

O'REILLY®

Second
Edition

Generative Deep Learning

Teaching Machines to Paint, Write,
Compose, and Play



David Foster
Foreword by Karl Friston

Elogios por el aprendizaje profundo generativo

Aprendizaje profundo generativo es una introducción accesible al conjunto de herramientas de aprendizaje profundo para el modelado generativo. Si es un profesional creativo al que le encanta jugar con el código y desea aplicar el aprendizaje profundo a su trabajo, este es el libro para usted.

—David Ha, jefe de estrategia, IA de estabilidad

Un libro excelente que profundiza en todas las técnicas principales detrás del aprendizaje profundo generativo de última generación. Encontrará explicaciones intuitivas y analogías inteligentes, respaldadas por ejemplos de código didácticos y muy legibles. ¡Una emocionante exploración de uno de los dominios más fascinantes de la IA!

—François Chollet, creador de Keras

Las explicaciones de David Foster sobre conceptos complejos son claras y concisas, y están enriquecidas con elementos visuales intuitivos, ejemplos de código y ejercicios. ¡Un excelente recurso para estudiantes y profesionales!

—Suzana Ilić, directora y principal responsable de programas de IA
Microsoft Azure OpenAI

La IA generativa es el próximo paso revolucionario en la tecnología de IA que tendrá un impacto masivo en el mundo. Este libro proporciona una excelente introducción a este campo y su increíble potencial y riesgos potenciales.

—Connor Leahy, director ejecutivo de Conjecture y cofundador de EleutherAI

Predecir el mundo significa comprenderlo, en todas sus modalidades. En ese sentido, la IA generativa está resolviendo el núcleo mismo de la

inteligencia.

—Jonas Andrulis, fundador y director ejecutivo de Aleph Alpha

La IA generativa está remodelando innumerables industrias e impulsando una nueva generación de herramientas creativas. Este libro es la manera perfecta de comenzar con el modelado generativo y comenzar a construir usted mismo con esta tecnología revolucionaria.

—Ed Newton-Rex, vicepresidente de audio de Stability AI y compositor

David me enseñó todo lo que sé sobre el aprendizaje automático y tiene una habilidad especial para explicar los conceptos subyacentes. El aprendizaje profundo generativo es mi recurso de referencia para la IA generativa y se encuentra en un estante junto a mi escritorio, entre mi pequeña colección de libros técnicos favoritos.

—Zack Thoutt, CPO de AutoSalesVelocity

Es probable que la IA generativa tenga un profundo impacto en la sociedad. Este libro ofrece una introducción al campo accesible sin escatimar en detalles técnicos.

—Raza Habib, cofundadora de Humanloop

Cuando la gente me pregunta cómo empezar con la IA generativa, siempre recomiendo el libro de David. La segunda edición es asombrosa porque cubre los modelos más potentes, como los modelos de difusión y Transformers. ¡Definitivamente una herramienta imprescindible para cualquier persona interesada en la creatividad computacional!

-Dr. Tristan Behrens, experto en IA y artista de música IA residente en KI Salon Heilbronn

Denso en conocimientos tecnológicos, esta es mi literatura de referencia número uno cuando tengo ideas sobre la IA generativa. Debería estar en la estantería de todo científico de datos.

—Martin Musiol, fundador de generativeAI.net

El libro cubre la taxonomía completa de los modelos generativos con excelente detalle. Una de las mejores cosas que encontré sobre el libro

es que cubre la importante teoría detrás de los modelos, además de solidificar la comprensión del lector con ejemplos prácticos. Debo señalar que el capítulo sobre GAN es una de las mejores explicaciones que he leído y proporciona medios intuitivos para ajustar sus modelos. El libro cubre una amplia gama de modalidades de IA generativa, incluidos texto, imágenes y música. Un gran recurso para cualquiera que se inicie en GenAI.

—Aishwarya Srinivasan, científica de datos, Nube de Google

Aprendizaje profundo generativo

SEGUNDA EDICION

Enseñar a las máquinas a pintar, escribir, componer e interpretar

David Foster Prólogo de Karl Friston

Aprendizaje profundo generativo por David Foster

Copyright © 2023 Applied Data Science Partners Ltd. Todos los derechos reservados.

Impreso en los Estados Unidos de América.

Publicado por O'Reilly Media, Inc., 1005 Gravenstein

Carretera Norte, Sebastopol, CA 95472.

Los libros de O'Reilly se pueden comprar para uso educativo, comercial o promocional de ventas. También hay ediciones en línea disponibles para la mayoría de los títulos (<http://oreilly.com>). Para obtener más información, comuníquese con nuestro departamento de ventas corporativo/institucional: 800998-9938 o corporativo@oreilly.com.

Editora de adquisiciones: Nicole Butterfield

Editora de desarrollo: Michele Cronin

Editor de producción: Christopher Faucher

Redactor: Charles Roumeliotis

Correctora: Rachel Head

Indexador: Judith McConville

Diseñador de interiores: David Futato

Diseñador de portada: Karen Montgomery

Ilustradora: Kate Dullea

Julio de 2019: Primera edición

Mayo de 2023: Segunda edición

Historial de revisiones de la segunda edición

2023-04-28: Primer lanzamiento

Ver <http://oreilly.com/catalog/errata.csp?isbn=9781098134181> para obtener detalles sobre la versión.

El logotipo de O'Reilly es una marca registrada de O'Reilly Media, Inc. Aprendizaje Profundo Generativo, la imagen de portada y la imagen comercial relacionada son marcas comerciales de O'Reilly Media, Inc. Las opiniones expresadas en este trabajo son las del autor y no representan las opiniones del editor. Si bien el editor y el autor han realizado esfuerzos de buena fe para garantizar que la información y las instrucciones contenidas en este trabajo sean precisas, el editor y el autor renuncian a toda responsabilidad por errores u omisiones, incluida, entre otras, la responsabilidad por los daños resultantes del uso o confianza en este trabajo. El uso de la información y las instrucciones contenidas en este trabajo es bajo su propio riesgo.

Si algún ejemplo de código u otra tecnología que este trabajo contiene o describe está sujeto a licencias de código abierto o derechos de propiedad intelectual de otros, es su responsabilidad asegurarse de que su uso cumpla con dichas licencias y/o derechos.

978-1-098-13418-1 [LSI]

Dedicatoria

Para Alina, el vector de ruido más bonito de todos.

Introducción

Este libro se está convirtiendo en parte de mi vida. Al encontrar una copia en mi sala de estar, le pregunté a mi hijo: "¿Cuándo conseguiste esto?". Él respondió: "Cuando me lo diste", desconcertado por mi momento de último año. Al revisar varias secciones juntas, llegué a considerar el aprendizaje profundo generativo como la anatomía de Gray de la IA generativa.

El autor analiza la anatomía de la IA generativa con una claridad increíble y una autoridad tranquilizadora. Ofrece un relato verdaderamente notable de un campo en rápido movimiento, respaldado con ejemplos pragmáticos, narrativas interesantes y referencias tan actuales que se lee como una historia viva.

A lo largo de sus deconstrucciones, el autor mantiene una sensación de asombro y entusiasmo por el potencial de la IA generativa, especialmente evidente en el convincente desenlace del libro. Habiendo dejado al descubierto la tecnología, nos recuerda que estamos en los albores de una nueva era de inteligencia, una era en la que la IA generativa es un espejo de nuestro lenguaje, nuestro arte, nuestra creatividad; reflejando no sólo lo que hemos creado, sino lo que podríamos crear, lo que podemos crear, limitado únicamente por "tu propia imaginación".

El tema central de los modelos generativos en inteligencia artificial resuena profundamente en mí, porque veo exactamente los mismos temas surgiendo en las ciencias naturales; es decir, una visión de nosotros mismos como modelos generativos de nuestro mundo vivido. Sospecho que en la próxima edición de este libro leeremos sobre la confluencia de la inteligencia artificial y la natural. Hasta entonces, guardaré esta edición junto a mi copia de Anatomía de Gray y otros tesoros en mi estantería.

Karl Friston
Profesor de Neurociencia
Colegio Universitario de Londres

Prefacio

Lo que no puedo crear, no lo entiendo.

—Richard Feynman

La IA generativa es una de las tecnologías más revolucionarias de nuestro tiempo y transforma la forma en que interactuamos con las máquinas. Su potencial para revolucionar la forma en que vivimos, trabajamos y jugamos ha sido objeto de innumerables conversaciones, debates y predicciones. Pero ¿y si esta poderosa tecnología tuviera un potencial aún mayor?

¿Qué pasaría si las posibilidades de la IA generativa se extendieran más allá de nuestra imaginación actual? El futuro de la IA generativa puede ser más emocionante de lo que jamás creímos posible...

Desde nuestros inicios hemos buscado oportunidades para generar creaciones originales y hermosas. Para los primeros humanos, esto tomó la forma de pinturas rupestres que representaban animales salvajes y patrones abstractos, creados con pigmentos colocados cuidadosa y metódicamente sobre la roca. La era romántica nos dio el dominio de las sinfonías de Tchaikovsky, con su capacidad de inspirar sentimientos de triunfo y tragedia a través de ondas sonoras, entrelazadas para formar hermosas melodías y armonías. Y en los últimos tiempos, nos hemos encontrado corriendo a las librerías a medianoche para comprar historias sobre un mago ficticio, porque la combinación de letras crea una narrativa que nos obliga a pasar la página y descubrir qué le sucede a nuestro héroe.

Por lo tanto, no sorprende que la humanidad haya comenzado a plantearse la pregunta fundamental sobre la creatividad: ¿podemos crear algo que sea en sí mismo creativo?

Ésta es la pregunta que la IA generativa pretende responder. Gracias a los recientes avances en metodología y tecnología, ahora podemos construir máquinas que pueden pintar obras de arte originales en un estilo determinado, escribir bloques de texto coherentes con una estructura a largo

plazo, componer música que sea agradable de escuchar y desarrollar estrategias ganadoras para juegos complejos, generando escenarios futuros imaginarios. Este es solo el comienzo de una revolución generativa que no nos dejará otra opción que encontrar respuestas a algunas de las preguntas más importantes sobre la mecánica de la creatividad y, en última instancia, sobre lo que significa ser humano.

En resumen, nunca ha habido un mejor momento para aprender sobre la IA generativa, ¡así que comencemos!

Objetivo y enfoque

Este libro no asume ningún conocimiento previo de la IA generativa. Desarrollaremos todos los conceptos clave desde cero de una manera intuitiva y fácil de seguir, así que no se preocupe si no tiene experiencia con la IA generativa. ¡Has venido al lugar correcto!

En lugar de cubrir únicamente las técnicas que están actualmente de moda, este libro sirve como una guía completa para el modelado generativo que cubre una amplia gama de familias de modelos. No existe una técnica que sea objetivamente mejor o peor que otra; de hecho, muchos modelos de última generación ahora combinan ideas de todo el amplio espectro de enfoques del modelado generativo. Por esta razón, es importante mantenerse al tanto de los avances en todas las áreas de la IA generativa, en lugar de centrarse en un tipo particular de técnica. Una cosa es segura: el campo de la IA generativa avanza rápidamente y nunca se sabe de dónde vendrá la próxima idea innovadora.

Con esto en mente, el enfoque que adoptaré es mostrarle cómo entrenar sus propios modelos generativos con sus propios datos, en lugar de depender de modelos disponibles y previamente entrenados.

Si bien en la actualidad existen muchos modelos generativos de código abierto impresionantes que se pueden descargar y ejecutar en unas pocas líneas de código, el objetivo de este libro es profundizar en su arquitectura y diseño desde los primeros principios, para que usted obtenga una comprensión completa de cómo funcionan, funcionan y pueden codificar ejemplos de cada técnica desde cero usando Python y Keras.

En resumen, este libro puede considerarse como un mapa del panorama actual de la IA generativa que cubre tanto la teoría como las aplicaciones prácticas, incluidos ejemplos completos de modelos clave de la literatura. Revisaremos el código de cada paso a paso, con indicaciones claras que muestran cómo el código implementa la teoría que sustenta cada técnica. Este libro puede leerse de principio a fin o utilizarse como libro de

referencia al que pueda recurrir. Sobre todo, espero que os resulte una lectura útil y agradable.

NOTA

A lo largo del libro encontrarás historias breves y alegóricas que ayudarán a explicar la mecánica de algunos de los modelos que construiremos. Creo que una de las mejores maneras de enseñar una nueva teoría abstracta es convertirla primero en algo que no sea tan abstracto, como una historia, antes de sumergirse en la explicación técnica. La historia y la explicación del modelo son simplemente las mismas mecánicas explicadas en dos dominios diferentes; por lo tanto, puede resultarle útil consultar la historia relevante mientras aprende sobre los detalles técnicos de cada modelo.

Requisitos previos

Este libro asume que tienes experiencia codificando en Python. Si no está familiarizado con Python, el mejor lugar para comenzar es LearnPython.org. Hay muchos recursos gratuitos en línea que le permitirán desarrollar suficiente conocimiento de Python para trabajar con los ejemplos de este libro.

Además, dado que algunos de los modelos se describen utilizando notación matemática, será útil tener una comprensión sólida del álgebra lineal (por ejemplo, multiplicación de matrices) y la teoría de probabilidad general. Un recurso útil es el libro de Deisenroth et al. *Mathematics for Machine Learning* (Cambridge University Press), que está disponible gratuitamente.

El libro no asume ningún conocimiento previo del modelado generativo (examinaremos los conceptos clave en el Capítulo 1) o TensorFlow y Keras (estas bibliotecas se presentarán en el Capítulo 2).

Hoja de ruta

Este libro está dividido en tres partes.

La Parte I es una introducción general al modelado generativo y al aprendizaje profundo, donde exploramos los conceptos centrales que sustentan todas las técnicas en partes posteriores del libro:

En el Capítulo 1, “Modelado generativo”, definimos el modelado generativo y consideramos un ejemplo de juguete que podemos usar para comprender algunos de los conceptos clave que son importantes para todos los modelos generativos. También presentamos la taxonomía de familias de modelos generativos que exploraremos en la Parte II de este libro.

En el Capítulo 2, "Aprendizaje profundo", comenzamos nuestra exploración del aprendizaje profundo y las redes neuronales construyendo nuestro primer ejemplo de un perceptrón multicapa (MLP) usando Keras. Luego adaptamos esto para incluir capas convolucionales y otras mejoras, para observar la diferencia en el rendimiento.

La Parte II recorre las seis técnicas clave que usaremos para construir modelos generativos, con ejemplos prácticos para cada una:

En el Capítulo 3, “Codificadores automáticos variacionales”, consideramos el auto codificador variacional (VAE) y vemos cómo se puede utilizar para generar imágenes de caras y realizar transformaciones entre caras en el espacio latente del modelo.

En el Capítulo 4, “Redes generativas adversarias”, exploramos las redes generativas adversarias (GAN) para la generación de imágenes, incluidas las GAN convolucionales profundas, las GAN condicionales y mejoras como la GAN de Wasserstein que hacen que el proceso de entrenamiento sea más estable.

En el Capítulo 5, "Modelos autorregresivos", centramos nuestra atención en los modelos autorregresivos, comenzando con una introducción a las redes

neuronales recurrentes, como las redes de memoria a corto plazo (LSTM) para la generación de texto y PixelCNN para la generación de imágenes.

En el Capítulo 6, “Normalización de modelos de flujo”, nos centramos en la normalización de flujos, incluida una exploración teórica intuitiva de la técnica y un ejemplo práctico de cómo construir un modelo RealNVP para generar imágenes.

En el Capítulo 7, “Modelos basados en energía”, cubrimos modelos basados en energía, incluidos métodos importantes como cómo entrenar usando divergencia contrastiva y muestreo usando dinámica de Langevin.

En el Capítulo 8, “Modelos de difusión”, nos sumergimos en una guía práctica para crear modelos de difusión, que impulsan muchos modelos de generación de imágenes de última generación, como DALL.E 2 y Difusión estable.

Finalmente, en la Parte III nos basamos en estos fundamentos para explorar el funcionamiento interno de modelos de última generación para la generación de imágenes, escritura, composición musical y aprendizaje por refuerzo basado en modelos:

En el Capítulo 9, "Transformadores", exploramos el linaje y los detalles técnicos de los modelos StyleGAN, así como otras GAN de última generación para la generación de imágenes, como VQ-GAN.

En el Capítulo 10, "GAN avanzadas", consideramos la arquitectura Transformer, que incluye un tutorial práctico para crear su propia versión de GPT para la generación de texto.

En el Capítulo 11, “Generación musical”, centramos nuestra atención en la generación musical, incluida una guía para trabajar con datos musicales y la aplicación de técnicas como Transformers y MuseGAN.

En el Capítulo 12, “Modelos mundiales”, vemos cómo se pueden utilizar los modelos generativos en el contexto del aprendizaje por refuerzo, con la aplicación de modelos mundiales y métodos basados en transformadores.

En el Capítulo 13, “Modelos multimodales”, explicamos el funcionamiento interno de cuatro modelos multimodales de última generación que incorporan más de un tipo de datos, incluidos DALL.E 2, Imagen y Stable Diffusion para conversión de texto a generación de imágenes y Flamingo, un modelo de lenguaje visual.

En el Capítulo 14, “Conclusión”, resumimos los hitos clave de la IA generativa hasta la fecha y analizamos las formas en que la IA generativa revolucionará nuestra vida diaria en los próximos años.

Cambios en la segunda edición

Gracias a todos los que leyeron la primera edición de este libro. Estoy muy contento de que muchos de ustedes lo hayan encontrado como un recurso útil y hayan brindado comentarios sobre cosas que les gustaría ver en la segunda edición. El campo del aprendizaje profundo generativo ha progresado significativamente desde que se publicó la primera edición en 2019, por lo que, además de actualizar el contenido existente, agregué varios capítulos nuevos para alinear el material con el estado actual del arte.

El siguiente es un resumen de las principales actualizaciones, en términos de capítulos individuales y mejoras generales del libro:

El capítulo 1 ahora incluye una sección sobre las diferentes familias de modelos generativos y una taxonomía de cómo se relacionan.

El capítulo 2 contiene diagramas mejorados y explicaciones más detalladas de conceptos clave.

El capítulo 3 se actualiza con un nuevo ejemplo trabajado y explicaciones que lo acompañan.

El capítulo 4 ahora incluye una explicación de las arquitecturas GAN condicionales.

El capítulo 5 ahora incluye una sección sobre modelos autorregresivos para imágenes (por ejemplo, PixelCNN).

El capítulo 6 es un capítulo completamente nuevo, que describe el modelo RealNVP.

El capítulo 7 también es un capítulo nuevo, que se centra en técnicas como la dinámica de Langevin y la divergencia contrastiva.

El Capítulo 8 es un capítulo recién escrito sobre la eliminación de ruido de los modelos de difusión que impulsan muchas de las aplicaciones de última generación actuales.

El Capítulo 9 es una ampliación del material proporcionado al final de la primera edición, con un enfoque más profundo en las arquitecturas de los distintos modelos StyleGAN y material nuevo sobre VQ-GAN.

El capítulo 10 es un nuevo capítulo que explora la arquitectura del transformador en detalle.

El Capítulo 11 incluye arquitecturas Transformer modernas, que reemplazan los modelos LSTM de la primera edición.

El Capítulo 12 incluye diagramas y descripciones actualizados, con una sección sobre cómo este enfoque está informando el aprendizaje por refuerzo de última generación en la actualidad.

El Capítulo 13 es un nuevo capítulo que explica en detalle cómo modelos impresionantes como DALL.E 2, Imagen, Difusión Estable y trabajo Flamenco.

El Capítulo 14 se actualiza para reflejar los avances sobresalientes en el campo desde la primera edición y brindar una visión más completa y detallada de dónde la IA generativa avanza hacia el futuro.

Todos los comentarios proporcionados como retroalimentación sobre la primera edición y los errores tipográficos identificados han sido abordados (¡hasta donde yo sé!).

Los objetivos de los capítulos se agregaron al comienzo de cada capítulo, para que pueda ver los temas clave cubiertos en el capítulo antes de comenzar a leer.

Algunas de las historias alegóricas se han reescrito para que sean más concisas y claras. ¡Me complace que tantos lectores hayan dicho que las historias les han ayudado a comprender mejor los conceptos clave!

Los títulos y subtítulos de cada capítulo se han alineado para que quede claro qué partes del capítulo se centran en la explicación y cuáles en la construcción de sus propios modelos.

Otros recursos

Recomiendo encarecidamente los siguientes libros como introducciones generales al aprendizaje automático y al aprendizaje profundo:

Aprendizaje automático práctico con Scikit-Learn, Keras y TensorFlow: conceptos, herramientas y técnicas para construir sistemas inteligentes, por Aurélien Géron (O'Reilly)

Aprendizaje profundo con Python, por Francois Chollet (Manning)

La mayoría de los artículos de este libro se obtienen a través de arXiv, un depósito gratuito de artículos de investigación científica. Ahora es común que los autores publiquen artículos en arXiv antes de que sean completamente revisados por pares. Revisar las presentaciones recientes es una excelente manera de mantenerse al tanto de los desarrollos más vanguardistas en el campo.

También recomiendo encarecidamente el sitio web Papers with Code, donde puede encontrar los últimos resultados de última generación en una variedad de tareas de aprendizaje automático, junto con enlaces a los artículos y repositorios oficiales de GitHub. Es un recurso excelente para cualquiera que quiera comprender rápidamente qué técnicas logran actualmente las puntuaciones más altas en una variedad de tareas y ciertamente me ha ayudado a decidir qué técnicas incluir en este libro.

Convenciones usadas en este libro

En este libro se utilizan las siguientes convenciones tipográficas:

Itálicas

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

Ancho fijo

Se utiliza para comandos y listados de programas, así como dentro de párrafos para hacer referencia a elementos del programa como nombres de variables o funciones.

Cursiva de ancho fijo

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

CONSEJO

Este elemento significa un consejo o sugerencia.

NOTA

Este elemento significa una nota general.

ADVERTENCIA

Este elemento significa una advertencia o precaución.

Base de código

Los ejemplos de código de este libro se pueden encontrar en un repositorio de GitHub. Me he asegurado deliberadamente de que ninguno de los modelos requiera cantidades prohibitivamente grandes de recursos computacionales para entrenarse, de modo que pueda comenzar a entrenar sus propios modelos sin tener que gastar mucho tiempo o dinero en hardware costoso. Hay una guía completa en el repositorio sobre cómo comenzar con Docker y configurar recursos en la nube con GPU en Google Cloud si es necesario.

Se han realizado los siguientes cambios en el código base desde la primera edición:

Todos los ejemplos ahora se pueden ejecutar desde un solo cuaderno, en lugar de importar parte del código desde módulos de toda la base de código. Esto es para que puedas ejecutar cada ejemplo celda por celda y profundizar exactamente en cómo se construye cada modelo, pieza por pieza.

Las secciones de cada cuaderno ahora están ampliamente alineadas entre los ejemplos.

Muchos de los ejemplos de este libro ahora utilizan fragmentos de código del sorprendente repositorio de código abierto de Keras; esto es para evitar la creación de un repositorio de código abierto completamente separado de ejemplos de IA generativa de Keras, cuando ya existen excelentes implementaciones disponibles a través del sitio web de Keras. He agregado referencias y enlaces a los autores originales del código que he utilizado desde el sitio web de Keras a lo largo de este libro y en el repositorio.

Agregué nuevas fuentes de datos y mejoré el proceso de recopilación de datos desde la primera edición; ahora hay un script que se puede ejecutar fácilmente para recopilar datos de las fuentes requeridas para entrenar los ejemplos del libro, usando herramientas como API de Kaggle.

Usando los ejemplos de código

El material complementario (ejemplos de código, ejercicios, etc.) está disponible para descargar en https://github.com/davidADSP/Generative_Deep_Learning_2nd_Edition.

Si tiene una pregunta técnica o un problema al utilizar los ejemplos de código, envíe un correo electrónico a bookquestions@oreilly.com.

Este libro está aquí para ayudarle a realizar su trabajo. En general, si se ofrece código de ejemplo con este libro, puede utilizarlo en sus programas y documentación. No es necesario que se comunique con nosotros para solicitar permiso a menos que esté reproduciendo una parte importante del código. Por ejemplo, escribir un programa que utilice varios fragmentos de código de este libro no requiere permiso. Vender o distribuir ejemplos de libros de O'Reilly requiere permiso.

Responder una pregunta citando este libro y citando código de ejemplo no requiere permiso. Incorporar una cantidad significativa de código de ejemplo de este libro en la documentación de su producto requiere permiso.

Apreciamos la atribución, pero no la exigimos. Una atribución generalmente incluye el título, autor, editor e ISBN. Por ejemplo: "Aprendizaje profundo generativo, 2da. edición, de David Foster (O'Reilly). Copyright 2023 Socios de ciencia de datos aplicada Ltd., 978-1-098-13418-1.

Si cree que su uso de ejemplos de código queda fuera del uso legítimo o del permiso otorgado anteriormente, no dude en contactarnos en permisos@oreilly.com.

Aprendizaje en línea de O'Reilly

NOTA

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

Nuestra red única de expertos e innovadores comparte su conocimiento y experiencia a través de libros, artículos y nuestra plataforma de aprendizaje en línea. La plataforma de aprendizaje en línea de O'Reilly le brinda acceso bajo demanda a cursos de capacitación en vivo, rutas de aprendizaje en profundidad, entornos de codificación interactivos y una amplia colección de textos y videos de O'Reilly y más de 200 editoriales más. Para obtener más información, visite <https://oreilly.com>.

Cómo contactarnos

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

O'Reilly Media, Inc.
1005 Gravenstein Carretera Norte
Sebastopol, CA 95472

800-998-9938 (en Estados Unidos o Canadá)

707-829-0515 (internacional o local)

707-829-0104 (fax)

Tenemos una página web para este libro, donde enumeramos erratas, ejemplos y cualquier información adicional. Puede acceder a esta página en <https://oreil.ly/generative-dl>.

Envíe un correo electrónico a bookquestions@oreilly.com para comentar o hacer preguntas técnicas sobre este libro.

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

Encuéntrenos en LinkedIn: <https://linkedin.com/company/oreillymedia>

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

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

Agradecimientos

Hay muchísimas personas a las que me gustaría agradecer por ayudarme a escribir este libro.

Primero, me gustaría agradecer a todos los que se han tomado el tiempo de revisar técnicamente el libro, en particular a Vishwesh Ravi Shrimali, Lipi Deepakshi Patnaik, Luba Elliot y Lorna Barclay. Gracias también a Samir Bico por ayudarnos a revisar y probar el código base que acompaña a este libro. Su aporte ha sido invaluable.

Además, un enorme agradecimiento a mis colegas de Asociados de ciencia de datos aplicada, Ross Witesczak, Amy Bull, Ali Parandeh, Zine Eddine, Joe Rowe, Gerta Salillari, Aleshia Parkes, Evelina Kireilyte, Riccardo Tolli, Mai Do, Khaleel Syed y Will Holmes. Agradezco enormemente su paciencia conmigo mientras me tomaba el tiempo para terminar el libro y espero con ansias todos los proyectos de aprendizaje automático que completaremos juntos en el futuro. Un agradecimiento especial a Ross: si no hubiéramos decidido iniciar un negocio juntos, este libro nunca habría tomado forma, así que ¡gracias por creer en mí como su socio comercial!

También quiero agradecer a cualquiera que alguna vez me haya enseñado algo de matemáticas. Fui muy afortunado de tener profesores de matemáticas fantásticos en la escuela, quienes desarrollaron mi interés en el tema y me alentaron a seguir adelante en la universidad. Me gustaría agradecerle por su compromiso y por hacer todo lo posible para compartir conmigo sus conocimientos sobre el tema.

Un enorme agradecimiento al personal de O'Reilly por guiarme a lo largo del proceso de escritura de este libro. Un agradecimiento especial para Michele Cronin, quien ha estado ahí en cada paso, brindándome comentarios útiles y enviándome recordatorios amigables para seguir completando capítulos. También a Nicole Butterfield, Christopher Faucher, Charles Roumeliotis y Suzanne Huston por poner en producción el libro y Mike Loukides por contactarme por primera vez para preguntarme si estaría interesado en escribir un libro. Todos ustedes han apoyado mucho este

proyecto desde el principio y quiero agradecerles por brindarme una plataforma en la que escribir sobre algo que amo.

A lo largo del proceso de escritura, mi familia ha sido una fuente constante de aliento y apoyo. ¡Muchas gracias a mi madre, Gillian Foster, por revisar cada línea de texto en busca de errores tipográficos y por enseñarme a sumar en primer lugar! Su atención al detalle ha sido de gran ayuda durante la revisión de este libro y estoy realmente agradecido por todas las oportunidades que tanto tú como papá me habéis brindado. Mi padre, Clive Foster, originalmente me enseñó a programar una computadora; este libro está lleno de ejemplos prácticos, y eso se debe a su temprana paciencia mientras yo jugueteaba con BASIC tratando de crear juegos de fútbol cuando era adolescente. Mi hermano, Rob Foster, es el genio más modesto que jamás hayas encontrado, particularmente en lingüística; charlar con él sobre la IA y el futuro del aprendizaje automático basado en texto ha sido increíblemente útil.

Por último, me gustaría agradecer a mi Nana, quien siempre fue una fuente constante de inspiración y diversión para todos nosotros. Su amor por la literatura fue una de las razones por las que decidí por primera vez que escribir un libro sería algo emocionante.

También me gustaría agradecer a mi esposa, Lorna Barclay. Además de brindarme un apoyo infinito y tazas de té durante todo el proceso de escritura, usted ha revisado rigurosamente cada palabra de este libro con meticuloso detalle. No podría haberlo hecho sin ti. Gracias por estar siempre ahí para mí y por hacer que este viaje sea mucho más agradable. Prometo que no hablaré de IA generativa en la mesa durante al menos unos días después de la publicación del libro.

Por último, me gustaría agradecer a nuestra hermosa hija Alina por brindarnos entretenimiento sin fin durante las largas noches de escritura de libros. Tus adorables risitas han sido la música de fondo perfecta para mi escritura. Gracias por ser mi inspiración y por mantenerme siempre alerta.

Ustedes son los verdaderos cerebros detrás de esta operación.

Parte I. Introducción al aprendizaje profundo generativo

La Parte I es una introducción general al modelado generativo y al aprendizaje profundo: ¡los dos campos que debemos comprender para comenzar con el aprendizaje profundo generativo!

En el Capítulo 1 definiremos el modelado generativo y consideraremos un ejemplo de juguete que podemos usar para comprender algunos de los conceptos clave que son importantes para todos los modelos generativos. También expondremos la taxonomía de familias de modelos generativos que exploraremos en la Parte II de este libro.

El Capítulo 2 proporciona una guía de las herramientas y técnicas de aprendizaje profundo que necesitaremos para comenzar a construir modelos generativos más complejos. En particular, construiremos nuestro primer ejemplo de una red neuronal profunda, un perceptrón multicapa (MLP), utilizando Keras. Luego adaptaremos esto para incluir capas convolucionales y otras mejoras, para observar la diferencia en el rendimiento.

Al final de la Parte I, comprenderá bien los conceptos básicos que sustentan todas las técnicas de las partes posteriores del libro.

Capítulo 1. Modelado generativo

METAS DEL CAPÍTULO

En este capítulo podrás:

- Conocer las diferencias clave entre los modelos generativo y discriminativo.
- Comprender las propiedades deseables de un modelo generativo a través de un ejemplo sencillo.
- Conocer los conceptos probabilísticos centrales que sustentan los modelos generativos.
- Explorar las diferentes familias de modelos generativos.
- Clonar el código base que acompaña a este libro para que puedas comenzar a construir modelos generativos.

Este capítulo es una introducción general al campo del modelado generativo.

Comenzaremos con una suave introducción teórica al modelado generativo y veremos cómo es la contraparte natural del modelado discriminativo más ampliamente estudiado. Luego estableceremos un marco que describa las propiedades deseables que debe tener un buen modelo generativo. También expondremos los conceptos probabilísticos centrales que es importante conocer para apreciar plenamente cómo los diferentes enfoques abordan el desafío del modelado generativo.

Esto nos llevará naturalmente a la penúltima sección, que expone las seis grandes familias de modelos generativos que dominan el campo hoy en día. La última sección explica cómo comenzar con el código base que acompaña a este libro.

¿Qué es el modelado generativo?

El modelado generativo se puede definir ampliamente de la siguiente manera:

El modelado generativo es una rama del aprendizaje automático que implica entrenar un modelo para producir nuevos datos similares a un conjunto de datos determinado.

¿Qué significa esto en la práctica? Supongamos que tenemos un conjunto de datos que contiene fotografías de caballos. Podemos entrenar un modelo generativo en este conjunto de datos para capturar las reglas que gobiernan las complejas relaciones entre píxeles en imágenes de caballos. Luego podemos tomar muestras de este modelo para crear imágenes novedosas y realistas de caballos que no existían en el conjunto de datos original. Este proceso se ilustra en la Figura 1-1.

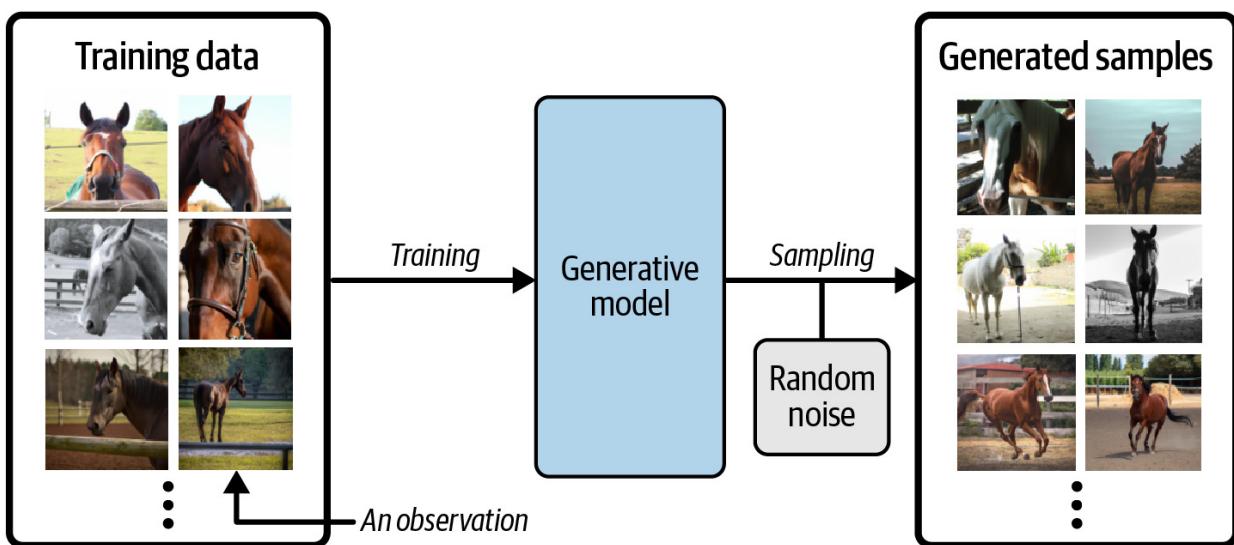


Figura 1-1. Un modelo generativo entrenado para generar fotografías realistas de caballos.

Para construir un modelo generativo, necesitamos un conjunto de datos que consta de muchos ejemplos de la entidad que estamos intentando generar. Esto se conoce como datos de entrenamiento y uno de esos puntos de datos se llama observación.

Cada observación consta de muchas características. Para un problema de generación de imágenes, las características suelen ser los valores de píxeles individuales; para un problema de generación de texto, las características podrían ser palabras individuales o grupos de letras. Nuestro objetivo es construir un modelo que pueda generar nuevos conjuntos de características

que parezcan haber sido creadas usando las mismas reglas que los datos originales. Conceptualmente, para la generación de imágenes esta es una tarea increíblemente difícil, considerando la gran cantidad de formas en que se pueden asignar valores de píxeles individuales y el número relativamente pequeño de tales disposiciones que constituyen una imagen de la entidad que estamos tratando de generar.

Un modelo generativo también debe ser probabilístico en lugar de determinista, porque queremos poder muestrear muchas variaciones diferentes del resultado, en lugar de obtener el mismo resultado cada vez. Si nuestro modelo es simplemente un cálculo fijo, como tomar el valor promedio de cada píxel en el conjunto de datos de entrenamiento, no es generativo. Un modelo generativo debe incluir un componente aleatorio que influya en las muestras individuales generadas por el modelo.

En otras palabras, podemos imaginar que existe alguna distribución probabilística desconocida que explica por qué es probable que algunas imágenes se encuentren en el conjunto de datos de entrenamiento y otras no. Nuestro trabajo es construir un modelo que imite esta distribución lo más fielmente posible y luego tomar muestras de él para generar observaciones nuevas y distintas que parezcan haber sido incluidas en el conjunto de entrenamiento original.

Modelado generativo versus discriminativo

Para comprender verdaderamente qué pretende lograr el modelado generativo y por qué es importante, es útil compararlo con su contraparte, el modelado discriminativo. Si ha estudiado aprendizaje automático, la mayoría de los problemas que haya enfrentado probablemente habrán sido de naturaleza discriminatoria. Para entender la diferencia, veamos un ejemplo.

Supongamos que tenemos un conjunto de datos de pinturas, algunas pintadas por Van Gogh y algunos de otros artistas. Con suficientes datos, podríamos entrenar un modelo discriminativo para predecir si una pintura determinada fue pintada por Van Gogh. Nuestro modelo aprendería que es más probable que ciertos colores, formas y texturas indiquen que una pintura es del maestro holandés, y para pinturas con estas características, el modelo aumentaría su predicción en consecuencia. La Figura 1-2 muestra el proceso

de modelado discriminativo; observe en qué se diferencia del proceso de modelado generativo que se muestra en la Figura 1-1.

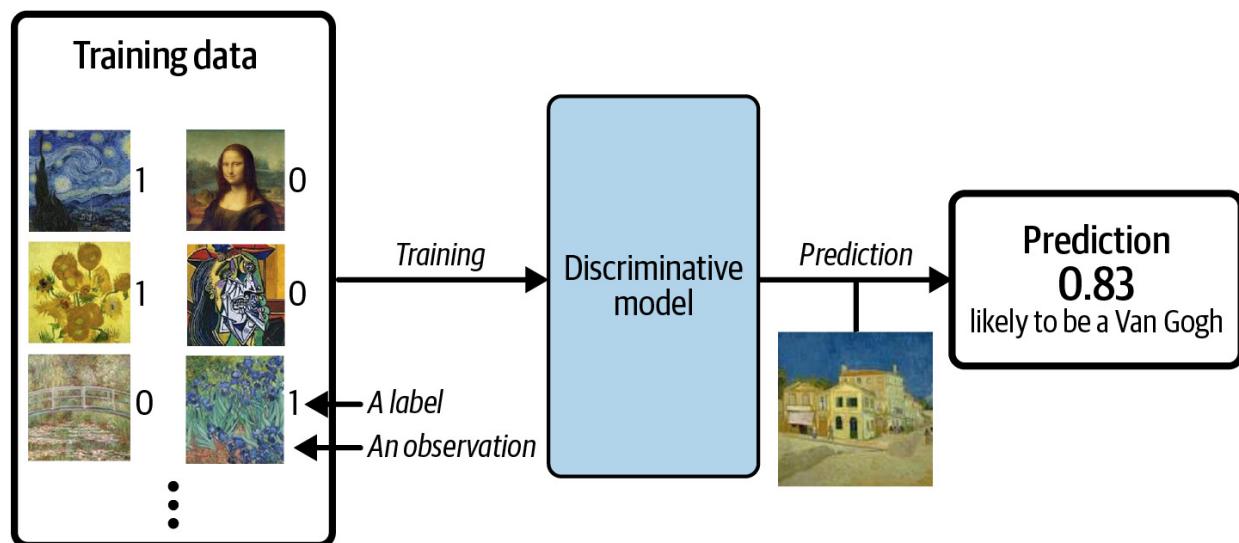


Figura 1-2. Un modelo discriminativo entrenado para predecir si una imagen determinada es pintada por Van Gogh

Al realizar un modelado discriminativo, cada observación en los datos de entrenamiento tiene una etiqueta. Para un problema de clasificación binaria como el de nuestro discriminador de artistas, las pinturas de Van Gogh se etiquetarían con 1 y las pinturas que no son de Van Gogh se etiquetarían con 0. Luego, nuestro modelo aprende a discriminar entre estos dos grupos y devuelve la probabilidad de que una nueva observación tenga la etiqueta 1, es decir, que haya sido pintada por Van Gogh.

Por el contrario, el modelado generativo no requiere que el conjunto de datos esté etiquetado porque se ocupa de generar imágenes completamente nuevas, en lugar de intentar predecir una etiqueta de una imagen determinada.

Definamos estos tipos de modelado formalmente, usando notación matemática:

Los modelos discriminativos estiman la $p(y|x)$.

Es decir, el modelado discriminativo tiene como objetivo modelar la probabilidad de una etiqueta y dada alguna observación x .

Los modelos generativos estiman la $p(x)$.

Es decir, el modelado generativo tiene como objetivo modelar la probabilidad de encontrar una observación x . El muestreo de esta distribución nos permite generar nuevas observaciones.

MODELOS GENERATIVOS CONDICIONALES

Tenga en cuenta que también podemos construir un modelo generativo para modelar la probabilidad condicional $p(x|y)$: la probabilidad de ver una observación x con una etiqueta específica y .

Por ejemplo, si nuestro conjunto de datos contiene diferentes tipos de frutas, podríamos decirle a nuestro modelo generativo que genere específicamente una imagen de una manzana.

Un punto importante a tener en cuenta es que incluso si pudiéramos construir un modelo discriminativo perfecto para identificar cuadros de Van Gogh, todavía no tendría idea de cómo crear un cuadro que se parezca a un Van Gogh. Sólo puede generar probabilidades frente a imágenes existentes, ya que para eso ha sido entrenado. En su lugar, necesitaríamos entrenar un modelo generativo y tomar muestras de este modelo para generar imágenes que tengan una alta probabilidad de pertenecer al conjunto de datos de entrenamiento original.

El auge del modelado generativo

Hasta hace poco, el modelado discriminativo ha sido la fuerza impulsora detrás de la mayor parte del progreso en el aprendizaje automático.

Esto se debe a que, para cualquier problema discriminativo, el correspondiente problema de modelado generativo suele ser mucho más difícil de abordar. Por ejemplo, es mucho más fácil entrenar un modelo para predecir si una pintura es de Van Gogh que entrenar a un modelo para generar una pintura al estilo de Van Gogh desde cero. De manera similar, es mucho más fácil entrenar un modelo para predecir si una página de texto fue escrita por Charles Dickens que construir un modelo para generar un conjunto de párrafos al estilo de Dickens. Hasta hace poco, la mayoría de los

desafíos generativos estaban simplemente fuera de nuestro alcance y muchos dudaban de que alguna vez pudieran resolverse. La creatividad se consideraba una capacidad puramente humana que la IA no podía rivalizar.

Sin embargo, a medida que las tecnologías de aprendizaje automático han madurado, esta suposición se ha debilitado gradualmente. En los últimos 10 años, muchos de los avances más interesantes en este campo se han producido a través de aplicaciones novedosas del aprendizaje automático a tareas de modelado generativo. Por ejemplo, la Figura 1-3 muestra el sorprendente progreso que ya se ha logrado en la generación de imágenes faciales desde 2014.

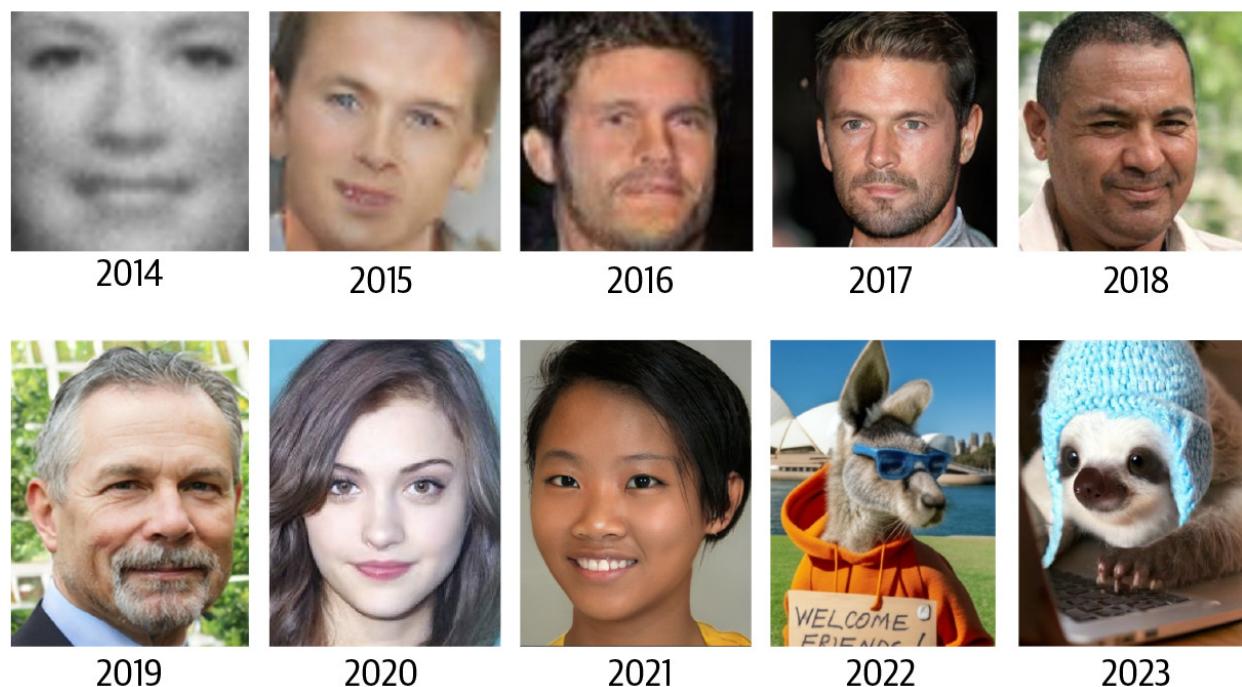


Figura 1-3. La generación de rostros mediante modelado generativo ha mejorado significativamente durante la última década (adaptado de Brundage et al., 2018)[1](#)

Además de ser más fácil de abordar, históricamente el modelado discriminativo ha sido más fácilmente aplicable a problemas prácticos en toda la industria que el modelado generativo. Por ejemplo, un médico puede beneficiarse de un modelo que predice si una imagen retiniana determinada muestra signos de glaucoma, pero no necesariamente se beneficiaría de un modelo que pueda generar imágenes novedosas del fondo del ojo.

Sin embargo, esto también está empezando a cambiar, con la proliferación de empresas que ofrecen servicios generativos dirigidos a problemas comerciales específicos. Por ejemplo, ahora es posible acceder a API que generan publicaciones de blog originales sobre un tema en particular, producir una variedad de imágenes de su producto en cualquier entorno que desee o escribir contenido de redes sociales y textos publicitarios que coincidan con su marca y mensaje objetivo. También existen claras aplicaciones positivas de la IA generativa para industrias como el diseño de juegos y la cinematografía, donde los modelos entrenados para producir video y música están comenzando a agregar valor.

Modelado generativo e IA

Además de los usos prácticos del modelado generativo (muchos de los cuales aún están por descubrir), hay tres razones más profundas por las que el modelado generativo puede considerarse la clave para desbloquear una forma mucho más sofisticada de inteligencia artificial que va más allá de lo que el modelado discriminativo por sí solo puede lograr.

En primer lugar, desde un punto de vista puramente teórico, no deberíamos limitar el entrenamiento de nuestras máquinas a simplemente categorizar datos. Para completar, también deberíamos preocuparnos de entrenar modelos que capturen una comprensión más completa de la distribución de datos, más allá de cualquier etiqueta particular. Este es sin duda un problema más difícil de resolver, debido a la alta dimensionalidad del espacio de resultados factibles y al número relativamente pequeño de creaciones que clasificaríamos como pertenecientes al conjunto de datos.

Sin embargo, como veremos, muchas de las mismas técnicas que han impulsado el desarrollo de modelos discriminativos, como el aprendizaje profundo, también pueden ser utilizados por modelos generativos.

En segundo lugar, como veremos en el capítulo 12, el modelado generativo se está utilizando ahora para impulsar el progreso en otros campos de la IA, como el aprendizaje por refuerzo (el estudio de agentes de enseñanza para optimizar una meta en un entorno mediante prueba y error). Supongamos que queremos entrenar un robot para que camine por un terreno determinado. Un enfoque tradicional sería realizar muchos experimentos en

los que el agente prueba diferentes estrategias en el terreno, o una simulación por computadora del terreno. Con el tiempo el agente aprendería qué estrategias tienen más éxito que otras y por tanto mejorarían gradualmente. Un desafío con este enfoque es que es bastante inflexible porque está capacitado para optimizar la política para una tarea en particular. Un enfoque alternativo que ha ganado fuerza recientemente es entrenar al agente para que aprenda un modelo mundial del entorno utilizando un modelo generativo, independiente de cualquier tarea en particular. El agente puede adaptarse rápidamente a nuevas tareas probando estrategias en su propio modelo mundial, en lugar de en el entorno real, que a menudo es computacionalmente más eficiente y no requiere reentrenamiento desde cero para cada nueva tarea.

Finalmente, si realmente queremos decir que hemos construido una máquina que ha adquirido una forma de inteligencia comparable a la de un ser humano, el modelado generativo seguramente debe ser parte de la solución. Uno de los mejores ejemplos de modelo generativo en el mundo natural es la persona que lee este libro.

Tómate un momento para considerar el increíble modelo generativo que eres. Puedes cerrar los ojos e imaginar cómo sería un elefante desde cualquier ángulo posible. Puedes imaginar varios finales diferentes y posibles para tu programa de televisión favorito, y puedes planificar tu semana con anticipación trabajando en varios futuros en tu mente y tomando medidas en consecuencia. La teoría neurocientífica actual sugiere que nuestra percepción de la realidad no es un modelo discriminativo altamente complejo que opera sobre nuestra información sensorial para producir predicciones de lo que estamos experimentando, sino que es un modelo generativo que se entrena desde el nacimiento para producir simulaciones de nuestro entorno que encajan con precisión el futuro. Algunas teorías incluso sugieren que el resultado de este modelo generativo es lo que percibimos directamente como realidad. Claramente, una comprensión profunda de cómo podemos construir máquinas para adquirir esta capacidad será fundamental para nuestra comprensión continua del funcionamiento del cerebro y de la inteligencia artificial en general.

Nuestro primer modelo generativo

Con esto en mente, comenzemos nuestro viaje hacia el apasionante mundo del modelado generativo. Para empezar, veremos un ejemplo de juguete de un modelo generativo e introduciremos algunas de las ideas que nos ayudarán a trabajar en las arquitecturas más complejas que encontraremos más adelante en el libro.

¡Hola Mundo!

Comencemos jugando un juego de modelado generativo en sólo dos dimensiones. Elegí una regla que se utilizó para generar el conjunto de puntos X en la Figura 1-4. Llamemos a esta regla pdata. Tu desafío es elegir un punto diferente $x = (x_1, x_2)$ en el espacio que parece haber sido generado por la misma regla.

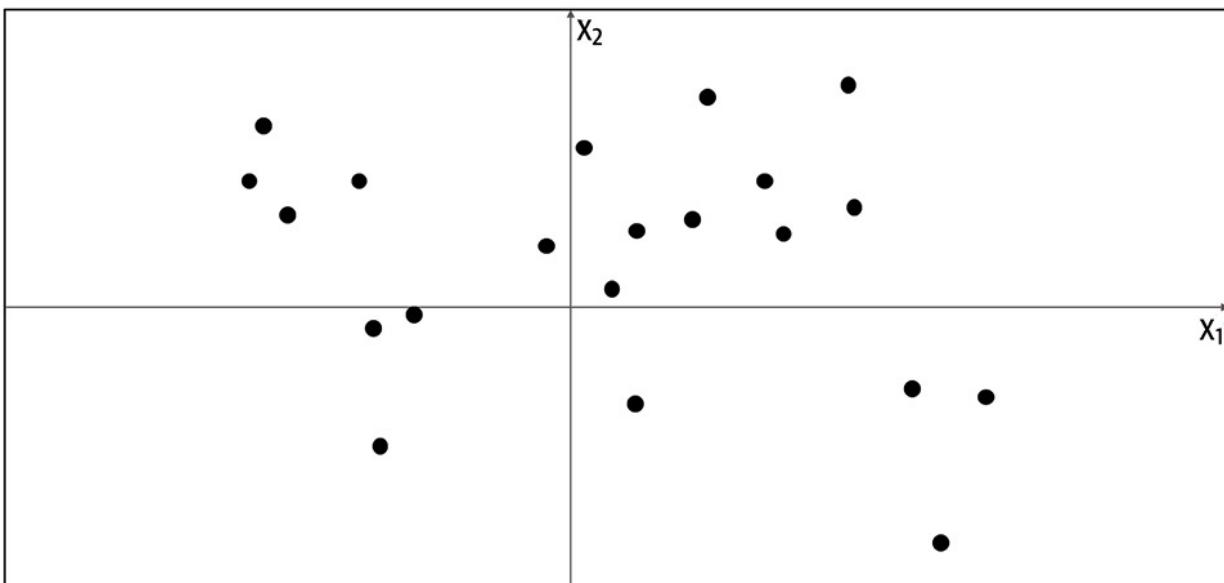


Figura 1-4. Un conjunto de puntos en dos dimensiones, generado por una regla desconocida pdata

¿Dónde elegiste? Probablemente hayas utilizado tu conocimiento de los puntos de datos existentes para construir un modelo mental, pmodel, de la ubicación en el espacio donde es más probable encontrar el punto. En este sentido, pmodel es una estimación de pdata.

