atención a diferentes partes o aspectos de la entrada, lo que permite al modelo capturar una gama más amplia de información y obtener una comprensión más detallada y contextualizada de los datos de entrada. La atención multicabezal es especialmente útil cuando una palabra tiene múltiples significados en una oración.

¹ Vaswani et al., 2017, “La atención es todo lo que necesitas”.

<https://arxiv.org/abs/1706.03762>.

² Dzmitry Bahdanau, Kyunghyun Cho y Yoshua Bengio, 2014, “Traducción automática neuronal mediante el aprendizaje conjunto de alineamiento y traducción”. <https://arxiv.org/abs/1409.0473>.

10. Entrenamiento de un Transformer para traducir del inglés al francés

Este capítulo cubre

- Tokenización de frases en inglés y francés en subpalabras
- Comprensión de la incrustación de palabras y la codificación posicional
- Entrenando a un Transformer desde cero para traducir del inglés al francés
- Usando el Transformer entrenado para traducir una frase del inglés al francés

En el último capítulo, construimos un Transformer desde cero que puede traducir entre dos idiomas, basándonos en el artículo “Attention Is All You Need”.¹ Específicamente, implementamos el mecanismo de autoatención, utilizando vectores de consulta, clave y valor para calcular la atención del producto escalar (SDPA).

Para comprender mejor la autoatención y los Transformers, en este capítulo utilizaremos la traducción del inglés al francés como caso práctico. Al explorar el proceso de entrenamiento de un modelo para convertir oraciones del inglés al francés, comprenderá a fondo la arquitectura de los Transformers y el funcionamiento del mecanismo de atención.

Imagínate haber reunido más de 47 000 pares de traducción del inglés al francés. Tu objetivo es entrenar al codificador-decodificador Transformer del capítulo anterior con este conjunto de datos. Este capítulo te guiará por todas las fases del proyecto. Primero, usarás la tokenización de subpalabras para dividir las frases en inglés y francés en tokens. Luego, crearás tus vocabularios en inglés y francés, que contienen todos los tokens únicos de cada idioma. Los vocabularios te permiten representar frases en inglés y francés como secuencias de índices. Después, usarás la incrustación de palabras para transformar estos índices (esencialmente vectores one-hot) en representaciones vectoriales compactas. Agregaremos codificaciones posicionales a las incrustaciones de palabras para formar incrustaciones de entrada. Las

codificaciones posicionales permiten al Transformer conocer el orden de los tokens en la secuencia.

Finalmente, entrenarás al codificador-decodificador Transformer del capítulo 9 para traducir del inglés al francés usando la colección de traducciones del inglés al francés como conjunto de datos de entrenamiento. Después del entrenamiento, aprenderás a traducir frases comunes del inglés al francés con el Transformer entrenado. Específicamente, usarás el codificador para capturar el significado de la frase en inglés. Luego usarás el decodificador en el Transformer entrenado para generar la traducción al francés de manera autorregresiva, comenzando con el token inicial "BOS". En cada paso de tiempo, el decodificador genera el siguiente token más probable basándose en tokens generados previamente y la salida del codificador, hasta que el token predicho es "EOS", que señala el final de la oración. El modelo entrenado puede traducir frases comunes en inglés con precisión como si estuvieras usando Google Translate para la tarea.

10.1 Tokenización de subpalabras

Como explicamos en el capítulo 8, existen tres métodos de tokenización: tokenización a nivel de carácter, tokenización a nivel de palabra y tokenización de subpalabras. En este capítulo, utilizaremos la tokenización de subpalabras, que busca un equilibrio entre los otros dos métodos. Mantiene las palabras de uso frecuente en su totalidad en el vocabulario y divide las palabras menos comunes o más complejas en subcomponentes.

En esta sección, aprenderá a tokenizar frases en inglés y francés en subpalabras. Posteriormente, creará diccionarios para asignar tokens a índices. Los datos de entrenamiento se convierten en secuencias de índices y se almacenan en lotes para fines de entrenamiento.

10.1.1 Tokenización de frases en inglés y francés

Vaya a <https://mng.bz/WVAw> para descargar el archivo zip que contiene las traducciones del inglés al francés que recopilé de diversas fuentes. Descomprima el archivo y guarde en2fr.csv en la carpeta /files/ de su computadora.

Cargaremos los datos e imprimiremos una frase en inglés, junto con su traducción al francés, de la siguiente manera:

```
import pandas as pd  
  
df=pd.read_csv("files/en2fr.csv")  
num_examples=len(df)  
print(f"there are {num_examples} examples in the training data")  
print(df.iloc[30856]["en"])  
print(df.iloc[30856]["fr"])
```

(1)
(2)
(3)
(4)

① Carga el archivo CSV

② Cuenta cuántos pares de frases hay en los datos

③ Imprime un ejemplo de una frase en inglés

④ Imprime la traducción al francés correspondiente

La salida del fragmento de código anterior es

```
there are 47173 examples in the training data  
How are you?  
Comment êtes-vous?
```

Hay 47.173 pares de traducciones del inglés al francés en los datos de entrenamiento. Hemos impreso la frase en inglés "¿Cómo estás?" y su correspondiente traducción al francés "¿Cómo estás?" como ejemplo.

Ejecute la siguiente línea de código en una nueva celda de este Jupyter Notebook para instalar la biblioteca `transformers` en su computadora:

```
!pip install transformers
```

A continuación, tokenizaremos las frases en inglés y francés del conjunto de datos. Utilizaremos el modelo XLM preentrenado de Hugging Face como tokenizador, ya que es excelente para gestionar múltiples idiomas, incluyendo frases en inglés y francés.

Listado 10.1 Un tokenizador preentrenado

```
from transformers import XLMTOKENIZER  
  
tokenizer = XLMTOKENIZER.from_pretrained("xlm-clm-enfr-1024")  
  
tokenized_en=tokenizer.tokenize("I don't speak French.") ①  
print(tokenized_en)  
tokenized_fr=tokenizer.tokenize("Je ne parle pas français.") ②  
print(tokenized_fr)  
print(tokenizer.tokenize("How are you?"))  
print(tokenizer.tokenize("Comment êtes-vous?")) ③
```

① Importa el tokenizador preentrenado

② Utiliza el tokenizador para tokenizar una oración en inglés

③ Tokeniza una oración en francés

La salida del listado de código 10.1 es

```
['i</w>', 'don</w>', "'t</w>", 'speak</w>',  
'fr', 'ench</w>', '.</w>']  
['je</w>', 'ne</w>', 'parle</w>', 'pas</w>',  
'franc', 'ais</w>', '.</w>']  
['how</w>', 'are</w>', 'you</w>', '?</w>']  
['comment</w>', 'et', 'es-vous</w>', '?</w>']
```

En el bloque de código anterior, usamos un tokenizador preentrenado del modelo XLM para dividir la oración en inglés "I don't speak French." en un grupo de tokens. En el capítulo 8, desarrollaste un tokenizador personalizado a nivel de palabra. Sin embargo, este capítulo introduce el uso de un tokenizador de subpalabras preentrenado más eficiente, que supera al tokenizador a nivel de palabra en efectividad. La oración "I don't speak French." se tokeniza en `['i', 'don', "'t", 'speak', 'fr', 'ench', '.']`. De manera similar, la oración en francés "Je ne parle pas français." se divide en seis tokens: `['je', 'ne', 'parle', 'pas', 'franc', 'ais', '.']`. También hemos tokenizado la frase en inglés "¿Cómo estás?" y su traducción al francés. Los resultados se muestran en las dos últimas líneas de la salida anterior.

NOTA: Es posible que haya notado que el modelo XLM usa "`</w>`" como separador de tokens, excepto cuando dos tokens forman parte de la misma palabra. La tokenización de subpalabras generalmente resulta en que cada token sea una palabra completa o un signo de puntuación, pero en ocasiones una palabra se divide en sílabas. Por ejemplo, la palabra "French" se divide en "fr" y "ench". Cabe destacar que el modelo no inserta `</w>` "fr" entre "ench", ya que estas sílabas juntas constituyen la palabra "French".

Los modelos de aprendizaje profundo, como Transformers, no pueden procesar texto sin formato directamente; por lo tanto, necesitamos convertir el texto en representaciones numéricas antes de alimentarlos a los modelos. Para ello, creamos un diccionario que asigna todos los tokens en inglés a números enteros.

Listado 10.2 Asignación de tokens ingleses a índices

```
from collections import Counter  
  
en=df["en"].tolist() ①  
  
en_tokens=[["BOS"]+tokenizer.tokenize(x)+["EOS"] for x in en] ②  
PAD=0  
UNK=1  
word_count=Counter()  
for sentence in en_tokens:  
    for word in sentence:  
        word_count[word]+=1  
frequency=word_count.most_common(50000) ③  
total_en_words=len(frequency)+2  
en_word_dict={w[0]:idx+2 for idx,w in enumerate(frequency)} ④  
en_word_dict["PAD"]=PAD  
en_word_dict["UNK"]=UNK  
en_idx_dict={v:k for k,v in en_word_dict.items()} ⑤
```

① Obtiene todas las oraciones en inglés del conjunto de datos de entrenamiento

② Tokeniza todas las oraciones en inglés

③ Cuenta la frecuencia de los tokens

④ Crea un diccionario para asignar tokens a índices

⑤ Crea un diccionario para asignar índices a tokens

Insertamos los tokens "BOS" (inicio de la oración) y "EOS" (fin de la oración) al principio y al final de cada frase, respectivamente. El diccionario en_word_dict asigna a cada token un valor entero único. Además, al token "PAD", utilizado para el relleno, se le asigna el entero 0, mientras que al token "UNK", que representa tokens desconocidos, se le asigna el entero 1. Un diccionario inverso, [] en_idx_dict, asigna los enteros (índices) a sus tokens correspondientes. Esta asignación inversa es esencial para convertir una secuencia de enteros en una secuencia de tokens, lo que nos permite reconstruir la frase original en inglés.

Usando el diccionario en_word_dict, podemos transformar la oración en inglés "No hablo francés" en su representación numérica. Este proceso implica buscar cada token en el diccionario para encontrar su valor entero correspondiente. Por ejemplo:

```
enidx=[en_word_dict.get(i,UNK) for i in tokenized_en]
print(enidx)
```

Las líneas de código anteriores producen el siguiente resultado:

```
[15, 100, 38, 377, 476, 574, 5]
```

Esto significa que la oración en inglés "No hablo francés" ahora está representada por una secuencia de números enteros [15, 100, 38, 377, 476, 574, 5].

También podemos revertir las representaciones numéricas a tokens usando el diccionario en_idx_dict. Este proceso implica asignar cada entero de la secuencia numérica a su token correspondiente, según lo definido en el diccionario. Así es como se hace:

```
entokens=[en_idx_dict.get(i,"UNK") for i in enidx] ①
print(entokens)
en_phrase="".join(entokens)
en_phrase=en_phrase.replace("</w>"," ")
for x in '''?:.;,'(-!&)%'''' :
```

```
en_phrase=en_phrase.replace(f" {x}",f"{x}")
print(en_phrase)
```

(4)

- ① Convierte índices en tokens
- ② Une tokens en una cadena
- ③ Reemplaza el separador con un espacio
- ④ Elimina el espacio antes de los signos de puntuación.

La salida del fragmento de código anterior es

```
['i</w>', 'don</w>', "'t</w>", 'speak</w>',
 'fr', 'ench</w>', '.</w>']
i don't speak french.
```

El diccionario `en_idx_dict` se utiliza para traducir los números a sus tokens originales. Posteriormente, estos tokens se transforman en la frase completa en inglés. Esto se realiza uniendo primero los tokens en una sola cadena y luego sustituyendo el separador `'</w>'` por un espacio. También eliminamos el espacio antes de los signos de puntuación. Observe que la frase en inglés restaurada tiene todas las letras en minúsculas porque el tokenizador preentrenado convierte automáticamente las mayúsculas a minúsculas para reducir el número de tokens únicos. Como verá en el siguiente capítulo, algunos modelos, como GPT2 y ChatGPT, no hacen esto; por lo tanto, tienen un vocabulario más amplio.

Ejercicio 10.1

En el listado 10.1, dividimos la oración "¿Cómo estás?" en tokens

`['how</w>', 'are</w>', 'you</w>', '?</w>']`. Siga los pasos de esta subsección para (i) convertir los tokens en índices usando el diccionario `en_word_dict`; (ii) convertir los índices de nuevo en tokens usando el diccionario `en_idx_dict`; (iii) restaurar la oración en inglés uniendo los tokens en una cadena, cambiando el separador `'</w>'` por un espacio y eliminando el espacio antes de los signos de puntuación.

Podemos aplicar los mismos pasos a las frases en francés para asignar tokens a índices y viceversa.

Listado 10.3 Asignación de tokens franceses a índices

```
fr=df["fr"].tolist()
fr_tokens=[["BOS"]+tokenizer.tokenize(x)+["EOS"] for x in fr] ①
word_count=Counter()
for sentence in fr_tokens:
    for word in sentence:
        word_count[word]+=1
frequency=word_count.most_common(50000) ②
total_fr_words=len(frequency)+2
fr_word_dict={w[0]:idx+2 for idx,w in enumerate(frequency)} ③
fr_word_dict["PAD"]=PAD
fr_word_dict["UNK"]=UNK
fr_idx_dict={v:k for k,v in fr_word_dict.items()} ④
```

① Tokeniza todas las oraciones en francés

② Cuenta la frecuencia de los tokens franceses

③ Crea un diccionario para asignar tokens franceses a índices

④ Crea un diccionario para asignar índices a tokens franceses

El diccionario `fr_word_dict` asigna un entero a cada token francés, mientras que `fr_idx_dict` mapea estos enteros a sus tokens franceses correspondientes. A continuación, demostraré cómo transformar la frase francesa «Je ne parle pas français» en su representación numérica:

```
fridx=[fr_word_dict.get(i,UNK) for i in tokenized_fr]
print(fridx)
```

La salida del fragmento de código anterior es

```
[28, 40, 231, 32, 726, 370, 4]
```

Los tokens de la frase en francés “Je ne parle pas français.” se convierten en una secuencia de números enteros, como se muestra.

Podemos transformar las representaciones numéricas en símbolos franceses usando el diccionario `fr_idx_dict`. Esto implica traducir cada número de la secuencia a su respectivo símbolo francés en el diccionario. Una vez recuperados los símbolos, se pueden unir para reconstruir la frase original en francés. Así es como se hace:

```
frtokens=[fr_idx_dict.get(i,"UNK") for i in fridx]
print(frtokens)
fr_phrase="".join(frtokens)
fr_phrase=fr_phrase.replace("</w>"," ")
for x in '''?:;.,'(-!&%'''':
    fr_phrase=fr_phrase.replace(f" {x}",f"{x}")
print(fr_phrase)
```

La salida del bloque de código anterior es

```
['je</w>', 'ne</w>', 'parle</w>', 'pas</w>',
 'franc', 'ais</w>', '.</w>']
je ne parle pas francais.
```

Es importante reconocer que la frase restaurada en francés no coincide exactamente con su forma original. Esta discrepancia se debe al proceso de tokenización, que transforma todas las mayúsculas en minúsculas y elimina las tildes en francés.

Ejercicio 10.2

En el listado 10.1, hemos dividido la oración «Comment êtes-vous?» en tokens `['comment</w>', 'et', 'es-vous</w>', '?</w>']`. Siga los pasos de esta subsección para (i) convertir los tokens en índices usando el diccionario `fr_word_dict`; (ii) convertir los índices de nuevo en tokens usando el diccionario `fr_idx_dict`; (iii) restaurar la frase en francés uniendo los tokens en una cadena, cambiando el separador `'</w>'` por un espacio y eliminando el espacio antes de los signos de puntuación.

Guarde los cuatro diccionarios en la carpeta `/files/` de su computadora para que pueda cargarlos y comenzar a traducir más tarde sin preocuparse por asignar primero tokens a índices y viceversa:

```
import pickle

with open("files/dict.p","wb") as fb:
    pickle.dump((en_word_dict,en_idx_dict,
                 fr_word_dict,fr_idx_dict),fb)
```

Los cuatro diccionarios ahora están guardados en un solo archivo pickle `dict.p`. También puedes descargar el archivo desde el repositorio de GitHub del libro.

10.1.2 Relleno de secuencias y creación de lotes

Dividiremos los datos de entrenamiento en lotes durante el entrenamiento para lograr eficiencia computacional y convergencia acelerada, como lo hemos hecho en capítulos anteriores.

Crear lotes para otros formatos de datos, como imágenes, es sencillo: basta con agrupar un número específico de entradas para formar un lote, ya que todas tienen el mismo tamaño. Sin embargo, en el procesamiento del lenguaje natural, la creación de lotes puede ser más compleja debido a la variación en la longitud de las oraciones. Para estandarizar la longitud dentro de un lote, se añaden rellenos a las secuencias más cortas. Esta uniformidad es crucial, ya que las representaciones numéricas que se introducen en el Transformer deben tener la misma longitud. Por ejemplo, las frases en inglés de un lote pueden variar en longitud (lo mismo puede ocurrir con las frases en francés). Para solucionar esto, se añaden ceros al final de las representaciones numéricas de las frases más cortas de un lote, garantizando así que todas las entradas del modelo del Transformer tengan la misma longitud.

NOTA: La incorporación de tokens `BOS` y `EOS` al principio y al final de cada oración, así como el relleno de secuencias más cortas dentro de un lote, es una característica distintiva de la traducción automática del lenguaje. Esta distinción surge del hecho de que la entrada consiste en oraciones o frases completas. Por el contrario, como verá en los dos próximos capítulos, el entrenamiento de un modelo de generación de texto no implica estos procesos; la entrada del modelo contiene un número predeterminado de tokens.

Comenzamos convirtiendo todas las frases en inglés en sus representaciones numéricas y luego aplicamos el mismo proceso a las frases en francés:

```

out_en_ids=[[en_word_dict.get(w,UNK) for w in s]
for s in en_tokens]
out_fr_ids=[[fr_word_dict.get(w,UNK) for w in s]
for s in fr_tokens]
sorted_ids=sorted(range(len(out_en_ids)),
                  key=lambda x:len(out_en_ids[x]))
out_en_ids=[out_en_ids[x] for x in sorted_ids]
out_fr_ids=[out_fr_ids[x] for x in sorted_ids]

```

A continuación, colocamos las representaciones numéricas en lotes para el entrenamiento:

```

import numpy as np

batch_size=128
idx_list=np.arange(0,len(en_tokens),batch_size)
np.random.shuffle(idx_list)

batch_indexes=[]
for idx in idx_list:
    batch_indexes.append(np.arange(idx,min(len(en_tokens),
                                             idx+batch_size)))

```

Tenga en cuenta que hemos ordenado las observaciones en el conjunto de datos de entrenamiento según la longitud de las frases en inglés antes de agruparlas en lotes. Este método garantiza que las observaciones de cada lote tengan una longitud comparable, lo que reduce la necesidad de relleno. Como resultado, este enfoque no solo reduce el tamaño total de los datos de entrenamiento, sino que también acelera el proceso de entrenamiento.

Para llenar secuencias en un lote a la misma longitud, definimos la siguiente función:

```

def seq_padding(X, padding=0):
    L = [len(x) for x in X]
    ML = max(L)
    padded_seq = np.array([np.concatenate([x,
                                           [padding] * (ML - len(x))])
                           if len(x) < ML else x for x in X])
    return padded_seq

```

(1)

(2)

① Descubra la longitud de la secuencia más larga del lote.

② Si un lote es más corto que la secuencia más larga, agregue 0 a la secuencia al final.

La función `seq_padding()` primero identifica la secuencia más larga del lote. Luego, añade ceros al final de las secuencias más cortas para garantizar que todas las secuencias del lote cumplan con esta longitud máxima.

Para conservar espacio, hemos creado una clase `Batch()` dentro del módulo local `ch09util.py` que descargaste en el último capítulo (ver figura 10.1).

Listado 10.4 Creación de una clase Batch() en el módulo local

```
import torch
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
class Batch:
    def __init__(self, src, trg=None, pad=0):
        src = torch.from_numpy(src).to(DEVICE).long()
        self.src = src
        self.src_mask = (src != pad).unsqueeze(-2)
        if trg is not None:
            trg = torch.from_numpy(trg).to(DEVICE).long()
            self.trg = trg[:, :-1]
            self.trg_y = trg[:, 1:]
            self.trg_mask = make_std_mask(self.trg, pad)
            self.ntokens = (self.trg_y != pad).data.sum()
```

①

②

③

④

① Crea una máscara de origen para ocultar el relleno al final de la oración

② Crea una entrada para el decodificador

③ Desplaza la entrada un token a la derecha y la usa como salida

④ Crea una máscara de objetivo

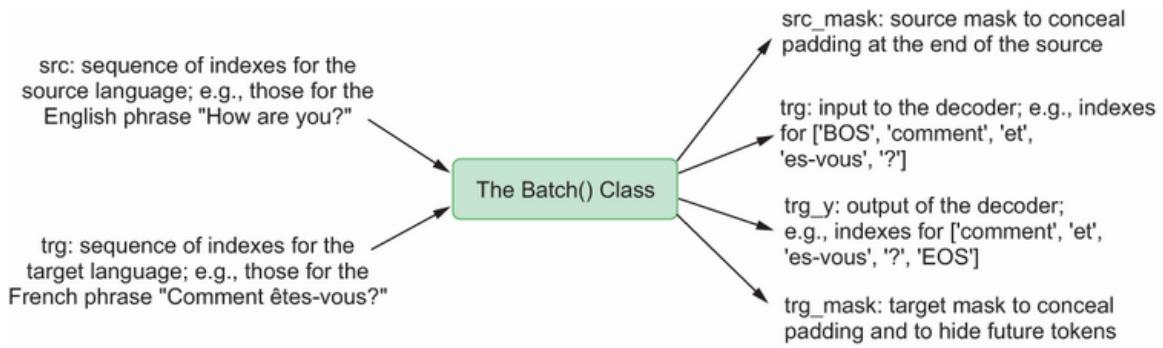


Figura 10.1 ¿Qué hace la clase `Batch()`? La clase `Batch()` toma dos entradas: `src` y `trg`, secuencias de índices para el idioma de origen y el idioma de destino, respectivamente. Añade varios atributos a los datos de entrenamiento: `src_mask`, la máscara de origen para ocultar el relleno; `modified trg`, la entrada al decodificador; `trg_y`, la salida al decodificador; `trg_mask`, la máscara de destino para ocultar el relleno y los tokens futuros.

La clase `Batch()` procesa un lote de frases en inglés y francés, convirtiéndolas a un formato adecuado para el entrenamiento. Para hacer esta explicación más tangible, considere la frase en inglés "How are you?" y su equivalente en francés "Comment êtes-vous?" como nuestro ejemplo. La clase `Batch()` recibe dos entradas: `src`, que es la secuencia de índices que representan los tokens en "How are you?", y `trg`, la secuencia de índices para los tokens en "Comment êtes-vous?". Esta clase genera un tensor, `src_mask`, para ocultar el relleno al final de la oración. Por ejemplo, la oración "¿Cómo estás?" se divide en seis tokens: `['BOS', 'how', 'are', 'you', '?', 'EOS']`. Si esta secuencia es parte de un lote con una longitud máxima de ocho tokens, se agregan dos ceros al final. El tensor `src_mask` indica al modelo que ignore los dos tokens finales en tales escenarios.

La clase `Batch()` también prepara la entrada y la salida para el decodificador del Transformer. Considere la frase francesa "Comment êtes-vous?", que se transforma en seis tokens: `['BOS', 'comment', 'et', 'es-vous', '?', 'EOS']`. Los índices de estos primeros cinco tokens sirven como entrada para el decodificador, llamado `trg`. A continuación, desplazamos esta entrada un token a la derecha para formar la salida del decodificador, `trg_y`. Por lo tanto, la entrada contiene los índices de `['BOS', 'comment', 'et', 'es-vous', '?']`, mientras que la salida contiene los índices de `['comment', 'et', 'es-vous', '?', 'EOS']`. Este enfoque refleja lo que discutimos en el capítulo 8 y está diseñado para forzar al modelo a predecir el siguiente token basándose en los anteriores.

La clase `Batch()` también genera una máscara, `trg_mask`, para la entrada del decodificador. El objetivo de esta máscara es ocultar los tokens subsiguientes en la entrada, garantizando que el modelo se base únicamente en

los tokens anteriores para realizar predicciones. Esta máscara es generada por la función `make_std_mask()`, definida en el módulo local `ch09util`:

```
import numpy as np
def subsequent_mask(size):
    attn_shape = (1, size, size)
    subsequent_mask = np.triu(np.ones(attn_shape),
    k=1).astype('uint8')
    output = torch.from_numpy(subsequent_mask) == 0
    return output
def make_std_mask(tgt, pad):
    tgt_mask=(tgt != pad).unsqueeze(-2)
    output=tgt_mask & subsequent_mask(tgt.size(-1)).
    type_as(tgt_mask.data)
    return output
```

La función `subsequent_mask()` genera una máscara específica para una secuencia, indicando al modelo que se centre únicamente en la secuencia real e ignore los ceros de relleno al final, que se utilizan únicamente para estandarizar la longitud de la secuencia. `make_std_mask()`, por otro lado, la función construye una máscara estándar para la secuencia objetivo. Esta máscara estándar tiene la doble función de ocultar tanto los ceros de relleno como los tokens futuros en la secuencia objetivo.

A continuación, importamos la clase `Batch()` del módulo local y la usamos para crear lotes de datos de entrenamiento:

```
from utils.ch09util import Batch

class BatchLoader():
    def __init__(self):
        self.idx=0
    def __iter__(self):
        return self
    def __next__(self):
        self.idx += 1
        if self.idx<=len(batch_indexes):
            b=batch_indexes[self.idx-1]
            batch_en=[out_en_ids[x] for x in b]
            batch_fr=[out_fr_ids[x] for x in b]
            batch_en=seq_padding(batch_en)
            batch_fr=seq_padding(batch_fr)
```

```
        return Batch(batch_en,batch_fr)
    raise StopIteration
```

La clase `BatchLoader()` crea lotes de datos destinados al entrenamiento. Cada lote de esta lista contiene 128 pares, cada uno de los cuales contiene representaciones numéricas de una frase en inglés y su correspondiente traducción al francés.

10.2 Incrustación de palabras y codificación posicional

Tras la tokenización en la última sección, las frases en inglés y francés se representan mediante secuencias de índices. En esta sección, utilizará la incrustación de palabras para transformar estos índices (en esencia, vectores one-hot) en representaciones vectoriales compactas. De esta forma, se captura la información semántica y la interrelación de los tokens en una frase. La incrustación de palabras también mejora la eficiencia del entrenamiento: en lugar de vectores one-hot voluminosos, utiliza vectores continuos de menor dimensión para reducir la complejidad y la dimensionalidad del modelo.

El mecanismo de atención procesa todos los tokens de una frase simultáneamente, en lugar de secuencialmente. Esto mejora su eficiencia, pero no le permite reconocer el orden secuencial de los tokens. Por lo tanto, añadiremos codificaciones posicionales a las incrustaciones de entrada mediante funciones seno y coseno de frecuencias variables.

10.2.1 Incrustación de palabras

Las representaciones numéricas de las frases en inglés y francés implican una gran cantidad de índices. Para determinar el número exacto de índices distintos necesarios para cada idioma, podemos contar el número de elementos únicos en los diccionarios `en_word_dict` y `fr_word_dict`. Esto genera el número total de tokens únicos en el vocabulario de cada idioma (los usaremos como entradas para el Transformador más adelante):

```
src_vocab = len(en_word_dict)
tgt_vocab = len(fr_word_dict)
```

```
print(f"there are {src_vocab} distinct English tokens")
print(f"there are {tgt_vocab} distinct French tokens")
```

La salida es

```
there are 11055 distinct English tokens
there are 11239 distinct French tokens
```

En nuestro conjunto de datos, hay 11 055 tokens únicos en inglés y 11 239 tokens únicos en francés. Utilizar la codificación one-hot para estos resultaría en un número excesivo de parámetros para entrenar. Para solucionar esto, emplearemos incrustaciones de palabras, que comprimen las representaciones numéricas en vectores continuos, cada uno con una longitud de

```
d_model = 256.
```

Esto se logra mediante el uso de la clase `Embeddings()`, que se define en el módulo local ch09util:

```
import math

class Embeddings(nn.Module):
    def __init__(self, d_model, vocab):
        super().__init__()
        self.lut = nn.Embedding(vocab, d_model)
        self.d_model = d_model

    def forward(self, x):
        out = self.lut(x) * math.sqrt(self.d_model)
        return out
```

La clase `Embeddings()` definida previamente utiliza la clase `Embedding()` de PyTorch. También multiplica el resultado por la raíz cuadrada de `d_model`, que es 256. Esta multiplicación compensa la división por la raíz cuadrada de `d_model` que se produce posteriormente durante el cálculo de las puntuaciones de atención. La clase `Embeddings()` reduce la dimensionaldad de las representaciones numéricas de las frases en inglés y francés. El funcionamiento de la clase de PyTorch se detalla `Embedding()` en el capítulo 8.

10.2.2 Codificación posicional

Para representar con precisión el orden de secuencia de elementos tanto en la entrada como en la salida, introducimos la clase `PositionalEncoding()` en el módulo local.

Listado 10.5 Una clase para calcular la codificación posicional

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, dropout, max_len=5000):          ①
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)
        pe = torch.zeros(max_len, d_model, device=DEVICE)
        position = torch.arange(0., max_len,
                               device=DEVICE).unsqueeze(1)
        div_term = torch.exp(torch.arange(
            0., d_model, 2, device=DEVICE)
            * -(math.log(10000.0) / d_model))
        pe_pos = torch.mul(position, div_term)
        pe[:, 0::2] = torch.sin(pe_pos)                                ②
        pe[:, 1::2] = torch.cos(pe_pos)                                ③
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        x=x+self.pe[:, :x.size(1)].requires_grad_(False)           ④
        out=self.dropout(x)
        return out
```

① Inicia la clase, permitiendo un máximo de 5.000 posiciones

② Aplica la función seno a los índices pares en el vector

③ Aplica la función coseno a los índices impares en el vector

④ Agrega codificación posicional a la incrustación de palabras

La clase `PositionalEncoding()` genera vectores para las posiciones de secuencia utilizando funciones seno para índices pares y coseno para índices impares. Es importante destacar que, en la clase `PositionalEncoding()`, `requires_grad_(False)` se incluye el argumento porque no es necesario

entrenar estos valores. Permanecen constantes en todas las entradas y no cambian durante el proceso de entrenamiento.

Por ejemplo, los índices de los seis tokens `['BOS', 'how', 'are', 'you', '?', 'EOS']` de la frase en inglés se procesan primero mediante una capa de incrustación de palabras. Este paso transforma estos índices en un tensor con las dimensiones (1, 6, 256): 1 significa que solo hay una secuencia en el lote; 6 significa que hay 6 tokens en la secuencia; 256 significa que cada token está representado por un vector de 256 valores. Tras este proceso de incrustación de palabras, la clase `PositionalEncoding()` se utiliza para calcular las codificaciones posicionales de los índices correspondientes a los tokens `['BOS', 'how', 'are', 'you', '?', 'EOS']`. Esto se hace para proporcionar al modelo información sobre la posición de cada token en la secuencia. Mejor aún, podemos indicar los valores exactos de las codificaciones posicionales de los seis tokens anteriores mediante el siguiente bloque de código:

```
from utils.ch09util import PositionalEncoding
import torch
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"

pe = PositionalEncoding(256, 0.1)          ①
x = torch.zeros(1, 8, 256).to(DEVICE)      ②
y = pe.forward(x)
print(f"the shape of positional encoding is {y.shape}") ③
print(y)                                  ④
```

① Crea una instancia de la clase `PositionalEncoding()` y establece la dimensión del modelo en 256

② Crea una incrustación de palabras y la rellena con ceros

③ Calcula la incrustación de entrada agregando codificación posicional a la incrustación de palabras

④ Imprime la incrustación de entrada, que es lo mismo que la codificación posicional, ya que la incrustación de palabras se establece en cero.

Primero creamos una instancia `pe` de la clase `PositionalEncoding()`, estableciendo la dimensión del modelo en 256 y la tasa de abandono en 0,1.

Dado que el resultado de esta clase es la suma de la incrustación de palabras y la codificación posicional, creamos una incrustación de palabras llena de ceros y la introducimos en `pe`. De esta manera, el resultado es el mismo que la codificación posicional.

Después de ejecutar el bloque de código anterior, verá el siguiente resultado:

```
the shape of positional encoding is torch.Size([1, 8, 256])
tensor([[ [ 0.0000e+00,  1.1111e+00,  0.0000e+00, ... ,  0.0000e+00,
          0.0000e+00,  1.1111e+00],
           [ 9.3497e-01,  6.0034e-01,  8.9107e-01, ... ,  1.1111e+00,
             1.1940e-04,  1.1111e+00],
           [ 0.0000e+00, -4.6239e-01,  1.0646e+00, ... ,  1.1111e+00,
             2.3880e-04,  1.1111e+00],
           ... ,
           [-1.0655e+00,  3.1518e-01, -1.1091e+00, ... ,  1.1111e+00,
             5.9700e-04,  1.1111e+00],
           [-3.1046e-01,  1.0669e+00, -0.0000e+00, ... ,  0.0000e+00,
             7.1640e-04,  1.1111e+00],
           [ 7.2999e-01,  8.3767e-01,  2.5419e-01, ... ,  1.1111e+00,
             8.3581e-04,  1.1111e+00]]], device='cuda:0')
```

El tensor anterior representa la codificación posicional de la frase en inglés "¿Cómo estás?". Es importante destacar que esta codificación posicional también tiene las dimensiones (1, 6, 256), que coinciden con el tamaño de la incrustación de palabras de "¿Cómo estás?". El siguiente paso consiste en combinar la incrustación de palabras y la codificación posicional en un solo tensor.

Una característica esencial de las codificaciones posicionales es que sus valores son los mismos independientemente de las secuencias de entrada. Esto significa que, independientemente de la secuencia de entrada específica, la codificación posicional del primer token siempre será el mismo vector de 256 valores, `[0.0000e+00, 1.1111e+00, ..., 1.1111e+00]` como se muestra en la salida anterior. De igual forma, la codificación posicional del segundo token siempre será `[9.3497e-01, 6.0034e-01, ..., 1.1111e+00]`, y así sucesivamente. Sus valores tampoco cambian durante el proceso de entrenamiento.

10.3 Entrenamiento del transformador para la traducción del inglés al francés

Nuestro modelo de traducción del inglés al francés puede considerarse un clasificador multicategoría. El objetivo principal es predecir el siguiente token del vocabulario francés al traducir una oración en inglés. Esto es similar al proyecto de clasificación de imágenes que analizamos en el capítulo 2, aunque este modelo es significativamente más complejo. Esta complejidad requiere una selección cuidadosa de la función de pérdida, el optimizador y los parámetros del bucle de entrenamiento.

En esta sección, detallaremos el proceso de selección de una función de pérdida y un optimizador adecuados. Entrenaremos el Transformer utilizando lotes de traducciones del inglés al francés como conjunto de datos de entrenamiento. Una vez entrenado el modelo, aprenderá a traducir frases comunes del inglés al francés.

10.3.1 Función de pérdida y el optimizador

Primero, importamos la función `create_model()` del módulo local `ch09util.py` y construimos un Transformer para que podamos entrenarlo para traducir del inglés al francés:

```
from utils.ch09util import create_model

model = create_model(src_vocab, tgt_vocab, N=6,
                     d_model=256, d_ff=1024, h=8, dropout=0.1)
```

El artículo "Attention Is All You Need" utiliza diversas combinaciones de hiperparámetros para construir el modelo. En este caso, elegimos una dimensión de modelo de 256 con 8 cabezas, ya que consideramos que esta combinación funciona bien al traducir del inglés al francés en nuestro contexto. Los lectores interesados podrían usar un conjunto de validación para ajustar los hiperparámetros y seleccionar el mejor modelo para sus propios proyectos.

Seguiremos el artículo original "Attention Is All You Need" y utilizaremos el suavizado de etiquetas durante el entrenamiento. El suavizado de etiquetas se utiliza comúnmente en el entrenamiento de redes neuronales profundas

para mejorar la generalización del modelo. Se utiliza para abordar problemas de exceso de confianza (la probabilidad predicha es mayor que la probabilidad real) y sobreajuste en las clasificaciones. Específicamente, modifica la forma en que el modelo aprende ajustando las etiquetas objetivo, con el objetivo de reducir la confianza del modelo en los datos de entrenamiento, lo que puede resultar en un mejor rendimiento con datos no analizados.

En una tarea de clasificación típica, las etiquetas objetivo se representan en un formato de codificación one-hot. Esta representación implica una certeza absoluta sobre la exactitud de la etiqueta para cada muestra de entrenamiento. El entrenamiento con certeza absoluta puede generar dos problemas principales. El primero es el sobreajuste: el modelo confía demasiado en sus predicciones, ajustándose demasiado a los datos de entrenamiento, lo que puede perjudicar su rendimiento con datos nuevos e inéditos. El segundo problema es la calibración deficiente: los modelos entrenados de esta manera suelen generar probabilidades excesivamente confiadas. Por ejemplo, podrían generar una probabilidad del 99 % para una clase correcta cuando, en realidad, la confianza debería ser menor.

El suavizado de etiquetas ajusta las etiquetas objetivo para que sean menos fiables. En lugar de tener una etiqueta objetivo de `[1, 0, 0]` para un problema de tres clases, se podría tener algo como `[0.9, 0.05, 0.05]`. Este enfoque fomenta que el modelo no confíe demasiado en sus predicciones al penalizar los resultados demasiado fiables. Las etiquetas suavizadas son una mezcla de la etiqueta original y una distribución sobre las demás etiquetas (normalmente la distribución uniforme).

Definimos la siguiente clase `LabelSmoothing()` en el módulo local `ch09util`.

Listado 10.6 Una clase para realizar el suavizado de etiquetas

```
class LabelSmoothing(nn.Module):
    def __init__(self, size, padding_idx, smoothing=0.1):
        super().__init__()
        self.criterion = nn.KLDivLoss(reduction='sum')
        self.padding_idx = padding_idx
        self.confidence = 1.0 - smoothing
        self.smoothing = smoothing
        self.size = size
```

```

        self.true_dist = None
def forward(self, x, target):
    assert x.size(1) == self.size
    true_dist = x.data.clone()
    true_dist.fill_(self.smoothing / (self.size - 2))
    true_dist.scatter_(1,
                       target.data.unsqueeze(1), self.confidence)
    true_dist[:, self.padding_idx] = 0
    mask = torch.nonzero(target.data == self.padding_idx)
    if mask.dim() > 0:
        true_dist.index_fill_(0, mask.squeeze(), 0.0)
    self.true_dist = true_dist
    output = self.criterion(x, true_dist.clone().detach())
    return output

```

① Extrae predicciones del modelo

② Extrae etiquetas reales de los datos de entrenamiento y les agrega ruido

③ Utiliza las etiquetas suavizadas como objetivos al calcular la pérdida

La clase `LabelSmoothing()` primero extrae las predicciones del modelo. Luego, suaviza las etiquetas reales en el conjunto de datos de entrenamiento añadiéndoles ruido. El parámetro `smoothing` controla la cantidad de ruido que se inyecta en la etiqueta real. La etiqueta `[1, 0, 0]` se suaviza a `[0.9, 0.05, 0.05]` si se establece `smoothing=0.1`, y se suaviza a `[0.95, 0.025, 0.025]` si se establece `smoothing=0.05`, por ejemplo. La clase calcula la pérdida comparando las predicciones con las etiquetas suavizadas.

Como en capítulos anteriores, utilizamos el optimizador Adam. Sin embargo, en lugar de usar una tasa de aprendizaje constante durante el entrenamiento, definimos la clase `NoamOpt()` en el módulo local para modificar la tasa de aprendizaje durante el proceso de entrenamiento:

```

class NoamOpt:
    def __init__(self, model_size, factor, warmup, optimizer):
        self.optimizer = optimizer
        self._step = 0
        self.warmup = warmup
        self.factor = factor
        self.model_size = model_size
        self._rate = 0

```

```

def step(self):
    self._step += 1
    rate = self.rate()
    for p in self.optimizer.param_groups:
        p['lr'] = rate
    self._rate = rate
    self.optimizer.step()
def rate(self, step=None):
    if step is None:
        step = self._step
    output = self.factor * (self.model_size ** (-0.5)) *
        min(step ** (-0.5), step * self.warmup ** (-1.5)) ③
    return output

```

① Define los pasos de calentamiento

② Un método `step()` para aplicar el optimizador para ajustar los parámetros del modelo

③ Calcula la tasa de aprendizaje en función de los pasos

La clase `NoamOpt()`, como se definió previamente, implementa una estrategia de ritmo de aprendizaje de calentamiento. Primero, aumenta el ritmo de aprendizaje linealmente durante los pasos iniciales de calentamiento del entrenamiento. Tras este período de calentamiento, la clase disminuye el ritmo de aprendizaje, ajustándolo proporcionalmente a la raíz cuadrada inversa del número de pasos de entrenamiento.

A continuación, creamos el optimizador para el entrenamiento:

```

from utils.ch09util import NoamOpt

optimizer = NoamOpt(256, 1, 2000, torch.optim.Adam(
    model.parameters(), lr=0, betas=(0.9, 0.98), eps=1e-9))

```

Para definir la función de pérdida para el entrenamiento, primero creamos la siguiente clase `SimpleLossCompute()` en el módulo local.

Listado 10.7 Una clase para calcular pérdidas

```
class SimpleLossCompute:  
    def __init__(self, generator, criterion, opt=None):  
        self.generator = generator  
        self.criterion = criterion  
        self.opt = opt  
    def __call__(self, x, y, norm):  
        x = self.generator(x)  
        loss = self.criterion(x.contiguous().view(-1, x.size(-1)),  
                             y.contiguous().view(-1)) / norm  
        loss.backward()  
        if self.opt is not None:  
            self.opt.step()  
            self.opt.optimizer.zero_grad()  
        return loss.data.item() * norm.float()
```

- ① Utiliza el modelo para hacer predicciones
 - ② Compara las predicciones con las etiquetas para calcular la pérdida, utilizando el suavizado de etiquetas
 - ③ Calcula gradientes con respecto a los parámetros del modelo
 - ④ Ajusta los parámetros del modelo (retropropagación)

La clase `SimpleLossCompute()` está diseñada con tres elementos clave: `generator`, que actúa como modelo de predicción; `criterion`, que es una función para calcular la pérdida; y `opt`, el optimizador. Esta clase procesa un lote de datos de entrenamiento, denotados como (x, y) , utilizando el generador de predicciones. Posteriormente, evalúa la pérdida comparando estas predicciones con las etiquetas y reales (lo cual gestiona la clase `LabelSmoothing()` definida anteriormente; las etiquetas y reales se suavizarán en el proceso). La clase calcula gradientes relativos a los parámetros del modelo y utiliza el optimizador para actualizarlos según corresponda.

Ahora estamos listos para definir la función de pérdida:

```
loss_func = SimpleLossCompute(  
    model.generator, criterion, optimizer)
```

A continuación, entrenaremos el Transformer utilizando los datos que preparamos anteriormente en el capítulo.

10.3.2 El bucle de entrenamiento

Podríamos dividir los datos de entrenamiento en un conjunto de entrenamiento y un conjunto de validación, y entrenar el modelo hasta que su rendimiento no mejore en el conjunto de validación, de forma similar a lo que hicimos en el capítulo 2. Sin embargo, para ahorrar espacio, entrenaremos el modelo durante 100 épocas. Calcularemos la pérdida y el número de tokens de cada lote. Después de cada época, calculamos la pérdida promedio como la relación entre la pérdida total y el número total de tokens.

Listado 10.8 Entrenamiento de un transformador para traducir del inglés al francés

```
for epoch in range(100):  
    model.train()  
    tloss=0  
    tokens=0  
    for batch in BatchLoader():  
        out = model(batch.src, batch.trg,  
                    batch.src_mask, batch.trg_mask) ①  
        loss = loss_func(out, batch.trg_y, batch.ntokens) ②  
        tloss += loss  
        tokens += batch.ntokens ③  
    print(f"Epoch {epoch}, average loss: {tloss/tokens}")  
    torch.save(model.state_dict(),"files/en2fr.pth") ④
```

① Hace predicciones usando el Transformador

② Calcula la pérdida y ajusta los parámetros del modelo

③ Cuenta la cantidad de tokens en el lote

④ Guarda los pesos en el modelo entrenado después del entrenamiento

Este proceso de entrenamiento tarda un par de horas si usa una GPU con CUDA. Puede tardar un día entero si usa entrenamiento con CPU. Una vez finalizado el entrenamiento, los pesos del modelo se guardan como en2fr.pth en su computadora. También puede descargar los pesos entrenados de mi sitio web (<https://gattonweb.uky.edu/faculty/lium/gai/ch9.zip>).

10.4 Traducción del inglés al francés con el modelo entrenado

Ahora que has entrenado el Transformador, puedes usarlo para traducir cualquier oración del inglés al francés. Definimos una función `translate()` como la que se muestra en la siguiente lista.

Listado 10.9 Definición de una función `translate()` para traducir del inglés al francés

```
def translate(eng):
    tokenized_en=tokenizer.tokenize(eng)
    tokenized_en=["BOS"]+tokenized_en+["EOS"]
    enidx=[en_word_dict.get(i,UNK) for i in tokenized_en]
    src=torch.tensor(enidx).long().to(DEVICE).unsqueeze(0)
    src_mask=(src!=0).unsqueeze(-2)
    memory=model.encode(src,src_mask)
    start_symbol=fr_word_dict["BOS"]
    ys = torch.ones(1, 1).fill_(start_symbol).type_as(src.data)
    translation=[]
    for i in range(100):
        out = model.decode(memory,src_mask,ys,
                           subsequent_mask(ys.size(1)).type_as(src.data))
        prob = model.generator(out[:, -1])
        _, next_word = torch.max(prob, dim=1)
        next_word = next_word.data[0]
        ys = torch.cat([ys, torch.ones(1, 1).type_as(
            src.data).fill_(next_word)], dim=1)
        sym = fr_idx_dict[ys[0, -1].item()]
        if sym != 'EOS':
            translation.append(sym)
        else:
            break
    trans="".join(translation)
    trans=trans.replace("</w>"," ")
    for x in '''?:;.,'(-!&%'''':
        trans=trans.replace(x,"")
```

```
        trans=trans.replace(f" {x}",f"{x}")
    print(trans)
    return trans
```

(4)

- ① Utiliza un codificador para convertir la frase en inglés en una representación vectorial
- ② Predice el siguiente token usando el decodificador
- ③ Deja de traducir cuando el siguiente token es “EOS”
- ④ Une los tokens predichos para formar una oración en francés

Para traducir una frase del inglés al francés, primero usamos el tokenizador para convertir la oración en inglés en tokens. Luego, añadimos “BOS” y “EOS” al principio y al final de la frase. Usamos el diccionario `en_word_dict` que creamos anteriormente en este capítulo para convertir los tokens en índices. Alimentamos la secuencia de índices al codificador en el modelo entrenado. El codificador genera una representación vectorial abstracta y la pasa al decodificador.

Basándose en la representación vectorial abstracta de la oración en inglés producida por el codificador, el decodificador en el modelo entrenado comienza a traducir de manera autorregresiva, empezando con el token inicial “BOS”. En cada paso de tiempo, el decodificador genera el siguiente token más probable basándose en tokens generados previamente, hasta que el token predicho es “EOS”, que señala el final de la oración. Tenga en cuenta que esto es ligeramente diferente del enfoque de generación de texto discutido en el capítulo 8, donde el siguiente token se elige aleatoriamente de acuerdo con sus probabilidades predichas. Aquí, el método para seleccionar el siguiente token es determinista, lo que significa que el token con la mayor probabilidad se elige con certeza porque nos preocupamos principalmente por la precisión. Sin embargo, puede cambiar a la predicción estocástica como hicimos en el capítulo 8 y usar muestreo `top-K` y temperatura si desea que su traducción sea creativa.

Finalmente, cambiamos el separador de tokens por un espacio y eliminamos el espacio antes de los signos de puntuación. El resultado es la traducción al francés en un formato limpio.

Probemos la función `translate()` con la frase en inglés “¡Hoy es un día hermoso!”:

```
from utils.ch09util import subsequent_mask

with open("files/dict.p","rb") as fb:
    en_word_dict,en_idx_dict,\ 
    fr_word_dict,fr_idx_dict=pickle.load(fb)
trained_weights=torch.load("files/en2fr.pth",
                           map_location=DEVICE)
model.load_state_dict(trained_weights)
model.eval()
eng = "Today is a beautiful day!"
translated_fr = translate(eng)
```

La salida es

aujourd'hui est une belle journee!

Puedes verificar que la traducción al francés significa efectivamente “¡Hoy es un día hermoso!” usando, por ejemplo, Google Translate.

Probemos con una oración más larga y veamos si el modelo entrenado puede traducirla correctamente:

```
eng = "A little boy in jeans climbs a small tree
      while another child looks on."
translated_fr = translate(eng)
```

La salida es

un petit garcon en jeans grimpe un petit arbre
tandis qu'un autre enfant regarde.

Cuando traduzco la salida anterior de nuevo al inglés usando Google Translate, dice: “un niño pequeño con jeans trepa un árbol pequeño mientras otro niño observa”, no es exactamente lo mismo que la oración original en inglés, pero el significado es el mismo.

A continuación, probaremos si el modelo entrenado genera la misma traducción para las dos oraciones en inglés: «No hablo francés» y «No hablo francés». Primero, probemos con la oración «No hablo francés».