Quizás decidiste que pmodel debería verse como la Figura 1-5: un cuadro rectangular donde se pueden encontrar puntos y un área fuera del cuadro

donde no hay posibilidad de encontrar ningún punto.

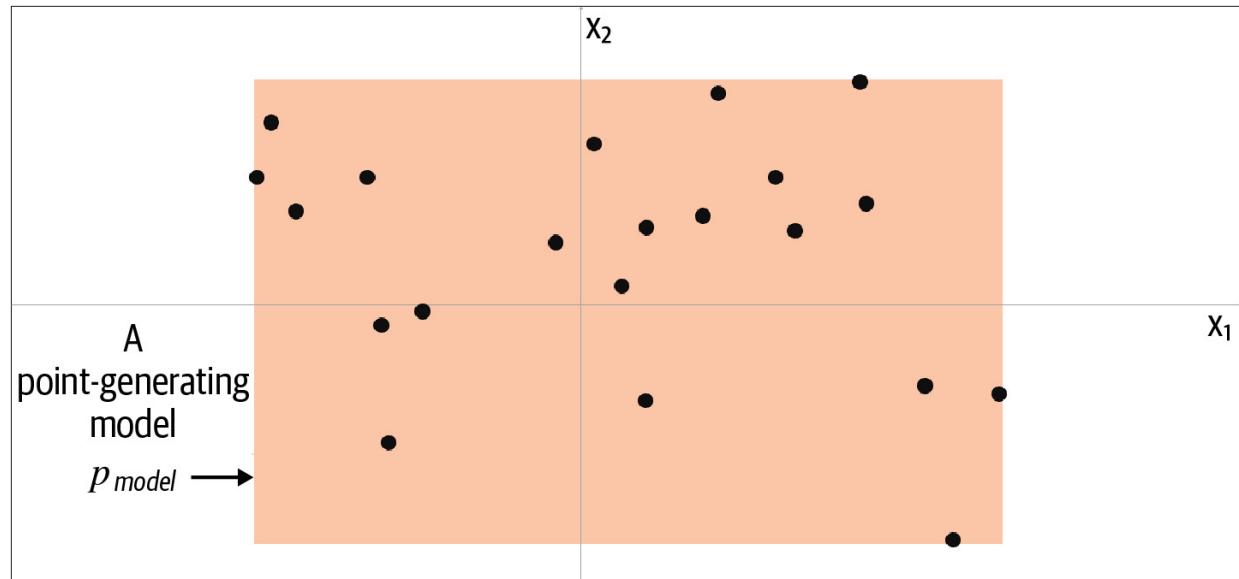


Figura 1-5. El cuadro naranja, p_{model} , es una estimación de la verdadera distribución generadora de datos, p_{data}

Para generar una nueva observación, simplemente puedes elegir un punto al azar dentro del cuadro o, más formalmente, una muestra de la distribución p_{model} . ¡Felicitaciones, acabas de construir tu primer modelo generativo! Has usado los datos de entrenamiento (los puntos negros) para construir un modelo (la región naranja) del que puedes tomar muestras fácilmente para generar otros puntos que parecen pertenecer al conjunto de entrenamiento.

Ahora formalicemos este pensamiento en un marco que pueda ayudarnos a comprender qué intenta lograr el modelado generativo.

El marco de modelado generativo

Podemos capturar nuestras motivaciones y objetivos para construir un modelo generativo en el siguiente marco.

EL MARCO DE MODELADO GENERATIVO

Tenemos un conjunto de datos de observaciones \mathbf{x} .

Suponemos que las observaciones se han generado según alguna distribución desconocida, pdata.

Queremos construir un modelo generativo pmodel que imite a pdata. Si logramos este objetivo, podemos tomar muestras de pmodel para generar observaciones que parecen haber sido extraídas de pdata.

Por tanto, las propiedades deseables de pmodel son:

Exactitud

Si pmodel es alto para una observación generada, debería parecer que se extrajo de pdata. Si pmodel es bajo para una observación generada, no debería parecer que se extrajo de pdata.

Generación

Debería ser posible muestrear fácilmente una nueva observación de pmodel.

Representación

Debería ser posible comprender cómo pmodel representa las diferentes características de alto nivel en los datos.

Revelemos ahora la verdadera distribución generadora de datos, pdata, y veamos cómo se aplica el marco a este ejemplo. Como podemos ver en la Figura 1-6, la regla de generación de datos es simplemente una distribución uniforme sobre la masa terrestre del mundo, sin posibilidad de encontrar un punto en el mar.

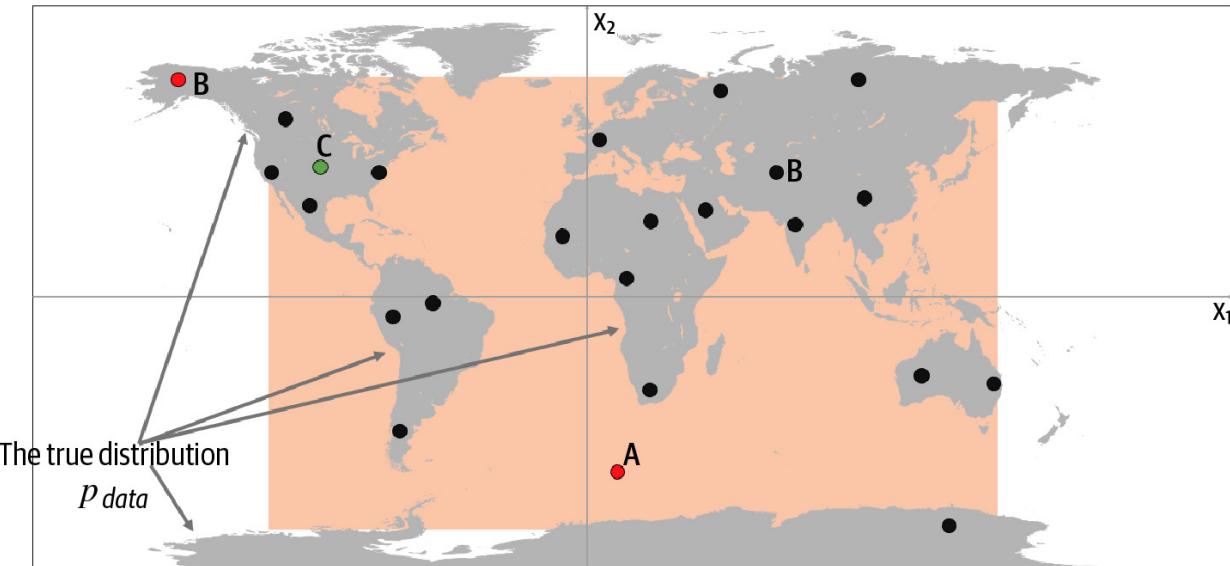


Figura 1-6. El cuadro naranja, p_{model} , es una estimación de la verdadera distribución generadora de datos, p_{data} (el área gris)

Claramente, nuestro modelo, p_{model} , es una simplificación excesiva de p_{data} .

Podemos inspeccionar los puntos A, B y C para comprender los éxitos y fracasos de nuestro modelo en términos de la precisión con la que imita a p_{data} :

El punto A es una observación generada por nuestro modelo pero no parece haber sido generada por p_{data} ya que está en medio del mar.

El punto B nunca podría haber sido generado por p_{model} ya que se encuentra fuera del cuadro naranja. Por lo tanto, nuestro modelo tiene algunas lagunas en su capacidad para producir observaciones en todo el rango de posibilidades potenciales.

El punto C es una observación que podría generarse mediante p_{model} y también mediante p_{data} .

A pesar de sus deficiencias, es fácil tomar muestras del modelo, porque es simplemente una distribución uniforme sobre el cuadro naranja. Podemos elegir fácilmente un punto al azar dentro de este cuadro para tomar una muestra de él.

Además, podemos decir con certeza que nuestro modelo es una representación simple de la distribución compleja subyacente que captura algunas de las características subyacentes de alto nivel. La verdadera distribución se separa en áreas con mucha masa terrestre (continentes) y aquellas sin masa terrestre (el mar).

Esta es una característica de alto nivel que también se aplica a nuestro modelo, excepto que tenemos un continente grande, en lugar de muchos.

Este ejemplo ha demostrado los conceptos fundamentales detrás del modelado generativo. Los problemas que abordaremos en este libro serán mucho más complejos y de mayor dimensión, pero el marco subyacente a través del cual abordaremos el problema será el mismo.

Aprendizaje de representación

Vale la pena profundizar un poco más en lo que queremos decir con aprender una representación de datos de alta dimensión, ya que es un tema que se repetirá a lo largo de este libro.

Supongamos que quisieras describir tu apariencia a alguien que te está buscando entre una multitud y no sabe cómo te ves. No comenzarías indicando, en una foto tuya, el color del píxel 1, luego el píxel 2, luego el píxel 3, etc. En cambio, harías la suposición razonable de que la otra persona tiene una idea general de cómo se ve un ser humano promedio. Luego modifique esta línea de base con características que describan grupos de píxeles, como Tengo el pelo muy rubio o uso gafas. Con no más de 10 de estas declaraciones, la persona podría mapear la descripción nuevamente en píxeles para generar una imagen tuya en su cabeza. La imagen no sería perfecta, pero sería lo suficientemente parecida a tu apariencia real como para que te encuentren posiblemente entre cientos de otras personas, incluso si nunca te han visto antes.

Esta es la idea central detrás del aprendizaje por representación.

En lugar de intentar modelar el espacio muestral de alta dimensión directamente, describimos cada observación en el conjunto de entrenamiento usando algún espacio latente de menor dimensión y luego aprendemos una

función de mapeo que puede tomar un punto en el espacio latente y mapearlo a un punto en el dominio original. En otras palabras, cada punto del espacio latente es una representación de alguna observación de alta dimensión.

¿Qué significa esto en la práctica? Supongamos que tenemos un conjunto de entrenamiento que consta de imágenes en escala de grises de latas de galletas (Figura 1-7).

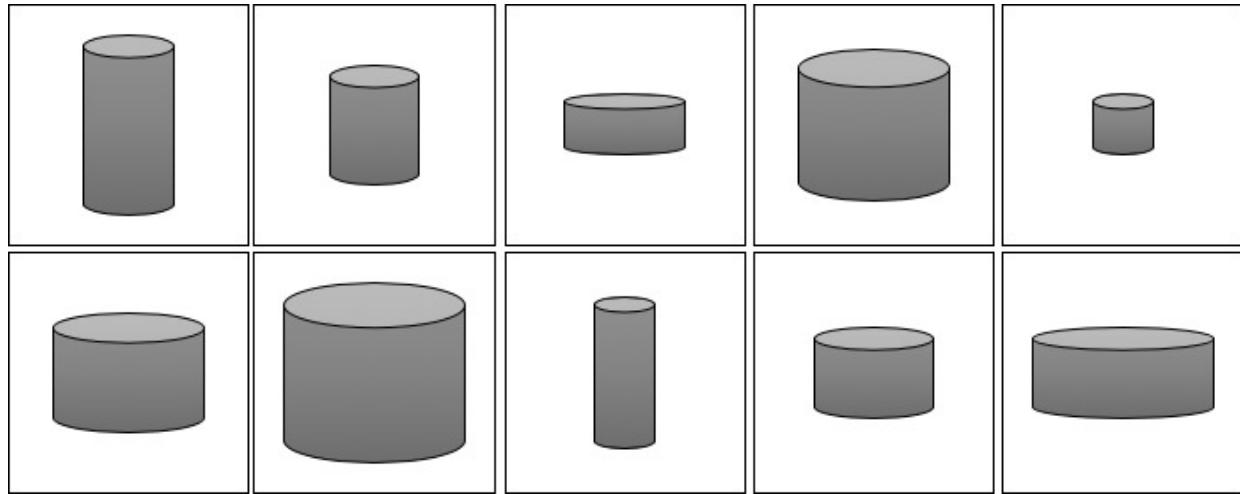


Figura 1-7. El conjunto de datos de lata de galletas

Para nosotros, es obvio que hay dos características que pueden representar de forma única cada una de estas latas: la altura y el ancho de la lata. Es decir, podemos convertir cada imagen de una lata en un punto en un espacio latente de solo dos dimensiones, aunque el conjunto de imágenes de entrenamiento se proporcione en un espacio de píxeles de alta dimensión. En particular, esto significa que también podemos producir imágenes de tins que no existen en el conjunto de entrenamiento, aplicando una función de mapeo adecuada f a un nuevo punto en el espacio latente, como se muestra en la Figura 1-8.

Darse cuenta de que el conjunto de datos original puede describirse mediante el espacio latente más simple no es tan fácil para una máquina: primero necesitaría establecer que la altura y el ancho son las dos dimensiones del espacio latente que mejor describen este conjunto de datos, luego aprendería

la función de mapeo f que Puede tomar un punto en este espacio y asignarlo a una imagen de lata de galletas en escala de grises.

El aprendizaje automático (y específicamente el aprendizaje profundo) nos brinda la capacidad de entrenar máquinas que puedan encontrar estas relaciones complejas sin guía humana.

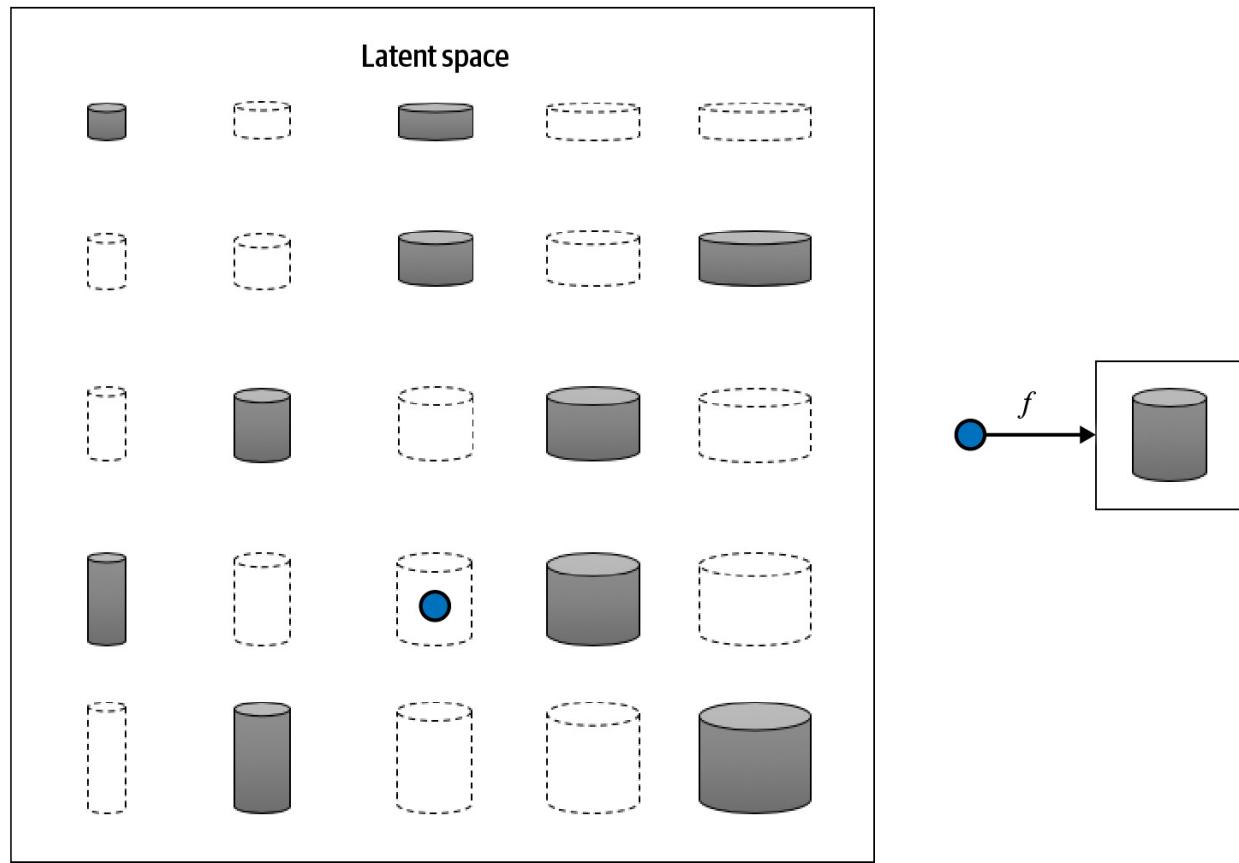


Figura 1-8. El espacio latente 2D de latas de galletas y la función f que mapea un punto en el espacio latente al dominio de la imagen original.

Uno de los beneficios de entrenar modelos que utilizan un espacio latente es que podemos realizar operaciones que afectan las propiedades de alto nivel de la imagen manipulando su vector de representación dentro del espacio latente más manejable. Por ejemplo, no es obvio cómo ajustar el sombreado de cada píxel para hacer que la imagen de una lata de galletas sea más alta. Sin embargo, en el espacio latente, se trata simplemente de aumentar la dimensión latente de altura y luego aplicar la función de mapeo para regresar

al dominio de la imagen. Veremos un ejemplo explícito de esto en el próximo capítulo, aplicado no a latas de galletas sino a caras.

El concepto de codificar el conjunto de datos de entrenamiento en un espacio latente para que podamos tomar muestras de él y decodificar el punto al dominio original es común a muchas técnicas de modelado generativo, como veremos en capítulos posteriores de este libro. Matemáticamente hablando, las técnicas de codificador-decodificador intentan transformar la variedad altamente no lineal en la que se encuentran los datos (por ejemplo, en el espacio de píxeles) en un espacio latente más simple del que se puede tomar muestras, de modo que es probable que cualquier punto en el espacio latente sea la representación de una imagen bien formada, como se muestra en la Figura 1-9.

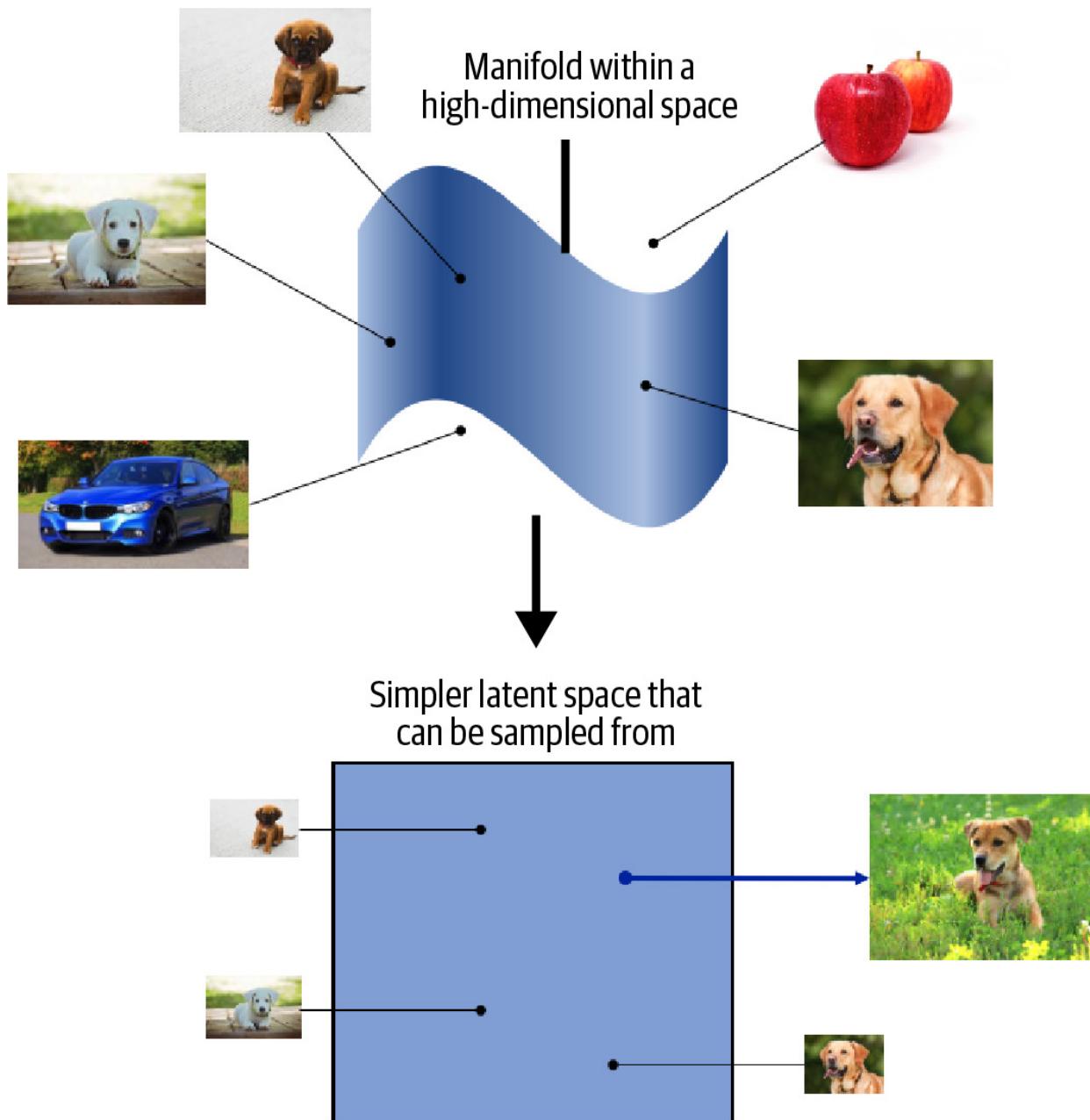


Figura 1-9. La variedad de perros en el espacio de píxeles de alta dimensión se asigna a un espacio latente más simple que se puede muestrear desde la Teoría central de la probabilidad

Ya hemos visto que el modelado generativo está estrechamente relacionado con el modelado estadístico de distribuciones de probabilidad. Por lo tanto, ahora tiene sentido introducir algunos conceptos probabilísticos y estadísticos básicos que se utilizarán a lo largo de este libro para explicar los antecedentes teóricos de cada modelo.

Si nunca has estudiado probabilidad o estadística, no te preocupes. Para construir muchos de los modelos de aprendizaje profundo que veremos más adelante en este libro, no es esencial tener una comprensión profunda de la teoría estadística. Sin embargo, para obtener una apreciación completa de la tarea que intentamos abordar, vale la pena intentar desarrollar una comprensión sólida de la teoría probabilística básica. De esta manera, tendrá las bases para comprender las diferentes familias de modelos generativos que se presentarán más adelante en este capítulo.

Como primer paso, definiremos cinco términos clave, vinculando cada uno de ellos con nuestro ejemplo anterior de un modelo generativo que modela el mapa mundial en dos dimensiones:

Espacio muestral

El espacio muestral es el conjunto completo de todos los valores que puede tomar una observación x .

NOTA

En nuestro ejemplo anterior, el espacio muestral consta de todos los puntos de latitud y longitud $x = (x_1, x_2)$ en el mapa mundial. Por ejemplo, $x = (40,7306, -73,9352)$ es un punto en el espacio muestral (Ciudad de Nueva York) que pertenece a la verdadera distribución generadora de datos. $x = (11.3493, 142.1996)$ es un punto en el espacio muestral que no pertenece a la verdadera distribución generadora de datos (está en el mar).

Función de densidad de probabilidad

Una función de densidad de probabilidad (o simplemente función de densidad) es una función $p(x)$ que asigna un punto x en el espacio muestral a un número entre 0 y 1. La integral de la función de densidad sobre todos los puntos en el espacio muestral debe ser igual a 1. , por lo que es una distribución de probabilidad bien definida.

NOTA

En el ejemplo del mapa mundial, la función de densidad de nuestro modelo generativo es 0 fuera del cuadro naranja y constante dentro del cuadro, de modo que la integral de la función de densidad en todo el espacio muestral es igual a 1.

Si bien solo hay una función de densidad verdadera $p_{\text{data}}(x)$ que se supone que generó el conjunto de datos observables, hay infinitas funciones de densidad $p_{\text{model}}(x)$ que podemos usar para estimar $p_{\text{data}}(x)$.

Modelado paramétrico

El modelado paramétrico es una técnica que podemos utilizar para estructurar nuestro enfoque para encontrar un modelo p adecuado (x). Un modelo paramétrico es una familia de funciones de densidad $p_\theta(x)$ que se puede describir utilizando un número finito de parámetros, θ .

NOTA

Si asumimos una distribución uniforme como nuestra familia de modelos, entonces el conjunto de todos los cuadros posibles que podríamos dibujar en la Figura 1-5 es un ejemplo de modelo paramétrico. En este caso, hay cuatro parámetros: las coordenadas de las esquinas inferior izquierda (θ_1, θ_2) y superior derecha (θ_3, θ_4) del cuadro.

Por lo tanto, cada función de densidad $p_\theta(x)$ en este modelo paramétrico (es decir, cada cuadro) puede representarse de forma única mediante cuatro números, $\theta = (\theta_1, \theta_2, \theta_3, \theta_4)$.

Probabilidad

La probabilidad $L(\theta|x)$ de un conjunto de parámetros θ es una función que mide la plausibilidad de θ , dado algún punto x observado. Se define de la siguiente manera:

$$L(\theta|x) = p_\theta(x)$$

Es decir, la probabilidad de θ dado algún punto x observado se define como el valor de la función de densidad parametrizada por θ , en el punto x . Si

tenemos un conjunto de datos completo X de observaciones independientes, entonces podemos escribir:

$$L(\theta|X) = \prod p_\theta(x)_{x \in X}$$

NOTA

En el ejemplo del mapa mundial, un cuadro naranja que solo cubriera la mitad izquierda del mapa tendría una probabilidad de 0; no podría haber generado el conjunto de datos, ya que hemos observado puntos en la mitad derecha del mapa. El cuadro naranja en la Figura 1-5 tiene una probabilidad positiva, ya que la función de densidad es positiva para todos los puntos de datos de este modelo.

Dado que puede ser bastante difícil trabajar con el producto de una gran cantidad de términos entre 0 y 1, a menudo usamos la probabilidad logarítmica ℓ en su lugar: $\ell(\theta|X) = \sum \log p_\theta(x)_{x \in X}$

Hay razones estadísticas por las que la probabilidad se define de esta manera, pero también podemos ver que esta definición intuitivamente tiene sentido. La probabilidad de un conjunto de parámetros θ se define como la probabilidad de ver los datos si la verdadera distribución generadora de datos fuera el modelo parametrizado por θ .

ADVERTENCIA

Tenga en cuenta que la probabilidad es función de los parámetros, no de los datos. No debe interpretarse como la probabilidad de que un determinado conjunto de parámetros sea correcto; en otras palabras, no es una distribución de probabilidad sobre el espacio de parámetros (es decir, no suma/integra a 1, con respecto a los parámetros).

Tiene sentido intuitivo que el enfoque del modelado paramétrico sea encontrar el valor óptimo θ del conjunto de parámetros que maximice la probabilidad de observar el conjunto de datos X .

Estimación de máxima verosimilitud

La estimación de máxima verosimilitud es la técnica que nos permite estimar θ —el conjunto de parámetros θ de una función de densidad $p_\theta(x)$ que tiene más probabilidades de explicar algunos datos observados X . Más formalmente:

$$\hat{\theta} = \operatorname{argmax}_{\theta} \ell(\theta|X)$$

También se llama estimación de máxima verosimilitud (MLE).

NOTA

En el ejemplo del mapa mundial, el MLE es el rectángulo más pequeño que aún contiene todos los puntos del conjunto de entrenamiento.

Las redes neuronales normalmente minimizan una función de pérdida, por lo que podemos hablar de manera equivalente sobre encontrar el conjunto de parámetros que minimizan la probabilidad logarítmica negativa: $\hat{\theta} = \operatorname{argmin}(-\ell(\theta|X)) = \operatorname{argmin}(-\log p_\theta(X))$

El modelado generativo puede considerarse como una forma de estimación de máxima verosimilitud, donde los parámetros θ son los pesos de las redes neuronales contenidas en el modelo. Estamos tratando de encontrar los valores de estos parámetros que maximicen la probabilidad de observar los datos dados (o, de manera equivalente, minimicen la probabilidad logarítmica negativa).

Sin embargo, para problemas de alta dimensión, generalmente no es posible calcular directamente $p_\theta(x)$: es intratable. Como veremos en la siguiente sección, diferentes familias de modelos generativos adoptan diferentes enfoques para abordar este problema.

Taxonomía del modelo generativo

Si bien todos los tipos de modelos generativos apuntan en última instancia a resolver la misma tarea, todos adoptan enfoques ligeramente diferentes para modelar la función de densidad $p_\theta(x)$. En términos generales, existen tres enfoques posibles:

1. Modelar explícitamente la función de densidad, pero restringir el modelo de alguna manera, de modo que la función de densidad sea manejable (es decir, que se pueda calcular).
1. Modelar explícitamente una aproximación manejable de la función de densidad.
1. Modelar implícitamente la función de densidad, mediante un proceso estocástico que genera datos directamente.

Estos se muestran en la Figura 1-10 como una taxonomía, junto con las seis familias de modelos generativos que exploraremos en la Parte II de este libro. Tenga en cuenta que estas familias no son mutuamente excluyentes: hay muchos ejemplos de modelos que son híbridos entre dos tipos diferentes de enfoques.

Debería pensar en las familias como enfoques generales diferentes del modelado generativo, en lugar de arquitecturas de modelos explícitas.

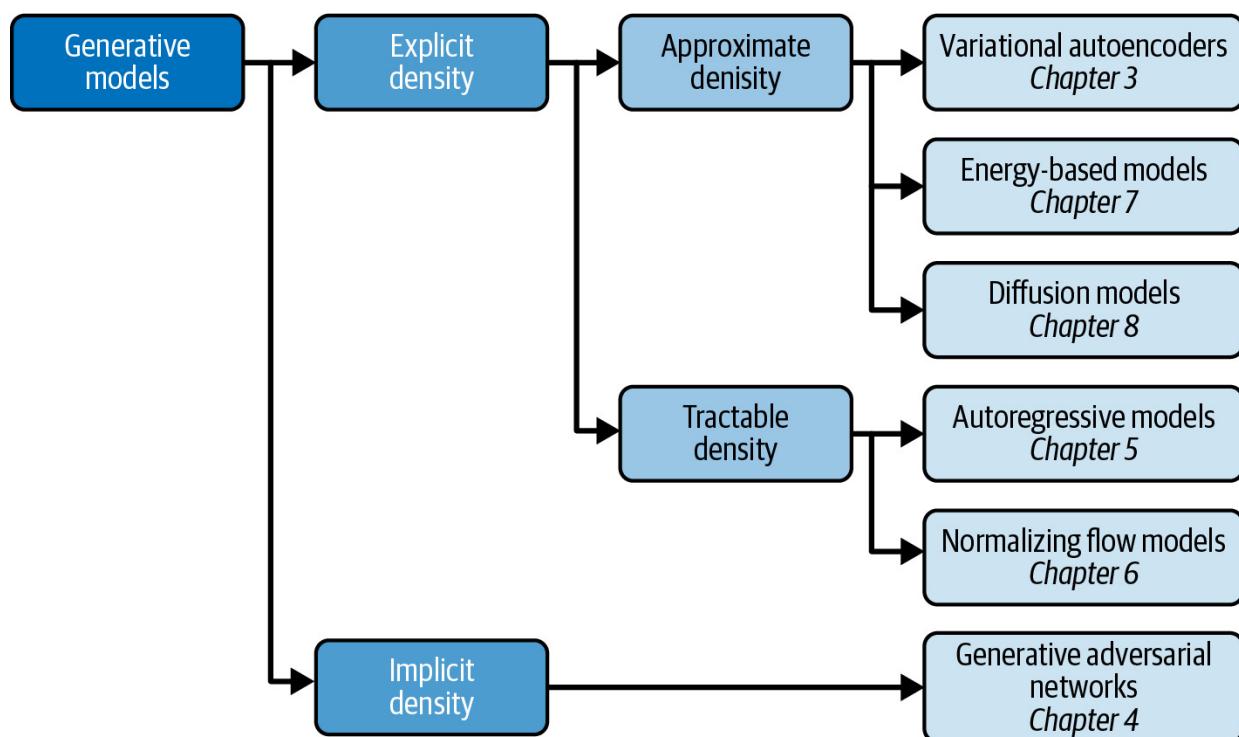


Figura 1-10. Una taxonomía de enfoques de modelado generativo.

La primera división que podemos hacer es entre modelos en los que la función de densidad de probabilidad $p(x)$ se modela explícitamente y aquellos en los que se modela implícitamente.

Los modelos de densidad implícita no pretenden estimar la densidad de probabilidad en absoluto, sino que se centran únicamente en producir un proceso estocástico que genera datos directamente.

El ejemplo más conocido de modelo generativo implícito es una red generativa adversaria. Podemos dividir aún más los modelos de densidad explícitos en aquellos que optimizan directamente la función de densidad (modelos manejables) y aquellos que solo optimizan una aproximación de la misma.

Los modelos manejables imponen restricciones a la arquitectura del modelo, de modo que la función de densidad tenga una forma que facilite su cálculo. Por ejemplo, los modelos autorregresivos imponen un orden en las características de entrada, de modo que la salida se pueda generar secuencialmente, por ejemplo, palabra por palabra o píxel por píxel. Los modelos de flujo de normalización aplican una serie de funciones manejables e invertibles a una distribución simple, para generar distribuciones más complejas.

Los modelos de densidad aproximada incluyen codificadores automáticos variacionales, que introducen una variable latente y optimizan una aproximación de la función de densidad conjunta.

Los modelos basados en energía también utilizan métodos aproximados, pero lo hacen mediante muestreo en cadena de Markov, en lugar de métodos variacionales. Los modelos de difusión se aproximan a la función de densidad entrenando un modelo para eliminar gradualmente el ruido de una imagen determinada que ha sido previamente corrompida.

Un hilo común que recorre todos los tipos de familias de modelos generativos es el aprendizaje profundo. Casi todos los modelos generativos sofisticados tienen una red neuronal profunda en su núcleo, porque pueden entrenarse desde cero para aprender las relaciones complejas que gobiernan la estructura de los datos, en lugar de tener que codificarlos con información

a priori. Exploraremos el aprendizaje profundo en el Capítulo 2, con ejemplos prácticos de cómo empezar a construir sus propias redes neuronales profundas.

La base de código de aprendizaje profundo generativo

La sección final de este capítulo lo preparará para comenzar a construir modelos generativos de aprendizaje profundo mediante la introducción del código base que acompaña a este libro.

CONSEJO

Muchos de los ejemplos de este libro están adaptados de las excelentes implementaciones de código abierto que están disponibles en el sitio web de Keras. Le recomiendo encarecidamente que consulte este recurso, ya que constantemente se agregan nuevos modelos y ejemplos.

Clonando el repositorio

Para comenzar, primero deberá clonar el repositorio de Git.

Git es un sistema de control de versiones de código abierto y le permitirá copiar el código localmente para que pueda ejecutar los cuadernos en su propia máquina o en un entorno basado en la nube. Es posible que ya lo tengas instalado, pero si no, sigue las instrucciones correspondientes a tu sistema operativo.

Para clonar el repositorio de este libro, navegue hasta la carpeta donde desea almacenar los archivos y escriba lo siguiente en su terminal:

```
git clone  
https://github.com/davidADSP/Generative_Deep_Learning_2nd_Edition.git
```

Ahora debería poder ver los archivos en una carpeta en su máquina.

Usando Docker

El código base de este libro está pensado para ser utilizado con Docker, una tecnología de contenedорización gratuita que hace que comenzar con una nueva base de código sea extremadamente fácil, independientemente de su arquitectura o sistema operativo. Si nunca ha usado Docker, no se preocupe: hay una descripción de cómo comenzar en el archivo README en el repositorio de libros.

Ejecutando en una GPU

Si no tienes acceso a tu propia GPU, ¡tampoco hay problema! Todos los ejemplos de este libro se entrenarán en una CPU, aunque esto llevará más tiempo que si usa una máquina habilitada para GPU. También hay una sección en el archivo README sobre la configuración de un entorno de Google Cloud que le brinda acceso a una GPU mediante pago por uso.

Resumen

Este capítulo presentó el campo del modelado generativo, una rama importante del aprendizaje automático que complementa el modelado discriminativo más ampliamente estudiado. Discutimos cómo el modelado generativo es actualmente una de las áreas más activas y apasionantes de la investigación en IA, con muchos avances recientes tanto en teoría como en aplicaciones.

Comenzamos con un ejemplo de juguete simple y vimos cómo el modelado generativo finalmente se enfoca en modelar la distribución subyacente de los datos. Esto presenta muchos desafíos complejos e interesantes, que resumimos en un marco para comprender las propiedades deseables de cualquier modelo generativo.

Luego analizamos los conceptos probabilísticos clave que ayudarán a comprender completamente los fundamentos teóricos de cada enfoque del modelado generativo y presentamos las seis familias diferentes de modelos generativos que exploraremos en la Parte II de este libro. También vimos

cómo comenzar con el código base de Aprendizaje Profundo Generativo, clonando el repositorio.

En el Capítulo 2, comenzaremos nuestra exploración del aprendizaje profundo y veremos cómo usar Keras para construir modelos que puedan realizar tareas de modelado discriminativas. Esto nos dará la base necesaria para abordar los problemas generativos de aprendizaje profundo en capítulos posteriores.

1 Miles Brundage et al., “El uso malicioso de la inteligencia artificial: Previsión, prevención y mitigación”, 20 de febrero de 2018,
https://www.eff.org/files/2018/02/20/malicious_ai_report_final.pdf.

Capítulo 2. Aprendizaje profundo

METAS DEL CAPÍTULO

En este capítulo podrá:

- Conocer los diferentes tipos de datos no estructurados que se pueden modelar mediante el aprendizaje profundo.
- Definir una red neuronal profunda y comprenda cómo se puede utilizar para modelar conjuntos de datos complejos.
- Construir un perceptrón multicapa para predecir el contenido de una imagen.
- Mejorar el rendimiento del modelo mediante el uso de capas convolucionales, de abandono y de normalización por lotes.

Comencemos con una definición básica de aprendizaje profundo:

El aprendizaje profundo es una clase de algoritmos de aprendizaje automático que utiliza múltiples capas apiladas de unidades de procesamiento para aprender representaciones de alto nivel a partir de datos no estructurados.

Para comprender completamente el aprendizaje profundo, debemos profundizar un poco más en esta definición. Primero, veremos los diferentes tipos de datos no estructurados que se pueden usar para modelar el aprendizaje profundo, luego nos sumergiremos en la mecánica de construir múltiples capas apiladas de unidades de procesamiento para resolver tareas de clasificación. Esto proporcionará la base para capítulos futuros en los que nos centraremos en el aprendizaje profundo para tareas generativas.

Datos para el aprendizaje profundo

Muchos tipos de algoritmos de aprendizaje automático requieren datos tabulares estructurados como entrada, organizados en columnas de características que describen cada observación. Por ejemplo, la edad de una persona, los ingresos y el número de visitas al sitio web de una persona en el último mes son características que podrían ayudar a predecir si la persona se suscribirá a un servicio en línea en particular durante el próximo mes. Podríamos usar una tabla estructurada de estas características para entrenar una regresión logística, un bosque aleatorio o el modelo XGBoost para predecir la variable de respuesta binaria: ¿la persona se suscribió (1) o no (0)? Aquí, cada característica individual contiene una pequeña cantidad de información sobre la

observación, y el modelo aprendería cómo estas características interactúan para influir en la respuesta.

Los datos no estructurados se refieren a cualquier dato que no esté organizado de forma natural en columnas de características, como imágenes, audio y texto. Por supuesto, existe una estructura espacial en una imagen, una estructura temporal en una grabación o un pasaje de texto, y una estructura espacial y temporal en los datos de vídeo, pero como los datos no llegan en columnas de características, se consideran no estructurados, como se muestra en Figura 2-1.

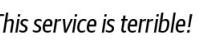
Structured data

ID	Age	Gender	Height (cm)	Location
0001	54	M	186	London
0002	35	F	166	New York
0003	62	F	170	Amsterdam
0004	23	M	164	London
0005	25	M	180	Cairo
0006	29	F	181	Beijing
0007	46	M	172	Chicago

Unstructured data


Images


Audio


Text

This service is terrible!

Your website is great!

Figura 2-1. La diferencia entre datos estructurados y no estructurados.

Cuando nuestros datos no están estructurados, los píxeles, frecuencias o caracteres individuales carecen casi por completo de información. Por ejemplo, saber que el píxel 234 de una imagen es de un tono marrón turbio no ayuda realmente a identificar si la imagen es de una casa o de un perro, y saber que el carácter 24 de una oración es una *e* no ayuda a predecir si el texto trata sobre fútbol o política.

Los píxeles o caracteres son en realidad sólo los hoyuelos del lienzo en los que se incrustan elementos informativos de nivel superior, como la imagen de una chimenea o la palabra *huelguista*.

Si la chimenea de la imagen se colocara en el otro lado de la casa, la imagen todavía contendría una chimenea, pero esta información ahora sería transportada por píxeles completamente diferentes. Si la palabra *delantero* apareciera un poco antes o después en el texto, el texto seguiría siendo sobre fútbol, pero diferentes posiciones de los caracteres proporcionarían esta información. La granularidad de los datos, combinada con el alto grado de dependencia espacial, destruye el concepto de píxel o carácter como característica informativa por derecho propio.

Por esta razón, si entrenamos modelos de regresión logística, bosque aleatorio o XGBoost con valores de píxeles sin procesar, el modelo entrenado a menudo tendrá un

rendimiento deficiente para todas las tareas de clasificación, excepto para las más simples. Estos modelos se basan en que las características de entrada sean informativas y no espacialmente dependientes. Por otro lado, un modelo de aprendizaje profundo puede aprender a crear características informativas de alto nivel por sí mismo, directamente a partir de datos no estructurados.

El aprendizaje profundo se puede aplicar a datos estructurados, pero su verdadero poder, especialmente con respecto al modelado generativo, proviene de su capacidad para trabajar con datos no estructurados. La mayoría de las veces queremos generar datos no estructurados, como nuevas imágenes o cadenas de texto originales, razón por la cual el aprendizaje profundo ha tenido un impacto tan profundo en el campo del modelado generativo.

Redes neuronales profundas

La mayoría de los sistemas de aprendizaje profundo son redes neuronales artificiales (ANN, o simplemente redes neuronales para abreviar) con múltiples capas ocultas apiladas. Por esta razón, el aprendizaje profundo se ha convertido casi en sinónimo de redes neuronales profundas. Sin embargo, cualquier sistema que emplee muchas capas para aprender representaciones de alto nivel de los datos de entrada también es una forma de aprendizaje profundo (por ejemplo, redes de creencias profundas).

Comencemos por desglosar exactamente qué entendemos por red neuronal y luego veamos cómo se pueden usar para aprender características de alto nivel a partir de datos no estructurados.

¿Qué es una red neuronal?

Una red neuronal consta de una serie de capas apiladas. Cada capa contiene unidades que están conectadas a las unidades de la capa anterior a través de un conjunto de pesos. Como veremos, hay muchos tipos diferentes de capas, pero una de las más comunes es la capa completamente conectada (o densa) que conecta todas las unidades de la capa directamente con cada unidad de la capa anterior.

Las redes neuronales donde todas las capas adyacentes están completamente conectadas se denominan perceptrones multicapa (MLP). Este es el primer tipo de red neuronal que estudiaremos.

En la Figura 2-2 se muestra un ejemplo de MLP.

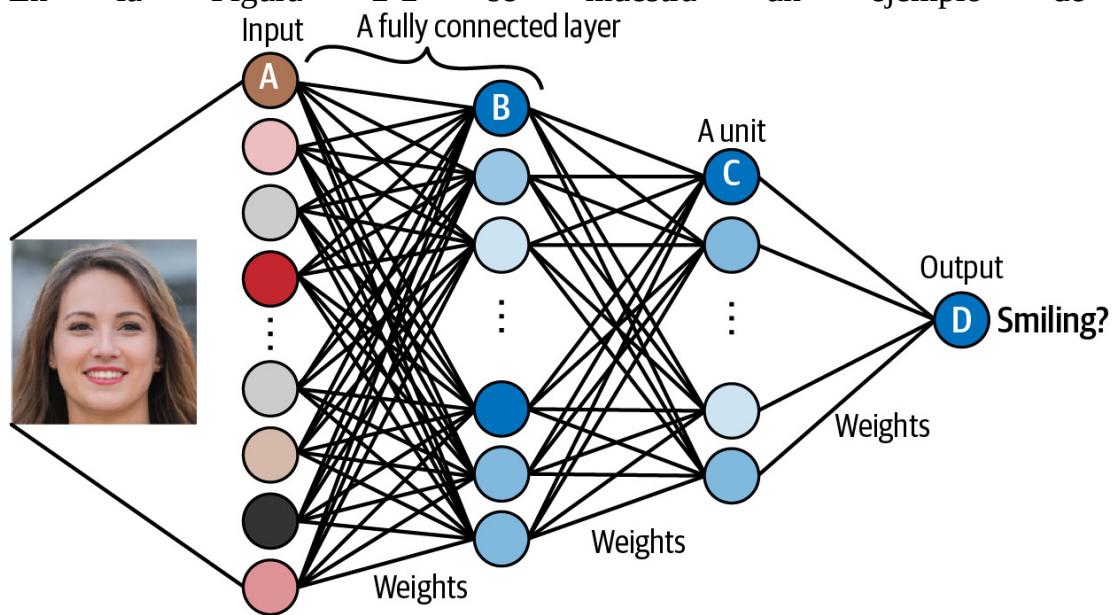


Figura 2-2. Un ejemplo de perceptrón multicapa que predice si una cara está sonriendo

La entrada (por ejemplo, una imagen) es transformada por cada capa a su vez, en lo que se conoce como un paso directo a través de la red, hasta llegar a la capa de salida. Específicamente, cada unidad aplica una transformación no lineal a una suma ponderada de sus entradas y pasa la salida a la capa siguiente. La capa de salida final es la culminación de este proceso, donde la unidad individual genera una probabilidad de que la entrada original pertenezca a una categoría particular (por ejemplo, sonreír).

La magia de las redes neuronales profundas radica en encontrar el conjunto de pesos para cada capa que dé como resultado las predicciones más precisas. El proceso de encontrar estos pesos es lo que entendemos por entrenar la red.

Durante el proceso de entrenamiento, se pasan lotes de imágenes a través de la red y los resultados previstos se comparan con la verdad del terreno. Por ejemplo, la red podría generar una probabilidad del 80 % para una imagen de alguien que realmente está sonriendo y una probabilidad del 23 % para una imagen de alguien que realmente no está sonriendo. Una predicción perfecta arrojaría 100% y 0% para estos ejemplos, por lo que hay una pequeña cantidad de error. Luego, el error en la predicción se propaga hacia atrás a través de la red, ajustando cada conjunto de pesos una pequeña cantidad en la dirección que mejora la predicción de manera más significativa. Este proceso se llama apropiadamente retropropagación. Gradualmente, cada unidad adquiere habilidades para identificar una característica particular que, en última instancia, ayuda a la red a hacer mejores predicciones.

Aprendizaje de características de alto nivel

La propiedad fundamental que hace que las redes neuronales sean tan poderosas es su capacidad para aprender características a partir de los datos de entrada, sin guía humana. En otras palabras, no necesitamos realizar ninguna ingeniería de características, ¡por eso las redes neuronales son tan útiles! Podemos dejar que el modelo decida cómo quiere organizar sus pesos, guiado únicamente por su deseo de minimizar el error en sus predicciones.

Por ejemplo, repasemos la red que se muestra en la figura 2-2, suponiendo que ya ha sido entrenado para predecir con precisión si una cara de entrada determinada está sonriendo:

1. La unidad A recibe el valor de un canal individual de un píxel de entrada.
2. La unidad B combina sus valores de entrada para que dispare con mayor potencia cuando está presente una característica particular de bajo nivel, como un borde.
3. La unidad C combina las características de bajo nivel para que se dispare más fuerte cuando se ve una característica de nivel superior, como dientes, en la imagen.
4. La Unidad D combina las funciones de alto nivel para que se dispare con mayor potencia cuando la persona de la imagen original está sonriendo.

Las unidades en cada capa posterior pueden representar aspectos cada vez más sofisticados de la entrada original, combinando características de nivel inferior de la capa anterior.

Sorprendentemente, esto surge naturalmente del proceso de capacitación: no necesitamos decirle a cada unidad qué buscar, o si debe buscar características de alto nivel o características de bajo nivel.

Las capas entre las capas de entrada y salida se denominan capas ocultas. Si bien nuestro ejemplo solo tiene dos capas ocultas, las redes neuronales profundas pueden tener muchas más.

Apilar una gran cantidad de capas permite que la red neuronal aprenda progresivamente características de nivel superior al acumular gradualmente información a partir de características de nivel inferior en capas anteriores. Por ejemplo, ResNet1, diseñado para el reconocimiento de imágenes, contiene 152 capas.

A continuación, nos sumergiremos directamente en el lado práctico del aprendizaje profundo y nos configuraremos con TensorFlow y Keras para que pueda comenzar a construir sus propias redes neuronales profundas.

TensorFlow y Keras

TensorFlow es una biblioteca Python de código abierto para aprendizaje automático, desarrollada por Google. TensorFlow es uno de los marcos más utilizados para crear soluciones de aprendizaje automático, con especial énfasis en la manipulación de tensores (de ahí el nombre). Proporciona la funcionalidad de bajo nivel necesaria para entrenar redes neuronales, como calcular el gradiente de expresiones diferenciables arbitrarias y ejecutar de manera eficiente operaciones tensoriales.

Keras es una API de alto nivel para construir redes neuronales, construida sobre TensorFlow (Figura 2-3). Es extremadamente flexible y muy fácil de usar, lo que lo convierte en una opción ideal para comenzar con el aprendizaje profundo. Además, Keras proporciona numerosos componentes básicos útiles que se pueden conectar para crear arquitecturas de aprendizaje profundo altamente complejas a través de su API funcional.



Figura 2-3. TensorFlow y Keras son excelentes herramientas para crear soluciones de aprendizaje profundo

Si recién está comenzando con el aprendizaje profundo, le recomiendo encarecidamente que utilice TensorFlow y Keras. Esta configuración le permitirá construir cualquier red que pueda imaginar en un entorno de producción, al mismo tiempo que le brindará una API fácil de aprender que permite el desarrollo rápido de nuevas ideas y conceptos. Comencemos viendo lo fácil que es construir un perceptrón multicapa usando Keras.

Perceptrón multicapa (MLP)

En esta sección, entrenaremos un MLP para clasificar una imagen determinada mediante el aprendizaje supervisado. El aprendizaje supervisado es un tipo de algoritmo de aprendizaje automático en el que la computadora se entrena con un conjunto de datos etiquetados. En otras palabras, el conjunto de datos utilizado para el entrenamiento incluye datos de entrada con las etiquetas de salida correspondientes. El objetivo del algoritmo es aprender un mapeo entre los datos de entrada y las etiquetas de salida, de modo que pueda hacer predicciones sobre datos nuevos e invisibles.

El MLP es un modelo discriminativo (más que generativo), pero el aprendizaje supervisado seguirá desempeñando un papel en muchos tipos de modelos generativos que exploraremos en capítulos posteriores de este libro, por lo que es un buen lugar para comenzar nuestro viaje.

EJECUTANDO EL CÓDIGO PARA ESTE EJEMPLO

El código de este ejemplo se puede encontrar en el cuaderno de Jupyter ubicado en notebooks/02_deeplearning/01_mlp/mlp.ipynb en el repositorio de libros.

Preparando los datos

Para este ejemplo, utilizaremos el conjunto de datos CIFAR-10, una colección de 60 000 imágenes en color de 32×32 píxeles que viene incluido con Keras de fábrica. Cada imagen se clasifica exactamente en una de 10 clases, como se muestra en la Figura 2-4.

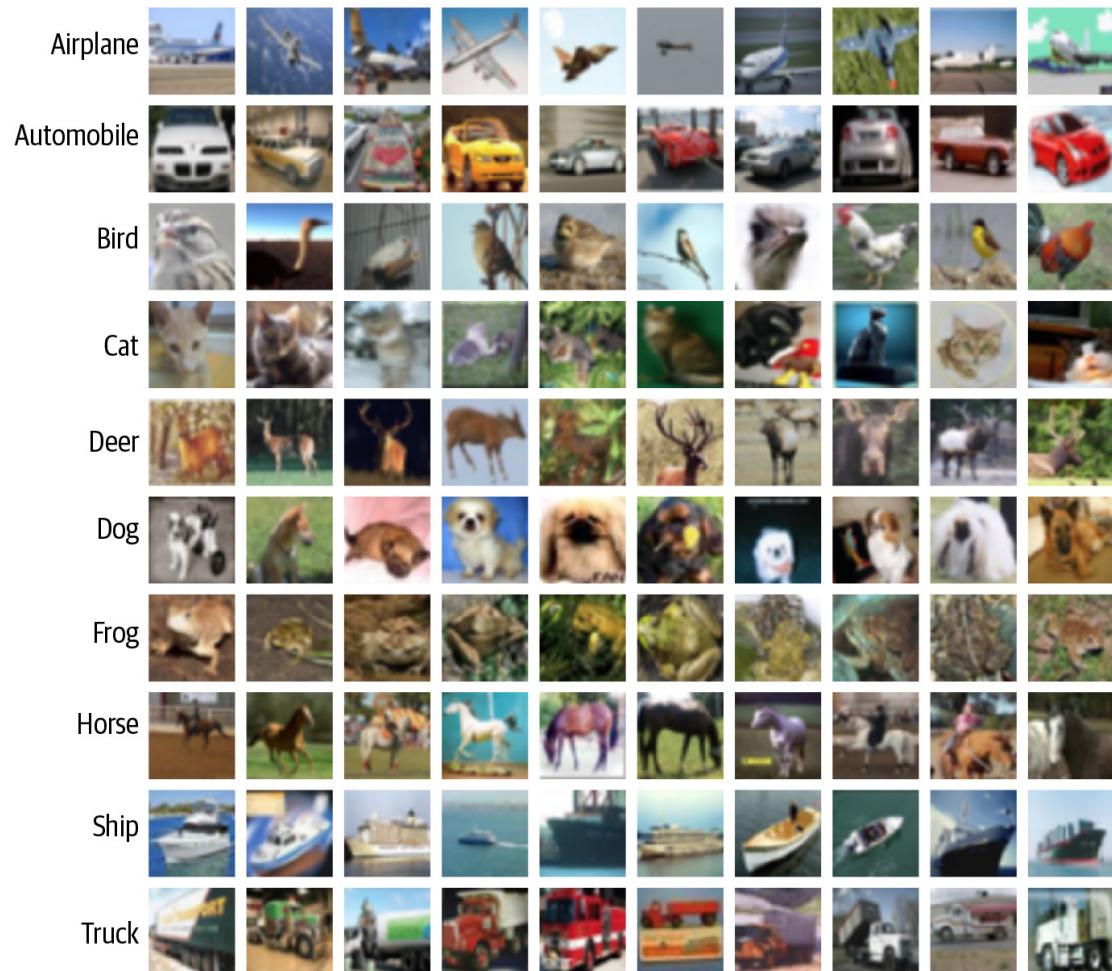


Figura 2-4. Imágenes de ejemplo del conjunto de datos CIFAR-10 (fuente: Krizhevsky, 2009)2

De forma predeterminada, los datos de la imagen constan de números enteros entre 0 y 255 para cada canal de píxeles. Primero debemos preprocesar las imágenes escalando estos valores para que estén entre 0 y 1, ya que las redes neuronales funcionan mejor cuando el valor absoluto de cada entrada es menor que 1.

También necesitamos cambiar el etiquetado de números enteros de las imágenes a vectores codificados en caliente, porque la salida de la red neuronal será una probabilidad de que la imagen pertenezca a cada clase. Si la etiqueta de entero de clase de una imagen es i , entonces su codificación onehot es un vector de longitud 10 (el número de clases) que tiene ceros en todos los elementos menos en el i -ésimo, que es 1.

Estos pasos se muestran en el Ejemplo 2-1.

Ejemplo 2-1. Preprocesamiento del conjunto de datos CIFAR-10

```
import numpy as np
from tensorflow.keras import datasets, utils
(x_train, y_train), (x_test, y_test) = datasets.cifar10.load_data()

NUM_CLASSES = 10

x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

y_train = utils.to_categorical(y_train, NUM_CLASSES)
y_test = utils.to_categorical(y_test, NUM_CLASSES)
```

1. Carga el conjunto de datos CIFAR-10. x_train y_{x_test} son arreglos numpy de la forma [50000, 32, 32, 3] y [10000, 32, 32, 3], respectivamente. y_train y y_test son arreglos numpy de la forma [50000, 1] y [10000, 1], respectivamente, conteniendo etiquetas enteras en el rango 0 a 9 para la clase de cada imagen.
2. Escala cada imagen, de modo que los valores de los canales de cada pixel caigan en el rango 0 a 1.
3. Codifica One-hot las etiquetas —las nuevas formas de y_train y y_test son [50000, 10] y [10000, 10], respectivamente.

Podemos ver que los datos de la imagen de entrenamiento (x_train) se almacenan en un tensor de forma [50000, 32, 32, 3]. No hay columnas ni filas en este conjunto de datos; en cambio, se trata de un tensor con cuatro dimensiones. Un tensor es simplemente una matriz multidimensional: es la extensión natural de una matriz a más

de dos dimensiones. La primera dimensión de este tensor hace referencia al índice de la imagen en el conjunto de datos, la segunda y la tercera se relacionan con el tamaño de la imagen y la última es el canal (es decir, rojo, verde o azul, ya que son imágenes RGB).

Por ejemplo, el Ejemplo 2-2 muestra cómo podemos encontrar el valor del canal de un píxel específico en una imagen.

Ejemplo 2-2. El valor del canal verde (1) del píxel en la posición (12,13) de la imagen 54

```
x_train[54, 12, 13, 1]
# 0.36862746
```

Construyendo el modelo

En Keras puede definir la estructura de una red neuronal como un modelo secuencial o utilizar la API funcional.

Un modelo secuencial es útil para definir rápidamente una pila lineal de capas (es decir, donde una capa sigue directamente a la capa anterior sin ninguna ramificación). Podemos definir nuestro modelo MLP usando la clase `Sequential` como se muestra en el Ejemplo 2-3.

Ejemplo 2-3. Construyendo nuestro MLP usando un modelo secuencial

```
from tensorflow.keras import layers, models

model = models.Sequential([
    layers.Flatten(input_shape=(32, 32, 3)),
    layers.Dense(200, activation = 'relu'),
    layers.Dense(150, activation = 'relu'),
    layers.Dense(10, activation = 'softmax'),
])
```

Muchos de los modelos de este libro requieren que la salida de una capa se pase a varias capas posteriores o, a la inversa, que una capa reciba entradas de varias capas anteriores. Para estos modelos, la clase `Sequential` no es adecuada y necesitaríamos usar la API funcional en su lugar, que es mucho más flexible.

CONSEJO

Recomiendo que incluso si recién estás comenzando a construir modelos lineales con Keras, sigas usando la API funcional en lugar de modelos secuenciales, ya que le

serán más útiles a largo plazo a medida que sus redes neuronales se vuelvan más complejas arquitectónicamente. La API funcional le brindará total libertad sobre el diseño de su red neuronal profunda.

El ejemplo 2-4 muestra el mismo MLP codificado utilizando la API funcional. Cuando utilizamos la API funcional, utilizamos la clase `Model` para definir las capas generales de entrada y salida del modelo.

Ejemplo 2-4. Construyendo nuestro MLP usando la API funcional

```
from tensorflow.keras import layers, models

input_layer = layers.Input(shape=(32, 32, 3))
x = layers.Flatten()(input_layer)
x = layers.Dense(units=200, activation = 'relu')(x)
x = layers.Dense(units=150, activation = 'relu')(x)
output_layer = layers.Dense(units=10, activation = 'softmax')(x)
model = models.Model(input_layer, output_layer)
```

Ambos métodos dan modelos idénticos; en la Figura 2-5 se muestra un diagrama de la arquitectura.

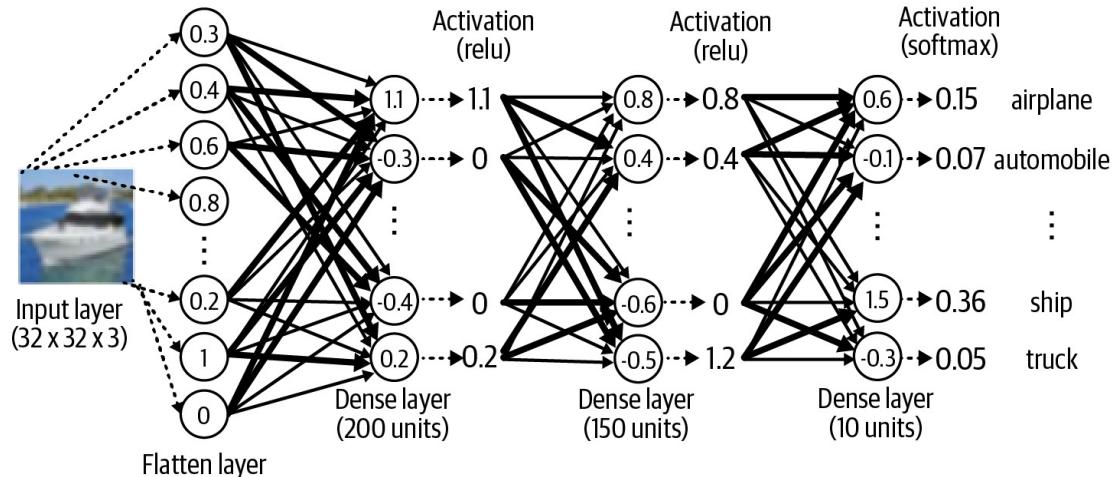


Figura 2-5. Un diagrama de la arquitectura MLP.

Veamos ahora con más detalle las diferentes capas y funciones de activación utilizadas dentro del MLP.

Capas

Para construir nuestro MLP, utilizamos tres tipos diferentes de capas: `Input`, `Flatten` y `Dense`.

La capa `Input` es un punto de entrada a la red. Le decimos a la red la forma que debe esperar cada elemento de datos como tupla. Tenga en cuenta que no especificamos el tamaño del lote; esto no es necesario ya que podemos pasar cualquier número de imágenes a la capa de entrada simultáneamente. No es necesario indicar explícitamente el tamaño del lote en la definición de la capa de entrada.

A continuación, aplanamos esta entrada en un vector, usando una capa `Flatten`. Esto da como resultado un vector de longitud 3,072 ($= 32 \times 32 \times 3$). La razón por la que hacemos esto es porque la capa `Dense` posterior requiere que su entrada sea plana, en lugar de una matriz multidimensional. Como veremos más adelante, otros tipos de capas requieren matrices multidimensionales como entrada, por lo que es necesario conocer la forma de entrada y salida requerida de cada tipo de capa para comprender cuándo es necesario usar `Flatten`.

La capa `Dense` es uno de los componentes básicos de una red neuronal. Contiene un número determinado de unidades que están densamente conectadas a la capa anterior, es decir, cada unidad de la capa está conectada a cada unidad de la capa anterior, a través de una única conexión que lleva un peso (que puede ser positivo o negativo). La salida de una unidad determinada es la suma ponderada de las entradas que recibe de la capa anterior, que luego pasa a través de una función de activación no lineal antes de enviarse a la siguiente capa. La función de activación es fundamental para garantizar que la red neuronal sea capaz de aprender funciones complejas y no solo genere una combinación lineal de sus entradas.

Funciones de activación

Hay muchos tipos de funciones de activación, pero tres de las más importantes son `ReLU`, `sigmoid` y `softmax`.

La función de activación `ReLU` (unidad lineal rectificada) se define como 0 si la entrada es negativa y, por lo demás, es igual a la entrada. La función de activación de `LeakyReLU` es muy similar a `ReLU`, con una diferencia clave: mientras que la función de activación `ReLU` devuelve 0 para valores de entrada menores que 0, la función `LeakyReLU` devuelve un pequeño número negativo proporcional a la entrada. Las unidades `ReLU` a veces pueden morir si siempre generan 0, debido a un gran sesgo hacia la preactivación de valores negativos. En este caso, el gradiente es 0 y, por lo tanto, no se propaga ningún error a través de esta unidad. Las activaciones de `LeakyReLU` solucionan este problema asegurándose siempre de que el gradiente sea distinto de cero. Las funciones basadas en `ReLU` se encuentran entre las activaciones más confiables para usar entre las capas de una red profunda para fomentar un entrenamiento estable.

La activación sigmoid es útil si desea que la salida de la capa se escale entre 0 y 1; por ejemplo, para problemas de clasificación binaria con una unidad de salida o problemas de clasificación de etiquetas múltiples, donde cada observación puede pertenecer a más de una clase. La Figura 2-6 muestra las funciones de activación ReLU, LeakyReLU y sigmoid una al lado de la otra para comparar.

```
model.fit(x_train, y_train, batch_size=32, epochs=10, shuffle=True)  ↻ ↑ ↓ ± ⌂
```

```
Epoch 1/10
1563/1563 [=====] - 3s 2ms/step - loss: 1.8377 - accuracy: 0.3369
Epoch 2/10
1563/1563 [=====] - 3s 2ms/step - loss: 1.6552 - accuracy: 0.4076
Epoch 3/10
1563/1563 [=====] - 3s 2ms/step - loss: 1.5743 - accuracy: 0.4396
Epoch 4/10
1563/1563 [=====] - 3s 2ms/step - loss: 1.5288 - accuracy: 0.4549
Epoch 5/10
1563/1563 [=====] - 3s 2ms/step - loss: 1.4888 - accuracy: 0.4706
Epoch 6/10
1563/1563 [=====] - 2s 2ms/step - loss: 1.4542 - accuracy: 0.4851
Epoch 7/10
1563/1563 [=====] - 3s 2ms/step - loss: 1.4332 - accuracy: 0.4908
Epoch 8/10
1563/1563 [=====] - 2s 2ms/step - loss: 1.4094 - accuracy: 0.4992
Epoch 9/10
1563/1563 [=====] - 2s 2ms/step - loss: 1.3896 - accuracy: 0.5045
Epoch 10/10
1563/1563 [=====] - 3s 2ms/step - loss: 1.3696 - accuracy: 0.5167
```

Figura 2-6. Las funciones de activación ReLU, LeakyReLU y sigmoid

La función de activación softmax es útil si desea que la suma total de la salida de la capa sea igual a 1; por ejemplo, para problemas de clasificación multiclas donde cada observación pertenece exactamente a una clase. Se define como:

$$y_i = \frac{x_i}{\sum_j x_j}$$

Aquí, J es el número total de unidades en la capa. En nuestra red neuronal, utilizamos una activación softmax en la capa final para garantizar que la salida sea un conjunto de 10 probabilidades cuya suma es 1, lo que puede interpretarse como la probabilidad de que la imagen pertenezca a cada clase.

En Keras, las funciones de activación se pueden definir dentro de una capa (Ejemplo 2-5) o como una capa separada (Ejemplo 2-6).

Ejemplo 2-5. Una función de activación de ReLU definida como parte de una capa Dense

```
x = layers.Dense(units=200, activation = 'relu')(x)
```

Ejemplo 2-6. Una función de activación de ReLU definida como su propia capa

```
x = layers.Dense(units=200)(x)
x = layers.Activation('relu')(x)
```

En nuestro ejemplo, pasamos la entrada a través de dos capas Dense, la primera con 200 unidades y la segunda con 150, ambas con funciones de activación ReLU.

Inspeccionando el modelo

Podemos usar el método `model.summary()` para inspeccionar la forma de la red en cada capa, como se muestra en la Tabla 2-1.

Capa (tipo)	Forma de salida	# de parámetros
Capa de entrada	(Ninguno, 32, 32, 3)	0
Flatten	(Ninguno, 3072)	0
Dense	(Ninguno, 200)	614,600
Dense	(Ninguno, 150)	30,150
Dense	(Ninguno, 10)	1,510
Parámetros totales	646,260	
Parámetros entrenables	646,260	
Parámetros no entrenables	0	

Observe cómo la forma de nuestra capa de entrada coincide con la forma de `x_train` y la forma de nuestra capa de salida densa coincide con la forma de `y_train`. Keras usa Ninguno como marcador para la primera dimensión para mostrar que aún no conoce la cantidad de observaciones que se pasarán a la red. De hecho, no es necesario; podríamos pasar tan fácilmente 1 observación a través de la red a la vez como 1,000. Esto se debe a que las operaciones tensoriales se realizan en todas las observaciones simultáneamente utilizando álgebra lineal; esta es la parte que maneja TensorFlow. También es la razón por la que se obtiene un aumento de rendimiento al entrenar redes neuronales profundas en GPU en lugar de CPU: las GPU están optimizadas para operaciones de tensor grandes, ya que estos cálculos también son necesarios para la manipulación de gráficos complejos.

El método de resumen también proporciona la cantidad de parámetros (pesos) que se entrenarán en cada capa. Si alguna vez descubre que su modelo se está entrenando demasiado lento, consulte el resumen para ver si hay capas que contengan una gran cantidad de pesos. Si es así, deberías considerar si la cantidad de unidades en la capa podría reducirse para acelerar el entrenamiento.

CONSEJO

¡Asegúrese de comprender cómo se calcula la cantidad de parámetros en cada capa! Es importante recordar que, de forma predeterminada, cada unidad dentro de una capa determinada también está conectada a una unidad de polarización adicional que siempre genera 1. Esto garantiza que la salida de la unidad pueda seguir siendo distinta de cero incluso cuando todas las entradas de la capa anterior sean 0.

Por lo tanto, el número de parámetros en la capa Dense de 200 unidades es $200 * (3,072 + 1) = 614,600$.

Compilando el modelo

En este paso, compilamos el modelo con un optimizador y una función de pérdida, como se muestra en el Ejemplo 2-7.

Ejemplo 2-7. Definición del optimizador y la función de pérdida

```
from tensorflow.keras import optimizers  
opt = optimizers.Adam(learning_rate=0.0005)  
model.compile(loss='categorical_crossentropy', optimizer=opt,  
metrics=['accuracy'])
```

Veamos ahora con más detalle lo que queremos decir con optimizadores y funciones de pérdida.

Funciones de pérdida

La red neuronal utiliza la función de pérdida para comparar su salida predicha con la verdad fundamental. Devuelve un único número por cada observación; cuanto mayor es este número, peor se ha desempeñado la red en esta observación.

Keras proporciona muchas funciones de pérdida integradas para elegir, o usted puede crear las suyas propias. Tres de los más utilizados son el error cuadrático medio, la entropía cruzada categórica y la entropía cruzada binaria. Es importante entender cuándo es apropiado utilizar cada uno.

Si su red neuronal está diseñada para resolver un problema de regresión (es decir, la salida es continua), entonces puede utilizar la pérdida de error cuadrático medio. Esta es la media de la diferencia al cuadrado entre la verdad fundamental e_i y el valor previsto p_i de cada unidad de salida, donde la media se toma sobre las n unidades de salida:

$$EEM = \sum_{i=1} (y_i - p_i)$$

Si está trabajando en un problema de clasificación en el que cada observación solo pertenece a una clase, entonces la entropía cruzada categórica es la función de pérdida correcta. Esto se define de la siguiente manera:

$$-\sum_{i=1} y_i \log(p_i)$$

Finalmente, si está trabajando en un problema de clasificación binaria con una unidad de salida, o en un problema de etiquetas múltiples donde cada observación puede pertenecer a múltiples clases simultáneamente, debe usar la entropía cruzada binaria:

$$\sum_{i=1} (y_i \log(p_i) + (1 - y_i) \log(1 - p_i))$$

Optimizadores

El optimizador es el algoritmo que se utilizará para actualizar los pesos en la red neuronal en función del gradiente de la función de pérdida. Uno de los optimizadores estables y más utilizados es Adam (Estimación del momento adaptativo).[3](#)

En la mayoría de los casos, no debería ser necesario modificar los parámetros por defecto del optimizador Adam, excepto la tasa de aprendizaje. Cuanto mayor sea la tasa de aprendizaje, mayor será el cambio en los pesos en cada paso del entrenamiento. Si bien el entrenamiento es inicialmente más rápido con una gran tasa de aprendizaje, la desventaja es que puede resultar en un entrenamiento menos estable y es posible que no encuentre el mínimo global de la función de pérdida. Este es un parámetro que quizás quieras sintonizar o ajustar durante el entrenamiento.

Otro optimizador común con el que te puedes encontrar es RMSProp (Propagación media cuadrática). Nuevamente, no debería necesitar ajustar demasiado los parámetros de este optimizador, pero vale la pena leer la documentación de Keras para comprender la función de cada parámetro.

Pasamos tanto la función de pérdida como el optimizador al método de compilación del modelo, así como un parámetro de métrica donde podemos especificar cualquier métrica adicional sobre la que nos gustaría informar durante el entrenamiento, como la precisión.

Entrenando el modelo

Hasta el momento, no le hemos mostrado ningún dato al modelo. Acabamos de configurar la arquitectura y compilar el modelo con una función de pérdida y un optimizador.

Para entrenar el modelo con los datos, simplemente llamamos al método de ajuste, como se muestra en el Ejemplo 2-8.

Ejemplo 2-8. Llamar al método de ajuste para entrenar el modelo

```
model.fit(x_train  
          , y_train  
          , batch_size = 32  
          , epochs = 10  
          , shuffle = True  
)
```

1. Los datos de la imagen sin procesar.
2. Las etiquetas de clase codificadas one-hot.
3. `batch_size` (el tamaño de lote) determina cuántas observaciones se pasarán a la red en cada paso de entrenamiento.
4. `epoch` (las épocas) determinan cuántas veces se mostrarán a la red los datos de entrenamiento completos.
5. Si `shuffle = True`, los lotes se extraerán aleatoriamente sin reemplazo de los datos de entrenamiento en cada paso de entrenamiento.

Esto comenzará a entrenar una red neuronal profunda para predecir la categoría de una imagen del conjunto de datos CIFAR-10. El proceso de formación funciona de la siguiente manera.

Primero, los pesos de la red se inicializan a pequeños valores aleatorios. Luego, la red realiza una serie de pasos de capacitación. En cada paso de entrenamiento, se pasa un lote de imágenes a través de la red y los errores se propagan hacia atrás para actualizar los pesos. El tamaño del lote determina cuántas imágenes hay en cada lote de pasos de entrenamiento. Cuanto mayor sea el tamaño del lote, más estable será el cálculo del gradiente, pero más lento será cada paso de entrenamiento.

CONSEJO

Usar todo el conjunto de datos para calcular el gradiente en cada paso de entrenamiento consumiría demasiado tiempo y consumiría demasiado tiempo, por lo que generalmente se usa un tamaño de lote entre 32 y 256. Ahora también se recomienda aumentar el tamaño del lote a medida que avanza el entrenamiento.[4](#)

Esto continúa hasta que se hayan visto todas las observaciones del conjunto de datos una vez. Esto completa la primera época. Luego, los datos pasan nuevamente a través de la red en lotes como parte de la segunda época. Este proceso se repite hasta que haya transcurrido el número especificado de épocas.

Durante el entrenamiento, Keras genera el progreso del procedimiento, como se muestra en la Figura 2-7. Podemos ver que el conjunto de datos de entrenamiento se dividió en 1563 lotes (cada uno de los cuales contiene 32 imágenes) y se mostró a la red 10 veces (es decir, durante 10 épocas), a una velocidad de aproximadamente 2 milisegundos por lote. La pérdida categórica de entropía cruzada cayó de 1.8377 a 1.3696, lo que resultó en un aumento de precisión del 33.69% después de la primera época al 51.67% después de la décima época.

```
model.fit(x_train, y_train, batch_size=32, epochs=10, shuffle=True)  ⟲ ↑ ↓ ± ⌂ ⟳  
Epoch 1/10  
1563/1563 [=====] - 3s 2ms/step - loss: 1.8377 - accuracy: 0.3369  
Epoch 2/10  
1563/1563 [=====] - 3s 2ms/step - loss: 1.6552 - accuracy: 0.4076  
Epoch 3/10  
1563/1563 [=====] - 3s 2ms/step - loss: 1.5743 - accuracy: 0.4396  
Epoch 4/10  
1563/1563 [=====] - 3s 2ms/step - loss: 1.5288 - accuracy: 0.4549  
Epoch 5/10  
1563/1563 [=====] - 3s 2ms/step - loss: 1.4888 - accuracy: 0.4706  
Epoch 6/10  
1563/1563 [=====] - 2s 2ms/step - loss: 1.4542 - accuracy: 0.4851  
Epoch 7/10  
1563/1563 [=====] - 3s 2ms/step - loss: 1.4332 - accuracy: 0.4908  
Epoch 8/10  
1563/1563 [=====] - 2s 2ms/step - loss: 1.4094 - accuracy: 0.4992  
Epoch 9/10  
1563/1563 [=====] - 2s 2ms/step - loss: 1.3896 - accuracy: 0.5045  
Epoch 10/10  
1563/1563 [=====] - 3s 2ms/step - loss: 1.3696 - accuracy: 0.5167
```

Figura 2-7. El resultado del método de ajuste.

Evaluación del modelo

Sabemos que el modelo logra una precisión del 51.9 % en el conjunto de entrenamiento, pero ¿cómo se desempeña con datos que nunca ha visto?

Para responder a esta pregunta podemos utilizar el método de evaluación proporcionado por Keras, como se muestra en el Ejemplo 2-9.

Ejemplo 2-9. Evaluación del rendimiento del modelo en el conjunto de pruebas

```
model.evaluate(x_test, y_test)
```

La Figura 2-8 muestra el resultado de este método.

```
10000/10000 [=====] - 1s 55us/step
[1.4358007415771485, 0.4896]
```

Figura 2-8. El resultado del método de evaluación.

El resultado es una lista de las métricas que estamos monitoreando: entropía cruzada categórica y precisión. Podemos ver que la precisión del modelo sigue siendo del 49,0% incluso en imágenes que nunca antes se habían visto. Tenga en cuenta que si el modelo adivinara aleatoriamente, lograría aproximadamente un 10% de precisión (porque hay 10 clases), por lo que 49.0 % es un buen resultado, dado que hemos utilizado una red neuronal muy básica.

Podemos ver algunas de las predicciones en el conjunto de prueba utilizando el método de predicción, como se muestra en el Ejemplo 2-10.

Ejemplo 2-10. Ver predicciones en el conjunto de prueba usando el método de predicción

```
CLASSES = np.array(['airplane', 'automobile', 'bird', 'cat',
'deer', 'dog',
', 'frog', 'horse', 'ship', 'truck'])
preds = model.predict(x_test)
preds_single = CLASSES[np.argmax(preds, axis = -1)]
actual_single = CLASSES[np.argmax(y_test, axis = -1)]
```

1. `preds` es una matriz de la forma `[10000, 10]`, es decir, un vector de 10 probabilidades de clase para cada observación.
2. Convertimos este conjunto de probabilidades nuevamente en una única predicción usando la función `argmax` de numpy. Aquí, `axis = -1` le dice a la función que colapse la matriz sobre la última dimensión (la dimensión de clases), de modo que la forma de `preds_single` sea `[10000, 1]`.

Podemos ver algunas de las imágenes junto con sus etiquetas y predicciones con el código del Ejemplo 2-11. Como era de esperar, alrededor de la mitad son correctas.

Ejemplo 2-11. Mostrar predicciones del MLP frente a las etiquetas actuales

```
import matplotlib.pyplot as plt

n_to_show = 10
indices = np.random.choice(range(len(x_test)), n_to_show)

fig = plt.figure(figsize=(15, 3))
fig.subplots_adjust(hspace=0.4, wspace=0.4)

for i, idx in enumerate(indices):
```

```

img = x_test[idx]
ax = fig.add_subplot(1, n_to_show, i+1)
ax.axis('off')
ax.text(0.5, -0.35, 'pred = ' + str(preds_single[idx]),
        fontsize=10
        , ha='center', transform=ax.transAxes)
ax.text(0.5, -0.7, 'act = ' + str(actual_single[idx]),
        fontsize=10
        , ha='center', transform=ax.transAxes)
ax.imshow(img)

```

La Figura 2-9 muestra una selección aleatoria de predicciones realizadas por el modelo, junto con las etiquetas verdaderas.

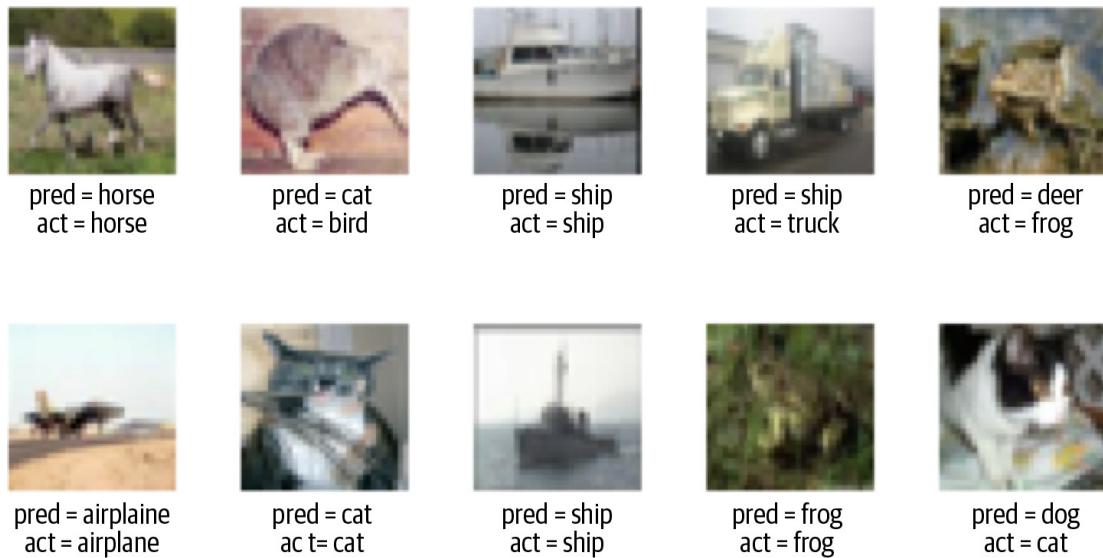


Figura 2-9. Algunas predicciones hechas por el modelo, junto con las etiquetas reales.

¡Felicitaciones! Acaba de construir un perceptrón multicapa utilizando Keras y lo utilizó para hacer predicciones sobre nuevos datos.

Aunque se trata de un problema de aprendizaje supervisado, cuando lleguemos a construir modelos generativos en capítulos futuros, muchas de las ideas centrales de este capítulo (como funciones de pérdida, funciones de activación y comprensión de las formas de las capas) seguirán siendo extremadamente importantes. A continuación veremos formas de mejorar este modelo introduciendo algunos tipos de capas nuevos.

Red neuronal convolucional (CNN)

Una de las razones por las que nuestra red aún no funciona tan bien como debería, es porque no hay nada en la red que tenga en cuenta la estructura espacial de las

imágenes de entrada. De hecho, nuestro primer paso es aplanar la imagen en un solo vector, para que podamos pasarla a la primera capa Dense.

Para lograr esto necesitamos usar una capa convolucional.

Capas convolucionales

Primero, debemos comprender qué se entiende por convolución en el contexto del aprendizaje profundo.

La Figura 2-10 muestra dos porciones diferentes de $3 \times 3 \times 1$ de una imagen en escala de grises convolucionada con un filtro (núcleo o kernel) de $3 \times 3 \times 1$. La convolución se realiza multiplicando el filtro en píxeles por la porción de la imagen y sumando los resultados. La salida es más positiva cuando la parte de la imagen coincide estrechamente con el filtro y más negativa cuando la parte de la imagen es la inversa del filtro.

El ejemplo superior resuena fuertemente con el filtro, por lo que produce un valor positivo grande. El ejemplo inferior no resuena mucho con el filtro, por lo que produce un valor cercano a cero.

3 × 3 portion of an image		Filter																		
<table border="1" style="border-collapse: collapse; width: 100%;"><tbody><tr><td style="background-color: #80E6AA;">0.6</td><td style="background-color: #D9EAD3;">0.4</td><td style="background-color: #80E6AA;">0.6</td></tr><tr><td style="background-color: #D9EAD3;">0.1</td><td style="background-color: #F0E68C;">-0.2</td><td style="background-color: #F0E68C;">-0.3</td></tr><tr><td style="background-color: #F0E68C;">-0.5</td><td style="background-color: #F0E68C;">-0.4</td><td style="background-color: #F0E68C;">-0.3</td></tr></tbody></table>	0.6	0.4	0.6	0.1	-0.2	-0.3	-0.5	-0.4	-0.3	×	<table border="1" style="border-collapse: collapse; width: 100%;"><tbody><tr><td style="background-color: #80E6AA;">1</td><td style="background-color: #80E6AA;">1</td><td style="background-color: #80E6AA;">1</td></tr><tr><td style="background-color: #D9EAD3;">0</td><td style="background-color: #D9EAD3;">0</td><td style="background-color: #D9EAD3;">0</td></tr><tr><td style="background-color: #F0E68C;">-1</td><td style="background-color: #F0E68C;">-1</td><td style="background-color: #F0E68C;">-1</td></tr></tbody></table>	1	1	1	0	0	0	-1	-1	-1
0.6	0.4	0.6																		
0.1	-0.2	-0.3																		
-0.5	-0.4	-0.3																		
1	1	1																		
0	0	0																		
-1	-1	-1																		
		= 2.8																		

3 × 3 portion of an image		Filter																		
<table border="1" style="border-collapse: collapse; width: 100%;"><tbody><tr><td style="background-color: #F0E68C;">-0.7</td><td style="background-color: #80E6AA;">0.6</td><td style="background-color: #80E6AA;">0.2</td></tr><tr><td style="background-color: #D9EAD3;">0.1</td><td style="background-color: #D9EAD3;">0.5</td><td style="background-color: #F0E68C;">-0.3</td></tr><tr><td style="background-color: #F0E68C;">-0.3</td><td style="background-color: #F0E68C;">-0.4</td><td style="background-color: #80E6AA;">0.5</td></tr></tbody></table>	-0.7	0.6	0.2	0.1	0.5	-0.3	-0.3	-0.4	0.5	×	<table border="1" style="border-collapse: collapse; width: 100%;"><tbody><tr><td style="background-color: #80E6AA;">1</td><td style="background-color: #80E6AA;">1</td><td style="background-color: #80E6AA;">1</td></tr><tr><td style="background-color: #D9EAD3;">0</td><td style="background-color: #D9EAD3;">0</td><td style="background-color: #D9EAD3;">0</td></tr><tr><td style="background-color: #F0E68C;">-1</td><td style="background-color: #F0E68C;">-1</td><td style="background-color: #F0E68C;">-1</td></tr></tbody></table>	1	1	1	0	0	0	-1	-1	-1
-0.7	0.6	0.2																		
0.1	0.5	-0.3																		
-0.3	-0.4	0.5																		
1	1	1																		
0	0	0																		
-1	-1	-1																		
		= -0.1																		

Figura 2-10. Un filtro convolucional de 3×3 aplicado a dos porciones de una imagen en escala de grises

Si movemos el filtro por toda la imagen de izquierda a derecha y de arriba a abajo, registrando la salida convolucional a medida que avanzamos, obtenemos una nueva matriz que selecciona una característica particular de la entrada, dependiendo de los valores del filtro.

Por ejemplo, la Figura 2-11 muestra dos filtros diferentes que resaltan los bordes horizontales y verticales.

EJECUTANDO EL CÓDIGO PARA ESTE EJEMPLO

Puede ver este proceso convolucional realizado manualmente en el cuaderno de Jupyter ubicado en notebooks/02_deeplearning/02_cnn/convolutions.ipynb en el repositorio.

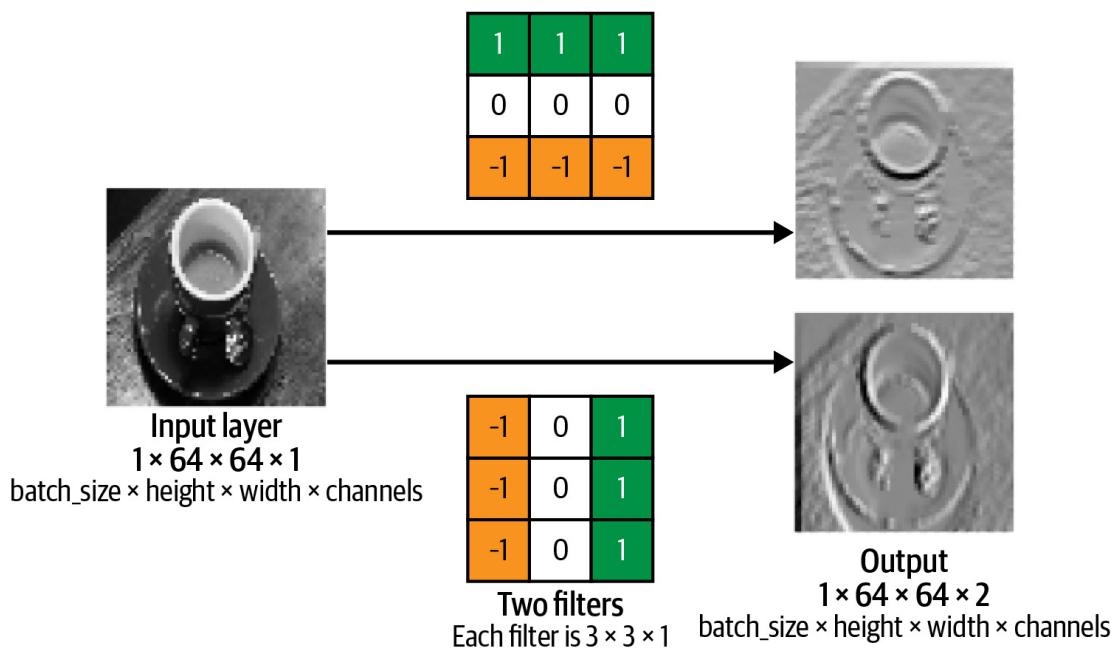


Figura 2-11. Dos filtros convolucionales aplicados a una imagen en escala de grises.

Una capa convolucional es simplemente una colección de filtros, donde los valores almacenados en los filtros son los pesos que aprende la red neuronal mediante el entrenamiento. Inicialmente son aleatorios, pero gradualmente los filtros adaptan sus pesos para comenzar a seleccionar características interesantes como bordes o combinaciones de colores particulares.

En Keras, la capa Conv2D aplica convoluciones a un tensor de entrada con dos dimensiones espaciales (como una imagen). Por ejemplo, el código que se muestra en el Ejemplo 2-12 crea una capa convolucional con dos filtros, para que coincida con el ejemplo de la Figura 2-11.

Ejemplo 2-12. Una capa Conv2D aplicada a imágenes de entrada en escala de grises

```
from tensorflow.keras import layers

input_layer = layers.Input(shape=(64, 64, 1))
conv_layer_1 = layers.Conv2D(
    filters = 2
    , kernel_size = (3,3)
    , strides = 1
    , padding = "same"
)(input_layer)
```

A continuación, veamos con más detalle dos de los argumentos de la capa Conv2D: `strides` (zancadas) y `padding` (relleno).

Stride

El parámetro `strides` (zancadas) es el tamaño de paso utilizado por la capa para mover los filtros a través de la entrada. Por lo tanto, aumentar la zancada reduce el tamaño del tensor de salida. Por ejemplo, cuando `strides = 2`, la altura y el ancho del tensor de salida serán la mitad del tamaño del tensor de entrada. Esto es útil para reducir el tamaño espacial del tensor ya que pasa por la red, al tiempo que aumenta el número de canales.

Padding

El parámetro de entrada `padding = "same"` rellena los datos de entrada con ceros para que el tamaño de salida de la capa sea exactamente el mismo que el tamaño de entrada cuando `strides = 1`.

La Figura 2-12 muestra un kernel de 3×3 que se pasa sobre una imagen de entrada de 5×5 , con `padding = "same"` y `strides = 1`. El tamaño de salida de esta capa convolucional también sería 5×5 , ya que el relleno permite que el núcleo se extienda sobre el borde de la imagen, de modo que encaje cinco veces en ambas direcciones.

Sin relleno, el núcleo sólo podía encajar tres veces en cada dirección, dando un tamaño de salida de 3×3 .

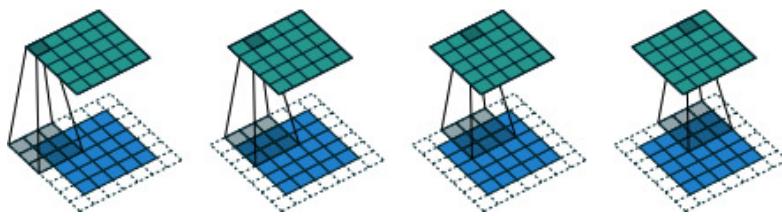


Figura 2-12. Un núcleo de $3 \times 3 \times 1$ (gris) que se pasa sobre una imagen de entrada de $5 \times 5 \times 1$ (azul), con `padding = "same"` y `strides = 1`, para generar la salida de $5 \times 5 \times 1$ (verde) (fuente : Dumoulin y Visin, 2018)⁵

Establecer `padding = "same"` es una buena manera de garantizar que pueda realizar un seguimiento fácilmente del tamaño del tensor a medida que pasa a través de muchas capas convolucionales. La forma de la salida de una capa convolucional con `padding = "same"` es:

```
altura de entrada
stride,
, filtros)
stride
```

Apilando capas convolucionales

La salida de una capa Conv2D es otro tensor de cuatro dimensiones, ahora de la forma (`batch_size, height, width, filters`) (tamaño de lote, altura, ancho, filtros), por lo que podemos apilar capas Conv2D una encima de otra para aumentar la profundidad de nuestra red neuronal y hacerla más poderosa. Para demostrar esto, imaginemos que estamos aplicando capas Conv2D al conjunto de datos CIFAR-10 y deseamos predecir la etiqueta de una imagen determinada. Tenga en cuenta que esta vez, en lugar de un canal de entrada (escala de grises), tenemos tres (rojo, verde y azul).

El ejemplo 2-13 muestra cómo construir una red neuronal convolucional simple que podríamos entrenar para tener éxito en esta tarea.

Ejemplo 2-13. Código para construir un modelo de red neuronal convolucional usando Keras

```
from tensorflow.keras import layers, models

input_layer = layers.Input(shape=(32, 32, 3))
conv_layer_1 = layers.Conv2D(
    filters = 10
    , kernel_size = (4, 4)
    , strides = 2
    , padding = 'same'
    )(input_layer)
conv_layer_2 = layers.Conv2D(
```

```

filters = 20
, kernel_size = (3, 3)
, strides = 2
, padding = 'same'
)(conv_layer_1)
flatten_layer = layers.Flatten()(conv_layer_2)
output_layer = layers.Dense(units=10, activation = 'softmax')
(flatten_layer)
model = models.Model(input_layer, output_layer)

```

Este código corresponde al diagrama mostrado en la Figura 2-13.

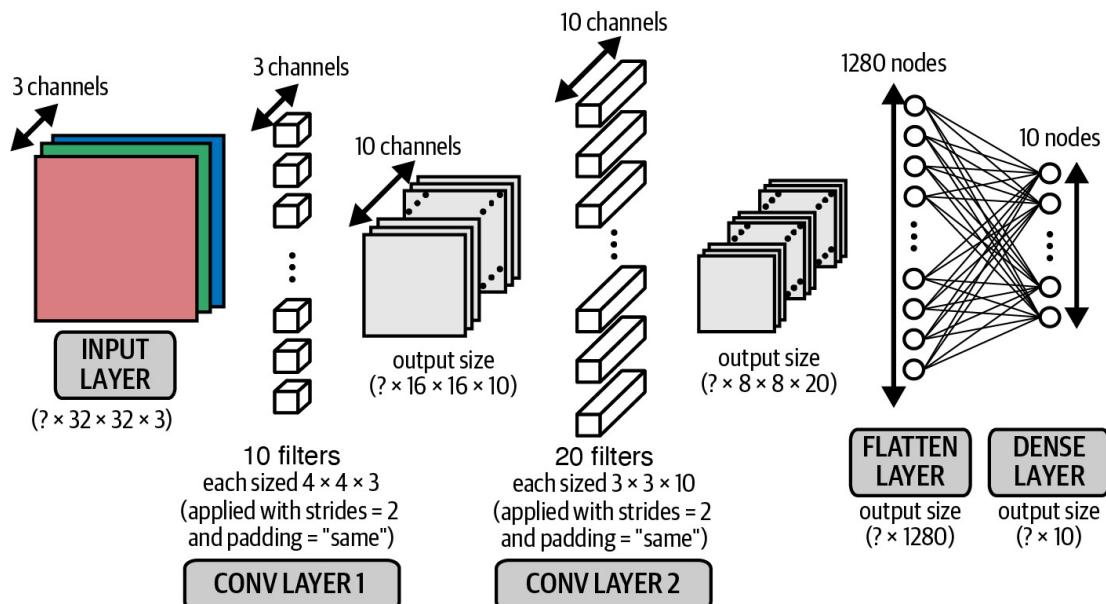


Figura 2-13. Un diagrama de una red neuronal convolucional.

Tenga en cuenta que ahora que estamos trabajando con imágenes en color, cada filtro en la primera capa convolucional tiene una profundidad de 3 en lugar de que 1 (es decir, cada filtro tiene forma $4 \times 4 \times 3$, en lugar de $4 \times 4 \times 1$). Esto es para hacer coincidir los tres canales (rojo, verde, azul) de la imagen de entrada. La misma idea se aplica a los filtros de la segunda capa convolucional que tienen una profundidad de 10, para que coincida con los 10 canales generados por la primera capa convolucional.

CONSEJO

En general, la profundidad de los filtros en una capa siempre es igual al número de canales emitidos por la capa anterior.

Inspeccionando el modelo

Es realmente informativo observar cómo cambia la forma del tensor a medida que los datos fluyen de una capa convolucional a la siguiente. Podemos usar el método `model.summary()` para inspeccionar la forma del tensor a medida que pasa a través de la red (Tabla 2-2).

Capa (tipo)	Forma de salida	# de parámetros
Capa de entrada	(Ninguno, 32, 32, 3)	
Conv2D	(Ninguno, 16, 16, 10)	490
Conv2D	(Ninguno, 8, 8, 20)	1,820
Flatten		(Ninguno, 1280)
Dense	(Ninguno, 10)	12,810
Parámetros totales		15,120
Parámetros entrenables		15,120
Parámetros no entrenables		

Recorramos nuestra red capa por capa, observando la forma del tensor a medida que avanzamos:

1. La forma de entrada es (`None, 32, 32, 3`): Keras usa `None` para representar el hecho de que podemos pasar cualquier cantidad de imágenes a través de la red simultáneamente. Dado que la red solo realiza álgebra tensorial, no necesitamos pasar imágenes a través de la red individualmente, sino que podemos pasárlas juntas como un lote.
2. La forma de cada uno de los 10 filtros en la primera capa convolucional es $4 \times 4 \times 3$. Esto se debe a que hemos elegido que cada filtro tenga una altura y un ancho de 4 (`kernel_size = (4, 4)`) y hay tres canales en la capa anterior (rojo, verde y azul).

Por lo tanto, el número de parámetros (o pesos) en la capa es $(4 \times 4 \times 3 + 1) \times 10 = 490$, donde $+ 1$ se debe a la inclusión de un término de sesgo adjunto a cada uno de los filtros. La salida de cada filtro será la multiplicación por píxeles de los pesos del filtro y la sección $4 \times 4 \times 3$ de la imagen que cubre. A medida que `strides = 2` y `padding = "same"`, el ancho y el alto de la salida se reducen a la mitad a 16, y como hay 10 filtros, la salida de la primera capa es un lote de tensores, cada uno de los cuales tiene forma [16, 16, 10].

3. En la segunda capa convolucional, elegimos que los filtros sean 3×3 y ahora tengan una profundidad de 10, para que coincida con el número de canales en la capa anterior. Dado que hay 20 filtros en esta capa, esto da un número total de parámetros (pesos) de $(3 \times 3 \times 10 + 1) \times 20 = 1820$.

Nuevamente, usamos `strides = 2` y `padding = "same"`, por lo que el ancho y el alto se reducen a la mitad. Esto nos da una forma de salida general de `(None, 8, 8, 20)`.

4. Ahora aplanamos el tensor usando la capa `Flatten` de Keras. Esto da como resultado un conjunto de $8 \times 8 \times 20 = 1280$ unidades.

Tenga en cuenta que no hay parámetros que aprender en una capa `Flatten` ya que la operación es solo una reestructuración del tensor.

5. Finalmente conectamos estas unidades a una capa `Dense` de 10 unidades con activación `softmax`, que representa la probabilidad de cada categoría en una tarea de clasificación de 10 categorías. Esto crea un extra de $1280 \times 10 = 12,810$ parámetros (pesos) para aprender.

Este ejemplo demuestra cómo podemos encadenar capas convolucionales para crear una red neuronal convolucional.

Antes de ver cómo se compara esto en precisión con nuestra red neuronal densamente conectada, examinaremos dos más técnicas que también pueden mejorar el rendimiento: normalización por lotes y abandono.

Normalización por lotes

Un problema común al entrenar una red neuronal profunda es garantizar que los pesos de la red permanezcan dentro de un rango razonable de valores; si comienzan a ser demasiado grandes, esto es una señal de que su red está sufriendo lo que se conoce como el problema del gradiente explosivo. A medida que los errores se propagan hacia atrás a través de la red, el cálculo del gradiente en las capas anteriores a veces puede crecer exponencialmente, provocando fluctuaciones salvajes en los valores de peso.

ADVERTENCIA

Si su función de pérdida comienza a devolver `NaN`, es probable que sus pesos hayan aumentado lo suficiente como para causar un error de desbordamiento.

Esto no necesariamente sucede inmediatamente cuando comienzas a entrenar la red. A veces puede estar entrenando felizmente durante horas cuando, de repente, la función de pérdida devuelve `NaN` y su red explota. Esto puede resultar increíblemente molesto. Para evitar que esto suceda, es necesario comprender la causa raíz del problema del gradiente explosivo.

Cambio covariado

Una de las razones para escalar los datos de entrada a una red neuronal es garantizar un inicio estable del entrenamiento durante las primeras iteraciones. Dado que los pesos de la red son inicialmente aleatorios, la entrada sin escala podría potencialmente crear valores de activación enormes que conduzcan inmediatamente a gradientes que explotan. Por ejemplo, en lugar de pasar valores de píxeles de 0 a 255 a la capa de entrada, normalmente escalamos estos valores entre -1 y 1 .

Debido a que la entrada está escalada, es natural esperar que las activaciones de todas las capas futuras también estén relativamente bien escaladas. Inicialmente esto puede ser cierto, pero a medida que los pesos de la red y los pesos se alejan de sus valores iniciales aleatorios, esta suposición puede comenzar a fallar. Este fenómeno se conoce como cambio covariado.

ANALOGÍA DEL CAMBIO COVARIADO

Imagina que llevas una gran pila de libros y te golpea una ráfaga de viento. Mueves los libros en dirección opuesta al viento para compensar, pero al hacerlo, algunos de los libros se mueven, de modo que la torre es un poco más inestable que antes. Al principio esto está bien, pero con cada ráfaga la pila se vuelve cada vez más inestable, hasta que finalmente los libros se han movido tanto que la pila se derrumba.

Este es un cambio covariado.

En relación con las redes neuronales, cada capa es como un libro en una pila.

Para permanecer estable, cuando la red actualiza los pesos, cada capa asume implícitamente que la distribución de su entrada de la capa inferior es aproximadamente consistente en todas las iteraciones. Sin embargo, dado que no hay nada que impida que las distribuciones de activación se desplacen significativamente en una determinada dirección, esto a veces puede provocar valores de peso desbocados y un colapso general de la red.

Entrenamiento mediante normalización por lotes

La normalización por lotes es una técnica que reduce drásticamente este problema. La solución es sorprendentemente sencilla. Durante el entrenamiento, una capa de normalización por lotes calcula la media y la desviación estándar de cada uno de sus canales de entrada en todo el lote y la normaliza restando la media y dividiéndola entre la desviación estándar. Luego hay dos parámetros aprendidos para cada canal, la escala (γ) y el desplazamiento (β). La salida es simplemente la entrada

normalizada, escalado por gamma y desplazado por beta. La Figura 2-14 muestra todo el proceso.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots m\}$;

Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Figura 2-14. El proceso de normalización por lotes (fuente: Ioffe y Szegedy, 2015) [6]

Podemos colocar capas de normalización por lotes después de capas densas o convolucionales para normalizar la salida.

CONSEJO

Refiriéndose a nuestro ejemplo anterior, es un poco como conectar las capas de libros con pequeños conjuntos de resortes ajustables que garantizan que no haya grandes cambios generales en sus posiciones con el tiempo.

Predicción mediante normalización por lotes

Quizás se pregunte cómo funciona esta capa en el momento de la predicción. Cuando se trata de predicción, es posible que solo queramos predecir una única observación,

por lo que no hay un lote sobre el cual calcular la media y la desviación estándar. Para solucionar este problema, durante el entrenamiento, una capa de normalización por lotes también calcula el promedio móvil de la media y desviación estándar de cada canal y almacena este valor como parte de la capa para usar en el momento de la prueba.

¿Cuántos parámetros hay dentro de una capa de normalización por lotes? Para cada canal de la capa anterior, es necesario aprender dos pesos: la escala (*gamma*) y el desplazamiento (*beta*). Estos son los parámetros entrenables. También es necesario calcular la media móvil y la desviación estándar para cada canal, pero como se derivan de los datos que pasan a través de la capa en lugar de entrenarse mediante retropropagación, se denominan parámetros no entrenables. En total, esto da cuatro parámetros para cada canal en la capa anterior, donde dos son entrenables y dos no entrenables.

En Keras, la capa `BatchNormalization` implementa la funcionalidad de normalización por lotes, como se muestra en el Ejemplo 214.

Ejemplo 2-14. Una capa `BatchNormalization` en Keras

```
from tensorflow.keras import layers
layers.BatchNormalization(momentum = 0.9)
```

El parámetro `momentum` es el peso que se le da al valor anterior al calcular la media móvil y la desviación estándar móvil.

Abandono

Al estudiar para un examen, es una práctica común que los estudiantes utilicen trabajos anteriores y preguntas de muestra para mejorar su conocimiento de la materia. Algunos estudiantes intentan memorizar las respuestas a estas preguntas, pero luego fracasan en el examen porque no han entendido realmente el tema. Los mejores estudiantes utilizan el material de práctica para mejorar su comprensión general, de modo que aún puedan responder correctamente cuando se enfrentan a nuevas preguntas que no habían visto antes.