```
eng = "I don't speak French."  
translated_fr = translate(eng)
```

La salida es

```
je ne parle pas français.
```

Ahora probemos la frase “No hablo francés”:

```
eng = "I do not speak French."  
translated_fr = translate(eng)
```

La salida esta vez es

```
je ne parle pas français.
```

Los resultados indican que las traducciones al francés de ambas oraciones son exactamente iguales. Esto sugiere que el componente codificador del Transformer capta correctamente la esencia semántica de ambas frases. Las representa como formas vectoriales abstractas continuas similares, que posteriormente se transmiten al decodificador. Este genera traducciones basadas en estos vectores y produce resultados idénticos.

Ejercicio 10.3

Usa la función `translate()` para traducir las siguientes dos oraciones del inglés al francés. Compara los resultados con los del Traductor de Google y comprueba si son iguales: (i) ¡Me encanta esquiar en invierno! (ii) ¿Cómo estás?

En este capítulo, entrenó un codificador-decodificador Transformer para traducir del inglés al francés utilizando más de 47 000 pares de traducciones. El modelo entrenado funciona bien, traduciendo frases comunes en inglés correctamente.

En los siguientes capítulos, explorarás los Transformers que solo funcionan con decodificadores. Aprenderás a construirlos desde cero y a usarlos para generar texto coherente, mejor que el texto generado en el capítulo 8 usando memoria a corto plazo.

Resumen

- Los transformadores procesan datos de entrada, como oraciones, en paralelo, a diferencia de las redes neuronales recurrentes, que procesan los datos secuencialmente. Este paralelismo mejora su eficiencia, pero no les permite reconocer el orden secuencial de la entrada. Para solucionar esto, los transformadores añaden codificaciones posicionales a las incrustaciones de entrada. Estas codificaciones posicionales son vectores únicos asignados a cada posición en la secuencia de entrada y se alinean dimensionalmente con las incrustaciones de entrada.
- El suavizado de etiquetas se utiliza comúnmente en el entrenamiento de redes neuronales profundas para mejorar la generalización del modelo. Se utiliza para abordar problemas de exceso de confianza (la probabilidad predicha es mayor que la probabilidad real) y sobreajuste en las clasificaciones. En concreto, modifica la forma en que el modelo aprende ajustando las etiquetas objetivo, con el objetivo de reducir la confianza del modelo en los datos de entrenamiento, lo que puede mejorar el rendimiento con datos no analizados.
- Basándose en la salida del codificador que captura el significado de la frase en inglés, el decodificador del Transformer entrenado comienza a traducir de forma autorregresiva, comenzando con el token inicial "BOS". En cada paso de tiempo, el decodificador genera el siguiente token más probable basándose en los tokens generados previamente, hasta que el token predicho es "EOS", lo que indica el final de la oración.

¹ Vaswani et al, 2017, “La atención es todo lo que necesitas”.

<https://arxiv.org/abs/1706.03762>.

11. Construyendo un Transformer generativo preentrenado desde cero

Este capítulo cubre

- Construyendo un Transformer generativo preentrenado desde cero
- Autoatención causal
- Extracción y carga de pesos de un modelo preentrenado
- Generación de texto coherente con GPT-2, el predecesor de ChatGPT y GPT-4

Generative Pretrained Transformer 2 (GPT-2) es un modelo de lenguaje grande (LLM) avanzado desarrollado por OpenAI y anunciado en febrero de 2019. Representa un hito significativo en el campo del procesamiento del lenguaje natural (PLN) y ha allanado el camino para el desarrollo de modelos aún más sofisticados, incluidos sus sucesores, ChatGPT y GPT-4.

GPT-2, una mejora respecto a su predecesor, GPT-1, fue diseñado para generar texto coherente y contextualmente relevante a partir de una consigna dada, demostrando una notable capacidad para imitar la generación de texto de forma similar a la humana en diversos estilos y temas. Tras su anuncio, OpenAI decidió inicialmente no lanzar al público la versión más potente de GPT-2 (la que desarrollará desde cero en este capítulo, con 1500 millones de parámetros). La principal preocupación era su posible uso indebido, como la generación de noticias engañosas, la suplantación de identidad en línea o la automatización de la producción de contenido abusivo o falso. Esta decisión desató un importante debate en las comunidades de IA y tecnología sobre la ética del desarrollo de IA y el equilibrio entre innovación y seguridad.

Posteriormente, OpenAI adoptó una estrategia de lanzamiento escalonado, lanzando gradualmente versiones más pequeñas del modelo, mientras monitoreaba su efecto y exploraba estrategias de implementación seguras. Finalmente, en noviembre de 2019, OpenAI lanzó el modelo completo, junto con varios conjuntos de datos y una herramienta para detectar texto generado por el modelo, lo que contribuyó al debate sobre el uso responsable de la IA. Gracias a esta versión, aprenderá a extraer los pesos preentrenados de GPT-2 y a cargarlos en el modelo GPT-2 que cree.

GPT-2 se basa en la arquitectura Transformer que analizamos en los capítulos 9 y 10. Sin embargo, a diferencia del traductor de inglés a francés que creó anteriormente, GPT-2 es un Transformer que solo utiliza decodificadores, lo que significa que no hay una pila de codificadores en el modelo. Al traducir una frase del inglés al francés, el codificador captura el significado de la frase en inglés y lo pasa al decodificador para generar la traducción. Sin embargo, en las tareas de generación de texto, el modelo no necesita un codificador para comprender un idioma diferente. En su lugar, genera texto basándose en los tokens anteriores de la oración, utilizando únicamente una arquitectura de decodificadores. Al igual que otros modelos Transformer, GPT-2 utiliza mecanismos de autoatención para procesar los datos de entrada en paralelo, lo que mejora significativamente la eficiencia y la eficacia del entrenamiento de LLM.

GPT-2 se entrena previamente con un amplio corpus de datos textuales, y básicamente predice la siguiente palabra de una oración a partir de las palabras que la preceden. Este entrenamiento permite al modelo aprender una amplia gama de patrones lingüísticos, gramática y conocimientos.

En este capítulo, aprenderá a construir GPT-2XL, la versión más grande de GPT-2, desde cero. Después, aprenderá a extraer los pesos preentrenados de Hugging Face (una comunidad de IA que aloja y colabora en modelos de aprendizaje automático, conjuntos de datos y aplicaciones) y a cargarlos en su propio modelo GPT-2. Usará su GPT-2 para generar texto mediante una instrucción. GPT-2 calcula las probabilidades de los siguientes tokens y toma muestras de estas probabilidades. Puede producir párrafos de texto coherentes y contextualmente relevantes según la instrucción que recibe. Además, como en el capítulo 8, puede controlar la creatividad del texto generado mediante el uso de la instrucción `temperature` y el muestreo `top-K`.

Si bien GPT-2 representa un avance notable en PNL, es fundamental moderar las expectativas y reconocer sus limitaciones inherentes. Es crucial no comparar GPT-2 directamente con ChatGPT o GPT-4, ya que GPT-2XL solo tiene 1500 millones de parámetros, en comparación con los 175 000 millones de ChatGPT y los 1,76 billones de GPT-4, que se estiman en 175 000 millones. Una de las principales limitaciones de GPT-2 es su falta de comprensión genuina del contenido que genera. El modelo predice la siguiente palabra de una secuencia basándose en la distribución de probabilidad de las palabras en sus datos de entrenamiento, lo que puede producir un texto sintácticamente correcto y aparentemente lógico. Sin embargo, el modelo carece de una comprensión real del significado de las palabras, lo que puede generar posibles imprecisiones, afirmaciones sin sentido o contenido superficial.

Otro factor clave es la limitada percepción contextual de GPT-2. Si bien puede mantener la coherencia en fragmentos cortos de texto, presenta dificultades con pasajes más largos, lo que puede resultar en pérdida de coherencia, contradicciones o contenido irrelevante. Debemos ser cautelosos y no sobreestimar la capacidad del modelo para generar contenido extenso que requiera una atención constante al contexto y al detalle. Por lo tanto, si bien GPT-2 representa un avance significativo en PNL, es importante abordar el texto generado con una buena dosis de escepticismo y establecer expectativas realistas.

11.1 Arquitectura GPT-2 y autoatención causal

GPT-2 funciona como un transformador basado exclusivamente en decodificadores (genera texto basándose en los tokens previos de la oración sin necesidad de que un codificador comprenda otro idioma), imitando el componente decodificador del traductor de inglés a francés descrito en los capítulos 9 y 10. A diferencia de su homólogo bilingüe, GPT-2 carece de codificador y, por lo tanto, no incorpora entradas derivadas del codificador en su proceso de generación de salida. El modelo se basa completamente en los tokens previos de la secuencia para generar su salida.

En esta sección, analizaremos la arquitectura de GPT-2. También profundizaremos en el mecanismo causal de autoatención, que constituye el núcleo del modelo GPT-2.

11.1.1 La arquitectura de GPT-2

GPT-2 está disponible en cuatro tamaños diferentes: pequeño (S), mediano (M), grande (L) y extragrande (XL), cada uno con diferentes capacidades. Nos centraremos principalmente en la versión más potente, GPT-2XL. El modelo GPT-2 más pequeño tiene alrededor de 124 millones de parámetros, mientras que la versión extragrande tiene alrededor de 1500 millones. Es el más potente de los modelos GPT-2, con el mayor número de parámetros. GPT-2XL puede comprender contextos complejos, generando texto coherente y con matices.

GPT-2 consta de numerosos bloques decodificadores idénticos. La versión extragrande tiene 48 bloques decodificadores, mientras que las otras tres versiones tienen 12, 24 y 36 bloques decodificadores, respectivamente. Cada uno de estos bloques decodificadores consta de dos subcapas distintas. La primera subcapa es una capa de autoatención causal, que explicaré en detalle más adelante. La segunda subcapa es una red de alimentación anticipada básica, posicional y totalmente conectada, como hemos visto en los bloques codificador y

decodificador del traductor de inglés a francés. Cada subcapa incorpora normalización de capas y una conexión residual para estabilizar el proceso de entrenamiento.

La figura 11.1 es un diagrama de la arquitectura de GPT-2.

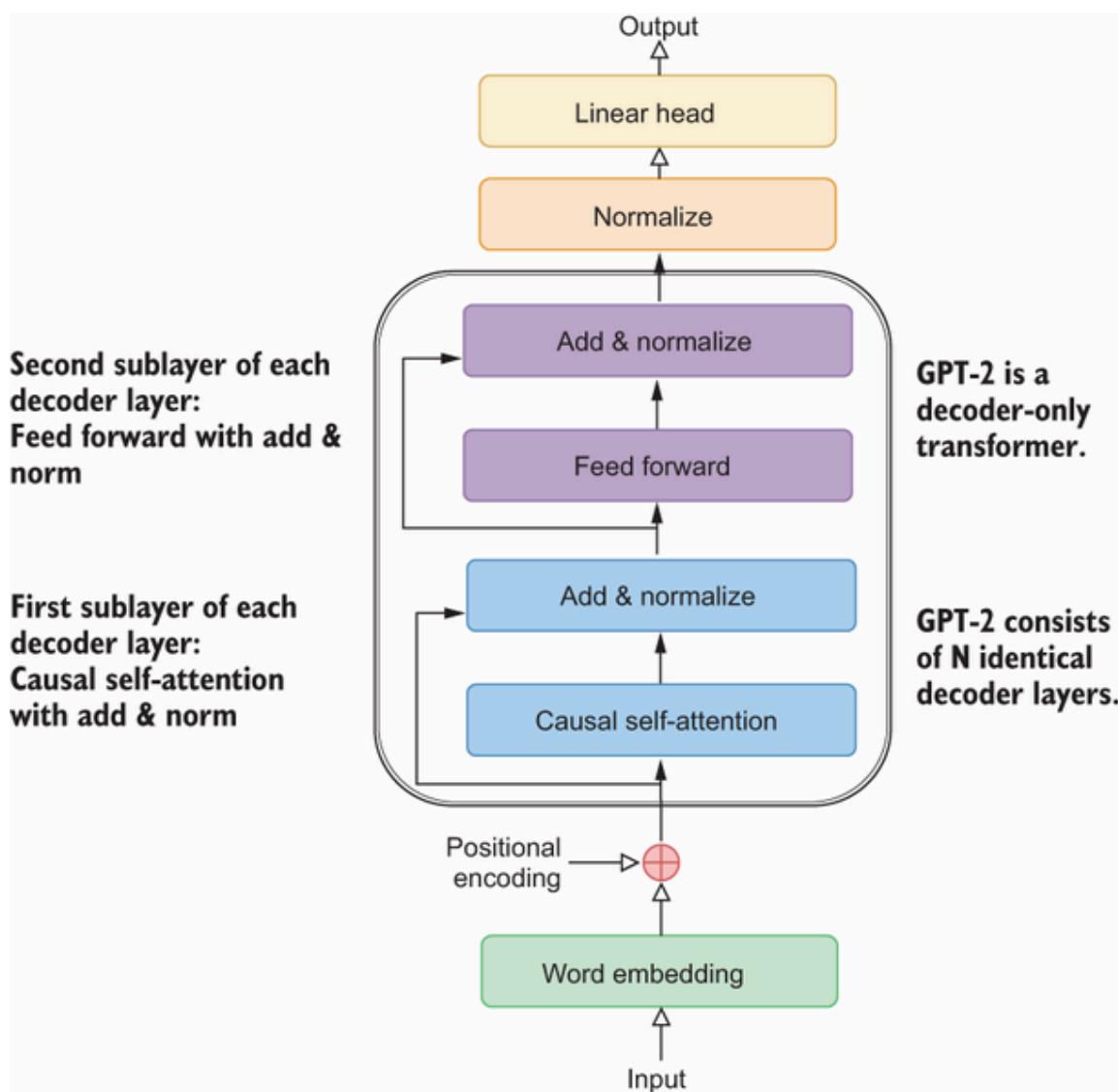


Figura 11.1 Arquitectura del modelo GPT-2. GPT-2 es un transformador solo decodificador, compuesto por N capas idénticas. Cada bloque decodificador contiene dos subcapas. La primera es una capa de autoatención causal. La segunda es una red de retroalimentación. Cada subcapa utiliza normalización de capas y una conexión residual. La entrada se procesa primero mediante incrustación de palabras y codificación posicional, y la suma se procesa posteriormente en el decodificador. La salida del decodificador se procesa mediante normalización de capas y una capa lineal.

GPT-2 primero pasa los índices de una secuencia de tokens mediante incrustación de palabras y codificación posicional para obtener la incrustación de entrada (explico en breve cómo funciona este proceso). La incrustación de entrada se pasa secuencialmente por N bloques decodificadores. Posteriormente, la salida se pasa por la normalización de capas y una capa lineal. El número de salidas en GPT-2 es el número de tokens únicos en el vocabulario (50.257 tokens para todas las versiones de GPT-2). El modelo está diseñado para predecir el siguiente token basándose en todos los tokens anteriores de la secuencia.

Para entrenar GPT-2, OpenAI utilizó un conjunto de datos llamado WebText, re-copilado automáticamente de internet. Este conjunto de datos contenía una amplia variedad de texto, incluyendo enlaces a sitios web como Reddit, que recibieron muchos votos positivos, con el objetivo de cubrir un amplio espectro de lenguajes y temas humanos. Se estima que este conjunto de datos contiene unos 40 GB de texto.

Los datos de entrenamiento se dividieron en secuencias de longitud fija (1024 tokens para todas las versiones de GPT-2) y se utilizaron como entradas. Las secuencias se desplazaron un token a la derecha y se utilizaron como salidas del modelo durante el entrenamiento. Dado que el modelo utiliza autoatención causal, en la que los tokens futuros de una secuencia se enmascaran (es decir, se ocultan) durante el proceso de entrenamiento, esto entrena al modelo para predecir el siguiente token basándose en todos los tokens anteriores de la secuencia.

11.1.2 Incrustación de palabras y codificación posicional en GPT-2

GPT-2 utiliza un método de tokenización de subpalabras llamado Codificador de Pares de Bytes (BPE) para descomponer el texto en tokens individuales (palabras completas o signos de puntuación en la mayoría de los casos, pero sílabas para palabras poco comunes). Estos tokens se asignan a un índice entre 0 y 50 256, ya que el tamaño del vocabulario es de 50 257. GPT-2 transforma el texto de los datos de entrenamiento en representaciones vectoriales que capturan su significado mediante la incrustación de palabras, de forma similar a lo que se hizo en los dos capítulos anteriores.

Para darle un ejemplo concreto, la frase "this is a prompt" se convierte primero en cuatro tokens a través de la tokenización BPE, `['this', ' is', ' a', ' prompt']`. Cada token se representa entonces mediante una variable one-hot de tamaño 50,257. El modelo GPT-2 los pasa a través de una capa de incrustación de palabras para comprimirlos en vectores condensados con valores de punto flotante de un tamaño mucho menor, como una longitud de 1,600 en GPT-2XL (las longitudes son 768, 1,024 y 1,280, para las otras tres versiones de GPT-2, respectivamente). Con la incrustación de palabras, la frase "this is a prompt" se representa mediante una matriz con un tamaño de $4 \times 1,600$ en lugar del original $4 \times 50,257$. La incrustación de palabras reduce significativamente el número de parámetros del modelo y hace que el entrenamiento sea más eficiente. El lado izquierdo de la figura 11.2 representa cómo funciona la incrustación de palabras.

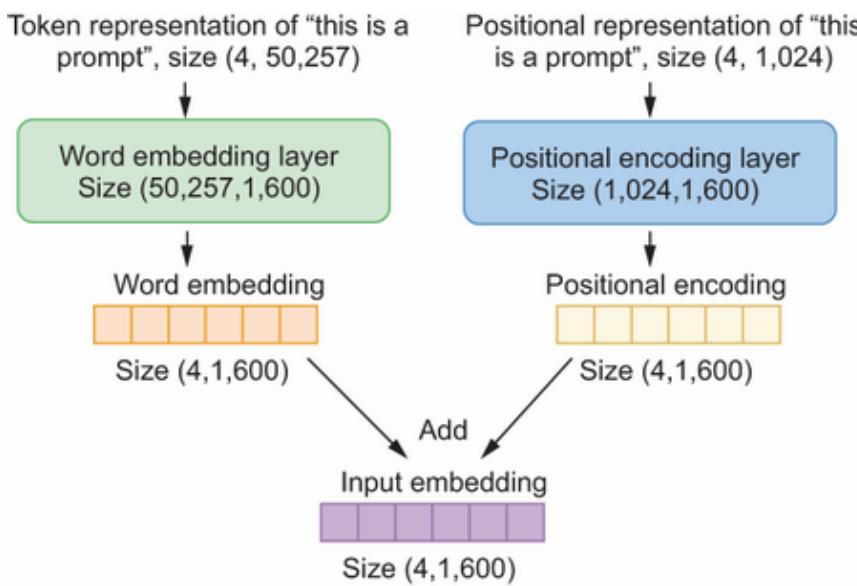


Figura 11.2. GPT-2 primero representa cada token de una secuencia con un vector one-hot de 50.276 valores. La representación del token de la secuencia pasa por una capa de incrustación de palabras para comprimirla en una incrustación de dimensión 1.600. GPT-2 también representa cada posición de la secuencia con un vector one-hot de 1.024 valores. La representación posicional de la secuencia pasa por una capa de codificación posicional para comprimirla en una incrustación de dimensión 1.600. La incrustación de palabras y la codificación posicional se suman para formar la incrustación de entrada.

GPT-2, al igual que otros Transformers, procesa los datos de entrada en paralelo, lo que, inherentemente, no le permite reconocer el orden de la secuencia de entrada. Para solucionar esto, necesitamos añadir codificaciones posicionales a las incrustaciones de entrada. GPT-2 adopta un enfoque único para la codificación posicional, que se aparta de la metodología descrita en el influyente artículo de 2017 "Attention Is All You Need". En cambio, la técnica de GPT-2 para la codificación posicional es paralela a la de las incrustaciones de palabras. Dada la capacidad del modelo para manejar hasta 1024 tokens en una secuencia de entrada, cada posición dentro de la secuencia se denota inicialmente mediante un vector one-hot del mismo tamaño. Por ejemplo, en la secuencia "this is a prompt", el primer token se representa mediante un vector one-hot donde todos los elementos son cero excepto el primero, que se establece en uno. El segundo token sigue el mismo ejemplo, representado por un vector donde todos los elementos excepto el segundo son cero. En consecuencia, la representación posicional de la frase "esto es un mensaje" se manifiesta como una matriz de 4×1024 , como se ilustra en la sección superior derecha de la figura 11.2.

Para generar la codificación posicional, la representación posicional de la secuencia se procesa a través de una red neuronal lineal, que está dimensionada en 1024×1600 . Los pesos dentro de esta red se inicializan aleatoriamente y posteriormente se refinan a través del proceso de entrenamiento. Como resultado, la codificación posicional para cada token en la secuencia es un vector de 1600 valores, que coincide con la dimensión del vector de incrustación de palabras.

La incrustación de entrada de una secuencia es la suma de su incrustación de palabras y codificación posicional, como se muestra en la parte inferior de la figura 11.2. En el contexto de la frase "this is a prompt", tanto la incrustación de palabras como la codificación posicional se estructuran como matrices de 4×1600 . Por lo tanto, la incrustación de entrada para "this is a prompt", que es la suma de estas dos matrices, mantiene una dimensionalidad de 4×1600 .

11.1.3 Autoatención causal en GPT-2

La autoatención causal es un mecanismo crucial dentro del modelo GPT-2 (y, en general, en la serie de modelos GPT), que permite al modelo generar texto condicionando la secuencia de tokens previamente generados. Es similar a la autoatención enmascarada en la primera subcapa de cada capa del decodificador del traductor de inglés a francés que analizamos en los capítulos 9 y 10, aunque su implementación difiere ligeramente.

NOTA: El concepto de "causal" en este contexto se refiere a la capacidad del modelo para garantizar que las predicciones para un token determinado solo puedan ser influenciadas por los tokens que lo preceden en la secuencia, respetando la dirección causal (en el tiempo) de la generación de texto. Esto es esencial para generar resultados de texto coherentes y contextualmente relevantes.

La autoatención es un mecanismo que permite que cada elemento de la secuencia de entrada preste atención a todos los demás elementos de la misma secuencia. En el contexto de modelos Transformer como GPT-2, la autoatención permite al modelo ponderar la importancia de otros elementos al procesar un elemento específico, capturando así el contexto y las relaciones entre las palabras de una oración.

Para garantizar la causalidad, el mecanismo de autoatención de GPT-2 se modifica para que cualquier token dado solo pueda prestar atención a sí mismo y a los tokens que lo han precedido en la secuencia. Esto se logra enmascarando los tokens futuros (es decir, los tokens que vienen después del token actual en la secuencia) en el cálculo de la atención, lo que garantiza que el modelo no pueda "ver" ni ser influenciado por tokens futuros al predecir el siguiente token en una secuencia. Por ejemplo, en la frase "this is a prompt", la máscara oculta las últimas tres palabras en el primer paso de tiempo cuando el modelo usa la palabra "this" para predecir la palabra "is". Para implementar esto, las posiciones correspondientes a los tokens futuros se establecen en menos infinito cuando calculamos las puntuaciones de atención. Después de la activación

de softmax, a los tokens futuros se les asignan pesos cero, eliminándolos efectivamente del cálculo de la atención.

Utilicemos un ejemplo concreto para ilustrar con precisión cómo funciona la autoatención causal en el código. La incrustación de entrada para la frase "este es un mensaje" es una matriz de 4×1600 tras la incrustación de palabras y la codificación posicional. A continuación, pasamos esta incrustación de entrada a través de N capas de decodificador en GPT-2. En cada capa de decodificador, pasa primero por la subcapa de autoatención causal, como se indica a continuación. La incrustación de entrada se pasa a través de tres redes neuronales para crear la consulta Q , la clave K y el valor V, como se muestra en la siguiente lista.

Listado 11.1 Creación de vectores query , key , y value

```
import torch
import torch.nn as nn

torch.manual_seed(42)
x=torch.randn((1,4,1600))                                     ①
c_attn=nn.Linear(1600,1600*3)                                ②
B,T,C=x.size()
q,k,v=c_attn(x).split(1600,dim=2)                           ③
print(f"the shape of Q vector is {q.size()}") 
print(f"the shape of K vector is {k.size()}") 
print(f"the shape of V vector is {v.size()}")                 ④
```

① Crea tres redes neuronales

② Crea una incrustación de entrada x

③ Pasa la entrada incrustando las tres redes neuronales para crear Q, K y V

④ Imprime los tamaños de Q , K y V

Primero creamos una matriz de tamaño 4×1600 , el mismo tamaño que la incrustación de entrada para "este es un mensaje". Luego, pasamos la incrustación de entrada a través de tres redes neuronales, cada una con un tamaño de 1600×1600 , para obtener la consulta Q, la clave K y el valor V. Si ejecuta el bloque de código anterior, verá el siguiente resultado:

```
the shape of Q vector is torch.Size([1, 4, 1600])
the shape of K vector is torch.Size([1, 4, 1600])
the shape of V vector is torch.Size([1, 4, 1600])
```

Las formas de Q, K y V son todas de 4×1600 . A continuación, en lugar de usar un solo cabezal, los dividimos en 25 cabezales paralelos. Cada cabezal presta atención a diferentes partes o aspectos de la entrada, lo que permite al modelo capturar una gama más amplia de información y obtener una comprensión más detallada y contextualizada de los datos de entrada. Como resultado, tenemos 25 conjuntos de Q, K y V:

```
hs=C//25
k = k.view(B, T, 25, hs).transpose(1, 2)
q = q.view(B, T, 25, hs).transpose(1, 2)
v = v.view(B, T, 25, hs).transpose(1, 2)          ①
print(f"the shape of Q vector is {q.size()}")      ②
print(f"the shape of K vector is {k.size()}")
print(f"the shape of V vector is {v.size()}")
```

① Divide Q, K y V en 25 cabezas

② Imprime el tamaño del cabezal múltiple Q, K y V

Si ejecuta el bloque de código anterior, verá el siguiente resultado:

```
the shape of Q vector is torch.Size([1, 25, 4, 64])
the shape of K vector is torch.Size([1, 25, 4, 64])
the shape of V vector is torch.Size([1, 25, 4, 64])
```

Las formas de Q, K y V ahora son $25 \times 4 \times 64$: esto significa que tenemos 25 cabezas; cada cabeza tiene un conjunto de consulta, clave y valor, todos con un tamaño de 4×64 .

A continuación, calculamos las puntuaciones de atención escaladas en cada cabeza:

```
import math
scaled_att = (q @ k.transpose(-2, -1)) *\
              (1.0 / math.sqrt(k.size(-1)))
print(scaled_att[0,0])
```

Las puntuaciones de atención escaladas son el producto escalar de Q y K en cada cabeza, escaladas por la raíz cuadrada de la dimensión de K, que es $1600/25 = 64$. Las puntuaciones de atención escaladas forman una matriz de 4×4 en cada cabeza, y las imprimimos en la primera cabeza:

```
tensor([[ 0.2334,  0.1385, -0.1305,  0.2664],
       [ 0.2916,  0.1044,  0.0095,  0.0993],
       [ 0.8250,  0.2454,  0.0214,  0.8667],
       [-0.1557,  0.2034,  0.2172, -0.2740]],
      grad_fn=<SelectBackward0>)
```

Las puntuaciones de atención escaladas en la primera cabeza también se muestran en la tabla inferior izquierda de la figura 11.3.

Ejercicio 11.1

El tensor `scaled_att` contiene las puntuaciones de atención escaladas en las 25 cabezas. Las hemos impreso en la primera cabeza anteriormente. ¿Cómo se imprimen las puntuaciones de atención escaladas en la segunda cabeza?

A continuación, aplicamos una máscara a las puntuaciones de atención escaladas para ocultar tokens futuros en la secuencia:

```
mask=torch.tril(torch.ones(4,4))                                ①
print(mask)
masked_scaled_att=scaled_att.masked_fill(\n    mask == 0, float('-inf'))                               ②
print(masked_scaled_att[0,0])
```

① Crea una máscara

② Aplica la máscara en los puntajes de atención escalados cambiando los valores a $-\infty$ para tokens futuros

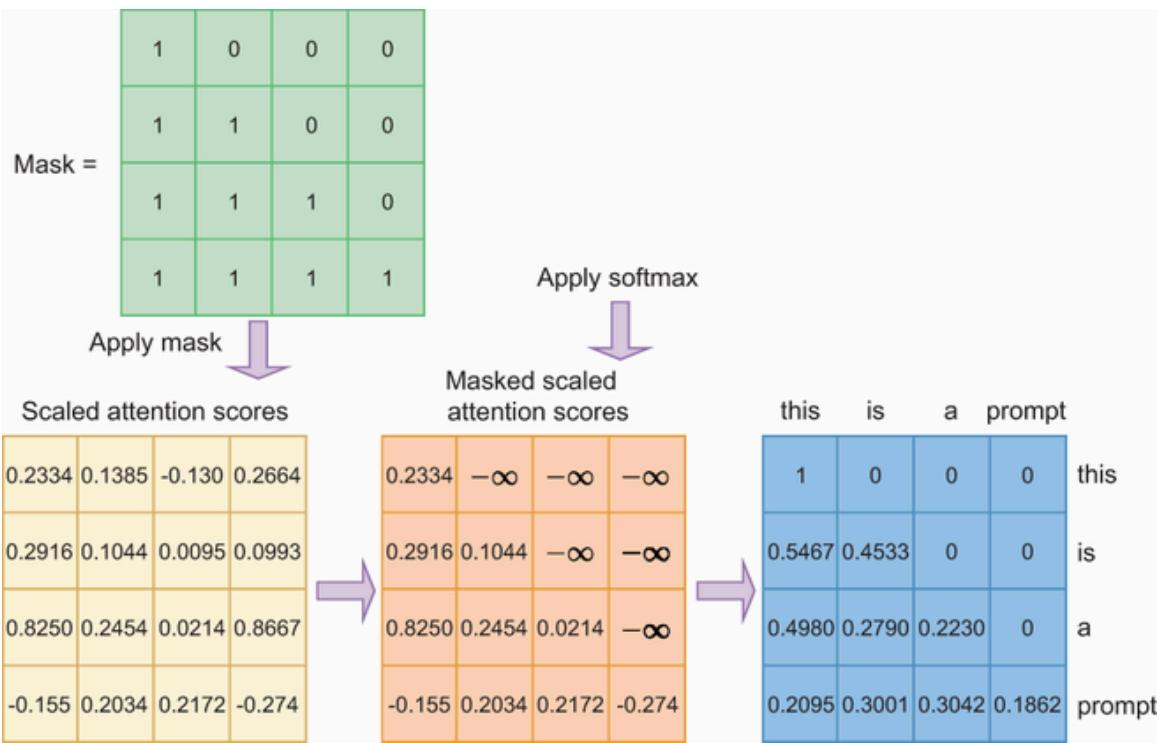


Figura 11.3 Cómo calcular los pesos de atención enmascarados en la autoatención causal. Se aplica una máscara a las puntuaciones de atención escaladas para que los valores correspondientes a los tokens futuros (aquellos por encima de la diagonal principal en el matriz) se convierte en $-\infty$. Luego, aplicamos la función softmax a las puntuaciones de atención escaladas enmascaradas y obtenemos los pesos de atención enmascarados. El enmascaramiento garantiza que las predicciones para un token dado solo puedan verse influenciadas por los tokens que lo preceden en la secuencia, no por tokens futuros. Esto es esencial para generar resultados de texto coherentes y contextualmente relevantes.

Si ejecuta el código anterior, verá el siguiente resultado:

```
tensor([[1., 0., 0., 0.],
       [1., 1., 0., 0.],
       [1., 1., 1., 0.],
       [1., 1., 1., 1.]])
tensor([[ 0.2334,      -inf,      -inf,      -inf],
       [ 0.2916,     0.1044,      -inf,      -inf],
       [ 0.8250,     0.2454,     0.0214,      -inf],
       [-0.1557,     0.2034,     0.2172, -0.2740]], grad_fn=<SelectBackward0>)
```

La máscara es una matriz de 4×4 , como se muestra en la parte superior de la figura 11.3. La mitad inferior de la máscara (valores por debajo de la diagonal principal) son unos, mientras que la mitad superior (valores por encima de la diagonal principal) son ceros. Al aplicar esta máscara a las puntuaciones de atención escaladas, los valores de la mitad superior de la matriz se convierten en $-\infty$ (parte inferior central de la figura 11.3). De esta manera, al aplicar la función softmax a las puntuaciones de atención escaladas, la mitad superior de la matriz de ponderaciones de atención se llena con ceros (parte inferior derecha de la figura 11.3).

```
import torch.nn.functional as F  
att = F.softmax(masked_scaled_att, dim=-1)  
print(att[0,0])
```

Imprimimos los pesos de atención en el primer cabezal con los siguientes valores:

```
tensor([[1.0000, 0.0000, 0.0000, 0.0000],  
       [0.5467, 0.4533, 0.0000, 0.0000],  
       [0.4980, 0.2790, 0.2230, 0.0000],  
       [0.2095, 0.3001, 0.3042, 0.1862]],  
      grad_fn=<SelectBackward0>)
```

La primera fila indica que, en el primer paso de tiempo, el token "this" solo se atiende a sí mismo y no a ningún token futuro. De igual manera, si observa la segunda fila, los tokens "this is" se atienden entre sí, pero no a los tokens futuros "a prompt".

NOTA: Los pesos de este ejemplo numérico no están entrenados, así que no los interprete literalmente. Los usamos como ejemplo para ilustrar cómo funciona la autoatención causal.

Ejercicio 11.2

Hemos impreso los pesos de atención en la primera cabeza. ¿Cómo se imprimen los pesos de atención en la última cabeza (es decir, la vigésimo quinta) ?

Finalmente, calculamos el vector de atención de cada cabeza como el producto escalar de los pesos de atención y el vector de valores. Los vectores de atención de las 25 cabezas se unen para formar un único vector de atención:

```
y=att@v  
y = y.transpose(1, 2).contiguous().view(B, T, C)  
print(y.shape)
```

La salida es

```
torch.Size([1, 4, 1600])
```

La salida final después de la autoatención causal es una matriz de 4×1600 , del mismo tamaño que la entrada a la subcapa de autoatención causal. Las capas del decodificador están diseñadas de tal manera que la entrada y la salida tienen las mismas dimensiones, y esto nos permite apilar muchas capas del decodificador juntas para aumentar la capacidad de representación del modelo y permitir la extracción jerárquica de características durante el entrenamiento.

11.2 Construyendo GPT-2XL desde cero

Ahora que comprende la arquitectura de GPT-2 y cómo funciona su ingrediente principal, la autoatención causal, creemos la versión más grande de GPT-2 desde cero.

En esta sección, aprenderá primero a usar el método de tokenización de subpalabras en GPT-2, el tokenizador del codificador de pares de bytes (BPE), para dividir el texto en tokens individuales. También aprenderá la función de activación GELU utilizada en la red de propagación hacia adelante en GPT-2. Después, codificará el mecanismo de autoatención causal y lo combinará con una red de propagación hacia adelante para formar un bloque decodificador. Finalmente, apilará 48 bloques decodificadores para crear el modelo GPT-2XL. El código de este capítulo está adaptado del excelente repositorio de GitHub de Andrej Karpathy (<https://github.com/karpathy/minGPT>). Le recomiendo leer el repositorio si desea profundizar en el funcionamiento de GPT-2.

11.2.1 Tokenización de BPE

GPT-2 utiliza un método de tokenización de subpalabras llamado codificador de pares de bytes (BPE), una técnica de compresión de datos adaptada para la tokenización de texto en tareas de PLN. Es especialmente conocida por su aplicación en el entrenamiento de LLM, como la serie GPT y BERT (Representaciones de Codificador Bidireccional a partir de Transformadores). El objetivo principal del BPE es codificar un fragmento de texto en una secuencia de tokens de forma que se equilibre el tamaño del vocabulario y la longitud del texto tokenizado.

La BPE funciona fusionando iterativamente el par de caracteres consecutivos más frecuentes de un conjunto de datos en un único token nuevo, sujeto a ciertas condiciones. Este proceso se repite hasta alcanzar el tamaño de vocabulario deseado o hasta que no se obtengan más fusiones. La BPE permite una representación eficiente del texto, equilibrando la tokenización a nivel de carácter y a nivel de palabra. Ayuda a reducir el tamaño del vocabulario sin aumentar

significativamente la longitud de la secuencia, lo cual es crucial para el rendimiento de los modelos de PLN.

En el capítulo 8, analizamos las ventajas y desventajas de los tres tipos de tokenización (a nivel de carácter, a nivel de palabra y a nivel de subpalabra). Además, implementamos un tokenizador a nivel de palabra desde cero en el capítulo 8 (y lo haremos de nuevo en el capítulo 12). Por lo tanto, en este capítulo, tomaremos prestado el método de tokenización directamente de OpenAI. El funcionamiento detallado de BPE queda fuera del alcance de este libro. Solo necesita saber que primero convierte el texto en tokens de subpalabra y luego los índices correspondientes.

Descargue el archivo `bpe.py` de Andrej Karpathy del repositorio de GitHub: <https://mng.bz/861B>, y el archivo se guardan en la carpeta `/utils/` de su ordenador. En este capítulo, lo usaremos como módulo local. Como explicó Andrej Karpathy en su repositorio de GitHub, el módulo se basa en la implementación de OpenAI (<https://mng.bz/E0lj>), pero se modificó ligeramente para facilitar su comprensión.

Para ver cómo el módulo `bpe.py` convierte texto en tokens y luego los indexa, probemos un ejemplo:

```
from utils.bpe import get_encoder  
  
example="This is the original text."  
bpe_encoder=get_encoder()  
response=bpe_encoder.encode_and_show_work(example)  
print(response["tokens"])
```

①
②
③

① El texto de una oración de ejemplo

② Instancia la clase `get_encoder()` del módulo `bpe.py`

③ Tokeniza el texto de ejemplo e imprime los tokens

La salida es

```
['This', ' is', ' the', ' original', ' text', '.']
```

El tokenizador BPE divide el texto de ejemplo "Este es el texto original" en seis tokens, como se muestra en la salida anterior. Tenga en cuenta que el tokeniza-

dor BPE no convierte letras mayúsculas en minúsculas. Esto conduce a una tokenización más significativa, pero también a un número mucho mayor de tokens únicos. De hecho, todas las versiones de los modelos GPT-2 tienen un tamaño de vocabulario de 50276, varias veces mayor que el tamaño del vocabulario de los capítulos anteriores

También podemos usar el módulo `bpe.py` para mapear tokens a índices:

```
print(response['bpe_idx'])
```

La salida es

```
[1212, 318, 262, 2656, 2420, 13]
```

La lista anterior contiene los seis índices correspondientes a los seis tokens en el texto de ejemplo “Este es el texto original”.

También podemos restaurar el texto en función de los índices:

```
from utils.bpe import BPETokenizer  
  
tokenizer = BPETokenizer() ①  
out=tokenizer.decode(torch.LongTensor(response['bpe_idx'])) ②  
print(out)
```

① Instancia la clase `BPETokenizer()` del módulo `bpe.py`

② Utiliza el tokenizador para restaurar texto en función de los índices

La salida del bloque de código anterior es

```
This is the original text.
```

Como puede ver, el tokenizador BPE ha restaurado el texto de ejemplo a su forma original.

Ejercicio 11.3

Utilice el tokenizador BPE para dividir la frase «este es un mensaje» en tokens. Después, asigne los tokens a los índices. Finalmente, restaure la frase basán-

11.2.2 La función de activación unitaria lineal del error gaussiano

La función de activación de la unidad lineal de error gaussiano (GELU) se utiliza en las subcapas de avance de cada bloque decodificador en GPT-2. GELU proporciona una combinación de propiedades de activación lineales y no lineales que han demostrado mejorar el rendimiento del modelo en tareas de aprendizaje profundo, en particular en PLN.

GELU ofrece una curva suave y no lineal que permite ajustes más sutiles durante el entrenamiento en comparación con otras funciones como la unidad lineal rectificada (ReLU). Esta suavidad ayuda a optimizar la red neuronal de forma más eficaz, ya que proporciona un gradiente más continuo para la retropropagación. Para comparar GELU con ReLU, nuestra función de activación de referencia, definamos primero una clase GELU():

```
class GELU(nn.Module):
    def forward(self, x):
        return 0.5*x*(1.0+torch.tanh(math.sqrt(2.0/math.pi)*\
            (x + 0.044715 * torch.pow(x, 3.0))))
```

La función ReLU no es diferenciable en todas partes, ya que presenta una curvatura. La función de activación GELU, en cambio, sí lo es y proporciona un mejor proceso de aprendizaje. A continuación, dibujamos una imagen de la función de activación GELU y la comparamos con ReLU.

Listado 11.2 Comparación de dos funciones de activación: GELU y ReLU

```
import matplotlib.pyplot as plt
import numpy as np

genu=GELU()
def relu(x):(1)
    y=torch.zeros(len(x))
    for i in range(len(x)):
        if x[i]>0:
            y[i]=x[i]
    return y
xs = torch.linspace(-6,6,300)
ys=relu(xs)
gs=genu(xs)
```

```

fig, ax = plt.subplots(figsize=(6,4),dpi=300)
plt.xlim(-3,3)
plt.ylim(-0.5,3.5)
plt.plot(xs, ys, color='blue', label="ReLU")          ②
plt.plot(xs, gs, "--", color='red', label="GELU")    ③
plt.legend(fontsize=15)
plt.xlabel("values of x")
plt.ylabel("values of  $\text{ReLU}(x)$  and  $\text{GELU}(x)$ ")
plt.title("The ReLU and GELU Activation Functions")
plt.show()

```

- ① Define una función para representar ReLU
- ② Traza la función de activación de ReLU en líneas sólidas
- ③ Traza la función de activación GELU en líneas discontinuas

Si ejecuta el bloque de código anterior, verá un gráfico como se muestra en la figura 11.4.

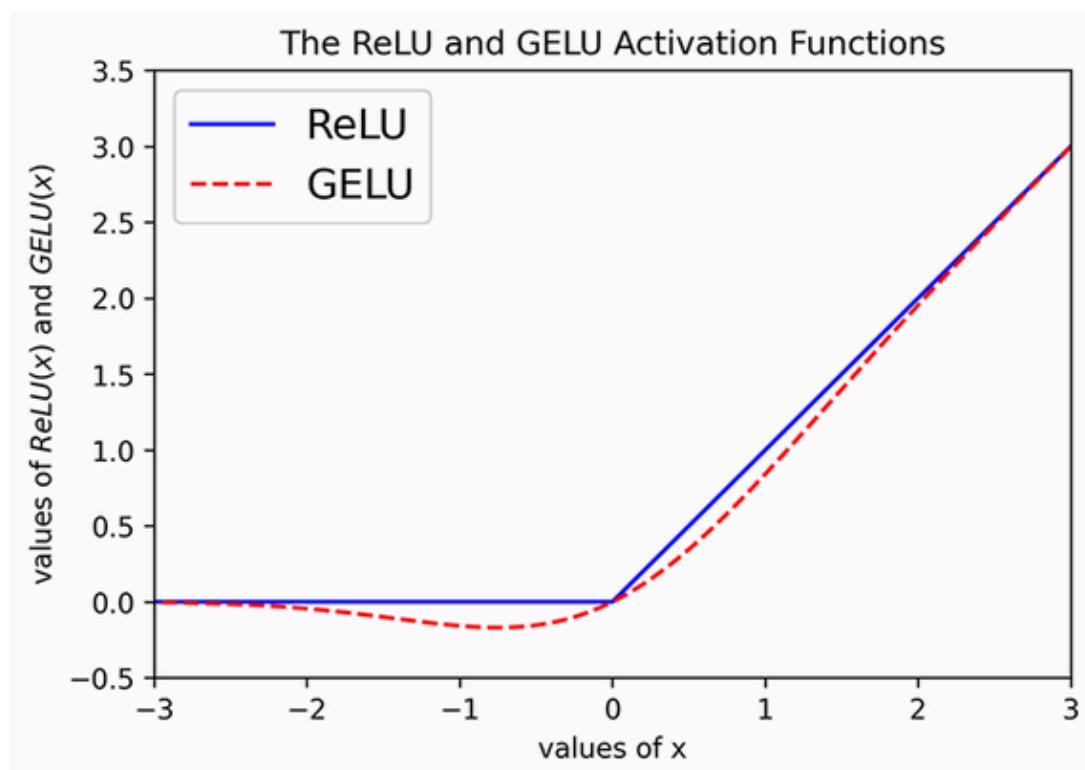


Figura 11.4 Comparación de la función de activación de GELU con ReLU. La línea continua representa la función de activación de ReLU, mientras que la línea discontinua representa la función de activación de GELU. ReLU no es diferenciable en todas partes, ya que presenta un punto de inflexión. GELU, en cambio, sí lo es. Esta suavidad en GELU ayuda a optimizar la red neuronal de forma más eficaz, ya que proporciona un gradiente más continuo para la retropropagación durante el proceso de entrenamiento.

Además, la formulación de GELU permite modelar las distribuciones de datos de entrada con mayor eficacia. Combina las propiedades del modelado de dis-

tribución lineal y gaussiana, lo que resulta especialmente beneficioso para los datos complejos y variados que se encuentran en las tareas de PLN. Esta capacidad ayuda a capturar patrones sutiles en los datos lingüísticos, mejorando la comprensión del modelo y la generación de texto.

11.2.3 Autoatención causal

Como explicamos anteriormente, la autoatención causal es el elemento central en los modelos GPT-2. A continuación, implementaremos este mecanismo desde cero en PyTorch

Primero especificamos los hiperparámetros en el modelo GPT-2XL que construiremos en este capítulo. Para ello, definimos una clase `Config()` con los valores que se muestran en la siguiente lista.

Listado 11.3 Especificación de hiperparámetros en GPT-2XL

```
class Config():
    def __init__(self):
        self.n_layer = 48
        self.n_head = 25
        self.n_embd = 1600
        self.vocab_size = 50257
        self.block_size = 1024
        self.embd_pdrop = 0.1
        self.resid_pdrop = 0.1
        self.attn_pdrop = 0.1

    config=Config()
```

① Define una clase `Config()`

② Coloca los hiperparámetros del modelo como atributos en la clase

③ Instancia la clase `Config()`

Definimos una clase `Config()` y creamos varios atributos en ella para ser usados como hiperparámetros en el modelo GPT-2XL. El atributo `n_layer` significa que el modelo GPT-2XL que construimos tendrá 48 capas decodificadoras (usamos los términos "bloque decodificador" y "capa decodificadora" indistintamente). El atributo `n_head` significa que dividiremos Q, K y V en 25 cabezas paralelas al calcular la autoatención causal. El atributo `n_embd` significa que la

dimensión de incrustación es 1,600: cada token será representado por un vector de 1,600 valores. El atributo `vocab_size` significa que hay 50,257 tokens únicos en el vocabulario. El atributo `block_size` significa que la secuencia de entrada al modelo GPT-2XL contiene como máximo 1,024 tokens. Las tasas de abandono están todas establecidas en 0.1.

En la sección anterior, expliqué en detalle cómo funciona la autoatención causal. A continuación, definimos una clase `CausalSelfAttention()` para implementarla.

Listado 11.4 Implementación de la autoatención causal

```
class CausalSelfAttention(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.c_attn = nn.Linear(config.n_embd, 3 * config.n_embd)
        self.c_proj = nn.Linear(config.n_embd, config.n_embd)
        self.attn_dropout = nn.Dropout(config.attn_pdrop)
        self.resid_dropout = nn.Dropout(config.resid_pdrop)
        self.register_buffer("bias", torch.tril(torch.ones(\ 
            config.block_size, config.block_size))
            .view(1, 1, config.block_size, config.block_size)) ①
        self.n_head = config.n_head
        self.n_embd = config.n_embd
    def forward(self, x):
        B, T, C = x.size()
        q, k ,v  = self.c_attn(x).split(self.n_embd, dim=2) ②
        hs = C // self.n_head
        k = k.view(B, T, self.n_head, hs).transpose(1, 2)
        q = q.view(B, T, self.n_head, hs).transpose(1, 2)
        v = v.view(B, T, self.n_head, hs).transpose(1, 2) ③
        att = (q @ k.transpose(-2, -1)) *\ 
            (1.0 / math.sqrt(k.size(-1)))
        att = att.masked_fill(self.bias[:,:,:,:T,:T] == 0, \
            float(' -inf'))
        att = F.softmax(att, dim=-1) ④
        att = self.attn_dropout(att)
        y = att @ v
        y = y.transpose(1, 2).contiguous().view(B, T, C) ⑤
        y = self.resid_dropout(self.c_proj(y))
        return y
```

① Crea una máscara y la registra como un búfer, ya que no necesita actualizarse

② Pasa la incrustación de entrada a través de tres redes neuronales para obtener Q, K y V

③ Divide Q, K y V en múltiples cabezas

④ Calcula los pesos de atención enmascarados en cada cabeza

⑤ Concatena vectores de atención en todas las cabezas en un único vector de atención

En PyTorch, `register_buffer` es un método que se utiliza para registrar un tensor como búfer. Las variables de un búfer no se consideran parámetros aprendibles del modelo; por lo tanto, no se actualizan durante la retropropagación. En el bloque de código anterior, creamos una máscara y la registramos como búfer. Esto tiene implicaciones en cómo extraemos y cargamos los pesos del modelo posteriormente: omitiremos las máscaras al recuperar los pesos de GPT-2XL.

Como explicamos en la primera sección, la incrustación de entrada se pasa a través de tres redes neuronales para obtener la consulta Q , la clave K y el valor V. Luego, las dividimos en 25 cabezas y calculamos la autoatención enmascarada en cada una. Después, unimos los 25 vectores de atención en un único vector de atención, que constituye la salida de la clase

`CausalSelfAttention()` anterior.

11.2.4 Construcción del modelo GPT-2XL

A continuación, agregamos una red de propagación hacia adelante a la subcapa de autoatención causal para formar un bloque decodificador, de la siguiente manera.

Listado 11.5 Construcción de un bloque decodificador

```
class Block(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.ln_1 = nn.LayerNorm(config.n_embd)
        self.attn = CausalSelfAttention(config)
        self.ln_2 = nn.LayerNorm(config.n_embd)
        self.mlp = nn.ModuleDict(dict(
            c_fc    = nn.Linear(config.n_embd, 4 * config.n_embd),
            c_proj = nn.Linear(4 * config.n_embd, config.n_embd),
```

```

        act      = GELU(),
        dropout = nn.Dropout(config.resid_pdrop),
    ))
m = self.mlp
self.mlpf=lambda x:m.dropout(m.c_proj(m.act(m.c_fc(x))))
def forward(self, x):
    x = x + self.attn(self.ln_1(x))                                (2)
    x = x + self.mlpf(self.ln_2(x))                                (3)
    return x

```

① Inicia la clase Block()

② La primera subcapa del bloque es la subcapa de autoatención causal, con normalización de capas y conexión residual.

③ La segunda subcapa del bloque es una red de propagación hacia adelante, con activación GELU, normalización de capas y conexión residual.

Cada bloque decodificador se compone de dos subcapas. La primera subcapa es el mecanismo de autoatención causal, que integra la normalización de capas y la conexión residual. La segunda subcapa dentro del bloque decodificador es la red de propagación hacia adelante, que incorpora la función de activación GELU, junto con la normalización de capas y la conexión residual.

Apilamos 48 capas de decodificador para formar el cuerpo principal del modelo GPT-2XL, como se muestra en la siguiente lista.

Listado 11.6 Construcción del modelo GPT-2XL

```

class GPT2XL(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.block_size = config.block_size
        self.transformer = nn.ModuleDict(dict(
            wte = nn.Embedding(config.vocab_size, config.n_embd),
            wpe = nn.Embedding(config.block_size, config.n_embd),
            drop = nn.Dropout(config.embd_pdrop),
            h = nn.ModuleList([Block(config)
                for _ in range(config.n_layer)]),
            ln_f = nn.LayerNorm(config.n_embd),))
        self.lm_head = nn.Linear(config.n_embd,
                               config.vocab_size, bias=False)
    def forward(self, idx, targets=None):
        b, t = idx.size()

```

```

pos = torch.arange(0,t,dtype=torch.long).unsqueeze(0)
tok_emb = self.transformer.wte(idx)
pos_emb = self.transformer.wpe(pos)                                         ①
x = self.transformer.drop(tok_emb + pos_emb)
for block in self.transformer.h:
    x = block(x)                                                       ②
x = self.transformer.ln_f(x)                                              ③
logits = self.lm_head(x)                                                 ④
loss = None
if targets is not None:
    loss=F.cross_entropy(logits.view(-1,logits.size(-1)),
                          targets.view(-1), ignore_index=-1)
return logits, loss

```

① Calcula la incrustación de entrada como la suma de la incrustación de palabras y la codificación posicional

② Pasa la incrustación de entrada a través de 48 bloques decodificadores

③ Aplica la normalización de capa una vez más

④ Conecta un cabezal lineal a la salida para que la cantidad de salidas sea igual a la cantidad de tokens únicos

Construimos el modelo en la clase `GPT2XL()` como explicamos en la primera sección de este capítulo. La entrada del modelo consiste en secuencias de índices correspondientes a los tokens del vocabulario. Primero, pasamos la entrada por la incrustación de palabras y la codificación posicional; luego, sumamos ambas para formar la incrustación de entrada. Esta incrustación pasa por 48 bloques decodificadores. Después, aplicamos la normalización de capas a la salida y le conectamos un cabezal lineal para que el número de salidas sea 50.257, el tamaño del vocabulario. Las salidas son los logits correspondientes a los 50.257 tokens del vocabulario. Posteriormente, aplicaremos la activación softmax a los logits para obtener la distribución de probabilidad sobre los tokens únicos del vocabulario al generar texto.

NOTA: Dado que el tamaño del modelo es demasiado grande, no lo trasladamos a una GPU. Esto reduce la velocidad de generación de texto, como veremos más adelante en este capítulo. Sin embargo, si tiene acceso a una GPU compatible con CUDA con gran capacidad de memoria (por ejemplo, superior a 32 GB), puede trasladar el modelo a una GPU para generar texto más rápido.

A continuación, crearemos el modelo GPT-2XL instanciando la clase `GPT2XL()` que definimos anteriormente:

```
model=GPT2XL(config)
num=sum(p.numel() for p in model.transformer.parameters())
print("number of parameters: %.2fM" % (num/1e6,))
```

También contamos el número de parámetros en el cuerpo principal del modelo. El resultado es

```
number of parameters: 1557.61M
```

La salida anterior muestra que GPT-2XL tiene más de 1500 millones de parámetros. Tenga en cuenta que esta cifra no incluye los parámetros del encabezado lineal al final del modelo. Dependiendo de la tarea posterior, podemos añadir diferentes encabezados al modelo. Dado que nos centramos en la generación de texto, hemos añadido un encabezado lineal para garantizar que el número de salidas sea igual al número de tokens únicos del vocabulario.

NOTA: En LLM como GPT-2, ChatGPT o BERT, un cabezal de salida se refiere a la capa final del modelo, responsable de producir la salida real a partir de la entrada procesada. Esta salida puede variar según la tarea posterior que realice el modelo. En la generación de texto, el cabezal de salida suele ser una capa lineal que transforma los estados ocultos finales en logits para cada token del vocabulario. Estos logits se pasan a través de una función softmax para generar una distribución de probabilidad sobre el vocabulario, que se utiliza para predecir el siguiente token de una secuencia. Para las tareas de clasificación, el cabezal de salida suele constar de una capa lineal seguida de una función softmax. La capa lineal transforma los estados ocultos finales del modelo en logits para cada clase, y la función softmax convierte estos logits en probabilidades para cada clase. La arquitectura específica del cabezal de salida puede variar según el modelo y la tarea, pero su función principal es asignar la entrada procesada al formato de salida deseado (p. ej., probabilidades de clase, probabilidades de token, etc.).

Finalmente, puedes imprimir la estructura del modelo GPT-2XL:

```
print(model)
```

La salida es

```
GPT2XL(  
    (transformer): ModuleDict(  
        (wte): Embedding(50257, 1600)  
        (wpe): Embedding(1024, 1600)  
        (drop): Dropout(p=0.1, inplace=False)  
        (h): ModuleList(  
            (0-47): 48 x Block(  
                (ln_1): LayerNorm((1600,), eps=1e-05,  
                    elementwise_affine=True)  
                (attn): CausalSelfAttention(  
                    (c_attn): Linear(in_features=1600, out_features=4800,  
                        bias=True)  
                    (c_proj): Linear(in_features=1600, out_features=1600,  
                        bias=True)  
                    (attn_dropout): Dropout(p=0.1, inplace=False)  
                    (resid_dropout): Dropout(p=0.1, inplace=False)  
                )  
                (ln_2): LayerNorm((1600,), eps=1e-05,  
                    elementwise_affine=True)  
                (mlp): ModuleDict(  
                    (c_fc): Linear(in_features=1600, out_features=6400,  
                        bias=True)  
                    (c_proj): Linear(in_features=6400, out_features=1600,  
                        bias=True)  
                    (act): GELU()  
                    (dropout): Dropout(p=0.1, inplace=False)  
                )  
            )  
        )  
        (ln_f): LayerNorm((1600,), eps=1e-05,  
            elementwise_affine=True)  
    )  
    (lm_head): Linear(in_features=1600, out_features=50257,  
        bias=False)  
)
```

Muestra los bloques y capas detallados en el modelo GPT-2XL.

¡Y así, has creado el modelo GPT-2XL desde cero!

11.3 Carga de pesos preentrenados y generación de texto

Aunque acaba de crear el modelo GPT-2XL, no está entrenado. Por lo tanto, no puede usarlo para generar texto con significado.

Dada la gran cantidad de parámetros del modelo, es imposible entrenarlo sin supercomputación, y mucho menos con la cantidad de datos necesaria. Afortunadamente, OpenAI publicó los pesos preentrenados de los modelos GPT-2, incluido el mayor, GPT-2XL, el 5 de noviembre de 2019 (véase la declaración en el sitio web de OpenAI, <https://openai.com/research/gpt-2-1-5b-release>, así como un informe del sitio web estadounidense de noticias tecnológicas The Verge, <https://mng.bz/NBm7>). Por lo tanto, cargaremos los pesos preentrenados para generar el texto de esta sección.

11.3.1 Carga de parámetros preentrenados en GPT-2XL

Utilizaremos la biblioteca `transformers` desarrollada por el equipo Hugging Face para extraer pesos preentrenados en GPT-2XL.

Primero, ejecute la siguiente línea de código en una nueva celda de este Jupyter Notebook para instalar la biblioteca `transformers` en su computadora:

```
!pip install transformers
```

A continuación, importamos el modelo GPT2 de la biblioteca `transformers` y extraemos los pesos preentrenados en GPT-2XL:

```
from transformers import GPT2LMHeadModel  
  
model_hf = GPT2LMHeadModel.from_pretrained('gpt2-xl')  
sd_hf = model_hf.state_dict()  
print(model_hf)
```

① Carga el modelo GPT-2XL preentrenado

② Extrae los pesos del modelo

③ Imprime la estructura del modelo del modelo original OpenAI GTP-2XL

La salida del bloque de código anterior es

```

GPT2LMHeadModel(
    (transformer): GPT2Model(
        (wte): Embedding(50257, 1600)
        (wpe): Embedding(1024, 1600)
        (drop): Dropout(p=0.1, inplace=False)
        (h): ModuleList(
            (0-47): 48 x GPT2Block(
                (ln_1): LayerNorm((1600,), eps=1e-05,
                    elementwise_affine=True)
                (attn): GPT2Attention(
                    (c_attn): Conv1D() (1)
                    (c_proj): Conv1D() (1)
                    (attn_dropout): Dropout(p=0.1, inplace=False)
                    (resid_dropout): Dropout(p=0.1, inplace=False)
                )
                (ln_2): LayerNorm((1600,), eps=1e-05,
                    elementwise_affine=True)
                (mlp): GPT2MLP(
                    (c_fc): Conv1D() (1)
                    (c_proj): Conv1D() (1)
                    (act): NewGELUActivation()
                    (dropout): Dropout(p=0.1, inplace=False)
                )
            )
        )
        (ln_f): LayerNorm((1600,), eps=1e-05,
            elementwise_affine=True)
    )
    (lm_head): Linear(in_features=1600, out_features=50257,
        bias=False)
)

```

① OpenAI utilizó una capa Conv1d en lugar de una capa lineal como hicimos nosotros

Si comparas la estructura de este modelo con la de la sección anterior, observarás que son iguales, salvo que las capas lineales se sustituyen por capas Conv1d. Como explicamos en los capítulos 9 y 10, en las redes de propagación hacia adelante, tratamos los valores de una entrada como elementos independientes en lugar de una secuencia. Por lo tanto, a menudo las llamamos redes convolucionales unidimensionales. Los puntos de control de OpenAI utilizan un módulo Conv1d en las zonas del modelo donde se utiliza una capa lineal. Por lo tanto, necesitamos transponer ciertas matrices de pesos al extraer los pesos del modelo de Hugging Face y colocarlos en nuestro modelo.

Para comprender cómo funciona esto, observemos los pesos en la primera capa de la red de propagación hacia adelante en el primer bloque decodificador del modelo OpenAI GPT-2XL. Podemos imprimir su forma de la siguiente manera:

```
print(model_hf.transformer.h[0].mlp.c_fc.weight.shape)
```

La salida es

```
torch.Size([1600, 6400])
```

La matriz de pesos en la capa Conv1d es un tensor con tamaño (1600, 6400).

Ahora, si observamos la misma matriz de peso en el modelo que acabamos de construir, su forma es

```
print(model.transformer.h[0].mlp.c_fc.weight.shape)
```

La salida esta vez es

```
torch.Size([6400, 1600])
```

La matriz de pesos de la capa lineal de nuestro modelo es un tensor con tamaño (6400, 1600), que es una matriz transpuesta de la matriz de pesos de OpenAI GPT-2XL. Por lo tanto, necesitamos transponer la matriz de pesos en todas las capas Conv1d del modelo OpenAI GPT-2XL antes de incorporarlas a nuestro modelo.

A continuación, nombramos los parámetros en el modelo original OpenAI GPT-2XL como `keys`:

```
keys = [k for k in sd_hf if not k.endswith('attn.masked_bias')]
```

Tenga en cuenta que hemos excluido los parámetros que terminan en `attn.masked_bias` en la línea de código anterior. OpenAI GPT-2 los utiliza para implementar el enmascaramiento de tokens en el futuro. Dado que hemos creado nuestro propio enmascaramiento en la clase `CausalSelfAttention()` y lo hemos registrado como un búfer en PyTorch, no necesitamos cargar parámetros que terminan en `attn.masked_bias` desde OpenAI.

Nombramos los parámetros en el modelo GPT-2XL que creamos desde cero como `sd`:

```
sd=model.state_dict()
```

A continuación, extraeremos los pesos preentrenados en OpenAI GPT-2XL y los colocaremos en nuestro propio modelo:

```
transposed = ['attn.c_attn.weight', 'attn.c_proj.weight',
              'mlp.c_fc.weight', 'mlp.c_proj.weight']          (1)
for k in keys:
    if any(k.endswith(w) for w in transposed):
        with torch.no_grad():
            sd[k].copy_(sd_hf[k].t())                  (2)
    else:
        with torch.no_grad():
            sd[k].copy_(sd_hf[k])                      (3)
```

① Descubre las capas en las que OpenAI usa un módulo Conv1d en lugar de un módulo lineal

② Para esas capas, transponemos la matriz de pesos antes de colocar los pesos en nuestro modelo.

③ De lo contrario, simplemente copia los pesos de OpenAI y colócalos en nuestro modelo

Extraemos los pesos preentrenados de OpenAI de Hugging Face y los incorporamos a nuestro propio modelo. Durante el proceso, nos aseguramos de transponer la matriz de pesos siempre que los puntos de control de OpenAI utilicen un módulo Conv1d en lugar de un módulo lineal simple.

Ahora nuestro modelo cuenta con pesos preentrenados de OpenAI. Podemos usarlo para generar texto coherente.

11.3.2 Definición de una función `generate()` para producir texto

Armados con pesos previamente entrenados del modelo OpenAI GPT-2XL, usaremos el modelo GPT2 que creamos desde cero para generar texto.

Al generar texto, alimentamos al modelo con una secuencia de índices correspondientes a tokens en una solicitud. El modelo predice el índice correspondiente al siguiente token y adjunta la predicción al final de la secuencia para formar una nueva secuencia. Luego, utiliza la nueva secuencia para realizar predicciones de nuevo. Continúa haciendo esto hasta que el modelo genera un número fijo de tokens nuevos o hasta que la conversación finaliza (indicado por el token especial `<| endoftext |>`).

El token especial `<| endoftext |>` en GPT

Los modelos GPT se entran con texto de diversas fuentes.

`<| endoftext |>` Durante esta fase, se utiliza un token único, , para delimitar el texto de diferentes orígenes. En la fase de generación de texto, es crucial determinar la conversación al encontrar este token especial. De lo contrario, se podría iniciar un nuevo tema sin relación, lo que resultaría en texto generado posteriormente sin relevancia para la conversación en curso.

Para ello, definimos una función `sample()` que añade un número determinado de índices nuevos a la secuencia actual. Esta función toma una secuencia de índices como entrada para alimentar el modelo GPT-2XL. Predice un índice a la vez y añade el nuevo índice al final de la secuencia en ejecución. Se detiene hasta que `max_new_tokens` se alcanza el número especificado de pasos de tiempo, o cuando el siguiente token predicho es `<| endoftext |>`, lo que indica el final de la conversación. Si no se detiene, el modelo iniciará aleatoriamente un tema no relacionado. La función `sample()` se define como se muestra en la siguiente lista.

Listado 11.7 Predicción iterativa del siguiente índice

```
model.eval()
def sample(idx, max_new_tokens, temperature=1.0, top_k=None):
    for _ in range(max_new_tokens):          ①
        if idx.size(1) <= config.block_size:
            idx_cond = idx
        else:
            idx_cond = idx[:, -config.block_size:]
    logits, _ = model(idx_cond)              ②
    logits = logits[:, -1, :] / temperature
    if top_k is not None:
        v, _ = torch.topk(logits, top_k)
        logits[logits < v[:, [-1]]] = -float('Inf') ③
    probs = F.softmax(logits, dim=-1)
```

```

        idx_next = torch.multinomial(probs, num_samples=1)
        if idx_next.item() == tokenizer.encoder['<|endoftext|>']:
            break
        idx = torch.cat((idx, idx_next), dim=1) ④
    return idx
⑤

```

- ① Genera un número fijo de nuevos índices
- ② Predice el próximo índice usando GPT-2XL
- ③ Si se utiliza el muestreo top-K, establece los logits debajo de las opciones top K en $-\infty$
- ④ Deja de predecir si el siguiente token es <| endoftext |>
- ⑤ Adjunta la nueva predicción al final de la secuencia

La función `sample()` utiliza GPT-2XL para añadir nuevos índices a una secuencia en ejecución. Incorpora dos argumentos, `temperature` y `top_k`, para modular la novedad de la salida generada, operando de la misma manera que se describe en el capítulo 8. La función devuelve una nueva secuencia de índices.

A continuación, definimos una función `generate()` para generar texto según una solicitud. Primero, convierte el texto de la solicitud en una secuencia de índices. Luego, envía la secuencia a la función `sample()` que acabamos de definir para generar una nueva secuencia de índices. Finalmente, la función `generate()` convierte la nueva secuencia de índices de nuevo en texto.

Listado 11.8 Una función para generar texto con GPT-2XL

```

def generate(prompt, max_new_tokens, temperature=1.0,
            top_k=None):
    if prompt == '':
        x = torch.tensor([[tokenizer.encoder[
            encoder['<|endoftext|>']]],
                         dtype=torch.long) ①
    else:
        x = tokenizer(prompt) ②
        y = sample(x, max_new_tokens, temperature, top_k) ③
        out = tokenizer.decode(y.squeeze()) ④
        print(out)

```

① Si el mensaje está vacío, utiliza <|endoftext|> como mensaje

② Convierte el mensaje en una secuencia de índices

③ Utiliza la función sample() para generar nuevos índices

④ Convierte la nueva secuencia de índices nuevamente a texto

La función `generate()` se asemeja a la versión que presentamos en el capítulo 8, pero con una diferencia notable: emplea GPT-2XL para la predicción, alejándose del modelo LSTM utilizado anteriormente. La función acepta una solicitud como entrada inicial y la transforma en una serie de índices que se introducen en el modelo para pronosticar el índice subsiguiente. Tras generar un número predeterminado de nuevos índices, la función revierte toda la secuencia de índices a formato textual.

11.3.3 Generación de texto con GPT-2XL

Ahora que hemos definido la función `generate()`, podemos usarla para generar texto.

En particular, la función `generate()` permite la generación de texto incondicional, lo que significa que el mensaje está vacío. El modelo generará texto aleatoriamente. Esto puede ser beneficioso en la escritura creativa: el texto generado puede usarse como inspiración o punto de partida para el propio trabajo creativo. Probémoslo:

```
prompt=""
torch.manual_seed(42)
generate(prompt, max_new_tokens=100, temperature=1.0,
         top_k=None)
```

La salida es

```
<|endoftext|>Feedback from Ham Radio Recalls
```

```
I discovered a tune sticking in my head --
I'd heard it mentioned on several occasions,
but hadn't investigated further.
```

```
The tune sounded familiar to a tune I'd previously
heard on the 550 micro. During that same time period
```

I've heard other people's recipients drone on the idea of the DSH-94013, notably Kim Weaver's instructions in her Interview on Radio Ham; and both Scott Mcystem and Steve Simmons' concepts.

Como puede ver, el resultado anterior es coherente y gramaticalmente correcto, pero puede que no sea exacto en cuanto a los hechos. Hice una búsqueda rápida en Google y el texto no parece estar copiado de ninguna fuente en línea

Ejercicio 11.4

Genere texto incondicionalmente estableciendo el mensaje como una cadena vacía, la temperatura en 0.9, el número máximo de tokens nuevos en 100 y `top_k` en 40. Establezca el número de semilla aleatoria en 42 en PyTorch. Observe cuál es el resultado

Para evaluar si GPT-2XL puede producir un texto coherente a partir de las fichas anteriores, usaremos la instrucción "Fui a la cocina y..." y generaremos 10 fichas adicionales después de la instrucción. Repetiremos este proceso cinco veces para determinar si el texto generado se corresponde con las actividades típicas de la cocina:

```
prompt="I went to the kitchen and"
for i in range(5):
    torch.manual_seed(i)
    generate(prompt, max_new_tokens=10, temperature=1.0,
              top_k=None)
```

La salida es

I went to the kitchen and said, you're
not going to believe this.
I went to the kitchen and noticed a female
producer open a drawer in which was
I went to the kitchen and asked who was
going to be right there and A
I went to the kitchen and took a small
vial of bourbon and a little
I went to the kitchen and found the bottle
of wine, and poured it into

Estos resultados indican que el texto generado incluye actividades como conversar con alguien, observar algo y tomar bebidas, todas ellas típicas de la cocina. Esto demuestra que GPT-2XL puede generar texto relevante para el contexto dado.

A continuación, usamos “Lexington es la segunda ciudad más grande del estado de Kentucky” como mensaje y le pedimos a la función `generate()` que agregue hasta 100 tokens nuevos:

```
prompt="Lexington is the second largest city  
in the state of Kentucky"  
torch.manual_seed(42)  
generate(prompt, max_new_tokens=100,  
temperature=1.0, top_k=None)
```

La salida es

Lexington is the second largest city in the state of Kentucky. It caters to those who want to make everything in tune with being with friends and enjoying a jaunt through the down to Earth lifestyle. To do so, they are blessed with several venues large and small to fill their every need while residing micro- cozy with nature within the landmarks of the city.
In a moment we look at ten up and coming suchache music acts from the Lexington area to draw upon your attention.

Lyrikhop

This Lexington-based group

Nuevamente, este texto es coherente. Aunque el contenido generado pueda no ser factualmente preciso, el modelo GPT-2XL está, fundamentalmente, entrenado para predecir el siguiente token basándose en los tokens anteriores en la oración. El resultado anterior muestra que el modelo ha logrado este objetivo: el texto generado es gramaticalmente correcto y aparentemente lógico. Demuestra la capacidad de recordar el texto en las primeras partes de la secuencia y generar palabras posteriores relevantes para el contexto. Por ejemplo, mientras que la primera oración habla de la ciudad de Lexington, unos 90 tokens después, el modelo menciona los espectáculos musicales de la zona.

Además, como se mencionó en la introducción, GPT-2 tiene sus limitaciones. No debe ajustarse al mismo estándar que ChatGPT o GPT-4, dado que su tamaño es inferior al 1% de ChatGPT y al 0,1% de GPT-4. GPT-3 tiene 175 mil millones de parámetros y produce texto más coherente que GPT-2, pero las ponderaciones preentrenadas no se publican.

A continuación, exploraremos cómo el muestreo `temperature` y `top-K` el texto generado por GPT-2XL afectan el texto generado. Estableceremos los valores `temperature` en 0.9 y `top_k` en 50, y mantendremos los demás argumentos sin cambios. Veamos cómo se ve el texto generado:

```
torch.manual_seed(42)  
generate(prompt, max_new_tokens=100, temperature=0.9,  
        top_k=50)
```

La salida es

Lexington is the second largest city in the state of Kentucky. It is also the state capital. The population of Lexington was 1,731,947 in the 2011 Census. The city is well-known for its many parks, including Arboretum, Zoo, Aquarium and the Kentucky Science Center, as well as its restaurants, such as the famous Kentucky Derby Festival.

In the United States, there are at least 28 counties in this state with a population of more than 100,000, according to the 2010 census.

El texto generado parece más coherente que antes. Sin embargo, el contenido no es exacto en cuanto a los hechos. Inventó muchos datos sobre la ciudad de Lexington, Kentucky, como «La población de Lexington era de 1 731 947 habitantes en el censo de 2011».

Ejercicio 11.5

Genere texto estableciendo `temperature` en 1.2 y `top_k` en Ninguno, y usando “Lexington es la segunda ciudad más grande del estado de Kentucky” como mensaje inicial. Establezca el número de semilla aleatoria en 42 en PyTorch y el número máximo de tokens nuevos en 100.

En este capítulo, aprendiste a construir GPT-2, el predecesor de ChatGPT y GPT-4, desde cero. Después, extrajiste los pesos preentrenados del modelo GPT-2XL publicado por OpenAI y los cargaste en tu modelo. Observaste el texto coherente generado por el modelo.

Debido al gran tamaño del modelo GPT-2XL (1500 millones de parámetros), es imposible entrenarlo sin supercomputación. En el siguiente capítulo, creará una versión reducida de un modelo GPT, con una estructura similar a la de GPT-2, pero con solo unos 5,12 millones de parámetros. Entrenará el modelo con el texto de las novelas de Ernest Hemingway. El modelo entrenado generará un texto coherente con un estilo similar al de Hemingway.

Resumen

- GPT-2 es un LLM avanzado desarrollado por OpenAI y anunciado en febrero de 2019. Representa un hito significativo en el campo del PLN y ha allanado el camino para el desarrollo de modelos aún más sofisticados, incluidos sus sucesores, ChatGPT y GPT-4
- GPT-2 es un Transformer basado únicamente en decodificadores, lo que significa que no incluye una pila de codificadores en el modelo. Al igual que otros modelos Transformer, GPT-2 utiliza mecanismos de autoatención para procesar los datos de entrada en paralelo, lo que mejora significativamente la eficiencia y la eficacia del entrenamiento de LLM.
- GPT-2 adopta un enfoque diferente para la codificación posicional que el utilizado en el artículo seminal de 2017 “Attention Is All You Need”. En cambio, la técnica de GPT-2 para la codificación posicional es paralela a la de las incrustaciones de palabras.
- La función de activación GELU se utiliza en las subcapas de avance de GPT-2. GELU proporciona una combinación de propiedades de activación lineales y no lineales que mejoran el rendimiento del modelo en tareas de aprendizaje profundo, especialmente en NLP y en el entrenamiento de LLM.
- Podemos construir un modelo GPT-2 desde cero e incorporar los pesos preentrenados publicados por OpenAI. El modelo GPT-2 que creaste puede generar texto coherente, igual que el modelo GPT-2 original de OpenAI.

12. Entrenando un transformador para generar texto

Este capítulo cubre

- Construcción de una versión reducida del modelo GPT-2XL adaptada a sus necesidades
- Preparación de datos para entrenar un transformador de estilo GPT
- Entrenando un transformador estilo GPT desde cero
- Generación de texto utilizando el modelo GPT entrenado

En el capítulo 11, desarrollamos el modelo GPT-2XL desde cero, pero no pudimos entrenarlo debido a su gran cantidad de parámetros. Entrenar un modelo con 1500 millones de parámetros requiere instalaciones de supercomputación y una enorme cantidad de datos. Por lo tanto, cargamos pesos preentrenados de OpenAI en nuestro modelo y luego usamos el modelo GPT-2XL para generar texto.

Sin embargo, aprender a entrenar un modelo Transformer desde cero es crucial por varias razones. En primer lugar, si bien este libro no aborda directamente el ajuste fino de un modelo preentrenado, comprender cómo entrenar un Transformer le proporciona las habilidades necesarias para el ajuste fino. Entrenar un modelo implica inicializar parámetros aleatoriamente, mientras que el ajuste fino implica cargar pesos preentrenados y seguir entrenando el modelo. En segundo lugar, entrenar o ajustar un Transformer le permite personalizar el modelo para satisfacer sus necesidades y dominio específicos, lo que puede mejorar significativamente su rendimiento y relevancia para su caso de uso. Finalmente, entrenar su propio Transformer o ajustar uno existente proporciona un mayor control sobre los datos y la privacidad, lo cual es particularmente importante para aplicaciones sensibles o el manejo de datos confidenciales. En resumen, dominar el entrenamiento y el ajuste fino de Transformers es esencial para cualquiera que busque aprovechar el poder de los modelos de lenguaje para aplicaciones específicas, manteniendo la privacidad y el control.

Por lo tanto, en este capítulo, construiremos una versión reducida del modelo GPT con aproximadamente 5 millones de parámetros. Este modelo más pequeño sigue la arquitectura del modelo GPT-2XL; las diferencias significativas residen en su composición de solo 3 bloques decodificadores y una dimensión de incrustación de 256, en comparación con los 48 bloques decodificadores y la dimensión de incrustación de 1600 del GPT-2XL original. Al reducir el modelo GPT a aproximadamente 5 millones de parámetros, podemos entrenarlo en un ordenador convencional.

El estilo del texto generado dependerá de los datos de entrenamiento. Al entrenar un modelo desde cero para la generación de texto, tanto la longitud como la variación del texto son cruciales. El material de entrenamiento debe ser lo suficientemente extenso como para que el modelo aprenda e imite un estilo de escritura particular de manera efectiva. Al mismo tiempo, si el material de entrenamiento carece de variación, el modelo puede simplemente replicar pasajes del texto de entrenamiento. Por otro lado, si el material es demasiado largo, el entrenamiento puede requerir recursos computacionales excesivos. Por lo tanto, utilizaremos tres novelas de Ernest Hemingway como nuestro material de entrenamiento: *El viejo y el mar*, *Adiós a las armas* y *Por quién doblan las campanas*. Esta selección garantiza que nuestros datos de entrenamiento tengan suficiente longitud y variación para un aprendizaje efectivo sin ser tan largos que el entrenamiento se vuelva impráctico.

Dado que los modelos GPT no pueden procesar texto sin procesar directamente, primero tokenizaremos el texto en palabras. Luego, crearemos un diccionario para asignar cada token único a un índice diferente. Con este diccionario, convertiremos el texto en una secuencia larga de enteros, lista para su entrada en una red neuronal.

Utilizaremos secuencias de 128 índices como entrada para entrenar el modelo GPT. Al igual que en los capítulos 8 y 10, desplazaremos la secuencia de entrada un token a la derecha y la utilizaremos como salida. Este enfoque obliga al modelo a predecir la siguiente palabra de una oración basándose en el token actual y todos los tokens anteriores de la secuencia.

Un desafío clave es determinar el número óptimo de épocas para entrenar el modelo. Nuestro objetivo no es simplemente minimizar la pérdida de entropía cruzada en el conjunto de entrenamiento, ya que esto podría provocar un sobreajuste, donde el modelo simplemente replica pasajes del texto de entrenamiento. Para abordar este problema, planeamos entrenar el modelo du-

rante 40 épocas. Guardaremos el modelo a intervalos de 10 épocas y evaluaremos qué versión puede generar texto coherente sin simplemente copiar pasajes del material de entrenamiento. Como alternativa, se podría usar un conjunto de validación para evaluar el rendimiento del modelo y decidir cuándo detener el entrenamiento, como hicimos en el capítulo 2.

Una vez entrenado nuestro modelo GPT, lo usaremos para generar texto autorregresivamente, como hicimos en el capítulo 11. Probaremos diferentes versiones del modelo entrenado. El modelo entrenado para 40 épocas produce un texto muy coherente, que captura el estilo distintivo de Hemingway. Sin embargo, también puede generar texto parcialmente copiado del material de entrenamiento, especialmente si la consigna es similar a pasajes del texto de entrenamiento. El modelo entrenado para 20 épocas también genera texto coherente, aunque con errores gramaticales ocasionales, pero es menos probable que copie directamente del texto de entrenamiento.

El objetivo principal de este capítulo no es necesariamente generar el texto más coherente posible, lo cual presenta desafíos significativos. Nuestro objetivo es enseñarle a construir un modelo de estilo GPT desde cero, adaptado a aplicaciones reales y a sus necesidades específicas. Más importante aún, este capítulo describe los pasos para entrenar un modelo GPT desde cero. Aprenderá a seleccionar el texto de entrenamiento según sus objetivos, tokenizarlo y convertirlo en índices, y preparar lotes de datos de entrenamiento. También aprenderá a determinar el número de épocas para el entrenamiento. Una vez entrenado el modelo, aprenderá a generar texto con él y a evitar generar texto copiado directamente del material de entrenamiento.

12.1 Creación y entrenamiento de un GPT desde cero

Nuestro objetivo es dominar la creación y el entrenamiento de un modelo GPT desde cero, adaptado a tareas específicas. Esta habilidad es crucial para aplicar los conceptos de este libro a problemas del mundo real.

Imagina que eres un gran admirador de la obra de Ernest Hemingway y deseas entrenar un modelo GPT para generar texto al estilo de Hemingway. ¿Cómo lo harías? Esta sección describe los pasos necesarios para esta tarea.

El primer paso es configurar un modelo GPT adecuado para el entrenamiento. Creará un modelo GPT con una estructura similar al modelo GPT-2 que creó en el capítulo 11, pero con muchos menos parámetros para que el

entrenamiento sea viable en tan solo unas horas. Por lo tanto, deberá determinar los hiperparámetros clave del modelo, como la longitud de la secuencia, la dimensión de incrustación, el número de bloques del decodificador y las tasas de abandono. Estos hiperparámetros son cruciales, ya que influyen tanto en la calidad del resultado del modelo entrenado como en la velocidad del entrenamiento.

A continuación, recopilará el texto original de varias novelas de Hemingway y lo depurará para garantizar que sea adecuado para el entrenamiento. Tokenizará el texto y asignará un entero diferente a cada token único para alimentarlo al modelo. Para preparar los datos de entrenamiento, descompondrá el texto en secuencias de enteros de cierta longitud y las usará como entradas. Luego, desplazará las entradas un token a la derecha y las usará como salidas. Este enfoque obliga al modelo a predecir el siguiente token basándose en el token actual y todos los tokens anteriores de la secuencia.

Una vez entrenado el modelo, lo usará para generar texto según una solicitud. Primero, convertirá el texto de la solicitud en una secuencia de índices y la alimentará al modelo entrenado. El modelo usa la secuencia para predecir iterativamente el siguiente token más probable. Después, convertirá la secuencia de tokens generada por el modelo de nuevo a texto.

En esta sección, primero analizaremos la arquitectura del modelo GPT para la tarea. Posteriormente, analizaremos los pasos necesarios para entrenar el modelo.

12.1.1 La arquitectura de una GPT para generar texto

Aunque GPT-2 está disponible en varios tamaños, todos comparten una arquitectura similar. El modelo GPT que construimos en este capítulo sigue el mismo diseño estructural que GPT-2, pero es significativamente más pequeño, lo que facilita su entrenamiento sin necesidad de supercomputación. La Tabla 12.1 presenta una comparación entre nuestro modelo GPT y las cuatro versiones de los modelos GPT-2.

Tabla 12.1 Una comparación de nuestro GPT con diferentes versiones de modelos GPT-2

	GPT-2S	GPT-2M	GPT-2L	GPT-2XL	Nuestro GPT
Dimensión de incrustación	768	1.024	1.280	1.600	256
Número de capas del decodificador	12	24	36	48	3
Número de cabezas	12	16	20	25	4
Longitud de la secuencia	1.024	1.024	1.024	1.024	128
Tamaño del vocabulario	50.257	50.257	50.257	50.257	10.600
Número de parámetros	124 millon es	350 millon es	774 millon es	1.558 millon es	5,12 millon es

En este capítulo, construiremos un modelo GPT con tres capas de decodificador y una dimensión de incrustación de 256 (lo que significa que cada token está representado por un vector de 256 valores después de la incrustación de palabras). Como mencionamos en el capítulo 11, los modelos GPT utilizan un método de codificación posicional diferente al utilizado en el artículo de 2017 “Attention Is All You Need”. En su lugar, utilizamos capas de incrustación para aprender las codificaciones posicionales para diferentes posiciones en una secuencia. Como resultado, cada posición en una secuencia también está representada por un vector de 256 valores. Para calcular la autoatención causal, utilizamos cuatro cabezas de atención paralelas para capturar diferentes aspectos de los significados de un token en la secuencia. Por lo tanto, cada cabeza de atención tiene una dimensión de $256/4 = 64$, similar a la de los modelos GPT-2. Por ejemplo, en GPT-2XL, cada cabeza de atención tiene una dimensión de $1600/25 = 64$.

La longitud máxima de secuencia en nuestro modelo GPT es de 128, mucho menor que la longitud máxima de secuencia de 1024 en los modelos GPT-2. Esta reducción es necesaria para mantener un número de parámetros manejable. Sin embargo, incluso con 128 elementos en una secuencia, el modelo puede aprender la relación entre los tokens y generar texto coherente.

Mientras que los modelos GPT-2 tienen un vocabulario de 50 257, nuestro modelo tiene un vocabulario mucho menor, de 10 600. Es importante tener en cuenta que el tamaño del vocabulario se determina principalmente por los datos de entrenamiento, y no es una opción predefinida. Si decide usar más texto para el entrenamiento, podría obtener un vocabulario más amplio.

La Figura 12.1 ilustra la arquitectura del Transformer, solo con decodificador, que crearemos en este capítulo. Es similar a la arquitectura de GPT-2 que se vio en el capítulo 11, excepto que es más pequeña. Como resultado, el número total de parámetros en nuestro modelo es de 5,12 millones, en comparación con los 1558 millones del modelo GPT-2XL que construimos en el capítulo 11. La Figura 12.1 muestra el tamaño de los datos de entrenamiento en cada paso del mismo.

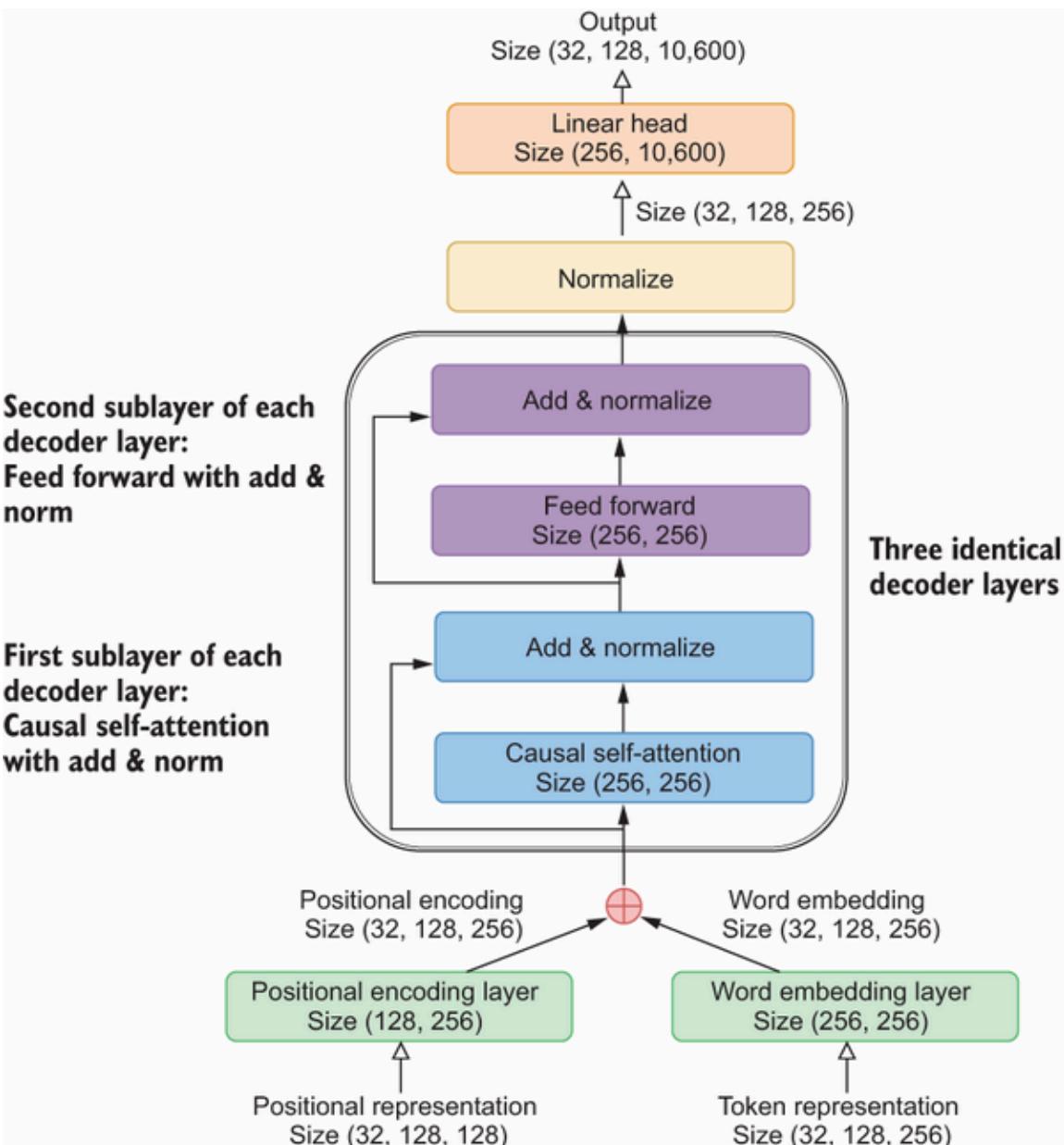


Figura 12.1 Arquitectura de un Transformer, solo con decodificador, diseñado para generar texto. El texto de tres novelas de Hemingway se tokeniza y luego se convierte en índices. Organizamos 128 índices en una secuencia, y cada lote contiene 32 de estas secuencias. La entrada primero se somete a la incrustación de palabras y la codificación posicional; la incrustación de entrada es la suma de estos dos componentes. Esta incrustación de entrada se procesa a través de tres capas del decodificador. A continuación, la salida se normaliza por capas y pasa por una capa lineal, lo que resulta en un tamaño de salida de 10 600, que corresponde al número de tokens únicos en el vocabulario.

La entrada del modelo GPT que creamos consiste en incrustaciones de entrada, que se ilustran en la parte inferior de la figura 12.1. Explicaremos cómo calcular estas incrustaciones en detalle en la siguiente subsección. En resumen, son la suma de las incrustaciones de palabras y las codificaciones posicionales de la secuencia de entrada.

La incrustación de entrada se pasa secuencialmente a través de tres capas de decodificación. De forma similar al modelo GPT-2XL que construimos en el capítulo 11, cada capa de decodificación consta de dos subcapas: una capa de autoatención causal y una red de retroalimentación. Además, aplicamos normalización de capas y conexiones residuales a cada subcapa. Después, la sa-

lida pasa por una normalización de capas y una capa lineal. El número de salidas en nuestro modelo GPT corresponde al número de tokens únicos en el vocabulario, que es 10 600. La salida del modelo son los logits del siguiente token. Posteriormente, aplicaremos la función softmax a estos logits para obtener la distribución de probabilidad sobre el vocabulario. El modelo está diseñado para predecir el siguiente token basándose en el token actual y todos los tokens anteriores en la secuencia.

12.1.2 El proceso de entrenamiento del modelo GPT para generar texto

Ahora que sabemos cómo construir el modelo GPT para la generación de texto, exploremos los pasos para entrenarlo. Nuestro objetivo es ofrecer una visión general del proceso de entrenamiento antes de profundizar en la codificación del proyecto.

El estilo del texto generado se ve influenciado por el texto de entrenamiento. Dado que nuestro objetivo es entrenar el modelo para generar texto al estilo de Ernest Hemingway, utilizaremos el texto de tres de sus novelas: *El viejo y el mar*, *Adiós a las armas* y *Por quién doblan las campanas*. Si eligiéramos solo una novela, los datos de entrenamiento carecerían de variedad, lo que provocaría que el modelo memorizara pasajes de la novela y generara texto idéntico a los datos de entrenamiento. Por el contrario, usar demasiadas novelas aumentaría el número de tokens únicos, lo que dificultaría entrenar el modelo eficazmente en poco tiempo. Por lo tanto, buscamos un equilibrio seleccionando tres novelas y combinándolas como datos de entrenamiento.

La figura 12.2 ilustra los pasos involucrados en el entrenamiento del modelo GPT para generar texto.

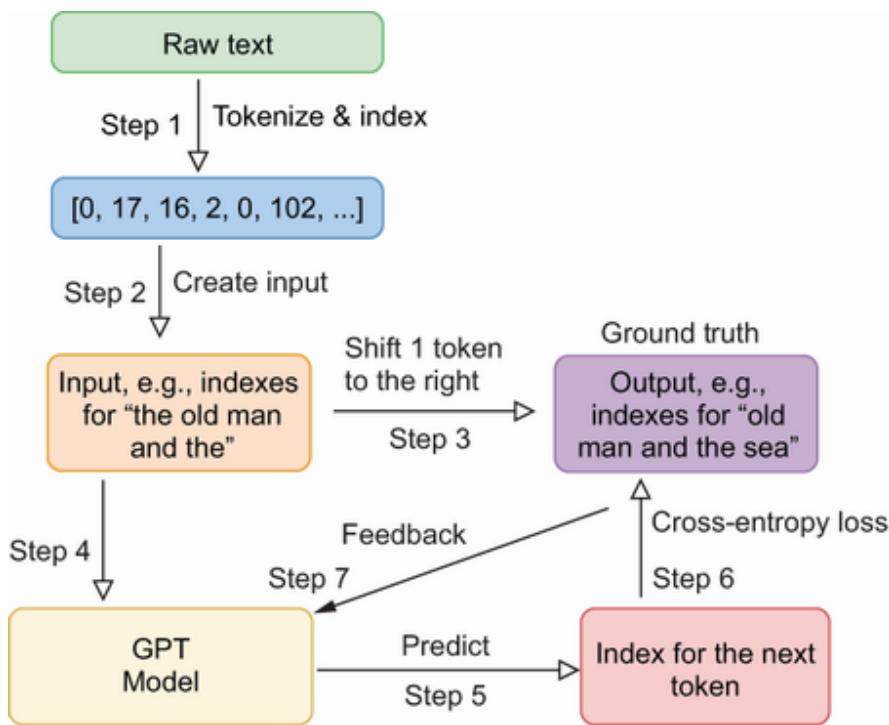


Figura 12.2 El proceso de entrenamiento de un Transformer solo decodificador para generar texto, al estilo Hemingway.

Al igual que en los tres capítulos anteriores, el primer paso del proceso de entrenamiento consiste en convertir el texto a formato numérico para poder alimentar el modelo con los datos de entrenamiento. En concreto, primero descomponemos el texto de las tres novelas en tokens mediante tokenización a nivel de palabra, como hicimos en el capítulo 8. En este caso, cada token es una palabra completa o un signo de puntuación (como dos puntos, un paréntesis o una coma). La tokenización a nivel de palabra es fácil de implementar y permite controlar el número de tokens únicos. Tras la tokenización, asignamos un índice único (es decir, un entero) a cada token, convirtiendo el texto de entrenamiento en una secuencia de enteros (véase el paso 1 en la figura 12.2).

A continuación, transformamos la secuencia de enteros en datos de entrenamiento dividiendo primero esta secuencia en secuencias de igual longitud (paso 2 en la figura 12.2). Permitimos una longitud máxima de 128 índices en cada secuencia. La elección de 128 nos permite capturar dependencias de largo alcance entre tokens en una oración, manteniendo al mismo tiempo un tamaño del modelo manejable. Sin embargo, el número 128 no es mágico: cambiarlo a, por ejemplo, 100 o 150, producirá resultados similares. Estas secuencias forman las características (la variable x) de nuestro modelo. Como hicimos en capítulos anteriores, desplazamos la secuencia de entrada un token a la derecha y la usamos como salida en los datos de entrenamiento (la variable y ; paso 3 en la figura 12.2).

Los pares de entrada y salida sirven como datos de entrenamiento (x, y). En el ejemplo de la oración "el viejo y el mar", usamos los índices correspondientes a "el viejo y el" como entrada x . Desplazamos la entrada un token a la derecha y usamos los índices de "el viejo y el mar" como salida y . En el primer paso de tiempo, el modelo usa "el" para predecir "viejo". En el segundo paso de tiempo, usa "el viejo" para predecir "hombre", y así sucesivamente.

Durante el entrenamiento, iterarás los datos de entrenamiento. En los pasos hacia adelante, introduces la secuencia de entrada x en el modelo GPT (paso 4). El GPT realiza una predicción basada en los parámetros actuales del modelo (paso 5). Calculas la pérdida de entropía cruzada comparando los siguientes tokens predichos con el resultado obtenido en el paso 3. En otras palabras, comparas la predicción del modelo con la realidad fundamental (paso 6). Finalmente, ajustas los parámetros del modelo GPT para que, en la siguiente iteración, las predicciones del modelo se acerquen más al resultado real, minimizando así la pérdida de entropía cruzada (paso 7). Ten en cuenta que el modelo está realizando esencialmente un problema de clasificación multicategoría: predice el siguiente token de entre todos los tokens únicos del vocabulario.

Repetirá los pasos 3 a 7 mediante varias iteraciones. Tras cada iteración, se ajustarán los parámetros del modelo para mejorar la predicción del siguiente token. Repetiremos este proceso durante 40 épocas y guardaremos el modelo entrenado cada 10. Como verá más adelante, si entrenamos el modelo durante demasiado tiempo, se sobreajustará y memorizará fragmentos de los datos de entrenamiento. El texto generado será entonces idéntico al de las novelas originales. Probaremos *ex post* qué versión del modelo genera texto coherente y, al mismo tiempo, no copia simplemente los datos de entrenamiento.

12.2 Tokenización del texto de las novelas de Hemingway

Ahora que comprende la arquitectura del modelo GPT y el proceso de entrenamiento, comenzemos con el primer paso: tokenizar e indexar el texto de las novelas de Hemingway.

Primero, procesaremos los datos de texto para prepararlos para el entrenamiento. Descompondremos el texto en tokens individuales, como hicimos en el capítulo 8. Dado que las redes neuronales profundas no pueden procesar

texto sin procesar directamente, crearemos un diccionario que asigne un índice a cada token, asignándolos a enteros. Después, organizaremos estos índices en lotes de datos de entrenamiento, lo cual será crucial para entrenar el modelo GPT en los pasos posteriores.

Utilizaremos la tokenización a nivel de palabra por su simplicidad al dividir el texto en palabras, a diferencia de la tokenización de subpalabras, más compleja, que requiere una comprensión detallada de la estructura lingüística. Además, la tokenización a nivel de palabra genera un menor número de tokens únicos que la tokenización de subpalabras, lo que reduce el número de parámetros en el modelo GPT.

12.2.1 Tokenización del texto

Para entrenar el modelo GPT, usaremos los archivos de texto sin procesar de tres novelas de Ernest Hemingway : *El viejo y el mar*, *Adiós a las armas* y *Por quién doblan las campanas*. Los archivos de texto se descargan del sitio web de Faded Page: <https://www.fadedpage.com>. He limpiado el texto eliminando los párrafos superior e inferior que no forman parte del libro original. Al preparar su propio texto de entrenamiento, es crucial eliminar toda la información irrelevante, como los detalles del proveedor, el formato y la información de la licencia. Esto garantiza que el modelo se centre únicamente en aprender el estilo de escritura presente en el texto. También he eliminado el texto entre capítulos que no es relevante para el texto principal. Puede descargar los tres archivos OldManAndSea.txt, FarewellToArms.txt y ToWhomTheBellTolls.txt del repositorio de GitHub del libro: <https://github.com/markhliu/DGAI>. Colóquelos en la carpeta /files/ de su computadora.

En el archivo de texto de *El viejo y el mar*, tanto la comilla doble inicial (") como la comilla doble de cierre (") se representan con comillas dobles rectas (""). Este no es el caso en los archivos de texto de las otras dos novelas. Por lo tanto, cargamos el texto de *El viejo y el mar* y cambiamos las comillas rectas por una cita inicial o una cita de cierre. Hacerlo nos permite diferenciar entre las comillas iniciales y las de cierre. Esto también ayudará a formatear el texto generado más adelante: eliminaremos el espacio después de la comilla inicial y el espacio antes de la comilla de cierre. Este paso se implementa como se muestra en la siguiente lista.

Listado 12.1 Cambio de comillas rectas a comillas de apertura y cierre

```

with open("files/OldManAndSea.txt", "r",
          encoding='utf-8-sig') as f:
    text=f.read()
text=list(text)
for i in range(len(text)):
    if text[i]=='''':
        if text[i+1]==' ' or text[i+1]=='\n':
            text[i]=''
    if text[i+1]!=' ' and text[i+1]!='\n':
        text[i]=''
    if text[i]=='''':
        if text[i-1]!=' ' and text[i-1]!='\n':
            text[i]=''
text=''.join(text)

```

- ① Carga el texto sin procesar y lo divide en caracteres individuales
- ② Si una comilla doble recta va seguida de un espacio o un salto de línea, la cambia a una comilla de cierre.
- ③ De lo contrario, cámbielo a una cita de apertura.
- ④ Convierte una comilla simple recta en un apóstrofo
- ⑤ Une caracteres individuales al texto

Si una comilla doble va seguida de un espacio o un salto de línea, la cambiaremos a una comilla de cierre; de lo contrario, a una comilla de apertura. El apóstrofo se introdujo como una comilla simple recta, y lo hemos cambiado a un apóstrofo en forma de comilla simple de cierre en el listado 12.1.

A continuación, cargamos el texto de las otras dos novelas y combinamos las tres novelas en un solo archivo.

Listado 12.2 Combinación del texto de tres novelas

```

with open("files/ToWhomTheBellTolls.txt", "r",
          encoding='utf-8-sig') as f:
    text1=f.read()

with open("files/FarewellToArms.txt", "r",
          encoding='utf-8-sig') as f:

```

```
text2=f.read()                                (2)

text=text+" "+text1+" "+text2                  (3)

with open("files/ThreeNovels.txt","w",
          encoding='utf-8-sig') as f:
    f.write(text)
print(text[:250])                            (4)
```

① Lee el texto de la segunda novela

② Lee el texto de la tercera novela

③ Combina el texto de las tres novelas

④ Guarda el texto combinado en la carpeta local

Cargamos el texto de las otras dos novelas, *Adiós a las Armas* y *Por Quién Doblan las Campanas*. Luego, combinamos el texto de las tres novelas para usarlo como datos de entrenamiento. Además, guardamos el texto combinado en un archivo local llamado ThreeNovels.txt para poder verificar posteriormente si el texto generado se copia directamente del texto original.

La salida del listado de códigos anterior es

```
He was an old man who fished alone in a skiff in the
Gulf Stream and he had gone eighty-four days now without
taking a fish. In the first forty days a boy had been
with him. But after forty days without a fish the boy's
parents had told him that th
```

La salida son los primeros 250 caracteres del texto combinado.

Tokenizaremos el texto usando un espacio como delimitador. Como se ve en el resultado anterior, los signos de puntuación como puntos (.), guiones (-) y apóstrofes (') se añaden a las palabras anteriores sin espacio. Por lo tanto, debemos insertar un espacio alrededor de todos los signos de puntuación.

Además, convertiremos los saltos de línea (\n) en espacios para que no se incluyan en el vocabulario. Convertir todas las palabras a minúsculas también es beneficioso en nuestro entorno, ya que garantiza que palabras como "The"

y "the" se reconozcan como el mismo token. Este paso ayuda a reducir el número de tokens únicos, lo que hace que el proceso de entrenamiento sea más eficiente. Para solucionar estos problemas, limpiaremos el texto como se muestra en la siguiente lista.

Listado 12.3 Agregar espacios alrededor de los signos de puntuación

```
text=text.lower().replace("\n", " ")                                ①

chars=set(text.lower())
punctuations=[i for i in chars if i.isalpha()==False
              and i.isdigit()==False]                           ②
print(punctuations)

for x in punctuations:
    text=text.replace(f"{{x}}", f" {x} ")
text_tokenized=text.split()                                         ③

unique_tokens=set(text_tokenized)
print(len(unique_tokens))                                         ④
```

① Reemplaza los saltos de línea con espacios

② Identifica todos los signos de puntuación

③ Inserta espacios alrededor de los signos de puntuación

④ Cuenta la cantidad de tokens únicos

Utilizamos el método `set()` para obtener todos los caracteres únicos del texto. Luego, utilizamos los métodos `isalpha()` y `isdigit()` para identificar y eliminar letras y números del conjunto de caracteres únicos, dejándonos solo con los signos de puntuación.

Si ejecuta el bloque de código anterior, el resultado será el siguiente:

```
[')', '.', '&', ':', '(', ';', '-', '!', "'", '"', ' ', '',
  "'", '?', ',', ''']  
10599
```

Esta lista incluye todos los signos de puntuación del texto. Añadimos espacios alrededor y dividimos el texto en tokens individuales mediante este método `split()`. El resultado indica que hay 10 599 tokens únicos en el texto de las tres novelas de Hemingway, un tamaño mucho menor que los 50 257 tokens de GPT-2. Esto reducirá significativamente el tamaño del modelo y el tiempo de entrenamiento.

Además, añadiremos un token más "UNK" para representar tokens desconocidos. Esto es útil en caso de que encontremos un prompt con tokens desconocidos, lo que nos permite convertirlos en un índice para alimentar al modelo. De lo contrario, solo podemos usar un prompt con los 10.599 tokens anteriores. Supongamos que incluye la palabra "tecnología" en el prompt. Dado que "tecnología" no es uno de los tokens en el diccionario `word_to_int`, el programa se bloqueará. Al incluir el token "UNK", puede evitar que el programa se bloquee en tales escenarios. Cuando entrena su propio GPT, siempre debe incluir el token "UNK", ya que es imposible incluir todos los tokens en su vocabulario. Para ello, añadimos "UNK" a la lista de tokens únicos y los asignamos a índices.

Listado 12.4 Asignación de tokens a índices

```
from collections import Counter

word_counts=Counter(text_tokenized)
words=sorted(word_counts, key=word_counts.get,
             reverse=True)
words.append("UNK")①
text_length=len(text_tokenized)
ntokens=len(words)②
print(f"the text contains {text_length} words")
print(f"there are {ntokens} unique tokens")
word_to_int={v:k for k,v in enumerate(words)}③
int_to_word={v:k for k,v in word_to_int.items()}④
print({k:v for k,v in word_to_int.items()
       if k in words[:10]})
print({k:v for k,v in int_to_word.items()
       if v in words[:10]})
```

① Agrega "UNK" a la lista de tokens únicos

② Cuenta el tamaño del vocabulario, `ntokens`, que será un hiperparámetro en nuestro modelo

③ Asigna tokens a índices

④ Asigna índices a tokens

La salida del bloque de código anterior es

```
the text contains 698207 words
there are 10600 unique tokens
{'.': 0, 'the': 1, ',': 2, "'": 3, '"': 4, 'and': 5,
'i': 6, 'to': 7, 'he': 8, 'it': 9}
{0: '.', 1: 'the', 2: ',', 3: "'", 4: '"', 5: 'and',
6: 'i', 7: 'to', 8: 'he', 9: 'it'}
```

El texto de las tres novelas contiene 698.207 tokens. Tras incluirlos "UNK" en el vocabulario, el número total de tokens únicos asciende a 10.600. El diccionario `word_to_int` asigna un índice diferente a cada token único. Por ejemplo, al token más frecuente, el punto (.), se le asigna un índice de 0, y a la palabra "the" se le asigna un índice de 1. El diccionario `int_to_word` traduce un índice de vuelta a un token. Por ejemplo, el índice 3 se traduce de vuelta a la comilla inicial ("), y el índice 4 se traduce de vuelta a la comilla final ("").

Imprimimos los primeros 20 tokens del texto y sus índices correspondientes:

```
print(text_tokenized[0:20])
wordidx=[word_to_int[w] for w in text_tokenized]
print([word_to_int[w] for w in text_tokenized[0:20]])
```

La salida es