El mismo principio se aplica al aprendizaje automático. Cualquier algoritmo de aprendizaje automático exitoso debe garantizar que se generalice a datos invisibles, en lugar de simplemente recordar el conjunto de datos de entrenamiento. Si un algoritmo funciona bien en el conjunto de datos de entrenamiento, pero no en el conjunto de datos de prueba, decimos que sufre de sobreajuste. Para contrarrestar este problema, utilizamos técnicas de regularización, que garantizan que el modelo sea penalizado si comienza a sobreajustarse.

Hay muchas formas de regularizar un algoritmo de aprendizaje automático, pero para el aprendizaje profundo, una de las más comunes es mediante el uso de capas de abandono. Esta idea fue introducida por Hinton et al. en 2012⁷ y presentado en un artículo de 2014 por Srivastava et al.⁸

Las capas Dropout son muy simples. Durante el entrenamiento, cada capa de abandono elige un conjunto aleatorio de unidades de la capa anterior y establece su salida en 0, como se muestra en la figura 2-15.

Increíblemente, esta simple adición reduce drásticamente el sobreajuste al garantizar que la red no se vuelva demasiado dependiente de ciertas unidades o grupos de unidades que, de hecho, solo recuerdan observaciones del conjunto de entrenamiento. Si utilizamos capas de abandono, la red no puede depender demasiado de ninguna unidad y, por lo tanto, el conocimiento se distribuye de manera más uniforme en toda la red.

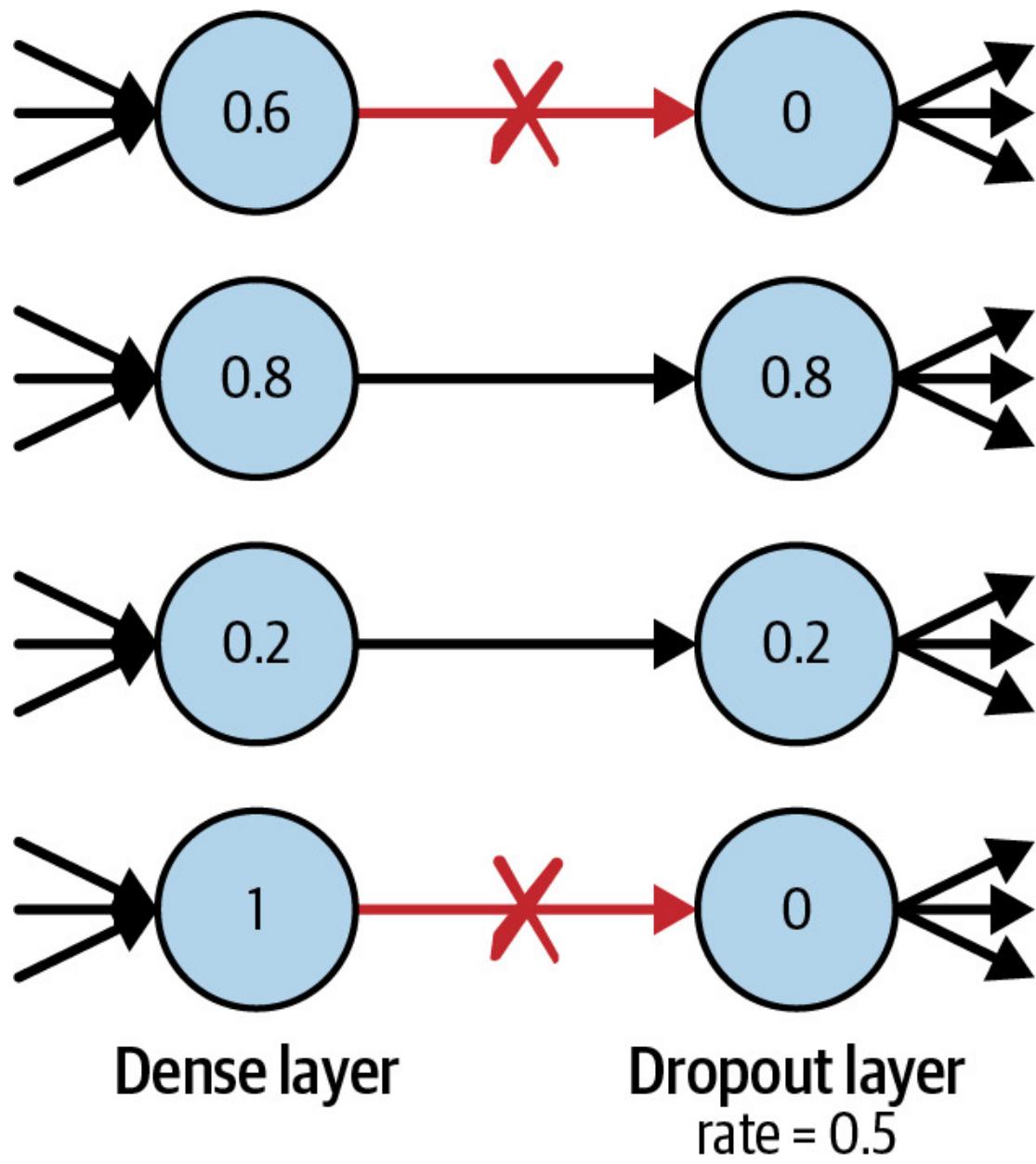


Figura 2-15. Una capa de abandono

Esto hace que el modelo sea mucho mejor a la hora de generalizar datos invisibles, porque la red ha sido entrenada para producir predicciones precisas incluso en condiciones desconocidas, como las provocadas por la eliminación de unidades aleatorias.

No hay pesos que aprender dentro de una capa de abandono, ya que las unidades que se descartan se deciden de forma estocástica. En el momento de la predicción, la capa de abandono no elimina ninguna unidad, por lo que se utiliza la red completa para hacer predicciones.

ANALOGÍA DEL ABANDONO

Volviendo a nuestra analogía, es un poco como un estudiante de matemáticas practicando trabajos anteriores con una selección aleatoria de fórmulas clave que faltan en su libro de fórmulas. De esta manera, aprende a responder preguntas mediante la comprensión de los principios básicos, en lugar de buscar siempre las fórmulas en los mismos lugares del libro. Cuando llegue el momento de los exámenes, le resultará mucho más fácil responder preguntas que nunca antes había visto, debido a su capacidad para generalizar más allá del material de entrenamiento.

La capa Dropout en Keras implementa esta funcionalidad, con el parámetro `rate` especificando la proporción de unidades que se eliminarán de la capa anterior, como se muestra en el Ejemplo 2-15.

Ejemplo 2-15. Una capa Dropout en Keras

```
from tensorflow.keras import layers  
layers.Dropout(rate = 0.25)
```

Las capas Dropout se usan más comúnmente después de capas densas, ya que son las más propensas a sobreajustarse debido a la mayor cantidad de pesos, aunque también puede usarlas después de capas convolucionales.

CONSEJO

También se ha demostrado que la normalización por lotes reduce el sobreajuste y, por lo tanto, muchas arquitecturas modernas de aprendizaje profundo no utilizan el abandono en absoluto, y dependen únicamente de la normalización por lotes para la regularización. Como ocurre con la mayoría de los principios del aprendizaje profundo, no existe una regla de oro que se aplique en todas las situaciones; la única manera de saber con certeza qué es lo mejor, es probar diferentes arquitecturas y ver cuál funciona mejor en un conjunto de datos reservados.

Construyendo la CNN

Ahora ha visto tres nuevos tipos de capas de Keras: Conv2D, BatchNormalization y Dropout. Juntemos estas piezas en un modelo de CNN y veamos cómo se desempeña en el conjunto de datos CIFAR-10.

EJECUTANDO EL CÓDIGO PARA ESTE EJEMPLO

Puede ejecutar el siguiente ejemplo en el cuaderno de Jupyter en el repositorio de libros llamado notebooks/02_deeplearning/02_cnn/cnn.ipynb.

La arquitectura del modelo que probaremos se muestra en el ejemplo 2-16.

Ejemplo 2-16. Código para construir un modelo CNN usando Keras

```
from tensorflow.keras import layers, models

input_layer = layers.Input((32,32,3))

x = layers.Conv2D(filters = 32, kernel_size = 3
                  , strides = 1, padding = 'same')(input_layer)
x = layers.BatchNormalization()(x)
x = layers.LeakyReLU()(x)

x = layers.Conv2D(filters = 32, kernel_size = 3, strides = 2,
                  padding = 'same')(x)
x = layers.BatchNormalization()(x)
x = layers.LeakyReLU()(x)x = layers.Conv2D(filters = 64, kernel_size =
3, strides = 1,
                  padding = 'same')(x)
x = layers.BatchNormalization()(x)
x = layers.LeakyReLU()(x)

x = layers.Conv2D(filters = 64, kernel_size = 3, strides = 2,
                  padding = 'same')(x)
x = layers.BatchNormalization()(x)
x = layers.LeakyReLU()(x)

x = layers.Flatten()(x)

x = layers.Dense(128)(x)
x = layers.BatchNormalization()(x)
x = layers.LeakyReLU()(x)
x = layers.Dropout(rate = 0.5)(x)

output_layer = layers.Dense(10, activation = 'softmax')(x)

model = models.Model(input_layer, output_layer)
```

Usamos cuatro capas Conv2D apiladas, cada una seguida de una BatchNormalization y una capa LeakyReLU. Después de aplanar el tensor resultante, pasamos los datos a través de una capa Dense de tamaño 128, seguida nuevamente por una BatchNormalization y una capa LeakyReLU. A esto le sigue inmediatamente una capa Dropout para la regularización, y la red finaliza con una capa Dense de salida de tamaño 10.

CONSEJO

El orden en el que se utilizan las capas de activación y normalización por lotes es una cuestión de preferencia. Por lo general, las capas de normalización por lotes se colocan antes de la activación, pero algunas arquitecturas exitosas usan estas capas al revés. Si elige utilizar la normalización por lotes antes de la activación, puede recordar el orden utilizando el acrónimo **BAD** (batch normalization, activation, dropout: normalización por lotes, activación y luego abandono).

El resumen del modelo se muestra en la Tabla 2-3.

Capa (tipo)	Forma de salida	# de Parámetros
Capa de entrada	(Ninguno, 32, 32, 3)	
Conv2D	(Ninguno, 32, 32, 32)	896
BatchNormalization	(Ninguno, 32, 32, 32)	128
LeakyReLU	(Ninguno, 32, 32, 32)	
Conv2D	(Ninguno, 16, 16, 32)	9,248
BatchNormalization	(Ninguno, 16, 16, 32)	128
LeakyReLU	(Ninguno, 16, 16, 32)	
Conv2D	(Ninguno, 16, 16, 64)	18,496
BatchNormalization	(Ninguno, 16, 16, 64)	256
LeakyReLU	(Ninguno, 16, 16, 64)	
Conv2D	(Ninguno, 8, 8, 64)	36,928
BatchNormalization	(Ninguno, 8, 8, 64)	256
LeakyReLU	(Ninguno, 8, 8, 64)	
Flatten	(Ninguno, 4096)	
Dense	(Ninguno, 128)	524,416
BatchNormalization	(Ninguno, 128)	512
LeakyReLU	(Ninguno, 128)	
Dropout	(Ninguno, 128)	
Dense	(Ninguno, 10)	1290
Parámetros totales		592,554
Parámetros entrenables		591,914
Parámetros no entrenables		640

CONSEJO

Antes de continuar, asegúrese de poder calcular manualmente la forma de salida y la cantidad de parámetros para cada capa. ¡Es un buen ejercicio para demostrarte a ti mismo que has comprendido completamente cómo se construye cada capa y cómo se conecta con la capa anterior!

No olvide incluir los pesos de sesgo que se incluyen como parte de las capas Conv2D y Dense.

Entrenando y evaluando la CNN

Compilamos y entrenamos el modelo exactamente de la misma manera que antes y llamamos al método de evaluación para determinar su precisión en el conjunto de reservas (Figura 2-16).

```
model.evaluate(x_test, y_test, batch_size=1000)  
10000/10000 [=====] - 15s 1ms/step  
[0.8423407137393951, 0.7155999958515167]
```

Figura 2-16. Rendimiento de CNN

Como puede ver, este modelo ahora logra una precisión del 71.5%, frente al 49.0% anterior. ¡Mucho mejor! La Figura 2-17 muestra algunas predicciones de nuestro nuevo modelo convolucional.

Esta mejora se logró simplemente cambiando la arquitectura del modelo para incluir capas convolucionales, de normalización por lotes y de abandono. Observe que la cantidad de parámetros es en realidad menor en nuestro nuevo modelo que en el modelo anterior, aunque la cantidad de capas es mucho mayor. Esto demuestra la importancia de experimentar con el diseño de su modelo y sentirse cómodo con la forma en que se pueden utilizar los diferentes tipos de capas para su beneficio. Al construir modelos generativos, se vuelve aún más importante comprender el funcionamiento interno de su modelo, ya que son las capas intermedias de su red que captura las características de alto nivel que más le interesan.



Figura 2-17. Predicciones de CNN

Resumen

Este capítulo presentó los conceptos básicos de aprendizaje profundo que necesitará para comenzar a construir modelos generativos profundos.

Comenzamos construyendo un perceptrón multicapa (MLP) usando Keras y entrenó el modelo para predecir la categoría de una imagen determinada del conjunto de datos CIFAR-10. Luego, mejoramos esta arquitectura introduciendo capas convolucionales, de normalización por lotes y de abandono para crear una red neuronal convolucional (CNN).

Un punto realmente importante que debemos aprender de este capítulo es que las redes neuronales profundas son completamente flexibles por diseño y realmente no existen reglas fijas cuando se trata de arquitectura de modelos. Existen pautas y mejores prácticas, pero no dude en experimentar con las capas y el orden en que aparecen. ¡No se sienta obligado a utilizar únicamente las arquitecturas sobre las que ha leído en este libro o en otros lugares! Como un niño con un conjunto de bloques de construcción, el diseño de su red neuronal sólo está limitado por su propia imaginación.

En el próximo capítulo, veremos cómo podemos utilizar estos componentes básicos para diseñar una red que pueda generar imágenes.

1. Kaiming He et al., "Aprendizaje residual profundo para el reconocimiento de imágenes", 10 de diciembre de 2015, <https://arxiv.org/abs/1512.03385>.

2. Alex Krizhevsky, “Aprendiendo múltiples capas de funciones desde imágenes minúsculas”, 8 de abril de 2009, <https://www.cs.toronto.edu/~kriz/learningfeatures-2009-TR.pdf>.
3. Diederik Kingma y Jimmy Ba, “Adam: A Method for Stochastic Optimization”, 22 de diciembre de 2014, <https://arxiv.org/abs/1412.6980v8>.
4. Samuel L. Smith et al., “No disminuya la tasa de aprendizaje, aumente el tamaño de lote”, 1 de noviembre de 2017, <https://arxiv.org/abs/1711.00489>.
5. Vincent Dumoulin y Francesco Visin, “Una guía para la convolución aritmética para el aprendizaje profundo”, 12 de enero de 2018, <https://arxiv.org/abs/1603.07285>.
6. Sergey Ioffe y Christian Szegedy, “Normalización por lotes: acelerando el entrenamiento en redes profundas mediante la reducción del cambio de covariables internas”, febrero 11, 2015, <https://arxiv.org/abs/1502.03167>.
7. Hinton et al., “Redes mediante la prevención de la coadaptación de detectores de características”, 3 de julio de 2012, <https://arxiv.org/abs/1207.0580>.
8. Nitish Srivastava et al., “Abandono: una forma sencilla de prevenir el sobreajuste en redes neuronales”, Journal of Machine Learning Research 15 (2014): 1929–1958, <http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf>.

Parte II. Métodos

En la Parte II nos sumergiremos en las seis familias de modelos generativos, incluida la teoría detrás de cómo funcionan y ejemplos prácticos de cómo construir cada tipo de modelo.

En el Capítulo 3 veremos nuestro primer modelo generativo de aprendizaje profundo, el auto codificador variacional. Esta técnica nos permitirá no sólo generar rostros realistas, sino también alterar imágenes existentes, por ejemplo, agregando una sonrisa o cambiando el color del cabello de alguien.

El capítulo 4 explora una de las técnicas de modelado generativo más exitosas de los últimos años, la red generativa adversarial. Veremos las formas en que el entrenamiento GAN se ha perfeccionado y adaptado para ampliar continuamente los límites de lo que el modelado generativo es capaz de lograr.

En el Capítulo 5 profundizaremos en varios ejemplos de modelos autorregresivos, incluidos LSTM y PixelCNN.

Esta familia de modelos trata el proceso de generación como un problema de predicción de secuencia: sustenta los modelos de generación de texto de última generación actuales y también se puede utilizar para la generación de imágenes.

En el Capítulo 6 cubriremos la familia de modelos de flujo de normalización, incluido RealNVP. Este modelo se basa en una fórmula de cambio de variables, que permite la transformación de una distribución simple, como una distribución gaussiana, en una distribución más compleja de manera que se preserve la manejabilidad.

El capítulo 7 presenta la familia de modelos basados en energía.

Estos modelos entranan una función de energía escalar para calificar la validez de una entrada determinada. Exploraremos una técnica para

entrenar modelos basados en energía llamada divergencia contrastiva y una técnica para muestrear nuevas observaciones llamada Dinámica de Langevin.

Finalmente, en el Capítulo 8 exploraremos la familia de modelos de difusión. Esta técnica se basa en la idea de agregar ruido de forma iterativa a una imagen y luego entrenar un modelo para eliminar el ruido, lo que nos brinda la capacidad de transformar ruido puro en muestras realistas.

Al final de la Parte II, habrá creado ejemplos prácticos de modelos generativos de cada una de las seis familias de modelos generativos y podrá explicar cómo funciona cada uno desde una perspectiva teórica.

Capítulo 3. Codificadores automáticos variacionales

METAS DEL CAPÍTULO

En este capítulo podrás:

- Descubrir cómo el diseño arquitectónico de los codificadores automáticos los hace perfectamente adecuados para el modelado generativo.
- Construir y entrenar un auto codificador desde cero usando Keras.
- Utilizar codificadores automáticos para generar nuevas imágenes, pero comprendiendo las limitaciones de este enfoque.
- Conocer la arquitectura del auto codificador variacional y cómo resuelve muchos de los problemas asociados con los codificadores automáticos estándar.
- Construir un auto codificador variacional desde cero usando Keras.
- Utilizar codificadores automáticos variacionales para generar nuevas imágenes.
- Utilizar codificadores automáticos variacionales para manipular imágenes generadas utilizando aritmética de espacio latente.

En 2013, Diederik P. Kingma y Max Welling publicaron un artículo que sentó las bases para un tipo de red neuronal conocida como auto codificador variacional (VAE).¹ Esta es ahora una de las arquitecturas de aprendizaje profundo más fundamentales y conocidas para la generación, modelado y un excelente lugar para comenzar nuestro viaje hacia el aprendizaje profundo generativo.

En este capítulo, comenzaremos construyendo un auto codificador estándar y luego veremos cómo podemos extender este marco para desarrollar un auto codificador variacional. A lo largo del camino, separaremos ambos tipos de modelos para comprender cómo funcionan a nivel granular. Para el final del capítulo, debe tener una comprensión completa de cómo construir y

manipular modelos basados en codificadores automáticos y, en particular, cómo construir un auto codificador variacional desde cero para generar imágenes basadas en su propio conjunto de datos.

Introducción

Comencemos con una historia sencilla que ayudará a explicar el problema fundamental que un auto codificador intenta resolver.

BRIAN, EL STITCH Y EL ARMARIO

Imagina que en el suelo, frente a ti, hay un montón de toda tu ropa: pantalones, blusas, zapatos y abrigos, todos de diferentes estilos. Tu estilista, Brian, está cada vez más frustrado por el tiempo que le lleva encontrar los artículos que necesitas, por lo que diseña un plan inteligente.

Te dice que organices tu ropa en un armario infinitamente alto y ancho (Figura 3-1). Cuando quieras solicitar un artículo en particular, simplemente debes decirle a Brian su ubicación y él lo coserá desde cero usando su confiable máquina de coser. Pronto resulta obvio que necesitarás colocar elementos similares uno cerca del otro, para que Brian pueda recrear con precisión cada elemento teniendo en cuenta solo su ubicación.



Figura 3-1. Un hombre parado frente a un guardarropa infinito en 2D (creado con Midjourney)

Después de varias semanas de práctica, Brian y tú os habéis adaptado a la comprensión mutua sobre la distribución del vestuario. Ahora es posible que le cuentes a Brian la ubicación de cualquier prenda de vestir que desee, ¡y él puede coserla con precisión desde cero!

Esto te da una idea: ¿qué pasaría si le dieras a Brian, una ubicación del guardarropa que estaba vacía? Para tu sorpresa, descubres que Brian es capaz de generar prendas de vestir completamente nuevas que no existían

antes. El proceso no es perfecto, pero ahora tienes opciones ilimitadas para generar ropa nueva, simplemente eligiendo un lugar vacío en el armario infinito y dejando que Brian haga su magia con la máquina de coser.

Exploraremos ahora cómo se relaciona esta historia con la creación de codificadores automáticos.

Auto codificadores

Un diagrama del proceso descrito por la historia se muestra en la figura 3-2. Usted desempeña el papel del codificador, moviendo cada prenda de vestir a un lugar en el armario. Este proceso se llama codificación. Brian desempeña el papel del decodificador, ocupa un lugar en el armario e intenta recrear el artículo. Este proceso se llama decodificación.

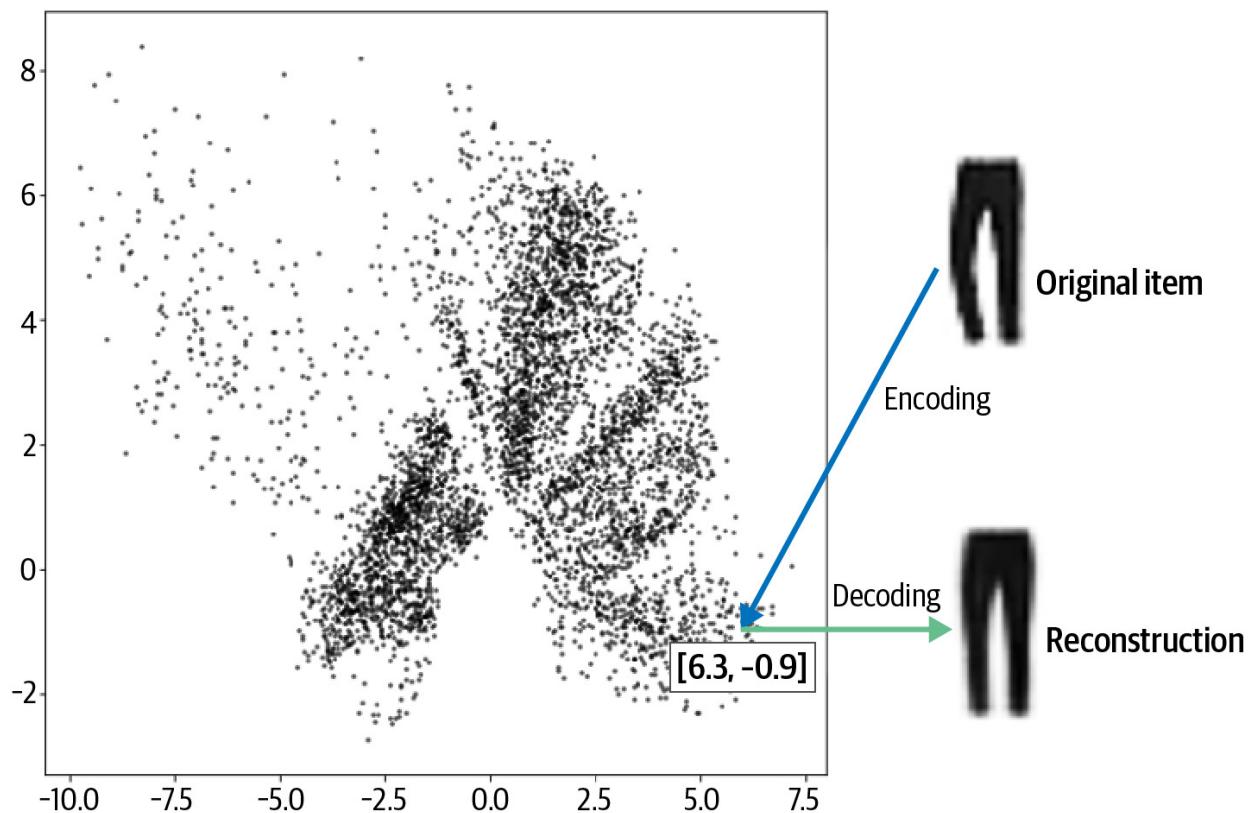


Figura 3-2. Prendas de vestir en el guardarropa infinito: cada punto negro representa una prenda de vestir

Cada ubicación del armario está representada por dos números (es decir, un vector 2D). Por ejemplo, los pantalones en la Figura 3-2 está codificada hasta el punto [6.3, -0.9]. Este vector también se conoce como incrustación porque el codificador intenta incrustar en él tanta información como sea posible, para que el decodificador pueda producir una reconstrucción precisa.

Un auto codificador es simplemente una red neuronal entrenada para realizar la tarea de codificar y decodificar un elemento, de modo que el resultado de este proceso sea lo más parecido posible al elemento original. Fundamentalmente, se puede utilizar como modelo generativo, porque podemos decodificar cualquier punto en el espacio 2D que queramos (en particular, aquellos que no son incrustaciones de artículos originales) para producir una prenda de vestir novedosa.

¡Veamos ahora cómo podemos construir un auto codificador usando Keras y aplícalo a un conjunto de datos real!

EJECUTANDO EL CÓDIGO PARA ESTE EJEMPLO

El código de este ejemplo se puede encontrar en el cuaderno de Jupyter ubicado en notebooks/03_vae/01_autoencoder/autoencoder.ipynb en el repositorio de libros.

El conjunto de datos Fashion-MNIST

Para este ejemplo, usaremos el conjunto de datos Fashion-MNIST, una colección de imágenes en escala de grises de prendas de vestir, cada una de un tamaño de 28×28 píxeles. En la Figura 3-3 se muestran algunas imágenes de ejemplo del conjunto de datos.



Figura 3-3. Ejemplos de imágenes del conjunto de datos Fashion-MNIST

El conjunto de datos viene empaquetado con TensorFlow, por lo que se puede descargar como se muestra en el Ejemplo 3-1.

Ejemplo 3-1. Cargando el conjunto de datos Fashion-MNIST

```
from tensorflow.keras import datasets  
(x_train,y_train), (x_test,y_test) =  
datasets.fashion_mnist.load_data()
```

Estas son imágenes en escala de grises de 28×28 (valores de píxeles entre 0 y 255) listos para usar, que debemos preprocesar para garantizar que los valores de píxeles estén escalados entre 0 y 1.

También rellenaremos cada imagen a 32×32 para facilitar la manipulación de la forma del tensor a medida que pasa a través de la red, como se muestra en el Ejemplo 3-2.

Ejemplo 3-2. Preprocesamiento de los datos

```
def preprocess(imgs):  
    imgs = imgs.astype("float32") / 255.0  
    imgs = np.pad(imgs, ((0, 0), (2, 2), (2, 2)),  
    constant_values=0.0)  
    imgs = np.expand_dims(imgs, -1)  
    return imgs  
  
x_train = preprocess(x_train)  
x_test = preprocess(x_test)
```

A continuación, debemos comprender la estructura general de un auto codificador para poder codificarlo usando TensorFlow y Keras.

La arquitectura del auto codificador

Un auto codificador es una red neuronal compuesta por dos partes:

- Una red codificadora que comprime datos de entrada de alta dimensión, como una imagen, en un vector de incrustación de menor dimensión.
- Una red decodificadora que descomprime un vector de incrustación determinado de regreso al dominio original (por ejemplo, de regreso a una imagen)

Un diagrama de la arquitectura de la red se muestra en la figura 3-4. Una imagen de entrada se codifica en un vector de incrustación latente z , que luego se decodifica nuevamente al espacio de píxeles original.

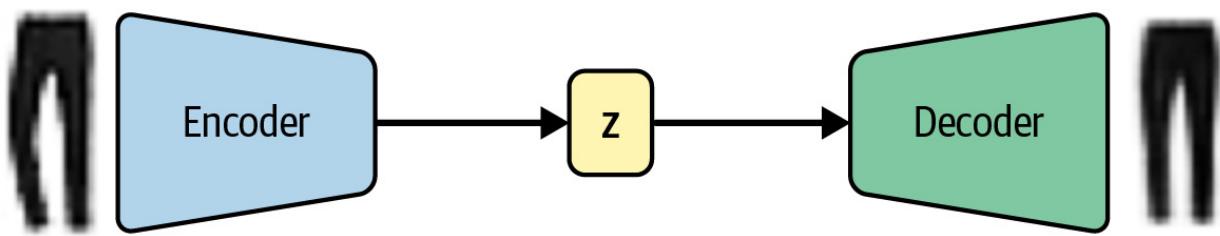


Figura 3-4. Diagrama de arquitectura del auto codificador

El auto codificador está entrenado para reconstruir una imagen, después de que haya pasado por el codificador y haya regresado a través del decodificador. Esto puede parecer extraño al principio: ¿por qué querías reconstruir un conjunto de imágenes que ya tienes disponible para ti? Sin embargo, como veremos, es el espacio de incrustación (también llamado espacio latente) la parte interesante del auto codificador, ya que el muestreo de este espacio nos permitirá generar nuevas imágenes.

Primero definamos qué entendemos por incrustación. La incrustación (z) es una compresión de la imagen original en un espacio latente de dimensiones inferiores. La idea es que al elegir cualquier punto en el espacio latente, podemos generar imágenes novedosas al pasar este punto a través del

decodificador, ya que el decodificador ha aprendido cómo convertir puntos en el espacio latente en imágenes viables.

En nuestro ejemplo, incrustaremos imágenes en un espacio latente bidimensional. Esto nos ayudará a visualizar el espacio latente, ya que podemos trazar puntos fácilmente en 2D. En la práctica, el espacio latente de un auto codificador suele tener más de dos dimensiones para tener más libertad para capturar mayores matices en las imágenes.

AUTOENCODIFICADORES COMO MODELOS ELIMINADORES DE RUIDO

Los codificadores automáticos se pueden utilizar para limpiar imágenes ruidosas, ya que el codificador aprende que no es útil capturar la posición del ruido aleatorio dentro del espacio latente para reconstruir el original. Para tareas como esta, un espacio latente 2D probablemente sea demasiado pequeño para codificar suficiente información relevante a partir de la entrada. Sin embargo, como veremos, aumentar la dimensionalidad del espacio latente rápidamente genera problemas si queremos utilizar el auto codificador como modelo generativo.

Veamos ahora cómo construir el codificador y el decodificador.

El codificador

En un auto codificador, el trabajo del codificador es tomar la imagen de entrada y asignarla a un vector de incrustación en el espacio latente. La arquitectura del codificador que construiremos se muestra en la Tabla 3-1.

Capa (tipo)	Forma de salida	# de parámetros
Capa de entrada	(Ninguno, 32, 32, 1)	
Conv2D	(Ninguno, 16, 16, 32)	320
Conv2D	(Ninguno, 8, 8, 64)	18,496
Conv2D	(Ninguno, 4, 4, 128)	73,856
Aplanar	(Ninguno, 2048)	
Denso	(Ninguno, 2)	4,098

Parámetros totales	96,770
Parámetros entrenables	96,770
Parámetros no entrenables	

Para lograr esto, primero creamos una capa de Entrada para la imagen y la pasamos a través de tres capas Conv2D en secuencia, cada una de los cuales captura características de nivel cada vez más alto.

Usamos una zancada de 2 para reducir a la mitad el tamaño de la salida de cada capa, mientras aumentamos el número de canales. La última capa convolucional se aplana y se conecta a una capa Dense de tamaño 2, que representa nuestro espacio latente bidimensional.

El ejemplo 3-3 muestra cómo construir esto en Keras.

Ejemplo 3-3. El codificador

```
encoder_input = layers.Input(
    shape=(32, 32, 1), name = "encoder_input"
)
x = layers.Conv2D(32, (3, 3), strides = 2, activation = 'relu',
padding="same")(encoder_input)
x = layers.Conv2D(64, (3, 3), strides = 2, activation = 'relu',
padding="same")(x)
x = layers.Conv2D(128, (3, 3), strides = 2, activation = 'relu',
padding="same")(x)
shape_before_flattening = K.int_shape(x)[1:]

x = layers.Flatten()(x)
encoder_output = layers.Dense(2, name="encoder_output")(x)
encoder = models.Model(encoder_input, encoder_output)
```

1. Defina la capa de entrada del codificador (la imagen).
2. Apile las capas Conv2D secuencialmente una encima de la otra.
3. Aplane la última capa convolucional a un vector.
4. Conecte este vector a las incrustaciones 2D con una capa Dense
5. El modelo Keras que define el codificador: un modelo que toma una imagen de entrada y la codifica en una incrustación 2D.

CONSEJO

Le recomiendo encarecidamente que experimente con la cantidad de capas y filtros convolucionales para comprender cómo la arquitectura afecta la cantidad total de parámetros del modelo, el rendimiento del modelo y el tiempo de ejecución del modelo.

El decodificador

El decodificador es una imagen espectral del codificador; en lugar de capas convolucionales, utilizamos capas de transposición convolucionales, como se muestra en la Tabla 3-2.

Capa (tipo)	Forma de salida	# de Parámetros
Capa de entrada	(Ninguno, 2)	
Denso	(Ninguno, 2048)	6,144
Remodelar	(Ninguno, 4, 4, 128)	
Conv2DTransponer	(Ninguno, 8, 8, 128)	147,584
Conv2DTransponer	(Ninguno, 16, 16, 64)	73,792
Conv2DTransponer	(Ninguno, 32, 32, 32)	18,464
Conv2D	(Ninguno, 32, 32, 1)	289
Parámetros totales		246,273
Parámetros entrenables		246,273
Parámetros no entrenables		

CAPAS DE TRANSPOSICIÓN CONVOLUCIONAL

Las capas convolucionales estándar nos permiten reducir a la mitad el tamaño de un tensor de entrada en ambas dimensiones (alto y ancho), estableciendo `strides = 2`.

La capa de transposición convolucional utiliza el mismo principio que una capa convolucional estándar (pasar un filtro a través de la imagen), pero se

diferencia en que establecer `strides = 2` duplica el tamaño del tensor de entrada en ambas dimensiones.

En una capa de transposición convolucional, el parámetro de pasos determina el relleno de ceros interno entre los píxeles de la imagen, como se muestra en la Figura 3-5.

Aquí, se pasa un filtro de $3 \times 3 \times 1$ (gris) a través de una imagen de $3 \times 3 \times 1$ (azul) con pasos = 2, para producir una imagen de $6 \times$

Tensor de salida 6×1 (verde).

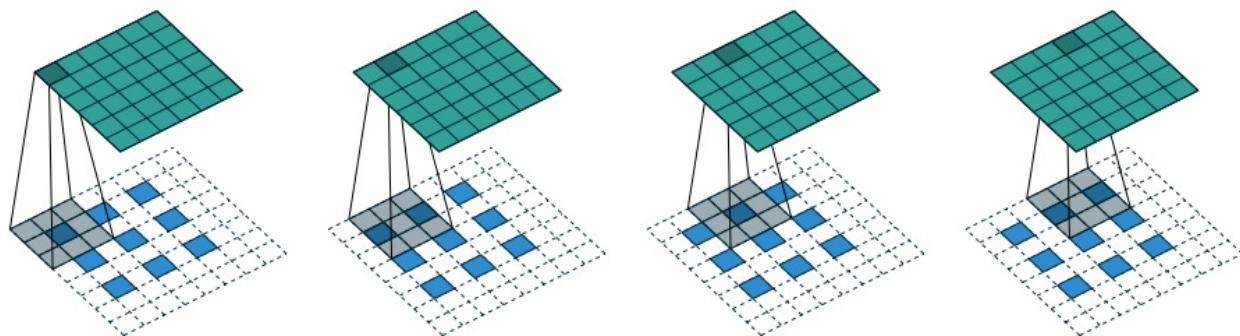


Figura 3-5. Un ejemplo de capa de transposición convolucional (fuente: Dumoulin y Visín, 2018)2

En Keras, la capa `Conv2DTranspose` nos permite realizar operaciones de transposición convolucional en tensores.

Al apilar estas capas, podemos expandir gradualmente el tamaño de cada capa, usando pasos de 2, hasta volver a la dimensión de la imagen original de 32×32 .

El ejemplo 3-4 muestra cómo construimos el decodificador en Keras.

Ejemplo 3-4. El decodificador

```
decoder_input = layers.Input(shape=(2,), name="decoder_input")
x = layers.Dense(np.prod(shape_before_flattening))
(decoder_input)
x = layers.Reshape(shape_before_flattening)(x)
x = layers.Conv2DTranspose(
    128, (3, 3), strides=2, activation = 'relu', padding="same")
```

```

)(x)
x = layers.Conv2DTranspose(
    64, (3, 3), strides=2, activation = 'relu', padding="same"
)(x)
x = layers.Conv2DTranspose(
    32, (3, 3), strides=2, activation = 'relu', padding="same"
)(x)
decoder_output = layers.Conv2D(
    1,
    (3, 3),
    strides = 1,
    activation="sigmoid",
    padding="same",
    name="decoder_output"
)(x)

decoder = models.Model(decoder_input, decoder_output)

```

1. Defina la capa de entrada del decodificador (la incrustación).
2. Conecte la entrada a una capa Dense.
3. Transforme este vector en un tensor que pueda introducirse como entrada en la primera capa Conv2DTranspose.
4. Apile las capas Conv2DTranspose una encima de la otra.
5. El modelo Keras que define el decodificador: un modelo que toma una incrustación en el espacio latente y la decodifica en el dominio de la imagen original.

Uniendo el codificador al decodificador

Para entrenar el codificador y el decodificador simultáneamente, necesitamos definir un modelo que represente el flujo de una imagen a través del codificador y regresa a través del decodificador.

Afortunadamente, Keras hace que esto sea extremadamente fácil, como puede ver en el Ejemplo 3-5. Observe la forma en que especificamos que la salida del auto codificador es simplemente la salida del codificador después de haber pasado por el decodificador.

Ejemplo 3-5. El auto codificador completo

```
autoencoder = Model(encoder_input, decoder(encoder_output))
```

1. El modelo Keras que define el auto codificador completo: un modelo que toma una imagen y la pasa a través del codificador y vuelve a pasar por el decodificador para generar una reconstrucción de la imagen original.

Ahora que hemos definido nuestro modelo, solo necesitamos compilarlo con una función de pérdida y un optimizador, como se muestra en el ejemplo 3-6. La función de pérdida generalmente se elige para que sea el error cuadrático medio (RMSE) o la entropía cruzada binaria entre los píxeles individuales de la imagen original y la reconstrucción.

Ejemplo 3-6. Compilando el auto codificador

```
# Compile the autoencoder
autoencoder.compile(optimizer="adam",
loss="binary_crossentropy")
```

ELEGIR LA FUNCIÓN DE PÉRDIDA

La optimización para RMSE significa que la salida generada se distribuirá simétricamente alrededor de los valores promedio de píxeles (porque una sobreestimación se penaliza de manera equivalente a una subestimación).

Por otro lado, la pérdida de entropía cruzada binaria es asimétrica: penaliza los errores hacia los extremos más que los errores hacia el centro. Por ejemplo, si el valor real del píxel es alto (por ejemplo, 0,7), generar un píxel con un valor de 0,8 se penaliza más que generar un píxel con un valor de 0,6. Si el valor real del píxel es bajo (por ejemplo, 0,3), generar un píxel con un valor de 0,2 se penaliza más que generar un píxel con un valor de 0,4.

Esto tiene el efecto de que la pérdida de entropía cruzada binaria produce imágenes ligeramente más borrosas que la pérdida de RMSE (ya que tiende a impulsar las predicciones hacia 0,5), pero a veces esto es deseable ya que RMSE puede generar bordes obviamente pixelizados.

No existe una elección correcta o incorrecta; debe elegir la que funcione mejor para su caso de uso después de experimentar.

Ahora podemos entrenar el auto codificador pasando las imágenes de entrada como entrada y salida, como se muestra en el ejemplo 3-7.

Ejemplo 3-7. Entrenando el auto codificador

```
autoencoder.fit(  
    x_train,  
    x_train,  
    epochs=5,  
    batch_size=100,  
    shuffle=True,  
    validation_data=(x_test, x_test),  
)
```

Ahora que nuestro auto codificador está entrenado, lo primero que debemos verificar es que sea capaz de reconstruir con precisión las imágenes de entrada.

Reconstruyendo imágenes

Podemos probar la capacidad de reconstruir imágenes pasando imágenes del conjunto de prueba a través del auto codificador y comparando la salida con las imágenes originales. El código para esto se muestra en el Ejemplo 3-8.

Ejemplo 3-8. Reconstrucción de imágenes utilizando el auto codificador

```
example_images = x_test[:5000]  
predictions = autoencoder.predict(example_images)
```

En la Figura 3-6 puede ver algunos ejemplos de imágenes originales (fila superior), los vectores 2D después de la codificación y los elementos reconstruidos después de la decodificación (fila inferior).

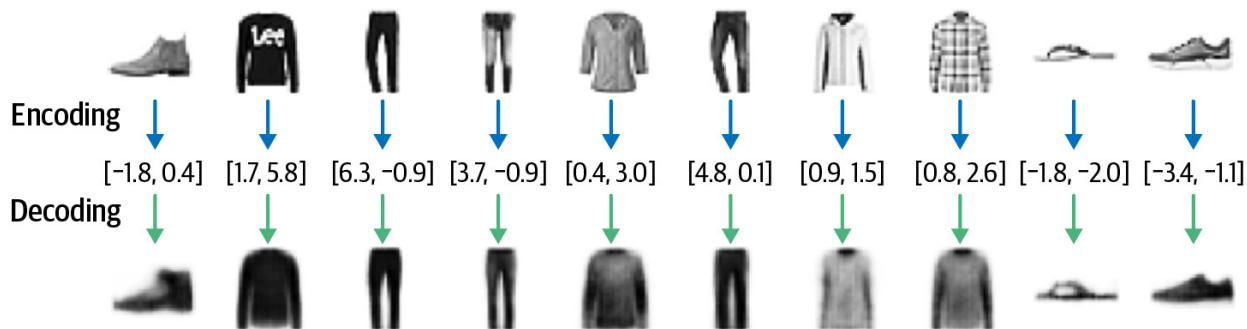


Figura 3-6. Ejemplos de codificación y decodificación de prendas de vestir.

Observe cómo la reconstrucción no es perfecta: todavía hay algunos detalles de las imágenes originales que no se capturan en el proceso de decodificación, como los logotipos. Esto se debe a que al reducir cada imagen a solo dos números, naturalmente perdemos algo de información.

Investigaremos ahora cómo el codificador representa imágenes en el espacio latente.

Visualizando el espacio latente

Podemos visualizar cómo se incrustan las imágenes en el espacio latente pasando el conjunto de prueba a través del codificador y trazando las incrustaciones resultantes, como se muestra en el Ejemplo 39.

Ejemplo 3-9. Incrustar imágenes usando el codificador

```
embeddings = encoder.predict(example_images)
plt.figure(figsize=(8, 8))
plt.scatter(embeddings[:, 0], embeddings[:, 1], c="black",
alpha=0.5, s=3)
plt.show()
```

El gráfico resultante es el gráfico de dispersión que se muestra en la Figura 3-2: cada punto negro representa una imagen que se ha incrustado en el espacio latente.

Para comprender mejor cómo está estructurado este espacio latente, podemos utilizar las etiquetas que vienen con el conjunto de datos Fashion-MNIST, que describen el tipo de artículo en cada imagen. Hay 10 grupos en total, que se muestran en la Tabla 3-3.

IDENTIFICACIÓN	Etiqueta de ropa
Camiseta/top	
Pantalón	
Pull-over	
Vestido	

Abrigo
Sandalia
Camisa
Zapatilla
Bolsa
Botín

Podemos colorear cada punto según la etiqueta de la imagen correspondiente para producir el gráfico en la Figura 3-7.

¡Ahora la estructura se vuelve muy clara! Aunque las etiquetas de la ropa nunca se le mostraron al modelo durante el entrenamiento, el auto codificador ha agrupado naturalmente elementos que parecen iguales en las mismas partes del espacio latente. Por ejemplo, la nube de puntos azul oscuro en la esquina inferior derecha del espacio latente son imágenes diferentes de pantalones y la nube roja de puntos hacia el centro son botines.

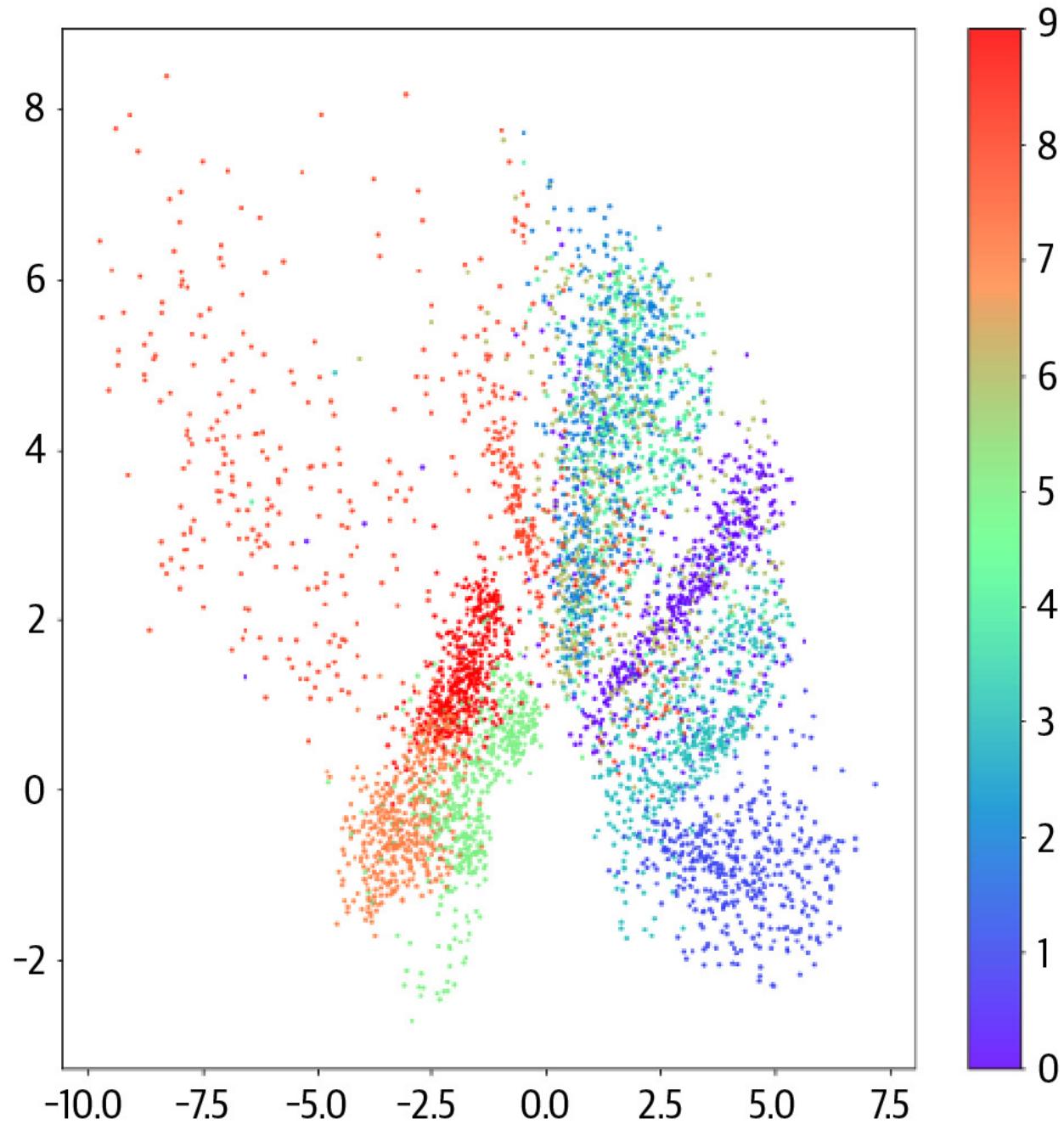


Figura 3-7. Trama del espacio latente, coloreada por la etiqueta de la ropa.

Generando nuevas imágenes

Podemos generar imágenes novedosas muestreando algunos puntos en el espacio latente y usando el decodificador para convertirlos nuevamente en espacio de píxeles, como se muestra en el Ejemplo 3-10.

Ejemplo 3-10. Generando imágenes novedosas usando el decodificador

```
mins, maxs = np.min(embeddings, axis=0), np.max(embeddings, axis=0)
sample = np.random.uniform(mins, maxs, size=(18, 2))
reconstructions = decoder.predict(sample)
```

Algunos ejemplos de imágenes generadas se muestran en la Figura 3-8, junto con sus incrustaciones en el espacio latente.

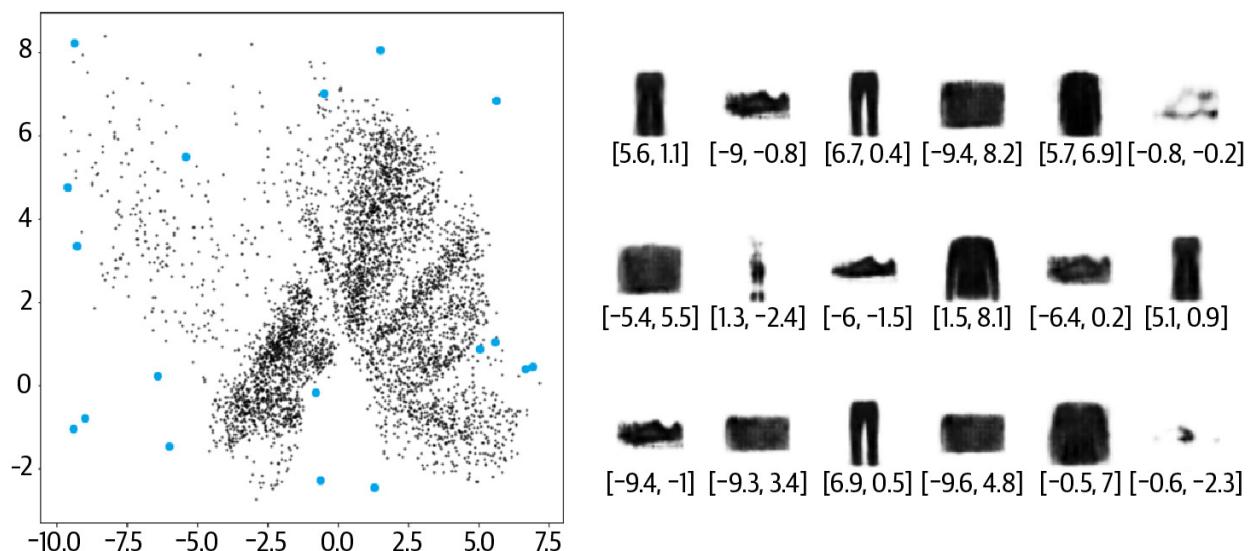


Figura 3-8. Prendas de vestir generadas

Cada punto azul se asigna a una de las imágenes que se muestran a la derecha del diagrama, con el vector de incrustación que se muestra debajo. Observe cómo algunos de los elementos generados son más realistas que otros. ¿Por qué es esto?

Para responder a esto, primero hagamos algunas observaciones sobre la distribución general de puntos en el espacio latente, volviendo a la Figura 3-7:

Algunas prendas de vestir están representadas en un área muy pequeña y otras en un área mucho más grande.

La distribución no es simétrica con respecto al punto (0, 0), o acotado. Por ejemplo, hay muchos más puntos con valores positivos en el eje y que

negativos, y algunos puntos incluso se extienden hasta un valor del eje y > 8.

Hay grandes espacios entre colores que contienen pocos puntos.

En realidad, estas observaciones hacen que el muestreo del espacio latente sea bastante desafiante. Si superponemos el espacio latente con imágenes de puntos decodificados en una cuadrícula, como se muestra en la Figura 3-9, podemos comenzar a comprender por qué el decodificador no siempre genera imágenes con un estándar satisfactorio.

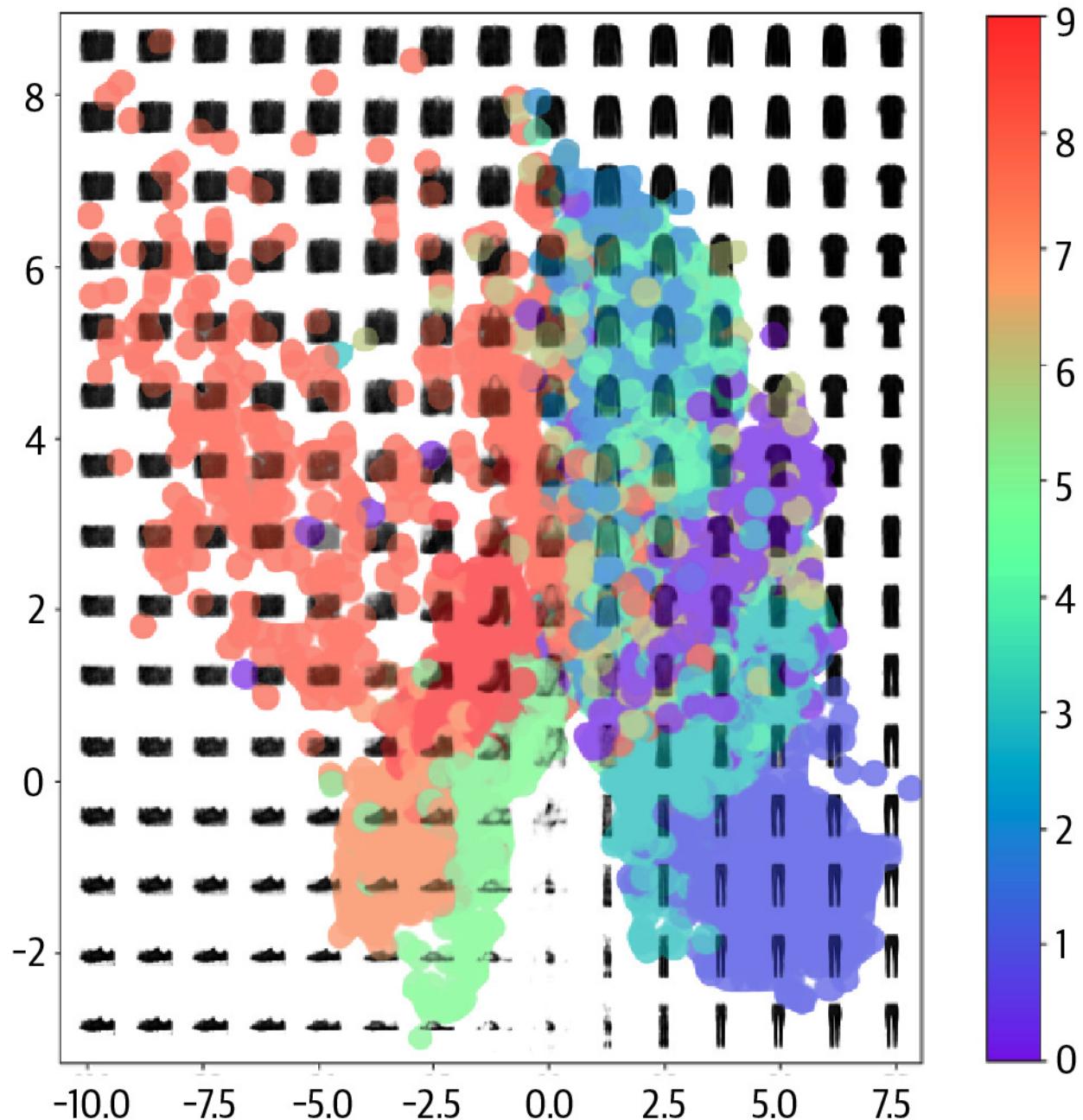


Figura 3-9. Una cuadrícula de incrustaciones decodificadas, superpuestas con las incrustaciones de las imágenes originales en el conjunto de datos, coloreadas por tipo de elemento.

En primer lugar, podemos ver que si seleccionamos puntos uniformemente en un espacio acotado que definimos, es más probable que muestreemos algo que se decodifica para parecerse a una bolsa (ID 8) que a un botín (ID 9),

porque la parte de el espacio latente tallado para los bolsos (naranja) es mayor que la zona del botín (rojo).

En segundo lugar, no es obvio cómo deberíamos elegir un punto aleatorio en el espacio latente, ya que la distribución de estos puntos no está definida. ¡Técnicamente, estaríamos justificados al elegir cualquier punto en el plano 2D!

Ni siquiera está garantizado que los puntos se centren en $(0, 0)$. Esto hace que el muestreo de nuestro espacio latente sea problemático.

Por último, podemos ver agujeros en el espacio latente donde ninguna de las imágenes originales está codificada. Por ejemplo, hay grandes espacios en blanco en los bordes del dominio; el auto codificador no tiene motivos para garantizar que los puntos aquí se decodifiquen en prendas de vestir reconocibles, ya que aquí se codifican muy pocas imágenes del conjunto de entrenamiento.

Incluso los puntos que son centrales pueden no ser decodificados en imágenes bien formadas. Esto se debe a que el auto codificador no está obligado a garantizar que el espacio sea continuo. Por ejemplo, aunque el punto $(-1, -1)$ podría decodificarse para dar una imagen satisfactoria de una sandalia, no existe ningún mecanismo para garantizar que el punto $(-1.1, -1.1)$ también produzca una imagen satisfactoria de una sandalia.

En dos dimensiones esta cuestión es sutil; el auto codificador solo tiene una pequeña cantidad de dimensiones con las que trabajar, por lo que, naturalmente, tiene que aplastar los grupos de ropa, lo que resulta en que el espacio entre los grupos de ropa sea relativamente pequeño. Sin embargo, a medida que comenzamos a utilizar más dimensiones en el espacio latente para generar imágenes más complejas, como rostros, este problema se vuelve aún más evidente. Si le damos rienda suelta al auto codificador sobre cómo utiliza el espacio latente para codificar imágenes, habrá enormes brechas entre grupos de puntos similares sin ningún incentivo para que los espacios intermedios generen imágenes bien formadas.

Para resolver estos tres problemas, necesitamos convertir nuestro auto codificador en un auto codificador variacional.

Autocodificadores variacionales

Para explicarlo, revisemos el guardarropa infinito y hagamos algunos cambios...

REVISITANDO EL ARMARIO INFINITO

Supongamos ahora que, en lugar de colocar cada prenda de vestir en un solo punto del armario, decides asignar un área general donde es más probable que se encuentre la prenda.

Considera que este enfoque más relajado en la ubicación de las prendas ayudará a resolver el problema actual relacionado con las discontinuidades locales en el guardarropa.

Además, para garantizar que no seas demasiado descuidado con el nuevo sistema de colocación, estás de acuerdo con Brian en que intentarás colocar el centro del área de cada prenda lo más cerca posible del centro del armario y que la desviación de la prenda desde el centro debe estar lo más cerca posible de un metro (ni más pequeño ni más grande). Cuanto más te alejes de esta regla, más tendrás que pagarle a Brian como estilista.

Después de varios meses de operar con estos dos cambios simples, das un paso atrás y admirás el nuevo diseño del guardarropa, junto con algunos ejemplos de nuevas prendas que Brian ha generado. ¡Mucho mejor! Hay mucha diversidad en los artículos generados, y esta vez no hay ejemplos de prendas de mala calidad. ¡Parece que los dos cambios han marcado la diferencia!

Intentemos ahora comprender qué debemos hacer con nuestro modelo de auto codificador para convertirlo en un auto codificador variacional y así convertirlo en un modelo generativo más sofisticado.

Las dos partes que necesitamos cambiar son el codificador y la función de pérdida.

El codificador

En un auto codificador, cada imagen se asigna directamente a un punto del espacio latente. En un auto codificador variacional, cada imagen se asigna a una distribución normal multivariada alrededor de un punto en el espacio latente, como se muestra en la figura 3-10.

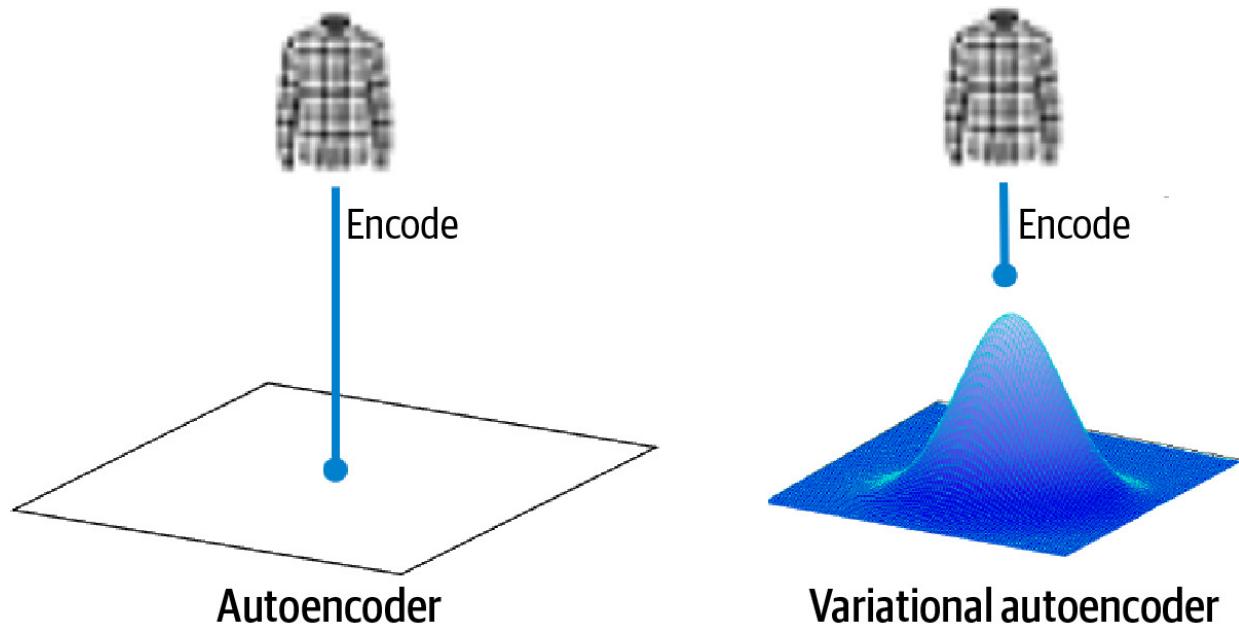


Figura 3-10. La diferencia entre los codificadores en un auto codificador y un auto codificador variacional

LA DISTRIBUCIÓN NORMAL MULTIVARIADA

Una distribución normal (o distribución gaussiana) $N(\mu, \sigma)$ es una distribución de probabilidad caracterizada por una forma de curva de campana distintiva, definida por dos variables: la media (μ) y la varianza (σ^2). La desviación estándar (σ) es la raíz cuadrada de la varianza.

La función de densidad de probabilidad de la distribución normal en una dimensión es:

$$f(x | \mu, \sigma) =$$

$$\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

La Figura 3-11 muestra varias distribuciones normales en una dimensión, para diferentes valores de media y varianza. La curva roja es la normal estándar (o normal unitaria) $N(0, 1)$: la distribución normal con media igual a 0 y varianza igual a 1.

Podemos muestrear un punto z de una distribución normal con media μ y desviación estándar σ usando la siguiente ecuación:

$$z = \mu + \sigma\epsilon$$

donde ϵ se toma como muestra de una distribución normal estándar.

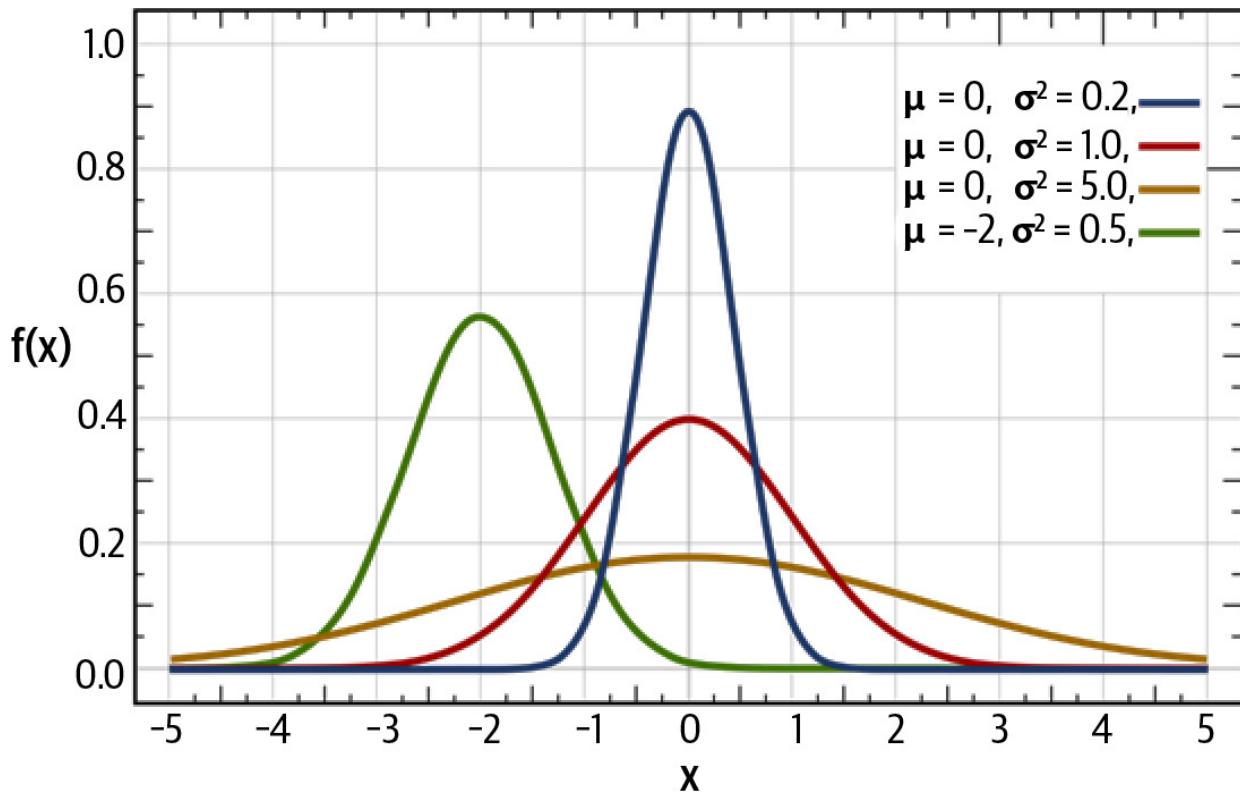


Figura 3-11. La distribución normal en una dimensión (fuente: Wikipedia)

El concepto de distribución normal se extiende a más de una dimensión: la función de densidad de probabilidad para una distribución normal multivariada (o Distribución gaussiana multivariada) $N(\mu, \Sigma)$ en k dimensiones con el vector medio μ y la matriz de covarianza simétrica Σ son los siguientes:

$\exp (-$

$(x-\mu)^{-1}$

$(x-\mu))$

$$f(x_1, \dots, x_k) = \sqrt{(2\pi)^k |\Sigma|}$$

En este libro, normalmente usaremos distribuciones normales multivariadas isotrópicas, donde la matriz de covarianza es diagonal. Esto significa que la distribución es independiente en cada dimensión (es decir, podemos muestrear un vector donde cada elemento se distribuye normalmente con media y varianza independientes). Este es el caso de la distribución normal multivariada que usaremos en nuestro auto codificador variacional.

Una distribución normal estándar multivariada $N(0, I)$ es una distribución multivariada con un vector medio de valor cero y una matriz de covarianza identidad.

NORMAL VERSUS GAUSSIANO

En este libro, los términos normal y gaussiano se usan indistintamente y generalmente se implica la naturaleza isotrópica y multivariada de la distribución. Por ejemplo, “tomamos muestras de una distribución gaussiana” puede interpretarse en el sentido de “muestreamos a partir de una distribución gaussiana isotrópica multivariada”.

El codificador solo necesita asignar cada entrada a un vector medio y un vector de varianza y no necesita preocuparse por la covarianza entre dimensiones. Los codificadores automáticos variacionales suponen que no existe correlación entre las dimensiones en el espacio latente.

Los valores de la varianza siempre son positivos, por lo que en realidad elegimos asignar el logaritmo de la varianza, ya que esto puede tomar cualquier número real en el rango $(-\infty, \infty)$. De esta manera, podemos utilizar una red neuronal como codificador para realizar el mapeo desde la imagen de entrada a los vectores de media y varianza logarítmica.

En resumen, el codificador tomará cada imagen de entrada y la codificará en dos vectores que juntos definen una distribución normal multivariada en el espacio latente:

z_{mean}

El punto medio de la distribución

$z_{\log \text{var}}$

El logaritmo de la varianza de cada dimensión.

Podemos muestrear un punto z de la distribución definida por estos valores usando la siguiente ecuación:

$$z = z_{\text{media}} + z_{\sigma} * \epsilon$$

donde: $z_{\sigma} = \exp(z_{\log \text{var}} * 0,5) \epsilon \approx N(0, I)$

CONSEJO

La derivación de la relación entre z_{σ} (σ) y $z_{\log \text{var}}$ ($\log(\sigma^2)$) es la siguiente:

$$\sigma = \exp(\log(\sigma)) = \exp(2 \log(\sigma)/2) = \exp(\log(\sigma)/2)$$

El decodificador de un auto codificador variacional es idéntico al decodificador de un auto codificador simple, lo que proporciona la arquitectura general que se muestra en la Figura 3-12.

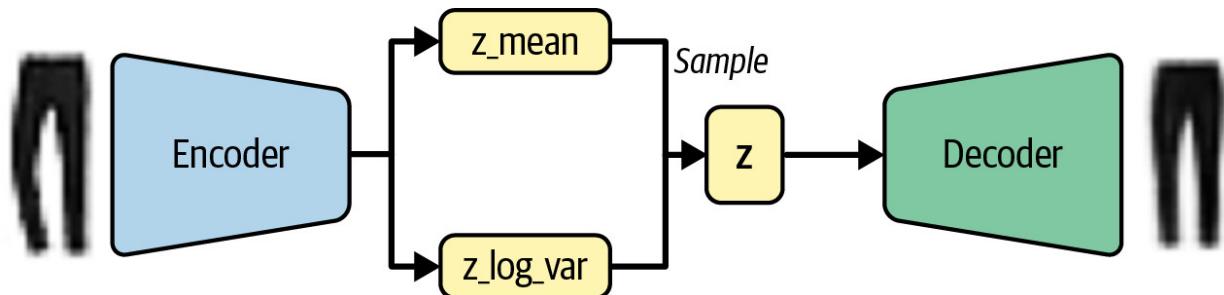


Figura 3-12. Diagrama de arquitectura VAE

¿Por qué ayuda este pequeño cambio en el codificador?

Anteriormente, vimos que no era necesario que el espacio latente fuera continuo; incluso si el punto $(-2, 2)$ se decodifica en una imagen bien formada de una sandalia, no es necesario que $(-2.1, 2.1)$ mire similar. Ahora, dado que estamos muestreando un punto aleatorio de un área alrededor de z_{mean} , el decodificador debe garantizar que todos los puntos en la misma vecindad produzcan imágenes muy similares cuando se decodifiquen, de modo que la pérdida de reconstrucción siga siendo pequeña. Esta es una propiedad muy interesante que garantiza que incluso cuando elegimos un punto en el espacio latente que nunca ha sido visto por el decodificador, es probable que se decodifique en una imagen bien formada.

Construyendo el codificador VAE

Veamos ahora cómo construimos esta nueva versión del codificador en Keras.

EJECUTANDO EL CÓDIGO PARA ESTE EJEMPLO

El código de este ejemplo se puede encontrar en el cuaderno de Jupyter ubicado en notebooks/03_vae/02_vae_fashion/vae_fashion.ipynb en el repositorio de libros.

El código ha sido adaptado del excelente tutorial VAE creado por Francois Chollet, disponible en el sitio web de Keras.

Primero, necesitamos crear un nuevo tipo de capa de muestreo que nos permita tomar muestras de la distribución definida por z_{mean} y $z_{\log_{\text{var}}}$, como se muestra en el Ejemplo 3-11.

Ejemplo 3-11. La capa de muestreo

```
class Sampling(layers.Layer):
    def call(self, inputs):
        z_mean, z_log_var = inputs
        batch = tf.shape(z_mean)[0]
        dim = tf.shape(z_mean)[1]
        epsilon = K.random_normal(shape=(batch, dim))
        return z_mean + tf.exp(0.5 * z_log_var) * epsilon
```

1. Creamos una nueva capa subclasicando la base de Keras.
2. Clase Layer (consulte la barra lateral "Subclases de la clase Layer").
3. Usamos el truco de reparametrización (ver la barra lateral "Truco de reparametrización") para construir una muestra a partir de la distribución normal parametrizada por `z_mean` y `z_log_var`.

SUBCLASIFICAR LA CLASE DE CAPA

Puede crear nuevas capas en Keras subclasicando la clase abstracta Layer y definiendo el método de llamada, que describe cómo la capa transforma un tensor.

Por ejemplo, en el auto codificador variacional, podemos crear una capa de muestreo que pueda manejar el muestreo de `z` de una distribución normal con parámetros definidos por `z_mean` y `z_log_var`.

Esto es útil cuando desea aplicar una transformación a un tensor que aún no está incluido como uno de los tipos de capa Keras listos para usar.

EL TRUCO DE LA REPARAMETERIZACIÓN

En lugar de tomar muestras directamente de una distribución normal con los parámetros `z_mean` y `z_log_var`, podemos tomar muestras `epsilon` de una normal estándar y luego ajustar manualmente la muestra para tener la media y la varianza correctas.

Esto se conoce como truco de reparametrización y es importante porque significa que los gradientes pueden retropropagarse libremente a través de la capa. Al mantener toda la aleatoriedad de la capa contenida dentro de la variable `epsilon`, se puede demostrar que la derivada parcial de la salida de la capa con respecto a su entrada es determinista (es decir, independiente del `epsilon` aleatorio), lo cual es esencial para la retropropagación a través de la capa para ser posible.

El código completo para el codificador, incluida la nueva capa de muestreo se muestra en el Ejemplo 3-12.

Ejemplo 3-12. El codificador

```

encoder_input = layers.Input(
    shape=(32, 32, 1), name="encoder_input"
)
x = layers.Conv2D(32, (3, 3), strides=2, activation="relu",
padding="same")(
    encoder_input
)
x = layers.Conv2D(64, (3, 3), strides=2, activation="relu",
padding="same")(x)
x = layers.Conv2D(128, (3, 3), strides=2, activation="relu",
padding="same")(x)
shape_before_flattening = K.int_shape(x)[1:]

x = layers.Flatten()(x)
z_mean = layers.Dense(2, name="z_mean")(x)
z_log_var = layers.Dense(2, name="z_log_var")(x)
z = Sampling()([z_mean, z_log_var])

encoder = models.Model(encoder_input, [z_mean, z_log_var, z],
name="encoder")

```

1. En lugar de conectar la capa `Flatten` directamente al espacio latente 2D, lo conectamos a las capas `z_mean` y `z_log_var`.
2. La capa de muestreo toma muestras de un punto `z` en el espacio latente a partir de la distribución normal definida por los parámetros `z_mean` y `z_log_var`.
3. El modelo Keras que define el codificador: un modelo que toma una imagen de entrada y genera `z_mean`, `z_log_var` y un punto de muestra `z` de la distribución normal definida por estos parámetros.

En la Tabla 3-4 se muestra un resumen del codificador.

Capa (tipo)	Forma de salida	# de parámetros	Conectado a
Capa de entrada (entrada)	(Ninguno, 32,32, 1)	0	[]
Conv2D (conv2d_1)	(Ninguno, 16,16, 32)	320	[input]
Conv2D (conv2d_2)	(Ninguno, 8, 8, 64)	18,496	[conv2d_1]
Conv2D (conv2d_3)	(Ninguno, 4, 4,	73,856	[conv2d_2]

	128)		
Flatten (flatten)	(Ninguno, 2048)	0	[conv2d_3]
Denso (z_media)	(Ninguno, 2)	4,098	[flatten]
Denso (z_log_var)	(Ninguno, 2)	4,098	[flatten]
Sampling (z)	(Ninguno, 2)	0	[z_media, z_log_var]
Parámetros totales	100,868		
Parámetros entrenables	100,868		
Parámetros no entrenables	0		

La única otra parte del auto codificador original que debemos cambiar es la función de pérdida.

La función de pérdida

Anteriormente, nuestra función de pérdida solo consistía en la pérdida de reconstrucción entre imágenes y sus intentos de copia después de pasar por el codificador y decodificador. La pérdida de reconstrucción también aparece en un auto codificador variacional, pero ahora necesitamos un componente adicional: el término de divergencia Kullback-Leibler (KL).

La divergencia KL es una forma de medir en qué medida una distribución de probabilidad difiere de otra. En un VAE, queremos medir en qué medida nuestra distribución normal con parámetros z_mean y z_log_var difiere de una distribución normal estándar. En este caso especial, se puede demostrar que la divergencia KL tiene la siguiente forma cerrada: $kl_loss = -0.5 * \sum(1 + z_log_var - z_mean^2 - \exp(z_log_var))$

o en notación matemática:

$$D_{KL} [\text{norte } (\mu, \sigma) \| \text{norte } (0, 1)] = -$$

$$\sum (1 + \log(\sigma) - \mu - \sigma^2)$$

La suma se toma sobre todas las dimensiones en el espacio latente. kl_loss se minimiza a 0 cuando $z_mean = 0$ y $z_log_var = 0$ para todas las

dimensiones. A medida que estos dos términos comienzan a diferir de 0, kl_loss aumenta.

En resumen, el término de divergencia KL penaliza a la red por codificar observaciones en variables z_mean y z_log_var que difieren significativamente de los parámetros de una distribución normal estándar, es decir, z_mean = 0 y z_log_var = 0.

¿Por qué ayuda esta adición a la función de pérdida?

En primer lugar, ahora tenemos una distribución bien definida que podemos usar para elegir puntos en el espacio latente: la distribución normal estándar. En segundo lugar, dado que este término intenta forzar todas las distribuciones codificadas hacia la distribución normal estándar, hay menos posibilidades de que se formen grandes brechas entre los grupos de puntos. En cambio, el codificador intentará utilizar el espacio alrededor del origen de forma simétrica y eficiente.

En el artículo original de VAE, la función de pérdida para un VAE era simplemente la suma de la pérdida de reconstrucción y el término de pérdida de divergencia de KL. Una variante de esto (la β -VAE) incluye un factor que pondera la divergencia de KL para garantizar que esté bien equilibrada con la pérdida de reconstrucción. Si ponderamos demasiado la pérdida de reconstrucción, la pérdida de KL no tendrá el efecto regulador deseado y veremos los mismos problemas que experimentamos con el auto codificador simple.

Si el término de divergencia de KL tiene demasiada ponderación, la pérdida de divergencia de KL dominará y las imágenes reconstruidas serán deficientes. Este término de ponderación es uno de los parámetros que debes ajustar cuando entrenas tu VAE.

Entrenando el auto codificador variacional

El ejemplo 3-13 muestra cómo construimos el modelo VAE general como una subclase de la clase abstracta del modelo Keras. Esto nos permite incluir el cálculo del término de divergencia KL de la función de pérdida en un método train_step personalizado.

Ejemplo 3-13. Entrenamiento de VAE

```
class VAE(models.Model):
    def __init__(self, encoder, decoder, **kwargs):
        super(VAE, self).__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder
        self.total_loss_tracker = metrics.Mean(name="total_loss")
        self.reconstruction_loss_tracker = metrics.Mean(
            name="reconstruction_loss"
        )
        self.kl_loss_tracker = metrics.Mean(name="kl_loss")

    @property
    def metrics(self):
        return [
            self.total_loss_tracker,
            self.reconstruction_loss_tracker,
            self.kl_loss_tracker,
        ]

    def call(self, inputs):
        z_mean, z_log_var, z = encoder(inputs)
        reconstruction = decoder(z)
        return z_mean, z_log_var, reconstruction

    def train_step(self, data):
        with tf.GradientTape() as tape:
            z_mean, z_log_var, reconstruction = self(data)
            reconstruction_loss = tf.reduce_mean(
                500
                * losses.binary_crossentropy(
                    data, reconstruction, axis=(1, 2, 3)
                )
            )
            kl_loss = tf.reduce_mean(
                tf.reduce_sum(
                    -0.5
                    * (1 + z_log_var - tf.square(z_mean) -
tf.exp(z_log_var)),
                    axis = 1,
                )
            )
            total_loss = reconstruction_loss + kl_loss

        grads = tape.gradient(total_loss, self.trainable_weights)
        self.optimizer.apply_gradients(zip(grads,
self.trainable_weights))
```

```

    self.total_loss_tracker.update_state(total_loss)

self.reconstruction_loss_tracker.update_state(reconstruction_los
s)
    self.kl_loss_tracker.update_state(kl_loss)

return {m.name: m.result() for m in self.metrics}

vae = VAE(encoder, decoder)
vae.compile(optimizer="adam")
vae.fit(
    train,
    epochs=5,
    batch_size=100
)

```

Esta función describe lo que nos gustaría que devolviera lo que llamamos VAE en una imagen de entrada particular.

Esta función describe un paso de entrenamiento del VAE, incluido el cálculo de la función de pérdida.

Se utiliza un valor beta de 500 en la pérdida de reconstrucción.

La pérdida total es la suma de la pérdida de reconstrucción y la pérdida de divergencia de KL.

CINTA DE GRADIENTE

La cinta de gradiente de TensorFlow es un mecanismo que permite calcular los gradientes de las operaciones ejecutadas durante un paso hacia adelante de un modelo. Para usarlo, debe empaquetar el código que realiza las operaciones que desea diferenciar en un contexto `tf.GradientTape()`. Una vez que haya registrado las operaciones, puede calcular el gradiente de la función de pérdida con respecto a algunas variables llamando a `tape.gradient()`. Luego, los gradientes se pueden usar para actualizar las variables con el optimizador.

Este mecanismo es útil para calcular el gradiente de funciones de pérdida personalizadas (como lo hemos hecho aquí) y también para crear bucles de entrenamiento personalizados, como veremos en el Capítulo 4.

Análisis del auto codificador variacional

Ahora que hemos entrenado nuestro VAE, podemos usar el codificador para codificar las imágenes en el conjunto de prueba y trazar los valores z_{mean} en el espacio latente. También podemos tomar muestras de una distribución normal estándar para generar puntos en el espacio latente y usar el decodificador para decodificar estos puntos nuevamente en el espacio de píxeles para ver cómo funciona el VAE.

La Figura 3-13 muestra la estructura del nuevo espacio latente, junto con algunos puntos muestreados y sus imágenes decodificadas.

Inmediatamente podemos ver varios cambios en cómo se organiza el espacio latente.

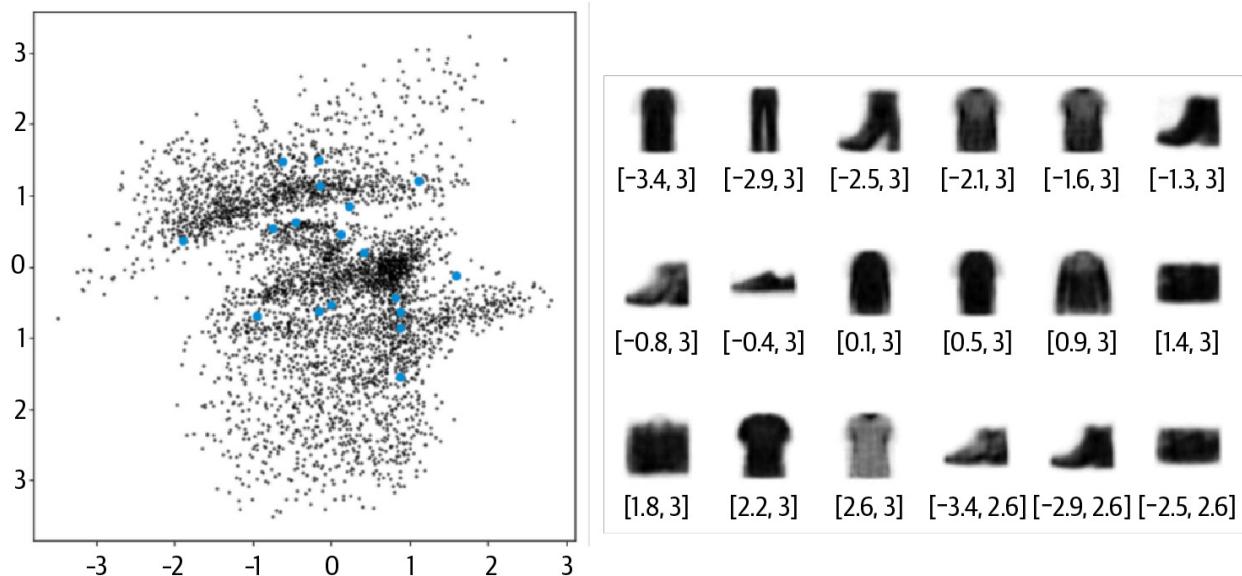


Figura 3-13. El nuevo espacio latente: los puntos negros muestran el valor z_{mean} de cada imagen codificada, mientras que los puntos azules muestran algunos puntos muestreados en el espacio latente (con sus imágenes decodificadas a la derecha)

En primer lugar, el término de pérdida de divergencia KL garantiza que los valores `z_mean` y `z_log_var` de las imágenes codificadas nunca se alejen demasiado de una distribución normal estándar. En segundo lugar, no hay tantas imágenes mal formadas ya que el espacio latente ahora es mucho más continuo, debido a que el codificador ahora es estocástico, en lugar de determinista.

Finalmente, al colorear los puntos en el espacio latente por tipo de ropa (Figura 3-14), podemos ver que no hay un trato preferencial para ningún tipo en particular. El gráfico de la derecha muestra el espacio transformado en valores p ; podemos ver que cada color está representado aproximadamente por igual. Nuevamente, es importante recordar que las etiquetas no se utilizaron en absoluto durante el entrenamiento; El VAE ha aprendido por sí mismo las distintas formas de vestimenta para ayudar a minimizar las pérdidas por reconstrucción.

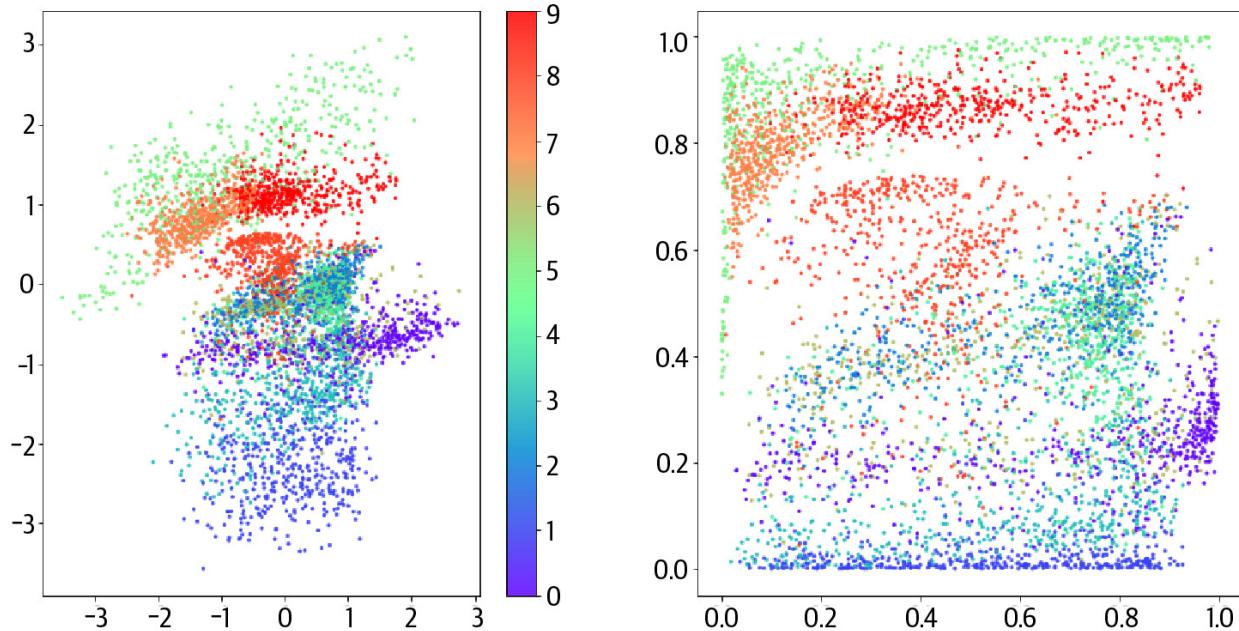


Figura 3-14. El espacio latente del VAE coloreado por tipo de ropa.

Explorando el espacio latente

Hasta ahora, todo nuestro trabajo sobre codificadores automáticos y codificadores automáticos variacionales se ha limitado a un espacio latente con dos dimensiones. Esto nos ha ayudado a visualizar el funcionamiento

interno de un VAE en la página y comprender por qué los pequeños ajustes que hicimos en la arquitectura del auto codificador ayudaron a transformarlo en una clase de red más potente que se puede utilizar para el modelado generativo.

Dirijamos ahora nuestra atención a un conjunto de datos más complejo y veamos las cosas sorprendentes que los codificadores automáticos variacionales pueden lograr cuando aumentamos la dimensionalidad del espacio latente.

EJECUTANDO EL CÓDIGO PARA ESTE EJEMPLO

El código para este ejemplo se puede encontrar en el cuaderno de Jupyter ubicado en notebooks/03_vae/03_faces/vae_faces.ipynb en el repositorio de libros.

El conjunto de datos de CelebA

Usaremos el conjunto de datos CelebFaces Attributes (CelebA) para entrenar nuestro próximo auto codificador variacional. Se trata de una colección de más de 200.000 imágenes en color de rostros de celebridades, cada una con varias etiquetas (por ejemplo, con sombrero, sonriendo, etc.). En la Figura 3-15 se muestran algunos ejemplos.



Figura 3-15. Algunos ejemplos del conjunto de datos de CelebA (fuente: Liu et al., 2015)3

Por supuesto, no necesitamos las etiquetas para entrenar el VAE, pero serán útiles más adelante, cuando comencemos a explorar cómo se capturan estas características en el espacio latente multidimensional. Una vez que nuestro VAE esté entrenado, podemos tomar muestras del espacio latente para generar nuevos ejemplos de rostros de celebridades.

El conjunto de datos de CelebA también está disponible a través de Kaggle, por lo que puede descargar el conjunto de datos ejecutando el script de descarga del conjunto de datos de Kaggle en el repositorio de libros, como se muestra en el ejemplo 3-14. Esto guardará las imágenes y los metadatos que las acompañan localmente en la carpeta /data.

Ejemplo 3-14. Descargando el conjunto de datos de CelebA
`scripts/download_kaggle_data.sh jessicali9530 celeba-dataset`

Usamos la función Keras `image_dataset_from_directory` para crear un conjunto de datos TensorFlow apuntando al directorio donde se almacenan las imágenes, como se muestra en el Ejemplo 3-15. Esto nos permite leer lotes de imágenes en la memoria solo cuando sea necesario (por ejemplo, durante el entrenamiento), de modo que podamos trabajar con conjuntos de datos grandes y no preocuparnos por tener que colocar todo el conjunto de datos en la memoria. También cambia el tamaño de las imágenes a 64×64 , interpolando entre valores de píxeles.

Ejemplo 3-15. Preprocesamiento del conjunto de datos de CelebA

```
train_data = utils.image_dataset_from_directory(  
    "/app/data/celeba-dataset/img_align_celeba/img_align_celeba",  
    labels=None,  
    color_mode="rgb",  
    image_size=(64, 64),  
    batch_size=128,  
    shuffle=True,  
    seed=42,  
    interpolation="bilinear",  
)
```

Los datos originales se escalan en el rango [0, 255] para indicar la intensidad de los píxeles, que reescalamos al rango [0, 1] como se muestra en el Ejemplo 3-16.

Ejemplo 3-16. Preprocesamiento del conjunto de datos de CelebA

```
def preprocess(img):  
    img = tf.cast(img, "float32") / 255.0  
    return img  
  
train = train_data.map(lambda x: preprocess(x))
```

Entrenando al auto codificador variacional

La arquitectura de red para el modelo de caras es similar al ejemplo de Fashion-MNIST, con algunas pequeñas diferencias:

- Nuestros datos ahora tienen tres canales de entrada (RGB) en lugar de uno (escala de grises). Esto significa que debemos cambiar el número

de canales en la capa de transposición convolucional final del decodificador a 3.

- Usaremos un espacio latente con 200 dimensiones en lugar de 2. Dado que las caras son mucho más complejas que las imágenes Fashion-MNIST, aumentamos la dimensionalidad del espacio latente para que la red pueda codificar una cantidad satisfactoria de detalles de las imágenes.
- Hay capas de normalización por lotes después de cada capa convolucional para estabilizar el entrenamiento. Aunque cada lote tarda más en ejecutarse, la cantidad de lotes necesarios para alcanzar la misma pérdida se reduce considerablemente.
- Aumentamos el factor β para la divergencia KL a 2.000. Este es un parámetro que requiere ajuste; Para este conjunto de datos y arquitectura, se encontró que este valor genera buenos resultados.

Las arquitecturas completas del codificador y decodificador se muestran en las Tablas 3-5 y 3-6, respectivamente.

Capa (tipo)	Forma de salida	# de parámetros	Conectado a
Capa de entrada (entrada)	(Ninguno, 32, 32, 3)	0	[]
Conv2D (conv2d_1)	(Ninguno, 16, 16, 128)	3,584	[entrada]
Normalización por lotes (bn_1)	(Ninguno, 16, 16, 128)	512	[conv2d_1]
LeakyReLU (lr_1)	(Ninguno, 16, 16, 128)	0	[bn_1]
Conv2D (conv2d_2)	(Ninguno, 8, 8, 128)	147,584	[lr_1]
Normalización por lotes (bn_2)	(Ninguno, 8, 8, 128)	512	[conv2d_2]
LeakyReLU (lr_2)	(Ninguno, 8, 8, 128)	0	[bn_2]
Conv2D (conv2d_3)	(Ninguno, 4, 4,	147,584	[lr_2]

	128)		
Normalización por lotes (bn_3)	(Ninguno, 4, 4, 128)	512	[conv2d_3]
LeakyReLU (lr_3)	(Ninguno, 4, 4, 128)	0	[bn_3]
Conv2D (conv2d_4)	(Ninguno, 2, 2, 128)	147,584	[lr_3]
Normalización por lotes (bn_4)	(Ninguno, 2, 2, 128)	512	[conv2d_4]
LeakyReLU (lr_4)	(Ninguno, 2, 2, 128)	0	[bn_4]
Aplanar (flatten)	(Ninguno, 512)	0	[lr_4]
Denso (z_mean)	(Ninguno, 200)	102,600	[aplanar]
Denso (z_log_var)	(Ninguno, 200)	102,600	[aplanar]
Muestreo (z)	(Ninguno, 200)	[z_mean, z_log_var]	
Parámetros totales	653,584		
Parámetros entrenables	652,560		
Parámetros no entrenables	1,024		

Capa (tipo)	Forma de salida	# de parámetros
Capa de entrada	(Ninguno, 200)	0
Denso	(Ninguno, 512)	102,912
Normalización por lotes	(Ninguno, 512)	2,048
LeakyReLU	(Ninguno, 512)	0
remodelar	(Ninguno, 2, 2, 128)	0
Conv2DTransponer	(Ninguno, 4, 4, 128)	147,584
Normalización por lotes	(Ninguno, 4, 4, 128)	512
LeakyReLU	(Ninguno, 4, 4, 128)	0
Conv2DTransponer	(Ninguno, 8, 8, 128)	147,584
Normalización por lotes	(Ninguno, 8, 8, 128)	512

LeakyReLU	(Ninguno, 8, 8, 128)	0
Conv2DTransponer	(Ninguno, 16, 16, 128)	147,584
Normalización por lotes	(Ninguno, 16, 16, 128)	512
LeakyReLU	(Ninguno, 16, 16, 128)	0
Conv2DTransponer	(Ninguno, 32, 32, 128)	147,584
Normalización por lotes	(Ninguno, 32, 32, 128)	512
LeakyReLU	(Ninguno, 32, 32, 128)	0
Conv2DTransponer	(Ninguno, 32, 32, 3)	3,459
Parámetros totales	700,803	
Parámetros entrenables	698,755	
Parámetros no entrenables	2,048	

¡Después de aproximadamente cinco épocas de entrenamiento, nuestro VAE debería poder producir imágenes novedosas de rostros de celebridades!

Análisis del auto codificador variacional

Primero, echemos un vistazo a una muestra de rostros reconstruidos.

La fila superior de la Figura 3-16 muestra las imágenes originales y la fila inferior muestra las reconstrucciones una vez que han pasado por el codificador y decodificador.

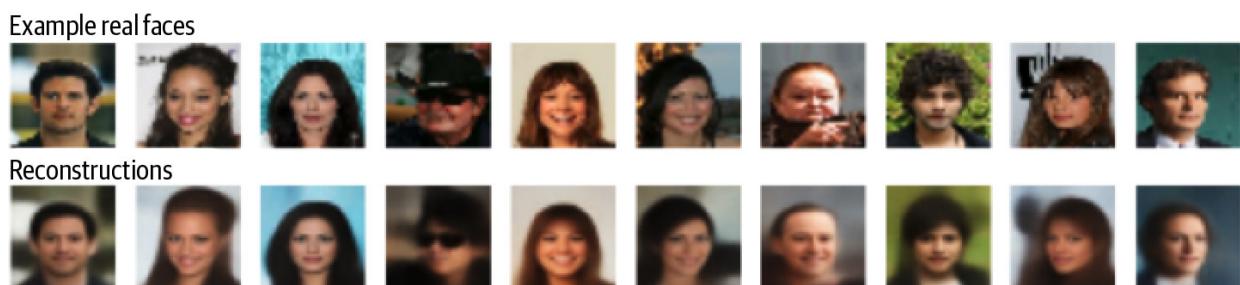


Figura 3-16. Rostros reconstruidos, tras pasar por el codificador y decodificador

Podemos ver que el VAE ha capturado con éxito las características clave de cada rostro: el ángulo de la cabeza, el peinado, la expresión, etc. Faltan

algunos detalles finos, pero es importante recordar que el objetivo de construir variaciones Los codificadores automáticos no deben lograr una pérdida de reconstrucción perfecta. Nuestro objetivo final es tomar muestras del espacio latente para generar nuevas caras.

Para que esto sea posible debemos comprobar que la distribución de puntos en el espacio latente se asemeja aproximadamente a una distribución normal estándar multivariada. Si vemos dimensiones que son significativamente diferentes de una distribución normal estándar, probablemente deberíamos reducir el factor de pérdida de reconstrucción, ya que el término de divergencia de KL no está teniendo suficiente efecto.

Las primeras 50 dimensiones en nuestro espacio latente se muestran en la figura 3-17. No hay ninguna distribución que se destaque por ser significativamente diferente de la normal, ¡así que podemos pasar a generar algunas caras!

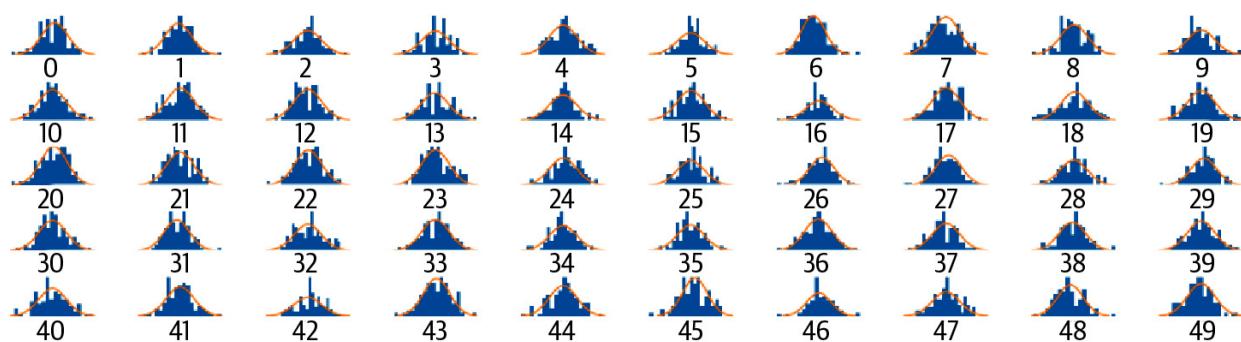


Figura 3-17. Distribuciones de puntos para las primeras 50 dimensiones en el espacio latente.

Generando caras nuevas

Para generar caras nuevas, podemos usar el código del Ejemplo 317.

Ejemplo 3-17. Generando nuevas caras a partir del espacio latente

```
grid_width, grid_height = (10, 3)
z_sample = np.random.normal(size=(grid_width * grid_height,
200))

reconstructions = decoder.predict(z_sample)
```

```

fig = plt.figure(figsize=(18, 5))
fig.subplots_adjust(hspace=0.4, wspace=0.4)
for i in range(grid_height * grid_width):
    ax = fig.add_subplot(grid_height, grid_width, i + 1)
    ax.axis("off")
    ax.imshow(reconstructions[i, :, :])

```

1. Muestreo de 30 puntos de una distribución normal multivariada estándar con 200 dimensiones.
2. Decodifica los puntos muestreados.
3. ¡Traza las imágenes!

El resultado se muestra en la Figura 3-18.

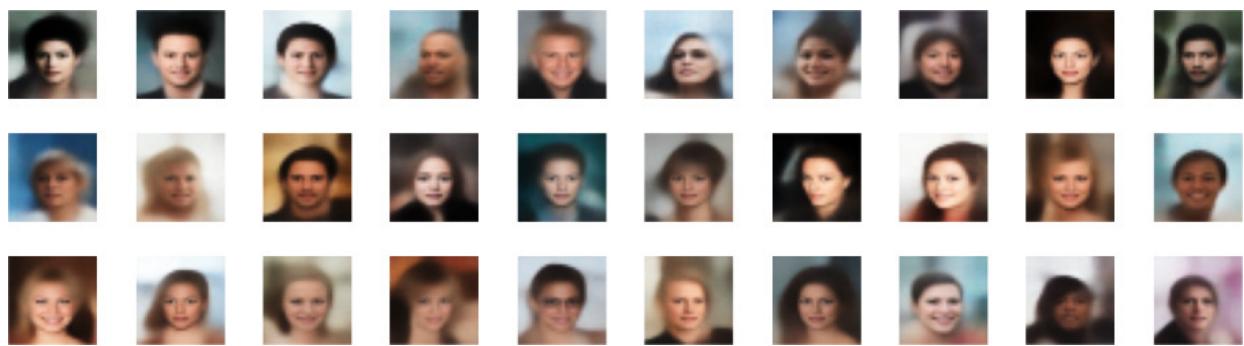


Figura 3-18. Nuevas caras generadas

Sorprendentemente, el VAE es capaz de tomar el conjunto de puntos que tomamos como muestra de una distribución normal estándar y convertir cada uno en una imagen convincente del rostro de una persona. ¡Este es nuestro primer vistazo del verdadero poder de los modelos generativos!

A continuación, veamos si podemos empezar a utilizar el espacio latente para realizar algunas operaciones interesantes en las imágenes generadas.

Aritmética del espacio latente

Un beneficio de mapear imágenes en un espacio latente de dimensiones inferiores es que podemos realizar aritmética en vectores en este espacio latente que tiene un análogo visual cuando se decodifica nuevamente en el dominio de la imagen original.

Por ejemplo, supongamos que queremos tomar una imagen de alguien que parece triste y sonreírle. Para hacer esto primero necesitamos encontrar un vector en el espacio latente que apunte en la dirección de la sonrisa creciente. Agregar este vector a la codificación de la imagen original en el espacio latente nos dará un nuevo punto que, cuando se decodifique, debería darnos una versión más sonriente de la imagen original.

Entonces, ¿cómo podemos encontrar el vector de la sonrisa? Cada imagen en el conjunto de datos de CelebA está etiquetado con atributos, uno de los cuales es Sonriendo. Si tomamos la posición promedio de las imágenes codificadas en el espacio latente con el atributo Sonriendo y restamos la posición promedio de las imágenes codificadas que no tienen el atributo Sonriendo, obtendremos el vector que apunta en la dirección de Sonriendo, que es exactamente lo que nosotros necesitamos.

Conceptualmente, estamos realizando la siguiente aritmética vectorial en el espacio latente, donde alfa es un factor que determina qué parte del vector de características se suma o resta:

$$z_{\text{new}} = z + \alpha * \text{feature_vector}$$

Veamos esto en acción. La Figura 3-19 muestra varias imágenes que han sido codificadas en el espacio latente. Luego sumamos o restamos múltiplos de un determinado vector (por ejemplo, Sonriendo, Cabello_Negro, Anteojos, Joven, Hombre, Cabello_Rubio) para obtener diferentes versiones de la imagen, cambiando solo la característica relevante.