```
['he', 'was', 'an', 'old', 'man', 'who', 'fished',
'alone', 'in', 'a', 'skiff', 'in', 'the', 'gulf',
'stream', 'and', 'he', 'had', 'gone', 'eighty']
[8, 16, 98, 110, 67, 85, 6052, 314, 14, 11, 1039,
14, 1, 3193, 507, 5, 8, 25, 223, 3125]
```

A continuación, dividiremos los índices en secuencias de igual longitud para usarlas como datos de entrenamiento.

12.2.2 Creación de lotes para entrenamiento

Usaremos una secuencia de 128 tokens como entrada del modelo. Luego, desplazaremos la secuencia un token a la derecha y la usaremos como salida.

Especificamente, creamos pares de (x, y) para fines de entrenamiento. Cada x es una secuencia con 128 índices. Elegimos 128 para lograr un equilibrio entre la velocidad de entrenamiento y la capacidad del modelo para capturar dependencias de largo alcance. Un valor demasiado alto puede ralentizar el entrenamiento, mientras que un valor demasiado bajo puede impedir que el modelo capture las dependencias de largo alcance eficazmente.

Una vez que tenemos la secuencia x , desplazamos la ventana de secuencia un token a la derecha y la usamos como el objetivo y . Desplazar la secuencia un token a la derecha y usarla como salida durante la generación de la secuencia es una técnica común en el entrenamiento de modelos de lenguaje, incluyendo GPT. Lo hemos hecho en los capítulos 8 a 10. El siguiente bloque de código crea los datos de entrenamiento:

```
import torch

seq_len=128
xys=[]
for n in range(0, len(wordidx)-seq_len-1):
    x = wordidx[n:n+seq_len]
    y = wordidx[n+1:n+seq_len+1]
    xys.append((torch.tensor(x), (torch.tensor(y))))
```

① Establece la longitud de la secuencia en 128 índices

② La secuencia de entrada x contiene 128 índices consecutivos en el texto de entrenamiento.

③ Desplaza x una posición a la derecha y lo utiliza como salida y

④ Agrega el par (x, y) a los datos de entrenamiento.

Hemos creado una lista xys para contener pares de (x, y) como datos de entrenamiento. Como hicimos en capítulos anteriores, organizamos los datos de entrenamiento en lotes para estabilizar el entrenamiento. Elegimos un tamaño de lote de 32:

```

from torch.utils.data import DataLoader

torch.manual_seed(42)
batch_size=32
loader = DataLoader(xys, batch_size=batch_size,
shuffle=True)

x,y=next(iter(loader))
print(x)
print(y)
print(x.shape,y.shape)

```

Imprimimos un par de x e y como ejemplo. El resultado es

```

tensor([[ 3, 129,    9, ...,   11, 251, 10],
       [ 5, 41, 32, ..., 995, 52, 23],
       [ 6, 25, 11, ..., 15, 0, 24],
       ...,
       [1254, 0, 4, ..., 15, 0, 3],
       [ 17, 8, 1388, ..., 0, 8, 16],
       [ 55, 20, 156, ..., 74, 76, 12]]))
tensor([[ 129, 9, 23, ..., 251, 10, 1],
       [ 41, 32, 34, ..., 52, 23, 1],
       [ 25, 11, 59, ..., 0, 24, 25],
       ...,
       [ 0, 4, 3, ..., 0, 3, 93],
       [ 8, 1388, 1, ..., 8, 16, 1437],
       [ 20, 156, 970, ..., 76, 12, 29]]))
torch.Size([32, 128]) torch.Size([32, 128])

```

Cada x e y tienen una forma de (32, 128). Esto significa que en cada lote de datos de entrenamiento hay 32 pares de secuencias, y cada secuencia contiene 128 índices. Al pasar un índice a través de la capa `nn.Embedding()`, PyTorch busca la fila correspondiente en la matriz de incrustación y devuelve el vector de incrustación para ese índice, evitando así la necesidad de crear vectores one-hot potencialmente muy grandes. Por lo tanto, al pasar x a través de la capa de incrustación de palabras, es como si se convirtiera primero en un tensor one-hot con una dimensión de (32, 128, 256). De forma similar, al pasar x a través de la capa de codificación posicional (implementada por la capa `nn.Embedding()`), es como si se convirtiera primero en un tensor one-hot con una dimensión de (32, 128, 128).

12.3 Creación de un GPT para generar texto

Ahora que tenemos los datos de entrenamiento listos, crearemos un modelo GPT desde cero para generar texto. El modelo que construiremos tiene una arquitectura similar a la del modelo GPT-2XL que construimos en el capítulo 11. Sin embargo, en lugar de tener 48 capas de decodificación, usaremos solo 3. Las dimensiones de incrustación y el tamaño del vocabulario son mucho menores, como expliqué anteriormente en este capítulo. Como resultado, nuestro modelo GPT tendrá muchos menos parámetros que GPT-2XL.

Seguiremos los mismos pasos que en el capítulo 11. A lo largo del camino, resaltaremos las diferencias entre nuestro modelo GPT y GPT-2XL y explicaremos las razones de estas modificaciones.

12.3.1 Hiperparámetros del modelo

La red de propagación hacia adelante del bloque decodificador utiliza la función de activación de la unidad lineal de error gaussiano (GELU). Se ha demostrado que GELU mejora el rendimiento del modelo en tareas de aprendizaje profundo, en particular en el procesamiento del lenguaje natural. Esto se ha convertido en una práctica estándar en los modelos GPT. Por lo tanto, definimos una clase GELU de la siguiente manera, como hicimos en el Capítulo 11:

```
import torch
from torch import nn
import math

device="cuda" if torch.cuda.is_available() else "cpu"
class GELU(nn.Module):
    def forward(self, x):
        return 0.5*x*(1.0+torch.tanh(math.sqrt(2.0/math.pi)*\
            (x + 0.044715 * torch.pow(x, 3.0))))
```

En el capítulo 11, no utilizamos una GPU ni siquiera durante la etapa de generación de texto, ya que el modelo era demasiado grande y una GPU normal se quedaría sin memoria si cargáramos el modelo en ella.

En este capítulo, sin embargo, nuestro modelo es significativamente más pequeño. Lo trasladaremos a la GPU para un entrenamiento más rápido. También generaremos texto usando el modelo en la GPU.

Utilizamos una clase `Config()` para incluir todos los hiperparámetros utilizados en el modelo:

```
class Config():
    def __init__(self):
        self.n_layer = 3
        self.n_head = 4
        self.n_embd = 256
        self.vocab_size = ntokens
        self.block_size = 128
        self.embd_pdrop = 0.1
        self.resid_pdrop = 0.1
        self.attn_pdrop = 0.1
config=Config()
```

Los atributos de la clase `Config()` se utilizan como hiperparámetros en nuestro modelo GPT. Establecemos el atributo `n_layer` en 3, lo que indica que nuestro modelo GPT tiene tres capas de decodificador. El atributo `n_head` se establece en 4, lo que significa que dividiremos los vectores de consulta Q , clave K y valor V en 4 cabezas paralelas al calcular la autoatención causal. El atributo `n_embd` se establece en 256, lo que significa que la dimensión de incrustación es 256: cada token estará representado por un vector de 256 valores. El atributo `vocab_size` se determina por el número de tokens únicos en el vocabulario. Como se explicó en la sección anterior, hay 10 600 tokens únicos en nuestro texto de entrenamiento. El atributo `block_size` se establece en 128, lo que significa que la secuencia de entrada contiene un máximo de 128 tokens. Establecemos las tasas de abandono en 0,1, como hicimos en el capítulo 11.

12.3.2 Modelado del mecanismo causal de autoatención

La autoatención causal se define de la misma manera que en el capítulo 11:

```
import torch.nn.functional as F
class CausalSelfAttention(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.c_attn = nn.Linear(config.n_embd, 3 * config.n_embd)
        self.c_proj = nn.Linear(config.n_embd, config.n_embd)
        self.attn_dropout = nn.Dropout(config.attn_pdrop)
        self.resid_dropout = nn.Dropout(config.resid_pdrop)
        self.register_buffer("bias", torch.tril(torch.ones(\
```

```

        config.block_size, config.block_size))
    .view(1, 1, config.block_size, config.block_size))
self.n_head = config.n_head
self.n_embd = config.n_embd

def forward(self, x):
    B, T, C = x.size()
    q, k ,v  = self.c_attn(x).split(self.n_embd, dim=2)
    hs = C // self.n_head
    k = k.view(B, T, self.n_head, hs).transpose(1, 2)
    q = q.view(B, T, self.n_head, hs).transpose(1, 2)
    v = v.view(B, T, self.n_head, hs).transpose(1, 2)

    att = (q @ k.transpose(-2, -1)) *\
        (1.0 / math.sqrt(k.size(-1)))
    att = att.masked_fill(self.bias[:, :, :T, :T] == 0, \
                           float(' -inf'))
    att = F.softmax(att, dim=-1)
    att = self.attn_dropout(att)
    y = att @ v
    y = y.transpose(1, 2).contiguous().view(B, T, C)
    y = self.resid_dropout(self.c_proj(y))
    return y

```

Al calcular la autoatención causal, la incrustación de entrada se pasa a través de tres redes neuronales para obtener la consulta Q , la clave K y el valor V. A continuación, dividimos cada una de ellas en cuatro núcleos paralelos y calculamos la autoatención enmascarada dentro de cada núcleo. Después, concatenamos los cuatro vectores de atención en un único vector de atención, que se utiliza como salida de la clase `CausalSelfAttention()`.

12.3.3 Construcción del modelo GPT

Combinamos una red de propagación hacia adelante con la subcapa causal de autoatención para formar un bloque decodificador. La red de propagación hacia adelante aporta no linealidad al modelo. Sin ella, el Transformador sería simplemente una serie de operaciones lineales, lo que limita su capacidad para capturar relaciones complejas de datos. Además, la red de propagación hacia adelante procesa cada posición de forma independiente y uniforme, lo que permite la transformación de las características identificadas por el mecanismo de autoatención. Esto facilita la captura de diversos aspectos de los datos de entrada, aumentando así la capacidad del modelo para representar información. Un bloque decodificador se define como sigue:

```

class Block(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.ln_1 = nn.LayerNorm(config.n_embd)
        self.attn = CausalSelfAttention(config)
        self.ln_2 = nn.LayerNorm(config.n_embd)
        self.mlp = nn.ModuleDict(dict(
            c_fc    = nn.Linear(config.n_embd, 4 * config.n_embd),
            c_proj = nn.Linear(4 * config.n_embd,
            config.n_embd),
            act     = GELU(),
            dropout = nn.Dropout(config.resid_pdrop),
        ))
        m = self.mlp
        self.mlpf=lambda x:m.dropout(m.c_proj(m.act(
            m.c_fc(x))))
    def forward(self, x):
        x = x + self.attn(self.ln_1(x))
        x = x + self.mlpf(self.ln_2(x))
        return x

```

Cada bloque decodificador de nuestro modelo GPT consta de dos subcapas: una subcapa de autoatención causal y una red de retroalimentación. Aplicamos normalización de capas y una conexión residual a cada subcapa para mejorar la estabilidad y el rendimiento. A continuación, superponemos tres capas decodificadoras para formar el cuerpo principal de nuestro modelo GPT.

Listado 12.5 Construcción de un modelo GPT

```

class Model(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.block_size = config.block_size
        self.transformer = nn.ModuleDict(dict(
            wte = nn.Embedding(config.vocab_size,
            config.n_embd),
            wpe = nn.Embedding(config.block_size,
            config.n_embd),
            drop = nn.Dropout(config.embd_pdrop),
            h = nn.ModuleList([Block(config)

```

```

        for _ in range(config.n_layer)]),
        ln_f = nn.LayerNorm(config.n_embd),))
    self.lm_head = nn.Linear(config.n_embd,
    config.vocab_size, bias=False)
    for pn, p in self.named_parameters():
        if pn.endswith('c_proj.weight'):
            torch.nn.init.normal_(p, mean=0.0,
            std=0.02/math.sqrt(2 * config.n_layer))

def forward(self, idx, targets=None):
    b, t = idx.size()
    pos=torch.arange(0,t,dtype=\
                    torch.long).unsqueeze(0).to(device) ①
    tok_emb = self.transformer.wte(idx)
    pos_emb = self.transformer.wpe(pos)
    x = self.transformer.drop(tok_emb + pos_emb)
    for block in self.transformer.h:
        x = block(x)
    x = self.transformer.ln_f(x)
    logits = self.lm_head(x)
    return logits

```

- ① Mueve la codificación posicional a la GPU habilitada para CUDA, si está disponible

La codificación posicional se crea dentro de la clase `Model()`. Por lo tanto, debemos transferirla a una GPU compatible con CUDA (si está disponible) para garantizar que todas las entradas del modelo estén en el mismo dispositivo. De lo contrario, se generará un mensaje de error.

La entrada del modelo consiste en secuencias de índices correspondientes a los tokens del vocabulario. Pasamos la entrada por incrustación de palabras y codificación posicional, y sumamos ambas para formar la incrustación de entrada. Esta incrustación pasa por los tres bloques decodificadores. Después, aplicamos la normalización de capas a la salida y le conectamos un cabezal lineal para que el número de salidas sea 10 600, el tamaño del vocabulario. Las salidas son los logits correspondientes a los 10 600 tokens del vocabulario. Posteriormente, aplicaremos la función de activación softmax a los logits para obtener la distribución de probabilidad de los tokens únicos del vocabulario al generar texto.

A continuación, crearemos nuestro modelo GPT instanciando la clase `Model()` que definimos anteriormente:

```

model=Model(config)
model.to(device)
num=sum(p.numel() for p in model.transformer.
parameters())
print("number of parameters: %.2fM" % (num/1e6,))
print(model)

```

La salida es

```

number of parameters: 5.12M
Model(
    (transformer): ModuleDict(
        (wte): Embedding(10600, 256)
        (wpe): Embedding(128, 256)
        (drop): Dropout(p=0.1, inplace=False)
        (h): ModuleList(
            (0-2): 3 x Block(
                (ln_1): LayerNorm((256,), eps=1e-05,
                    elementwise_affine=True)
                (attn): CausalSelfAttention(
                    (c_attn): Linear(in_features=256, out_features=768,
                        bias=True)
                    (c_proj): Linear(in_features=256, out_features=256,
                        bias=True)
                    (attn_dropout): Dropout(p=0.1, inplace=False)
                    (resid_dropout): Dropout(p=0.1, inplace=False)
                )
                (ln_2): LayerNorm((256,), eps=1e-05,
                    elementwise_affine=True)
                (mlp): ModuleDict(
                    (c_fc): Linear(in_features=256, out_features=1024,
                        bias=True)
                    (c_proj): Linear(in_features=1024, out_features=256,
                        bias=True)
                    (act): GELU()
                    (dropout): Dropout(p=0.1, inplace=False)
                )
            )
        )
    )
    (ln_f): LayerNorm((256,), eps=1e-05,
        elementwise_affine=True)
)
(lm_head): Linear(in_features=256, out_features=10600,

```

```
bias=False)  
)
```

Nuestro modelo GPT tiene 5,12 millones de parámetros. Su estructura es similar a la de GPT-2XL. Si compara el resultado anterior con el del capítulo 11, verá que las únicas diferencias residen en los hiperparámetros, como la dimensión de incrustación, el número de capas del decodificador, el tamaño del vocabulario, etc.

12.4 Entrenamiento del modelo GPT para generar texto

En esta sección, entrenará el modelo GPT que acaba de crear con los lotes de datos de entrenamiento que preparamos anteriormente en este capítulo. Una pregunta relacionada es cuántas épocas debemos entrenar el modelo. Si bien entrenar muy pocas épocas puede generar texto incoherente, entrenar demasiadas épocas puede generar un modelo sobreajustado, lo que puede generar texto idéntico a los pasajes del texto de entrenamiento.

Por lo tanto, entrenaremos el modelo durante 40 épocas. Guardaremos el modelo cada 10 épocas y evaluaremos qué versión del modelo entrenado puede generar texto coherente sin simplemente copiar fragmentos del texto de entrenamiento. Otro enfoque posible es crear un conjunto de validación y detener el entrenamiento cuando el rendimiento del modelo converja en dicho conjunto, como hicimos en el capítulo 2.

12.4.1 Entrenamiento del modelo GPT

Como siempre, usaremos el optimizador Adam. Dado que nuestro modelo GPT realiza esencialmente una clasificación multicategoría, usaremos la pérdida de entropía cruzada como función de pérdida:

```
lr=0.0001  
optimizer = torch.optim.Adam(model.parameters(), lr=lr)  
loss_func = nn.CrossEntropyLoss()
```

Entrenaremos el modelo durante 40 épocas, como se muestra en el siguiente listado.

```

model.train()
for i in range(1,41):
    tloss = 0.
    for idx, (x,y) in enumerate(loader):
        x,y=x.to(device),y.to(device)
        output = model(x)
        loss=loss_func(output.view(-1,output.size(-1)),
                       y.view(-1))          ②
    optimizer.zero_grad()
    loss.backward()
    nn.utils.clip_grad_norm_(model.parameters(),1) ③
    optimizer.step()                                ④
    tloss += loss.item()
    print(f'epoch {i} loss {tloss/(idx+1)}')
    if i%10==0:
        torch.save(model.state_dict(),f'files/GPTe{i}.pth') ⑤

```

- ① Itera a través de todos los lotes de datos de entrenamiento
- ② Compara las predicciones del modelo con los resultados reales
- ③ Recorta la norma del gradiente a 1
- ④ Ajusta los parámetros del modelo para minimizar la pérdida
- ⑤ Guarda el modelo después de cada diez épocas

Durante el entrenamiento, pasamos todas las secuencias de entrada x de un lote a través del modelo para obtener predicciones. Comparamos estas predicciones con las secuencias de salida y del lote y calculamos la pérdida de entropía cruzada. A continuación, ajustamos los parámetros del modelo para minimizar esta pérdida. Cabe destacar que hemos recortado la norma de gradiente a 1 para evitar el posible problema de gradientes explosivos.

Recorte de norma de gradiente

El recorte de la norma de gradiente es una técnica utilizada en el entrenamiento de redes neuronales para evitar el problema del gradiente explosivo. Este problema se produce cuando los gradientes de la función de pérdida con respecto a los parámetros del modelo se vuelven excesivamente grandes, lo que provoca un entrenamiento inestable y un rendimiento deficiente del modelo. En el recorte de la norma de gradiente, los gradientes se redu-

cen si su norma (magnitud) supera un umbral determinado. Esto garantiza que los gradientes no sean demasiado grandes, manteniendo un entrenamiento estable y mejorando la convergencia.

Este proceso de entrenamiento tarda un par de horas si tienes una GPU compatible con CUDA. Tras el entrenamiento, se guardarán en tu ordenador los cuatro archivos GPTe10.pth, GPTe20.pth, ..., GPTe40.pth. También puedes descargar los modelos entrenados desde mi sitio web:

<https://gattonweb.uky.edu/faculty/lium/gai/GPT.zip>.

12.4.2 Una función para generar texto

Ahora que tenemos varias versiones del modelo entrenado, podemos generar texto y comparar el rendimiento de las diferentes versiones. Podemos evaluar cuál tiene el mejor rendimiento y usarla para generar texto.

Similar al proceso en GPT-2XL, la generación de texto comienza introduciendo una secuencia de índices (que representan tokens) al modelo como solicitud. El modelo predice el índice del siguiente token, que luego se añade a la solicitud para formar una nueva secuencia. Esta nueva secuencia se reintroduce en el modelo para realizar predicciones adicionales, y este proceso se repite hasta generar el número deseado de tokens nuevos.

Para facilitar este proceso, definimos una función `sample()`. Esta función toma como entrada una secuencia de índices que representa el estado actual del texto. A continuación, predice iterativamente y añade nuevos índices a la secuencia hasta `max_new_tokens` alcanzar el número especificado de tokens nuevos. La siguiente lista muestra la implementación.

Listado 12.7 Una función `sample()` para predecir índices subsiguientes

```
def sample(idx, weights, max_new_tokens, temperature=1.0,
          top_k=None):
    model.eval()
    model.load_state_dict(torch.load(weights,
                                      map_location=device))
    original_length=len(idx[0])
    for _ in range(max_new_tokens):
        if idx.size(1) <= config.block_size:
            idx_cond = idx
        else:
```

```

        idx_cond = idx[:, -config.block_size:]
        logits = model(idx_cond.to(device))                                ③
        logits = logits[:, -1, :] / temperature
        if top_k is not None:
            v, _ = torch.topk(logits, top_k)
            logits[logits < v[:, [-1]]] = -float('Inf')
        probs = F.softmax(logits, dim=-1)
        idx_next=torch.multinomial(probs, num_samples=1)
        idx = torch.cat((idx, idx_next.cpu()), dim=1)                      ④
    return idx[:, original_length:]                                     ⑤

```

① Carga una versión del modelo entrenado

② Genera un número fijo de nuevos índices

③ Utiliza el modelo para hacer predicciones

④ Adjunta el nuevo índice al final de la secuencia

⑤ Solo genera los nuevos índices

Uno de los argumentos de la función `sample()` es `weights`, que representa los pesos entrenados de uno de los modelos guardados en su computadora. A diferencia de la función `sample()` que definimos en el capítulo 11, nuestra función aquí devuelve solo los índices recién generados, sin incluir los índices originales que se le proporcionaron `sample()`. Realizamos este cambio para adaptarnos a los casos en que el indicador contiene tokens desconocidos. En tales casos, nuestra función `sample()` garantiza que la salida final conserve el indicador original. De lo contrario, todos los tokens desconocidos se reemplazarían con `"UNK"` en la salida final.

A continuación, definimos una función `generate()` para generar texto a partir de una solicitud. La función primero convierte la solicitud en una secuencia de índices. Luego, la utiliza `sample()` para generar una nueva secuencia de índices. Después, la función `generate()` concatena todos los índices y los convierte de nuevo en texto. La implementación se muestra en la siguiente lista.

Listado 12.8 Una función para generar texto con el modelo GPT entrenado

```

UNK=word_to_int["UNK"]
def generate(prompt, weights, max_new_tokens, temperature=1.0,
            top_k=None):
    assert len(prompt)>0, "prompt must contain at least one token" ①
    text=prompt.lower().replace("\n", " ")
    for x in punctuations:
        text=text.replace(f"{{x}}", f" {x} ")
    text_tokenized=text.split()
    idx=[word_to_int.get(w,UNK) for w in text_tokenized]
    idx=torch.LongTensor(idx).unsqueeze(0)
    idx=sample(idx, weights, max_new_tokens,
               temperature=1.0, top_k=None)
    tokens=[int_to_word[i] for i in idx.squeeze().numpy()]
    text=" ".join(tokens)
    for x in '""').:;!?,-'":
        text=text.replace(f" {{x}}", f"{{x}}")
    for x in '""(-'":
        text=text.replace(f"{{x}} ", f"{{x}}")
    return prompt+" "+text

```

① Se asegura de que el mensaje no esté vacío

② Convierte el mensaje en una secuencia de índices

③ Utiliza la función sample() para generar nuevos índices

④ Convierte la nueva secuencia de índices nuevamente a texto

Nos aseguramos de que el mensaje no esté vacío. De lo contrario, recibirá un mensaje de error que indica que "el mensaje debe contener al menos un token". La función `generate()` le permite seleccionar la versión del modelo que desea usar especificando los pesos guardados en su computadora. Por ejemplo, puede elegir "files/GPTe10.pth" como valor del argumento de pesos de la función. La función convierte el mensaje en una serie de índices, que luego se introducen en el modelo para predecir el siguiente índice. Tras generar un número fijo de nuevos índices, la función convierte toda la secuencia de índices a formato textual.

12.4.3 Generación de texto con diferentes versiones del modelo entrenado

A continuación, experimentaremos con diferentes versiones del modelo entrenado para generar texto.

Podemos usar el token desconocido "UNK" como indicador para la generación de texto incondicional. Esto es especialmente beneficioso en nuestro contexto, ya que queremos comprobar si el texto generado se copia directamente del texto de entrenamiento. Si bien un indicador único y muy diferente del texto de entrenamiento es poco probable que conduzca a pasajes directamente del texto de entrenamiento, es más probable que el texto generado incondicionalmente provenga del texto de entrenamiento.

Primero usamos el modelo después de 20 épocas de entrenamiento para generar texto incondicionalmente:

```
prompt="UNK"
for i in range(10):
    torch.manual_seed(i)
    print(generate(prompt, 'files/GPTe20.pth',
    max_new_tokens=20)[4:]))
```

La salida es

way." "kümmel," i said. "it's the way to talk about it," robert jordan said. "but do not realize how far he is ruined." "pero in the fog, robert jordan thought. and then, without looking at last, so good, he pot of yellow rice and fish and the boy loved him. "no," the boy said. the line now. it's wonderful." "he's crazy about the brave." candle to us. "and if the maria kisses thee again i will commence kissing thee myself. it?" "do you have to for the moment." robert jordan got up and walked away in. a uniform for my father, he thought. i'll say them later. just then he and more practical to read and relax in the evening; of all the things he had enjoyed the next in bed and rolled himself a cigarette. when he gave them a log to a second grenade."

Configuramos el indicador "UNK" y solicitamos a la función `generate()` que genere incondicionalmente 20 tokens nuevos 10 veces. Usamos este método `manual_seed()` para corregir las semillas aleatorias y así poder reproducir los resultados. Como puede ver, los 10 pasajes cortos generados aquí son gramaticalmente correctos y parecen pasajes de las novelas de Hemingway. Por ejemplo, la palabra "kummel" en el primer pasaje era un tipo de licor que se mencionaba con frecuencia en "*Adiós a las Armas*". Al mismo tiempo, ninguno de los 10 pasajes anteriores se copió directamente del texto de entrenamiento.

A continuación, utilizamos el modelo después de 40 épocas de entrenamiento en lugar de generar texto incondicionalmente y vemos qué sucede:

```
prompt="UNK"
for i in range(10):
    torch.manual_seed(i)
    print(generate(prompt,'files/GPTe40.pth',
    max_new_tokens=20)[4:]))
```

La salida es

```
way." "kümmel, and i will enjoy the killing. they must
have brought me a spit," robert jordan said. "but do
not tell me that he saw anything." "not in the first
time he had bit the ear like that and held onto it,
his neck and jaws pot of yellow rice with fish. it was
cold now in the head and he could not see the the line
of his mouth. he thought." "the laughing hurt him." "i
can candle made? that was the worst day of my life until
one other day." "don'?" "do you have to for the moment."
robert jordan took the glasses and opened the. that's
what they don't marry." i reached for her hand. "don
and more grenades. that was the last for next year.
it crossed the river away from the front in a revolutionary
army," robert jordan said. "that's really nonsense. it's
```

Los 10 breves pasajes generados aquí son, una vez más, gramaticalmente correctos y parecen pasajes de las novelas de Hemingway. Sin embargo, si los examina con atención, gran parte del octavo pasaje está copiado directamente de la novela "*Adiós a las armas*". Esta parte `they don't marry."` `i reached for her hand.` "don" también aparece en la novela. Puede compro-

barlo buscando en el archivo ThreeNovels.txt que guardó previamente en su ordenador.

Ejercicio 12.1

Genere un pasaje de texto con 50 tokens nuevos incondicionalmente utilizando el modelo entrenado durante 10 épocas. Establezca la semilla aleatoria en 42 y mantenga la `temperature` muestreo `top-K` predeterminado. Examine si el pasaje generado es gramaticalmente correcto y si alguna parte se copió directamente del texto de entrenamiento.

Como alternativa, puedes usar una instrucción única que no esté en el texto de entrenamiento para generar texto nuevo. Por ejemplo, podrías usar "el anciano vio al tiburón cerca de" como instrucción y pedirle a la función `generate()` que agregue 20 tokens nuevos a la instrucción, repitiendo este proceso 10 veces:

```
prompt="the old man saw the shark near the"
for i in range(10):
    torch.manual_seed(i)
    print(generate(prompt, 'files/GPTe40.pth',
                  max_new_tokens=20))
    print("-"*50)
```

La salida es

```
the old man saw the shark near the old man's head with his
tail out and the old man hit him squarely in the center of
the old man saw the shark near the boat with one hand. he had
no feeling of the morning but he started to pull on it gently
the old man saw the shark near the old man's head. then he
went back to another man in and leaned over and dipped the
the old man saw the shark near the fish now, and the old
man was asleep in the water as he rowed he was out of the
the old man saw the shark near the boat. it was a nice-boat.
he saw the old man's head and he started the old man saw the
shark near the boat to see him clearly and he was afraid that
he was higher out of the water and the old the old man saw
the shark near the old man's head and then, with his tail
lashing and his jaws clicking, the shark plowed the old man
saw the shark near the line with his tail which was not sweet
smelling it. the old man knew that the fish was coming the
```

```
old man saw the shark near the fish with his jaws hooked and  
the old man stabbed him in his left eye. the shark still hung  
the old man saw the shark near the fish and he started to shake  
his head again. the old man was asleep in the stern and he
```

El texto generado es gramaticalmente correcto y coherente, y se asemeja mucho a pasajes de la novela de Hemingway, *El viejo y el mar*. Dado que utilizamos el modelo entrenado para 40 épocas, existe una mayor probabilidad de generar un texto que refleje directamente los datos de entrenamiento. Sin embargo, usar una indicación única puede reducir esta probabilidad.

Al configurar `temperature` y usar el muestreo `top-K`, podemos controlar aún más la diversidad del texto generado. En este caso, con un mensaje como "el anciano vio al tiburón cerca de..." y una temperatura de 0,9 con un muestreo top-50, el resultado se mantiene prácticamente gramaticalmente correcto:

```
prompt="the old man saw the shark near the"  
for i in range(10):  
    torch.manual_seed(i)  
    print(generate(prompt,'files/GPTe20.pth',  
    max_new_tokens=20, temperature=0.9,top_k=50))  
    print("-"*50)
```

La salida es

```
The old man saw the shark near the boat. then he swung  
the great fish that was more comfortable in the sun.  
the old man could the old man saw the shark near the  
boat with one hand. he wore his overcoat and carried  
the submachine gun muzzle down, carrying it in the old  
man saw the shark near the boat with its long dip sharply  
and the old man stabbed him in the morning. he could not  
see the old man saw the shark near the fish that was now  
heavy and long and grave he had taken no part in. he was  
still under the old man saw the shark near the boat. it  
was a nice little light. then he rowed out and the old  
man was asleep over the old man saw the shark near the  
boat to come. "old man's shack and i'll fill the water  
with him in the old man saw the shark near the boat and  
then rose with his lines close him over the stern. "no,"  
the oldman the old man saw the shark near the line with
```

his tail go under. he was cutting away onto the bow and his face was just a the old man saw the shark near the fish with his tail that he swung him in. the shark's head was out of water and the old man saw the shark near the boat and he started to cry. he could almost have them come down and whipped him in again.

Dado que utilizamos el modelo entrenado para 20 épocas en lugar de 40, el resultado es menos coherente y presenta errores gramaticales ocasionales. Por ejemplo, la frase «with its long dip sharply» (con su larga caída bruscamente) en el tercer pasaje no es gramaticalmente correcta. Sin embargo, el riesgo de generar texto copiado directamente de los datos de entrenamiento también es menor.

Ejercicio 12.2

Genere un pasaje de texto con 50 tokens nuevos utilizando el modelo entrenado para 40 épocas. Use "el anciano vio al tiburón cerca de" como instrucción; establezca "" random seed en 42, "" temperature en 0.95 y "" top_k en 100. Compruebe si el pasaje generado es gramaticalmente correcto y si alguna parte del texto se copió directamente del texto de entrenamiento.

En este capítulo, aprendiste a construir y entrenar un modelo Transformer de estilo GPT desde cero. En concreto, creaste una versión simplificada del modelo GPT-2 con solo 5,12 millones de parámetros. Utilizando tres novelas de Ernest Hemingway como datos de entrenamiento, entrenaste el modelo con éxito. También generaste un texto coherente y estilísticamente consistente con la obra de Hemingway.

Resumen

- El estilo del texto generado a partir de un modelo GPT se verá muy influenciado por los datos de entrenamiento. Para una generación de texto eficaz, es importante lograr un equilibrio entre la longitud del texto y la variación del material de entrenamiento. El conjunto de datos de entrenamiento debe ser lo suficientemente grande como para que el modelo aprenda y emule un estilo de escritura específico con precisión. Sin embargo, si el conjunto de datos carece de diversidad, el modelo podría terminar reproduciendo pasajes directamente del texto de entrenamiento. Por el contrario, conjuntos de datos de entrenamiento demasiado largos pueden requerir recursos computacionales excesivos para el entrenamiento.
- Elegir los hiperparámetros correctos en el modelo GPT es crucial para el éxito del entrenamiento del modelo y la generación de texto. Configurar hiperparámetros demasiado grandes puede resultar en un exceso de parámetros. Esto resulta en un mayor tiempo de entrenamiento y un modelo sobreajustado. Configurar hiperparámetros demasiado pequeños puede dificultar el aprendizaje eficaz del modelo y capturar el estilo de escritura en los datos de entrenamiento. Esto puede generar texto generado incoherente.
- El número adecuado de épocas de entrenamiento es importante para la generación de texto. Si bien entrenar con muy pocas épocas puede generar un texto incoherente, entrenar con demasiadas épocas puede generar un modelo sobreajustado que genere texto idéntico a los pasajes del texto de entrenamiento.

Parte 4. Aplicaciones y nuevos desarrollos

Esta parte cubre algunas aplicaciones de los modelos generativos de capítulos anteriores, así como algunos nuevos desarrollos en el campo de la IA generativa.

En los capítulos 13 y 14, aprenderá dos maneras de generar música: MuseGAN, que trata una pieza musical como un objeto multidimensional similar a una imagen, y Music Transformer, que trata una pieza musical como una secuencia de eventos musicales. El capítulo 15 le presenta los modelos de difusión, que constituyen la base de todos los principales transformadores de texto a imagen (como DALL-E 2 o Imagen). El capítulo 16 utiliza la biblioteca LangChain para combinar modelos de lenguaje extensos preentrenados con las API de Wolfram Alpha y Wikipedia para crear un asistente personal inteligente y de cero disparos.

14. Construcción y entrenamiento de un Transformer musical

Este capítulo cubre

- Representación de música con mensajes de control y valores de velocidad
- Tokenizar la música en una secuencia de índices
- Construcción y entrenamiento de un Transformer musical
- Generando eventos musicales usando el Transformer entrenado
- Convertir eventos musicales a un archivo MIDI reproducible

¿Triste porque tu músico favorito ya no está con nosotros? ¡Ya no te preocupes más: la IA generativa puede traerlo de vuelta al escenario!

Tomemos como ejemplo Layered Reality, una empresa londinense que trabaja en un proyecto llamado Elvis Evolution.¹ ¿El objetivo? Resucitar al legendario Elvis Presley mediante IA. Al introducir una amplia gama de material de archivo oficial de Elvis, incluyendo videoclips, fotografías y música, en un sofisticado modelo informático, esta IA de Elvis aprende a imitar su canto, habla, baile y andar con un parecido asombroso. ¿El resultado? Una actuación digital que captura la esencia del mismísimo Rey.

El proyecto Elvis Evolution es un ejemplo brillante del efecto transformador de la IA generativa en diversas industrias. En el capítulo anterior, exploraste el uso de MuseGAN para crear música que pudiera pasar por auténticas composiciones multipista. MuseGAN considera una pieza musical como un objeto multidimensional, similar a una imagen, y genera piezas musicales complejas que se asemejan a las del conjunto de datos de entrenamiento. Tanto la música real como la generada por IA son evaluadas por un crítico, quien ayuda a refinar la música generada por IA hasta que sea indistinguible de la original.

En este capítulo, adoptarás un enfoque diferente para la creación musical con IA, tratándola como una secuencia de eventos musicales. Aplicaremos técnicas de generación de texto, como se explicó en los capítulos 11 y 12, para predecir el siguiente elemento de una secuencia. En concreto, desarrollarás un modelo de estilo GPT para predecir el siguiente evento musical basándote

en todos los eventos anteriores de la secuencia. Los Transformers de estilo GPT son ideales para esta tarea gracias a su escalabilidad y al mecanismo de autoatención, que les ayuda a capturar dependencias de largo alcance y comprender el contexto. Esto los hace muy eficaces para la predicción y generación de secuencias en una amplia gama de contenido, incluida la música. El Transformer musical que crearás tiene 20,16 millones de parámetros, lo suficientemente grandes como para capturar las relaciones a largo plazo de las diferentes notas en las piezas musicales, pero lo suficientemente pequeños como para ser entrenados en un tiempo razonable.

Usaremos la música de piano de Maestro del grupo Magenta de Google como datos de entrenamiento. Aprenderás a convertir un archivo MIDI (interfaz digital de instrumento musical) en una secuencia de notas musicales, similar a los datos de texto sin procesar en el procesamiento del lenguaje natural (PLN). Luego, descompondrás las notas musicales en pequeños fragmentos llamados eventos musicales, similares a los tokens en el PLN. Dado que las redes neuronales solo aceptan entradas numéricas, asignarás cada token de evento único a un índice. De esta forma, las piezas musicales de los datos de entrenamiento se convierten en secuencias de índices, listas para ser introducidas en las redes neuronales.

Para entrenar al Transformador musical a predecir el siguiente token basándose en el token actual y todos los tokens anteriores de la secuencia, crearemos secuencias de 2048 índices como entradas (características x). Luego, desplazaremos las secuencias un índice a la derecha y las usaremos como salidas (objetivos y). Alimentaremos pares de (x, y) al Transformador musical para entrenar el modelo. Una vez entrenado, usaremos una secuencia corta de índices como indicador y la alimentaremos al Transformador musical para predecir el siguiente token, que luego se adjunta al indicador para formar una nueva secuencia. Esta nueva secuencia se devuelve al modelo para realizar más predicciones, y este proceso se repite hasta que la secuencia alcance la longitud deseada.

Verás que el Transformer musical entrenado puede generar música realista que imita el estilo del conjunto de datos de entrenamiento. Además, a diferencia de la música generada en el capítulo 13, aprenderás a controlar la creatividad de la pieza musical. Lo lograrás escalando los logits predichos con el parámetro de temperatura, tal como hiciste en capítulos anteriores al controlar la creatividad del texto generado.

14.1 Introducción al transformador musical

El concepto de Transformador musical se introdujo en 2018.² Este enfoque innovador extiende la arquitectura del Transformador, inicialmente diseñada para tareas de PLN, al campo de la generación musical. Como se explicó en capítulos anteriores, los Transformadores emplean mecanismos de autoatención para comprender eficazmente el contexto y capturar dependencias a largo plazo entre los elementos de una secuencia.

De forma similar, el Transformador musical está diseñado para generar una secuencia de notas musicales aprendiendo de un vasto conjunto de datos musicales existentes. El modelo se entrena para predecir el siguiente evento musical de una secuencia basándose en eventos anteriores, comprendiendo los patrones, las estructuras y las relaciones entre los diferentes elementos musicales de los datos de entrenamiento.

Un paso crucial en el entrenamiento de un Transformador musical consiste en comprender cómo representar la música como una secuencia de eventos musicales únicos, similar a los tokens en PLN. En el capítulo anterior, aprendiste a representar una pieza musical como un objeto 4D. En este capítulo, explorarás un enfoque alternativo para la representación musical, específicamente la representación musical basada en la interpretación mediante mensajes de control y valores de velocidad.³ Con base en esto, convertirás una pieza musical en cuatro tipos de eventos musicales: nota activada, nota desactivada, desplazamiento temporal y velocidad.

La nota activada indica el inicio de la ejecución de una nota musical, especificando su altura. La nota desactivada indica el final de una nota, indicando al instrumento que deje de tocarla. El desplazamiento temporal representa el tiempo transcurrido entre dos eventos musicales. La velocidad mide la fuerza o velocidad con la que se ejecuta una nota; los valores más altos corresponden a un sonido más fuerte y sonoro. Cada tipo de evento musical tiene muchos valores diferentes. Cada evento único se asignará a un índice diferente, transformando efectivamente una pieza musical en una secuencia de índices. A continuación, aplicará los modelos GPT, como se explica en los capítulos 11 y 12, para crear un transformador musical solo para decodificador que prediga el siguiente evento musical de la secuencia.

En esta sección, comenzarás aprendiendo sobre la representación musical basada en la interpretación mediante mensajes de control y valores de velo-

ciudad. Luego, explorarás cómo representar piezas musicales como secuencias de eventos musicales. Finalmente, aprenderás los pasos para construir y entrenar un Transformer para generar música.

14.1.1 Representación musical basada en la interpretación

La representación musical basada en la interpretación suele lograrse mediante el formato MIDI, que captura los matices de una interpretación musical mediante mensajes de control y valores de velocidad. En MIDI, las notas musicales se representan mediante mensajes de nota activada y nota desactivada, que incluyen información sobre la altura y la velocidad de cada nota.

Como explicamos en el capítulo 13, el valor de la altura va de 0 a 127, y cada valor corresponde a un semitono en una octava. Por ejemplo, el valor de 60 corresponde a una nota C4, mientras que el de 74 corresponde a una nota D5. El valor de velocidad, también de 0 a 127, representa la dinámica de la nota; los valores más altos indican una interpretación más fuerte o contundente. Al combinar estos mensajes de control y valores de velocidad, una secuencia MIDI puede capturar los detalles expresivos de una interpretación en vivo, lo que permite una reproducción expresiva mediante instrumentos y software compatibles con MIDI.

Para darle un ejemplo concreto de cómo una pieza musical puede representarse mediante mensajes de control y valores de velocidad, considere las cinco notas que se muestran en la siguiente lista.

Listado 14.1 Notas de ejemplo en una representación musical basada en la interpretación

```
<[SNote] time: 1.032552083333333 type: note_on,  
value: 74, velocity: 86>  
<[SNote] time: 1.044270833333333 type: note_on,  
value: 38, velocity: 77>  
<[SNote] time: 1.2265625 type: note_off,  
value: 74, velocity: None>  
<[SNote] time: 1.239583333333333 type: note_on,  
value: 73, velocity: 69>  
<[SNote] time: 1.2408854166666665 type: note_on,  
value: 37, velocity: 64>
```

Estas son las primeras cinco notas de una pieza musical en el conjunto de datos de entrenamiento que usarás en este capítulo. La primera nota tiene una marca de tiempo de aproximadamente 1,03 segundos, y una nota con un valor de altura 74 (D5) comienza a reproducirse a una velocidad de 86. Al observar la segunda nota, puedes inferir que, después de aproximadamente 0,01 segundos (ya que la marca de tiempo ahora es de 1,04 segundos), una nota con un valor de altura 38 comienza a reproducirse a una velocidad de 77, y así sucesivamente.

Estas notas musicales son similares al texto sin procesar en PLN; no podemos introducirlas directamente en un transformador musical para entrenar el modelo. Primero debemos tokenizar las notas y luego convertir los tokens en índices antes de introducirlos en el modelo.

Para tokenizar las notas musicales, representaremos la música con incrementos de 0,01 segundos para reducir el número de pasos de tiempo en la pieza musical. Además, separaremos los mensajes de control de los valores de velocidad y los trataremos como elementos diferentes de la pieza musical. Específicamente, representaremos la música mediante una combinación de eventos de nota activada, nota desactivada, desplazamiento temporal y velocidad. Una vez hecho esto, las cinco notas musicales anteriores se pueden representar mediante los siguientes eventos (algunos se omiten para mayor brevedad).

Listado 14.2 Representación tokenizada de una pieza musical

```
<Event type: time_shift, value: 99>,
<Event type: time_shift, value: 2>,
<Event type: velocity, value: 21>,
<Event type: note_on, value: 74>,
<Event type: time_shift, value: 0>,
<Event type: velocity, value: 19>,
<Event type: note_on, value: 38>,
<Event type: time_shift, value: 17>,
<Event type: note_off, value: 74>,
<Event type: time_shift, value: 0>,
<Event type: velocity, value: 17>,
<Event type: note_on, value: 73>,
<Event type: velocity, value: 16>,
<Event type: note_on, value: 37>,
```