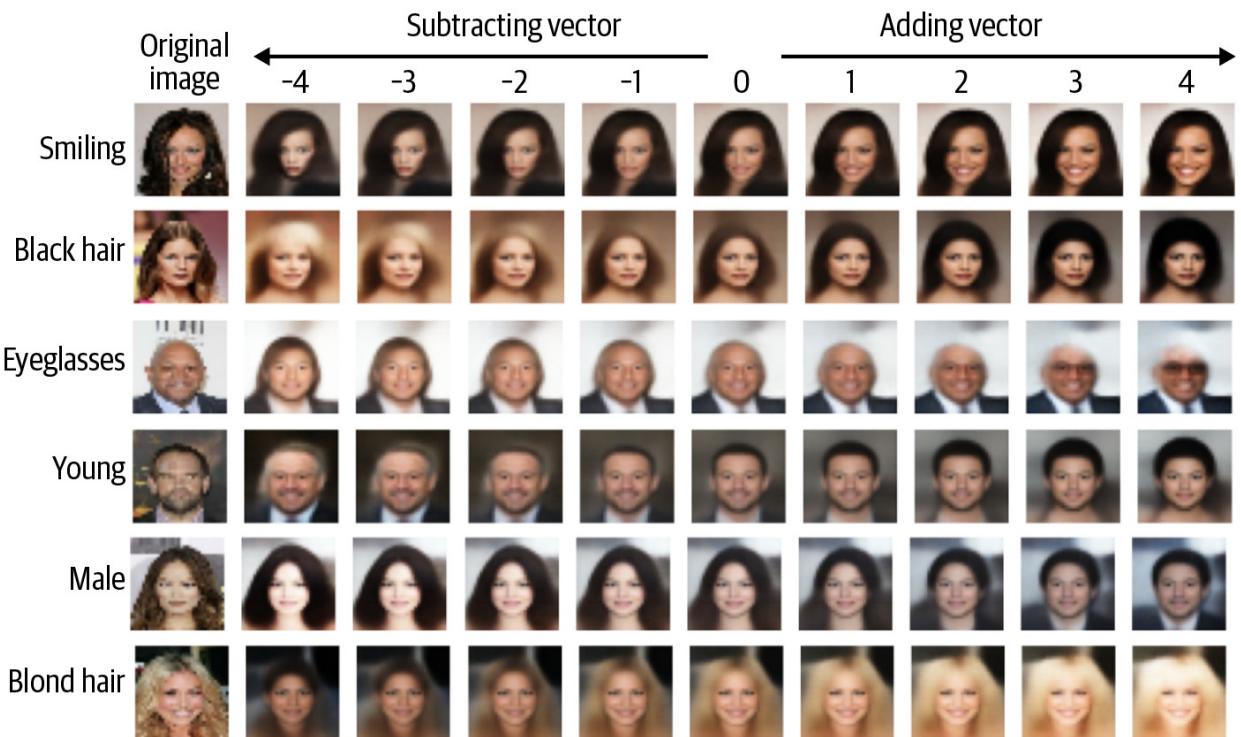


Figura 3-19. Agregar y restar características a y desde caras

Es notable que aunque estemos moviendo el punto una distancia significativamente grande en el espacio latente, la imagen central permanece aproximadamente igual, excepto por la característica que queremos manipular. Esto demuestra el poder de los codificadores automáticos variacionales para capturar y ajustar características de alto nivel en imágenes.

Transformación entre caras

Podemos usar una idea similar para transformar dos caras.

Imagine dos puntos en el espacio latente, A y B, que representan dos imágenes. Si comenzaste en el punto A y caminaste hacia el punto B en línea recta, decodificando cada punto de la línea a medida que avanzas, verías una transición gradual desde la cara inicial hasta la cara final.

Matemáticamente, estamos atravesando una línea recta, que se puede describir mediante la siguiente ecuación:

$$z_{new} = z_A * (1 - \alpha) + z_B * \alpha$$

Aquí, α es un número entre 0 y 1 que determina qué tan lejos estamos de la línea, lejos del punto A.

La Figura 3-20 muestra este proceso en acción. Tomamos dos imágenes, las codificamos en el espacio latente y luego decodificamos puntos a lo largo de la línea recta entre ellas a intervalos regulares.

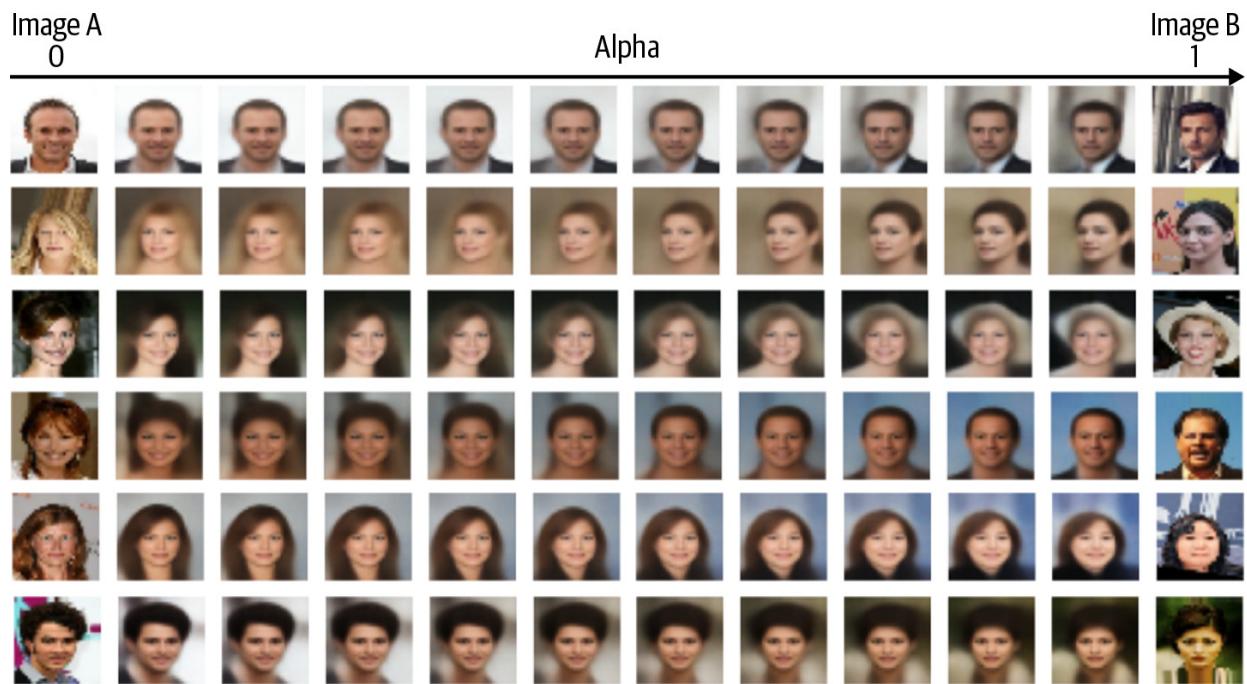


Figura 3-20. Morphing entre dos caras

Vale la pena señalar la suavidad de la transición: incluso cuando hay múltiples características que cambiar simultáneamente (por ejemplo, quitarse las gafas, color de cabello, género), el VAE logra lograrlo de manera fluida, lo que demuestra que el espacio latente del VAE es verdaderamente un espacio continuo que puede ser recorrido y explorado para generar multitud de rostros humanos diferentes.

Resumen

En este capítulo hemos visto cómo los codificadores automáticos variacionales son una herramienta poderosa en la caja de herramientas del

modelado generativo. Comenzamos explorando cómo se pueden usar codificadores automáticos simples para mapear imágenes de alta dimensión en un espacio latente de baja dimensión, de modo que se puedan extraer características de alto nivel de los píxeles individualmente no informativos. Sin embargo, rápidamente descubrimos que había algunos inconvenientes al utilizar codificadores automáticos simples como modelo generativo; por ejemplo, el muestreo del espacio latente aprendido era problemático.

Los codificadores automáticos variacionales resuelven estos problemas introduciendo aleatoriedad en el modelo y restringiendo cómo se distribuyen los puntos en el espacio latente. Vimos que con unos pocos ajustes menores, podemos transformar nuestro auto codificador en un auto codificador variacional, dándole así el poder de ser un verdadero modelo generativo.

Finalmente, aplicamos nuestra nueva técnica al problema de la generación de rostros y vimos cómo podemos simplemente decodificar puntos de una distribución normal estándar para generar nuevos rostros.

Además, al realizar aritmética vectorial dentro del espacio latente, podemos lograr algunos efectos sorprendentes, como la transformación de rostros y la manipulación de características.

En el próximo capítulo, exploraremos un tipo diferente de modelo que sigue siendo una opción popular para el modelado de la generación de imágenes: la red generativa adversarial.

-
1. Diederik P. Kingma y Max Welling, "Bayes variacionales de codificación automática", 20 de diciembre de 2013, <https://arxiv.org/abs/1312.6114>.
 2. Vincent Dumoulin y Francesco Visin, "Una guía para la convolución aritmética para el aprendizaje profundo", 12 de enero de 2018, <https://arxiv.org/abs/1603.07285>.
 3. Ziwei Liu et al., "Conjunto de datos de atributos de celebridades a gran escala (CelebA)", 2015, <http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html>.

Capítulo 4. Redes adversarias generativas

METAS DEL CAPÍTULO

En este capítulo podrás:

- Conocer el diseño arquitectónico de una red generativa adversarial (GAN).
- Crear y entrenar una GAN convolucional profunda (DCGAN) desde cero utilizando Keras.
- Utilizar DCGAN para generar nuevas imágenes.
- Comprender algunos de los problemas comunes que se enfrentan al entrenar un DCGAN.
- Descubrir cómo la arquitectura Wasserstein GAN (WGAN) aborda estos problemas.
- Comprender las mejoras adicionales que se pueden realizar en WGAN, como incorporar un término de penalización de gradiente (GP) en la función de pérdida.
- Crear una WGAN-GP desde cero usando Keras.
- Utilizar una WGAN-GP para generar caras.
- Descubrir cómo una GAN condicional (CGAN) le brinda la capacidad de condicionar la salida generada en una etiqueta determinada.
- Crear y entrenar una CGAN en Keras y utilizarla para manipular una imagen generada.

En 2014, Ian Goodfellow et al. presentó un artículo titulado “Generative Adversarial Nets”¹ en la Conferencia sobre sistemas de procesamiento neuronal de información (NeurIPS) en Montreal. La introducción de redes generativas adversarias (o GAN, como se las conoce más comúnmente) se considera ahora como un punto de inflexión clave en la historia del modelado generativo, ya que las ideas centrales presentadas en este artículo han generado algunos de los modelos generativos más exitosos e impresionantes jamás creados.

Este capítulo primero expondrá los fundamentos teóricos de las GAN, luego veremos cómo construir nuestra propia GAN usando Keras.

Introducción

Comencemos con una breve historia para ilustrar algunos de los conceptos fundamentales utilizados en el proceso de capacitación de GAN.

BRICKKI BRICKS Y LOS FALSADORES

Es su primer día en su nuevo trabajo como jefe de control de calidad de Brickki, una empresa que se especializa en producir bloques de construcción de alta calidad de todas las formas y tamaños (Figura 4-1).



Figura 4-1. La línea de producción de una empresa que fabrica ladrillos de diferentes formas y tamaños (creada con Midjourney)

Inmediatamente se le alerta sobre un problema con algunos de los artículos que salen de la línea de producción. Un competidor ha comenzado a hacer copias falsificadas de los ladrillos Brickki y ha encontrado una manera de mezclarlos en las bolsas que reciben sus clientes. Decide convertirse en un experto en distinguir entre los ladrillos falsificados y los reales, para poder interceptar los ladrillos forjados en la línea de producción antes de

entregárselos a los clientes. Con el tiempo, al escuchar los comentarios de los clientes, gradualmente te volverás más hábil para detectar falsificaciones.

Los falsificadores no están contentos con esto: reaccionan a sus capacidades de detección mejoradas haciendo algunos cambios en su proceso de falsificación, por lo que ahora la diferencia entre los ladrillos reales y los falsos es aún más difícil de detectar.

No eres alguien que se dé por vencido, te capacitas para identificar las falsificaciones más sofisticadas y tratas de ir un paso por delante de los falsificadores. Este proceso continúa, con los falsificadores actualizando iterativamente sus tecnologías de creación de ladrillos mientras usted intenta ser cada vez más experto en interceptar sus falsificaciones.

Cada semana que pasa, se vuelve cada vez más difícil distinguir entre los ladrillos Brickki reales y los creados por los falsificadores. Parece que este simple juego del gato y el ratón es suficiente para impulsar una mejora significativa tanto en la calidad de la falsificación como en la calidad de la detección.

La historia de Brickki Bricks y los falsificadores describe el proceso de formación de una red generativa de confrontación.

Una GAN es una batalla entre dos adversarios, el generador y el discriminador. El generador intenta convertir el ruido aleatorio en observaciones que parecen haber sido muestreadas del conjunto de datos original, y el discriminador intenta predecir si una observación proviene del conjunto de datos original o es una de las falsificaciones del generador. En la Figura 4-2 se muestran ejemplos de las entradas y salidas de las dos redes.

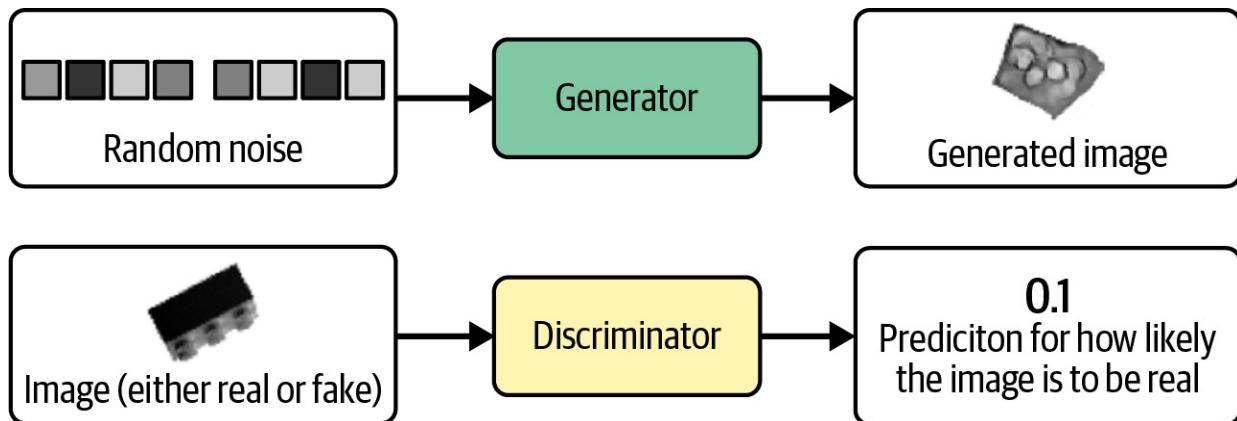


Figura 4-2. Entradas y salidas de las dos redes en una GAN

Al comienzo del proceso, el generador genera imágenes ruidosas y el discriminador predice aleatoriamente. La clave para las GAN radican en cómo alternamos el entrenamiento de las dos redes, de modo que a medida que el generador se vuelve más hábil para engañar al discriminador, el discriminador debe adaptarse para mantener su capacidad de identificar correctamente qué observaciones son falsas. Esto impulsa al generador a encontrar nuevas formas de engañar al discriminador, y así el ciclo continúa.

GAN convolucional profunda (DCGAN)

Para ver esto en acción, comenzemos a construir nuestra primera GAN en Keras, para generar imágenes de ladrillos.

Seguiremos de cerca uno de los primeros artículos importantes sobre GAN, “Aprendizaje de representación no supervisado con redes adversarias generativas convolucionales profundas”. [2] En este artículo de 2015, los autores muestran cómo construir una GAN convolucional profunda para generar imágenes realistas a partir de una variedad de conjuntos de datos. También introducen varios cambios que mejoran significativamente la calidad de las imágenes generadas.

EJECUTANDO EL CÓDIGO PARA ESTE EJEMPLO

El código de este ejemplo se puede encontrar en el cuaderno de Jupyter ubicado en notebooks/04_gan/01_dcgan/dcgan.ipynb en el repositorio de

libros.

El conjunto de datos de ladrillos

Primero, necesitarás descargar los datos de entrenamiento. Usaremos el conjunto de datos Imágenes de ladrillos LEGO que está disponible a través de Kaggle. Esta es una colección renderizada por computadora de 40.000 imágenes fotográficas de 50 ladrillos de juguete diferentes, tomadas desde múltiples ángulos. Algunas imágenes de ejemplo de los productos Brickki se muestran en la Figura 4-3.



Figura 4-3. Ejemplos de imágenes del conjunto de datos Bricks

Puede descargar el conjunto de datos ejecutando el script de descarga del conjunto de datos de Kaggle en el repositorio de libros, como se muestra en el Ejemplo 4-1. Esto guardará las imágenes y los metadatos que las acompañan localmente en la carpeta /data.

Ejemplo 4-1. Descargando el conjunto de datos de Ladrillos

```
bash scripts/download_kaggle_data.sh joosthazelzet lego-brickimages
```

Usamos la función Keras `image_dataset_from_directory` para crear un conjunto de datos Tensorflow apuntando al directorio donde se almacenan las imágenes, como se muestra en el Ejemplo 4-2. Esto nos permite leer lotes de imágenes en la memoria solo cuando sea necesario (por ejemplo, durante el entrenamiento), de modo que podamos trabajar con conjuntos de datos grandes y no preocuparnos por tener que colocar todo el conjunto de datos en la memoria. También cambia el tamaño de las imágenes a 64×64 , interpolando entre valores de píxeles.

Ejemplo 4-2. Crear un conjunto de datos de TensorFlow a partir de archivos de imagen en un directorio

```

train_data = utils.image_dataset_from_directory(
    "/app/data/lego-brick-images/dataset/",
    labels=None,
    color_mode="grayscale",
    image_size=(64, 64),
    batch_size=128,
    shuffle=True,
    seed=42,
    interpolation="bilinear",
)

```

Los datos originales se escalan en el rango [0, 255] para indicar la intensidad de los píxeles. Cuando entrenamos GAN, reescalamos los datos al rango [-1, 1] para que podamos usar la función de activación tanh en la capa final del generador, que tiende a proporcionar gradientes más fuertes que la función sigmoidea (Ejemplo 4-3).

Ejemplo 4-3. Preprocesamiento del conjunto de datos de Bricks

```

def preprocess(img):
    img = (tf.cast(img, "float32") - 127.5) / 127.5
    return img

train = train_data.map(lambda x: preprocess(x))

```

Veamos ahora cómo construimos el discriminador.

El discriminador

El objetivo del discriminador es predecir si una imagen es real o falsa. Este es un problema de clasificación de imágenes supervisado, por lo que podemos usar una arquitectura similar a las que trabajamos en el Capítulo 2: capas convolucionales apiladas, con un único nodo de salida.

La arquitectura completa del discriminador que construiremos se muestra en la Tabla 4-1.

Capa (tipo)	Forma de salida	# de parámetros
Capa de entrada	(Ninguno, 64, 64, 1)	0
Conv2D	(Ninguno, 32, 32, 64)	1,024

LeakyReLU	(Ninguno, 32, 32, 64)	0
Dropout	(Ninguno, 32, 32, 64)	0
Conv2D	(Ninguno, 16, 16, 128)	131,072
Normalización por lotes	(Ninguno, 16, 16, 128)	512
LeakyReLU	(Ninguno, 16, 16, 128)	0
Dropout	(Ninguno, 16, 16, 128)	0
Conv2D	(Ninguno, 8, 8, 256)	524,288
Normalización por lotes	(Ninguno, 8, 8, 256)	1,024
LeakyReLU	(Ninguno, 8, 8, 256)	0
Abandonar	(Ninguno, 8, 8, 256)	0
Conv2D	(Ninguno, 4, 4, 512)	2,097,152
Normalización por lotes	(Ninguno, 4, 4, 512)	2,048
LeakyReLU	(Ninguno, 4, 4, 512)	0
Abandonar	(Ninguno, 4, 4, 512)	0
Conv2D	(Ninguno, 1, 1, 1)	8,192
Aplanar	(Ninguno, 1)	0
Parámetros totales	2,765,312	
Parámetros entrenables	2,763,520	
Parámetros no entrenables	1,792	

El código Keras para construir el discriminador se proporciona en el ejemplo 4-4.

Ejemplo 4-4. el discriminador

```
discriminator_input = layers.Input(shape=(64, 64, 1))
x = layers.Conv2D(64, kernel_size=4, strides=2, padding="same",
use_bias = False)(discriminator_input)
x = layers.LeakyReLU(0.2)(x)
x = layers.Dropout(0.3)(x)
x = layers.Conv2D(
    128, kernel_size=4, strides=2, padding="same", use_bias =
False
```

```

)(x)
x = layers.BatchNormalization(momentum = 0.9)(x)
x = layers.LeakyReLU(0.2)(x)
x = layers.Dropout(0.3)(x)
x = layers.Conv2D(
    256, kernel_size=4, strides=2, padding="same", use_bias =
False
)(x)
x = layers.BatchNormalization(momentum = 0.9)(x)
x = layers.LeakyReLU(0.2)(x)
x = layers.Dropout(0.3)(x)
x = layers.Conv2D(
    512, kernel_size=4, strides=2, padding="same", use_bias =
False
)(x)
x = layers.BatchNormalization(momentum = 0.9)(x)
x = layers.LeakyReLU(0.2)(x)
x = layers.Dropout(0.3)(x)
x = layers.Conv2D(
    1,
    kernel_size=4,
    strides=1,
    padding="valid",
    use_bias = False,
    activation = 'sigmoid'
)(x)
discriminator_output = layers.Flatten()(x)

discriminator = models.Model(discriminator_input,
discriminator_output)

```

1. Defina la capa de entrada del discriminador (la imagen).
2. Apile las capas Conv2D una encima de la otra, con capas BatchNormalization, activación LeakyReLU y Dropout intercaladas.
3. Aplane la última capa convolucional; en este punto, la forma del tensor es $1 \times 1 \times 1$, por lo que no hay necesidad de una capa Dense final.
4. El modelo de Keras que define el discriminador: un modelo que toma una imagen de entrada y genera un único número entre 0 y 1.

Observe cómo usamos un paso de 2 en algunas de las capas Conv2D para reducir la forma espacial del tensor a medida que pasa a través de la red (64 en la imagen original, luego 32, 16, 8, 4 y finalmente 1), mientras aumenta el número de canales (1 en la imagen de entrada en escala de grises, luego 64, 128, 256 y finalmente 512), antes de colapsar en una sola predicción.

Usamos una activación sigmoidea en la capa Conv2D final para generar un número entre 0 y 1.

El generador

Ahora construyamos el generador. La entrada al generador será un vector extraído de una distribución normal estándar multivariada. El resultado es una imagen del mismo tamaño que una imagen de los datos de entrenamiento originales.

Esta descripción puede recordarle el decodificador de un auto codificador variacional. De hecho, el generador de una GAN cumple exactamente el mismo propósito que el decodificador de un VAE: convertir un vector en el espacio latente en una imagen. El concepto de mapear desde un espacio latente al dominio original es muy común en el modelado generativo, ya que nos brinda la capacidad de manipular vectores en el espacio latente para cambiar características de alto nivel de las imágenes en el dominio original.

La arquitectura del generador que construiremos se muestra en la Tabla 4-2.

Capa (tipo)	Forma de salida	# de parámetros
Capa de entrada	(Ninguno, 100)	0
remodelar	(Ninguno, 1, 1, 100)	0
Conv2DTransponer	(Ninguno, 4, 4, 512)	819,200
Normalización por lotes	(Ninguno, 4, 4, 512)	2,048
ReLU	(Ninguno, 4, 4, 512)	0
Conv2DTransponer	(Ninguno, 8, 8, 256)	2,097,152
Normalización por lotes	(Ninguno, 8, 8, 256)	1,024
ReLU	(Ninguno, 8, 8, 256)	0
Conv2DTransponer	(Ninguno, 16, 16, 128)	524,288
Normalización por lotes	(Ninguno, 16, 16, 128)	512
ReLU	(Ninguno, 16, 16, 128)	0
Conv2DTransponer	(Ninguno, 32, 32, 64)	131,072
Normalización por lotes	(Ninguno, 32, 32, 64)	256

ReLU	(Ninguno, 32, 32, 64)	0
Conv2DTransponer	(Ninguno, 64, 64, 1)	1,024
Parámetros totales	3,576,576	
Parámetros entrenables	3,574,656	
Parámetros no entrenables	1,920	

El código para construir el generador se da en el Ejemplo 4-5.

Ejemplo 4-5. El generador

```
generator_input = layers.Input(shape=(100,))
x = layers.Reshape((1, 1, 100))(generator_input)
x = layers.Conv2DTranspose(
    512, kernel_size=4, strides=1, padding="valid", use_bias =
False
)(x)
x = layers.BatchNormalization(momentum=0.9)(x)
x = layers.LeakyReLU(0.2)(x)
x = layers.Conv2DTranspose(
    256, kernel_size=4, strides=2, padding="same", use_bias =
False
)(x)
x = layers.BatchNormalization(momentum=0.9)(x)
x = layers.LeakyReLU(0.2)(x)
x = layers.Conv2DTranspose(
    128, kernel_size=4, strides=2, padding="same", use_bias =
False
)(x)
x = layers.BatchNormalization(momentum=0.9)(x)
x = layers.LeakyReLU(0.2)(x)
x = layers.Conv2DTranspose(
    64, kernel_size=4, strides=2, padding="same", use_bias = False
)(x)
x = layers.BatchNormalization(momentum=0.9)(x)
x = layers.LeakyReLU(0.2)(x)
generator_output = layers.Conv2DTranspose(
    1,
    kernel_size=4,
    strides=2,
    padding="same",
    use_bias = False,
    activation = 'tanh'
)(x)
generator = models.Model(generator_input, generator_output)
```

1. Defina la capa de entrada del generador: un vector de longitud 100.
2. Usamos una capa Reshape para dar un tensor de $1 \times 1 \times 100$, de modo que podamos comenzar a aplicar operaciones de transposición convolucional.
3. Pasamos esto a través de cuatro capas Conv2DTranspose, con capas BatchNormalization y LeakyReLU intercaladas.
4. La capa final Conv2DTranspose utiliza una función de activación tanh para transformar la salida al rango $[-1, 1]$, para que coincida con el dominio de la imagen original.
5. El modelo de Keras que define el generador: un modelo que acepta un vector de longitud 100 y genera un tensor de forma $[64, 64, 1]$.

Observe cómo usamos una zancada de 2 en algunas de las capas Conv2DTranspose para aumentar la forma espacial del tensor a medida que pasa a través de la red (1 en el vector original, luego 4, 8, 16, 32 y finalmente 64), mientras disminuye el número de canales (512, luego 256, 128, 64, y finalmente 1 para que coincida con la salida en escala de grises).

SOBREMUESTRAR VERSUS TRANSPONER CON CONV2D

Una alternativa al uso de capas Conv2DTranspose es usar una capa UpSampling2D seguida de una capa normal Conv2D con paso 1, como se muestra en el Ejemplo 4-6.

Ejemplo 4-6. Ejemplo de muestreo ascendente

```
x = layers.UpSampling2D(size = 2)(x)
x = layers.Conv2D(256, kernel_size=4, strides=1, padding="same")
(x)
```

La capa UpSampling2D simplemente repite cada fila y columna de su entrada para duplicar el tamaño. La capa Conv2D con zancada 1 luego realiza la operación de convolución. Es una idea similar a la transposición convolucional, pero en lugar de llenar los espacios entre píxeles con ceros, el muestreo ascendente simplemente repite los valores de píxeles existentes.

Se ha demostrado que el método Conv2DTranspose puede generar artefactos o pequeños patrones de tablero de ajedrez en la imagen de salida (consulte la Figura 4-4) que estropean la calidad de la salida. Sin embargo, todavía se utilizan en muchas de las GAN más impresionantes de la literatura y han demostrado ser una herramienta poderosa en la caja de herramientas del profesional del aprendizaje profundo.

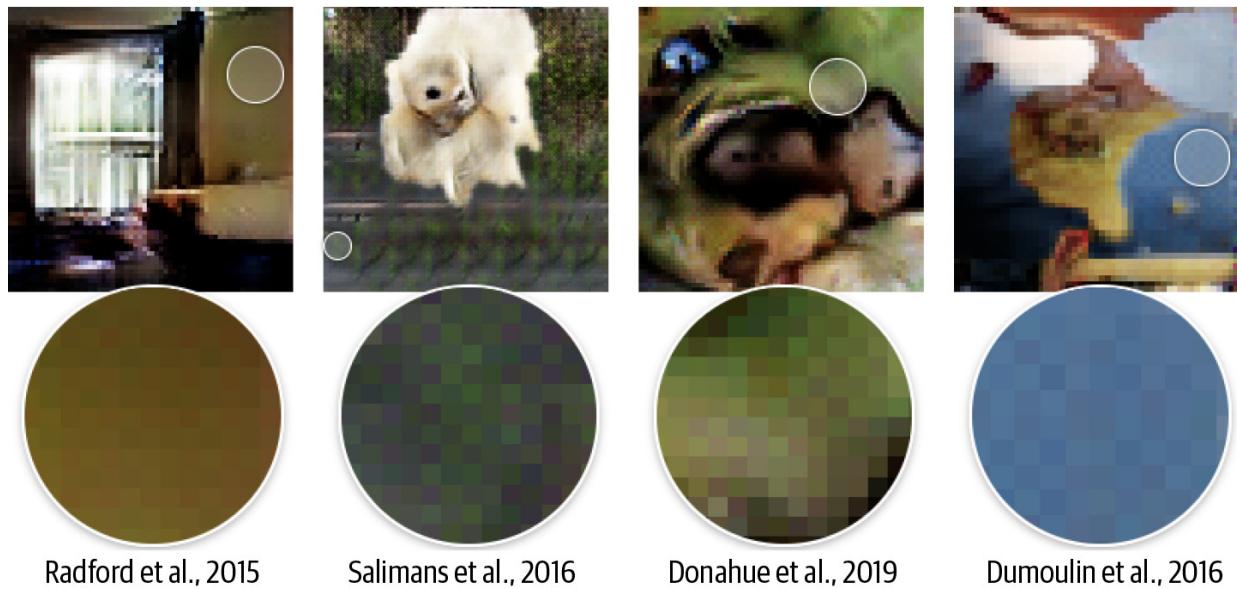


Figura 4-4. Artefactos al utilizar capas de transposición convolucionales (fuente: Odena et al., 2016) [3]

Ambos métodos: UpSampling2D + Conv2D y Conv2DTranspose: son formas aceptables de volver a transformar al dominio de imagen original. Realmente es un caso de probar ambos métodos en su propio entorno de problemas y ver cuál produce mejores resultados.

Entrenando la DCGAN

Como hemos visto, las arquitecturas del generador y discriminador en un DCGAN son muy simples y no tan diferentes de los modelos VAE que vimos en el capítulo 3. La clave para comprender las GAN radica en comprender el proceso de capacitación del generador y el discriminador.

Podemos entrenar al discriminador creando un conjunto de entrenamiento donde algunas de las imágenes sean observaciones reales del conjunto de

entrenamiento y otras sean resultados falsos del generador.

Luego tratamos esto como un problema de aprendizaje supervisado, donde las etiquetas son 1 para las imágenes reales y 0 para las imágenes falsas, con la entropía cruzada binaria como función de pérdida.

¿Cómo debemos entrenar el generador? Necesitamos encontrar una manera de calificar cada imagen generada para que pueda optimizarse hacia imágenes de alta puntuación. ¡Afortunadamente, tenemos un discriminador que hace exactamente eso! Podemos generar un lote de imágenes y pasárlas por el discriminador para obtener una puntuación para cada imagen. La función de pérdida para el generador es entonces simplemente la entropía cruzada binaria entre estas probabilidades y un vector de unos, porque queremos entrenar al generador para que produzca imágenes que el discriminador cree que son reales.

Fundamentalmente, debemos alternar el entrenamiento de estas dos redes, asegurándonos de actualizar solo los pesos de una red a la vez. Por ejemplo, durante el proceso de entrenamiento del generador, solo se actualizan los pesos del generador.

Si permitiéramos que los pesos del discriminador cambiaron también, el discriminador simplemente se ajustaría para que sea más probable predecir que las imágenes generadas sean reales, lo cual no es el resultado deseado. Queremos que las imágenes generadas se predigan cerca de 1 (real) porque el generador es fuerte, no porque el discriminador sea débil.

En la Figura 4-5 se muestra un diagrama del proceso de entrenamiento para el discriminador y el generador.

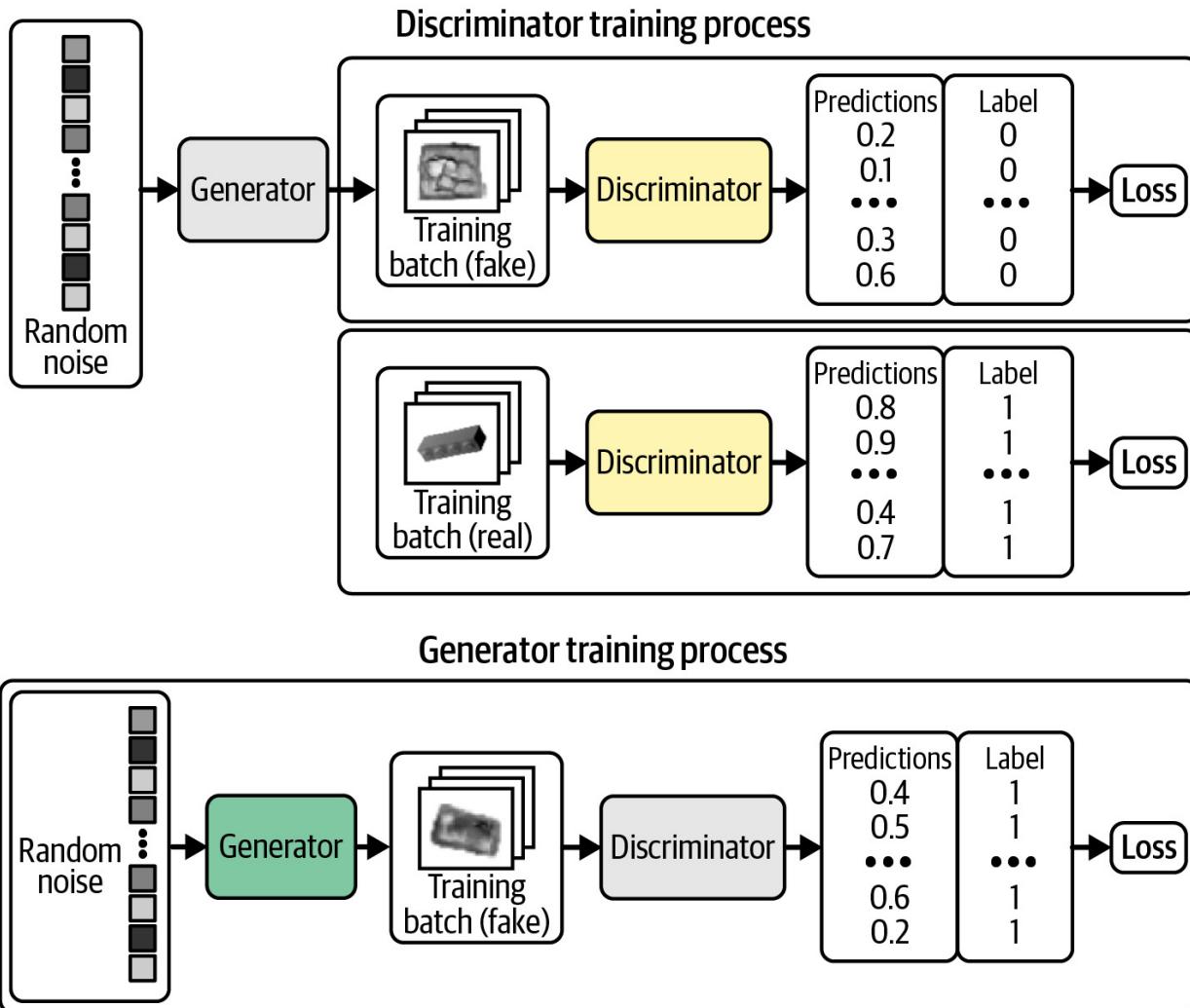


Figura 4-5. Entrenamiento del DCGAN: los cuadros grises indican que los pesos se congelan durante el entrenamiento

Keras nos brinda la capacidad de crear una función `train_step` personalizada para implementar esta lógica. El ejemplo 4-7 muestra la clase de modelo DCGAN completa.

Ejemplo 4-7. Compilando la DCGAN

```
class DCGAN(models.Model):
    def __init__(self, discriminator, generator, latent_dim):
        super(DCGAN, self).__init__()
        self.discriminator = discriminator
        self.generator = generator
        self.latent_dim = latent_dim
```

```

    def compile(self, d_optimizer, g_optimizer):
        super(DCGAN, self).compile()
        self.loss_fn = losses.BinaryCrossentropy()
        self.d_optimizer = d_optimizer
        self.g_optimizer = g_optimizer
        self.d_loss_metric = metrics.Mean(name="d_loss")
        self.g_loss_metric = metrics.Mean(name="g_loss")

    @property
    def metrics(self):
        return [self.d_loss_metric, self.g_loss_metric]

    def train_step(self, real_images):
        batch_size = tf.shape(real_images)[0]
        random_latent_vectors = tf.random.normal(
            shape=(batch_size, self.latent_dim)
        )
        with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
            generated_images = self.generator(
                random_latent_vectors, training = True
            )
            real_predictions = self.discriminator(real_images,
training = True)
            fake_predictions = self.discriminator(
                generated_images, training = True
            )
            real_labels = tf.ones_like(real_predictions)
            real_noisy_labels = real_labels + 0.1 *tf.random.uniform(
                tf.shape(real_predictions)
            )
            fake_labels = tf.zeros_like(fake_predictions)
            fake_noisy_labels = fake_labels - 0.1 *
tf.random.uniform(
                tf.shape(fake_predictions)
            )
            d_real_loss = self.loss_fn(real_noisy_labels,
real_predictions)
            d_fake_loss = self.loss_fn(fake_noisy_labels,
fake_predictions)
            d_loss = (d_real_loss + d_fake_loss) / 2.0
            g_loss = self.loss_fn(real_labels, fake_predictions)
            gradients_of_discriminator = disc_tape.gradient(
                d_loss, self.discriminator.trainable_variables
            )
            gradients_of_generator = gen_tape.gradient(
                g_loss, self.generator.trainable_variables
            )

```

```

        self.d_optimizer.apply_gradients(
            zip(gradients_of_discriminator,
                 discriminator.trainable_variables)
        )
        self.g_optimizer.apply_gradients(
            zip(gradients_of_generator,
                 generator.trainable_variables)
        )
        self.d_loss_metric.update_state(d_loss)
        self.g_loss_metric.update_state(g_loss)
    return {m.name: m.result() for m in self.metrics}

dcgan = DCGAN(
    discriminator=discriminator, generator=generator,
    latent_dim=100
)

dcgan.compile(
    d_optimizer=optimizers.Adam(
        learning_rate=0.0002, beta_1 = 0.5, beta_2 = 0.999
    ),
    g_optimizer=optimizers.Adam(
        learning_rate=0.0002, beta_1 = 0.5, beta_2 = 0.999
    ),
)
dcgan.fit(train, epochs=300)

```

1. La función de pérdida para el generador y el discriminador es Cruzentropía binaria.
2. Para entrenar la red, primero tome muestras de un lote de vectores de una distribución normal estándar multivariada.
3. A continuación, páselos por el generador para producir un lote de imágenes generadas.
4. Ahora pídale al discriminador que prediga la realidad del lote de imágenes reales...
...y el lote de imágenes generadas.
5. La pérdida del discriminador es la entropía cruzada binaria promedio entre las imágenes reales (con etiqueta 1) y las imágenes falsas (con etiqueta 0).
6. La pérdida del generador es la entropía cruzada binaria entre las predicciones del discriminador para las imágenes generadas y una etiqueta de 1.

7. Actualice los pesos del discriminador y del generador por separado.

El discriminador y el generador luchan constantemente por el dominio, lo que puede hacer que el proceso de entrenamiento de DCGAN sea inestable. Idealmente, el proceso de entrenamiento encontrará un equilibrio que permita al generador aprender información significativa del discriminador y la calidad de las imágenes comenzará a mejorar. Después de suficientes épocas, el discriminador tiende a terminar dominando, como se muestra en la figura 4-6, pero esto puede no ser un problema ya que es posible que el generador ya haya aprendido a producir imágenes de calidad suficientemente alta en este punto.

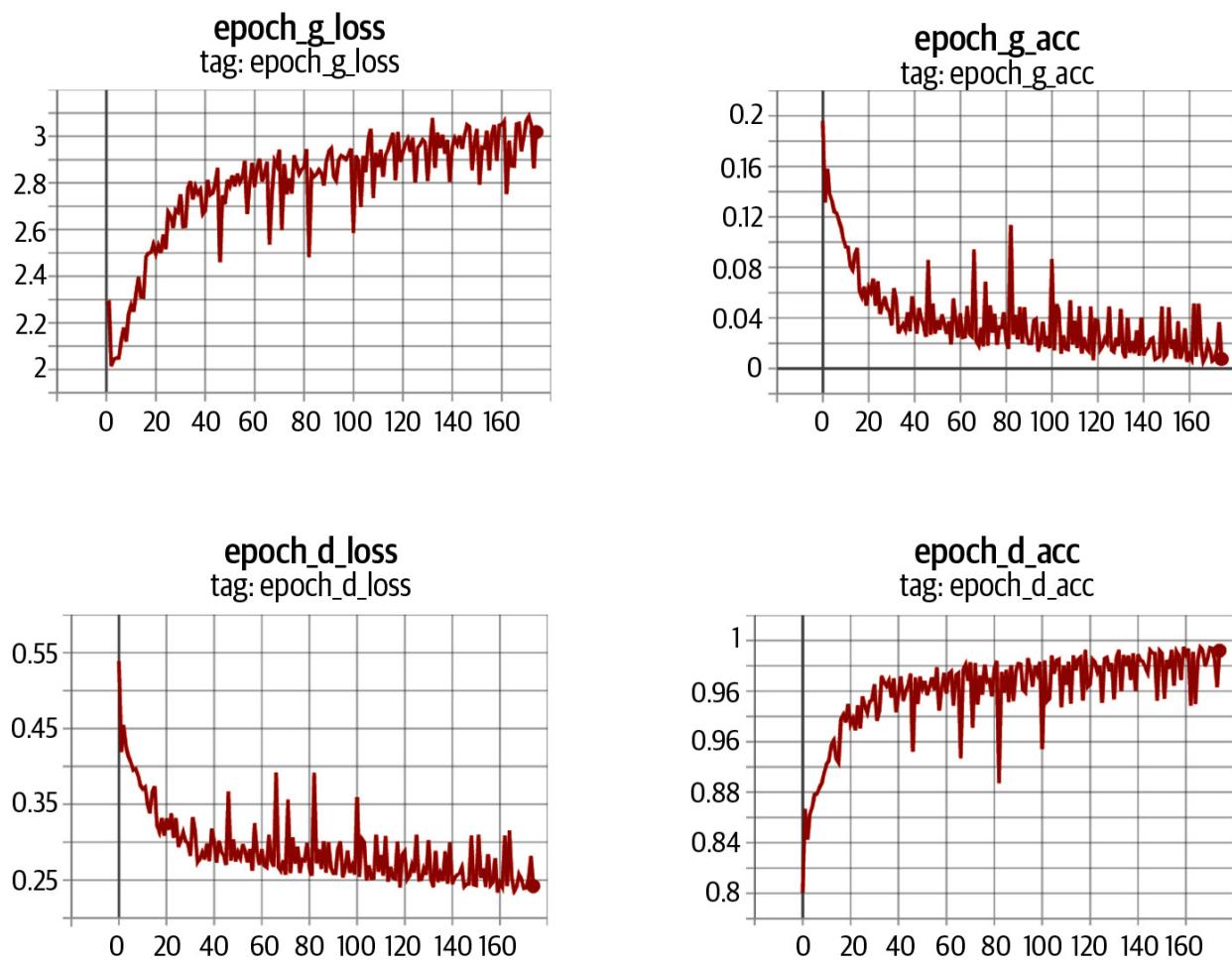


Figura 4-6. Pérdida y precisión del discriminador y generador durante el entrenamiento.

AÑADIR RUIDO A LAS ETIQUETAS

Un truco útil al entrenar GAN es agregar una pequeña cantidad de ruido aleatorio a las etiquetas de entrenamiento. Esto ayuda a mejorar la estabilidad del proceso de entrenamiento y a mejorar la nitidez de las imágenes generadas.

Este suavizado de etiquetas actúa como una forma de domesticar al discriminador, de modo que se le presente una tarea más desafiante y no domine al generador.

Análisis del DCGAN

Al observar las imágenes producidas por el generador en épocas específicas durante el entrenamiento (Figura 4-7), queda claro que el generador se está volviendo cada vez más experto en producir imágenes que podrían haberse extraído del conjunto de entrenamiento.

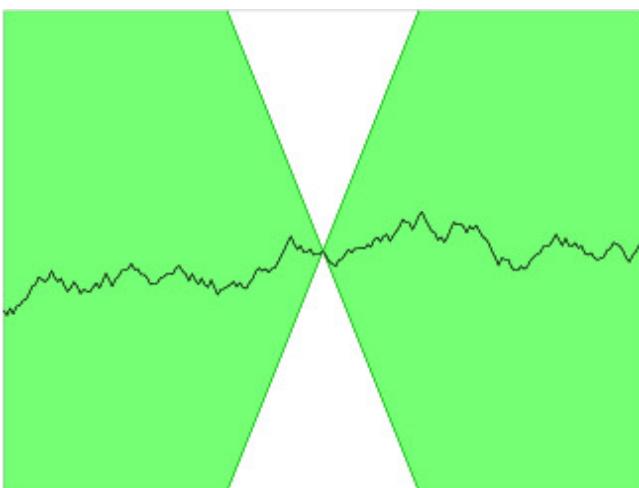


Figura 4-7. Salida del generador en épocas específicas durante el entrenamiento.

Es algo milagroso que una red neuronal sea capaz de convertir ruido aleatorio en algo significativo. Vale la pena recordar que no le hemos proporcionado al modelo ninguna característica adicional más allá de los píxeles sin procesar, por lo que tiene que desarrollar conceptos de alto nivel, como cómo dibujar sombras, cuboides y círculos, completamente por sí solo.

Otro requisito de un modelo generativo exitoso es que no solo reproduzca imágenes del conjunto de entrenamiento.

Para probar esto, podemos encontrar la imagen del conjunto de entrenamiento más cercana a un ejemplo generado en particular. Una buena medida de la distancia es la distancia L1, definida como:

```
def compare_images(img1, img2):
    return np.mean(np.abs(img1 - img2))
```

La Figura 4-8 muestra las observaciones más cercanas en el conjunto de entrenamiento para una selección de imágenes generadas. Podemos ver que si bien existe cierto grado de similitud entre las imágenes generadas y el conjunto de entrenamiento, no son idénticas. Esto muestra que el generador ha comprendido estas características de alto nivel y puede generar ejemplos distintos de los que ya ha visto.

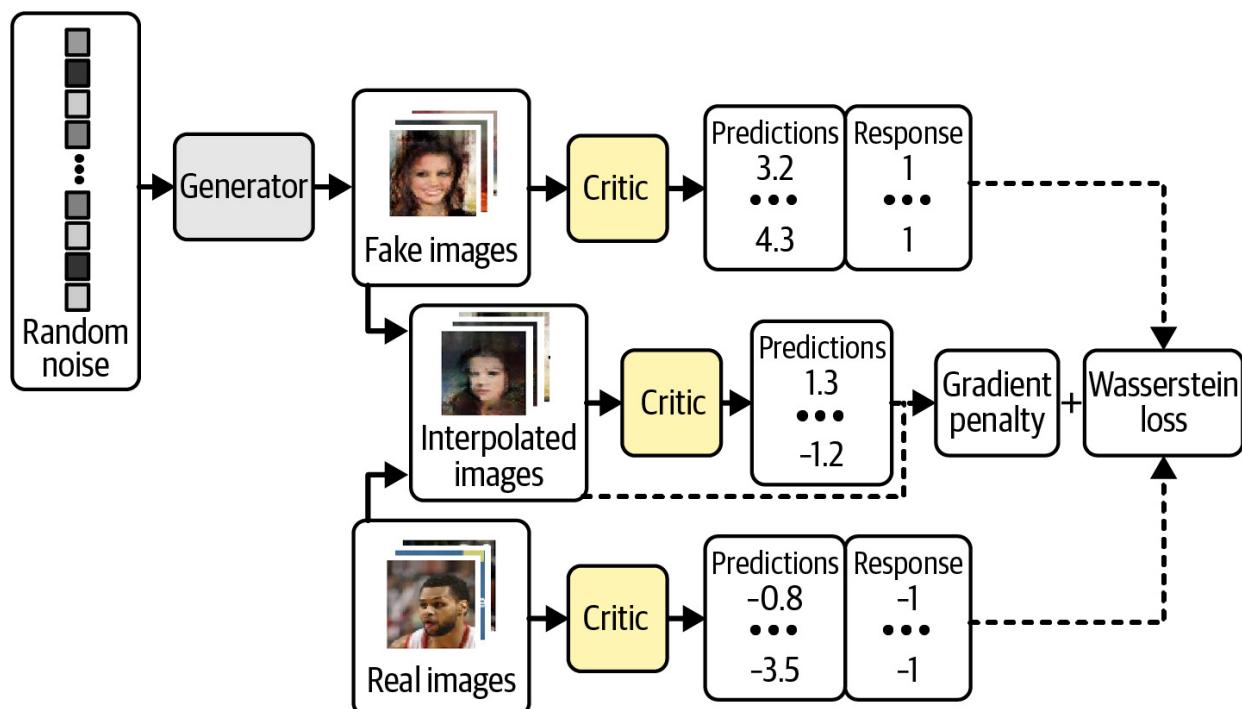


Figura 4-8. Coincidencias más cercanas de imágenes generadas a partir del conjunto de entrenamiento

Entrenamiento GAN: consejos y trucos

Si bien las GAN son un gran avance para el modelado generativo, también son notoriamente difíciles de entrenar. Exploraremos algunos de los problemas y desafíos más comunes que se encuentran al entrenar GAN en esta sección, junto con posibles soluciones. En la siguiente sección, veremos algunos ajustes más fundamentales al marco GAN que podemos realizar para remediar muchos de estos problemas.

El discriminador domina al generador.

Si el discriminador se vuelve demasiado fuerte, la señal de la función de pérdida se vuelve demasiado débil para impulsar mejoras significativas en el generador. En el peor de los casos, el discriminador aprende perfectamente a separar las imágenes reales de las falsas y los gradientes desaparecen por completo, sin necesidad de entrenamiento alguno, como se puede ver en la Figura 4-9.

Figura 4-9. Ejemplo de salida cuando el discriminador domina al generador

Si descubre que la función de pérdida de su discriminador colapsa, necesita encontrar formas de debilitarlo. Pruebe las siguientes sugerencias:

Aumente el parámetro de velocidad de las capas Dropout en el discriminador para amortiguar la cantidad de información que fluye a través de la red.

Reducir la tasa de aprendizaje del discriminador.

Reducir el número de filtros convolucionales en el discriminador.

Agregue ruido a las etiquetas al entrenar al discriminador.

Voltee las etiquetas de algunas imágenes al azar cuando entrene al discriminador.

El generador domina al discriminador

Si el discriminador no es lo suficientemente potente, el generador encontrará formas de engañarlo fácilmente con una pequeña muestra de imágenes casi idénticas. Esto se conoce como colapso modal.

Por ejemplo, supongamos que entrenáramos el generador en varios lotes sin actualizar el discriminador en el medio. El generador se inclinaría a encontrar una sola observación (también conocida como moda) que siempre engaña al discriminador y comenzaría a asignar cada punto en el espacio de entrada latente a esta imagen. Además, los gradientes de la función de pérdida colapsarían hasta cerca de 0, por lo que no podría recuperarse de este estado.

Incluso si luego intentáramos volver a entrenar al discriminador para que deje de ser engañado por este punto, el generador simplemente encontraría otro modo que engañe al discriminador, dado que ya se ha vuelto insensible a su entrada y, por lo tanto, no tiene incentivos para diversificar su producción.

El efecto del colapso modal se puede ver en la Figura 4-10.

Figura 4-10. Ejemplo de colapso de modo cuando el generador domina al discriminador

Si descubre que su generador sufre un colapso de modo, puede intentar fortalecer el discriminador utilizando las sugerencias opuestas a las enumeradas en la sección anterior. Además, puede intentar reducir la tasa de aprendizaje de ambas redes y aumentar el tamaño del lote.

Pérdida no informativa

Dado que el modelo de aprendizaje profundo se compila para minimizar la función de pérdida, sería natural pensar que cuanto menor sea la función de pérdida del generador, mejor será la calidad de las imágenes producidas. Sin embargo, dado que el generador solo se clasifica frente al discriminador actual y el discriminador mejora constantemente, no podemos comparar la función de pérdida evaluada en diferentes puntos del proceso de entrenamiento. De hecho, en la Figura 4-6, la función de pérdida del generador en realidad aumenta con el tiempo, aunque la calidad de las imágenes está mejorando claramente. Esta falta de correlación entre la pérdida del generador y la calidad de la imagen a veces hace que el entrenamiento de GAN sea difícil de monitorear.

Hiperparámetros

Como hemos visto, incluso con GAN simples, hay una gran cantidad de hiperparámetros que ajustar. Además de la arquitectura general tanto del discriminador como del generador, hay que considerar los parámetros que gobiernan la normalización de lotes, el abandono, la tasa de aprendizaje, las capas de activación, los filtros convolucionales, el tamaño del núcleo, el paso a paso, el tamaño de lote y el tamaño del espacio latente. Las GAN son muy sensibles a cambios muy leves en todos estos parámetros, y encontrar un conjunto de parámetros que funcione suele ser un caso de prueba y error informado, en lugar de seguir un conjunto de pautas establecidas.

Por eso es importante comprender el funcionamiento interno de GAN y saber cómo interpretar la función de pérdida, para poder identificar ajustes sensibles a los hiperparámetros que podrían mejorar la estabilidad del modelo.

Afrontar los desafíos de GAN

En los últimos años, varios avances clave han mejorado drásticamente la estabilidad general de los modelos GAN y han disminuido la probabilidad de que se produzcan algunos de los problemas enumerados anteriormente, como el colapso del modo.

En lo que resta de este capítulo examinaremos la GAN Wasserstein con penalización de gradiente (WGAN-GP), que realiza varios ajustes clave al marco GAN que hemos explorado hasta ahora para mejorar la estabilidad y la calidad del proceso de generación de imágenes.

GAN Wasserstein con penalización de gradiente (WGAN-GP)

En esta sección construiremos un WGAN-GP para generar rostros a partir del conjunto de datos de CelebA que utilizamos en el Capítulo 3.

EJECUTANDO EL CÓDIGO PARA ESTE EJEMPLO

El código de este ejemplo se puede encontrar en el cuaderno de Jupyter ubicado en notebooks/04_gan/02_wgan_gp/wgan_gp.ipynb en el repositorio de libros.

El código ha sido adaptado del excelente tutorial WGAN-GP creado por Aakash Kumar Nain, disponible en el sitio web de Keras.

Wasserstein GAN (WGAN), presentado en un artículo de 2017 por Arjovsky et al.⁴, fue uno de los primeros grandes pasos hacia la estabilización del entrenamiento GAN. Con algunos cambios, los autores pudieron mostrar cómo entrenar GAN que tienen las dos propiedades siguientes (citadas del artículo):

Una métrica de pérdida significativa que se correlaciona con la convergencia del generador y la estabilidad mejorada de calidad de la muestra del proceso de optimización.

Específicamente, el artículo presenta la función de pérdida de Wasserstein tanto para el discriminador como para el generador.

El uso de esta función de pérdida en lugar de la entropía cruzada binaria da como resultado una convergencia más estable de la GAN.

En esta sección definiremos la función de pérdida de Wasserstein y luego veremos qué otros cambios necesitamos realizar en la arquitectura del modelo y el proceso de entrenamiento para incorporar nuestra nueva función de pérdida.

Puede encontrar la clase de modelo completa en el cuaderno de Jupyter ubicado en el capítulo 05/wgan-gp/faces/train.ipynb en el repositorio de libros.

Pérdida de Wasserstein

Primero recordemos la definición de pérdida de entropía cruzada binaria, la función que estamos usando actualmente para entrenar al discriminador y generador de GAN (Ecuación 4-1).

Ecuación 4-1. Pérdida de entropía cruzada binaria

$$\sum (y_i \log(p_i) + (1 - y_i) \log(1 - p_i)) i=1$$

Para entrenar el discriminador GAN D, calculamos la pérdida al comparar predicciones para imágenes reales $p_i = D(x_i)$ con la respuesta $y_i = 1$ y predicciones para imágenes generadas $p_i = D(G(z_i))$ con la respuesta $y_i = 0$. Por lo tanto, para el discriminador GAN, que minimiza la función de pérdida, se puede escribir como se muestra en la Ecuación 4-2.

Ecuación 4-2. Minimización de pérdida del discriminador GAN

$$\begin{aligned} \min - & (E_{x \sim p} \\ & [\log D(x)] + E_{z \sim p} \\ & [\log(1 - D(G(z)))]) \end{aligned}$$

Para entrenar el generador GAN G, calculamos la pérdida al comparar las predicciones para las imágenes generadas $p_i = D(G(z_i))$ con la respuesta $y_i = 1$. Por lo tanto, para el generador GAN, minimizar la función de pérdida se puede escribir como se muestra en la Ecuación 4-3.

Ecuación 4-3. Minimización de pérdidas del generador GAN

$$\begin{aligned} \min - & (E_{z \sim p} \\ & [\log D(G(z))]) \end{aligned}$$

Ahora comparemos esto con la función de pérdida de Wasserstein.

Primero, la pérdida de Wasserstein requiere que usemos $y_i = 1$ y $y_i = -1$ como etiquetas, en lugar de 1 y 0. También eliminamos la activación sigmoidea de la capa final del discriminador, de modo que las predicciones p_i ya no estén obligadas a caer en el rango $[0, 1]$ pero ahora puede ser cualquier número en el rango $(-\infty, \infty)$. Por esta razón, el discriminador en una WGAN generalmente se hace referencia como un crítico que genera una puntuación en lugar de una probabilidad.

La función de pérdida de Wasserstein se define de la siguiente manera:

$$\sum (y_i p_i)_{i=1}^n$$

Para entrenar al crítico WGAN D, calculamos la pérdida al comparar predicciones para imágenes reales $p_i = D(x_i)$ con la respuesta $y_i = 1$ y predicciones para imágenes generadas $p_i = D(G(z_i))$ con la respuesta $y_i = -1$. Por lo tanto, para el crítico de WGAN, minimizando la función de pérdida se puede escribir de la siguiente manera:

$$\min - (E_{x \sim p}$$

$$[D(x)] - E_{z \sim p}$$

$$[D(G(z))])$$

En otras palabras, el crítico de WGAN intenta maximizar la diferencia entre sus predicciones para imágenes reales y las imágenes generadas.

Para entrenar el generador WGAN, calculamos la pérdida al comparar las predicciones para las imágenes generadas $p_i = D(G(z_i))$ con la respuesta $y_i = 1$. Por lo tanto, para el generador WGAN, minimizar la función de pérdida se puede escribir de la siguiente manera:

$$\min - (E_{z \sim p}$$

$$[D(G(z))])$$

En otras palabras, el generador WGAN intenta producir imágenes que el crítico puntúe lo más alto posible (es decir, se engaña al crítico haciéndole creer que son reales).

La restricción de Lipschitz

Puede que le sorprenda que ahora permitamos que el crítico genere cualquier número en el rango $(-\infty, \infty)$, en lugar de aplicar una función sigmoidea para restringir la salida al rango habitual $[0, 1]$. Por lo tanto, la pérdida de Wasserstein puede ser muy grande, lo cual es inquietante: ¡normalmente se deben evitar grandes cantidades en las redes neuronales!

De hecho, los autores del artículo de WGAN muestran que para la función de pérdida de Wasserstein funcione, también debemos imponer una restricción adicional a la crítica. Específicamente, se requiere que la crítica sea una función continua de 1-Lipschitz.

Separaremos esto para entender lo que significa con más detalle.

La crítica es una función D que convierte una imagen en una predicción. Decimos que esta función es 1-Lipschitz si satisface la siguiente desigualdad para dos imágenes de entrada cualesquiera, x_1 y x_2 : $|D(x_1) - D(x_2)| \leq 1$

$$|x_1 - x_2|$$

Aquí, $|x_1 - x_2|$ es la diferencia absoluta promedio en píxeles entre dos imágenes y $|D(x_1) - D(x_2)|$ es la diferencia absoluta entre las predicciones críticas. Esencialmente, requerimos un límite en la velocidad a la que las predicciones del crítico pueden cambiar entre dos imágenes (es decir, el valor absoluto del gradiente debe ser como máximo 1 en todas partes). Podemos ver esto aplicado a una función 1D continua de Lipschitz en la Figura 4-11: en ningún punto la línea entra al cono, dondequieras que coloques el cono en la línea. En otras palabras, existe un límite en la velocidad a la que la línea puede subir o bajar en cualquier punto.

Figura 4-11. Una función continua de Lipschitz (fuente: Wikipedia)

CONSEJO

Para aquellos que quieran profundizar en la lógica matemática detrás de por qué la pérdida de Wasserstein sólo funciona cuando se aplica esta restricción, Jonathan Hui ofrece una excelente explicación.

Haciendo cumplir la restricción de Lipschitz

En el artículo original de WGAN, los autores muestran cómo es posible hacer cumplir la restricción de Lipschitz recortando los pesos del crítico para que se encuentren dentro de un rango pequeño, $[-0,01, 0,01]$, después de cada lote de entrenamiento.

Una de las críticas a este enfoque es que la capacidad del crítico para aprender se ve muy disminuida, ya que estamos recortando su peso. De hecho, incluso en el artículo original de WGAN los autores escriben: “El recorte de peso es claramente una terrible manera de imponer una restricción de Lipschitz”. Una crítica fuerte es fundamental para el éxito de una WGAN, ya que sin gradientes precisos, el generador no puede aprender a adaptar sus pesos para producir mejores muestras.

Por lo tanto, otros investigadores han buscado formas alternativas de imponer la restricción de Lipschitz y mejorar la capacidad de la WGAN para aprender características complejas. Uno de esos métodos es Wasserstein GAN con penalización de gradiente.

En el artículo que presenta esta variante,⁵ los autores muestran cómo la restricción de Lipschitz se puede aplicar directamente incluyendo un término de penalización de gradiente en la función de pérdida para el crítico que penaliza al modelo si la norma de gradiente se desvía de 1. Esto resulta en una proceso de entrenamiento estable.

En la siguiente sección, veremos cómo incorporar este término adicional en la función de pérdida de nuestro crítico.

La pérdida de penalización por gradiente

La Figura 4-12 es un diagrama del proceso de entrenamiento del crítico de un WGAN-GP. Si comparamos esto con el proceso de entrenamiento del discriminador original de la Figura 4-5, podemos ver que la adición clave es la pérdida de penalización de gradiente incluida como parte de la función de pérdida general, junto con la pérdida de Wasserstein por las imágenes reales y falsas.

Figura 4-12. El proceso de formación crítica de WGAN-GP

La pérdida de penalización de gradiente mide la diferencia al cuadrado entre la norma del gradiente de las predicciones con respecto a las imágenes de entrada y 1. Naturalmente, el modelo se inclinará a encontrar pesos que

garanticen que el término de penalización de gradiente se minimice, fomentando así que el modelo se ajuste a la restricción de Lipschitz.

Es difícil calcular este gradiente en todas partes durante el proceso de entrenamiento, por lo que WGAN-GP evalúa el gradiente solo en un puñado de puntos. Para garantizar una mezcla equilibrada, utilizamos un conjunto de imágenes interpoladas que se encuentran en puntos elegidos al azar a lo largo de líneas que conectan el lote de imágenes reales con el lote de imágenes falsas por pares, como se muestra en la Figura 4-13.

Figura 4-13. Interpolando entre imágenes

En el ejemplo 4-8, mostramos cómo se calcula la penalización de gradiente en el código.

Ejemplo 4-8. La función de pérdida de penalización de gradiente

```
def gradient_penalty(self, batch_size, real_images,
fake_images):
    alpha = tf.random.normal([batch_size, 1, 1, 1], 0.0, 1.0)
    diff = fake_images - real_images
    interpolated = real_images + alpha * diff

    with tf.GradientTape() as gp_tape:
        gp_tape.watch(interpolated)
        pred = self.critic(interpolated, training=True)

    grads = gp_tape.gradient(pred, [interpolated])[0]
    norm = tf.sqrt(tf.reduce_sum(tf.square(grads), axis=[1, 2,
    3]))
    gp = tf.reduce_mean((norm - 1.0) ** 2)
    return gp
```

1. Cada imagen del lote recibe un número aleatorio, entre 0 y 1, almacenado como vector alfa.
2. Se calcula un conjunto de imágenes interpoladas.
3. Se pide al crítico que califique cada una de estas imágenes interpoladas.
4. El gradiente de las predicciones se calcula con respecto a las imágenes de entrada.

5. Se calcula la norma L2 de este vector.
6. La función devuelve la distancia promedio al cuadrado entre la norma L2 y 1.

Entrenando la WGAN-GP

Un beneficio clave de usar la función de pérdida de Wasserstein es que ya no necesitamos preocuparnos por equilibrar el entrenamiento del crítico y el generador; de hecho, cuando usamos la función de pérdida de Wasserstein, el crítico debe estar entrenado para la convergencia antes de actualizar el generador, para garantizar que los gradientes para la actualización del generador sean precisos. Esto contrasta con una GAN estándar, donde es importante no permitir que el discriminador se vuelva demasiado fuerte.

Por lo tanto, con las GAN de Wasserstein, podemos simplemente entrenar al crítico varias veces entre actualizaciones del generador, para asegurarnos de que esté cerca de la convergencia. Una proporción típica utilizada es de tres a cinco actualizaciones críticas por actualización del generador.

Ahora hemos introducido los dos conceptos clave detrás de WGAN-GP: la pérdida de Wasserstein y el término de penalización de gradiente que se incluye en la función de pérdida crítica. El paso de entrenamiento del modelo WGAN que incorpora todas estas ideas se muestra en el Ejemplo 4-9.

Ejemplo 4-9. Entrenamiento de WGAN-GP

```
def train_step(self, real_images):
    batch_size = tf.shape(real_images)[0]
    for i in range(3):
        random_latent_vectors = tf.random.normal(
            shape=(batch_size, self.latent_dim)
        )

        with tf.GradientTape() as tape:
            fake_images = self.generator(
                random_latent_vectors, training = True
            )
            fake_predictions = self.critic(fake_images, training =
True)
            real_predictions = self.critic(real_images, training =
```

```

    True)

        c_wass_loss = tf.reduce_mean(fake_predictions) -
tf.reduce_mean(
            real_predictions
)
c_gp = self.gradient_penalty(
            batch_size, real_images, fake_images
)
c_loss = c_wass_loss + c_gp * self.gp_weight

        c_gradient = tape.gradient(c_loss,
self.critic.trainable_variables)
        self.c_optimizer.apply_gradients(
            zip(c_gradient, self.critic.trainable_variables)
)
random_latent_vectors = tf.random.normal(
            shape=(batch_size, self.latent_dim)
)
with tf.GradientTape() as tape:
    fake_images = self.generator(random_latent_vectors,
training=True)
    fake_predictions = self.critic(fake_images, training=True)
    g_loss = -tf.reduce_mean(fake_predictions)

        gen_gradient = tape.gradient(g_loss,
self.generator.trainable_variables)
        self.g_optimizer.apply_gradients(
            zip(gen_gradient, self.generator.trainable_variables)
)

self.c_loss_metric.update_state(c_loss)
self.c_wass_loss_metric.update_state(c_wass_loss)
self.c_gp_metric.update_state(c_gp)
self.g_loss_metric.update_state(g_loss)
return {m.name: m.result() for m in self.metrics}

```

1. Realice tres actualizaciones críticas.
2. Calcule la pérdida de Wasserstein para el crítico: la diferencia entre la predicción promedio de las imágenes falsas y las imágenes reales.
3. Calcule el término de penalización del gradiente (consulte el Ejemplo 4-8).
4. La función de pérdida crítica es una suma ponderada de la pérdida de Wasserstein y la penalización por gradiente.
5. Actualiza los pesos de la crítica.

6. Calcule la pérdida de Wasserstein para el generador.
7. Actualiza los pesos del generador.

NORMALIZACIÓN DE LOTES EN UN WGAN-GP

Una última consideración que debemos tener en cuenta antes de entrenar un WGAN-GP es que la normalización por lotes no debe usarse en la fase crítica. Esto se debe a que la normalización por lotes crea una correlación entre imágenes del mismo lote, lo que hace que la penalización de gradiente sea menos efectiva.

Los experimentos han demostrado que los WGAN-GP aún pueden producir resultados excelentes incluso sin la normalización por lotes en el crítico.

Ahora hemos cubierto todas las diferencias clave entre una GAN estándar y una WGAN-GP. Recordar:

- Un WGAN-GP utiliza la pérdida de Wasserstein.
- El WGAN-GP se entrena utilizando etiquetas de 1 de verdad y -1 por falso.
- No hay activación sigmoidea en la capa final de la crítica.
- Incluya un término de penalización de gradiente en la función de pérdida del crítico.
- Entrena al crítico varias veces para cada actualización del generador.
- No hay capas de normalización por lotes en el crítico.

Análisis de la WGAN-GP

Echemos un vistazo a algunos ejemplos de salidas del generador, después de 25 épocas de entrenamiento (Figura 4-14).

Figura 4-14. Ejemplos de caras de WGAN-GP

El modelo ha aprendido los atributos significativos de alto nivel de una cara y no hay signos de colapso del modo.

También podemos ver cómo las funciones de pérdida del modelo evolucionan con el tiempo (Figura 4-15): las funciones de pérdida tanto del crítico como del generador son altamente estables y convergentes.

Si comparamos la salida WGAN-GP con la salida VAE del capítulo anterior, podemos ver que las imágenes GAN son generalmente más nítidas, especialmente la definición entre el cabello y el fondo. Esto es cierto en general; Los VAE tienden a producir imágenes más suaves que desdibujan los límites de color, mientras que se sabe que las GAN producen imágenes más nítidas y bien definidas.

Figura 4-15. Curvas de pérdida WGAN-GP: la pérdida crítica (epoch_c_loss) se descompone en pérdida de Wasserstein (epoch_c_wass) y pérdida de penalización de gradiente (epoch_c_gp)

También es cierto que las GAN son generalmente más difíciles de entrenar que las VAE y tardan más en alcanzar una calidad satisfactoria. Sin embargo, muchos modelos generativos de última generación hoy en día están basados en GAN, ya que las recompensas por entrenar GAN a gran escala en GPU durante un período de tiempo más largo son significativas.

GAN condicional (CGAN)

Hasta ahora en este capítulo, hemos creado GAN que pueden generar imágenes realistas a partir de un conjunto de entrenamiento determinado.

Sin embargo, no hemos podido controlar el tipo de imagen que nos gustaría generar: por ejemplo, un rostro masculino o femenino, o un ladrillo grande o pequeño. Podemos muestrear un punto aleatorio del espacio latente, pero no tenemos la capacidad de comprender fácilmente qué tipo de imagen se producirá si se elige la variable latente.

En la parte final de este capítulo centraremos nuestra atención en la construcción de una GAN donde podamos controlar la salida, la llamada GAN condicional. Esta idea, introducida por primera vez en “Redes adversas

generativas condicionales” por Mirza y Osindero en 2014,⁶ es una extensión relativamente simple de la arquitectura GAN.

EJECUTANDO EL CÓDIGO PARA ESTE EJEMPLO

El código de este ejemplo se puede encontrar en el cuaderno de Jupyter ubicado en notebooks/04_gan/03_cgan/cgan.ipynb en el repositorio de libros.

El código ha sido adaptado del excelente tutorial CGAN creado por Sayak Paul, disponible en el sitio web de Keras.

Arquitectura CGAN

En este ejemplo, condicionaremos nuestro CGAN al atributo de cabello rubio del conjunto de datos de caras. Es decir, podremos especificar explícitamente si queremos generar una imagen con el pelo rubio o no. Esta etiqueta se proporciona como parte del conjunto de datos de CelebA.

La arquitectura CGAN de alto nivel se muestra en la Figura 4-16.

Figura 4-16. Entradas y salidas del generador y crítico en un CGAN

La diferencia clave entre una GAN estándar y una CGAN es que en una CGAN pasamos información adicional al generador y al crítico relacionada con la etiqueta. En el generador, esto simplemente se agrega a la muestra del espacio latente como un vector codificado en caliente. En la crítica, agregamos la información de la etiqueta como canales adicionales a la imagen RGB. Hacemos esto repitiendo el vector codificado one-hot para llenar la misma forma que las imágenes de entrada.

Los CGAN funcionan porque el crítico ahora tiene acceso a información adicional sobre el contenido de la imagen, por lo que el generador debe asegurarse de que su salida coincida con la etiqueta proporcionada, para seguir engañando al crítico. Si el generador produjera imágenes perfectas

que no coincidieran con la etiqueta de la imagen, el crítico podría decir que eran falsas simplemente porque las imágenes y las etiquetas no coincidían.

CONSEJO

En nuestro ejemplo, nuestra etiqueta codificada en caliente tendrá una longitud de 2, porque hay dos clases (Rubia y No Rubia). Sin embargo, puedes tener tantas etiquetas como quieras; por ejemplo, puedes entrenar una CGAN en el conjunto de datos Fashion-MNIST para generar uno de los 10 artículos de moda diferentes, incorporando un vector de etiquetas codificadas en caliente de longitud 10 en la entrada del generador y 10 canales de etiquetas codificadas en caliente adicionales en la entrada del crítico.

El único cambio que debemos hacer en la arquitectura es concatenar la información de la etiqueta con las entradas existentes del generador y el crítico, como se muestra en el Ejemplo 4-10.

Ejemplo 4-10. Capas de entrada en el CGAN

```
critic_input = layers.Input(shape=(64, 64, 3))
label_input = layers.Input(shape=(64, 64, 2))
x = layers.concatenate(axis = -1)([critic_input, label_input])
...
generator_input = layers.Input(shape=(32,))
label_input = layers.Input(shape=(2,))
x = layers.concatenate(axis = -1)([generator_input,
label_input])
x = layers.Reshape((1,1, 34))(x)
```

1. Los canales de imagen y los canales de etiquetas se pasan por separado al crítico y se concatenan.
2. El vector latente y las clases de etiqueta se pasan por separado al generador y se concatenan antes de remodelarlos.

Entrenando a la CGAN

También debemos hacer algunos cambios en el `train_step` de la CGAN para que coincida con los nuevos formatos de entrada del generador y crítico, como se muestra en el Ejemplo 4-11.

Ejemplo 4-11. El train_step del CGAN

```
def train_step(self, data):
    real_images, one_hot_labels = data

    image_one_hot_labels = one_hot_labels[:, None, None, :]
    image_one_hot_labels = tf.repeat(
        image_one_hot_labels, repeats=64, axis = 1
    )
    image_one_hot_labels = tf.repeat(
        image_one_hot_labels, repeats=64, axis = 2
    )

    batch_size = tf.shape(real_images)[0]

    for i in range(self.critic_steps):
        random_latent_vectors = tf.random.normal(
            shape=(batch_size, self.latent_dim)
        )

        with tf.GradientTape() as tape:
            fake_images = self.generator(
                [random_latent_vectors, one_hot_labels], training =
True
            )

            fake_predictions = self.critic([fake_images,
image_one_hot_labels], training =
True
            )
            real_predictions = self.critic(
                [real_images, image_one_hot_labels], training =
True
            )
            c_wass_loss = tf.reduce_mean(fake_predictions) -
tf.reduce_mean(
            real_predictions
)
            c_gp = self.gradient_penalty(
                batch_size, real_images, fake_images,
image_one_hot_labels
            )
            c_loss = c_wass_loss + c_gp * self.gp_weight

            c_gradient = tape.gradient(c_loss,
self.critic.trainable_variables)
            self.c_optimizer.apply_gradients(
                zip(c_gradient, self.critic.trainable_variables)
```

```

    )

random_latent_vectors = tf.random.normal(
    shape=(batch_size, self.latent_dim)
)

with tf.GradientTape() as tape:
    fake_images = self.generator(
        [random_latent_vectors, one_hot_labels], training=True
    )
    fake_predictions = self.critic(
        [fake_images, image_one_hot_labels], training=True
    )
    g_loss = -tf.reduce_mean(fake_predictions)

    gen_gradient = tape.gradient(g_loss,
        self.generator.trainable_variables)
    self.g_optimizer.apply_gradients(
        zip(gen_gradient, self.generator.trainable_variables)
)

```

1. Las imágenes y etiquetas se descomprimen a partir de los datos de entrada.
2. Los vectores codificados en caliente se expanden a imágenes codificadas en caliente que tienen el mismo tamaño espacial que las imágenes de entrada (64×64).
3. El generador ahora recibe una lista de dos entradas: los vectores latentes aleatorios y los vectores de etiquetas codificados en caliente.
4. El crítico ahora cuenta con una lista de dos entradas: las imágenes falsas/reales y los canales de etiquetas codificadas en caliente.
5. La función de penalización de gradiente también requiere que se pasen los canales de etiquetas codificadas en caliente a medida que utiliza el crítico.
6. Los cambios realizados en el paso de entrenamiento crítico también se aplican al paso de entrenamiento del generador.

Análisis de la CGAN

Podemos controlar la salida de CGAN pasando una etiqueta codificada one-hot particular a la entrada del generador. Por ejemplo, para generar una cara con cabello no rubio, pasamos el vector [1, 0]. Para generar una cara con cabello rubio, pasamos el vector [0, 1].

El resultado de CGAN se puede ver en la Figura 4-17. Aquí, mantenemos los mismos vectores latentes aleatorios en todos los ejemplos y cambiamos solo el vector de etiqueta condicional. Está claro que CGAN ha aprendido a utilizar el vector de etiqueta para controlar únicamente el atributo de color de cabello de las imágenes. Es impresionante que el resto de la imagen apenas cambie; esto es una prueba de que las GAN pueden organizar puntos en el espacio latente de tal manera que las características individuales se pueden desacoplar entre sí.

Figura 4-17. Salida del CGAN cuando los vectores Rubio y No Rubio se añaden a la muestra latente

CONSEJO

Si hay etiquetas disponibles para su conjunto de datos, generalmente es una buena idea incluirlas como entrada a su GAN, incluso si no necesariamente necesita condicionar la salida generada en la etiqueta, ya que tienden a mejorar la calidad de las imágenes generadas. Puede pensar en las etiquetas simplemente como una extensión altamente informativa de la entrada de píxeles.

Resumen

En este capítulo exploramos tres modelos diferentes de redes generativas adversarias (GAN): el GAN convolucional profundo (DCGAN), la más sofisticada Wasserstein GAN con penalización de gradiente (WGAN-GP) y la GAN condicional (CGAN).

Todas las GAN se caracterizan por una arquitectura de generador versus discriminador (o crítico), en la que el discriminador intenta "detectar la diferencia" entre imágenes reales y falsas y el generador pretende engañar al discriminador. Al equilibrar la forma en que se entrena estos dos adversarios, el generador GAN puede aprender gradualmente a producir observaciones similares a las del conjunto de entrenamiento.

Primero vimos cómo entrenar un DCGAN para generar imágenes de ladrillos de juguete. Pudo aprender a representar de manera realista objetos 3D como imágenes, incluidas representaciones precisas de sombras, formas y texturas. También exploramos las diferentes formas en que el entrenamiento de GAN puede fallar, incluido el colapso del modo y los gradientes que desaparecen.

Luego exploramos cómo la función de pérdida de Wasserstein solucionó muchos de estos problemas e hizo que el entrenamiento de GAN fuera más predecible y confiable. WGAN-GP sitúa el requisito 1Lipschitz en el centro del proceso de formación incluyendo un término en la función de pérdida para llevar la norma del gradiente hacia 1.

Aplicamos WGAN-GP al problema de la generación de rostros y vimos cómo simplemente eligiendo puntos de una distribución normal estándar, podemos generar rostros nuevos. Este proceso de muestreo es muy similar a un VAE, aunque las caras producidas por una GAN son bastante diferentes: a menudo más nítidas y con una mayor distinción entre las diferentes partes de la imagen.

Finalmente, construimos un CGAN que nos permitió controlar el tipo de imagen que se genera. Esto funciona pasando la etiqueta como entrada al crítico y al generador, brindando así a la red la información adicional que necesita para condicionar la salida generada en una etiqueta determinada.

En general, hemos visto cómo el marco GAN es extremadamente flexible y puede adaptarse a muchos dominios de problemas interesantes. En particular, las GAN han impulsado avances significativos en el campo de la generación de imágenes con muchas extensiones interesantes al marco subyacente, como veremos en el Capítulo 10.

En el próximo capítulo, exploraremos una familia diferente de modelos generativos que es ideal para modelar datos secuenciales: los modelos autorregresivos.

1. Ian J. Goodfellow et al., “Generative Adversarial Nets”, 10 de junio de 2014, <https://arxiv.org/abs/1406.2661>

2. Alec Radford et al., “Aprendizaje de representación no supervisado con Redes adversarias generativas convolucionales profundas”, 7 de enero de 2016, <https://arxiv.org/abs/1511.06434>.
3. Augustus Odena et al., “Deconvolution and Checkerboard Artifacts”, 17 de octubre de 2016, <https://distill.pub/2016/deconv-checkerboard>.
4. Martin Arjovsky et al., “Wasserstein GAN”, 26 de enero de 2017, <https://arxiv.org/abs/1701.07875>.
5. Ishaan Gulrajani et al., “Improved Training of Wasserstein GANs”, marzo 31, 2017, <https://arxiv.org/abs/1704.00028>.
6. Mehdi Mirza y Simon Osindero, “Conditional Generative Adversarial Nets”, 6 de noviembre de 2014, <https://arxiv.org/abs/1411.1784>.

Capítulo 5. Modelos autoregresivos

METAS DEL CAPÍTULO

En este capítulo podrá:

- Descubrir por qué los modelos autorregresivos son adecuados para generar datos secuenciales, como texto.
- Aprender a procesar y tokenizar datos de texto.
- Conocer el diseño arquitectónico de redes neuronales recurrentes (RNN).
- Crear y entrenar una red de memoria a corto plazo (LSTM) desde cero utilizando Keras.
- Utilizar la LSTM para generar texto nuevo.
- Conocer otras variaciones de RNN, incluidas las unidades recurrentes cerradas (GRU) y las celdas bidireccionales.
- Comprender cómo se pueden tratar los datos de una imagen como una secuencia de píxeles.
- Conocer el diseño arquitectónico de una PixelCNN.
- Crear una PixelCNN desde cero usando Keras.
- Utilizar la PixelCNN para generar imágenes.

Hasta ahora, hemos explorado dos familias diferentes de modelos generativos que han involucrado variables latentes: codificadores automáticos variacionales (VAE) y redes generativas adversarias (GAN). En ambos casos, se introduce una nueva variable con una distribución de la que es fácil tomar muestras y el modelo aprende cómo decodificar esta variable nuevamente en el dominio original.

Ahora centraremos nuestra atención en los modelos autorregresivos, una familia de modelos que simplifican el problema del modelado generativo, tratándolo como un proceso secuencial.

Los modelos autorregresivos condicionan las predicciones a valores anteriores de la secuencia, en lugar de a una variable aleatoria latente. Por

lo tanto, intentan modelar explícitamente la distribución generadora de datos en lugar de una aproximación de la misma (como en el caso de los VAE).

En este capítulo exploraremos dos modelos autorregresivos diferentes: redes de memoria a corto plazo y PixelCNN. Aplicaremos el LSTM a datos de texto y el PixelCNN a datos de imágenes. Cubriremos en detalle otro modelo autorregresivo de gran éxito, el Transformador, en el Capítulo 9.

Introducción

Para entender cómo funciona un LSTM, primero haremos una visita a una extraña prisión, donde los reclusos han formado una sociedad literaria...

LA SOCIEDAD LITERARIA PARA LOS MALHECHORES PROBLEMÁTICOS

Edward Sopp odiaba su trabajo como director de prisión. Pasaba sus días cuidando a los prisioneros y no tenía tiempo para seguir su verdadera pasión: escribir cuentos. Se estaba quedando sin inspiración y necesitaba encontrar una manera de generar contenido nuevo.

Un día, se le ocurrió una idea brillante que le permitiría producir nuevas obras de ficción en su estilo, manteniendo a la vez ocupados a los reclusos: ¡haría que los reclusos escribieran colectivamente las historias para él! Calificó a la nueva sociedad como Sociedad Literaria para malhechores problemáticos, o LSTM (Figura 5-1).

Figura 5-1. Una gran celda de prisioneros leyendo libros (creada con Midjourney)

La prisión es particularmente extraña porque sólo consta de una celda grande que contiene 256 presos.

Cada prisionero tiene una opinión sobre cómo debería continuar la historia actual de Edward. Todos los días, Edward publica la última palabra de su novela en la celda, y es el trabajo de los internos para actualizar individualmente sus opiniones sobre el estado actual de la historia, en base a la nueva palabra y las opiniones de los internos del día anterior.

Cada prisionero utiliza un proceso de pensamiento específico para actualizar su propia opinión, lo que implica equilibrar la información de la nueva palabra entrante y las opiniones de otros prisioneros con sus propias creencias anteriores. Primero, deciden qué parte de la opinión de ayer desean olvidar, teniendo en cuenta la información de la nueva palabra y las opiniones de otros prisioneros en la celda. También utilizan esta información para formar nuevos pensamientos y decidir hasta qué punto quieren mezclarlos con las viejas creencias que han elegido conservar del día anterior. Esto forma entonces la nueva opinión del prisionero para ese día.

Sin embargo, los prisioneros son reservados y no siempre cuentan a sus compañeros de prisión todas sus opiniones. Cada uno de ellos también utiliza la última palabra elegida y las opiniones de los demás reclusos para decidir qué parte de su opinión desean revelar.

Cuando Edward quiere que la celda genere la siguiente palabra en la secuencia, cada uno de los prisioneros le cuenta sus opiniones revelables al guardia en la puerta, quien combina esta información para finalmente decidir la siguiente palabra que se agregará al final de la novela. Esta nueva palabra luego regresa a la celda y el proceso continúa hasta que se completa la historia completa.

Para entrenar a los reclusos y al guardia, Edward introduce en la celda secuencias cortas de palabras que ha escrito previamente y controla si la siguiente palabra elegida por los reclusos es correcta. Los actualiza sobre su precisión y poco a poco empiezan a aprender a escribir historias con su propio estilo único.

Después de muchas iteraciones de este proceso, Edward descubre que el sistema ha logrado generar texto de apariencia realista. Satisfecho con los

resultados, publica una colección de los cuentos generados en su nuevo libro, titulado Fábulas de E. Sopo.

La historia del Sr. Sopo y sus fábulas colaborativas es una analogía de una de las técnicas autorregresivas más notorias para datos secuenciales como el texto: la red de memoria a largo plazo.

Red de memoria a corto plazo (LSTM)

Una LSTM es un tipo particular de red neuronal recurrente (RNN). Las RNN contienen una capa (o celda) recurrente que es capaz de manejar datos secuenciales haciendo que su propia salida en un paso de tiempo particular forme parte de la entrada al siguiente paso de tiempo.

Cuando se introdujeron por primera vez las RNN, las capas recurrentes eran muy simples y consistían únicamente en un operador \tanh que garantizaba que la información pasada entre pasos de tiempo se escalara entre -1 y 1 . Sin embargo, se demostró que este enfoque adolece del problema del gradiente de desaparición y no escala bien a largas secuencias de datos.

Las células LSTM se introdujeron por primera vez en 1997 en un artículo de Sepp Hochreiter y Jürgen Schmidhuber.¹ En el artículo, los autores describen cómo los LSTM no sufren el mismo problema de gradiente de fuga que experimentan las RNN vainilla y se pueden entrenar en secuencias que son cientos de pasos de tiempo largos. Desde entonces, la arquitectura LSTM se ha adaptado y mejorado, y variaciones como las unidades recurrentes cerradas (que se analizan más adelante en este capítulo) ahora se utilizan ampliamente y están disponibles como capas en Keras.

Los LSTM se han aplicado a una amplia gama de problemas que involucran datos secuenciales, incluido el pronóstico de series temporales, el análisis de sentimientos y la clasificación de audio. En este capítulo utilizaremos LSTM para abordar el desafío de la generación de texto.

EJECUTANDO EL CÓDIGO PARA ESTE EJEMPLO

El código de este ejemplo se puede encontrar en el cuaderno de Jupyter ubicado en notebooks/05_autoregressive/01_lstm/lstm.ipynb en el repositorio de libros.

El conjunto de datos de recetas

Usaremos el conjunto de datos de Epicurious Recipes que está disponible a través de Kaggle. Se trata de un conjunto de más de 20.000 recetas, acompañadas de metadatos, como información nutricional y listas de ingredientes.

Puede descargar el conjunto de datos ejecutando el script de descarga del conjunto de datos de Kaggle en el repositorio de libros, como se muestra en el Ejemplo 5-1. Esto guardará las recetas y los metadatos que las acompañan localmente en la carpeta /data.

Ejemplo 5-1. Descargando el conjunto de datos de recetas de Epicurio

```
bash scripts/download_kaggle_data.sh hugodarwood epirecipes
```

El ejemplo 5-2 muestra cómo se pueden cargar y filtrar los datos para que solo queden recetas con un título y una descripción. En el Ejemplo 5-3 se proporciona un ejemplo de una cadena de texto de receta.

Ejemplo 5-2. Cargando los datos

```
with open('/app/data/epirecipes/full_format_recipes.json') as
    json_data:
        recipe_data = json.load(json_data)
        filtered_data = [
            'Recipe for ' + x['title'] + ' | ' + ' '.join(x['directions'])
            for x in recipe_data
            if 'title' in x
            and x['title'] is not None
            and 'directions' in x
            and x['directions'] is not None
        ]
```

Ejemplo 5-3. Una cadena de texto del conjunto de datos de Recetas

Receta de Persillade de Jamón con Ensalada de Patata a la Mostaza y Puré de Guisantes | Pica suficientes hojas de perejil para medir 1 cucharada; reservar. Pique las hojas y los tallos restantes y cocine a fuego lento con caldo y ajo en una cacerola pequeña, tapado, durante 5 minutos.

Mientras tanto, espolvorea la gelatina sobre agua en un tazón mediano y deja que se ablande durante 1 minuto.

Cuele el caldo a través de un colador de malla fina y viértalo en un recipiente con gelatina y revuelva para que se disuelva. Condimentar con sal y pimienta. Coloque el tazón en un baño de hielo y enfrié a temperatura ambiente, revolviendo. Mezcle el jamón con el perejil reservado y divídalo en frascos.

Vierta la gelatina encima y enfrié hasta que cuaje, al menos 1 hora. Batir la mayonesa, la mostaza, el vinagre, 1/4 de cucharadita de sal y 1/4 de cucharadita de pimienta en un tazón grande. Agregue el apio, los pepinillos y las papas. Pulsos de guisantes con mejorana, aceite, 1/2 cucharadita de pimienta y

1/4 cucharadita de sal en un procesador de alimentos hasta obtener un puré grueso. Coloque capas de guisantes y luego ensalada de papas sobre el jamón.

Antes de ver cómo construir una red LSTM en Keras, primero debemos dar un rápido rodeo para comprender la estructura de los datos de texto y en qué se diferencian de los datos de imágenes que hemos visto hasta ahora en este libro.

Trabajar con datos de texto

Existen varias diferencias clave entre los datos de texto y de imagen que significan que muchos de los métodos que funcionan bien para los datos de imagen no son tan fácilmente aplicables a los datos de texto. En particular:

- Los datos de texto se componen de fragmentos discretos (ya sean caracteres o palabras), mientras que los píxeles de una imagen son puntos en un espectro de color continuo. Podemos hacer fácilmente que un píxel verde sea más azul, pero no es obvio cómo debemos hacer para que la palabra gato se parezca más a la palabra perro, por ejemplo. Esto significa que podemos aplicar fácilmente la retropropagación a los datos de la imagen, ya que podemos calcular el gradiente de nuestra función de pérdida con respecto a píxeles individuales para establecer la dirección en la que se deben cambiar los

colores de los píxeles para minimizar la pérdida. Con datos de texto discretos, obviamente no podemos aplicar la retropropagación de la misma manera, por lo que necesitamos encontrar una manera de solucionar este problema.

- Los datos de texto tienen una dimensión temporal pero no una dimensión espacial, mientras que los datos de imágenes tienen dos dimensiones espaciales pero ninguna dimensión temporal. El orden de las palabras es muy importante en los datos de texto y las palabras no tendrían sentido al revés, mientras que las imágenes normalmente se pueden invertir sin afectar el contenido. Además, a menudo existen dependencias secuenciales a largo plazo entre palabras que el modelo debe capturar: por ejemplo, la respuesta a una pregunta o la transmisión del contexto de un pronombre. Con los datos de imagen, todos los píxeles se pueden procesar simultáneamente.
- Los datos de texto son muy sensibles a pequeños cambios en las unidades individuales (palabras o caracteres). Los datos de imagen generalmente son menos sensibles a los cambios en las unidades de píxeles individuales: la imagen de una casa aún sería reconocible como una casa incluso si se modificaron algunos píxeles, pero con datos de texto, cambiar incluso unas pocas palabras puede alterar drásticamente el significado del pasaje o hacerlo sin sentido. Esto hace que sea muy difícil entrenar un modelo para generar un texto coherente, ya que cada palabra es vital para el significado general del pasaje.
- Los datos de texto tienen una estructura gramatical basada en reglas, mientras que los datos de imágenes no siguen reglas establecidas sobre cómo se deben asignar los valores de píxeles. Por ejemplo, no tendría sentido gramatical en ningún contexto escribir "El gato se sentó sobre el tener". También existen reglas semánticas que son extremadamente difíciles de modelar; No tendría sentido decir "Estoy en la playa", aunque gramaticalmente no hay nada malo en esta afirmación.

AVANCES EN APRENDIZAJE PROFUNDO GENERATIVO BASADO EN TEXTO

Hasta hace poco, la mayoría de los modelos generativos de aprendizaje profundo más sofisticados se han centrado en datos de imágenes, porque muchos de los desafíos presentados en la lista anterior estaban fuera del

alcance incluso de las técnicas más avanzadas. Sin embargo, en los últimos cinco años se han logrado avances sorprendentes en el campo del aprendizaje profundo generativo basado en texto, gracias a la introducción de la arquitectura del modelo de transformador, que exploraremos en el Capítulo 9.

Con estos puntos en mente, ahora echemos un vistazo a los pasos que debemos seguir para que los datos de texto tengan la forma correcta para entrenar una red LSTM.

Tokenización

El primer paso es limpiar y tokenizar el texto.

La *tokenización* es el proceso de dividir el texto en unidades individuales, como palabras o caracteres.

La forma en que tokenice su texto dependerá de lo que intente lograr con su modelo de generación de texto. Existen ventajas y desventajas en el uso de tokens de palabras y caracteres, y su elección afectará la forma en que necesita limpiar el texto antes del modelado y el resultado de su modelo.

Si usa tokens de palabras:

- Todo el texto se puede convertir a minúsculas, para garantizar que las palabras en mayúscula al comienzo de las oraciones se tokenicen de la misma manera que las mismas palabras que aparecen en el medio de una oración. Sin embargo, en algunos casos esto puede no ser deseable; por ejemplo, algunos nombres propios, como nombres o lugares, pueden beneficiarse de permanecer en mayúscula para que se tokenicen de forma independiente.
- El *vocabulario* del texto (el conjunto de palabras distintas en el conjunto de entrenamiento) puede ser muy amplio, y algunas palabras aparecen muy escasamente o quizás sólo una vez. Puede ser aconsejable reemplazar las palabras escasas con un token para palabras desconocidas, en lugar de incluirlas como tokens separados, para reducir la cantidad de pesos que la red neuronal necesita aprender.

- Las palabras pueden *derivarse*, lo que significa que se reducen a su forma más simple, de modo que los diferentes tiempos de un verbo permanezcan tokenizados juntos. Por ejemplo, explorar, explorar, explorar y explorar se derivarían de cejas.
- Deberá tokenizar la puntuación o eliminarla por completo.
- El uso de tokenización de palabras significa que el modelo nunca podrá predecir palabras fuera del vocabulario de entrenamiento.

Si usas fichas de personaje:

- El modelo puede generar secuencias de caracteres que forman nuevas palabras fuera del vocabulario de entrenamiento; esto puede ser deseable en algunos contextos, pero no en otros.
- Las letras mayúsculas se pueden convertir en minúsculas o permanecer como tokens separados.
- El vocabulario suele ser mucho más reducido cuando se utiliza la tokenización de caracteres. Esto es beneficioso para la velocidad de entrenamiento del modelo, ya que hay menos pesos que aprender en la capa de salida final.

Para este ejemplo, usaremos tokenización de palabras en minúsculas, sin derivación de palabras. También tokenizaremos los signos de puntuación, ya que nos gustaría que el modelo prediga cuándo debe terminar las oraciones o usar comas, por ejemplo.

El código del ejemplo 5-4 limpia y tokeniza el texto.

Ejemplo 5-4. Tokenización

```
def pad_punctuation(s):
    s = re.sub(f"([{string.punctuation}])", r' \1 ', s)
    s = re.sub(' +', ' ', s)
    return s

text_data = [pad_punctuation(x) for x in filtered_data]

text_ds =
tf.data.Dataset.from_tensor_slices(text_data).batch(32).shuffle
(100
0)
```

```

vectorize_layer = layers.TextVectorization(
    standardize = 'lower',
    max_tokens = 10000, output_mode = "int",
    output_sequence_length = 200 + 1,
)
vectorize_layer.adapt(text_ds)
vocab = vectorize_layer.get_vocabulary()

```

- Rellene los signos de puntuación para tratarlos como palabras separadas.
- Convierta a un conjunto de datos de TensorFlow.
- Cree una capa `TextVectorization` de Keras para convertir texto a minúsculas, asigne a las 10,000 palabras más frecuentes un token entero correspondiente y recorte o rellene la secuencia a 201 tokens de longitud.
- Aplique la capa `TextVectorization` a los datos de entrenamiento.
- La variable de vocabulario almacena una lista de tokens de palabras.

Un ejemplo de una receta después de la tokenización se muestra en el ejemplo 5-5. La longitud de la secuencia que utilizamos para entrenar el modelo es un parámetro del proceso de entrenamiento. En este ejemplo, elegimos usar una longitud de secuencia de 200, por lo que rellenamos o recortamos la receta a una longitud mayor que esta, para permitirnos crear la variable de destino (más sobre esto en la siguiente sección). Para lograr la longitud deseada, el final del vector se rellena con ceros.

PARAR FICHAS

El token 0 se conoce como token de parada, lo que significa que la cadena de texto ha llegado a su fin.

Ejemplo 5-5. La receta del ejemplo 5-3 tokenizada

[26	16	557	1	8	298	335	189
4		1054	494	27	332	228		
235	262		5	594	11	133	22	311
2		332	45	262	4	671		
4	70		8	171	4	81	6	9 65
80		3	121	3	59			
12	2	299		3	88	650	20	39

6	9	29	21	4	67		
529	11	164	2	320	171	102	9
374	13	643	306	25	21		
8	650	4	42	5	931	2	63
8	24	4	33	2	114		
21	6	178	181	1245	4	60	5
140	112	3	48	2	117		
557	8	285	235	4	200	292	980
2	107	650	28	72	4		
108	10	114	3	57	204	11	172
2	73	110	482	3	298		
3	190	3	11	23	32	142	24
3	4	11	23	32	142		
33	6	9	30	21	2	42	6
353	3	3224	3	4	150		
2	437	494	8	1281	3	37	3
11	23	15	142	33	3		
4	11	23	32	142	24	6	9
291	188	5	9	412	572		
2	230	494	3	46	335	189	3
20	557	2	0	0	0		
0	0	0	0	0]			

En el Ejemplo 5-6, podemos ver un subconjunto de la lista de tokens asignados a sus respectivos índices. La capa reserva el token 0 para el relleno (es decir, es el token de parada) y el token 1 para palabras desconocidas que quedan fuera de las 10,000 palabras principales (por ejemplo, *persillade*). A las otras palabras se les asignan tokens en orden de frecuencia. El número de palabras a incluir en el vocabulario también es un parámetro del proceso de formación. Cuantas más palabras incluyas, menos tokens desconocidos verás en el texto; sin embargo, su modelo deberá ser más grande para adaptarse al mayor tamaño del vocabulario.

Ejemplo 5-6. El vocabulario de la capa `TextVectorization` 0:

```

0:
1: [UNK]
2: .
3: ,
4: and
5: to6: in
7: the
8: with
9: a

```

Creando el conjunto de entrenamiento

Nuestro LSTM estará entrenado para predecir la siguiente palabra en una secuencia, dada una secuencia de palabras que preceden a este punto.

Por ejemplo, podríamos alimentar al modelo con fichas de pollo asado con hervido y esperaríamos que el modelo generara la siguiente palabra adecuada (por ejemplo, patatas, en lugar de plátanos).

Por lo tanto, podemos simplemente desplazar toda la secuencia en un token para crear nuestra variable objetivo.

El paso de generación del conjunto de datos se puede lograr con el código del Ejemplo 5-7.

Ejemplo 5-7. Creando el conjunto de datos de entrenamiento

```
def prepare_inputs(text):
    text = tf.expand_dims(text, -1)
    tokenized_sentences = vectorize_layer(text)
    x = tokenized_sentences[:, :-1]
    y = tokenized_sentences[:, 1:]
    return x, y

train_ds = text_ds.map(prepare_inputs)
```

1. Cree el conjunto de entrenamiento que consta de tokens de receta (la entrada) y el mismo vector desplazado en un token (el objetivo).

La arquitectura LSTM

La arquitectura del modelo LSTM general se muestra en la tabla 5-1. La entrada al modelo es una secuencia de tokens enteros y la salida es la probabilidad de cada palabra en el vocabulario de 10,000 palabras que aparece a continuación en la secuencia. Para comprender cómo funciona esto en detalle, necesitamos introducir dos nuevos tipos de capas, Incrustación y LSTM.

Capa (tipo)	Forma de salida	# de parámetros

Capa de entrada	(Ninguno Ninguno)	0
Incrustar	(Ninguno, Ninguno, 100)	1,000,000
LSTM	(Ninguno, Ninguno, 128)	117,248
Denso	(Ninguno, Ninguno, 10000)	1,290,000
Parámetros totales		2,407,248
Parámetros entrenables		2,407,248
Parámetros no entrenables		0

LA CAPA DE ENTRADA DEL LSTM

Tenga en cuenta que la capa de entrada no necesita que especifiquemos la longitud de la secuencia de antemano. Tanto el tamaño del lote como la longitud de la secuencia son flexibles (de ahí la forma (None, None)). Esto se debe a que todas las capas posteriores son independientes de la longitud de la secuencia que se pasa.

La capa de incrustación

Una capa de incrustación es esencialmente una tabla de búsqueda que convierte cada token entero en un vector de longitud `embedding_size`, como se muestra en la Figura 5-2. El modelo aprende los vectores de búsqueda como pesos. Por lo tanto, el número de pesos aprendidos por esta capa es igual al tamaño del vocabulario multiplicado por la dimensión del vector de incrustación (es decir, $10,000 \times 100 = 1,000,000$).

Figura 5-2. Una capa de incrustación es una tabla de búsqueda para cada token entero.

Incorporamos cada token entero en un vector continuo porque permite que el modelo aprenda una representación para cada palabra que se puede actualizar mediante retropropagación. También podríamos codificar en caliente cada token de entrada, pero se prefiere usar una capa de incrustación porque hace que la incrustación sea entrenable, lo que le da al

modelo más flexibilidad para decidir cómo incrustar cada token para mejorar su rendimiento.

Por lo tanto, la capa de Entrada pasa un tensor de secuencias enteras de forma [batch_size, seq_length] a la capa Embedding, que genera un tensor de forma [batch_size, seq_length, embedding_size]. Luego se pasa a la capa LSTM (Figura 5-3).

Figura 5-3. Una única secuencia que fluye a través de una capa de incrustación.

La capa LSTM

Para comprender la capa LSTM, primero debemos observar cómo funciona una capa recurrente general.

Una capa recurrente tiene la propiedad especial de poder procesar datos de entrada secuenciales x_1, \dots, x_n . Consiste en una celda que actualiza su estado oculto, h_t , a medida que cada elemento de la secuencia x_t pasa a través de ella, un paso de tiempo a la vez.

El estado oculto es un vector con una longitud igual al número de unidades en la celda; puede considerarse como la comprensión actual de la secuencia por parte de la celda. En el paso de tiempo t , la celda usa el valor anterior del estado oculto, h_{t-1} , junto con los datos del paso de tiempo actual x_t para producir un vector de estado oculto actualizado, h_t . Este proceso recurrente continúa hasta el final de la secuencia. Una vez finalizada la secuencia, la capa genera el estado oculto final de la celda, h_n , que luego se pasa a la siguiente capa de la red. Este proceso se muestra en la Figura 5-4.

Figura 5-4. Un diagrama simple de una capa recurrente.

Para explicar esto con más detalle, desenrollemos el proceso para que podamos ver exactamente cómo una única secuencia pasa a través de la

capa (Figura 5-5).

PESOS CELULARES

Es importante recordar que todas las celdas de este diagrama comparten los mismos pesos (ya que en realidad son la misma celda). No hay diferencia entre este diagrama y la Figura 5-4; es simplemente una forma diferente de dibujar la mecánica de una capa recurrente.

Figura 5-5. Cómo fluye una única secuencia a través de una capa recurrente

Aquí, representamos el proceso recurrente dibujando una copia de la celda en cada paso de tiempo y mostramos cómo el estado oculto se actualiza constantemente a medida que fluye a través de las celdas. Podemos ver claramente cómo el estado oculto anterior se combina con el punto de datos secuencial actual (es decir, el vector de palabra incrustado actual) para producir el siguiente estado oculto. La salida de la capa es el estado oculto final de la celda, después de que se haya procesado cada palabra en la secuencia de entrada.

ADVERTENCIA

El hecho de que la salida de la celda se llame estado oculto es una convención de nomenclatura desafortunada: en realidad no está oculto y no deberías pensar en ello como tal. De hecho, el último estado oculto es la salida general de la capa, y aprovecharemos el hecho de que podemos acceder al estado oculto en cada paso de tiempo individual más adelante en este capítulo.

La célula LSTM

Ahora que hemos visto cómo funciona una capa recurrente genérica, echemos un vistazo al interior de una celda LSTM individual.

El trabajo de la celda LSTM es generar un nuevo estado oculto, h_t , dado su estado oculto anterior, h_{t-1} , y la incrustación de palabras actual, x_t . En resumen, la longitud de h_t es igual a la cantidad de unidades en el LSTM. Este es un parámetro que se establece cuando defines la capa y no tiene nada que ver con la longitud de la secuencia.

ADVERTENCIA

Asegúrese de no confundir el término celda con unidad. Hay una celda en una capa LSTM que se define por la cantidad de unidades que contiene, de la misma manera que la celda de prisionero de nuestra historia anterior contenía muchos prisioneros. A menudo dibujamos una capa recurrente como una cadena de celdas desenrolladas, ya que ayuda a visualizar cómo se actualiza el estado oculto en cada paso de tiempo.

Una célula LSTM mantiene un estado celular, C_t , que puede considerarse como las creencias internas de la célula sobre el estado actual de la secuencia. Esto es distinto del estado oculto, h_t , que finalmente la celda genera después del último paso de tiempo. El estado de la celda tiene la misma longitud que el estado oculto (el número de unidades en la celda).