```
<Event type: time_shift, value: 0>
```

```
...
```

Contaremos los cambios de tiempo en incrementos de 0,01 segundos y tokenizaremos los cambios de tiempo de 0,01 segundos a 1 segundo con 100 valores diferentes. Por lo tanto, los eventos de cambio de tiempo se tokenizan en 1 de 100 tokens de evento únicos: un valor de 0 indica un lapso de tiempo de 0,01 segundos, 1 indica un lapso de tiempo de 0,02 segundos, y así sucesivamente, hasta 99, que indica un lapso de tiempo de 1 segundo. Si un cambio de tiempo dura más de 1 segundo, puede usar varios tokens de cambio de tiempo para indicarlo. Por ejemplo, los dos primeros tokens en el listado 14.2 son tokens de cambio de tiempo, con valores 99 y 2, respectivamente, que indican lapsos de tiempo de 1 segundo y 0,03 segundos. Esto coincide con la marca de tiempo de la primera nota musical en el listado 14.1: 1,0326 segundos.

El listado 14.2 también muestra que la velocidad es un tipo de evento musical independiente. Colocamos el valor de velocidad en 32 intervalos equidistantes, convirtiendo los valores originales de velocidad, que van de 0 a 127, en 1 de 32 valores, que van de 0 a 31. Por esta razón, el valor original de velocidad de 86 en la primera nota del listado 14.1 ahora se representa como un evento de velocidad con un valor de 21 en el listado 14.2 (el número 86 se encuentra en el intervalo 22 · y Python utiliza indexación basada en cero).

La Tabla 14.1 muestra el significado de cuatro tipos de diferentes eventos tokenizados, sus rangos de valores y el significado de cada evento simbólico.

Tabla 14.1 Significados de los diferentes tokens de eventos

Tipo de token de evento	Rango de valores del token de evento	Significado de los tokens de evento
<code>note_on</code>	0–127	Comenzar a tocar con un valor de tono determinado. Por ejemplo, <code>note_on</code> con un valor de 74, significa comenzar a tocar la nota D5.
<code>note_off</code>	0–127	Soltar una nota. Por ejemplo, <code>note_off</code> con el valor 60, significa dejar de tocar la nota C4.
<code>time_shift</code>	0–99	Los <code>time_shift</code> valores se expresan en incrementos de 0,01 segundos. Por ejemplo, 0 indica 0,01 segundos, 2 indica 0,03 segundos y 99 indica 1 segundo.
<code>velocity</code>	0–31	Los valores de velocidad originales se colocan en 32 contenedores. Se utiliza el valor del contenedor. Por ejemplo, un valor de velocidad original de 86 ahora tiene un valor tokenizado de 21.

Similar al enfoque del PLN, convertiremos cada token único en un índice para poder introducir los datos en las redes neuronales. Según la tabla 14.1, hay 128 tokens únicos de evento de nota activada, 128 tokens de evento de nota desactivada, 32 tokens de evento de velocidad y 100 tokens de evento de desplazamiento temporal. Esto da un total de $128 + 128 + 32 + 100 = 388$ tokens únicos. Por consiguiente, convertimos estos 388 tokens únicos en índices que van del 0 al 387, según las asignaciones proporcionadas en la tabla 14.2.

Tabla 14.2 Asignación de tokens de eventos a índices y de índices a tokens

Tipo de token	Rango de índice	Token de evento para indexar	Índice del token de evento
<code>note_on</code>	0–127	El valor del token <code>note_on</code> . Por ejemplo, al token <code>note_on</code> con un valor de 74 se le asigna un valor de índice de 74.	Si el rango del índice es de 0 a 127, establezca el tipo de token <code>note_on</code> y el valor en el valor del índice. Por ejemplo, el valor de índice 63 se asigna a un token <code>note_on</code> con el valor 63.
<code>note_off</code>	128–255	128 más el valor del token <code>note_off</code> . Por ejemplo, al token <code>note_off</code> con valor 60 se le asigna un valor de índice de 188 (ya que $128 + 60 = 188$).	Si el rango del índice es de 128 a 255, establezca el tipo de token <code>note_off</code> y el valor en el índice menos 128. Por ejemplo, el índice 180 se asigna al token <code>note_off</code> con valor 52.
<code>time_shift</code>	256–355	256 más el valor del token <code>time_shift</code> . Por ejemplo, al token <code>time_shift</code> con valor 16 se le asigna un valor de índice de 272 (ya que $256 + 16 = 272$).	Si el rango del índice es de 256 a 355, establezca el tipo de token <code>time_shift</code> y el valor en el índice menos 256. Por ejemplo, el índice 288 se asigna al token <code>time_shift</code> con valor 32.

<code>velocity</code>	356– 387	356 más el valor del token de velocidad. Por ejemplo, al token de velocidad con un valor de 21 se le asigna un valor de índice de 377 (ya que $356 + 21 = 377$).	Si el rango de índice es de 356 a 387, establezca el tipo de token <code>velocity</code> y el valor en el índice menos 356. Por ejemplo, el índice 380 se asigna al token <code>velocity</code> con valor 24.
-----------------------	-------------	---	---

La tercera columna de la tabla 14.2 describe la conversión de tokens de evento a índices. A los tokens de nota activada se les asignan valores de índice de 0 a 127, donde el valor del índice corresponde al número de altura en el token. A los tokens de nota desactivada se les asignan valores de índice de 128 a 255, siendo el valor del índice 128 más el número de altura. A los tokens de desplazamiento temporal se les asignan valores de índice de 256 a 355, siendo el valor del índice 256 más el valor del desplazamiento temporal. Por último, a los tokens de velocidad se les asignan valores de índice de 356 a 387, siendo el valor del índice 356 más el número de bin de velocidad.

Usando esta asignación de token a índice, convertiremos cada pieza musical en una secuencia de índices. Aplicaremos esta conversión a todas las piezas musicales del conjunto de datos de entrenamiento y usaremos las secuencias resultantes para entrenar nuestro Transformer musical (cuyos detalles se explicarán más adelante). Una vez entrenado, usaremos el Transformer para generar música en forma de secuencia de índices. El paso final es convertir esta secuencia de nuevo a formato MIDI para poder reproducir y disfrutar de la música en un ordenador.

La última columna de la tabla 14.2 proporciona orientación sobre cómo convertir índices de nuevo en tokens de evento. Primero, determinamos el tipo de token según el rango en el que se encuentra el índice. Los cuatro rangos de la segunda columna de la tabla 14.2 corresponden a los cuatro tipos de token de la primera columna. Para obtener el valor de cada tipo de token, restamos el valor del índice por 0, 128, 256 y 356 para los cuatro tipos de token, respectivamente. Estos eventos tokenizados se convierten en notas musicales en formato MIDI, listas para su reproducción en un ordenador.

14.1.2 La arquitectura del transformador musical

En el capítulo 9, construimos un Transformer codificador-decodificador, y en los capítulos 11 y 12, nos centramos en Transformers que solo funcionan con decodificadores. A diferencia de las tareas de traducción de idiomas, donde el codificador captura el significado del idioma fuente y lo pasa al decodificador para generar la traducción, la generación musical no requiere que un codificador comprenda un idioma diferente. En su lugar, el modelo genera tokens de eventos posteriores basados en tokens de eventos previos en la secuencia musical. Por lo tanto, construiremos un Transformer que solo funciona con decodificadores para nuestra tarea de generación musical.

Nuestro Transformador musical, al igual que otros modelos de Transformadores, utiliza mecanismos de autoatención para capturar las dependencias de largo alcance entre los diferentes eventos musicales de una pieza musical, generando así música coherente y realista. Si bien nuestro Transformador musical difiere en tamaño de los modelos GPT que construimos en los capítulos 11 y 12, comparte la misma arquitectura central. Sigue el mismo diseño estructural que los modelos GPT-2, pero es significativamente más pequeño, lo que facilita su entrenamiento sin necesidad de supercomputadoras.

Específicamente, nuestro Transformador musical consta de seis capas de decodificación con una dimensión de incrustación de 512, lo que significa que cada token se representa mediante un vector de 512 valores tras la incrustación de palabras. En lugar de utilizar las funciones seno y coseno para la codificación posicional, como en el artículo original de 2017 «Attention Is All You Need», utilizamos capas de incrustación para aprender las codificaciones posicionales de las diferentes posiciones en una secuencia. Como resultado, cada posición en una secuencia también se representa mediante un vector de 512 valores. Para calcular la autoatención causal, utilizamos ocho cabezas de atención paralelas para capturar diferentes aspectos del significado de una ficha en la secuencia, asignando a cada cabeza de atención una dimensión de 64 (512/8).

En comparación con el tamaño de vocabulario de 50,257 en los modelos GPT-2, nuestro modelo tiene un tamaño de vocabulario mucho menor, de 390 (388 tokens de evento diferentes, además de un token para indicar el final de una secuencia y un token para llenar secuencias más cortas; más adelante explicaré por qué se necesita el relleno). Esto nos permite establecer la longitud máxima de secuencia en nuestro Transformador musical en 2,048, que es

mucho mayor que la longitud máxima de secuencia de 1,024 en los modelos GPT-2. Esta opción es necesaria para capturar las relaciones a largo plazo de las notas musicales en una secuencia. Con estos valores de hiperparámetros, nuestro Transformador musical tiene un tamaño de 20.16 millones de parámetros.

La Figura 14.1 ilustra la arquitectura del Transformador musical que crearemos en este capítulo. Es similar a la arquitectura de los modelos GPT que creó en los capítulos 11 y 12. La Figura 14.1 también muestra el tamaño de los datos de entrenamiento a medida que pasan por el modelo durante el entrenamiento.

La entrada del Transformador musical que construimos consta de incrustaciones de entrada, como se muestra en la parte inferior de la figura 14.1. La incrustación de entrada es la suma de la incrustación de palabras y la codificación posicional de la secuencia de entrada. Esta incrustación de entrada se pasa secuencialmente a través de seis bloques decodificadores.

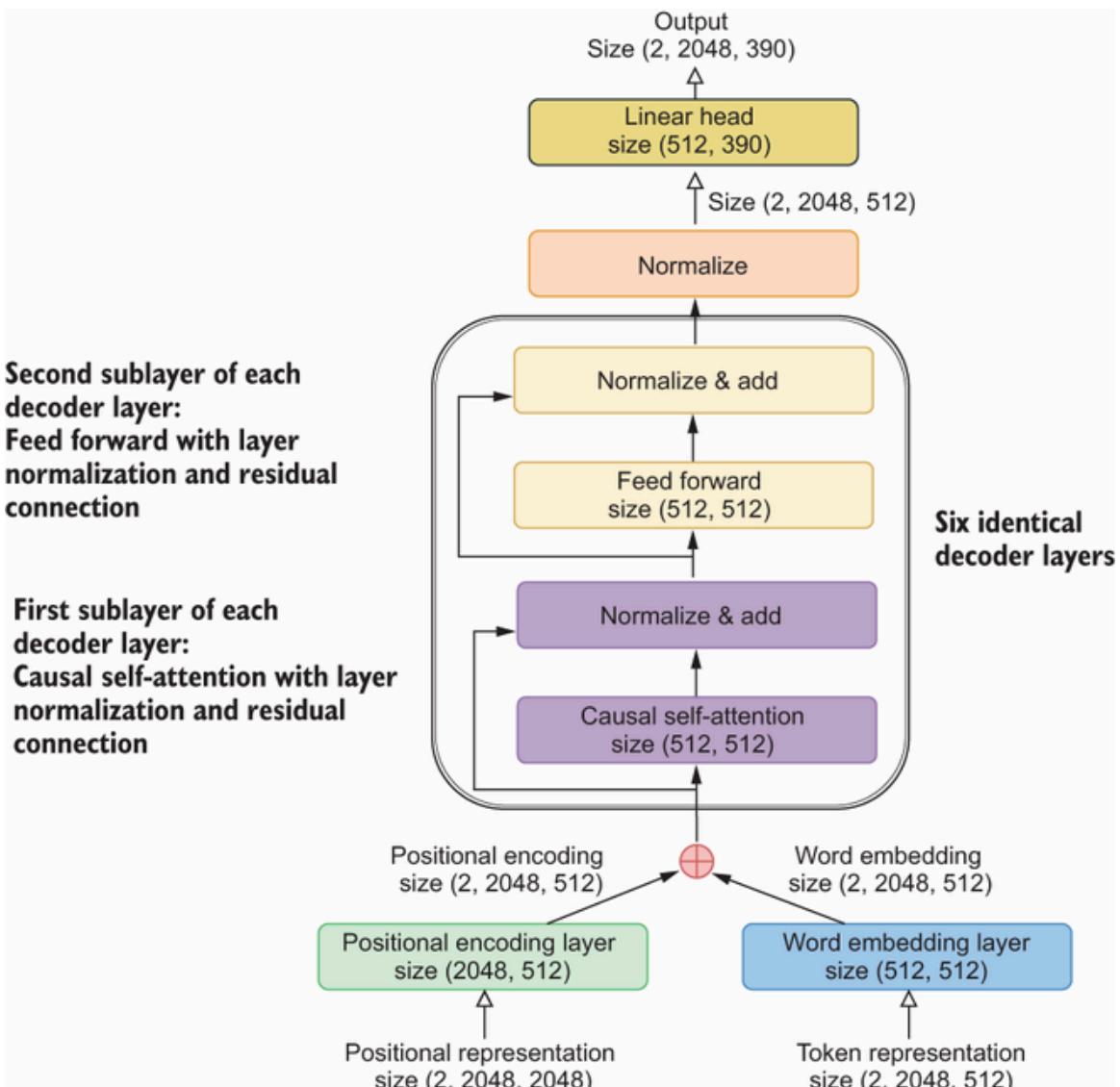


Figura 14.1 Arquitectura de un transformador musical. Los archivos de música en formato MIDI se convierten primero en secuencias de eventos musicales. Estos eventos se tokenizan y se convierten en índices. Organizamos estos índices en secuencias de 2048 elementos, y cada lote contiene dos de estas secuencias. La secuencia de entrada se somete primero a la incrustación de palabras y la codificación posicional; la incrustación de entrada es la suma de estos dos componentes. Esta incrustación de entrada se procesa a través de seis capas de decodificación, cada una de las cuales utiliza mecanismos de autoatención para capturar las relaciones entre los diferentes eventos musicales de la secuencia. Tras pasar por las capas de decodificación, la salida se normaliza para garantizar la estabilidad durante el proceso de entrenamiento. A continuación, pasa por una capa lineal, lo que resulta en un tamaño de salida de 390, que corresponde al número de tokens únicos en el vocabulario. Esta salida final representa los logits predichos para el siguiente evento musical de la secuencia.

Como se explicó en los capítulos 11 y 12, cada capa del decodificador consta de dos subcapas: una capa de autoatención causal y una red de retroalimentación. Además, aplicamos normalización de capas y conexiones residuales a cada subcapa para mejorar la estabilidad y la capacidad de aprendizaje del modelo.

Tras pasar por las capas del decodificador, la salida se normaliza y se introduce en una capa lineal. El número de salidas de nuestro modelo corresponde al número de tokens de eventos musicales únicos en el vocabulario, que es 390. La salida del modelo son los logits del siguiente token de evento musical.

Posteriormente, aplicaremos la función softmax a estos logits para obtener la distribución de probabilidad de todos los posibles tokens de evento. El modelo está diseñado para predecir el siguiente token de evento basándose en el token actual y todos los tokens anteriores en la secuencia musical, lo que permite generar secuencias coherentes y musicalmente sensatas.

14.1.3 Entrenamiento del transformador musical

Ahora que entendemos cómo construir un transformador de música para la generación de música, describamos el proceso de entrenamiento para el transformador de música.

El estilo musical generado por el modelo se ve influenciado por las piezas musicales utilizadas para el entrenamiento. Utilizaremos interpretaciones de piano del grupo Magenta de Google para entrenar nuestro modelo. La Figura 14.2 ilustra los pasos del entrenamiento del Transformer para la generación de música.

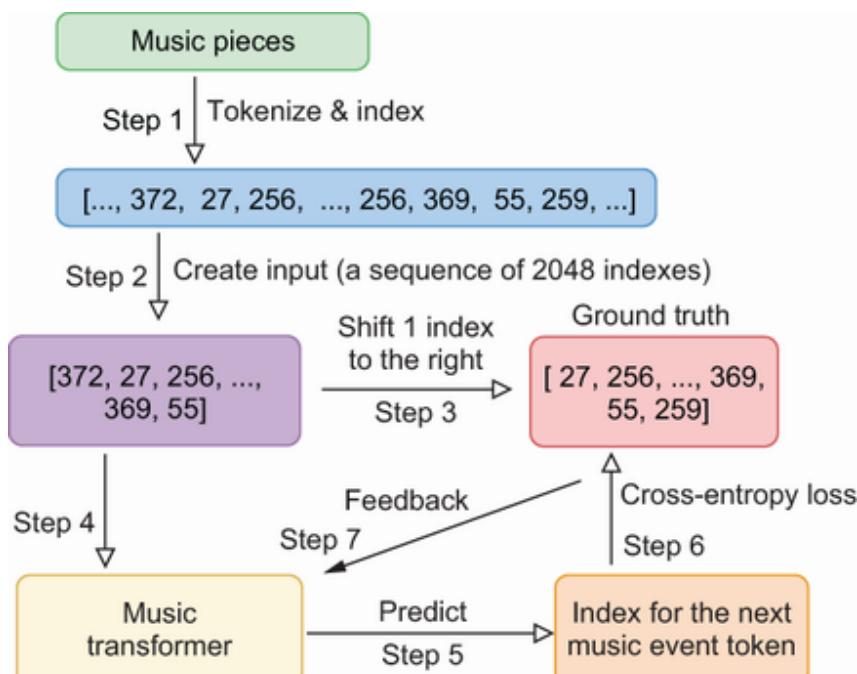


Figura 14.2 El proceso de entrenamiento de un transformador de música para generar música

Similar al enfoque que hemos adoptado en las tareas de PLN, el primer paso en el proceso de entrenamiento de nuestro Transformador musical es convertir los datos de entrenamiento sin procesar a formato numérico para que puedan incorporarse al modelo. En concreto, comenzamos convirtiendo los archivos MIDI del conjunto de entrenamiento en secuencias de notas musicales. A continuación, tokenizamos estas notas convirtiéndolas en uno de 388

eventos/tokens únicos. Tras la tokenización, asignamos un índice único (es decir, un entero) a cada token, convirtiendo las piezas musicales del conjunto de entrenamiento en secuencias de enteros (véase el paso 1 en la figura 14.2).

A continuación, transformamos la secuencia de enteros en datos de entrenamiento dividiendo esta secuencia en secuencias de igual longitud (paso 2 en la figura 14.2). Permitimos una longitud máxima de 2048 índices en cada secuencia. La elección de 2048 nos permite capturar dependencias de largo alcance entre eventos musicales en una secuencia musical para crear música realista. Estas secuencias forman las características (la variable x) de nuestro modelo. Como hicimos en capítulos anteriores al entrenar modelos GPT para generar texto, deslizamos la ventana de secuencia de entrada un índice a la derecha y la usamos como salida en los datos de entrenamiento (la variable y ; paso 3 en la figura 14.2). Al hacerlo, obligamos a nuestro modelo a predecir el siguiente token musical en una secuencia basándose en el token actual y todos los tokens anteriores en la secuencia musical.

Los pares de entrada y salida sirven como datos de entrenamiento (x, y) para el Transformador musical. Durante el entrenamiento, iterará a través de los datos de entrenamiento. En los pasos hacia adelante, alimenta la secuencia de entrada x a través del Transformador musical (paso 4). El Transformador musical luego hace una predicción basada en los parámetros actuales en el modelo (paso 5). Calcula la pérdida de entropía cruzada comparando los siguientes tokens predichos con la salida obtenida del paso 3. En otras palabras, compara la predicción del modelo con la verdad fundamental (paso 6). Finalmente, ajustará los parámetros en el Transformador musical para que en la siguiente iteración, las predicciones del modelo se acerquen más a la salida real, minimizando la pérdida de entropía cruzada (paso 7). El modelo está esencialmente realizando un problema de clasificación multicategoría: está prediciendo el siguiente token de todos los tokens musicales únicos en el vocabulario.

Repetirá los pasos 3 a 7 mediante varias iteraciones. Tras cada iteración, se ajustarán los parámetros del modelo para mejorar la predicción del siguiente token. Este proceso se repetirá durante 50 épocas.

Para generar una nueva pieza musical con el modelo entrenado, obtenemos una pieza musical del conjunto de prueba, la tokenizamos y la convertimos en una secuencia larga de índices. Usaremos los primeros 250 índices, por ejemplo, como indicador (200 o 300 darán resultados similares). Luego, le pe-

dimos al Transformer musical entrenado que genere nuevos índices hasta que la secuencia alcance una longitud determinada (por ejemplo, 1000 índices). Finalmente, convertimos la secuencia de índices de nuevo en un archivo MIDI para su reproducción en el ordenador.

14.2 Tokenización de piezas musicales

Una vez comprendido la estructura del Transformer musical y su metodología de entrenamiento, comenzaremos con el primer paso: la tokenización e indexación de las composiciones musicales en nuestro conjunto de datos de entrenamiento.

Comenzaremos empleando una representación basada en la interpretación (como se explicó en la primera sección) para representar piezas musicales como notas musicales, similares al texto sin formato en PLN. Después, dividiremos estas notas musicales en una serie de eventos, similares a los tokens en PLN. A cada evento único se le asignará un índice diferente. Utilizando esta asignación, transformaremos todas las piezas musicales del conjunto de datos de entrenamiento en secuencias de índices.

A continuación, estandarizaremos estas secuencias de índices a una longitud fija, concretamente secuencias con 2048 índices, y las usaremos como entradas de características (x) . Al desplazar la ventana un índice a la derecha, generaremos las secuencias de salida correspondientes (y) . A continuación, agruparemos los pares de entrada y salida (x, y) en lotes, preparándolos para el entrenamiento del transformador musical más adelante en este capítulo.

Como necesitaremos las bibliotecas `pretty_midi` y `music21` para procesar archivos MIDI, ejecute la siguiente línea de código en una nueva celda en la aplicación Jupyter Notebook:

```
!pip install pretty_midi music21
```

14.2.1 Descarga de datos de entrenamiento

Obtendremos las interpretaciones de piano del conjunto de datos MAESTRO, disponible a través del grupo Magenta de Google

(<https://storage.googleapis.com/magentadata/datasets/maestro/v2.0.0/maestro-v2.0.0-midi.zip>), y descargaremos el archivo ZIP. Tras la descarga, descom-

prímalo y mueva la carpeta resultante, /maestro-v2.0.0/, al directorio /files/ de su ordenador.

Asegúrese de que la carpeta /maestro-v2.0.0/ contenga 4 archivos (uno de ellos llamado "maestro-v2.0.0.json") y 10 subcarpetas. Cada subcarpeta debe contener más de 100 archivos MIDI. Para familiarizarse con el sonido de las piezas musicales de los datos de entrenamiento, intente abrir algunos archivos MIDI con su reproductor de música preferido.

A continuación, dividiremos los archivos MIDI en subconjuntos de entrenamiento, validación y prueba. Para empezar, cree tres subcarpetas dentro de /files/maestro-v2.0.0/:

```
import os

os.makedirs("files/maestro-v2.0.0/train", exist_ok=True)
os.makedirs("files/maestro-v2.0.0/val", exist_ok=True)
os.makedirs("files/maestro-v2.0.0/test", exist_ok=True)
```

Para facilitar el procesamiento de archivos MIDI, visite el repositorio de GitHub de Kevin Yang en <https://github.com/jason9693/midi-neural-processor>, descargue el archivo processor.py y colóquelo en la carpeta /utils/ de su computadora. También puede obtener el archivo del repositorio de GitHub del libro: <https://github.com/markhliu/DGAI>. Usaremos este archivo como módulo local para transformar un archivo MIDI en una secuencia de índices y viceversa. Este enfoque nos permite concentrarnos en el desarrollo, entrenamiento y uso de un transformador musical sin complicarnos con los detalles de la conversión de formatos musicales. Al mismo tiempo, proporcionaré un ejemplo sencillo de cómo funciona este proceso para que usted mismo pueda convertir entre un archivo MIDI y una secuencia de índices usando el módulo.

Además, necesitas descargar el archivo ch14util.py del repositorio de GitHub del libro y colocarlo en el directorio /utils/ de tu computadora. Usaremos el archivo ch14util.py como otro módulo local para definir el modelo de Transformer musical.

El archivo maestro-v2.0.0.json, dentro de la carpeta /maestro-v2.0.0/, contiene los nombres de todos los archivos MIDI y sus subconjuntos designados (en-

trenamiento, validación o prueba). Con base en esta información, clasificaremos los archivos MIDI en tres subcarpetas correspondientes.

Listado 14.3 División de datos de entrenamiento en subconjuntos de entrenamiento, validación y prueba

```
import json
import pickle
from utils.processor import encode_midi

file="files/maestro-v2.0.0/maestro-v2.0.0.json"

with open(file,"r") as fb:
    maestro_json=json.load(fb) ①

for x in maestro_json:
    mid=rf'files/maestro-v2.0.0/{x["midi_filename"]}'
    split_type = x["split"] ②
    f_name = mid.split("/)[-1] + ".pickle"
    if(split_type == "train"):
        o_file = rf'files/maestro-v2.0.0/train/{f_name}'
    elif(split_type == "validation"):
        o_file = rf'files/maestro-v2.0.0/val/{f_name}'
    elif(split_type == "test"):
        o_file = rf'files/maestro-v2.0.0/test/{f_name}'
    prepped = encode_midi(mid)
    with open(o_file,"wb") as f:
        pickle.dump(prepped, f)
```

① Carga el archivo JSON

② Itera a través de todos los archivos en los datos de entrenamiento

③ Coloca un archivo en una subcarpeta de entrenamiento, validación o prueba según las instrucciones del archivo JSON

El archivo de notación de objetos JavaScript (JSON) que descargaste clasifica cada archivo del conjunto de datos de entrenamiento en uno de tres subconjuntos: entrenamiento, validación y prueba. Tras ejecutar el código anterior, si exploras las carpetas /train/, /val/ y /test/ en tu ordenador, deberías encontrar varios archivos en cada una. Para verificar la cantidad de archivos en

cada una de estas tres carpetas, puedes realizar las siguientes comprobaciones:

```
train_size=len(os.listdir('files/maestro-v2.0.0/train'))
print(f"there are {train_size} files in the train set")
val_size=len(os.listdir('files/maestro-v2.0.0/val'))
print(f"there are {val_size} files in the validation set")
test_size=len(os.listdir('files/maestro-v2.0.0/test'))
print(f"there are {test_size} files in the test set")
```

La salida del bloque de código anterior es

```
there are 967 files in the train set
there are 137 files in the validation set
there are 178 files in the test set
```

Los resultados muestran que hay 967, 137 y 178 piezas musicales en los subconjuntos de entrenamiento, validación y prueba, respectivamente.

14.2.2 Tokenización de archivos MIDI

A continuación, representaremos cada archivo MIDI como una secuencia de notas musicales.

Listado 14.4 Conversión de un archivo MIDI a una secuencia de notas musicales

```
import pickle
from utils.processor import encode_midi
import pretty_midi
from utils.processor import (_control_preprocess,
                            _note preprocess,_divide_note,
                            _make_time_sift_events,_snote2events)

file='MIDI-Unprocessed_Chamber1_MID--AUDIO_07_R3_2018_wav--2'
name=rf'files/maestro-v2.0.0/2018/{file}.midi'          (1)

events=[]
notes=[]
song=pretty_midi.PrettyMIDI(name)
for inst in song.instruments:
    inst_notes=inst.notes
```

```

ctrls=_control_preprocess([ctrl for ctrl in
    inst.control_changes if ctrl.number == 64])
notes += _note_preprocess(ctrls, inst_notes)                                ②
dnotes = _divide_note(notes)                                                 ③
dnotes.sort(key=lambda x: x.time)
for i in range(5):
    print(dnotes[i])

```

① Selecciona un archivo MIDI del conjunto de datos de entrenamiento

② Extrae eventos musicales de la música.

③ Coloca todos los eventos musicales en la lista dnotes

Seleccionamos un archivo MIDI del conjunto de datos de entrenamiento y usamos el módulo local processor.py para convertirlo en una secuencia de notas musicales. El resultado del código anterior es

```

<[SNote] time: 1.032552083333333 type: note_on,
value: 74, velocity: 86>
<[SNote] time: 1.044270833333333 type: note_on,
value: 38, velocity: 77>
<[SNote] time: 1.2265625 type: note_off,
value: 74, velocity: None>
<[SNote] time: 1.239583333333333 type: note_on,
value: 73, velocity: 69>
<[SNote] time: 1.2408854166666665 type: note_on,
value: 37, velocity: 64>

```

La salida que se muestra aquí muestra las primeras cinco notas musicales del archivo MIDI. Es posible que haya observado que la representación temporal en la salida es continua. Algunas notas musicales contienen los atributos "a" `note_on` y "a" `velocity`, lo que complica el proceso de tokenización debido a la gran cantidad de eventos musicales únicos resultantes de la naturaleza continua de la representación temporal. Además, la combinación de diferentes valores "`note_on` y `velocity`" es grande (cada uno puede asumir 128 valores distintos, de 0 a 127), lo que resulta en un vocabulario excesivamente extenso. Esto, a su vez, haría que el entrenamiento fuera poco práctico.

Para mitigar este problema y disminuir el tamaño del vocabulario, convertimos estas notas musicales en eventos tokenizados:

```

cur_time = 0
cur_vel = 0
for snote in dnotes:
    events += _make_time_sift_events(prev_time=cur_time,      ①
                                      post_time=snote.time)
    events += _snote2events(snote=snote, prev_vel=cur_vel)   ②
    cur_time = snote.time
    cur_vel = snote.velocity
indexes=[e.to_int() for e in events]
for i in range(15):                                         ③
    print(events[i])

```

① Discretiza el tiempo para reducir el número de eventos únicos

② Convierte notas musicales en eventos

③ Imprime los primeros 15 eventos

El resultado es el siguiente:

```

<Event type: time_shift, value: 99>
<Event type: time_shift, value: 2>
<Event type: velocity, value: 21>
<Event type: note_on, value: 74>
<Event type: time_shift, value: 0>
<Event type: velocity, value: 19>
<Event type: note_on, value: 38>
<Event type: time_shift, value: 17>
<Event type: note_off, value: 74>
<Event type: time_shift, value: 0>
<Event type: velocity, value: 17>
<Event type: note_on, value: 73>
<Event type: velocity, value: 16>
<Event type: note_on, value: 37>
<Event type: time_shift, value: 0>

```

La pieza musical ahora está representada por cuatro tipos de eventos: nota activada, nota desactivada, desplazamiento temporal y velocidad. Cada tipo de evento incluye valores diferentes, lo que resulta en un total de 388 eventos únicos, como se detalla en la tabla 14.2. Los detalles de la conversión de un archivo MIDI en una secuencia de estos eventos únicos no son esenciales para construir y entrenar un transformador musical. Por lo tanto, no profun-

dizaremos en este tema; los lectores interesados pueden consultar Huang et al. (2018), citado anteriormente. Solo necesita saber cómo usar el módulo processor.py para transformar un archivo MIDI en una secuencia de índices y viceversa. En la siguiente subsección, aprenderá cómo lograrlo.

14.2.3 Preparación de los datos de entrenamiento

Hemos aprendido a convertir piezas musicales en tokens y luego en índices. El siguiente paso consiste en preparar los datos de entrenamiento para usarlos en el entrenamiento del Transformador musical más adelante en este capítulo. Para ello, definimos la función `create_xys()` que se muestra en la siguiente lista.

Listado 14.5 Creación de datos de entrenamiento

```
import torch,os,pickle

max_seq=2048
def create_xys(folder):
    files=[os.path.join(folder,f) for f in os.listdir(folder)]
    xys=[]
    for f in files:
        with open(f,"rb") as fb:
            music=pickle.load(fb)
            music=torch.LongTensor(music)
            x=torch.full((max_seq,),389, dtype=torch.long)
            y=torch.full((max_seq,),389, dtype=torch.long)      ①
            length=len(music)
            if length<=max_seq:
                print(length)
                x[:length]=music                         ②
                y[:length-1]=music[1:]                   ③
                y[length-1]=388                         ④
            else:
                x=music[:max_seq]
                y=music[1:max_seq+1]
            xys.append((x,y))
    return xys
```

① Crea secuencias (x, y), con longitudes iguales de 2048 índices y establece el índice 399 como índice de relleno

② Utiliza una secuencia de hasta 2048 índices como entrada

③ Desliza la ventana un índice hacia la derecha y la usa como salida

④ Establece el índice final como 388

Como hemos visto repetidamente a lo largo de este libro, en las tareas de predicción de secuencias, utilizamos una secuencia x como entrada. Luego, desplazamos la secuencia una posición a la derecha para crear la secuencia de salida. Este enfoque obliga al modelo a predecir el siguiente elemento basándose en el elemento actual y todos los elementos anteriores de la secuencia.

Para preparar los datos de entrenamiento para nuestro Transformer musical, construiremos pares (x, y) , donde x es la entrada e y es la salida. Tanto x e y , contienen 2.048 índices, lo suficientemente largos para capturar las relaciones a largo plazo de las notas musicales en una secuencia, pero no demasiado largos para obstaculizar el proceso de entrenamiento.

Iteraremos todas las piezas musicales del conjunto de datos de entrenamiento que descargamos. Si una pieza musical supera los 2048 índices de longitud, usaremos los primeros 2048 índices como entrada x . Para la salida y , usaremos los índices desde la segunda posición hasta la 2049. En el caso poco frecuente de que la pieza musical tenga una longitud menor o igual a 2048 índices, rellenaremos la secuencia con el índice 389 para asegurar que tanto x como y tengan una longitud de 2048 índices. Además, usaremos el índice 388 para indicar el final de la secuencia y .

Como se mencionó en la primera sección, hay un total de 388 tokens de eventos únicos, indexados de 0 a 387. Dado que usamos 388 para señalar el final de la secuencia y y 389 para llenar secuencias, tenemos un total de 390 índices únicos, que van de 0 a 389.

Ahora podemos aplicar la función `create_xys()` al subconjunto del tren:

```
trainfolder='files/maestro-v2.0.0/train'  
train=create_xys(trainfolder)
```

La salida es

```
15  
5  
1643
```

1771

586

Esto muestra que, de las 967 piezas musicales del subconjunto de trenes, solo 5 son más cortas que 2048 índices. Sus longitudes se muestran en la salida anterior.

También aplicamos la función `create_xys()` a los subconjuntos de validación y prueba:

```
valfolder='files/maestro-v2.0.0/val'  
testfolder='files/maestro-v2.0.0/test'  
print("processing the validation set")  
val=create_xys(valfolder)  
print("processing the test set")  
test=create_xys(testfolder)
```

La salida es

```
processing the validation set  
processing the test set  
1837
```

Esto demuestra que todas las piezas musicales del subconjunto de validación tienen más de 2048 índices. Solo una pieza musical del subconjunto de prueba tiene menos de 2048 índices.

Imprimamos un archivo del subconjunto de validación y veamos cómo se ve:

```
val1, _ = val[0]  
print(val1.shape)  
print(val1)
```

El resultado es el siguiente:

```
torch.Size([2048])  
tensor([324, 366, 67, ..., 60, 264, 369])
```

La secuencia x del primer par del conjunto de validación tiene una longitud de 2048 índices, con valores como 324, 367, etc. Usemos el módulo `processor.py` para decodificar la secuencia a un archivo MIDI para que puedas escuchar su sonido:

```
from utils.processor import decode_midi  
  
file_path="files/val1.midi"  
decode_midi(val1.cpu().numpy(), file_path)
```

La función `decode_midi()` convierte una secuencia de índices en un archivo MIDI reproducible en su computadora. Después de ejecutar el bloque de código anterior, abra el archivo val1.midi con un reproductor de música en su computadora para escuchar cómo suena.

Ejercicio 14.1

Usa la función `decode_midi()` del módulo local `processor.py` para convertir la primera pieza musical del subconjunto de entrenamiento en un archivo MIDI. Guárdalo como train1.midi en tu ordenador. Ábrelo con un reproductor de música y obtén una idea del tipo de música que usamos como datos de entrenamiento.

Por último, creamos un cargador de datos para que los datos estén en lotes para el entrenamiento:

```
from torch.utils.data import DataLoader  
  
batch_size=2  
trainloader=DataLoader(train,batch_size=batch_size,  
shuffle=True)
```

Para evitar que la GPU se quede sin memoria, usaremos un tamaño de lote de 2, ya que hemos creado secuencias muy largas, cada una con 2048 índices. Si es necesario, reduzca el tamaño del lote a uno o cambie al entrenamiento de CPU.

Con esto, nuestros datos de entrenamiento están preparados. En las siguientes dos secciones, construiremos un transformador musical desde cero y lo entrenaremos con los datos de entrenamiento que acabamos de preparar.

14.3 Construyendo un GPT para generar música

Ahora que nuestros datos de entrenamiento están preparados, construiremos un modelo GPT desde cero para la generación de música. La arquitectura de este modelo será similar a la del modelo GPT-2XL que desarrollamos en el capítulo 11 y al generador de texto del capítulo 12. Sin embargo, el tamaño de nuestro Transformador de música variará debido a los hiperparámetros específicos que seleccionemos.

Para ahorrar espacio, colocaremos la construcción del modelo en el módulo local ch14util.py. Nos centraremos en los hiperparámetros elegidos para el transformador musical. En concreto, determinaremos los valores del `n_layer`: el número de capas del decodificador en el modelo; `n_head`: el número de cabezas paralelas que se usarán para calcular la autoatención causal; `n_embd`: la dimensión de incrustación; y `block_size`: el número de tokens en la secuencia de entrada.

14.3.1 Hiperparámetros en el transformador de música

Abre el archivo ch14util.py que descargaste anteriormente del repositorio de GitHub del libro. Dentro encontrarás varias funciones y clases idénticas a las definidas en el capítulo 12.

Como en todos los modelos GPT que hemos visto en este libro, la red de propagación hacia adelante del bloque decodificador utiliza la función de activación de la unidad lineal de error gaussiano (GELU). Por consiguiente, definimos una clase GELU en ch14util.py, tal como hicimos en el capítulo 12.

Empleamos una clase `Config()` para almacenar todos los hiperparámetros utilizados en el Transformer musical:

```
from torch import nn
class Config():
    def __init__(self):
        self.n_layer = 6
        self.n_head = 8
        self.n_embd = 512
        self.vocab_size = 390
        self.block_size = 2048
        self.embd_pdrop = 0.1
        self.resid_pdrop = 0.1
```

```
        self.attn_pdrop = 0.1
config=Config()
device="cuda" if torch.cuda.is_available() else "cpu"
```

Los atributos de la clase `Config()` sirven como hiperparámetros para nuestro Transformador musical. Asignamos un valor de 6 al atributo `n_layer`, lo que indica que nuestro Transformador musical consta de 6 capas de decodificación. Esto supera el número de capas de decodificación del modelo GPT que construimos en el capítulo 12. Cada capa de decodificación procesa la secuencia de entrada e introduce un nivel de abstracción o representación. A medida que la información atraviesa más capas, el modelo es capaz de capturar patrones y relaciones más complejas en los datos. Esta profundidad es crucial para que nuestro Transformador musical comprenda y genere piezas musicales complejas.

El atributo `n_head` se establece en 8, lo que significa que dividiremos los vectores de consulta Q, clave K y valor V en ocho cabezas paralelas durante el cálculo de la autoatención causal. El atributo `n_embd` se establece en 512, lo que indica una dimensión de incrustación de 512: cada token de evento estará representado por un vector de 512 valores. El atributo `vocab_size` está determinado por el número de tokens únicos en el vocabulario, que es 390. Como se explicó anteriormente, hay 388 tokens de evento únicos, y agregamos 1 token para indicar el final de la secuencia y otro token para llenar secuencias más cortas para que todas las secuencias tengan una longitud de 2048. El atributo `block_size` se establece en 2048, lo que indica que la secuencia de entrada contiene un máximo de 2048 tokens. Establecimos las tasas de abandono en 0,1, como en los capítulos 11 y 12.

Como todos los Transformers, nuestro Transformer musical emplea mecanismos de autoatención para capturar las relaciones entre los diferentes elementos de una secuencia. Por lo tanto, definimos una clase `CausalSelfAttention()` en el módulo local `ch14util`, idéntica a la clase `CausalSelfAttention()` definida en el capítulo 12.

14.3.2 Construyendo un transformador de música

Combinamos una red de propagación hacia adelante con la subcapa de autoatención causal para formar un bloque decodificador (es decir, una capa decodificadora). Aplicamos normalización de capas y una conexión residual a cada subcapa para mejorar la estabilidad y el rendimiento. Para ello, defini-

mos una clase `Block()` en el módulo local para crear un bloque decodificador, idéntica a la clase `Block()` definida en el capítulo 12.

A continuación, apilamos seis bloques decodificadores uno encima del otro para formar el cuerpo principal de nuestro Transformador musical. Para ello, definimos una clase `Model()` en el módulo local. Como en todos los modelos GPT que hemos visto en este libro, utilizamos la codificación posicional aprendida mediante la clase `Embedding()` en PyTorch, en lugar de la codificación posicional fija del artículo original de 2017 «Attention Is All You Need». Consulta el capítulo 11 para conocer las diferencias entre los dos métodos de codificación posicional.

La entrada del modelo consiste en secuencias de índices correspondientes a los tokens de eventos musicales del vocabulario. Pasamos la entrada por incrustación de palabras y codificación posicional, y sumamos ambas para formar la incrustación de entrada. Esta incrustación pasa por las seis capas del decodificador. Después, aplicamos la normalización de capas a la salida y le conectamos un cabezal lineal para que el número de salidas sea 390, el tamaño del vocabulario. Las salidas son los logits correspondientes a los 390 tokens del vocabulario. Posteriormente, aplicaremos la función de activación softmax a los logits para obtener la distribución de probabilidad sobre los tokens musicales únicos del vocabulario al generar música.

A continuación, crearemos nuestro Transformer musical instanciando la clase `Model()` que definimos en el módulo local:

```
from utils.ch14util import Model

model=Model(config)
model.to(device)
num=sum(p.numel() for p in model.transformer.parameters())
print("number of parameters: %.2fM" % (num/1e6,))
print(model)
```

La salida es

```
number of parameters: 20.16M
Model(
  (transformer): ModuleDict(
    (wte): Embedding(390, 512)
    (wpe): Embedding(2048, 512)
```

```

(drop): Dropout(p=0.1, inplace=False)
(h): ModuleList(
    (0-5): 6 x Block(
        (ln_1): LayerNorm((512,), eps=1e-05,
            elementwise_affine=True)
        (attn): CausalSelfAttention(
            (c_attn): Linear(in_features=512, out_features=1536,
                bias=True)
            (c_proj): Linear(in_features=512, out_features=512,
                bias=True)
            (attn_dropout): Dropout(p=0.1, inplace=False)
            (resid_dropout): Dropout(p=0.1, inplace=False)
        )
        (ln_2): LayerNorm((512,), eps=1e-05,
            elementwise_affine=True)
        (mlp): ModuleDict(
            (c_fc): Linear(in_features=512, out_features=2048,
                bias=True)
            (c_proj): Linear(in_features=2048, out_features=512,
                bias=True)
            (act): GELU()
            (dropout): Dropout(p=0.1, inplace=False)
        )
    )
)
(ln_f): LayerNorm((512,), eps=1e-05,
    elementwise_affine=True)
)
(lm_head): Linear(in_features=512, out_features=390,
    bias=False)
)

```

Nuestro Transformer musical consta de 20,16 millones de parámetros, una cifra considerablemente menor que la del GPT-2XL, que cuenta con más de 1500 millones de parámetros. Sin embargo, nuestro Transformer musical supera el tamaño del generador de texto que construimos en el capítulo 12, que contiene tan solo 5,12 millones de parámetros. A pesar de estas diferencias, los tres modelos se basan en la arquitectura Transformer, que solo utiliza decodificador. Las variaciones residen únicamente en los hiperparámetros, como la dimensión de incrustación, el número de capas del decodificador, el tamaño del vocabulario, etc.

14.4 Entrenamiento y uso del transformador de música

En esta sección, entrenarás el transformador musical que acabas de construir con los lotes de datos de entrenamiento que preparamos anteriormente en este capítulo. Para agilizar el proceso, entrenaremos el modelo durante 100 épocas y luego detendremos el entrenamiento. Si te interesa, puedes utilizar el conjunto de validación para determinar cuándo detener el entrenamiento, según el rendimiento del modelo en dicho conjunto, como hicimos en el capítulo 2.

Una vez entrenado el modelo, le proporcionaremos un mensaje en forma de secuencia de índices. A continuación, solicitaremos al transformador musical entrenado que genere el siguiente índice. Este nuevo índice se añade al mensaje y el mensaje actualizado se devuelve al modelo para otra predicción. Este proceso se repite iterativamente hasta que la secuencia alcanza una longitud determinada.

A diferencia de la música generada en el capítulo 13, podemos controlar la creatividad de la pieza musical aplicando diferentes temperaturas.

14.4.1 Entrenamiento del transformador musical

Como siempre, usaremos el optimizador Adam para el entrenamiento. Dado que nuestro transformador musical ejecuta esencialmente una tarea de clasificación multicategoría, utilizaremos la pérdida de entropía cruzada como función de pérdida:

```
lr=0.0001  
optimizer = torch.optim.Adam(model.parameters(), lr=lr)  
loss_func=torch.nn.CrossEntropyLoss(ignore_index=389)
```

El argumento `ignore_index=389` en la función de pérdida anterior le indica al programa que ignore el índice 389 siempre que aparezca en la secuencia de destino (es decir, la secuencia y), ya que este índice se usa únicamente para fines de relleno y no representa ningún token de evento específico en la pieza musical.

Luego entrenaremos el modelo durante 100 épocas.

Listado 14.6 Entrenamiento del transformador de música para generar música

```
model.train()
for i in range(1,101):
    tloss = 0.
    for idx, (x,y) in enumerate(trainloader):          ①
        x,y=x.to(device),y.to(device)
        output = model(x)
        loss=loss_func(output.view(-1,output.size(-1)),
                        y.view(-1))                         ②
        optimizer.zero_grad()
        loss.backward()
        nn.utils.clip_grad_norm_(model.parameters(),1)  ③
        optimizer.step()                            ④
        print(f'epoch {i} loss {tloss/(idx+1)}')
    torch.save(model.state_dict(),f'files/musicTrans.pth')  ⑤
```

- ① Itera a través de todos los lotes de datos de entrenamiento
- ② Compara las predicciones del modelo con los resultados reales
- ③ Recorta la norma del gradiente a 1
- ④ Ajusta los parámetros del modelo para minimizar la pérdida
- ⑤ Guarda el modelo después del entrenamiento

Durante el entrenamiento, introducimos todas las secuencias de entrada x de un lote en el modelo para obtener predicciones. A continuación, comparamos estas predicciones con las secuencias de salida y correspondientes del lote y calculamos la pérdida de entropía cruzada. Después, ajustamos los parámetros del modelo para minimizar esta pérdida. Es importante destacar que hemos recortado la norma de gradiente a 1 para evitar el posible problema de gradientes explosivos.

El proceso de entrenamiento descrito anteriormente tarda aproximadamente 3 horas si tiene una GPU compatible con CUDA. Tras el entrenamiento, los pesos del modelo entrenado, musicTrans.pth, se guardan en su ordenador. También puede descargar los pesos entrenados desde mi sitio web:
<https://mng.bz/V2pW>.

14.4.2 Generación de música con el Transformer entrenado

Ahora que tenemos un Transformer musical entrenado, podemos proceder con la generación de música.

Similar al proceso de generación de texto, la generación de música comienza introduciendo una secuencia de índices (que representan tokens de eventos) al modelo como mensaje. Seleccionaremos una pieza musical del conjunto de prueba y usaremos los primeros 250 eventos musicales como mensaje:

```
from utils.processor import decode_midi

prompt, _ = test[42]
prompt = prompt.to(device)
len_prompt=250
file_path = "files/prompt.midi"
decode_midi(prompt[:len_prompt].cpu().numpy(),
            file_path=file_path)
```

Seleccionamos aleatoriamente un índice (42, en nuestro caso) y lo usamos para recuperar una canción del subconjunto de prueba. Conservamos solo los primeros 250 eventos musicales, que posteriormente alimentaremos al modelo entrenado para predecir los siguientes. Para fines de comparación, guardaremos el aviso como un archivo MIDI, prompt.midi, en la carpeta local.

Ejercicio 14.2

Utilice la función `decode_midi()` para convertir los primeros 250 eventos musicales de la segunda pieza musical del conjunto de prueba en un archivo MIDI. Guárdelo como prompt2.midi en su computadora.

Para agilizar el proceso de generación de música, definiremos una función `sample()`. Esta función acepta una secuencia de índices como entrada, que representa una pieza musical corta. A continuación, predice iterativamente y añade nuevos índices a la secuencia hasta `seq_length` alcanzar una longitud especificada. La implementación se muestra en la siguiente lista.

Listado 14.7 Una función `sample()` en la generación de música

```

softmax=torch.nn.Softmax(dim=-1)
def sample(prompt,seq_length=1000,temperature=1):
    gen_seq=torch.full((1,seq_length),389,dtype=torch.long).
        to(device)
    idx=len(prompt)
    gen_seq[..., :idx]=prompt.type(torch.long).to(device)
    while(idx < seq_length):
        y=softmax(model(gen_seq[..., :idx])/temperature)[..., :388] ①②
        probs=y[:, idx-1, :]
        distrib=torch.distributions.categorical.
            Categorical(probs=probs)
        next_token=distrib.sample()
③
        gen_seq[:, idx]=next_token
        idx+=1
④
return gen_seq[:, :idx]

```

① Genera los nuevos índices hasta que la secuencia alcanza una longitud determinada

② Divide la predicción por la temperatura y luego aplica la función softmax en los logits

③ Muestras de la distribución de probabilidad prevista para generar un nuevo índice

④ Genera la secuencia completa

Uno de los parámetros de la función `sample()` es la temperatura, que regula la creatividad de la música generada. Si es necesario, consulte el capítulo 8 para saber cómo funciona. Dado que podemos ajustar la originalidad y diversidad de la música generada únicamente con el parámetro de temperatura, `top-K` en este caso hemos omitido el muestreo para simplificar. Como ya hemos tratado el muestreo `top-K` tres veces en este libro (en los capítulos 8, 11 y 12), los lectores interesados pueden experimentar incorporando el muestreo `top-K` a la función `sample()`.

A continuación, cargaremos los pesos entrenados en el modelo:

```

model.load_state_dict(torch.load("files/musicTrans.pth",
    map_location=device))
model.eval()

```

Luego llamamos a la función `sample()` para generar una pieza musical:

```
from utils.processor import encode_midi

file_path = "files/prompt.midi"
prompt = torch.tensor(encode_midi(file_path))
generated_music=sample(prompt, seq_length=1000)
```

Primero, utilizamos la función `encode_midi()` del módulo `processor.py` para convertir el archivo MIDI, `prompt.midi`, en una secuencia de índices. Luego, usamos esta secuencia como indicador en la función `sample()` para generar una pieza musical compuesta por 1000 índices.

Finalmente, convertimos la secuencia de índices generada al formato MIDI:

```
music_data = generated_music[0].cpu().numpy()
file_path = 'files/musicTrans.midi'
decode_midi(music_data, file_path=file_path)
```

Utilizamos la función `decode_midi()` del módulo `processor.py` para transformar la secuencia de índices generada en un archivo MIDI, `musicTrans.midi`, en su ordenador. Abra ambos archivos, `prompt.midi` y `musicTrans.midi`, en su ordenador y escúchelos. La música de `prompt.midi` dura unos 10 segundos. La música de `musicTrans.midi` dura unos 40 segundos, y los últimos 30 segundos son música nueva generada por el Transformador de música. La música generada debería sonar como la pieza musical de mi sitio web:

<https://mng.bz/x6dg>.

El bloque de código anterior puede producir un resultado similar al siguiente:

```
info removed pitch: 52
info removed pitch: 83
info removed pitch: 55
info removed pitch: 68
```

En la música generada, puede haber casos en los que sea necesario eliminar ciertas notas. Por ejemplo, si la pieza musical generada intenta desactivar la nota 52, pero esta nunca se activó inicialmente, no podemos desactivarla. Por lo tanto, debemos eliminar dichas notas.

Ejercicio 14.3

Genere una pieza musical de 1200 notas utilizando el modelo Music Transformer entrenado, manteniendo el parámetro de temperatura en 1. Utilice la secuencia de índices del archivo prompt2.midi que acaba de generar en el ejercicio 14.2 como indicador. Guarde la música generada en un archivo llamado musicTrans2.midi en su computadora.

Puedes aumentar la creatividad de la música estableciendo el argumento de temperatura en un valor mayor que 1, de la siguiente manera:

```
file_path = "files/prompt.midi"
prompt = torch.tensor(encode_midi(file_path))
generated_music=sample(prompt, seq_length=1000,temperature=1.5)
music_data = generated_music[0].cpu().numpy()
file_path = 'files/musicHiTemp.midi'
decode_midi(music_data, file_path=file_path)
```

Ajustamos la temperatura a 1.5. La música generada se guarda como musicHiTemp.midi en tu ordenador. Abre el archivo y escucha la música generada para ver si detectas alguna diferencia con la música del archivo musicTrans.midi.

Ejercicio 14.4

Genere una pieza musical de 1000 índices utilizando el modelo Music Transformer entrenado, estableciendo el parámetro de temperatura en 0.7. Use la secuencia de índices del archivo prompt.midi como indicador. Guarde la música generada en un archivo llamado musicLowTemp.midi en su computadora. Abra este archivo para escuchar la música generada y comprobar si hay diferencias apreciables entre la nueva pieza musical y la del archivo musicTrans.midi.

En este capítulo, aprendiste a construir y entrenar un Transformer musical desde cero, basado en la arquitectura de Transformer basada únicamente en decodificador que usaste en capítulos anteriores. En el siguiente capítulo, explorarás los modelos basados en difusión, que son la base de los Transformers de texto a imagen, como DALL-E 2 de OpenAI e Imagen de Google.

Resumen

- La representación musical basada en la interpretación nos permite representar una pieza musical como una secuencia de notas, que incluye mensajes de control y valores de velocidad. Estas notas pueden resumirse en cuatro tipos de eventos musicales: nota activada, nota desactivada, desplazamiento temporal y velocidad. Cada tipo de evento puede asumir diversos valores. En consecuencia, podemos transformar una pieza musical en una secuencia de tokens y, posteriormente, en índices.
- Un transformador musical adapta la arquitectura Transformer, diseñada originalmente para tareas de PLN, para la generación musical. Este modelo está diseñado para generar secuencias de notas musicales aprendiendo de un amplio conjunto de datos musicales existentes. Está entrenado para predecir la siguiente nota en una secuencia basándose en notas anteriores, reconociendo patrones, estructuras y relaciones entre diversos elementos musicales en los datos de entrenamiento.
- Al igual que en la generación de texto, podemos utilizar la temperatura para regular la creatividad de la música generada.

¹ Chloe Veltman, 15 de marzo de 2024. “El hecho de que tu cantante favorito esté muerto no significa que no puedas verlo 'en vivo'”. <https://mng.bz/r1de>.

² Cheng-Zhi Anna Huang, Ashish Vaswani, Jakob Uszkoreit, Noam Shazeer, Ian Simon, Curtis Hawthorne, Andrew M. Dai, Matthew D. Hoffman, Monica Dinculescu y Douglas Eck, 2018, “Music Transformer”.

<https://arxiv.org/abs/1809.04281>.

³ Véase, por ejemplo, Hawthorne et al., 2018, “Habilitación del modelado y la generación de música de piano factorizada con el conjunto de datos MAESTRO”. <https://arxiv.org/abs/1810.12247>.

15. Modelos de difusión y transformadores de texto a imagen

Este capítulo cubre

- Cómo funcionan la difusión hacia adelante y la difusión hacia atrás
- Cómo construir y entrenar un modelo U-Net de eliminación de ruido
- Uso de la U-Net entrenada para generar imágenes de flores
- Conceptos detrás de los transformadores de texto a imagen
- Escritura de un programa en Python para generar una imagen a través de texto con DALL-E 2

En los últimos años, los modelos de lenguaje grande (LLM) multimodales han cobrado gran importancia por su capacidad para gestionar diversos formatos de contenido, como texto, imágenes, vídeo, audio y código. Un ejemplo notable de ello son los transformadores de texto a imagen, como DALL-E 2 de OpenAI, Imagen de Google y Stable Diffusion de Stability AI. Estos modelos son capaces de generar imágenes de alta calidad a partir de descripciones textuales.

Estos modelos de texto a imagen constan de tres componentes esenciales: un codificador de texto que comprime el texto en una representación latente, un método para incorporar información textual al proceso de generación de imágenes y un mecanismo de difusión para refinar gradualmente la imagen y producir un resultado realista. Comprender el mecanismo de difusión es crucial para comprender los transformadores de texto a imagen, ya que estos forman la base de todos los transformadores de texto a imagen líderes. Por ello, en este capítulo, comenzarás construyendo y entrenando un modelo de difusión para generar imágenes de flores. Esto te proporcionará una comprensión profunda del proceso de difusión hacia adelante, donde el ruido se añade gradualmente a las imágenes hasta que se convierten en ruido aleatorio. Posteriormente, entrenarás un modelo para revertir el proceso de difusión eliminando gradualmente el ruido de las imágenes hasta que el modelo pueda generar una nueva imagen limpia a partir del ruido aleatorio, similar a las del conjunto de datos de entrenamiento.

Los modelos de difusión se han convertido en la opción predilecta para generar imágenes de alta resolución. Su éxito reside en su capacidad para simular y revertir un complejo proceso de adición de ruido, lo que simula una comprensión profunda de la estructura de las imágenes y su construcción a partir de patrones abstractos. Este método no solo garantiza una alta calidad, sino que también mantiene un equilibrio entre diversidad y precisión en las imágenes generadas.

A continuación, explicaremos el funcionamiento conceptual de un transformador de texto a imagen. Nos centraremos en el modelo de preentrenamiento contrastivo de lenguaje e imagen (CLIP), desarrollado por OpenAI, diseñado para comprender y vincular información visual y textual. CLIP procesa dos tipos de entradas: imágenes y texto (normalmente en forma de subtítulos o descripciones). Estas entradas se gestionan por separado mediante dos codificadores del modelo.

La rama de imágenes de CLIP emplea un Transformador de Visión (ViT) para codificar imágenes en un espacio vectorial de alta dimensión, extrayendo características visuales en el proceso. Por otro lado, la rama de texto utiliza un modelo de lenguaje basado en Transformadores para codificar descripciones textuales en el mismo espacio vectorial, capturando características semánticas del texto. CLIP se ha entrenado con numerosos pares de imágenes y descripciones de texto coincidentes para alinear estrechamente las representaciones de los pares coincidentes en el espacio vectorial.

Los transformadores de texto a imagen de OpenAI, como DALL-E 2, incorporan CLIP como componente principal. En este capítulo, aprenderá a obtener una clave API de OpenAI y a escribir un programa en Python para generar imágenes con DALL-E 2 a partir de descripciones de texto.

15.1 Introducción a los modelos de difusión de eliminación de ruido

El concepto de modelos basados en la difusión se puede ilustrar con el siguiente ejemplo. Considere el objetivo de generar imágenes de flores de alta resolución mediante un modelo basado en la difusión. Para ello, primero se adquiere un conjunto de imágenes de flores de alta calidad para el entrenamiento. A continuación, se le indica al modelo que introduzca gradualmente pequeñas cantidades de ruido aleatorio en estas imágenes, un proceso cono-

cido como difusión hacia adelante. Tras varios pasos de adición de ruido, las imágenes de entrenamiento se convierten en ruido aleatorio. La siguiente fase consiste en entrenar el modelo para revertir este proceso, comenzando con imágenes de ruido puro y reduciéndolo progresivamente hasta que las imágenes sean indistinguibles de las del conjunto de entrenamiento original.

Una vez entrenado, el modelo recibe imágenes de ruido aleatorio con las que trabajar. Elimina sistemáticamente el ruido de la imagen a lo largo de numerosas iteraciones hasta generar una imagen de flor de alta resolución similar a las del conjunto de entrenamiento. Este es el principio fundamental de los modelos basados en la difusión.¹

En esta sección, primero explorará los fundamentos matemáticos de los modelos basados en difusión. A continuación, se adentrará en la arquitectura de las U-Nets, el tipo de modelo utilizado para eliminar el ruido de las imágenes y producir imágenes de flores de alta resolución. En concreto, la U-Net emplea un mecanismo de atención de producto escalable (SDPA), similar al que ha visto en los modelos Transformer de los capítulos 9 a 12. Finalmente, aprenderá el proceso de entrenamiento de los modelos basados en difusión y el proceso de generación de imágenes del modelo entrenado.

15.1.1 El proceso de difusión hacia adelante

Varios artículos han propuesto modelos basados en la difusión con mecanismos subyacentes similares.² Utilicemos las imágenes de flores como ejemplo concreto para explicar la idea detrás de los modelos de difusión con eliminación de ruido. La Figura 15.1 muestra un diagrama del funcionamiento del proceso de difusión hacia adelante.

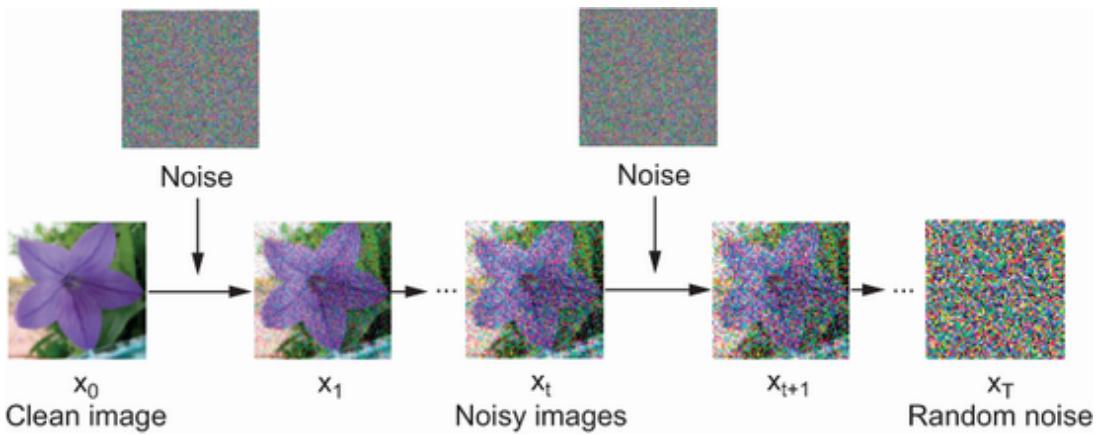


Figura 15.1 Diagrama del proceso de difusión hacia adelante. Partimos de una imagen limpia del conjunto de entrenamiento, x_0 , y le añadimos ruido ϵ_0 para formar una imagen ruidosa $x_1 = \sqrt{1 - \beta_1} x_0 + \sqrt{\beta_1} \epsilon_0$. Repetimos este proceso durante 1000 intervalos de tiempo hasta que la imagen x_{1000} se convierta en ruido aleatorio.

Supongamos que las imágenes de flores, x_0 (ilustradas en la imagen izquierda de la figura 15.1), siguen una distribución de $q(x)$. En el proceso de difusión hacia adelante, añadiremos pequeñas cantidades de ruido a las imágenes en cada uno de los pasos $T = 1000$. El tensor de ruido se distribuye normalmente y tiene la misma forma que las imágenes de flores: $(3, 64, 64)$, lo que significa tres canales de color, con una altura y una anchura de 64 píxeles.

Pasos de tiempo en modelos de difusión

En los modelos de difusión, los pasos de tiempo se refieren a las etapas discretas durante el proceso de añadir gradualmente ruido a los datos y su posterior reversión para generar muestras. La fase directa de un modelo de difusión añade ruido progresivamente a lo largo de una serie de pasos de tiempo, transformando los datos de su estado original y limpio a una distribución ruidosa. Durante la fase inversa, el modelo opera en una serie similar de pasos de tiempo, pero en orden inverso. Elimina sistemáticamente el ruido de los datos para reconstruir el original o generar nuevas muestras de alta fidelidad. Cada paso de tiempo en este proceso inverso implica predecir el ruido añadido en el paso directo correspondiente y restarlo, eliminando así gradualmente el ruido de los datos hasta alcanzar un estado limpio.

En el paso de tiempo 1, agregamos ruido ϵ_0 a la imagen x_0 , de modo que obtenemos una imagen ruidosa x_1 :

$$x_1 = \sqrt{1 - \beta_1}x_0 + \sqrt{\beta_1}\epsilon_0 \quad (15.1)$$

Es decir, x_1 es una suma ponderada de x_0 y ϵ_0 , donde β_1 mide el peso asignado al ruido. El valor de β cambia en diferentes pasos de tiempo; de ahí el subíndice en β_1 . Si asumimos que x_0 y ϵ_0 son independientes entre sí y siguen una distribución normal estándar (es decir, con media 0 y varianza 1), la imagen ruidosa x_1 también seguirá una distribución normal estándar. Esto es fácil de demostrar, ya que

$$\text{mean}(x_1) = \sqrt{1 - \beta_1}\text{mean}(x_0) + \sqrt{\beta_1}\text{mean}(\epsilon_0) = 0$$

y

$$\begin{aligned} \text{var}(x_1) &= \text{var}\left(\sqrt{1 - \beta_1}x_0\right) + \text{var}\left(\sqrt{\beta_1}\epsilon_0\right) \\ &= (1 - \beta_1)\text{var}(x_0) + \beta_1\text{var}(\epsilon_0) = 1 - \beta_1 + \beta_1 = 1 \end{aligned}$$

Podemos seguir añadiendo ruido a la imagen durante los siguientes pasos de tiempo $T-1$ de modo que

$$x_{t+1} = \sqrt{1 - \beta_{t+1}}x_t + \sqrt{\beta_{t+1}}\epsilon_t \quad (15.2)$$

Podemos usar un truco de reparametrización y definir $\alpha_t = 1 - \beta_t$ y

$$\bar{\alpha}_t = \prod_{k=1}^t \alpha_k$$

para permitirnos muestrear x_t en cualquier paso de tiempo arbitrario t , donde t puede tomar cualquier valor en $[1, 2, \dots, T-1, T]$. Entonces tenemos

$$x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1 - \bar{\alpha}_t} \epsilon \quad (15.3)$$

Donde ϵ es una combinación de $\epsilon_0, \epsilon_1, \dots$, y ϵ_{t-1} , aprovechando que podemos sumar dos distribuciones normales para obtener una nueva distribución normal. Véase, por ejemplo, el blog de Lilian Weng en <https://mng.bz/Aalg> para una demostración.

El extremo izquierdo de la figura 15.1 muestra una flor limpia, x_0 , del conjunto de entrenamiento. En el primer paso de tiempo, le inyectamos ruido ϵ_0 para formar una imagen ruidosa x_1 (segunda imagen en la figura 15.1). Repetimos este proceso durante 1000 pasos de tiempo, hasta que la imagen se convierte en ruido aleatorio (la imagen más a la derecha).

15.1.2 Uso del modelo U-Net para eliminar el ruido de las imágenes

Ahora que comprende el proceso de difusión hacia adelante, analicemos el proceso de difusión inversa (es decir, el proceso de eliminación de ruido). Si podemos entrenar un modelo para revertir el proceso de difusión hacia adelante, podemos alimentarlo con ruido aleatorio y pedirle que produzca una imagen de flor ruidosa. Luego, podemos volver a alimentar la imagen ruidosa al modelo entrenado y producir una imagen más clara, aunque aún ruidosa. Podemos repetir iterativamente el proceso durante varios pasos de tiempo hasta obtener una imagen limpia, indistinguible de las imágenes del conjunto de entrenamiento. El uso de múltiples pasos de inferencia en el proceso de difusión inversa, en lugar de un solo paso, es crucial para reconstruir gradualmente datos de alta calidad a partir de una distribución ruidosa. Permite una generación de datos más controlada, estable y de alta calidad.

Para ello, crearemos un modelo U-Net de denoising. La arquitectura U-Net, diseñada originalmente para la segmentación de imágenes biomédicas, se caracteriza por su forma simétrica, con una trayectoria de contracción (codificador) y una trayectoria de expansión (decodificador), conectadas por una capa de cuello de botella. En el contexto de la denoising, los modelos U-Net están adaptados para eliminar el ruido de las imágenes, preservando al mismo tiempo los detalles importantes. Las U-Nets superan a las redes con-

volucionales simples en tareas de denoising gracias a su eficiente captura de características locales y globales en las imágenes.

La figura 15.2 es un diagrama de la estructura de la U-Net de eliminación de ruido que utilizamos en este capítulo.

El modelo toma como entrada una imagen con ruido y el paso de tiempo en el que se encuentra (x_t , y t en la ecuación 15.3) y predice el ruido en la imagen (es decir, ϵ). Dado que la imagen con ruido es una suma ponderada de la imagen original limpia y el ruido (véase la ecuación 15.3), conocer el ruido nos permite deducir y reconstruir la imagen original.

La ruta de contracción (es decir, el codificador; lado izquierdo de la figura 15.2) consta de múltiples capas convolucionales y capas de agrupamiento. Reduce progresivamente el tamaño de la imagen, extrayendo y codificando características en diferentes niveles de abstracción. Esta parte de la red aprende a reconocer patrones y características relevantes para la eliminación de ruido.

La capa de cuello de botella (parte inferior de la figura 15.2) conecta las rutas del codificador y del decodificador. Consta de capas convolucionales y se encarga de capturar las representaciones más abstractas de la imagen.

La ruta expansiva (es decir, el decodificador; lado derecho de la figura 15.2) consta de capas de sobremuestreo y capas convolucionales. Sobremuestra progresivamente los mapas de características, reconstruyendo la imagen a la vez que incorpora características del codificador mediante conexiones de salto. Las conexiones de salto (indicadas con líneas discontinuas en la figura 15.2) son cruciales en los modelos U-Net, ya que permiten que el modelo conserve detalles precisos de la imagen de entrada al combinar características de bajo y alto nivel. A continuación, explico brevemente cómo funcionan las conexiones de salto.

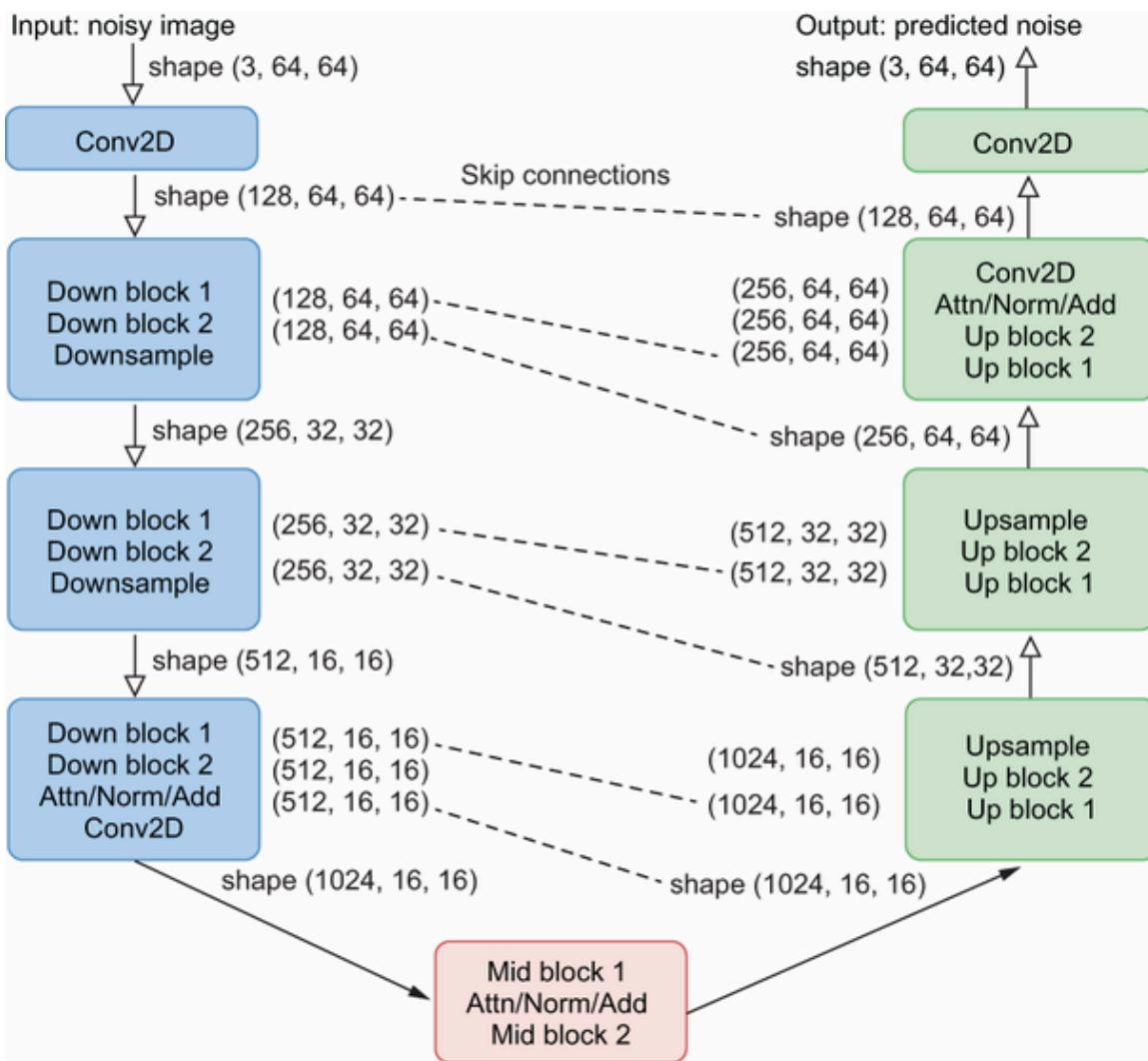


Figura 15.2 Arquitectura del modelo U-Net de eliminación de ruido. La arquitectura U-Net se caracteriza por su forma simétrica, con una trayectoria de contracción (codificador) y una trayectoria de expansión (decodificador), conectadas por una capa de cuello de botella. El modelo está diseñado para eliminar el ruido de las imágenes, preservando al mismo tiempo los detalles importantes. La entrada del modelo es una imagen con ruido, junto con el paso de tiempo en el que se encuentra, y la salida es el ruido predicho en la imagen.

En un modelo U-Net, las conexiones de salto se implementan concatenando los mapas de características de la ruta del codificador con los correspondientes en la ruta del decodificador. Estos mapas de características suelen tener las mismas dimensiones espaciales, pero pueden haber sido procesados de forma diferente debido a las distintas rutas que han recorrido. Durante el proceso de codificación, la imagen de entrada se reduce progresivamente de resolución, lo que puede provocar la pérdida de información espacial (como bordes y texturas). Las conexiones de salto ayudan a preservar esta información al pasar directamente los mapas de características del codificador al decodificador, evitando así el cuello de botella de información.

Por ejemplo, la línea discontinua en la parte superior de la figura 15.2 indica que el modelo concatena la *salida* de la capa Conv2D del codificador, cuya forma es (128, 64, 64), con la *entrada* de la capa Conv2D del decodificador,

cuya forma también es (128, 64, 64). Como resultado, la entrada final de la capa Conv2D del decodificador tiene la forma (256, 64, 64).

Al combinar las características abstractas de alto nivel del decodificador con las características detalladas de bajo nivel del codificador, las conexiones de salto permiten al modelo reconstruir mejor los detalles finos en la imagen denominada. Esto es especialmente importante en las tareas de denominación, donde la conservación de los detalles sutiles de la imagen es crucial.

El mecanismo de atención de producto escalar (SDPA) se implementa tanto en el bloque final de la trayectoria de contracción como en el bloque final de la trayectoria de expansión en nuestro modelo U-Net de eliminación de ruido, acompañado de normalización de capas y conexiones residuales (como se muestra en la figura 15.2 con la etiqueta Attn/Norm/Add). Este mecanismo SDPA es esencialmente el mismo que desarrollamos en el capítulo 9; la diferencia clave radica en su aplicación a píxeles de imagen en lugar de tokens de texto.

El uso de conexiones de salto y el tamaño del modelo generan extracciones de características redundantes en nuestra U-Net de eliminación de ruido, lo que garantiza que no se pierda ninguna característica importante durante el proceso. Sin embargo, el gran tamaño del modelo también dificulta la identificación de características relevantes, como si se buscara una aguja en un pajar. El mecanismo de atención permite al modelo enfatizar las características significativas e ignorar las irrelevantes, mejorando así la eficacia del proceso de aprendizaje.

15.1.3 Un plan para entrenar el modelo de eliminación de ruido U-Net

La salida de la U-Net de denoising es el ruido inyectado en la imagen ruidosa. El modelo se entrena para minimizar la diferencia entre la salida (ruido predicho) y la realidad fundamental (ruido real).

El modelo de eliminación de ruido U-Net aprovecha la capacidad de la arquitectura U-Net para capturar el contexto local y global, lo que lo hace eficaz para eliminar el ruido y preservar detalles importantes como bordes y texturas. Estos modelos se utilizan ampliamente en diversas aplicaciones, como la eliminación de ruido de imágenes médicas, la restauración de imá-

genes fotográficas y otras. La Figura 15.3 muestra un diagrama del proceso de entrenamiento de nuestro modelo de eliminación de ruido U-Net.

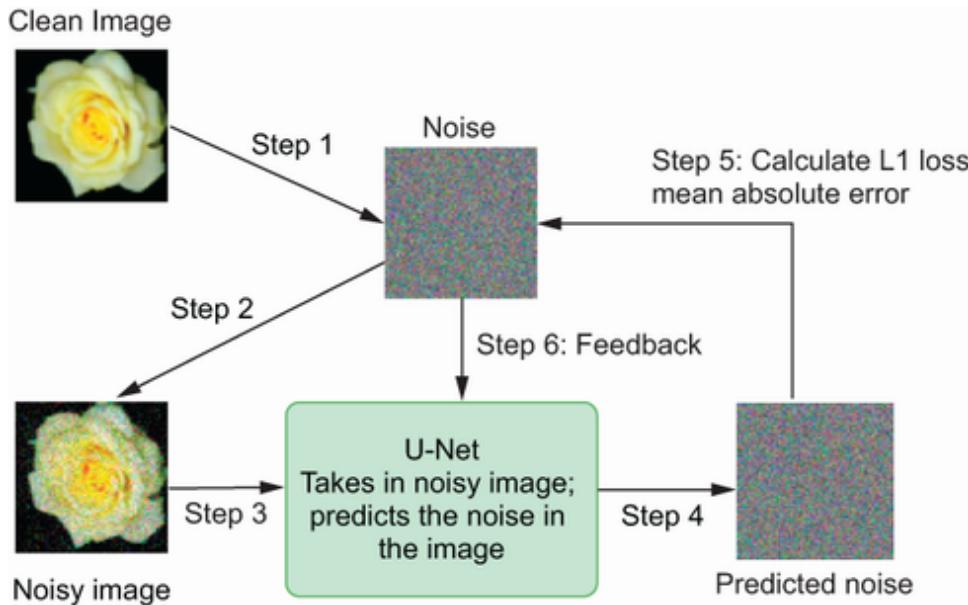


Figura 15.3 Proceso de entrenamiento del modelo U-Net de eliminación de ruido. Primero, obtenemos imágenes limpias de flores como conjunto de entrenamiento. Añadimos ruido a las imágenes limpias de flores y las presentamos al modelo U-Net. El modelo predice el ruido en las imágenes con ruido. Comparamos el ruido predicho con el ruido real injectado en las imágenes de flores y ajustamos los pesos del modelo para minimizar el error absoluto medio.

El primer paso es recopilar un conjunto de datos de imágenes de flores. Utilizaremos el conjunto de datos Oxford 102 Flower como conjunto de entrenamiento. Ajustaremos el tamaño de todas las imágenes a una resolución fija de 64×64 píxeles y normalizaremos los valores de los píxeles en el rango $[-1, 1]$. Para la eliminación de ruido, necesitamos pares de imágenes limpias y ruidosas. Añadiremos ruido sintéticamente a las imágenes limpias de flores para crear sus contrapartes ruidosas (paso 2 de la figura 15.3), según la fórmula especificada en la ecuación 15.3.

A continuación, construiremos un modelo U-Net de eliminación de ruido con la estructura que se muestra en la figura 15.2. Durante cada época de entrenamiento, iteraremos el conjunto de datos por lotes. Añadimos ruido a las imágenes de flores y presentamos las imágenes con ruido al modelo U-Net (paso 3), junto con los intervalos de tiempo t en los que se encuentran las imágenes con ruido. El modelo U-Net predice el ruido en las imágenes con ruido (paso 4) basándose en los parámetros actuales del modelo.

Comparamos el ruido predicho con el real y calculamos la pérdida L1 (es decir, el error absoluto medio) a nivel de píxel (paso 5). En estas situaciones, se suele preferir la pérdida L1, ya que es menos sensible a los valores atípicos

que la pérdida L2 (error cuadrático medio). A continuación, ajustamos los parámetros del modelo para minimizar la pérdida L1 (paso 6) de modo que, en la siguiente iteración, el modelo realice mejores predicciones. Repetimos este proceso en varias iteraciones hasta que los parámetros del modelo converjan.

15.2 Preparación de los datos de entrenamiento

Utilizaremos el conjunto de datos Oxford 102 Flower, disponible gratuitamente en Hugging Face, como datos de entrenamiento. El conjunto de datos contiene aproximadamente 8000 imágenes de flores y se puede descargar directamente mediante la biblioteca *de conjuntos de datos* que instaló anteriormente.

Para ahorrar espacio, colocaremos la mayoría de las funciones y clases auxiliares en dos módulos locales: ch15util.py y unet_util.py. Descargue estos dos archivos del repositorio de GitHub del libro

(<https://github.com/markhliu/DGAI>) y colóquelos en la carpeta /utils/ de su ordenador. Los programas de Python de este capítulo están adaptados del repositorio de GitHub de Hugging Face (<https://github.com/huggingface/diffusers>) y del de Filip Basara (<https://github.com/filipbasara0/simple-diffusion>).

Usarás Python para descargar el conjunto de datos a tu computadora. Después, demostraremos el proceso de difusión hacia adelante añadiendo ruido gradualmente a las imágenes limpias del conjunto de datos de entrenamiento hasta que se conviertan en ruido aleatorio. Finalmente, colocarás los datos de entrenamiento en lotes para que podamos usarlos para entrenar el modelo U-Net de eliminación de ruido más adelante en este capítulo.

En este capítulo, utilizará las siguientes bibliotecas de Python: datasets, einops, diffusers y openai. Para instalarlas, ejecute la siguiente línea de código en una nueva celda de su aplicación Jupyter Notebook en su computadora:

```
!pip install datasets einops diffusers openai
```

Siga las instrucciones en pantalla para finalizar la instalación.

15.2.1 Imágenes de flores como datos de entrenamiento

El método `load_dataset()` de la biblioteca *de conjuntos de datos* que instalaste anteriormente te permite descargar directamente el conjunto de datos Oxford 102 Flower de Hugging Face. Luego, usaremos la biblioteca *matplotlib* para mostrar algunas imágenes de flores en el conjunto de datos y así tener una idea de cómo se ven las imágenes en el conjunto de datos de entrenamiento.

Ejecute las líneas de código que se muestran en la siguiente lista en una celda de Jupyter Notebook.

Listado 15.1 Descarga y visualización de imágenes de flores

```
from datasets import load_dataset
from utils.ch15util import transforms

dataset = load_dataset("huggan/flowers-102-categories",
    split="train",)                                         ①
dataset.set_transform(transforms)

import matplotlib.pyplot as plt
from torchvision.utils import make_grid

# Plot all the images of the 1st batch in grid
grid = make_grid(dataset[:16]["input"], 8, 2)             ②
plt.figure(figsize=(8,2),dpi=300)
plt.imshow(grid.numpy().transpose((1,2,0)))
plt.axis("off")
plt.show()
```

① Descarga las imágenes de Hugging Face

② Traza las primeras 16 imágenes

Tras ejecutar el código anterior, verá las primeras 16 imágenes de flores del conjunto de datos, como se muestra en la figura 15.4. Se trata de imágenes a color de alta resolución de varios tipos de flores. Hemos estandarizado el tamaño de cada imagen a (3, 64, 64).



Figura 15.4 Las primeras 16 imágenes del conjunto de datos Oxford 102 Flower.

Colocamos el conjunto de datos en lotes de 4 para usarlos posteriormente para entrenar el modelo U-Net de eliminación de ruido. Elegimos un tamaño de lote de 4 para mantener la memoria lo suficientemente pequeña como para que quepa en una GPU durante el entrenamiento. Ajuste el tamaño de lote a 2 o incluso a 1 si la memoria de su GPU es pequeña:

```
import torch
resolution=64
batch_size=4
train_dataloader=torch.utils.data.DataLoader(
    dataset, batch_size=batch_size, shuffle=True)
```

A continuación, codificaremos y visualizaremos el proceso de difusión hacia adelante.

15.2.2 Visualización del proceso de difusión hacia adelante

Hemos definido una clase `DDIMScheduler()` en el módulo local ch15util.py que acabas de descargar. Examina la clase en el archivo; la usaremos para añadir ruido a las imágenes. También la usaremos para generar imágenes limpias posteriormente, junto con el modelo de denoising U-Net entrenado. La clase `DDIMScheduler()` gestiona el tamaño y la secuencia de los pasos de denoising, lo que permite una inferencia determinista que puede producir muestras de alta calidad mediante el proceso de denoising.

Primero seleccionamos cuatro imágenes limpias del conjunto de entrenamiento y generamos tensores de ruido que tienen la misma forma que estas imágenes:

```
clean_images=next(iter(train_dataloader))["input"]*2-1      ①
print(clean_images.shape)
```

```
nums=clean_images.shape[0]
noise=torch.randn(clean_images.shape)
print(noise.shape)
```

2

- ① Obtiene cuatro imágenes limpias
 - ② Genera un tensor, ruido, que tiene la misma forma que las imágenes limpias; cada valor del ruido sigue una distribución normal estándar independiente.

La salida del bloque de código anterior es

```
torch.Size([4, 3, 64, 64])  
torch.Size([4, 3, 64, 64])
```

Tanto las imágenes como los tensores de ruido tienen una forma de (4, 3, 64, 64), lo que significa 4 imágenes en el lote y 3 canales de color por imagen, y la altura y el ancho de las imágenes son 64 píxeles.

Durante el proceso de difusión hacia adelante, existen 999 imágenes transicionales con ruido entre las imágenes limpias (x_0 , como se explicó en la primera sección) y el ruido aleatorio (x_T). Las imágenes transicionales con ruido son la suma ponderada de la imagen limpia y el ruido. A medida que t varía de 0 a 1000, la ponderación de la imagen limpia disminuye gradualmente y la ponderación del ruido aumenta gradualmente, como se especifica en la ecuación 15.3.

A continuación, generamos y visualizamos algunas imágenes ruidosas de transición.

Listado 15.2 Visualización del proceso de difusión hacia adelante

```
allimgs=torch.cat((allimgs,noisy_images))
```

(4)

```
import torchvision  
imgs=torchvision.utils.make_grid(allimgs,4,6)  
fig = plt.figure(dpi=300)  
plt.imshow((imgs.permute(2,1,0)+1)/2)  
plt.axis("off")  
plt.show()
```

(5)

① Instancia la clase DDIMScheduler() con 1000 pasos de tiempo

② Observa los pasos de tiempo 200, 400, 600, 800 y 1000

③ Crea imágenes ruidosas en estos pasos de tiempo

④ Concatena imágenes ruidosas con imágenes limpias

⑤ Muestra todas las imágenes

El método `add_noise()` de la clase `DDIMScheduler()` toma tres argumentos: `clean_images`, `noise`, y `timesteps`. Produce una suma ponderada de la imagen limpia y el ruido, que es una imagen con ruido. Además, el peso es una función del intervalo de tiempo t . A medida que el intervalo de tiempo t pasa de 0 a 1000, el peso de la imagen limpia disminuye y el del ruido aumenta. Si ejecuta el código anterior, verá una imagen similar a la de la figura 15.5.

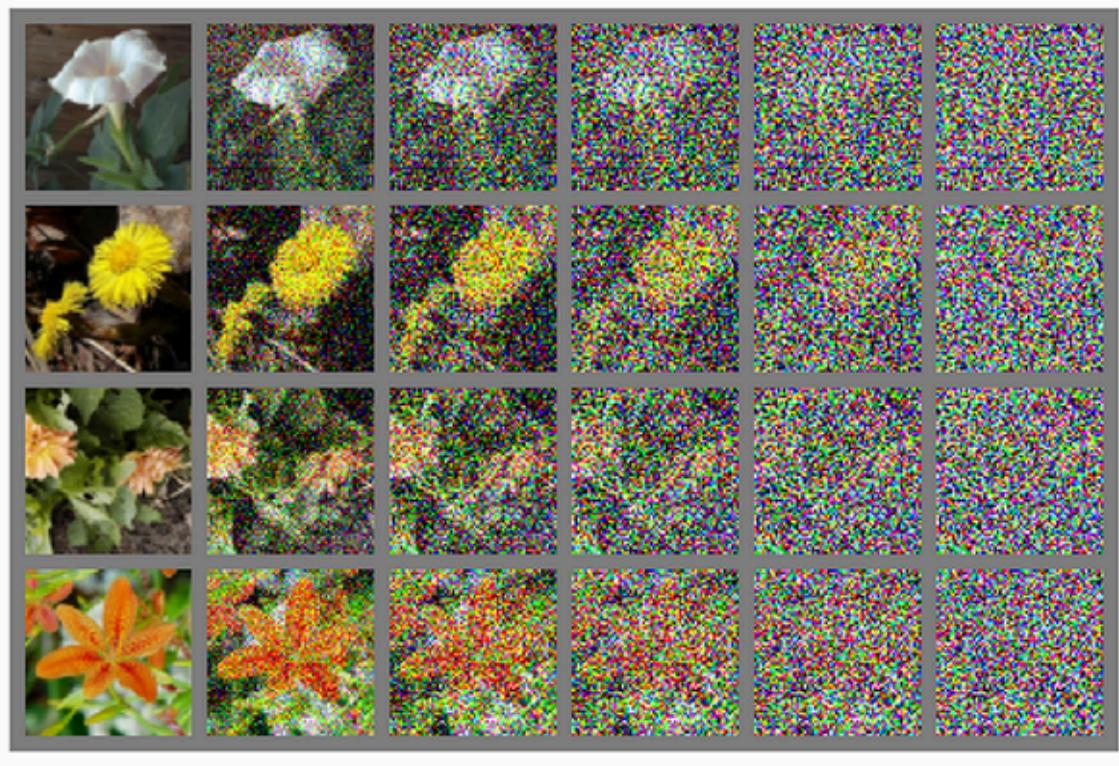


Figura 15.5 Proceso de difusión hacia adelante. Las cuatro imágenes de la primera columna son imágenes limpias del conjunto de datos de entrenamiento. A continuación, añadimos ruido gradualmente a estas imágenes desde el paso de tiempo 1 hasta el paso de tiempo 1000. A medida que aumenta el paso de tiempo, se inyecta más ruido en las imágenes. Las cuatro imágenes de la segunda columna corresponden a imágenes después de 200 pasos de tiempo. La tercera columna contiene imágenes después de 400 pasos de tiempo y tienen más ruido que las de la segunda columna. La última columna contiene imágenes después de 1000 pasos de tiempo y son 100 % ruido aleatorio.

La primera columna contiene las cuatro imágenes limpias y sin ruido. A medida que nos desplazamos hacia la derecha, añadimos gradualmente más ruido a las imágenes. La última columna contiene ruido aleatorio puro.

15.3 Construcción de un modelo U-Net de eliminación de ruido

Anteriormente en este capítulo, analizamos la arquitectura del modelo U-Net de eliminación de ruido. En esta sección, los guiaré en su implementación con Python y PyTorch.

El modelo U-Net que vamos a construir es bastante grande, con más de 133 millones de parámetros, lo que refleja la complejidad de su tarea. Está diseñado para capturar características locales y globales dentro de una imagen mediante un proceso de submuestreo y sobremuestreo de la entrada. El modelo utiliza múltiples capas convolucionales interconectadas mediante conexiones de salto, que combinan características de varios niveles de la red.

Esta arquitectura ayuda a mantener la información espacial, facilitando un aprendizaje más eficaz.

Dado el tamaño considerable del modelo U-Net de eliminación de ruido y su extracción de características redundante, se emplea el mecanismo de atención SDPA para que el modelo se centre en los aspectos más relevantes de la entrada para la tarea en cuestión. Para calcular la atención SDPA, aplazaremos la imagen y trataremos sus píxeles como una secuencia. Posteriormente, utilizaremos SDPA para aprender las dependencias entre los diferentes píxeles de la imagen, de forma similar a como aprendimos las dependencias entre los diferentes tokens del texto en el capítulo 9.

15.3.1 El mecanismo de atención en el modelo U-Net de eliminación de ruido

Para implementar el mecanismo de atención, hemos definido una clase `Attention()` en el módulo local `ch15util.py`, como se muestra en la siguiente lista de código.

Listado 15.3 El mecanismo de atención en el modelo U-Net de eliminación de ruido

```
import torch
from torch import nn, einsum
from einops import rearrange

class Attention(nn.Module):
    def __init__(self, dim, heads=4, dim_head=32):
        super().__init__()
        self.scale = dim_head**-0.5
        self.heads = heads
        hidden_dim = dim_head * heads
        self.to_qkv = nn.Conv2d(dim, hidden_dim * 3, 1,
                             bias=False)
        self.to_out = nn.Conv2d(hidden_dim, dim, 1)
    def forward(self, x):
        b, c, h, w = x.shape
        qkv = self.to_qkv(x).chunk(3, dim=1)
        q, k, v = map(
            lambda t: rearrange(t, 'b (h c) x y -> b h c (x y)', h=self.heads), qkv)
        q = q * self.scale
        sim = einsum('b h d i, b h d j -> b h i j', q, k)
        attn = sim.softmax(dim=-1)
        out = einsum('b h i j, b h d j -> b h i d', attn, v)
```

(1)
(2)
(3)
(4)

```
        out = rearrange(out, 'b h (x y) d -> b (h d) x y',
                     x=h, y=w)
        return self.to_out(out)
attn=Attention(128)
x=torch.rand(1,128,64,64)
out=attn(x)
print(out.shape)
```

(5)

- ① Pasa la entrada a través de tres capas lineales para obtener la consulta, la clave y el valor
- ② Divide la consulta, la clave y el valor en cuatro encabezados
- ③ Calcula los pesos de atención
- ④ Calcula el vector de atención en cada cabeza
- ⑤ Concatena los cuatro vectores de atención en uno

La salida después de ejecutar el código anterior es

```
torch.Size([1, 128, 64, 64])
```

El mecanismo de atención utilizado aquí, SDPA, es el mismo que utilizamos en el capítulo 9, donde aplicamos SDPA a una secuencia de índices que representan tokens en texto. En este caso, lo aplicamos a los píxeles de una imagen. Tratamos los píxeles aplanados de una imagen como una secuencia y usamos SDPA para extraer dependencias entre las diferentes áreas de la imagen de entrada, mejorando así la eficiencia del proceso de eliminación de ruido.

El Listado 15.3 demuestra cómo funciona SDPA en nuestro contexto. Para dar un ejemplo concreto, hemos creado una imagen hipotética, x , con dimensiones $(1, 128, 64, 64)$, lo que indica una imagen en el lote, 128 canales de características y un tamaño de 64×64 píxeles en cada canal. La entrada x se procesa a través de la capa de atención. Específicamente, cada canal de características de la imagen se aplana en una secuencia de $64 \times 64 = 4096$ píxeles. Esta secuencia se pasa a través de tres capas distintas de la red neuronal para generar la consulta Q, la clave K y el valor V, que posteriormente

se dividen en cuatro cabezas. El vector de atención en cada cabeza se calcula de la siguiente manera:

$$\text{Attention } (Q, K, V) = \text{softmax} \left(\frac{Q * K^T}{\sqrt{d_k}} \right) * V$$

donde d_k representa la dimensión del vector clave K. Los vectores de atención de las cuatro cabezas se concatenan nuevamente en un solo vector de atención.

15.3.2 El modelo U-Net de eliminación de ruido

En el módulo local `unet_util.py` que acaba de descargar, hemos definido una clase `UNet()` que representa el modelo de eliminación de ruido U-Net. Consulte la definición en el archivo; más adelante le daré una breve explicación de su funcionamiento. El siguiente código presenta una parte de la clase `UNet()`.

Listado 15.4 Definición de la clase `UNet()`

```
class UNet(nn.Module):
    ...
    def forward(self, sample, timesteps): ①
        if not torch.is_tensor(timesteps):
            timesteps = torch.tensor([timesteps],
                                      dtype=torch.long,
                                      device=sample.device)
        timesteps = torch.flatten(timesteps)
        timesteps = timesteps.broadcast_to(sample.shape[0])
        t_emb = sinusoidal_embedding(timesteps,
                                     self.hidden_dims[0])
        t_emb = self.time_embedding(t_emb)
        x = self.init_conv(sample)
        r = x.clone()
        skips = []
        for block1, block2, attn, downsample in self.down_blocks: ③
            x = block1(x, t_emb)
            skips.append(x)
            x = block2(x, t_emb)
            x = attn(x)
            skips.append(x)
```

```

        x = downsample(x)
        x = self.mid_block1(x, t_emb)
        x = self.mid_attn(x)
        x = self.mid_block2(x, t_emb)
        for block1, block2, attn, upsample in self.up_blocks:
            x = torch.cat((x, skips.pop()), dim=1)
            x = block1(x, t_emb)
            x = torch.cat((x, skips.pop()), dim=1)
            x = block2(x, t_emb)
            x = attn(x)
            x = upsample(x)
        x = self.out_block(torch.cat((x, r), dim=1), t_emb)
        out = self.conv_out(x)
        return {"sample": out}

```

④
⑤
⑥

- ① El modelo toma un lote de imágenes ruidosas y los pasos de tiempo como entrada.
- ② Los pasos de tiempo incorporados se agregan a las imágenes como entradas en varias etapas.
- ③ Pasa la entrada a través de la ruta de contracción
- ④ Pasa la entrada a través de la ruta del cuello de botella
- ⑤ Pasa la entrada a través de la ruta expansiva, con conexiones de salto
- ⑥ La salida es el ruido previsto en las imágenes de entrada.

La función de la U-Net de eliminación de ruido es predecir el ruido en las imágenes de entrada en función de los intervalos de tiempo en los que se encuentran. Como se describe en la ecuación 15.3, una imagen ruidosa en cualquier intervalo de tiempo t , x_t , se puede representar como la suma ponderada de la imagen limpia, x_0 , y el ruido aleatorio estándar distribuido normalmente, ϵ . El peso asignado a la imagen limpia disminuye y el peso asignado al ruido aleatorio aumenta a medida que el intervalo de tiempo t progresiona de 0 a T . Por lo tanto, para deducir el ruido en imágenes ruidosas, la U-Net de eliminación de ruido necesita saber en qué intervalo de tiempo se encuentra una imagen ruidosa.

Los pasos de tiempo se incrustan mediante funciones seno y coseno de forma similar a la codificación posicional en Transformers (discutida en los capítulos 9 y 10), lo que da como resultado un vector de 128 valores. Estas incrustaciones se expanden para que coincidan con las dimensiones de las características de la imagen en las distintas capas del modelo. Por ejemplo, en el primer bloque descendente, las incrustaciones de tiempo se transmiten a una forma de (128, 64, 64) antes de añadirse a las características de la imagen, que también tienen dimensiones de (128, 64, 64).

A continuación, creamos un modelo U-Net de eliminación de ruido instantaneamente la clase `UNet()` en el módulo local:

```
from utils.unet_util import UNet

device="cuda" if torch.cuda.is_available() else "cpu"
resolution=64
model=UNet(3,hidden_dims=[128,256,512,1024],
           image_size=resolution).to(device)
num=sum(p.numel() for p in model.parameters())
print("number of parameters: %.2fM" % (num/1e6,))
print(model)
```

La salida es

```
number of parameters: 133.42M
```

El modelo tiene más de 133 millones de parámetros, como se puede ver en el resultado anterior. Dada la gran cantidad de parámetros, el proceso de entrenamiento de este capítulo será largo, requiriendo aproximadamente de 3 a 4 horas de entrenamiento con GPU. Sin embargo, para quienes no tengan acceso al entrenamiento con GPU, las ponderaciones entrenadas también están disponibles en mi sitio web. El enlace a estas ponderaciones se proporcionará en la siguiente sección.

15.4 Entrenamiento y uso del modelo de eliminación de ruido U-Net

Ahora que tenemos los datos de entrenamiento y el modelo U-Net de eliminación de ruido, estamos listos para entrenar el modelo utilizando los datos

de entrenamiento.

Durante cada época de entrenamiento, recorreremos todos los lotes de los datos de entrenamiento. Para cada imagen, seleccionaremos aleatoriamente un paso de tiempo y añadiremos ruido a las imágenes limpias de los datos de entrenamiento según este valor, lo que genera una imagen con ruido. Estas imágenes con ruido y sus correspondientes valores de paso de tiempo se incorporan al modelo de desruido U-Net para predecir el ruido en cada imagen. Comparamos el ruido predicho con la realidad fundamental (el ruido real añadido a la imagen) y ajustamos los parámetros del modelo para minimizar el error absoluto medio entre el ruido predicho y el real.

Tras el entrenamiento, usaremos el modelo entrenado para generar imágenes de flores. Realizaremos esta generación en 50 pasos de inferencia (es decir, estableceremos los valores de los pasos de tiempo en 980, 960, ..., 20 y 0). Comenzando con ruido aleatorio, lo introduciremos en el modelo entrenado para obtener una imagen con ruido. Esta imagen con ruido se devuelve al modelo entrenado para eliminar el ruido. Repetimos este proceso durante 50 pasos de inferencia, lo que da como resultado una imagen indistinguible de las flores del conjunto de entrenamiento.

15.4.1 Entrenamiento del modelo U-Net de eliminación de ruido

A continuación, definiremos primero el optimizador y el programador de tasa de aprendizaje para el proceso de entrenamiento.

Utilizaremos el optimizador AdamW, una variante del optimizador Adam que hemos utilizado a lo largo de este libro. El optimizador AdamW, propuesto inicialmente por Ilya Loshchilov y Frank Hutter, desacopla la descomposición de pesos (una forma de regularización) de los pasos de optimización.³ En lugar de aplicar la descomposición de pesos directamente a los gradientes, AdamW la aplica directamente a los parámetros (pesos) después del paso de optimización. Esta modificación ayuda a lograr un mejor rendimiento de generalización al evitar que la tasa de descomposición se adapte junto con las tasas de aprendizaje. Los lectores interesados pueden obtener más información sobre el optimizador AdamW en el artículo original de Loshchilov y Hutter.

También utilizaremos un programador de la tasa de aprendizaje de la biblioteca de difusores para ajustar la tasa de aprendizaje durante el proceso

de entrenamiento. Inicialmente, usar una tasa de aprendizaje más alta puede ayudar al modelo a escapar de los mínimos locales, mientras que reducirla gradualmente en etapas posteriores del entrenamiento puede ayudar al modelo a converger de forma más estable y precisa hacia un mínimo global. El programador de la tasa de aprendizaje se define como se muestra en la siguiente lista.

Listado 15.5 Elección del optimizador y tasa de aprendizaje en el entrenamiento

```
from diffusers.optimization import get_scheduler

num_epochs=100
optimizer=torch.optim.AdamW(model.parameters(),
lr=0.0001, betas=(0.95,0.999),weight_decay=0.00001,
eps=1e-8)
lr_scheduler=get_scheduler(
    "cosine",
    optimizer=optimizer,
    num_warmup_steps=300,
    num_training_steps=(len(train_dataloader) *
    num_epochs))
```

① Entrenará el modelo durante 100 épocas

② Utiliza el optimizador AdamW

③ Utiliza el programador de tasa de aprendizaje en la biblioteca de difusores para controlar la tasa de aprendizaje

La definición exacta de la función `get_scheduler()` está disponible en GitHub por Hugging Face: <https://mng.bz/ZVo5>. En los primeros 300 pasos de entrenamiento (pasos de calentamiento), la tasa de aprendizaje aumenta linealmente de 0 a 0,0001 (la tasa de aprendizaje que configuramos en el optimizador AdamW). Después de 300 pasos, la tasa de aprendizaje disminuye siguiendo los valores de la función seno entre 0,0001 y 0. Entrenamos el modelo durante 100 épocas en la siguiente lista.

Listado 15.6 Entrenamiento del modelo U-Net de eliminación de ruido

```

for epoch in range(num_epochs):
    model.train()
    tloss = 0
    print(f"start epoch {epoch}")
    for step, batch in enumerate(train_dataloader):
        clean_images = batch["input"].to(device)*2-1
        nums = clean_images.shape[0]
        noise = torch.randn(clean_images.shape).to(device)
        timesteps = torch.randint(0,
                                   noise_scheduler.num_train_timesteps,
                                   (nums, ),
                                   device=device).long()
        noisy_images = noise_scheduler.add_noise(
            clean_images, noise, timesteps)

```

①

```

noise_pred = model(noisy_images,
                    timesteps)["sample"]
loss=torch.nn.functional.l1_loss(noise_pred, noise)
loss.backward()
optimizer.step()
lr_scheduler.step()
optimizer.zero_grad()
tloss += loss.detach().item()
if step%100==0:
    print(f"step {step}, average loss
{tloss/(step+1)}")
torch.save(model.state_dict(),'files/diffusion.pth')

```

②

③

④

① Agrega ruido para limpiar imágenes en el conjunto de entrenamiento

② Utiliza el U-Net de eliminación de ruido para predecir el ruido en imágenes ruidosas

③ Compara el ruido previsto con el ruido real para calcular la pérdida

④ Ajusta los parámetros del modelo para minimizar el error absoluto medio

Durante cada época, recorremos todos los lotes de imágenes limpias de flores del conjunto de entrenamiento. Introducimos ruido en estas imágenes limpias y las alimentamos a la U-Net de denoising para predecir el ruido en estas imágenes. A continuación, comparamos el ruido predicho con el ruido

real y ajustamos los parámetros del modelo para minimizar el error absoluto medio (píxel a píxel) entre ambos.

El proceso de entrenamiento descrito aquí toma varias horas con entrenamiento por GPU. Tras el entrenamiento, los pesos del modelo entrenado se guardan en su ordenador. También puede descargar los pesos entrenados desde mi sitio web: <https://mng.bz/RNID>. Descomprima el archivo después de descargarlo.

15.4.2 Uso del modelo entrenado para generar imágenes de flores

Para generar imágenes de flores, usaremos 50 pasos de inferencia. Esto significa que veremos 50 pasos de tiempo igualmente espaciados entre $t = 0$ y $t = T$, con $T = 1000$ en nuestro caso. Por lo tanto, los 50 pasos de tiempo de inferencia son $t = 980, 960, 940, \dots, 20$ y 0 . Comenzaremos con ruido aleatorio puro, que corresponde a la imagen en $t = 1000$. Usamos el modelo de eliminación de ruido entrenado U-Net para eliminarlo y crear una imagen ruidosa en $t = 980$. Luego presentamos la imagen ruidosa en $t = 980$ al modelo entrenado para eliminarlo y obtener la imagen ruidosa en $t = 960$. Repetimos el proceso durante muchas iteraciones hasta obtener una imagen en $t = 0$, que es una imagen limpia. Este proceso se implementa a través del método `generate()` en la clase `DDIMScheduler()` dentro del módulo local `ch15util.py`.

Listado 15.7 Definición de un método `generate()` en la clase `DDIMScheduler()`

```
@torch.no_grad()
def generate(self, model, device, batch_size=1,
            generator=None, eta=1.0, use_clipped_model_output=True,
            num_inference_steps=50):
    imgs = []
    image = torch.randn((batch_size, model.in_channels,
                        model.sample_size, model.sample_size),
                        generator=generator).to(device) ①

    self.set_timesteps(num_inference_steps)
    for t in tqdm(self.timesteps): ②
        model_output = model(image, t)["sample"] ③
        image = self.step(model_output, t, image, eta,
```

```

        use_clipped_model_output=\n        use_clipped_model_output)\n    img = unnormalize_to_zero_to_one(image)\n    img = img.cpu().permute(0, 2, 3, 1).numpy()\n    imgs.append(img)\n\n    image = unnormalize_to_zero_to_one(image)\n    image = image.cpu().permute(0, 2, 3, 1).numpy()\n\nreturn {"sample": image}, imgs

```

- ① Utiliza ruido aleatorio como punto de partida (es decir, imagen en t = 1000)
- ② Utiliza 50 pasos de tiempo de inferencia (t = 980, 960, 940, ..., 20, 0)
- ③ Utiliza el modelo de eliminación de ruido U-Net entrenado para predecir el ruido
- ④ Crea una imagen basada en el ruido previsto
- ⑤ Guarda imágenes intermedias en una lista, imgs

En este método `generate()`, también creamos una lista, `imgs`, para almacenar todas las imágenes intermedias en los pasos de tiempo $t = 980, 960, \dots, 20$ y 0. Las usaremos para visualizar el proceso de eliminación de ruido más adelante. El método `generate()` devuelve un diccionario con las imágenes generadas y la lista, `imgs`.

A continuación, utilizaremos el método `generate()` anterior para crear 10 imágenes limpias.

Listado 15.8 Generación de imágenes con el modelo de eliminación de ruido U-Net entrenado

```

sd=torch.load('files/diffusion.pth',map_location=device)\nmodel.load_state_dict(sd)\nwith torch.no_grad():\n    generator = torch.manual_seed(1)\n    generated_images,imgs = noise_scheduler.generate(\n        model,device,\n        num_inference_steps=50,\n        generator=generator,\n        eta=1.0,

```

```

        use_clipped_model_output=True,
        batch_size=10)

imgnp=generated_images["sample"]
import matplotlib.pyplot as plt
plt.figure(figsize=(10,4),dpi=300)
for i in range(10):
    ax = plt.subplot(2,5, i + 1)
    plt.imshow(imgnp[i])
    plt.xticks([])
    plt.yticks([])
    plt.tight_layout()
plt.show()

```

(2)

(3)

① Establece la semilla aleatoria en 1 para que los resultados sean reproducibles

② Utiliza el método generate() definido para crear 10 imágenes limpias

③ Traza las imágenes generadas

Establecemos la semilla aleatoria en 1. Como resultado, si utiliza el modelo entrenado de mi sitio web, obtendrá resultados idénticos a los que se muestran en la figura 15.6. Utilizamos el método `generate()` definido anteriormente para crear 10 imágenes limpias, mediante 50 pasos de inferencia.

Luego, graficamos las 10 imágenes en una cuadrícula de 2×5 , como se muestra en la figura 15.6.



Figura 15.6 Imágenes de flores creadas por el modelo de eliminación de ruido U-Net entrenado.

Como puede ver en la figura 15.6, las imágenes de flores generadas parecen reales y se parecen a las del conjunto de datos de entrenamiento.

Ejercicio 15.1

Modifique el código 15.8 y cambie la semilla aleatoria a 2. Mantenga el resto del código igual. Vuelva a ejecutar el código y observe cómo se ven las imágenes generadas.

El método `generate()` también devuelve una lista, `imgs`, que contiene todas las imágenes de los 50 pasos intermedios. Las usaremos para visualizar el proceso de eliminación de ruido.

Listado 15.9 Visualización del proceso de eliminación de ruido

```
steps=imgs[9::10]                                ①
imgs20=[]
for j in [1,3,6,9]:
    for i in range(5):
        imgs20.append(steps[i][j])                ②
plt.figure(figsize=(10,8),dpi=300)
for i in range(20):                                ③
    k=i%5
    ax = plt.subplot(4,5, i + 1)
    plt.imshow(imgs20[i])
    plt.xticks([])
    plt.yticks([])
    plt.tight_layout()
    plt.title(f't={800-200*k}', fontsize=15, c="r")
plt.show()
```

① Mantiene los pasos de tiempo 800, 600, 400, 200 y 0

② Selecciona 4 conjuntos de flores de 10

③ Traza las 20 imágenes en una cuadrícula de 4×5

La lista, `imgs`, contiene 10 conjuntos de imágenes en los 50 pasos de inferencia, $t = 980, 960, \dots, 20, 0$. Por lo tanto, hay un total de 500 imágenes en la lista. Seleccionamos cinco pasos de tiempo ($t = 800, 600, 400, 200$ y 0) para cuatro flores diferentes (las imágenes 2.^a, 4.^a, 7.^a y 10.^a de la figura 15.6). A continuación, graficamos las 20 imágenes en una cuadrícula de 4×5 , como se muestra en la figura 15.7.



Figura 15.7 Cómo el modelo U-Net entrenado con eliminación de ruido convierte gradualmente el ruido aleatorio en imágenes limpias de flores. Alimentamos el ruido aleatorio al modelo entrenado para obtener la imagen en el paso de tiempo 980. Luego, alimentamos el modelo con la imagen ruidosa en $t = 980$ para obtener la imagen en $t = 960$. Repetimos este proceso 50 pasos de inferencia hasta obtener la imagen en $t = 0$. La primera columna de esta figura muestra las cuatro flores en $t = 800$; la segunda columna muestra las mismas cuatro flores en $t = 600$; la última columna muestra las cuatro flores en $t = 0$ (es decir, imágenes limpias de flores).

La primera columna de la figura 15.7 muestra las cuatro imágenes de flores en $t = 800$. Se acercan al ruido aleatorio. La segunda columna muestra las flores en $t = 600$, y empiezan a parecerse a flores. A medida que nos desplazamos hacia la derecha, las imágenes se vuelven cada vez más nítidas. La columna más a la derecha muestra las cuatro imágenes limpias de flores en $t = 0$.

Ahora que comprende el funcionamiento de los modelos de difusión, analizaremos la generación de texto a imagen. El proceso de generación de imágenes de los transformadores de texto a imagen, como DALL-E 2, Imagen y Stable Diffusion, es muy similar al proceso de difusión inversa que analizamos anteriormente en este capítulo, salvo que el modelo toma la incrustación de texto como señal de acondicionamiento al generar una imagen.

15.5 Transformadores de texto a imagen

Los transformadores de texto a imagen, como DALL-E 2 de OpenAI, Imagen de Google y Stable Diffusion de Stability AI, utilizan modelos de difusión para generar imágenes a partir de descripciones textuales. Un componente importante de estos transformadores es el modelo de difusión. El proceso de generación de texto a imagen implica codificar el texto de entrada en una representación latente, que posteriormente se utiliza como señal de acondicionamiento para el modelo de difusión. Estos transformadores aprenden a generar imágenes realistas que corresponden a la descripción textual mediante la eliminación iterativa de ruido de un vector de ruido aleatorio, guiados por el texto codificado.

La clave de todos estos transformadores de texto a imagen reside en un modelo que comprenda el contenido en diferentes modalidades. En este caso, el modelo debe comprender las descripciones textuales y vincularlas con las imágenes, y viceversa.

En esta sección, usaremos el modelo CLIP de OpenAI como ejemplo. CLIP es un componente clave de DALL-E 2. Analizaremos cómo se entrenó CLIP para comprender la conexión entre las descripciones de texto y las imágenes. A continuación, usaremos un breve programa en Python para generar una imagen a partir de una instrucción de texto mediante DALL-E 2 de OpenAI.

15.5.1 CLIP: Un transformador multimodal

En los últimos años, la intersección de la visión artificial y el procesamiento del lenguaje natural (PLN) ha experimentado avances significativos, uno de los cuales es la creación del modelo CLIP por OpenAI. Este innovador modelo está diseñado para comprender e interpretar imágenes en el contexto del lenguaje natural, una capacidad con un inmenso potencial para diversas aplicaciones, como la generación y clasificación de imágenes.

El modelo CLIP es un transformador multimodal que conecta los datos visuales y textuales. Está entrenado para comprender imágenes asociándolas con sus correspondientes descripciones textuales. A diferencia de los modelos tradicionales, que requieren el etiquetado explícito de las imágenes, CLIP utiliza un amplio conjunto de datos de imágenes y sus descripciones en lenguaje natural para obtener una representación más generalizable de los conceptos visuales.

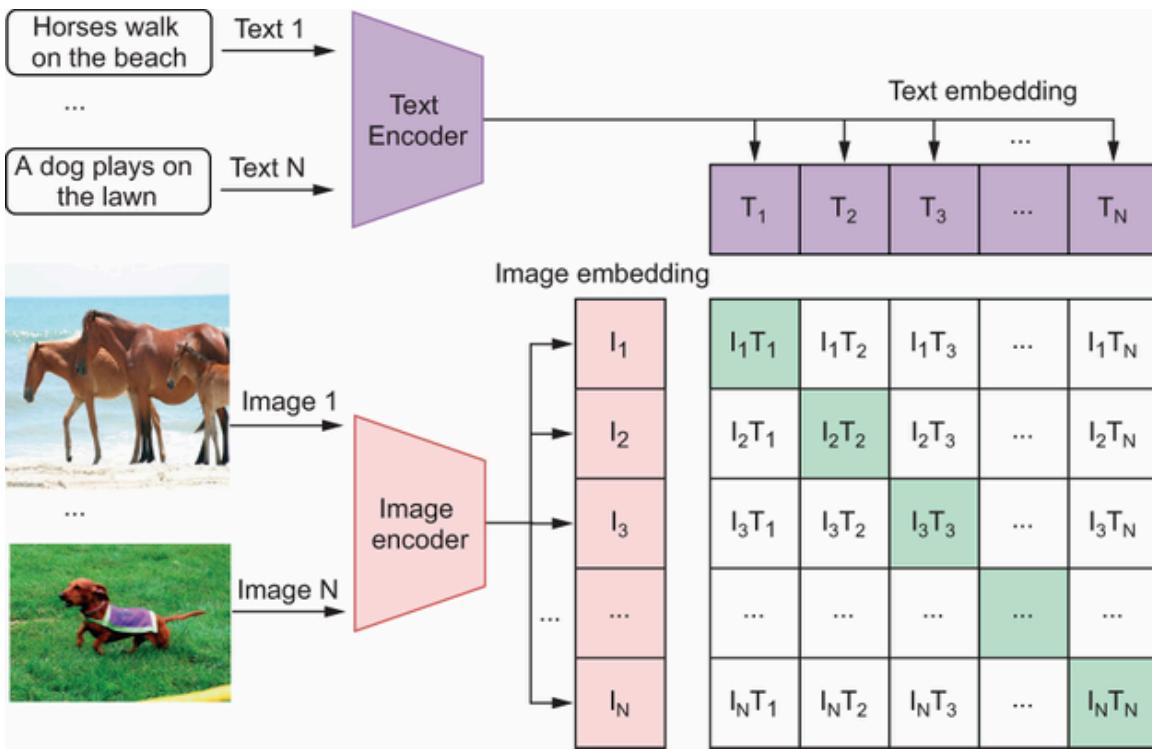


Figura 15.8. Cómo se entrena el modelo CLIP de OpenAI. Se recopila un conjunto de datos de entrenamiento a gran escala de pares de texto-imagen. El codificador de texto del modelo comprime la descripción textual en una incrustación de texto con valor D. El codificador de imágenes convierte la imagen correspondiente en una incrustación de imagen también con valores D. Durante el entrenamiento, un lote de N pares de texto-imagen se convierte en N incrustaciones de texto y N incrustaciones de imagen. CLIP utiliza un enfoque de aprendizaje contrastivo para maximizar la similitud entre las incrustaciones emparejadas (la suma de los valores diagonales en la figura) y minimizar la similitud entre las incrustaciones de pares de texto-imagen no coincidentes (la suma de los valores fuera de la diagonal en la figura).

El entrenamiento del modelo CLIP, ilustrado en la figura 15.8, comienza con la recopilación de un conjunto de datos a gran escala que incluye imágenes y sus descripciones textuales asociadas. OpenAI utiliza diversas fuentes, incluyendo conjuntos de datos públicos y datos web, para garantizar una amplia variedad de contenido visual y textual. Posteriormente, el conjunto de datos se preprocesa para estandarizar las imágenes de forma uniforme y tokenizar el texto, preparándolo para su introducción en el modelo.

CLIP emplea una arquitectura de doble codificador, compuesta por un codificador de imágenes y un codificador de texto. El codificador de imágenes procesa las imágenes de entrada, mientras que el codificador de texto procesa las descripciones textuales correspondientes. Estos codificadores proyectan las imágenes y el texto en un espacio de incrustación compartido donde pueden compararse y alinearse.

La base del entrenamiento de CLIP reside en su enfoque de aprendizaje contrastivo. Para cada lote de N pares de imagen-texto del conjunto de datos, el modelo busca maximizar la similitud entre las incrustaciones emparejadas (medida por la suma de los valores diagonales en la figura 15.8) y minimizar

la similitud entre las incrustaciones de pares de texto-imagen no coincidentes (la suma de los valores fuera de la diagonal). La figura 15.9 muestra un diagrama de cómo los transformadores de texto a imagen, como DALL-E 2, generan imágenes realistas basadas en indicaciones de texto.

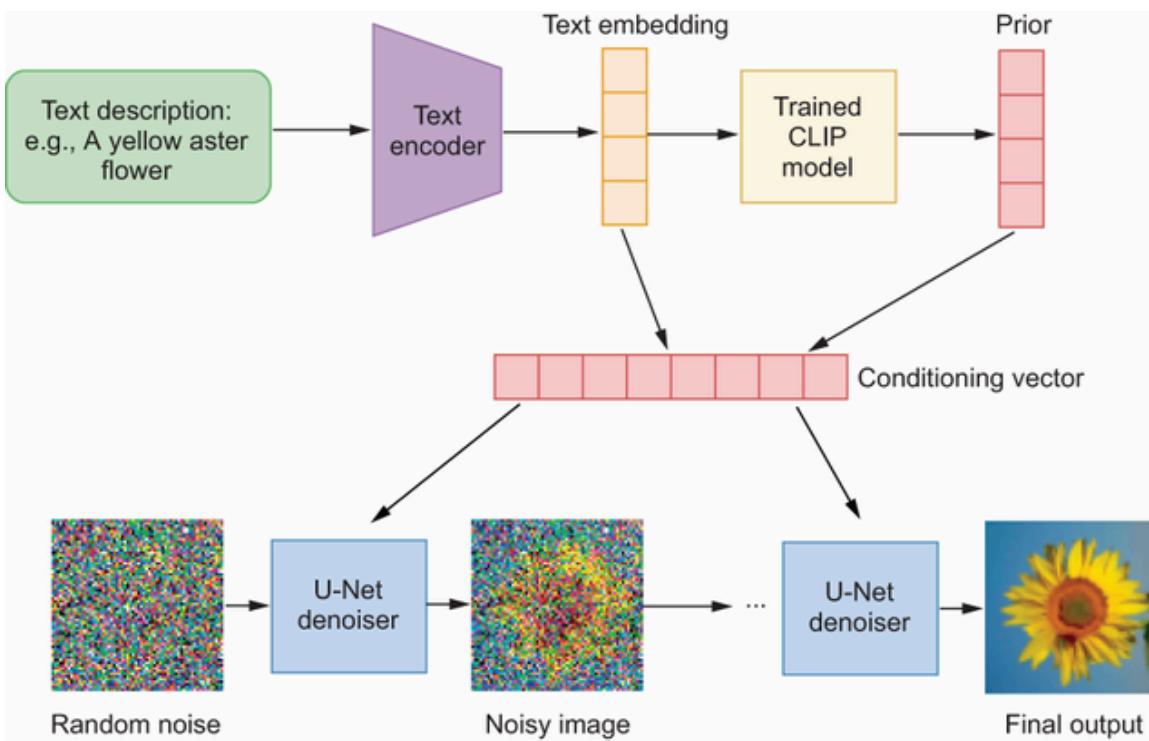


Figura 15.9 Cómo los transformadores de texto a imagen, como DALL-E 2, crean imágenes basadas en indicaciones de texto. El codificador de texto del transformador de texto a imagen entrenado convierte primero la descripción del texto de la indicación en una incrustación de texto. Esta incrustación se introduce en el modelo CLIP para obtener un vector a priori que representa la imagen en el espacio latente. La incrustación de texto y la a priori se concatenan en un vector de acondicionamiento. Para generar una imagen, el denoiser U-Net toma primero un vector de ruido aleatorio como entrada para generar una imagen con ruido utilizando el vector de acondicionamiento. A continuación, toma la imagen con ruido y el vector de acondicionamiento como entrada y genera otra imagen con menos ruido. El proceso se repite durante varias iteraciones hasta obtener el resultado final: una imagen limpia.

El proceso de generación de imágenes de los transformadores de texto a imagen es similar al proceso de difusión inversa que analizamos anteriormente en este capítulo. Tomemos como ejemplo DALL-E 2, propuesto por investigadores de OpenAI en 2022.⁴ El codificador de texto del modelo convierte primero la descripción de texto del mensaje en una incrustación de texto. Esta incrustación se introduce en el modelo CLIP para obtener un vector a priori que representa la imagen en el espacio latente. La incrustación de texto y la a priori se concatenan en un vector de condicionamiento. En la primera iteración, introducimos un vector de ruido aleatorio en el denoiser U-Net del modelo y le pedimos que genere una imagen ruidosa a partir del vector de condicionamiento. En la segunda iteración, introducimos la imagen ruidosa de la iteración anterior en el denoiser U-Net y le pedimos que genere otra imagen ruidosa a partir del vector de condicionamiento. Repeti-

mos este proceso durante muchas iteraciones y el resultado final es una imagen limpia.

15.5.2 Generación de texto a imagen con DALL-E 2

Ahora que comprende cómo funcionan los transformadores de texto a imagen, escribamos un programa en Python para interactuar con DALL-E 2 para crear una imagen basada en una solicitud de texto.

Primero, debe solicitar una clave API de OpenAI. OpenAI ofrece varios niveles de precios que varían según la cantidad de tokens procesados y el tipo de modelos utilizados. Vaya a <https://chat.openai.com/auth/login> y haga clic en el botón "Registrarse" para crear una cuenta. Despues, inicie sesión en su cuenta y vaya a <https://platform.openai.com/api-keys> para ver su clave API. Guárdela en un lugar seguro para usarla más adelante. Podemos generar una imagen usando DALL-E 2 de OpenAI.

Listado 15.10 Generación de imágenes con DALL-E 2

```
from openai import OpenAI

openai_api_key=your actual OpenAI API key here, in quotes      ①
client=OpenAI(api_key=openai_api_key)                            ②

response = client.images.generate(
    model="dall-e-2",
    prompt="an astronaut in a space suit riding a unicorn",
    size="512x512",
    quality="standard",
    n=1,
)
image_url = response.data[0].url                                ③
print(image_url)                                                 ④
```

① Asegúrate de proporcionar tu clave API de OpenAI real aquí, entre comillas

② Instancia la clase OpenAI() para crear un agente

③ Utiliza el método images.generate() para generar una imagen basada en el mensaje de texto

④ Imprime la URL de la imagen

Debes colocar la clave de API de OpenAI que obtuviste anteriormente en el listado 15.10. Creamos un agente instanciando la clase `OpenAI()`. Para generar una imagen, necesitamos especificar el modelo, un mensaje de texto y el tamaño de la imagen. Usamos "un astronauta con traje espacial montando un unicornio" como mensaje, y el listado de código proporciona una URL para visualizar y descargar la imagen. La URL caduca en una hora y la imagen resultante se muestra en la figura 15.10.

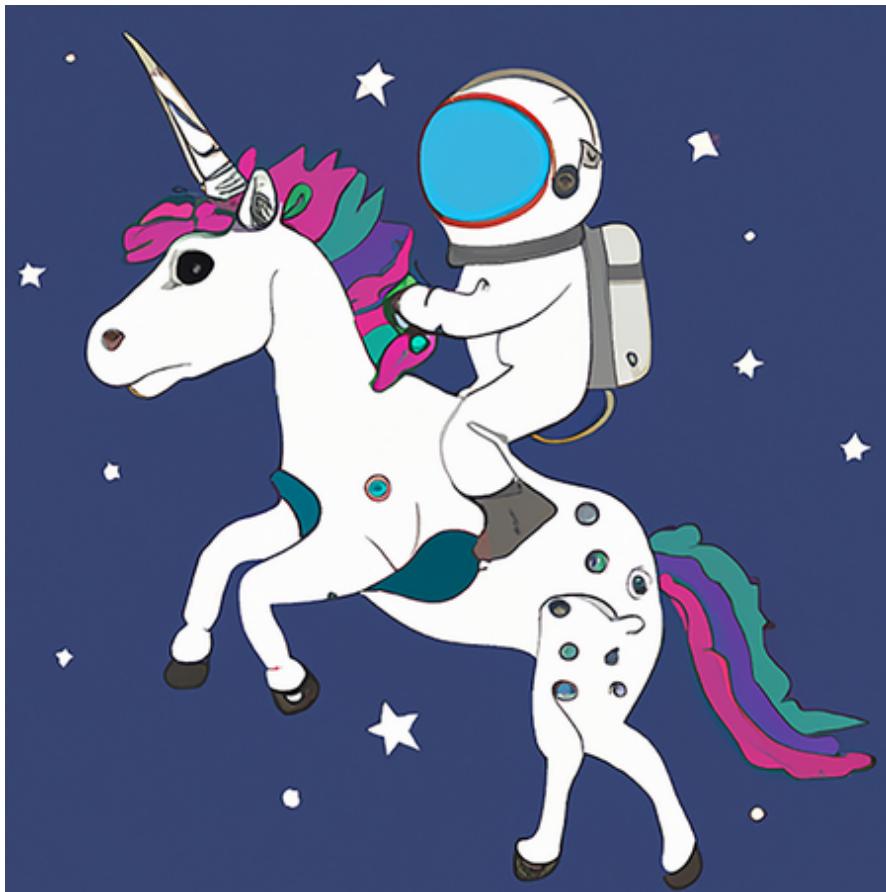


Figura 15.10 Una imagen generada por DALL-E 2 con el texto indicativo “un astronauta en un traje espacial montando un unicornio”

Ejecute el listado 15.10 y vea qué imagen genera DALLE-2. Tenga en cuenta que el resultado será diferente, ya que la salida de DALLE-2 (y de todos los LLM) es estocástica, no determinista.

Ejercicio 15.2

Solicita una clave API de OpenAI. Luego, modifica el código 15.10 para generar una imagen con el texto "un gato con traje trabajando en una computadora".

En este capítulo, aprendiste el funcionamiento interno de los modelos basados en difusión y su importancia en los transformadores de texto a imagen, como el modelo CLIP de OpenAI. También descubriste cómo obtener tu clave API de OpenAI y usaste un breve script de Python para generar imágenes a partir de descripciones de texto con DALL-E 2, que incorpora CLIP.

En el siguiente capítulo, continuarás usando la clave API de OpenAI obtenida anteriormente para usar LLMs preentrenados y generar contenido diverso, como texto, audio e imágenes. Además, integrarás la biblioteca LangChain de Python con otras API, lo que te permitirá crear un asistente personal completo.

Resumen

- En la difusión hacia adelante, añadimos gradualmente pequeñas cantidades de ruido aleatorio a las imágenes limpias hasta que se transforman en ruido puro. Por el contrario, en la difusión inversa, comenzamos con ruido aleatorio y empleamos un modelo de denoising para eliminar progresivamente el ruido de las imágenes, transformándolo de nuevo en una imagen limpia.
- La arquitectura U-Net, diseñada originalmente para la segmentación de imágenes biomédicas, presenta una forma simétrica con una ruta de codificación contraída y una ruta de decodificación expansiva, conectadas por una capa de cuello de botella. Durante la eliminación de ruido, las U-Nets están adaptadas para eliminar el ruido a la vez que se conservan los detalles. Las conexiones de salto vinculan los mapas de características del codificador y del decodificador con las mismas dimensiones espaciales, lo que ayuda a preservar información espacial, como bordes y texturas, que podría perderse durante el submuestreo en el proceso de codificación.

- La incorporación de un mecanismo de atención en un modelo U-Net de eliminación de ruido le permite concentrarse en las características importantes e ignorar las irrelevantes. Al tratar los píxeles de la imagen como una secuencia, el mecanismo de atención aprende las dependencias de los píxeles, de forma similar a cómo aprende las dependencias de los tokens en el procesamiento del lenguaje natural (PLN). Esto mejora la capacidad del modelo para identificar características relevantes eficazmente.
- Los transformadores de texto a imagen, como DALL-E 2 de OpenAI, Imagen de Google y Stable Diffusion de Stability AI, utilizan modelos de difusión para crear imágenes a partir de descripciones textuales. Codifican el texto en una representación latente que condiciona el modelo de difusión, el cual, a su vez, elimina el ruido de forma iterativa de un vector de ruido aleatorio, guiado por el texto codificado, para generar imágenes realistas que coinciden con la descripción textual.

¹ Jascha Sohl-Dickstein, Eric A. Weiss, Niru Maheswaranathan y Surya Ganguli, 2015, “Aprendizaje profundo no supervisado utilizando termodinámica de no equilibrio”. Conferencia internacional sobre aprendizaje automático, <http://arxiv.org/abs/1503.03585>.

² Sohl-Dickstein et al., 2015, “Aprendizaje profundo no supervisado mediante termodinámica de no equilibrio”, <https://arxiv.org/abs/1503.03585>. Yang Song y Stefano Ermon, 2019, “Modelado generativo mediante la estimación de gradientes de la distribución de datos”.
<https://arxiv.org/abs/1907.05600>. Jonathan Ho, Ajay Jain y Pieter Abbeel, 2020, “Modelos probabilísticos de difusión con eliminación de ruido”, <https://arxiv.org/abs/2006.11239>.

³ Ilya Loshchilov y Frank Hutter, 2017, “Regularización de decaimiento de peso desacoplado”. <https://arxiv.org/abs/1711.05101>.

⁴ Aditya Rames, Prafulla Dhariwal, Alex Nichol, Casey Chu y Mark Chen, 2022, “Generación jerárquica de imágenes condicionales de texto con latentes CLIP”. <https://arxiv.org/abs/2204.06125>.

16. Modelos de lenguaje grandes preentrenados y la biblioteca LangChain

Este capítulo cubre

- Uso de modelos de lenguaje grandes previamente entrenados para la generación de texto, imágenes, voz y código
- Técnicas de incitación de pocos disparos, de un disparo y de cero disparos
- Creando un asistente personal de cero disparos con LangChain
- Limitaciones y preocupaciones éticas de la IA generativa

El auge de los modelos de lenguaje grande (LLM) preentrenados ha transformado el campo del procesamiento del lenguaje natural (PLN) y las tareas generativas. La serie GPT de OpenAI, un ejemplo notable, muestra las amplias capacidades de estos modelos para producir texto, imágenes, voz e incluso código realistas. El uso eficaz de estos LLM preentrenados es esencial por varias razones. Nos permite implementar funcionalidades avanzadas de IA sin necesidad de grandes recursos para desarrollarlos y entrenarlos. Además, comprender estos LLM sienta las bases para aplicaciones innovadoras que aprovechan el PLN y la IA generativa, impulsando el progreso en diversas industrias.

En un mundo cada vez más influenciado por la IA, dominar la integración y personalización de LLM preentrenados ofrece una ventaja competitiva crucial. A medida que la IA evoluciona, aprovechar estos modelos sofisticados se vuelve vital para la innovación y el éxito en el panorama digital.

Normalmente, estos modelos se operan mediante interfaces basadas en navegador, que varían según el LLM y funcionan de forma independiente. Cada modelo tiene fortalezas y especialidades únicas. La interfaz a través de un navegador limita nuestra capacidad para aprovechar al máximo el potencial de cada LLM específico. El uso de lenguajes de programación

como Python, en particular a través de herramientas como la biblioteca LangChain, ofrece importantes ventajas por las siguientes razones.

La función de Python en la interacción con los LLM mejora la automatización de flujos de trabajo y procesos. Los scripts de Python, capaces de ejecutarse de forma autónoma, facilitan operaciones ininterrumpidas sin necesidad de entrada manual. Esto resulta especialmente beneficioso para empresas que gestionan regularmente grandes cantidades de datos. Por ejemplo, un script de Python podría generar informes mensuales de forma autónoma consultando un LLM, sintetizando la información obtenida y difundiendo estos hallazgos por correo electrónico o en una base de datos. Python ofrece un mayor nivel de personalización y control en la gestión de las interacciones con los LLM que las interfaces basadas en navegador, lo que permite crear código personalizado para satisfacer necesidades operativas específicas, como la implementación de lógica condicional, el procesamiento de múltiples solicitudes en bucles o la gestión de excepciones. Esta adaptabilidad es esencial para personalizar los resultados y cumplir objetivos empresariales o investigaciones específicos.

La extensa colección de bibliotecas de Python lo hace ideal para integrar LLM con software y sistemas existentes. Un excelente ejemplo de ello es la biblioteca LangChain, que amplía la funcionalidad de Python con LLM.

LangChain permite la combinación de múltiples LLM o la integración de las capacidades de LLM con otros servicios, como la API de Wikipedia o la API de Wolfram Alpha, que se abordarán más adelante en este capítulo.

Esta capacidad de encadenar diferentes servicios permite la construcción de sofisticados sistemas de IA de múltiples pasos donde las tareas son segmentadas y gestionadas por los modelos o servicios más adecuados, mejorando tanto el rendimiento como la precisión.

Para ello, en este capítulo, aprenderá primero a usar la API de OpenAI para crear diversos contenidos mediante programación en Python: texto, imágenes, voz y código Python. También aprenderá la diferencia entre la generación de contenido de pocas instancias, de una sola instancia y de cero instancias. La generación de pocas instancias implica proporcionar al modelo múltiples ejemplos para ayudarle a comprender la tarea, mientras que la generación de una sola instancia o de cero instancias implica proporcionar un solo ejemplo o ninguno.

Los LLM modernos, como ChatGPT, se entrenan con conocimientos previos de hace unos meses, por lo que no pueden proporcionar información reciente o en tiempo real, como las condiciones meteorológicas, el estado de los vuelos o la cotización de las acciones. Aprenderás a combinar los LLM con las API de Wolfram Alpha y Wikipedia mediante la biblioteca Lang-Chain para crear un asistente personal inteligente y sin experiencia.

A pesar de las impresionantes capacidades de los LLM, estos carecen de una comprensión intrínseca del contenido. Esto puede dar lugar a errores de lógica, inexactitudes factuales y la incapacidad de comprender conceptos o matices complejos. El rápido avance y la aplicación generalizada de estos modelos también generan diversas preocupaciones éticas, como sesgos, desinformación, privacidad y derechos de autor. Estas cuestiones exigen una cuidadosa consideración y medidas proactivas para garantizar que el desarrollo y la implementación de los LLM se ajusten a los estándares éticos y los valores sociales.

16.1 Generación de contenido con la API de OpenAI

Si bien existen otros programas de maestría en derecho (LLM), como LLAMA de Meta y Gemini de Google, la serie GPT de OpenAI es la más destacada. Por lo tanto, en este capítulo, utilizamos los GPT de OpenAI como ejemplos.

OpenAI permite usar LLM para generar diversos contenidos, como texto, imágenes, audio y código. Se puede acceder a su servicio mediante un navegador web o una API. En este capítulo, nos centraremos en la generación de contenido con programas Python mediante una API debido a las ventajas de interactuar con LLM con Python, mencionadas anteriormente.

Necesita su clave API de OpenAI para que los programas de este capítulo funcionen. Supongo que ya la obtuvo en el capítulo 15. De lo contrario, vuelve al capítulo 15 para obtener instrucciones detalladas sobre cómo obtenerla.

En esta sección me centraré principalmente en la generación de texto, pero proporcionaré un ejemplo para cada uno de los casos de generación de código, imagen y voz.

Este capítulo trata sobre el uso de varias bibliotecas nuevas de Python. Para instalarlas, ejecute las siguientes líneas de código en una nueva celda de la aplicación Jupyter Notebook en su computadora:

```
!pip install --upgrade openai langchain_openai langchain
!pip install wolframalpha langchainhub
!pip install --upgrade --quiet wikipedia
```

Siga las instrucciones en pantalla para finalizar la instalación.

16.1.1 Tareas de generación de texto con la API de OpenAI

Puede generar texto para muchos propósitos diferentes, como responder preguntas, resumir textos y escribir de forma creativa.

Al realizar una pregunta a OpenAI GPT, tenga en cuenta que todos los LLM, incluidos los OpenAI GPT, se entranan con datos históricos recopilados mediante rastreo web automatizado. Al momento de escribir este artículo, GPT-4 se entrenó con datos hasta diciembre de 2023, con un retraso de tres meses. GPT-3.5 se entrenó con datos hasta septiembre de 2021.

Primero, hagamos una pregunta a GPT sobre hechos históricos. Ingrese las líneas de código de la siguiente lista en una nueva celda.

Listado 16.1 Comprobación de hechos históricos con la API de OpenAI

```
from openai import OpenAI

openai_api_key=put your actual OpenAI API key here, in quotes
client=OpenAI(api_key=openai_api_key)

completion = client.chat.completions.create(
    model="gpt-3.5-turbo",
    messages=[
        {"role": "system", "content": "
            You are a helpful assistant, knowledgeable about
            recent facts.'''},
        {"role": "user", "content": "
            Who won the Nobel Prize in Economics in 2000?'''}
    ]
)
```

(1)
(2)
(3)
(4)

```
)  
print(completion.choices[0].message.content)
```

- ① Proporciona su clave API de OpenAI
- ② Crea una instancia de clase OpenAI() y la nombra cliente
- ③ Define el rol del sistema
- ④ Hace la pregunta

Asegúrese de proporcionar su clave de API de OpenAI en el listado 16.1. Primero, instanciamos la clase `OpenAI()` y la llamamos `client`. En el método `chat.completions.create()`, especificamos el modelo como `gpt-3.5-turbo`. El sitio <https://platform.openai.com/docs/models> ofrece varios modelos. Puede usar gpt-4 o gpt-3.5-turbo para la generación de texto. El primero ofrece mejores resultados, pero también implica un mayor gasto. Usaremos el segundo en la mayoría de los casos, ya que nuestros ejemplos son bastante simples y ofrecen resultados igualmente buenos.

El parámetro `messages` del bloque de código anterior consta de varios objetos de mensaje, cada uno de los cuales contiene un rol (que puede ser "sistema", "usuario" o "asistente") y contenido. Un mensaje del sistema determina el comportamiento del asistente; en ausencia de un mensaje del sistema, la configuración predeterminada lo caracteriza como "un asistente útil". Los mensajes de usuario incluyen consultas o comentarios que el asistente debe responder. Por ejemplo, en el ejemplo anterior, el mensaje de usuario es "¿Quién ganó el Premio Nobel de Economía en 2000?". La salida es

```
The Nobel Prize in Economics in 2000 was awarded  
to James J. Heckman and Daniel L. McFadden for their  
work on microeconomics and microeconomic theory.
```

OpenAI ha proporcionado la respuesta correcta.

También puedes pedirle al Máster que escriba un ensayo sobre un tema específico. A continuación, te pediremos que escribas un ensayo breve sobre la importancia de la automotivación:

```
completion = client.chat.completions.create(  
    model="gpt-3.5-turbo",  
    n=1,  
    messages=[  
        {"role": "system", "content":  
            '''You are a helpful assistant, capable of  
            writing essays.'''},  
        {"role": "user", "content":  
            '''Write a short essay on the importance of  
            self-motivation.'''}  
    ]  
)  
print(completion.choices[0].message.content)
```

El argumento `n=1` aquí indica al asistente que genere una respuesta. Si desea varias respuestas, puede establecer `n` en un número diferente. El valor predeterminado para `n` es 1. El resultado es

Self-motivation is a key factor in achieving success and personal growth in various aspects of life. It serves as the driving force behind our actions, decisions, and goals, pushing us to overcome obstacles and challenges along the way. One of the primary benefits of self-motivation is that it helps individuals take initiative and control of their lives...

El resultado consta de seis párrafos, y solo he incluido las primeras frases.

Puedes visitar el repositorio de GitHub del libro

(<https://github.com/markhliu/DGAI>) para ver el ensayo completo. Como puedes ver, la redacción es coherente, concisa y sin errores gramaticales.

Incluso puedes pedirle al GPT de OpenAI que escriba un chiste para ti:

```
completion = client.chat.completions.create(  
    model="gpt-3.5-turbo",  
    messages=[  
        {"role": "system", "content":  
            '''You are a helpful assistant, capable of telling jokes.'''},  
        {"role": "user", "content":  
            '''Tell me a math joke.'''}  
    ]
```

```
)  
print(completion.choices[0].message.content)
```

Le pedimos que contara un chiste de matemáticas y el resultado es

```
Why was the equal sign so humble? Because he knew  
he wasn't less than or greater than anyone else!
```

Puedes mantener conversaciones con el asistente. El parámetro de mensajes incluye automáticamente el historial de conversaciones. Por ejemplo, después de ejecutar el bloque de código anterior, si ejecutas lo siguiente:

```
completion = client.chat.completions.create(  
    model="gpt-3.5-turbo",  
    messages=[  
        {"role": "user", "content":  
            '''Haha, that's funny! Tell me another one.'''
        }  
    ]  
)  
print(completion.choices[0].message.content)
```

Recibirás una respuesta similar a

```
Sure! Why was the math book sad? Because it had too  
many problems.
```

La consulta del usuario "¡Jaja, qué gracioso! Cuéntame otro" solo tiene sentido en el contexto de los mensajes anteriores, donde le pides al asistente que cuente un chiste matemático. Otras funciones de generación de texto incluyen el resumen y la clasificación de textos, ejemplos que verás más adelante en este capítulo.

16.1.2 Generación de código con la API de OpenAI

Codex está diseñado específicamente para comprender y generar código. Es compatible con múltiples lenguajes de programación y puede traducir descripciones en lenguaje natural a código. Codex ahora forma parte integral de las GPT de OpenAI.

A continuación, le pedimos a OpenAI GPT que produzca un programa en Python para trazar una curva sinusoidal:

```
completion = client.chat.completions.create(  
    model="gpt-3.5-turbo",  
    messages=[  
        {"role": "system", "content":  
            '''You are a helpful assistant,  
            capable of generating Python programs.'''},  
        {"role": "user", "content":  
            '''Write a Python program to plot a sine graph.'''}  
    ]  
)  
print(completion.choices[0].message.content)
```

Tenga en cuenta que ChatGPT utiliza GPT-3.5-Turbo para gestionar eficazmente tanto el aspecto conversacional como las tareas de generación de código. No proporciona un modelo de Codex independiente solo para la generación de código. El resultado es

Sure! Here's a simple Python program using the matplotlib library to plot a sine graph:

```
```python  
import matplotlib.pyplot as plt
import numpy as np

Generate x values from 0 to 2*pi
x = np.linspace(0, 2*np.pi, 100)

Calculate y values using the sine function
y = np.sin(x)

Plot the sine graph
plt.figure()
plt.plot(x, y)
plt.title('Sine Graph')
plt.xlabel('x')
plt.ylabel('sin(x)')
plt.grid(True)
plt.show()
```
```

You can run this program in a Python environment with the `matplotlib` library installed to visualize the sine graph.

Si copia y pega el programa Python generado en una celda de Jupyter Notebook y lo ejecuta, verá una imagen similar a la figura 16.1.

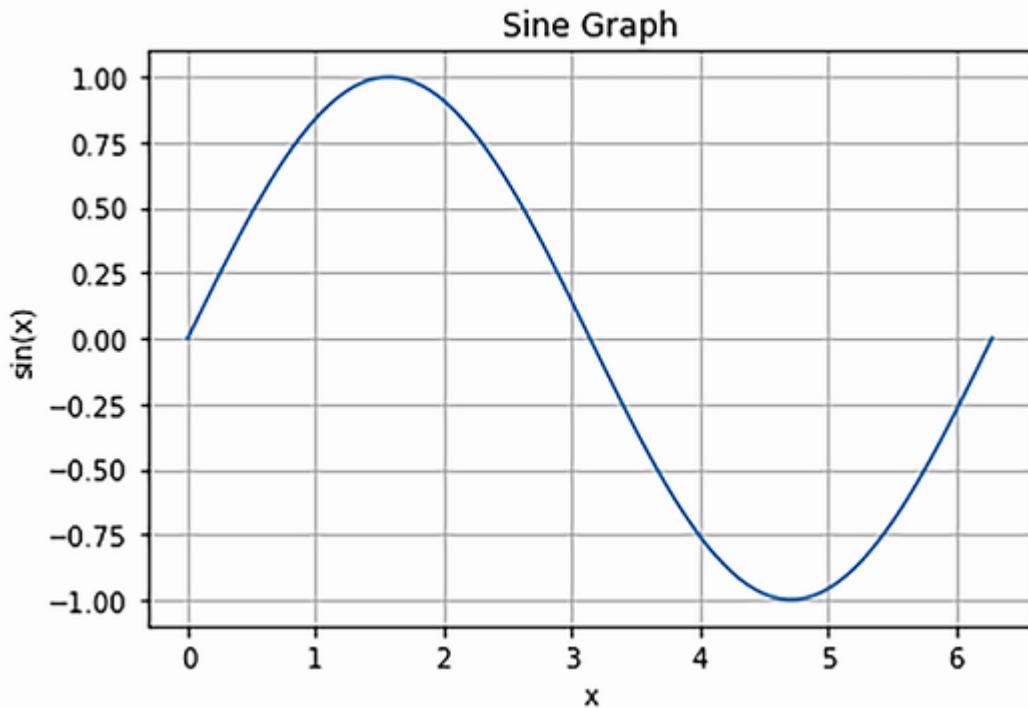


Figura 16.1: Uso de OpenAI GPT-3.5 para generar código Python y trazar una curva sinusoidal. Usamos la descripción textual "Escribir un programa Python para trazar una gráfica sinusoidal" para solicitarle que genere un programa Python. A continuación, ejecutamos el programa para crear la gráfica.

El LLM no solo proporciona el código Python, sino que también le permite saber que necesita ejecutar el código en un entorno Python con la biblioteca `matplotlib` instalada.

16.1.3 Generación de imágenes con OpenAI DALL-E 2

DALL-E 2 es un modelo de IA desarrollado por OpenAI, diseñado para generar imágenes a partir de descripciones textuales. Es el sucesor del modelo DALL-E original y representa un avance en el campo de la IA generativa para contenido visual.

DALL-E 2 utiliza un modelo de difusión similar al que analizamos en el capítulo 15, que comienza con un patrón aleatorio de píxeles y lo perfecciona

gradualmente hasta obtener una imagen coherente que coincide con el texto de entrada. Ha mejorado el DALL-E original al producir imágenes de mayor calidad con representaciones más precisas y detalladas de las descripciones textuales.

La incorporación de DALL-E 2 a la serie GPT de OpenAI nos permite no solo generar texto, sino también crear imágenes basadas en indicaciones textuales. A continuación, le pedimos a DALL-E 2 que cree una imagen de alguien pescando en la orilla del río:

```
response = client.images.generate(  
    model="dall-e-2",  
    prompt="someone fishing at the river bank",  
    size="512x512",  
    quality="standard",  
    n=1,  
)  
image_url = response.data[0].url  
print(image_url)
```

El bloque de código genera una URL. Si hace clic en ella, verá una imagen similar a la de la figura 16.2.



Figura 16.2 Una imagen generada por DALL-E 2 con el mensaje de texto “alguien pescando en la orilla del río”

La URL caduca en una hora, así que asegúrese de acceder a ella lo antes posible. Además, la imagen generada por DALL-E 2 es ligeramente diferente incluso si usa el mismo mensaje de texto, ya que la salida se genera aleatoriamente.

16.1.4 Generación de voz con la API de OpenAI

La conversión de texto a voz (TTS) es una tecnología que convierte texto escrito en palabras habladas. TTS se entrena mediante transformadores multimodales, donde la entrada es texto y la salida es audio. En el contexto de ChatGPT, la integración de las capacidades de TTS permite que el LLM no solo genere respuestas textuales, sino que también las lea en voz alta. A continuación, solicitamos a la API de OpenAI que convierta un texto corto en voz:

```
response = client.audio.speech.create(  
    model="tts-1-hd",  
    voice="shimmer",  
    input='''This is an audio file generated by  
    OpenAI's text to speech AI model.'''  
)  
response.stream_to_file("files/speech.mp3")
```

Tras ejecutar la celda de código anterior, se guarda en su ordenador un archivo, speech.mp3, que puede escuchar. La documentación (<https://platform.openai.com/docs/guides/text-to-speech>) ofrece opciones de voz. Aquí hemos seleccionado esta opción `shimmer`. Otras opciones incluyen `alloy`, `echo`, etc.

16.2 Introducción a LangChain

LangChain es una biblioteca de Python diseñada para facilitar el uso de LLM en diversas aplicaciones. Proporciona un conjunto de herramientas y abstracciones que facilitan la creación, implementación y gestión de aplicaciones basadas en LLM como GPT-3, GPT-4 y otros modelos similares.

LangChain simplifica la interacción con diferentes LLM y aplicaciones, permitiendo a los desarrolladores centrarse en la lógica de sus aplicaciones sin preocuparse por las especificaciones del modelo subyacente. Es espe-

cialmente adecuado para crear un agente "sabelotodo" al encadenar un LLM con aplicaciones como Wolfram Alpha y Wikipedia, que pueden proporcionar información en tiempo real o datos recientes. La arquitectura modular de LangChain facilita la integración de diferentes componentes, lo que permite al agente aprovechar las ventajas de varios LLM y aplicaciones.

16.2.1 La necesidad de la biblioteca LangChain

Imagine que su objetivo es construir un agente inteligente y sin experiencia que pueda producir contenido diverso, recuperar información en tiempo real y responder preguntas objetivas. Quiere que el agente acceda automáticamente a la fuente correcta para recuperar la información relevante según la tarea en cuestión, sin necesidad de indicarle explícitamente qué hacer. La biblioteca LangChain es la herramienta ideal para ello.

En este proyecto, aprenderás a usar la biblioteca LangChain para combinar LLM con las API de Wolfram Alpha y Wikipedia y crear un agente inteligente y de cero disparos. Usamos la API de Wolfram Alpha para recuperar información en tiempo real y la API de Wikipedia para responder preguntas sobre datos recientes. LangChain nos permite crear un agente que utiliza múltiples herramientas para responder una pregunta. El agente primero comprende la consulta y luego decide qué herramienta usar para responderla.

Para demostrarle que incluso los LLM más avanzados carecen de estas habilidades, preguntemos quién ganó el premio al Mejor Actor en los Premios de la Academia de 2024:

```
completion = client.chat.completions.create(  
    model="gpt-4",  
    messages=[  
        {"role": "system", "content":  
            '''You are a helpful assistant, knowledgeable about  
            recent facts.'''},  
        {"role": "user", "content":  
            '''Who won the Best Actor Award in 2024 Academy  
            Awards?'''}  
    ]
```

```
)  
print(completion.choices[0].message.content)
```

La salida es

I'm sorry, but I cannot provide real-time information or make predictions about future events such as the 2024 Academy Awards. For the most accurate and up-to-date information, I recommend checking reliable sources or news outlets closer to the date of the awards show.

Realicé esta consulta el 17 de marzo de 2024 y GPT-4 no pudo responderla. Es posible que al realizar la misma consulta, obtenga la respuesta correcta porque el modelo se ha actualizado con datos más recientes. En ese caso, cambie la pregunta a un evento ocurrido hace unos días y debería obtener una respuesta similar.

Por lo tanto, usaremos LangChain para encadenar un LLM con las API de Wolfram Alpha y Wikipedia. Wolfram Alpha es eficaz en cálculos científicos y en la recuperación de información en tiempo real, mientras que Wikipedia es famosa por proporcionar información sobre eventos y hechos históricos y recientes.

16.2.2 Uso de la API de OpenAI en LangChain

La biblioteca langchain-openai que instaló anteriormente en este capítulo le permite usar las GPT de OpenAI con una ingeniería de indicaciones mínima. Solo necesita explicar lo que desea que haga el LLM en un lenguaje sencillo.

A continuación se muestra un ejemplo de cómo le pedimos que corrija errores gramaticales en el texto:

```
from langchain_openai import OpenAI  
  
llm = OpenAI(openai_api_key=openai_api_key)  
  
prompt = """  
Correct the grammar errors in the text:
```

```
i had went to stor buy phone. No good. returned  
get new phone.
```

```
"""
```

```
res=llm.invoke(prompt)  
print(res)
```

La salida es

```
I went to the store to buy a phone, but it was  
no good. I returned it and got a new phone.
```

Tenga en cuenta que no utilizamos ingeniería de indicaciones. Tampoco especificamos qué modelo usar. LangChain encontró el mejor modelo para el trabajo basándose en los requisitos de la tarea y otros factores como el coste, la latencia y el rendimiento. Además, formatea y estructura automáticamente las consultas para que se ajusten al modelo utilizado. La indicación anterior simplemente solicita al agente, en un lenguaje sencillo, que corrija los errores gramaticales del texto. Devuelve el texto con la gramática correcta, como se muestra en el resultado anterior.

Aquí hay otro ejemplo. Le pedimos al agente que nombrara la capital de Kentucky:

```
prompt = """  
What is the capital city of the state of Kentucky?  
"""  
res=llm.invoke(prompt)  
print(res)
```

La salida es

```
The capital city of Kentucky is Frankfort.
```

Nos dice la respuesta correcta, que es Frankfort, Kentucky.

16.2.3 Incitación de disparo cero, de un disparo y de pocos disparos

Las indicaciones de pocas oportunidades, de una oportunidad y de cero oportunidades se refieren a diferentes maneras de proporcionar ejemplos o instrucciones a los LLM para guiar sus respuestas. Estas técnicas se utilizan para ayudar al modelo a comprender la tarea en cuestión y generar resultados más precisos o relevantes.

En la inducción de cero intentos, se le asigna al modelo una tarea o pregunta sin ejemplos. La instrucción suele incluir una descripción clara de lo que se espera, pero el modelo debe generar una respuesta basándose únicamente en su conocimiento y comprensión preexistentes. En la inducción de un solo intento, se le proporciona al modelo un solo ejemplo para ilustrar la tarea. En la inducción de pocos intentos, se le proporcionan múltiples ejemplos para ayudarle a comprender la tarea. La inducción de pocos intentos se basa en la idea de que proporcionar más ejemplos puede ayudar al modelo a comprender mejor el patrón o las reglas de la tarea, lo que resulta en respuestas más precisas.

Todas sus interacciones hasta el momento con los GPT de OpenAI no han dado resultados esperados, ya que no les ha proporcionado ningún ejemplo.

Probemos un ejemplo de incitación de pocos intentos. Supongamos que desea que el LLM realice un análisis de sentimiento: quiere que clasifique una oración como positiva o negativa. Puede proporcionar varios ejemplos en la incitación:

```
prompt = """
The movie is awesome! // Positive
It is so bad! // Negative
Wow, the movie was incredible! // Positive
How horrible the movie is! //
"""

res=llm.invoke(prompt)
print(res)
```

La salida es

Negative

En la consigna, proporcionamos tres ejemplos. Dos reseñas se clasificaron como positivas y una como negativa. Luego, incluimos la frase: "¡Qué horrible es la película!". El LLM la clasificó correctamente como negativa.

En el ejemplo anterior, sepáramos la oración // del sentimiento correspondiente. Puedes usar otros separadores, como ->, siempre que sea coherente.

A continuación se muestra un ejemplo de una solicitud de una sola toma:

```
prompt = """
Car -> Driver
Plane ->
"""

res=llm.invoke(prompt)
print(res)
```

La salida es

Pilot

Al proporcionar un solo ejemplo, en realidad le estamos preguntando al LLM: "¿Qué es para un avión lo que un conductor es para un automóvil?"
El LLM respondió correctamente Pilot .

Ejercicio 16.1

Supongamos que desea preguntarle al LLM: "¿Qué es para un jardín lo que un chef es para una cocina?" Utilice preguntas de una sola vez para obtener la respuesta.

Finalmente, aquí hay un ejemplo de aviso de disparo cero:

```
prompt = """
Is the tone in the sentence "Today is a great day
for me" positive, negative, or neutral?
```

```
"""
res=llm.invoke(prompt)
print(res)
```

La salida es

Positive

No proporcionamos ejemplos en la instrucción. Sin embargo, proporcionamos instrucciones en un lenguaje sencillo para pedirle al LLM que clasifique el tono de la oración como positivo, negativo o neutro.

16.3 Un agente sabelotodo de tiro cero en LangChain

En esta sección, aprenderá a crear un agente inteligente y de cero disparos en LangChain. Utilizará las GPT de OpenAI para generar diversos contenidos, como texto, imágenes y código. Para compensar la incapacidad de LLM para proporcionar información en tiempo real, aprenderá a añadir las API de Wolfram Alpha y Wikipedia a sus herramientas.

Wolfram Alpha es un motor de conocimiento computacional diseñado para gestionar consultas factuales en línea, especializado en tareas numéricas y computacionales, especialmente en los campos de la ciencia y la tecnología. Al integrar la API de Wolfram Alpha, el agente puede responder prácticamente cualquier pregunta sobre diversos temas. Si Wolfram Alpha no puede proporcionar una respuesta, utilizaremos Wikipedia como fuente secundaria para preguntas basadas en hechos sobre temas específicos.

La figura 16.3 es un diagrama de los pasos que seguiremos para crear el agente sabelotodo de disparo cero en esta sección.

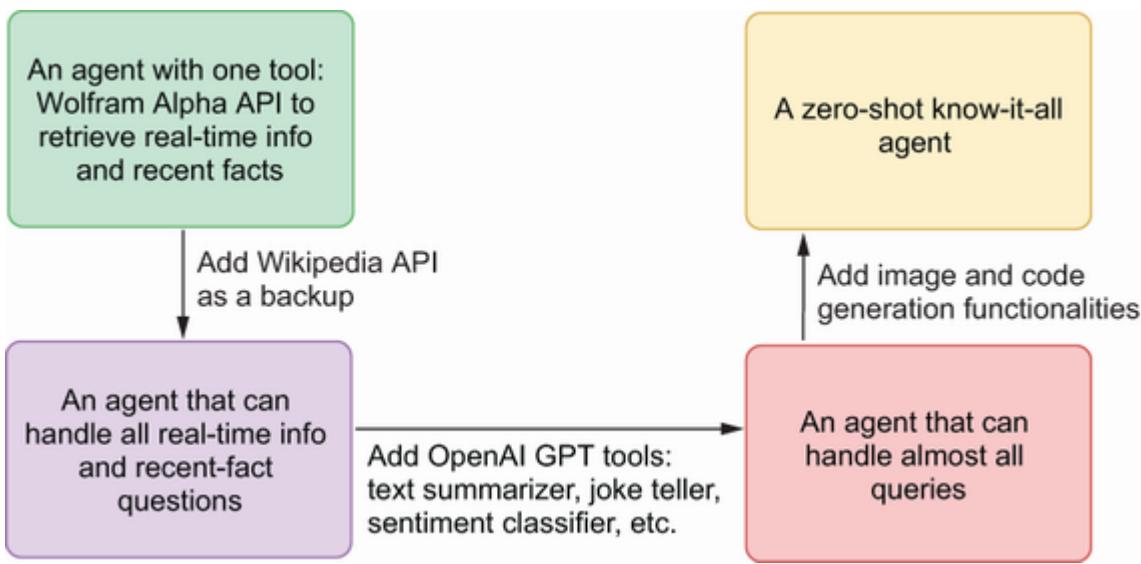


Figura 16.3 Pasos para crear un agente sabelotodo de disparo cero con la biblioteca LangChain

En concreto, encontraremos primero, crearemos un agente en LangChain con una sola herramienta (la API Wolfram Alpha) para responder preguntas relacionadas con información en tiempo real y datos recientes. Después, añadiremos la API de Wikipedia como respaldo para preguntas relacionadas con datos recientes. Incorporaremos varias herramientas que utilizan la API de OpenAI, como un resumidor de texto, un contador de chistes y un clasificador de sentimientos. Finalmente, añadiremos funciones de generación de imágenes y código.

16.3.1 Solicitud de una clave API de Wolfram Alpha

Wolfram Alpha te ofrece hasta 2000 llamadas API no comerciales al mes de forma gratuita. Para obtener una clave API, primero visita <https://account.wolfram.com/login/create> y completa los pasos para crear una cuenta.

La cuenta Wolfram solo te da acceso a través del navegador; necesitas solicitar una clave API en <https://products.wolframalpha.com/api/>. Una vez allí, haz clic en "Obtener acceso a la API" en la esquina inferior izquierda. Debería aparecer un pequeño cuadro de diálogo; completa los campos "Nombre" y "Descripción", selecciona "API simple" en el menú desplegable y haz clic en "Enviar", como se muestra en la figura 16.4.

Get a New App ID

To send queries to Wolfram|Alpha from your application, you need to request a unique App ID for your application. Please fill in information below about your application

Name *

zero-shot know-it-all agent

Description *

Create a zero-shot know-it-all agent

API *

Simple API



Submit

Figura 16.4 Solicitud de un AppID de Wolfram Alpha

Después, tu AppID debería aparecer en una nueva ventana. Copia la clave API y guárdala en un archivo para usarla más adelante.

A continuación se explica cómo puede utilizar la API de Wolfram Alpha para realizar operaciones matemáticas:

```
import os

os.environ['WOLFRAM_ALPHA_APPID'] =
"your Wolfram Alpha AppID"

from langchain_community.utilities.wolfram_alpha
```

```
import WolframAlphaAPIWrapper
wolfram = WolframAlphaAPIWrapper()
res=wolfram.run("how much is 23*55+123?")
print(res)
```

La salida es

```
Assumption: 23×55 + 123
Answer: 1388
```

La API Wolfram Alpha proporciona la respuesta correcta.

También incluiremos la API de Wikipedia para responder a diversos temas. No es necesario solicitar una clave API si tiene instalada la biblioteca de Wikipedia en su ordenador. A continuación, se muestra un ejemplo del uso de la API de Wikipedia en la biblioteca LangChain:

```
from langchain.tools import WikipediaQueryRun
from langchain_community.utilities
import WikipediaAPIWrapper

wikipedia = WikipediaQueryRun(api_wrapper=
WikipediaAPIWrapper())
res=wikipedia.run("University of Kentucky")
print(res)
```

La salida es

```
Page: University of Kentucky
Summary: The University of Kentucky (UK, UKY, or U of K) is a public land-grant research university in Lexington, Kentucky. Founded in 1865 by John Bryan Bowman as the Agricultural and Mechanical College of Kentucky, the university is one of the state's two land-grant universities (the other being Kentucky State University)...
```

Hemos omitido la mayor parte del resultado para mayor brevedad.

16.3.2 Creación de un agente en LangChain

A continuación, crearemos un agente en LangChain, con solo la API de Wolfram Alpha en la caja de herramientas. En este contexto, un agente se refiere a una entidad individual diseñada para gestionar tareas o procesos específicos mediante interacciones de lenguaje natural. Posteriormente, iremos añadiendo gradualmente más herramientas a la cadena para que el agente pueda gestionar más tareas.

Listado 16.2 Creación de un agente en LangChain

```
os.environ['OPENAI_API_KEY'] = openai_api_key
from langchain.agents import load_tools
from langchain_openai import ChatOpenAI
from langchain import hub
from langchain.agents import AgentExecutor,
create_react_agent
from langchain_openai import OpenAI

prompt = hub.pull("hwchase17/react")
llm = ChatOpenAI(model_name='gpt-3.5-turbo')          ①
tool_names = ["wolfram-alpha"]
tools = load_tools(tool_names, llm=llm)               ②
agent = create_react_agent(llm, tools, prompt)
agent_executor = AgentExecutor(agent=agent, tools=tools,
                               handle_parsing_errors=True, verbose=True)    ③

res=agent_executor.invoke({"input": """
What is the temperature in Lexington, Kentucky now?
"""})
print(res["output"])                                ④
```

① Define qué LLM utilizar

② Agrega Wolfram Alpha a la caja de herramientas

③ Define un agente

④ Le hace una pregunta al agente

En LangChain `hwchase17/react`, se refiere a un tipo específico de configuración del agente ReAct. ReAct significa Acción Reactiva, un marco dentro de LangChain diseñado para optimizar el uso de las capacidades del modelo de lenguaje en combinación con otras herramientas para resolver tareas complejas de forma eficaz. Consulte

https://python.langchain.com/docs/how_to/migrate_agent/ para obtener más información. Al crear un agente en LangChain, debe especificar las herramientas que utilizará. En el ejemplo anterior, solo usamos una herramienta: la API Wolfram Alpha.

Como ejemplo, preguntamos la temperatura actual en Lexington, Kentucky, y aquí está el resultado:

```
> Entering new AgentExecutor chain...
I should use Wolfram Alpha to find the current
temperature in Lexington, Kentucky.
Action: wolfram_alpha
Action Input: temperature in Lexington,
KentuckyAssumption: temperature | Lexington,
Kentucky
Answer: 44 °F (wind chill: 41 °F)
(27 minutes ago)I now know the current temperature
in Lexington, Kentucky.
Final Answer: The temperature in Lexington,
Kentucky is 44 °F with a wind chill of 41 °F.

> Finished chain.
The temperature in Lexington, Kentucky is 44 °F
with a wind chill of 41 °F.
```

El resultado no solo muestra la respuesta final, que indica que la temperatura actual en Lexington, Kentucky, es de 44 grados Fahrenheit, sino que también muestra la cadena de pensamiento. Utiliza Wolfram Alpha como fuente para obtener la respuesta.

También podemos agregar Wikipedia a la caja de herramientas:

```
tool_names += ["wikipedia"]
tools = load_tools(tool_names, llm=llm)
agent = create_react_agent(llm, tools, prompt)
agent_executor = AgentExecutor(agent=agent,
```

```
tools=tools, handle_parsing_errors=True,verbose=True)

res=agent_executor.invoke({"input": """
Who won the Best Actor Award in 2024 Academy Awards?
"""})
print(res["output"])
```

Pregunto quién ganó el premio al Mejor Actor en los Premios Oscar de 2024, y el agente usa Wikipedia para obtener la respuesta correcta:

```
I need to find information about the winner of
the Best Actor Award at the 2024 Academy Awards.
Action: wikipedia
Action Input: 2024 Academy Awards Best Actor
...
Cillian Murphy won the Best Actor Award at the 2024
Academy Awards for his performance in Oppenheimer.
```

En el resultado anterior, el agente decide primero usar Wikipedia como herramienta para resolver el problema. Tras buscar en varias fuentes de Wikipedia, proporciona la respuesta correcta.

A continuación, aprenderá a agregar varias herramientas OpenAI GPT a la caja de herramientas del agente.

16.3.3 Agregar herramientas mediante GPT de OpenAI

Primero agregamos un resumidor de texto para que el agente pueda resumir el texto.

Listado 16.3 Agregar un resumidor de texto a la caja de herramientas del agente

```
from langchain.agents import Tool
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain

temp = PromptTemplate(input_variables=["text"],
template="Write a one sentence summary of the
following text: {text}")
```

(1)

```

summarizer = LLMChain(llm=llm, prompt=temp) ②

sum_tool = Tool.from_function(
    func=summarizer.run,
    name="Text Summarizer",
    description="A tool for summarizing texts") ③
tools+=[sum_tool]
agent = create_react_agent(llm, tools, prompt)
agent_executor = AgentExecutor(agent=agent, tools=tools,
                               handle_parsing_errors=True, verbose=True)
res=agent_executor.invoke({"input":'
    '''Write a one sentence summary of the following text:
The University of Kentucky's Master of Science
in Finance (MSF) degree prepares students for
a professional career in the finance and banking
industries. The program is designed to provide
rigorous and focused training in finance,
broaden opportunities in your career, and
sharpened skills for the fast-changing
and competitive world of modern finance.'''
})
print(res["output"])

```

① Define una plantilla

② Define una función de resumen

③ Agrega el resumidor como herramienta

④ Redefine el agente con la caja de herramientas actualizada

Primero proporcionamos una plantilla para resumir el texto. Luego, definimos una función de resumen y la añadimos a la caja de herramientas. Finalmente, redefinimos el agente usando la caja de herramientas actualizada y le pedimos que resuma el texto de ejemplo en una sola oración. Asegúrese de que su solicitud tenga el mismo formato que el descrito en la plantilla para que el agente sepa qué herramienta usar.

El resultado del listado 16.3 es

```

> Entering new AgentExecutor chain...
I need to summarize the text provided.
Action: Summarizer

```

```
...
> Finished chain.

The University of Kentucky's MSF program
offers specialized training in finance to
prepare students for successful careers in
the finance and banking industries.
```

El agente elige el resumidor como herramienta para la tarea, ya que la entrada coincide con la plantilla descrita en la función del resumidor. Usamos dos oraciones largas como entrada de texto y la salida anterior es un resumen de una sola oración.

Puedes añadir tantas herramientas como quieras. Por ejemplo, puedes añadir una herramienta para contar un chiste sobre un tema específico:

```
temp = PromptTemplate(input_variables=["text"],
                      template="Tell a joke on the following subject: {subject}")

joke_teller = LLMChain(llm=llm, prompt=temp)

tools+=[Tool.from_function(name='Joke Teller',
                           func=joke_teller.run,
                           description='A tool for telling jokes')]
agent = create_react_agent(llm, tools, prompt)
agent_executor = AgentExecutor(agent=agent,
                               tools=tools, handle_parsing_errors=True, verbose=True)

res=agent_executor.invoke({"input":
    '''Tell a joke on the following subject: coding'''})
print(res["output"])
```

La salida es

```
> Entering new AgentExecutor chain...
I should use the Joke Teller tool to find a
coding-related joke.
Action: Joke Teller
Action Input: coding
Observation: Why was the JavaScript developer sad?

Because he didn't know how to "null" his feelings.
Thought:That joke was funny!
```

Final Answer: Why was the JavaScript developer sad?
Because he didn't know how to "null" his feelings.

> Finished chain.

Why was the JavaScript developer sad? Because he
didn't know how to "null" his feelings.

Le pedimos al agente que cuente un chiste sobre programación . El agente identifica a Joke Teller como la herramienta. El chiste está efectivamente relacionado con la programación.

Ejercicio 16.2

Añade una herramienta al conjunto de herramientas del agente para realizar análisis de sentimientos. Llámala "Clasificador de Sentimientos". Luego, pídele al agente que clasifique el texto "esta película es regular" como positivo, negativo o neutral.

16.3.4 Agregar herramientas para generar código e imágenes

Puedes agregar varias herramientas a la caja de herramientas de LangChain. Los lectores interesados pueden encontrar más detalles en https://python.langchain.com/docs/how_to/#tools. A continuación, añadimos herramientas para generar otros formatos de contenido, como código e imágenes.

Para agregar una herramienta para generar código, puede hacer lo siguiente:

```
temp = PromptTemplate(input_variables=["text"],  
template='''Write a Python program based on the  
description in the following text: {text}''')  
  
code_generator = LLMChain(llm=llm, prompt=temp)  
  
tools+=[Tool.from_function(name='Code Generator',  
func=code_generator.run,  
description='A tool to generate code')]  
agent = create_react_agent(llm, tools, prompt)  
agent_executor = AgentExecutor(agent=agent,  
tools=tools, handle_parsing_errors=True, verbose=True)
```

```
res=agent_executor.invoke({"input":  
    '''Write a Python program based on the  
    description in the following text:  
    write a python program to plot a sine curve and a cosine  
    curve in the same graph. The sine curve is in solid line  
    and the cosine curve is in dashed line. Add a legend to  
    the graph. Set the x-axis range to -5 to 5. The title  
    should be "Comparing sine and cosine curves."  
    '''}  
    print(res["output"])
```

La salida es

```
> Entering new AgentExecutor chain...  
I should use the Code Generator tool to generate the  
Python program based on the given description.  
Action: Code Generator  
Action Input: Write a Python program to plot a sine curve  
and a cosine curve in the same graph. The sine curve is in  
solid line and the cosine curve is in dashed line. Add a  
legend to the graph. Set the x-axis range to -5 to 5.  
The title should be "Comparing sine and cosine curves."  
Observation: import matplotlib.pyplot as plt  
import numpy as np  
  
x = np.linspace(-5, 5, 100)  
y1 = np.sin(x)  
y2 = np.cos(x)
```

```
plt.plot(x, y1, label='Sine Curve', linestyle='solid')  
plt.plot(x, y2, label='Cosine Curve', linestyle='dashed')  
plt.legend()  
plt.title('Comparing Sine and Cosine Curves')  
plt.xlim(-5, 5)  
plt.show()
```

Thought: The Python program has been successfully generated to plot the sine and cosine curves. I now know the final answer.

Final Answer: The Python program to plot a sine curve and a cosine curve in the same graph with the specified requirements has been generated.

```
> Finished chain.
```

The Python program to plot a sine curve and a cosine curve in the same graph with the specified requirements has been generated.

Si ejecuta el código generado en una celda, verá una imagen como en la figura 16.5.

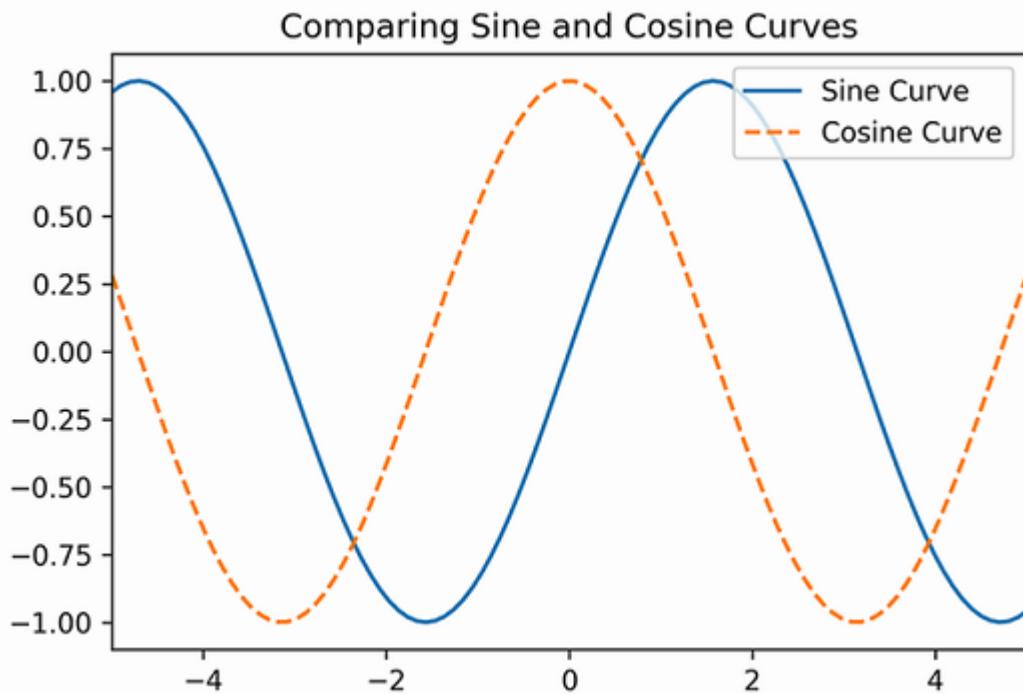


Figura 16.5. Adición de una herramienta en LangChain para generar código Python. La herramienta genera código para trazar curvas de seno y coseno en el mismo gráfico, con leyenda y estilos de línea.

Para agregar un generador de imágenes, puede hacer lo siguiente:

```
from langchain_community.utilities.dalle_image_generator
import DalleAPIWrapper
temp = PromptTemplate(input_variables=["text"],
template="Create an image base on the following
text: {text}")

grapher = LLMChain(llm=llm, prompt=temp)

tools+=[Tool.from_function(name='Text to image',
    func=grapher.run,
    description='A tool for text to image')]
agent = create_react_agent(llm, tools, prompt)
agent_executor = AgentExecutor(agent=agent,
    tools=tools, handle_parsing_errors=True, verbose=True)
```

```
image_url = DallEAPIWrapper().run(agent_executor.  
invoke({"input": '''Create an image base on the following  
text: a horse grazes on the grassland.'''}))["output"]  
print(image_url)
```

El resultado es una URL para visualizar y descargar una imagen. Le pedimos al agente que creara una imagen de un caballo pastando en la pradera. La imagen se muestra en la figura 16.6.



Figura 16.6 Una imagen generada por un agente sabelotodo en LangChain

Con esto, has aprendido a crear un agente inteligente y de cero disparos en LangChain. Puedes agregar más herramientas según lo que quieras que el agente logre.

16.4 Limitaciones y preocupaciones éticas de los LLM

Los programas de maestría en derecho (LLM), como la serie GPT de OpenAI, han logrado avances significativos en el campo del PLN y la IA generativa. A pesar de sus impresionantes capacidades, estos modelos no están exentos de limitaciones. Comprender estas limitaciones es crucial tanto para aprovechar sus fortalezas como para mitigar sus debilidades.

Al mismo tiempo, el rápido avance y la aplicación generalizada de estos modelos también han generado numerosas preocupaciones éticas, como sesgos, inexactitudes, vulneraciones de la privacidad y violaciones de derechos de autor. Estas cuestiones exigen una cuidadosa consideración y medidas proactivas para garantizar que el desarrollo y la implementación de los LLM se ajusten a los estándares éticos y los valores sociales.

En esta sección, exploraremos las limitaciones de los LLM, analizaremos la persistencia de estos problemas y presentaremos ejemplos de fracasos notables para subrayar la importancia de abordar estos desafíos. También examinaremos las principales preocupaciones éticas asociadas con los LLM y propondremos vías para mitigarlas.

16.4.1 Limitaciones de los LLM

Una de las limitaciones fundamentales de los LLM es su falta de comprensión y razonamiento auténticos. Si bien pueden generar respuestas coherentes y contextualmente relevantes, carecen de una comprensión intrínseca del contenido. Esto puede dar lugar a errores de lógica, inexactitudes factuales y la incapacidad de comprender conceptos o matices complejos.

Esto se manifiesta en muchos errores garrafales cometidos por los LLM. El libro "Inteligente hasta que sea tonto" ofrece muchos ejemplos entretenidos de errores similares cometidos por GPT-3 y ChatGPT.¹ Por ejemplo, considere esta pregunta: La Sra. March le dio té y gachas a la madre, mientras vestía al bebé con tanta ternura como si fuera suyo. ¿Quién es la madre del bebé? La respuesta de GPT-3 es la Sra. March.

Para ser justos, con el rápido avance de los LLM, muchos de estos errores se corrigen con el tiempo. Sin embargo, los LLM aún cometen errores menores. Un artículo de LinkedIn de junio de 2023, escrito por David Johnston (<https://www.linkedin.com/pulse/intelligence-tests-langs-fail-why-david-johnston/>), pone a prueba la inteligencia de los LLM en una docena de problemas que los humanos pueden resolver fácilmente. Los LLM, incluido el GPT-4, tienen dificultades con estos problemas. Uno de ellos es el siguiente: nombrar un animal cuya longitud de palabra sea igual al número de patas menos el número de colas.

Este error no se ha corregido al momento de escribir este artículo. La Figura 16.7 es una captura de pantalla de la respuesta de GPT-4 al usar la interfaz de un navegador.

ChatGPT 4 ▾

 **You**
Name an animal such that the length of the word is equal to the number of legs they have minus the number of tails they have.

 **ChatGPT**
An animal that fits this description is a "bee." A bee has six legs and one tail (stinger), so the number of legs minus the number of tails is five, which is equal to the length of the word "bee."

Figura 16.7 Cómo GPT-4 todavía comete errores de bajo nivel

La salida en la figura 16.7 muestra que, según GPT-4, cinco es igual al número de letras de la palabra “abeja”.

16.4.2 Preocupaciones éticas para los LLM

Una de las preocupaciones éticas más apremiantes es la posibilidad de que los modelos LLM perpetúen y amplifiquen los sesgos en sus datos de entrenamiento. Dado que estos modelos aprenden de grandes conjuntos de datos, a menudo derivados de contenido generado por humanos, pueden heredar sesgos relacionados con el género, la raza, la etnia y otros factores sociales. Esto puede generar resultados sesgados que refuerzan los estereotipos y la discriminación.

Para mitigar el sesgo, es fundamental adoptar conjuntos de datos de entrenamiento diversos e inclusivos, implementar algoritmos de detección y corrección de sesgos, y garantizar la transparencia en el desarrollo y la evaluación de modelos. Es especialmente importante establecer una colaboración a nivel de toda la industria para establecer estándares en las prácticas de mitigación de sesgos y promover el desarrollo responsable de la IA.

Sin embargo, debemos tener en cuenta que no debemos sobrecorregir. Un contraejemplo es que Gemini de Google sobrecorrige los estereotipos en la generación de imágenes al incluir a personas de color en grupos como los soldados alemanes de la era nazi.²

Otra preocupación para los LLM es su potencial para la desinformación y la manipulación. Estos programas tienen la capacidad de generar textos realistas y persuasivos, que pueden explotarse para crear y difundir desinformación, propaganda o contenido manipulador. Esto supone riesgos significativos para el discurso público, la democracia y la confianza en la información.

La solución a esta preocupación reside en desarrollar sistemas robustos de moderación de contenido. Establecer directrices para un uso responsable y fomentar la colaboración entre desarrolladores de IA, legisladores y medios de comunicación son pasos cruciales para combatir la desinformación.

La tercera preocupación se relaciona con la privacidad. La gran cantidad de datos utilizados para entrenar los LLM plantea problemas de privacidad, ya que información confidencial puede revelarse inadvertidamente en los resultados del modelo. Además, la posibilidad de que los LLM se utilicen en ciberataques o para eludir las medidas de seguridad plantea importantes riesgos de seguridad.

Además, los datos utilizados para entrenar LLMs se recopilan en su mayoría sin autorización. Los partidarios argumentan que la forma en que se utilizan los datos para entrenar LLMs es transformadora: el modelo no se limita a regurgitar los datos, sino que los utiliza para generar contenido nuevo y original. Esta transformación podría calificar bajo la doctrina del "uso justo", que permite el uso limitado de material con derechos de autor sin permiso si el uso añade una nueva expresión o significado. Los críticos argumentan que los LLMs se entrena en grandes cantidades de textos con derechos de autor sin permiso, lo que va más allá de lo que podría considerarse un uso justo. La escala de los datos utilizados y la ingesta directa de material con derechos de autor sin transformación durante el entrenamiento podrían considerarse una infracción. El debate está en curso. Las leyes actuales de derechos de autor no se diseñaron teniendo en cuenta la IA generativa, lo que genera ambigüedades sobre cómo se aplican a tecnologías como los LLMs. Es un debate que probablemente deba ser resuelto por los órganos legislativos y judiciales para proporcionar directrices claras y garantizar que los intereses de todas las partes estén representados de forma justa.

Las preocupaciones éticas en torno a los LLM son multifacéticas y requieren un enfoque holístico. La colaboración entre investigadores, desarrolladores y legisladores de IA es crucial para desarrollar directrices y marcos éticos que guíen el desarrollo y la implementación responsables de estos potentes modelos. A medida que continuamos aprovechando el potencial de los LLM, las consideraciones éticas deben permanecer prioritarias en nuestros esfuerzos para garantizar que la IA avance en armonía con los valores sociales y el bienestar humano.

Resumen

- La indicación de pocas tomas significa brindarles a los LLM múltiples ejemplos para ayudarlos a comprender la tarea, mientras que la indicación de una sola toma o cero tomas significa que se brinda un solo ejemplo o ningún ejemplo.
- LangChain es una biblioteca de Python diseñada para facilitar el uso de LLM en diversas aplicaciones. Simplifica la interacción con diferentes LLM y aplicaciones. Permite al agente acceder automáticamente a la herramienta adecuada en función de la tarea en cuestión, sin necesidad de indicarle explícitamente qué hacer.
- Los LLM modernos previamente entrenados, como la serie GPT de OpenAI, pueden crear diversos formatos de contenido, como texto, imágenes, audio y código.
- A pesar de sus impresionantes logros, los LLM carecen de una verdadera comprensión del contenido o de la capacidad de razonamiento. Estas limitaciones pueden conducir a errores de lógica, inexactitudes factuales y a la incapacidad de comprender conceptos o matices complejos. Además, el rápido avance y la aplicación generalizada de estos modelos han generado numerosas preocupaciones éticas, como sesgos, desinformación, violación de la privacidad y violaciones de derechos de autor. Estas cuestiones exigen una cuidadosa consideración y medidas proactivas para garantizar que el desarrollo y la implementación de los LLM se ajusten a los estándares éticos y los valores sociales.

¹ Maggiori, Emmanuel, 2023, *Inteligente hasta que se vuelve tonto: por qué la inteligencia artificial sigue cometiendo errores épicos (y por qué la burbuja de la IA estallará)*, Applied Maths Ltd. Edición Kindle.

² Adi Robertson, 21 de febrero de 2024, “Google se disculpa por 'fallar el tiro' después de que Gemini generara nazis racialmente diversos”. The Verge, <https://mng.bz/2ga9>.

Apéndice A. Instalación de Python, Jupyter Notebook y PyTorch

Existen varias maneras de instalar Python y administrar bibliotecas y paquetes en su computadora. Este libro utiliza Anaconda, una distribución de Python de código abierto, un gestor de paquetes y una herramienta de gestión de entornos. Anaconda destaca por su facilidad de uso y su capacidad para facilitar la instalación sin esfuerzo de numerosas bibliotecas y paquetes, cuya instalación podría ser engorrosa o prácticamente imposible de otro modo.

En concreto, Anaconda permite a los usuarios instalar paquetes a través de tanto "conda install" como "pip install" amplían la gama de recursos disponibles. Este apéndice le guiará en la creación de un entorno virtual de Python dedicado para todos los proyectos de este libro. Esta segmentación garantiza que las bibliotecas y los paquetes utilizados en este libro permanezcan aislados de cualquier biblioteca utilizada en otros proyectos no relacionados, eliminando así cualquier posible interferencia.

Usaremos Jupyter Notebook como nuestro entorno de desarrollo integrado (IDE). Te guiaré en la instalación de Jupyter Notebook en el entorno virtual de Python que acabas de crear. Finalmente, te guiaré en el proceso de instalación de PyTorch, Torchvision y Torchaudio, dependiendo de si tu ordenador cuenta con una GPU compatible con CUDA (Compute Unified Device Architecture).

A.1 Instalación de Python y configuración de un entorno virtual

En esta sección, te guiaré en el proceso de instalación de Anaconda en tu computadora, según tu sistema operativo. Después, crearás un entorno virtual de Python para todos los proyectos de este libro. Finalmente, instalarás Jupyter Notebook como tu IDE para ejecutar los programas de Python de este libro.

A.1.1 Instalación de Anaconda

Para instalar Python a través de la distribución Anaconda, siga estos pasos.

Primero, vaya a <https://www.anaconda.com/download/success> y desplácese hasta el final de la página web. Busque y descargue la versión más reciente de Python 3, adaptada a su sistema operativo (Windows, macOS o Linux).

Si usa Windows, descargue el instalador gráfico más reciente de Python 3 desde este enlace. Haga clic en el instalador y siga las instrucciones. Para confirmar la instalación correcta de Anaconda en su computadora, busque la aplicación "Anaconda Navigator". Si puede iniciar la aplicación, Anaconda se habrá instalado correctamente.

Para usuarios de macOS, se recomienda el instalador gráfico más reciente de Python 3 para Mac, aunque también existe la opción de instalación desde la línea de comandos. Ejecute el instalador y siga las instrucciones. Verifique la instalación correcta de Anaconda buscando la aplicación "Anaconda Navigator" en su computadora. Si puede iniciar la aplicación, Anaconda se habrá instalado correctamente.

El proceso de instalación para Linux es más complejo que para otros sistemas operativos, ya que no hay instalador gráfico. Empiece por identificar la última versión de Linux. Seleccione el paquete x86 o Power8 y Power9 adecuado. Haga clic para descargar el script de instalación más reciente. Este script se guarda generalmente en la carpeta de Descargas de su ordenador por defecto. Instale Anaconda ejecutando el script en una terminal. Al finalizar la instalación, actívelo ejecutando el siguiente comando:

```
source ~/.bashrc
```

Para acceder a Anaconda Navigator, ingrese el siguiente comando en una terminal:

```
anaconda-navigator
```

Si puede iniciar correctamente Anaconda Navigator en su sistema Linux, la instalación de Anaconda estará completa.

Ejercicio A.1

Instale Anaconda en su computadora según su sistema operativo. Después, abra la aplicación Anaconda Navigator para confirmar la instalación.

A.1.2 Configuración de un entorno virtual de Python

Se recomienda encarecidamente crear un entorno virtual independiente para este libro. Llamémoslo *dgai*. Ejecute el siguiente comando en el prompt de Anaconda (Windows) o en una terminal (Mac y Linux):

```
conda create -n dgai
```

Tras pulsar la tecla Intro del teclado, siga las instrucciones en pantalla y pulse y cuando se le solicite. Para activar el entorno virtual, ejecute el siguiente comando en el mismo símbolo del sistema de Anaconda (Windows) o terminal (Mac y Linux):

```
conda activate dgai
```

El entorno virtual aísla los paquetes y bibliotecas de Python que utiliza para este libro de otros paquetes y bibliotecas que utiliza para otros fines. Esto evita cualquier interferencia no deseada.

Ejercicio A.2

Crea un entorno virtual de Python, "dgai", en tu ordenador. Después, actívalo.

A.1.3 Instalación de Jupyter Notebook

Ahora, instalemos Jupyter Notebook en el entorno virtual recién creado en su computadora.

Primero, active el entorno virtual ejecutando la siguiente línea de código en el símbolo del sistema de Anaconda (en Windows) o en una terminal (en Mac o Linux):

```
conda activate dgai
```

Para instalar Jupyter Notebook en el entorno virtual, ejecute el comando

```
conda install notebook
```

Siga las instrucciones en pantalla hasta el final para instalar la aplicación.

Para iniciar Jupyter Notebook, ejecute el siguiente comando:

```
jupyter notebook
```

La aplicación Jupyter Notebook se abrirá en su navegador predeterminado.

Ejercicio A.3

Instale Jupyter Notebook en el entorno virtual de Python *dgai*. A continuación, abra la aplicación Jupyter Notebook en su ordenador para confirmar la instalación.

A.2 Instalación de PyTorch

En esta sección, te guiaré en la instalación de PyTorch, dependiendo de si tienes una GPU compatible con CUDA en tu ordenador. El sitio web oficial de PyTorch, <https://pytorch.org/get-started/locally/>, ofrece actualizaciones sobre la instalación de PyTorch con o sin CUDA. Te recomiendo que consultes el sitio web para ver las actualizaciones.

CUDA solo está disponible en Windows o Linux, no en Mac. Para saber si tu computadora tiene una GPU compatible con CUDA, abre Windows PowerShell (en Windows) o una terminal (en Linux) y ejecute el siguiente comando:

```
nvidia-smi
```

Si su computadora tiene una GPU compatible con CUDA, debería ver un resultado similar al de la figura A.1. Además, anote la versión de CUDA, como se muestra en la esquina superior derecha de la figura, ya que la necesitará más adelante al instalar PyTorch. La figura A.1 muestra que la versión de CUDA en mi computadora es la 11.8. Es posible que la versión sea diferente en su computadora.

```
PS C:\Users\mark> nvidia-smi
Sat Nov 18 07:53:08 2023
+-----+
| NVIDIA-SMI 522.06      Driver Version: 522.06      CUDA Version: 11.8 |
+-----+
GPU  Name        TCC/WDDM	Bus-Id      Disp.A	Volatile Uncorr. ECC	
Fan  Temp  Perf  Pwr:Usage/Cap	Memory-Usage	GPU-Util  Compute M.	
			MIG M.
+-----+			
0  NVIDIA GeForce ... WDDM	00000000:01:00.0 On	N/A	
37%   31C     P8    12W / 175W	614MiB / 8192MiB	5%       Default	
			N/A
+-----+
```

Figura A.1 Comprobación de si su computadora tiene una GPU habilitada para CUDA

Si ve un mensaje de error después de ejecutar el comando `nvidia-smi`, su computadora no tiene una GPU habilitada para CUDA.

En la primera subsección, explicaré cómo instalar PyTorch si no tienes una GPU compatible con CUDA en tu ordenador. Puedes usar la CPU para entrenar todos los modelos de IA generativa de este libro. Solo que lleva mucho más tiempo. Sin embargo, te proporcionaré los modelos preentrenados para que puedas observar la IA generativa en acción.

Por otro lado, si estás usando un sistema operativo Windows o Linux y tienes una GPU habilitada para CUDA en tu computadora, te guiaré a través de la instalación de PyTorch con CUDA en la segunda subsección.

A.2.1 Instalación de PyTorch sin CUDA

Para instalar PyTorch con entrenamiento de CPU, primero active el entorno virtual `dgai` ejecutando la siguiente línea de código en el indicador de Anaconda (en Windows) o en una terminal (en Mac o Linux):

```
conda activate dgai
```

Debería ver "*dgai*" delante del símbolo del sistema, lo que indica que ahora se encuentra en el entorno virtual *dgai*. Para instalar PyTorch, ejecute el siguiente comando:

```
conda install pytorch torchvision  
torchaudio ccpuonly -c pytorch
```

Siga las instrucciones en pantalla para finalizar la instalación. Aquí, instalamos tres bibliotecas juntas: PyTorch, Torchaudio y Torchvision. Torchaudio es una biblioteca para procesar audio y señales, y la necesitamos para generar música en este libro. También usaremos ampliamente la biblioteca Torchvision para procesar imágenes.

Si tu Mac tiene un chip Apple o una GPU AMD con macOS 12.3 o posterior, puedes usar el nuevo backend Metal Performance Shaders para acelerar el entrenamiento de la GPU. Más información disponible en <https://developer.apple.com/metal/pytorch/> y <https://pytorch.org/get-started/locally/>.

Para comprobar si las tres bibliotecas se instalaron correctamente en su computadora, ejecute las siguientes líneas de código:

```
import torch, torchvision, torchaudio  
  
print(torch.__version__)  
print(torchvision.__version__)  
print(torchaudio.__version__)
```

La salida en mi computadora dice

```
2.0.1  
0.15.2  
2.0.2
```

Si no ve ningún mensaje de error, habrá instalado PyTorch correctamente en su computadora.

A.2.2 Instalación de PyTorch con CUDA

Para instalar PyTorch con CUDA, primero averigüe la versión de CUDA de su GPU, como se muestra en la esquina superior derecha de la figura A.1. Mi versión de CUDA es la 11.8, así que la usaré como ejemplo en la instalación.

Si va al sitio web de PyTorch en <https://pytorch.org/get-started/locally/>, verá una interfaz interactiva como se muestra en la figura A.2.

Una vez allí, seleccione su sistema operativo, Conda como paquete, Python como lenguaje y CUDA 11.8 o CUDA 12.1 como plataforma (según lo que haya averiguado en el paso anterior). Si la versión de CUDA en su equipo no es ni la 11.8 ni la 12.1, elija la más cercana a su versión y funcionará. Por ejemplo, si un equipo tiene la versión 12.4 de CUDA y alguien usó CUDA 12.1, la instalación se realizará correctamente.

El comando que debe ejecutar se mostrará en el panel inferior. Por ejemplo, si uso Windows y tengo CUDA 11.8 en mi GPU, el comando se muestra en el panel inferior de la figura A.2.

| PyTorch Build | Stable (2.1.1) | | Preview (Nightly) | |
|-------------------|--|-----------|-------------------|------------|
| Your OS | Linux | Mac | Windows | |
| Package | Conda | Pip | LibTorch | Source |
| Language | Python | | | C++ / Java |
| Compute Platform | CUDA 11.8 | CUDA 12.1 | ROCM 5.6 | CPU |
| Run this Command: | <pre>conda install pytorch torchvision torchaudio pytorch-cuda=11.8 -c pytorch -c nvidia</pre> | | | |

Figura A.2 La interfaz interactiva sobre cómo instalar PyTorch

Una vez que sepa qué comando ejecutar para instalar PyTorch con CUDA, active el entorno virtual ejecutando la siguiente línea de código en el símbolo del sistema de Anaconda (Windows) o en una terminal (Linux):

```
conda activate dgai
```

Luego, ejecute la línea de comando que encontró en el paso anterior. Para mí, la línea de comando es...

```
conda install pytorch torchvision torchaudio  
pytorch-cuda=11.8 -c pytorch -c nvidia
```

Siga las instrucciones en pantalla para finalizar la instalación. Aquí instalamos tres bibliotecas juntas: PyTorch, Torchaudio y Torchvision. Torchaudio es una biblioteca para procesar audio y señales, y la necesitamos para generar música en este libro. También usamos ampliamente la biblioteca Torchvision para procesar imágenes.

Para asegurarse de tener PyTorch instalado correctamente, ejecute las siguientes líneas de código en una nueva celda en Jupyter Notebook:

```
import torch, torchvision, torchaudio  
  
print(torch.__version__)  
print(torchvision.__version__)  
print(torchaudio.__version__)  
device="cuda" if torch.cuda.is_available() else "cpu"  
print(device)
```

El resultado en mi ordenador es el siguiente:

```
2.0.1  
0.15.2  
2.0.2  
cuda
```

La última línea del resultado `cuda` indica que he instalado PyTorch con CUDA. Si ha instalado PyTorch sin CUDA en su ordenador, el resultado es `cpu`.

Ejercicio A.4

Instala PyTorch, Torchvision y Torchaudio en tu ordenador según tu sistema operativo y si cuenta con aceleración de entrenamiento por GPU.

Después, imprime las versiones de las tres bibliotecas que acabas de instalar.

Apéndice B. Lectores mínimamente calificados y fundamentos del aprendizaje profundo

Este libro está dirigido a entusiastas del aprendizaje automático y científicos de datos de diversos sectores empresariales con conocimientos intermedios de programación en Python y interesados en aprender sobre IA generativa. A través de este libro, los lectores aprenderán a crear contenido novedoso e innovador (como imágenes, texto, números, formas y audio) que puede beneficiar a las empresas de sus empleadores y a impulsar sus propias carreras profesionales.

Este libro está diseñado para quienes tienen un conocimiento sólido de Python. Debe estar familiarizado con tipos de variables como enteros, flotantes, cadenas y booleanos. También debe sentirse cómodo creando bucles *for* y *while*, y comprender la ejecución condicional y las ramificaciones (por ejemplo, usando sentencias *if*, *elif* y *else*). El libro aborda el uso frecuente de funciones y clases de Python, y debe saber cómo instalar e importar bibliotecas y paquetes de Python de terceros. Si necesita repasar estas habilidades, el tutorial gratuito de Python en línea de W3Schools es un excelente recurso (<https://www.w3schools.com/python/>).

Además, debe tener conocimientos básicos de aprendizaje automático, en particular de redes neuronales y aprendizaje profundo. En este apéndice, revisaremos conceptos clave como funciones de pérdida, funciones de activación y optimizadores, esenciales para el desarrollo y entrenamiento de redes neuronales profundas. Sin embargo, este apéndice no pretende ser un tutorial exhaustivo sobre estos temas. Si encuentra lagunas en su comprensión, le recomendamos encarecidamente que las aborde antes de continuar con los proyectos de este libro. Un buen libro para este propósito es "*Aprendizaje profundo con PyTorch*" de Stevens, Antiga y Viehmann (2020).

[1](#)

No se requiere experiencia previa con PyTorch ni con IA generativa. En el capítulo 2, aprenderá los fundamentos de PyTorch, comenzando con sus ti-

pos de datos básicos. También implementará un proyecto integral de aprendizaje profundo en PyTorch para adquirir experiencia práctica. El objetivo del capítulo 2 es prepararle para usar PyTorch para construir y entrenar varios modelos generativos del libro.

B.1 Aprendizaje profundo y redes neuronales profundas

El aprendizaje automático (ML) representa un nuevo paradigma en IA. A diferencia de la IA tradicional basada en reglas, que implica programar reglas explícitas en una computadora, el ML implica alimentar a la computadora con diversos ejemplos y permitirle aprender las reglas por sí misma. El aprendizaje profundo es un subconjunto del ML que emplea redes neuronales profundas para este proceso de aprendizaje.

En esta sección, aprenderá sobre las redes neuronales y por qué algunas se consideran redes neuronales profundas.

B.1.1 Anatomía de una red neuronal

Una red neuronal busca imitar el funcionamiento del cerebro humano. Consta de una capa de entrada, una capa de salida y ninguna, una o más capas ocultas intermedias. El término "redes neuronales profundas" se refiere a redes con muchas capas ocultas, que suelen ser más potentes.

Comenzaremos con un ejemplo más simple que presenta dos capas ocultas, como se muestra en la figura B.1.

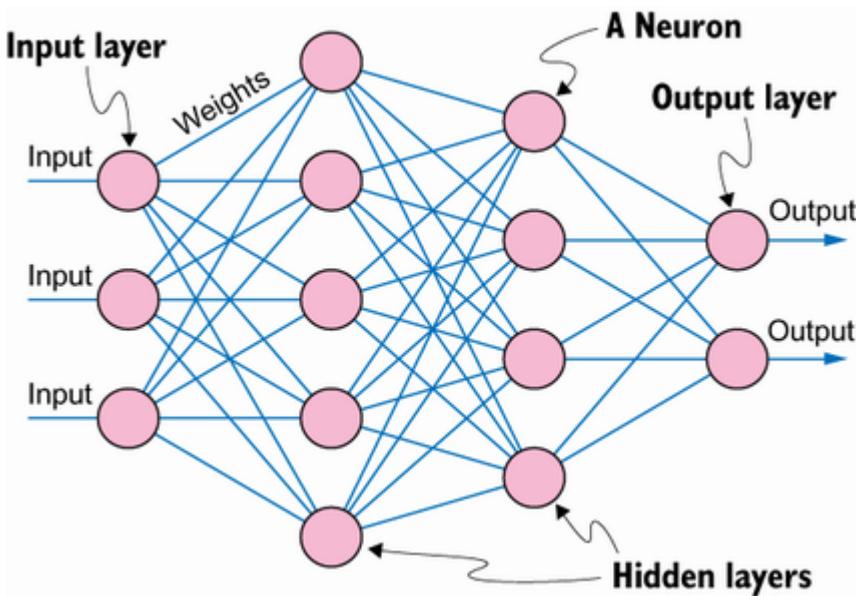


Figura B.1 Estructura de una red neuronal. Una red neuronal se compone de una capa de entrada; cero, una o más capas ocultas; y una capa de salida. Cada capa contiene una o más neuronas. Las neuronas de cada capa están conectadas con las de las capas anterior y posterior, y la fuerza de estas conexiones se representa mediante pesos. En esta figura, la red neuronal presenta una capa de entrada con tres neuronas, dos capas ocultas con seis y cuatro neuronas, respectivamente, y una capa de salida con dos neuronas.

Una red neuronal consta de una capa de entrada, un número variable de capas ocultas y una capa de salida. Cada capa está compuesta por una o más neuronas. Las neuronas de una capa se conectan con las neuronas de las capas anterior y siguiente, y la intensidad de la conexión se mide mediante pesos. En el ejemplo ilustrado en la figura B.1, la red neuronal presenta una capa de entrada con tres neuronas, dos capas ocultas con seis y cuatro neuronas, respectivamente, y una capa de salida con dos neuronas.

B.1.2 Diferentes tipos de capas en redes neuronales

Dentro de una red neuronal, existen diversos tipos de capas con distintas funciones. La más común es la capa densa, donde cada neurona está conectada con todas las neuronas de la capa siguiente. Debido a esta conectividad total, una capa densa también se denomina capa completamente conectada.

Otro tipo de capa neuronal de uso frecuente, especialmente en este libro, es la capa convolucional. Las capas convolucionales tratan la entrada como datos multidimensionales y son expertas en extraer patrones de ella. En nuestro libro, las capas convolucionales se emplean a menudo para extraer características espaciales de las imágenes.

Las capas convolucionales difieren de las capas completamente conectadas (densas) en varios aspectos clave. En primer lugar, cada neurona de una capa convolucional se conecta solo a una pequeña región de la entrada. Este diseño se basa en la comprensión de que, en los datos de imagen, los grupos locales de píxeles tienen mayor probabilidad de estar relacionados. Esta conectividad local reduce significativamente el número de parámetros, lo que aumenta la eficiencia de las redes neuronales convolucionales (CNN). En segundo lugar, las CNN utilizan pesos compartidos: se aplican los mismos pesos a diferentes regiones de la entrada. Este mecanismo es similar a deslizar un filtro por todo el espacio de entrada. Este filtro detecta características específicas (p. ej., bordes o texturas) independientemente de su posición en la entrada, lo que da lugar a la propiedad de invariancia de la traslación. Debido a su estructura, las CNN son más eficientes para el procesamiento de imágenes, ya que requieren menos parámetros que las redes completamente conectadas de tamaño similar. Esto se traduce en tiempos de entrenamiento más rápidos y menores costos computacionales. Además, las CNN suelen ser más eficaces para capturar jerarquías espaciales en datos de imagen. Analizamos las CNN en detalle en el capítulo 4.

El tercer tipo de red neuronal es la red neuronal recurrente (RNN). Las redes completamente conectadas tratan cada entrada de forma independiente, procesándola por separado sin considerar la relación ni el orden entre las diferentes entradas. En cambio, las RNN están diseñadas específicamente para procesar datos secuenciales. En una RNN, la salida en un intervalo de tiempo determinado depende no solo de la entrada actual, sino también de las anteriores. Esto permite a las RNN mantener una especie de memoria, capturando información de intervalos de tiempo anteriores para influir en el procesamiento de la entrada actual. Consulte el capítulo 8 para obtener más información sobre las RNN.

B.1.3 Funciones de activación

Las funciones de activación son un componente crucial de las redes neuronales, ya que funcionan como mecanismos que transforman las entradas en salidas y determinan cuándo debe activarse una neurona. Algunas funciones son similares a interruptores de encendido y apagado, y desempeñan un papel fundamental en el aumento del poder de las redes neuronales. Sin funciones de activación, las redes neuronales se limitarían a aprender únicamente relaciones lineales en los datos. Al introducir la no linealidad,

dad, las funciones de activación permiten la creación de relaciones complejas y no lineales entre entradas y salidas.

La función de activación más utilizada es la unidad lineal rectificada (ReLU). Una ReLU activa la neurona cuando la entrada es positiva, permitiendo el paso de la información. Cuando la entrada es negativa, la neurona se desactiva. Este sencillo comportamiento de activación y desactivación facilita el modelado de relaciones no lineales.

Otra función de activación comúnmente utilizada es la función sigmoidea, especialmente adecuada para problemas de clasificación binaria. Esta función comprime las entradas en un rango entre 0 y 1, representando eficazmente las probabilidades de un resultado binario.

Para tareas de clasificación multicategoría, se emplea la función softmax. Esta función transforma un vector de valores en una distribución de probabilidad donde la suma de los valores es 1. Esto es ideal para modelar las probabilidades de múltiples resultados.

Por último, cabe destacar la función de activación tanh. Similar a la función sigmoidea, tanh produce valores entre -1 y 1. Esta característica es especialmente útil cuando se trabaja con imágenes, ya que los datos de imagen a menudo contienen valores dentro de este rango.

B.2 Entrenamiento de una red neuronal profunda

Esta sección ofrece una descripción general de los pasos necesarios para entrenar una red neuronal. Un aspecto clave de este proceso es dividir el conjunto de datos de entrenamiento en un conjunto de entrenamiento, un conjunto de validación y un conjunto de prueba, lo cual es crucial para desarrollar una red neuronal profunda robusta. También analizaremos diversas funciones de pérdida y optimizadores utilizados en el entrenamiento de redes neuronales.

B.2.1 El proceso de formación

Una vez construida una red neuronal, el siguiente paso es recopilar un conjunto de datos de entrenamiento para entrenar el modelo. La Figura B.2 ilustra los pasos del proceso de entrenamiento.

En el lado izquierdo de la figura B.2, vemos la división inicial del conjunto de datos de entrenamiento en tres subconjuntos: el conjunto de entrenamiento, el conjunto de validación y el conjunto de prueba. Esta división es crucial para construir una red neuronal profunda robusta. El conjunto de entrenamiento es el subconjunto de datos utilizado para entrenar el modelo, donde este aprende patrones, ponderaciones y sesgos. El conjunto de validación se utiliza para evaluar el rendimiento del modelo durante el entrenamiento y para decidir cuándo detenerlo. El conjunto de prueba se utiliza para evaluar el rendimiento final del modelo una vez finalizado el entrenamiento, lo que proporciona una evaluación imparcial de su capacidad para generalizarse a datos nuevos e inéditos.

Durante la fase de entrenamiento, el modelo se entrena con los datos del conjunto de entrenamiento. Ajusta iterativamente sus parámetros para minimizar la función de pérdida (véase la siguiente subsección sobre las diferentes funciones de pérdida). Tras cada época, se evalúa el rendimiento del modelo utilizando el conjunto de validación. Si el rendimiento en el conjunto de validación continúa mejorando, el entrenamiento continúa. Si deja de mejorar, se detiene para evitar el sobreajuste.

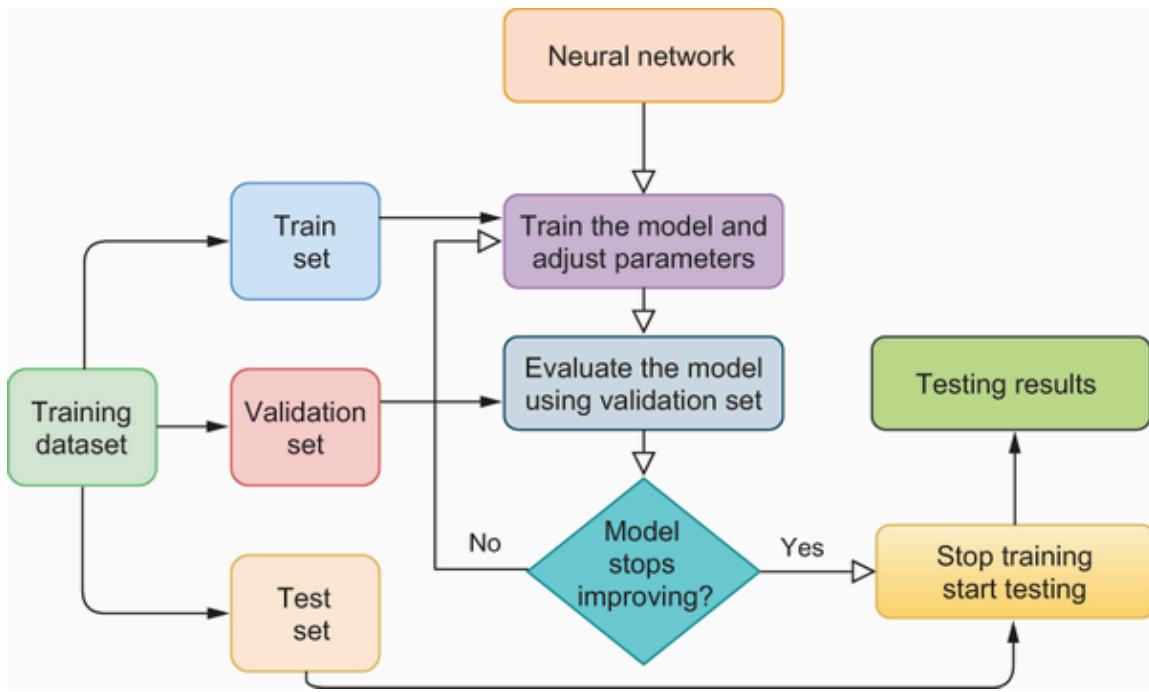


Figura B.2 Entrenamiento de una red neuronal. El conjunto de datos de entrenamiento se divide en tres subconjuntos: el conjunto de entrenamiento, el conjunto de validación y el conjunto de prueba. El proceso para entrenar una red neuronal implica los siguientes pasos. En la fase de entrenamiento, el conjunto de entrenamiento se utiliza para entrenar la red neuronal y ajustar sus parámetros para minimizar la función de pérdida. Durante cada iteración de entrenamiento, el modelo actualiza sus parámetros basándose en los datos en el conjunto de entrenamiento. En la fase de validación de cada iteración, el modelo se evalúa utilizando el conjunto de validación. El rendimiento en el conjunto de validación ayuda a determinar si el modelo sigue mejorando. Si el rendimiento del modelo en el conjunto de validación continúa mejorando, la siguiente iteración de entrenamiento continúa utilizando el conjunto de entrenamiento. Si el rendimiento del modelo en el conjunto de validación deja de mejorar, el proceso de entrenamiento se detiene para evitar el sobreajuste. Una vez completado el entrenamiento, el modelo entrenado se evalúa en el conjunto de prueba. Esta evaluación proporciona los resultados finales de la prueba, dando una estimación del rendimiento del modelo en datos no vistos.

Una vez finalizado el entrenamiento, comienza la fase de pruebas. El modelo se aplica al conjunto de pruebas (datos no vistos) para evaluar su rendimiento final y generar informes de resultados.

Dividir el conjunto de datos en tres conjuntos diferentes es esencial por varias razones. El subconjunto de entrenamiento permite al modelo aprender patrones y características de los datos y ajustar sus parámetros. El subconjunto de validación sirve para comprobar el sobreajuste, permitiendo la monitorización del rendimiento durante el entrenamiento. El subconjunto de prueba proporciona una evaluación imparcial de la capacidad de generalización del modelo, estimando su rendimiento en condiciones reales.

Al dividir adecuadamente los datos y utilizar cada conjunto para el propósito previsto, garantizamos que el modelo esté bien entrenado y evaluado de manera imparcial.

B.2.2 Funciones de pérdida

Las funciones de pérdida son esenciales para medir la precisión de nuestras predicciones y guiar el proceso de optimización al entrenar redes neuronales profundas.

Una función de pérdida comúnmente utilizada es el error cuadrático medio (MSE o pérdida L2). El MSE calcula la diferencia cuadrática promedio entre las predicciones del modelo y los valores reales. Una función de pérdida estrechamente relacionada es el error absoluto medio (MAE o pérdida L1). El MAE calcula la diferencia absoluta promedio entre las predicciones y los valores reales. El MAE se utiliza a menudo si los datos son ruidosos y presentan muchos valores atípicos, ya que penaliza los valores extremos en menor medida que la pérdida L2.

Para tareas de clasificación binaria, donde las predicciones son binarias (0 o 1), la función de pérdida preferida es la entropía cruzada binaria. Esta función mide la diferencia promedio entre las probabilidades predichas y las etiquetas binarias reales.

En tareas de clasificación multicategoría, donde las predicciones pueden tomar múltiples valores discretos, se emplea la función de pérdida de entropía cruzada categórica. Esta función mide la diferencia promedio entre las distribuciones de probabilidad predichas y las distribuciones reales.

Durante el entrenamiento de modelos de aprendizaje automático, como las redes neuronales profundas, ajustamos los parámetros del modelo para minimizar la función de pérdida. La magnitud del ajuste es proporcional a la primera derivada de la función de pérdida con respecto a los parámetros del modelo. La tasa de aprendizaje controla la velocidad de estos ajustes. Si la tasa de aprendizaje es demasiado alta, los parámetros del modelo pueden oscilar alrededor de los valores óptimos y nunca converger. Por el contrario, si la tasa de aprendizaje es demasiado baja, el proceso de aprendizaje se vuelve lento y los parámetros tardan mucho en converger.

B.2.3 Optimizadores

Los optimizadores son algoritmos que se utilizan en el entrenamiento de redes neuronales profundas para ajustar los pesos del modelo y minimizar

la función de pérdida. Guían el proceso de aprendizaje al determinar cómo deben actualizarse los parámetros del modelo en cada paso, mejorando así el rendimiento a lo largo del tiempo.

Un ejemplo de optimizador es un descenso de gradiente estocástico (SGD). Un SGD ajusta los pesos moviéndolos en la dirección del gradiente negativo de la función de pérdida. Actualiza los pesos utilizando un subconjunto de datos (minilote) en cada iteración, lo que ayuda a acelerar el proceso de entrenamiento y mejorar la generalización.

En este libro, el optimizador más utilizado es Adam (Estimación Adaptativa de Momentos). Adam combina las ventajas de otras dos extensiones de SGD: AdaGrad y RMSProp. Calcula tasas de aprendizaje adaptativo para cada parámetro basándose en estimaciones del primer y segundo momento de los gradientes. Esta adaptabilidad hace que Adam sea especialmente adecuado para problemas que involucran grandes conjuntos de datos o numerosos parámetros.

¹ Eli Stevens, Luca Antiga y Thomas Viehmann, 2020, *Aprendizaje profundo con PyTorch*, Manning Publications.