Miremos más de cerca una sola celda y cómo se actualiza el estado oculto (Figura 5-6).

Figura 5-6. Una celda LSTM

El estado oculto se actualiza en seis pasos:

1. El estado oculto del paso de tiempo anterior, h_{t-1} , y la incrustación de palabras actual, x_t , se concatenan y pasan a través de la puerta de olvido. Esta puerta es simplemente una capa Dense con matriz de pesos W_f , polarización b_f y una función de activación sigmoidea. El vector resultante, f_t , tiene una longitud igual al número de unidades de

la celda y contiene valores entre 0 y 1 que determinan qué parte del estado anterior de la celda, C_{t-1} , debe retenerse.

2. El vector concatenado también pasa a través de una puerta de entrada que, al igual que la puerta de olvido, es una capa Dense con matriz de pesos W_i , polarización b_i y una función de activación sigmoidea. La salida de esta puerta tiene una longitud igual al número de unidades en la celda y contiene valores entre 0 y 1 que determinan cuánta información nueva se agregará al estado anterior de la celda, C_{t-1} .
3. El vector concatenado se pasa a través de una capa Dense con matriz de pesos W_C , polarización b_C y una función de activación tanh para generar un vector C_t que contiene la nueva información que la célula quiere considerar conservar. También tiene una longitud igual al número de unidades de la celda y contiene valores entre -1 y 1.
4. f_t y C_{t-1} se multiplican por elementos y se suman a la multiplicación por elementos de i_t y C_t . Esto representa olvidar partes del estado celular anterior y luego agregar nueva información relevante para producir el estado celular actualizado, C_t .
5. El vector concatenado pasa a través de una puerta de salida: una capa Dense con matriz de pesos W_o , polarización b_o y una activación sigmoidea. El vector resultante, o_t , tiene una longitud igual al número de unidades en la celda y almacena valores entre 0 y 1 que determinan qué parte del estado actualizado de la celda, C_t , se debe generar desde la celda.
 1. o_t se multiplica elemento por elemento con el estado de celda actualizado, C_t , después de que se haya aplicado una activación tanh para producir el nuevo estado oculto, h_t .

LA CAPA LSTM DE KERAS

Toda esta complejidad está envuelta dentro del tipo de capa LSTM en Keras, ¡así que no tienes que preocuparte por implementarlo tú mismo!

Entrenando el LSTM

El código para construir, compilar y entrenar el LSTM se proporciona en el ejemplo 5-8.

Ejemplo 5-8. Construyendo, compilando y entrenando LSTM

```
inputs = layers.Input(shape=(None, ), dtype="int32")
x = layers.Embedding(10000, 100)(inputs)
x = layers.LSTM(128, return_sequences=True)(x)
outputs = layers.Dense(10000, activation = 'softmax')(x)
lstm = models.Model(inputs, outputs)

loss_fn = losses.SparseCategoricalCrossentropy()
lstm.compile("adam", loss_fn)
lstm.fit(train_ds, epochs=25)
```

1. La capa de Entrada no necesita que especifiquemos la longitud de la secuencia por adelantado (puede ser flexible), por lo que usamos None como marcador de posición.
2. La capa Embedding requiere dos parámetros, el tamaño del vocabulario (10,000 tokens) y la dimensionalidad del vector de incrustación (100).
3. Las capas LSTM requieren que especifiquemos la dimensionalidad del vector oculto (128). También elegimos devolver la secuencia completa de estados ocultos, en lugar de solo el estado oculto en el paso de tiempo final.
4. La capa Dense transforma los estados ocultos en cada paso de tiempo en un vector de probabilidades para el siguiente token.
5. El modelo general predice el siguiente token, dada una secuencia de entrada de tokens. Hace esto para cada token de la secuencia.
6. El modelo está compilado con SparseCategoricalCrossentropy: esto es lo mismo que entropía cruzada categórica, pero se usa cuando las etiquetas son números enteros en lugar de vectores codificados en caliente.
7. El modelo se ajusta al conjunto de datos de entrenamiento.

En la Figura 5-7 puede ver las primeras épocas del proceso de entrenamiento de LSTM; observe cómo el resultado del ejemplo se vuelve más comprensible a medida que disminuye la métrica de pérdida. La Figura 5-8 muestra la métrica de pérdida de entropía cruzada que cae a lo largo del proceso de entrenamiento.

Figura 5-7. Las primeras épocas del proceso de formación LSTM.

Figura 5-8. La métrica de pérdida de entropía cruzada del proceso de entrenamiento de LSTM por época

Análisis del LSTM

Ahora que hemos compilado y entrenado el LSTM, podemos comenzar a usarlo para generar largas cadenas de texto aplicando el siguiente proceso:

1. Alimente la red con una secuencia existente de palabras y pídale que prediga la siguiente palabra.
2. Agregue esta palabra a la secuencia existente y repita.

La red generará un conjunto de probabilidades para cada palabra de la que podamos tomar muestras. Por lo tanto, podemos hacer que la generación de texto sea estocástica, en lugar de determinista. Además, podemos introducir un parámetro de temperatura en el proceso de muestreo para indicar qué tan determinista nos gustaría que fuera el proceso.

EL PARÁMETRO DE TEMPERATURA

Una temperatura cercana a 0 hace que el muestreo sea más determinista (es decir, es muy probable que se elija la palabra con mayor probabilidad), mientras que una temperatura de 1 significa que cada palabra se elige con la probabilidad generada por el modelo.

Esto se logra con el código del Ejemplo 5-9, que crea una función de devolución de llamada que se puede usar para generar texto al final de cada época de entrenamiento.

Ejemplo 5-9. La función de retrollamada TextGenerator

```

class TextGenerator(callbacks.Callback):
    def __init__(self, index_to_word, top_k=10):
        self.index_to_word = index_to_word
        self.word_to_index = {
            word: index for index, word in
            enumerate(index_to_word)
        }

    def sample_from(self, probs, temperature):
        probs = probs ** (1 / temperature)
        probs = probs / np.sum(probs)
        return np.random.choice(len(probs), p=probs), probs

    def generate(self, start_prompt, max_tokens,
                temperature):
        start_tokens = [
            self.word_to_index.get(x, 1) for x in
            start_prompt.split()
        ]
        sample_token = None
        info = []
        while len(start_tokens) < max_tokens and sample_token
        != 0:
            x = np.array([start_tokens])y = self.model.predict(x)
            sample_token, probs = self.sample_from(y[0][-1],
            temperature)
            info.append({'prompt': start_prompt , 'word_probs':
            probs})
            start_tokens.append(sample_token)
            start_prompt = start_prompt + ' ' +
            self.index_to_word[sample_token]
            print(f"\nGenerated text:\n{start_prompt}\n")
        return info

    def on_epoch_end(self, epoch, logs=None):
        self.generate("recipe for", max_tokens = 100,
        temperature =
        1.0)

```

1. Cree un mapeo de vocabulario inverso (de palabra a token).
2. Esta función actualiza las probabilidades con un factor de escala de temperatura.
3. El mensaje de inicio es una cadena de palabras que le gustaría darle al modelo para iniciar el proceso de generación (por ejemplo, receta). Las palabras primero se convierten en una lista de tokens.

4. La secuencia se genera hasta que tenga una longitud máxima de tokens o se produzca un token de parada (0).
5. El modelo genera las probabilidades de que cada palabra sea la siguiente en la secuencia.
6. Las probabilidades pasan a través del muestreador para generar la siguiente palabra, parametrizada por temperatura.
7. Agregamos la nueva palabra al texto del mensaje, listo para la siguiente iteración del proceso generativo.

Echemos un vistazo a esto en acción, con dos valores de temperatura diferentes (Figura 5-9).

Figura 5-9. Salidas generadas a temperatura = 1,0 y temperatura = 0,2

Hay algunas cosas a tener en cuenta sobre estos dos pasajes.

Primero, ambos son estilísticamente similares a una receta del conjunto de entrenamiento original. Ambos comienzan con el título de una receta y generalmente contienen construcciones gramaticalmente correctas. La diferencia es que el texto generado con una temperatura de 1.0 es más aventurero y por lo tanto menos preciso que el ejemplo con una temperatura de 0.2. Por lo tanto, generar múltiples muestras con una temperatura de 1,0 generará más variedad, ya que el modelo toma muestras de una distribución de probabilidad con mayor varianza.

Para demostrar esto, la Figura 5-10 muestra los cinco tokens principales con las probabilidades más altas para una variedad de indicaciones, para ambos valores de temperatura.

Figura 5-10. Distribución de probabilidades de palabras siguiendo varias secuencias, para valores de temperatura de 1,0 y 0,2

El modelo es capaz de generar una distribución adecuada para la siguiente palabra más probable en una variedad de contextos. Por ejemplo, aunque al modelo nunca se le informó sobre partes de la oración como sustantivos, verbos o números, generalmente es capaz de separar palabras en estas clases y usarlas de una manera gramaticalmente correcta.

Además, el modelo es capaz de seleccionar un verbo apropiado para comenzar las instrucciones de la receta, según el título anterior. Para las verduras asadas, selecciona precalentar, preparar, calentar, poner o combinar como las posibilidades más probables, mientras que para el helado selecciona agregar, combinar, revolver, batir y mezclar. Esto muestra que el modelo tiene cierta comprensión contextual de las diferencias entre recetas según sus ingredientes.

Observe también cómo las probabilidades de los ejemplos con temperatura = 0.2 tienen una ponderación mucho mayor hacia el token de primera opción. Ésta es la razón por la que generalmente hay menos variedad en las generaciones cuando la temperatura es más baja.

Si bien nuestro modelo LSTM básico está haciendo un gran trabajo generando texto realista, está claro que todavía tiene dificultades para captar parte del significado semántico de las palabras que está generando. ¡Introduce ingredientes que probablemente no combinan bien juntos (por ejemplo, papas japonesas agrias, migas de nueces y sorbete)! En algunos casos, esto puede ser deseable (por ejemplo, si queremos que nuestro LSTM genere patrones de palabras interesantes y únicos), pero en otros casos, necesitaremos que nuestro modelo tenga una comprensión más profunda de las formas en que se pueden agrupar las palabras. y una memoria más larga de las ideas introducidas anteriormente en el texto.

En la siguiente sección, exploraremos algunas de las formas en que podemos mejorar nuestra red LSTM básica. En el Capítulo 9, veremos un nuevo tipo de modelo autorregresivo, el Transformador, que lleva el modelado del lenguaje al siguiente nivel.

Extensiones de la red neuronal recurrente (RNN)

El modelo de la sección anterior es un ejemplo sencillo de cómo se puede entrenar un LSTM para que aprenda a generar texto en un estilo determinado. En esta sección exploraremos varias extensiones de esta idea.

Redes recurrentes apiladas

La red que acabamos de ver contenía una única capa LSTM, pero también podemos entrenar redes con capas LSTM apiladas, de modo que se puedan aprender características más profundas del texto.

Para lograr esto, simplemente introducimos otra capa LSTM después de la primera. La segunda capa LSTM puede usar los estados ocultos de la primera capa como datos de entrada. Esto se muestra en la Figura 5-11 y la arquitectura general del modelo se muestra en la Tabla 5-2.

Figura 5-11. Diagrama de un RNN multicapa: gt denota estados ocultos de la primera capa y ht denota estados ocultos de la segunda capa

Capa (tipo)	Forma de salida	# de parámetros
Capa de entrada	(Ninguno Ninguno)	0
Incrustar	(Ninguno, Ninguno, 100)	1,000,000
LSTM	(Ninguno, Ninguno, 128)	117,248
LSTM	(Ninguno, Ninguno, 128)	131,584
Denso	(None, None, 10000)	1,290,000
Parámetros totales	2,538,832	
Parámetros entrenables	2,538,832	
Parámetros no entrenables	0	

El código para construir el LSTM apilado se proporciona en el Ejemplo 510.

Ejemplo 5-10. Construyendo un LSTM apilado

```
text_in = layers.Input(shape = (None, ))
embedding = layers.Embedding(total_words, embedding_size)
(text_in)
x = layers.LSTM(n_units, return_sequences = True)(x)
x = layers.LSTM(n_units, return_sequences = True)(x)
probabilites = layers.Dense(total_words, activation =
'softmax')(x)
model = models.Model(text_in, probabilites)
```

Unidades recurrentes cerradas

Otro tipo de capa RNN comúnmente utilizada es la unidad recurrente cerrada (GRU).² Las diferencias clave con la unidad LSTM son las siguientes:

1. Las puertas de entrada y olvido se reemplazan por puertas de reinicio y actualización.
2. No hay estado de celda ni puerta de salida, solo un estado oculto que sale de la celda.

El estado oculto se actualiza en cuatro pasos, como se ilustra en la figura 5-12.

Figura 5-12. Una sola celda GRU

El proceso es el siguiente:

1. El estado oculto del paso de tiempo anterior, $ht-1$, y la incrustación de palabra actual, xt , se concatenan y se utilizan para crear la puerta de reinicio. Esta puerta es una capa Dense, con matriz de pesos Wr y una función de activación sigmoidea. El vector resultante, rt , tiene una longitud igual al número de unidades en la celda y almacena valores entre 0 y 1 que

determinan qué parte del estado oculto anterior, $ht-1$, debe trasladarse al cálculo de las nuevas creencias de la célula.

2. La puerta de reinicio se aplica al estado oculto, $ht-1$, y se concatena con la incrustación de palabra actual, xt . Luego, este vector se alimenta a una capa Dense con una matriz de pesos W y una función de activación tanh para generar un vector, \tilde{ht} , que almacena las nuevas creencias de la célula. Tiene una longitud igual al número de unidades de la celda y almacena valores entre -1 y 1.
3. La concatenación del estado oculto del paso de tiempo anterior, $ht-1$, y la incrustación de palabras actual, xt , también se utilizan para crear la puerta de actualización. Esta puerta es una capa Dense con matriz de pesos Wz y una activación sigmoidea. El vector resultante, zt , tiene una longitud igual al número de unidades en la celda y almacena valores entre 0 y 1, que se utilizan para determinar qué cantidad de las nuevas creencias, \tilde{ht} , se mezclarán con el estado oculto actual, $ht-1$.
4. Las nuevas creencias de la célula, \tilde{ht} , y el estado oculto actual, $ht-1$, se mezclan en una proporción determinada por la puerta de actualización, zt , para producir el estado oculto actualizado, ht , que se genera desde la celda.

Células bidireccionales

Para problemas de predicción en los que el texto completo está disponible para el modelo en el momento de la inferencia, no hay razón para procesar la secuencia solo en dirección hacia adelante; también podría procesarse hacia atrás. Una capa Bidirectional aprovecha esto almacenando dos conjuntos de estados ocultos: uno que se produce como resultado de que la secuencia se procesa en la dirección habitual hacia adelante y otro que se produce cuando la secuencia se procesa hacia atrás. De esta manera, la capa puede aprender de la información anterior y posterior al paso de tiempo determinado.

En Keras, esto se implementa como un contenedor alrededor de una capa recurrente, como se muestra en el Ejemplo 5-11.

Ejemplo 5-11. Construyendo una capa GRU bidireccional

```
layer = layers.Bidirectional(layers.GRU(100))
```

ESTADO OCULTO

Los estados ocultos en la capa resultante son vectores de longitud igual al doble del número de unidades en la celda envuelta (una concatenación de los estados ocultos hacia adelante y hacia atrás). Así, en este ejemplo los estados ocultos de la capa son vectores de longitud 200.

Hasta ahora, solo hemos aplicado modelos autorregresivos (LSTM) a datos de texto. En la siguiente sección, veremos cómo los modelos autorregresivos también se pueden utilizar para generar imágenes.

PixelCNN

En 2016, van den Oord et al. [3] introdujeron un modelo que genera imágenes píxel a píxel al predecir la probabilidad del siguiente píxel en función de los píxeles anteriores. El modelo se llama PixelCNN y se puede entrenar para generar imágenes de forma autorregresiva.

Hay dos conceptos nuevos que debemos introducir para comprender PixelCNN: capas convolucionales enmascaradas y bloques residuales.

EJECUTANDO EL CÓDIGO PARA ESTE EJEMPLO

El código para este ejemplo se puede encontrar en el cuaderno de Jupyter ubicado en notebooks/05_autoregressive/02_pixelcnn/pixelcnn.ipynb en el repositorio de libros.

El código ha sido adaptado del excelente tutorial de PixelCNN creado por ADMoreau, disponible en el sitio web de Keras.

Capas convolucionales enmascaradas

Como vimos en el Capítulo 2, se puede usar una capa convolucional para extraer características de una imagen aplicando una serie de filtros. La salida de la capa en un píxel particular es una suma ponderada de los pesos del filtro multiplicados por los valores de la capa anterior sobre un pequeño cuadrado centrado en el píxel. Este método puede detectar bordes y texturas y, en capas más profundas, formas y características de nivel superior.

Si bien las capas convolucionales son extremadamente útiles para la detección de características, no se pueden usar directamente en un sentido autorregresivo, porque no hay ningún orden en los píxeles. Se basan en el hecho de que todos los píxeles se tratan por igual: ningún píxel se trata como el inicio o el final de la imagen. Esto contrasta con los datos de texto que ya hemos visto en este capítulo, donde hay un orden claro de los tokens para que los modelos recurrentes como los LSTM se puedan aplicar fácilmente.

Para que podamos aplicar capas convolucionales a la generación de imágenes en un sentido autorregresivo, primero debemos ordenar los píxeles y asegurarnos de que los filtros solo puedan ver los píxeles que preceden al píxel en cuestión. Luego podemos generar imágenes un píxel a la vez, aplicando filtros convolucionales a la imagen actual para predecir el valor del siguiente píxel a partir de todos los píxeles anteriores.

Primero debemos elegir un orden para los píxeles; una sugerencia sensata es ordenar los píxeles de arriba a la izquierda a abajo a la derecha, moviéndonos primero a lo largo de las filas y luego hacia abajo en las columnas.

Luego enmascaramos los filtros convolucionales para que la salida de la capa en cada píxel solo esté influenciada por los valores de los píxeles que preceden al píxel en cuestión. Esto se logra multiplicando una máscara de unos y ceros con la matriz de pesos del filtro, de modo que los valores de cualquier píxel que esté después del píxel de destino se pongan a cero.

En realidad, hay dos tipos diferentes de máscaras en un PixelCNN, como se muestra en la Figura 5-13:

- Tipo A, donde se enmascara el valor del píxel central
- Tipo B, donde el valor del píxel central no está enmascarado

Figura 5-13. Izquierda: una máscara de filtro convolucionarial; derecha: una máscara aplicada a un conjunto de píxeles para predecir la distribución del valor del píxel central (fuente: van den Oord et al., 2016)

La capa convolucionarial enmascarada inicial (es decir, la que se aplica directamente a la imagen de entrada) no puede

usar el píxel central, porque este es precisamente el píxel que queremos que la red adivine. Sin embargo, las capas posteriores pueden utilizar el píxel central porque éste se habrá calculado únicamente como resultado de la información de los píxeles anteriores en la imagen de entrada original.

Podemos ver en el Ejemplo 5-12 cómo se puede construir un MaskedConvLayer usando Keras.

Ejemplo 5-12. Un MaskedConvLayer en Keras

```
class MaskedConvLayer(layers.Layer):
    def __init__(self, mask_type, **kwargs):
        super(MaskedConvLayer, self).__init__()
        self.mask_type = mask_type
        self.conv = layers.Conv2D(**kwargs)

    def build(self, input_shape):
        self.conv.build(input_shape)
        kernel_shape = self.conv.kernel.get_shape()
        self.mask = np.zeros(shape=kernel_shape)
        self.mask[: kernel_shape[0] // 2, ...] = 1.0
        self.mask[kernel_shape[0] // 2, : kernel_shape[1] // 2,
... ] = 1.0
        if self.mask_type == "B":
            self.mask[kernel_shape[0] // 2, kernel_shape[1] // 2,
... ] = 1.0

    def call(self, inputs):
        self.conv.kernel.assign(self.conv.kernel * self.mask)
        return self.conv(inputs)
```

1. MaskedConvLayer se basa en la capa Conv2D normal.
2. La máscara se inicializa con todos ceros.
3. Los píxeles de las filas anteriores se desenmascaran con unos.
4. Los píxeles de las columnas anteriores que están en la misma fila se desenmascaran con unos.
5. Si el tipo de máscara es B, el píxel central se desenmascara con un uno.
6. La máscara se multiplica por los pesos del filtro.

Tenga en cuenta que este ejemplo simplificado supone una imagen en escala de grises (es decir, con un canal). Si tenemos imágenes en color, tendremos tres canales de color en los que también podemos ordenar de modo que, por ejemplo, el canal rojo preceda al canal azul, que precede al canal verde.

Bloques residuales

Ahora que hemos visto cómo enmascarar la capa convolucional, podemos comenzar a construir nuestro PixelCNN. El bloque de construcción central que utilizaremos es el bloque residual.

Un bloque residual es un conjunto de capas donde la salida se agrega a la entrada antes de pasar al resto de la red. En otras palabras, la entrada tiene una ruta rápida hacia la salida, sin tener que pasar por las capas intermedias; esto se denomina conexión de salto. La razón detrás de incluir una conexión de salto es que si la transformación óptima es simplemente mantener la entrada igual, esto se puede lograr simplemente poniendo a cero los pesos de las capas intermedias. Sin la conexión de salto, la red tendría que encontrar un mapeo de identidad a través de las capas intermedias, lo cual es mucho más difícil.

En la Figura 5-14 se muestra un diagrama del bloque residual en nuestro PixelCNN.

Figura 5-14. Un bloque residual de PixelCNN (los números de filtros están al lado de las flechas y los tamaños de los filtros están al lado de las capas)

Podemos construir un ResidualBlock usando el código que se muestra en el ejemplo 5-13.

Ejemplo 5-13. Un bloque residual

```
class ResidualBlock(layers.Layer):
    def __init__(self, filters, **kwargs):
        super(ResidualBlock, self).__init__(**kwargs)
        self.conv1 = layers.Conv2D(
            filters=filters // 2, kernel_size=1, activation="relu"
        )
        self.pixel_conv = MaskedConv2D(
            mask_type="B",
            filters=filters // 2,
            kernel_size=3,
            activation="relu",
            padding="same",
        )
        self.conv2 = layers.Conv2D(
            filters=filters, kernel_size=1, activation="relu"
```

```

    )

def call(self, inputs):
    x = self.conv1(inputs)
    x = self.pixel_conv(x)
    x = self.conv2(x)
    return layers.add([inputs, x])

```

1. La capa Conv2D inicial reduce a la mitad el número de canales.
2. La capa MaskedConv2D tipo B con un tamaño de núcleo de 3 solo utiliza información de cinco píxeles: tres píxeles en la fila encima del píxel de enfoque, uno a la izquierda y el píxel de enfoque en sí.
3. La capa final Conv2D duplica el número de canales para volver a coincidir con la forma de entrada.
4. La salida de las capas convolucionales se agrega a la entrada; esta es la conexión de salto.

Entrenando al PixelCNN

En el Ejemplo 5-14 reunimos toda la red PixelCNN, siguiendo aproximadamente la estructura establecida en el artículo original. En el artículo original, la capa de salida es una Conv2D de 256 filtros, con activación softmax. En otras palabras, la red intenta recrear su entrada prediciendo los valores de píxeles correctos, un poco como un auto codificador. La diferencia es que PixelCNN está restringido de modo que ninguna información de píxeles anteriores puede fluir para influir en la predicción de cada píxel, debido a la forma en que está diseñada la red, utilizando capas MaskedConv2D.

Un desafío con este enfoque es que la red no tiene forma de entender que un valor de píxel de, digamos, 200 está muy cerca de un valor de píxel de 201. Debe aprender cada valor de salida de píxel de forma independiente, lo que significa que el entrenamiento puede ser muy lento. incluso para los conjuntos de datos más simples. Por lo tanto, en nuestra implementación, simplificamos la entrada para que cada píxel pueda tomar solo uno de cuatro valores. De esta manera, podemos usar una capa de salida Conv2D de 4 filtros en lugar de 256.

Ejemplo 5-14. La arquitectura PixelCNN

```

inputs = layers.Input(shape=(16, 16, 1))
x = MaskedConv2D(mask_type="A"

```

```

        , filters=128
        , kernel_size=7
        , activation="relu"
        , padding="same")(inputs)
    for _ in range(5):
        x = ResidualBlock(filters=128)(x)
    for _ in range(2):
        x = MaskedConv2D(
            mask_type="B",
            filters=128,
            kernel_size=1,
            strides=1,
            activation="relu",
            padding="valid",
        )(x)

    out = layers.Conv2D(
        filters=4, kernel_size=1, strides=1,
        activation="softmax",
        padding="valid"
    )(x)

pixel_cnn = models.Model(inputs, out)

adam = optimizers.Adam(learning_rate=0.0005)
pixel_cnn.compile(optimizer=adam,
loss="sparse_categorical_crossentropy")

pixel_cnn.fit(
    input_data
    , output_data
    , batch_size=128
    , epochs=150
)

```

1. La entrada del modelo es una imagen en escala de grises de tamaño $16 \times 16 \times 1$, con entradas escaladas entre 0 y 1.
2. La primera capa MaskedConv2D tipo A con un tamaño de núcleo de 7 utiliza información de 24 píxeles: 21 píxeles en las tres filas sobre el píxel de enfoque y 3 a la izquierda (el píxel de enfoque en sí no se utiliza).
3. Cinco grupos de capas ResidualBlock se apilan secuencialmente.
4. Dos capas MaskedConv2D de tipo B con un tamaño de núcleo de 1 actúan como capas densas en la cantidad de canales para cada píxel.

5. La capa final Conv2D reduce la cantidad de canales a cuatro: la cantidad de niveles de píxeles para este ejemplo.
6. El modelo está diseñado para aceptar una imagen y generar una imagen de las mismas dimensiones.
7. Ajustar el modelo: input_data se escala en el rango [0, 1] (flotantes); output_data se escala en el rango [0, 3] (enteros).

Análisis de PixelCNN

Podemos entrenar nuestro PixelCNN con imágenes del conjunto de datos FashionMNIST que encontramos en el Capítulo 3. Para generar nuevas imágenes, debemos pedirle al modelo que prediga el siguiente píxel dados todos los píxeles anteriores, un píxel a la vez. ¡Este es un proceso muy lento en comparación con un modelo como un auto codificador variacional! Para una imagen en escala de grises de 32×32 , necesitamos hacer 1024 predicciones secuencialmente usando el modelo, en comparación con la única predicción que necesitamos hacer un VAE. Esta es una de las principales desventajas de los modelos autorregresivos como PixelCNN: el muestreo es lento debido a la naturaleza secuencial del proceso de muestreo.

Por este motivo, utilizamos un tamaño de imagen de 16×16 , en lugar de 32×32 , para acelerar la generación de nuevas imágenes. La clase de devolución de llamada de generación se muestra en el Ejemplo 5-15.

Ejemplo 5-15. Generando nuevas imágenes usando el PixelCNN

```
class ImageGenerator(callbacks.Callback):
    def __init__(self, num_img):
        self.num_img = num_img

    def sample_from(self, probs, temperature):
        probs = probs ** (1 / temperature)
        probs = probs / np.sum(probs)
        return np.random.choice(len(probs), p=probs)

    def generate(self, temperature):
        generated_images = np.zeros(
            shape=(self.num_img,) + (pixel_cnn.input_shape)[1:])
        batch, rows, cols, channels = generated_images.shape
        for row in range(rows):
            for col in range(cols):
```

```

        for channel in range(channels):
            probs = self.model.predict(generated_images)[
                :, row, col, :
            ]
            generated_images[:, row, col, channel] = [
                self.sample_from(x, temperature) for x in
            probs
            ]
            generated_images[:, row, col, channel] /= 4
    return generated_images

def on_epoch_end(self, epoch, logs=None):
    generated_images = self.generate(temperature = 1.0)
    display(
        generated_images,
        save_to = "./output/generated_img_%03d.png" %
    (epoch)
    s)
img_generator_callback = ImageGenerator(num_img=10)

```

1. Comience con un lote de imágenes vacías (todas ceros).
2. Recorra las filas, columnas y canales de la imagen actual, prediciendo la distribución del siguiente valor de píxel.
3. Muestreo de un nivel de píxel de la distribución prevista (para nuestro ejemplo, un nivel en el rango $[0, 3]$).
4. Convierta el nivel de píxel al rango $[0, 1]$ y sobrescriba el valor de píxel en la imagen actual, lista para la siguiente iteración del bucle.

En la Figura 5-15, podemos ver varias imágenes del conjunto de entrenamiento original, junto con imágenes generadas por PixelCNN.

Figura 5-15. Imágenes de ejemplo del conjunto de entrenamiento e imágenes generadas creadas por el modelo PixelCNN

¡El modelo hace un gran trabajo al recrear la forma general y el estilo de las imágenes originales! Es bastante sorprendente que podamos tratar las imágenes como una serie de tokens (valores de píxeles) y aplicar modelos autorregresivos como PixelCNN para producir muestras realistas.

Como se mencionó anteriormente, una de las desventajas de los modelos autorregresivos es que tardan en tomar muestras, razón por la cual en este libro se presenta un ejemplo simple de su aplicación. Sin embargo, como veremos en el Capítulo 10, se pueden aplicar formas más complejas de modelos autorregresivos a imágenes para producir resultados de última generación. En tales casos, la baja velocidad de generación es un precio necesario a pagar a cambio de productos de calidad excepcional.

Desde que se publicó el artículo original, se han realizado varias mejoras en la arquitectura y el proceso de capacitación de PixelCNN. La siguiente sección presenta uno de esos cambios (usando distribuciones mixtas) y demuestra cómo entrenar un modelo PixelCNN con esta mejora usando una función integrada de TensorFlow.

Distribuciones de mezclas

Para nuestro ejemplo anterior, redujimos la salida de PixelCNN a solo 4 niveles de píxeles para garantizar que la red no tuviera que aprender una distribución de más de 256 valores de píxeles independientes, lo que ralentizaría el proceso de entrenamiento. Sin embargo, esto está lejos de ser ideal: para imágenes en color, no queremos que nuestro lienzo se limite a sólo un puñado de colores posibles.

Para solucionar este problema, podemos hacer que la salida de la red sea una distribución mixta, en lugar de un softmax sobre 256 valores de píxeles discretos, siguiendo las ideas presentadas por Salimans et al. [4] Una distribución mixta es simplemente una mezcla de dos o más distribuciones de probabilidad. Por ejemplo, podríamos tener una distribución mixta de cinco distribuciones logísticas, cada una con parámetros diferentes. La distribución mixta también requiere una distribución categórica discreta que denota la probabilidad de elegir cada una de las distribuciones incluidas en la mezcla. Un ejemplo se muestra en la Figura 5-16.

Figura 5-16. Una distribución mixta de tres distribuciones normales con diferentes parámetros: la distribución categórica entre las tres distribuciones normales es $[0,5, 0,3, 0,2]$

Para tomar muestras de una distribución mixta, primero tomamos muestras de la distribución categórica para elegir una subdistribución particular y luego tomamos muestras de esta de la manera habitual. De esta manera, podemos crear distribuciones complejas con relativamente pocos parámetros. Por ejemplo, la distribución mixta de la Figura 5-16 solo requiere ocho parámetros: dos para la distribución categórica y una media y una varianza para cada una de las tres distribuciones normales. Esto es en comparación con los 255 parámetros que definirían una distribución categórica en todo el rango de píxeles.

Convenientemente, la biblioteca TensorFlow Probability proporciona una función que nos permite crear un PixelCNN con salida de distribución mixta en una sola línea. El ejemplo 5-16 ilustra cómo construir un PixelCNN usando esta función.

EJECUTANDO EL CÓDIGO PARA ESTE EJEMPLO

El código para este ejemplo se puede encontrar en el cuaderno de Jupyter en notebooks/05_autoregressive/03_pixelcnn_md/pixelcnn_md.ipynb en el repositorio de libros.

Ejemplo 5-16. Construyendo un PixelCNN usando la función TensorFlow

```
import tensorflow_probability as tfp

dist = tfp.distributions.PixelCNN(
    image_shape=(32, 32, 1),
    num_resnet=1,
    num_hierarchies=2,
    num_filters=32,
    num_logistic_mix=5,
    dropout_p=.3,
)
image_input = layers.Input(shape=(32, 32, 1))
log_prob = dist.log_prob(image_input)

model = models.Model(inputs=image_input, outputs=log_prob)
model.add_loss(-tf.reduce_mean(log_prob))

1. Defina PixelCNN como una distribución, es decir, la capa de salida es una distribución mixta formada por
```

- cinco distribuciones logísticas.
2. La entrada es una imagen en escala de grises de tamaño $32 \times 32 \times 1$.
 3. El modelo toma una imagen en escala de grises como entrada y genera la probabilidad logarítmica de la imagen bajo la distribución de mezcla calculada por PixelCNN.
 4. La función de pérdida es la probabilidad logarítmica negativa media sobre el lote de imágenes de entrada.

El modelo se entrena de la misma manera que antes, pero esta vez acepta valores de píxeles enteros como entrada, en el rango $[0, 255]$. Se pueden generar resultados a partir de la distribución utilizando la función de muestra, como se muestra en el Ejemplo 5-17.

Ejemplo 5-17. Muestreo de la distribución de mezcla de PixelCNN

```
dist.sample(10).numpy()
```

En la Figura 5-17 se muestran ejemplos de imágenes generadas. La diferencia con nuestros ejemplos anteriores es que ahora se utiliza toda la gama de valores de píxeles.

Figura 5-17. Salidas de PixelCNN utilizando una salida de distribución mixta

Resumen

En este capítulo hemos visto cómo se pueden aplicar modelos autorregresivos, como las redes neuronales recurrentes, para generar secuencias de texto que imitan un estilo particular de escritura, y también cómo un PixelCNN puede generar imágenes de forma secuencial, un píxel a la vez.

Exploramos dos tipos diferentes de capas recurrentes: memoria a corto plazo (LSTM) y unidad recurrente cerrada (GRU), y vimos cómo estas celdas se pueden apilar o hacer bidireccionales para formar arquitecturas de red más complejas. Construimos una LSTM para generar recetas realistas usando Keras y vimos cómo manipular la temperatura del proceso de muestreo para aumentar o disminuir la aleatoriedad de la salida.

También vimos cómo se pueden generar imágenes de forma autorregresiva, utilizando PixelCNN. Construimos un PixelCNN desde cero usando Keras, codificando las capas convolucionales enmascaradas y los bloques residuales para permitir que la información fluya a través de la red, de modo que solo los píxeles anteriores pudieran usarse para generar el píxel actual. Finalmente, discutimos cómo la biblioteca TensorFlow Probability proporciona una función PixelCNN independiente que implementa una distribución mixta como capa de salida, lo que nos permite mejorar aún más el proceso de aprendizaje.

En el próximo capítulo exploraremos otra familia de modelos generativos que modela explícitamente la distribución generadora de datos: los modelos de flujo normalizador.

1 Sepp Hochreiter y Jürgen Schmidhuber, "Memoria a largo plazo", Computación neuronal 9 (1997): 1735-1780, <https://www.bioinf.jku.at/publications/older/2604.pdf>.

2 Kyunghyun Cho et al., "Aprendizaje de representaciones de frases utilizando Codificador-decodificador RNN para traducción automática estadística", 3 de junio de 2014, <https://arxiv.org/abs/1406.1078>.

3 Aaron van den Oord et al., "Pixel Recurrent Neural Networks", agosto 19, 2016, <https://arxiv.org/abs/1601.06759>.

4 Tim Salimans et al., "PixelCNN++: Mejorando PixelCNN con probabilidad de mezcla logística discretizada y otras modificaciones", enero 19, 2017, <http://arxiv.org/abs/1701.05517>.

Capítulo 6. Modelos de flujo normalizadores

METAS DEL CAPÍTULO

En este capítulo podrás:

- Aprenda cómo los modelos de flujo de normalización utilizan la ecuación de cambio de variables.
- Ver cómo el determinante jacobiano juega un papel vital en nuestra capacidad de calcular una función de densidad explícita.
- Comprender cómo podemos restringir la forma del Jacobiano usando capas de acoplamiento.
- Ver cómo la red neuronal está diseñada para ser invertible.
- Crear un modelo RealNVP: un ejemplo particular de un flujo de normalización para generar puntos en 2D.
- Utilizar el modelo RealNVP para generar nuevos puntos que parezcan extraídos de la distribución de datos.
- Conocer dos extensiones clave del modelo RealNVP, GLOW y FFJORD.

Hasta ahora, hemos analizado tres familias de modelos generativos: codificadores automáticos variacionales, redes generativas adversarias y modelos autorregresivos. Cada uno presenta una forma diferente de abordar el desafío de modelar la distribución $p(x)$, ya sea introduciendo una variable latente que se pueda muestrear fácilmente (y transformar usando el decodificador en VAE o el generador en GAN), o modelando de manera manejable la distribución como una función de los valores de los elementos precedentes (modelos autorregresivos).

En este capítulo cubriremos una nueva familia de modelos generativos: los modelos de flujo de normalización. Como ya veremos, los flujos de normalización comparten similitudes tanto con los modelos autorregresivos como con los codificadores automáticos variacionales. Al igual que los

modelos autorregresivos, los flujos normalizadores pueden modelar de manera explícita y manejable la distribución generadora de datos $p(x)$. Al igual que los VAE, los flujos de normalización intentan mapear los datos en una distribución más simple, como una distribución gaussiana. La diferencia clave es que la normalización de los flujos impone una restricción a la forma de la función de mapeo, de modo que es invertible y, por lo tanto, puede usarse para generar nuevos puntos de datos.

Profundizaremos en esta definición en detalle en la primera sección de este capítulo antes de implementar un modelo de flujo de normalización llamado RealNVP usando Keras. También veremos cómo los flujos de normalización se pueden ampliar para crear modelos más potentes, como GLOW y FFJORD.

Introducción

Comenzaremos con una breve historia para ilustrar los conceptos clave detrás de la normalización de los flujos.

JACOB Y LA MÁQUINA F.L.O.W.

Al visitar un pequeño pueblo, notas una tienda de aspecto misterioso con un letrero encima de la puerta que dice DE JACOB. Intrigado, entras con cautela y le preguntas al anciano que está detrás del mostrador qué vende (Figura 6-1).

Figura 6-1. Dentro de una tienda steampunk, con una gran campana metálica (creada con Midjourney)

Él responde que ofrece un servicio de digitalización de cuadros, con una diferencia. Después de un breve momento hurgando en la parte trasera de la tienda, saca una caja plateada con las letras F.L.O.W. Él te dice que esto significa "encontrar la probabilidad de las acuarelas" (FLOW por sus siglas en inglés), que describe aproximadamente lo que hace la máquina. Decides probarla.

Vuelves al día siguiente y le entregas al comerciante un juego de tus cuadros favoritos y él los pasa por la máquina. El flujo. La máquina comienza a tararear y silbar y después de un rato genera un conjunto de números que aparecen generados aleatoriamente. El comerciante te entrega la lista y comienza a caminar hacia la caja para calcular cuánto le debes por el proceso de digitalización y el F.L.O.W. caja. Nada impresionado, le preguntas al comerciante qué debes hacer con esta larga lista de números y cómo puedes recuperar tus cuadros favoritos.

El comerciante pone los ojos en blanco, como si la respuesta fuera obvia. Vuelve a la máquina y pasa la larga lista de números, esta vez desde el lado opuesto. Escuchas el zumbido de la máquina nuevamente y esperas, desconcertado, hasta que finalmente tus pinturas originales desaparecen por donde entraron.

Aliviado de recuperar finalmente tus cuadros, decides que sería mejor guardarlos en el ático. Sin embargo, antes de que tengas la oportunidad de irte, el comerciante te lleva a un rincón diferente de la tienda, donde una campana gigante cuelga de las vigas. Golpea la curva de campana con un palo enorme, enviando vibraciones por toda la tienda.

Al instante, la máquina F.L.O.W. que tienes debajo del brazo comienza a silbar y a zumbar en reversa, como si acabaran de pasar un nuevo conjunto de números. Después de unos momentos, más bellas acuarelas comienzan a caer de la máquina F.L.O.W., pero no son los mismos que los que digitalizó originalmente. Se parecen en estilo y forma a su conjunto de pinturas original, ¡pero cada una es completamente única!

Le preguntas al comerciante cómo funciona este increíble dispositivo. Explica que la magia radica en el hecho de que ha desarrollado un proceso especial que garantiza que la transformación sea extremadamente rápida y sencilla de calcular, sin dejar de ser lo suficientemente sofisticado como para convertir las vibraciones producidas por la campana en los complejos patrones y formas presentes en las pinturas.

Al darte cuenta del potencial de este artilugio, pagas apresuradamente el dispositivo y sales de la tienda, feliz de tener una forma de generar nuevas

pinturas en tu estilo favorito, simplemente visitando la tienda, tocando el timbre y esperando tu mensaje F.L.O.W. máquina para hacer su magia!

La historia de Jacob y la máquina F.L.O.W. es una representación de un modelo de flujo normalizado. Exploraremos ahora la teoría de la normalización de flujos con más detalle, antes de implementar un ejemplo práctico utilizando Keras.

Normalización de flujos

La motivación de normalizar los modelos de flujo es similar a la de los codificadores automáticos variacionales, que exploramos en el capítulo 3. En resumen, en un auto codificador variacional, aprendemos una función de mapeo del codificador entre una distribución compleja y una distribución mucho más simple de la que podemos tomar muestras. Luego también aprendemos una función de mapeo del decodificador desde la distribución más simple a la distribución compleja, de modo que podamos generar un nuevo punto de datos muestreando un punto z de la distribución más simple y aplicando la transformación aprendida. Hablando probabilísticamente, el decodificador modela $p(x|z)$, pero el codificador es solo una aproximación $q(z|x)$ del $p(z|x)$ verdadero: el codificador y el decodificador son dos redes neuronales completamente distintas.

En un modelo de flujo de normalización, la función de decodificación está diseñada para ser exactamente inversa de la función de codificación y rápida de calcular, dando a los flujos de normalización la propiedad de manejabilidad. Sin embargo, las redes neuronales no son por defecto funciones reversibles. Esto plantea la cuestión de cómo podemos crear un proceso invertible que convierta entre una distribución compleja (como la distribución de generación de datos de un conjunto de acuarelas) y una distribución mucho más simple (como una distribución gaussiana en forma de campana) sin dejar de hacer uso de la flexibilidad y el poder del aprendizaje profundo.

Para responder a esta pregunta, primero debemos comprender una técnica conocida como cambio de variables. Para esta sección, trabajaremos con un ejemplo simple en solo dos dimensiones, para que pueda ver exactamente

cómo funcionan los flujos de normalización con gran detalle. Los ejemplos más complejos son sólo extensiones de las técnicas básicas presentadas aquí.

Cambio de variables

Supongamos que tenemos una distribución de probabilidad $p_X(x)$ definida sobre un rectángulo X en dos dimensiones ($x = (x_1, x_2)$), como se muestra en la Figura 6-2.

Figura 6-2. Una distribución de probabilidad $p_X(x)$ definida en dos dimensiones, mostrada en 2D (izquierda) y 3D (derecha)

Esta función se integra a 1 en el dominio de la distribución (es decir, x_1 en el rango $[1, 4]$ y x_2 en el rango $[0, 2]$), por lo que representa una distribución de probabilidad bien definida. Podemos escribir esto de la siguiente manera:

$$\int_0^2 \int_1^4 p_X(x) dx_1 dx_2 = 1$$

Digamos que queremos desplazar y escalar esta distribución para que, en cambio, se defina sobre un cuadrado unitario Z . Podemos lograr esto definiendo una nueva variable $z = (z_1, z_2)$ y una función f que asigna cada punto en X a exactamente un punto en Z de la siguiente manera:

$$z = f(x)$$

$$z_1 = \frac{x_1 - 1}{3}$$

$$z_2 = \frac{x_2}{2}$$

Tenga en cuenta que esta función es *reversible*. Es decir, hay una función g que asigna cada z a su x correspondiente. Esto es esencial para un cambio de variables, ya que de lo contrario no podemos mapear consistentemente

hacia adelante y hacia atrás entre los dos espacios. Podemos encontrar g simplemente reorganizando las ecuaciones que definen f , como se muestra en la Figura 6-3.

Figura 6-3. Cambiar variables entre X y Z

Ahora necesitamos ver cómo el cambio de variables de X a Z afecta la distribución de probabilidad $p_X(x)$. Podemos hacer esto reemplazando las ecuaciones que definen g en $p_X(x)$ para transformarla en una función $p_Z(z)$ que se define en términos de z :

$$p_z(z) = \frac{((3z_1+1)-1)(2z_2)}{9}$$

$$= \frac{2z_1z_2}{3}$$

Sin embargo, si ahora integramos $p_Z(z)$ sobre el cuadrado unitario, ¡podemos ver que tenemos un problema!

$$\int_0^1 \int_0^1 \frac{2z_1z_2}{3} dz_1 dz_2 = \frac{1}{6}$$

La función transformada $p_Z(z)$ ya no es una distribución de probabilidad válida, porque solo se integra a 1/6. Si queremos transformar nuestra distribución de probabilidad compleja sobre los datos en una distribución más simple de la que podamos tomar muestras, debemos asegurarnos de que se integre a 1.

El factor 6 que falta se debe al hecho de que el dominio de nuestra distribución de probabilidad transformada es seis veces más pequeño que el dominio original: el rectángulo original X tenía un área de 6, y esto se ha comprimido en un cuadrado unitario Z que solo tiene un área de 1. Por lo tanto, necesitamos multiplicar la nueva distribución de probabilidad por un factor de normalización que sea igual al cambio relativo en el área (o volumen en dimensiones superiores).

Afortunadamente, existe una manera de calcular este cambio de volumen para una transformación determinada: es el valor absoluto del determinante jacobiano de la transformación. ¡Descomprimamos eso!

El determinante jacobiano

El jacobiano de una función $z = f(x)$ es la matriz de sus derivadas parciales de primer orden, como se muestra aquí:

$$J = \frac{\partial z}{\partial x} = \begin{bmatrix} \frac{\partial z_1}{\partial x_1} & \cdots & \frac{\partial z_1}{\partial x_n} \\ \cdots & \cdots & \cdots \\ \frac{\partial z_m}{\partial x_1} & \cdots & \frac{\partial z_m}{\partial x_n} \end{bmatrix}$$

La mejor manera de explicar esto es con nuestro ejemplo. Si tomamos la derivada parcial de z_1 con respecto a x_1 , obtenemos $1/3$. Si tomamos la derivada parcial de z_1 con respecto a x_2 , obtenemos 0 . De manera similar, si tomamos la derivada parcial de z_2 con respecto a x_1 , obtenemos 0 . Por último, si tomamos la derivada parcial de z_2 con respecto a x_2 , obtenemos $1/2$.

Por tanto, la matriz jacobiana para nuestra función $f(x)$ queda como sigue:

$$J = \begin{pmatrix} \frac{1}{3} & 0 \\ 0 & \frac{1}{2} \end{pmatrix}$$

El *determinante* solo se define para matrices cuadradas y es igual al volumen con signo del paralelepípedo creado al aplicar la transformación representada por la matriz al (hiper)cubo unitario. Por lo tanto, en dos dimensiones, esta es solo el área con signo del paralelogramo creado al aplicar la transformación representada por la matriz al cuadrado unitario.

Existe una fórmula general para calcular el determinante de una matriz con n dimensiones, que se ejecuta en tiempo (n^3) .

Para nuestro ejemplo, sólo necesitamos la fórmula para dos dimensiones, que es simplemente la siguiente:

$$\det \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

Por tanto, para nuestro ejemplo, el determinante del jacobiano es $\frac{1}{3} \times \frac{1}{2} - 0 \times 0 = \frac{1}{6}$. ¡Este es el factor de escala de 1/6 que necesitamos para asegurarnos de que la distribución de probabilidad después de la transformación aún se integre a 1!

CONSEJO

Por definición, el determinante tiene signo, es decir, puede ser negativo. Por tanto, debemos tomar el valor absoluto del determinante jacobiano para obtener el cambio relativo de volumen.

La ecuación de cambio de variables

Ahora podemos escribir una única ecuación que describa el proceso para cambiar variables entre X y Z. Esto se conoce como ecuación de cambio de variables (Ecuación 61).

Ecuación 6-1. La ecuación de cambio de variables.

$$p_X(x) = p_Z(z) = \left| \det \frac{\partial z}{\partial x} \right|$$

¿Cómo nos ayuda esto a construir un modelo generativo? La clave es comprender que si $p_Z(z)$ es una distribución simple de la cual podemos tomar muestras fácilmente (por ejemplo, una gaussiana), entonces, en teoría, todo lo que necesitamos hacer es encontrar una función invertible apropiada $f(x)$ que pueda mapear de los datos X a Z y la función inversa correspondiente $g(z)$ que se puede utilizar para asignar un z muestreado a un punto x en el dominio original. Podemos usar la ecuación anterior que involucra al determinante jacobiano para encontrar una fórmula exacta y manejable para la distribución de datos $p(x)$.

Sin embargo, hay dos problemas importantes al aplicar esto en la práctica que primero debemos abordar.

En primer lugar, calcular el determinante de una matriz de alta dimensión es computacionalmente extremadamente costoso; específicamente, es (n^3) . Esto es completamente impráctico de implementar, ya que incluso las imágenes pequeñas en escala de grises de 32×32 píxeles tienen 1,024 dimensiones.

En segundo lugar, no es inmediatamente obvio cómo debemos proceder para calcular la función invertible $f(x)$. Podríamos usar una red neuronal para encontrar alguna función $f(x)$, pero no necesariamente podemos invertir esta red: ¡las redes neuronales solo funcionan en una dirección!

Para resolver estos dos problemas, necesitamos utilizar una arquitectura de red neuronal especial que garantice que la función de cambio de variables f sea invertible y tenga un determinante fácil de calcular.

Veremos cómo hacer esto en la siguiente sección utilizando una técnica llamada *transformaciones de valor real sin preservación de volumen* (RealNVP, por sus siglas en inglés).

RealNVP

RealNVP fue introducido por primera vez por Dinh et al. en 2017.¹ En este artículo, los autores muestran cómo construir una red neuronal que pueda transformar una distribución de datos compleja en una gaussiana simple, al tiempo que posee las propiedades deseadas de ser invertible y tener un jacobiano que se puede calcular fácilmente.

EJECUTANDO EL CÓDIGO PARA ESTE EJEMPLO

El código de este ejemplo se puede encontrar en el cuaderno de Jupyter ubicado en notebooks/06_normflow/01_realmvp/realmvp.ipynb en el repositorio de libros.

El código ha sido adaptado del excelente tutorial RealNVP creado por Mandolini Giorgio Maria et al. disponible en el sitio web de Keras.

El conjunto de datos de las dos lunas

El conjunto de datos que usaremos para este ejemplo es creado por la función `make_moons` de la biblioteca `sklearn` de Python. Esto crea un conjunto de datos ruidoso de puntos en 2D que se asemejan a dos medias lunas, como se muestra en la Figura 6-4.

Figura 6-4. El conjunto de datos de las dos lunas en dos dimensiones.

El código para crear este conjunto de datos se proporciona en el Ejemplo 6-1.

Ejemplo 6-1. Creando un conjunto de datos de lunas

```
data = datasets.make_moons(3000, noise=0.05)
[0].astype("float32")
norm = layers.Normalization()
norm.adapt(data)
normalized_data = norm(data)
```

- Cree un conjunto de datos de lunas ruidosas y no normalizadas de 3000 puntos.
- Normalice el conjunto de datos para que tenga media 0 y desviación estándar 1.

Construiremos un modelo RealNVP que pueda generar puntos en 2D que siguen una distribución similar al conjunto de datos de las dos lunas. Si bien este es un ejemplo muy simple, nos ayudará a comprender con gran detalle cómo funciona en la práctica un modelo de flujo normalizado.

Sin embargo, primero debemos introducir un nuevo tipo de capa, llamada capa de acoplamiento.

Capas de acoplamiento

Una *capa de acoplamiento* produce una escala y un factor de traducción para cada elemento de su entrada. En otras palabras, produce dos tensores que son exactamente del mismo tamaño que la entrada, uno para el factor de escala y otro para el factor de traducción, como se muestra en la Figura 6-5.

Figura 6-5. Una capa de acoplamiento genera dos tensores que tienen la misma forma que la entrada: un factor de escala (s) y un factor de traducción (t).

Para crear una capa de acoplamiento personalizada para nuestro ejemplo simple, podemos apilar capas densas para crear la salida de escala y un conjunto diferente de capas densas para crear la salida de traducción, como se muestra en el Ejemplo 6-2.

CONSEJO

Para imágenes, los bloques de capas de acoplamiento utilizan capas Conv2D en lugar de capas densas.

Ejemplo 6-2. Una capa de acoplamiento en Keras

```
def Coupling():
    input_layer = layers.Input(shape=2)

    s_layer_1 = layers.Dense(
        256, activation="relu",
        kernel_regularizer=regularizers.l2(0.01)
    )(input_layer)
    s_layer_2 = layers.Dense(
        256, activation="relu",
        kernel_regularizer=regularizers.l2(0.01)
    )(s_layer_1)
    s_layer_3 = layers.Dense(
        256, activation="relu",
        kernel_regularizer=regularizers.l2(0.01)
    )(s_layer_2)
    s_layer_4 = layers.Dense(
        256, activation="relu",
        kernel_regularizer=regularizers.l2(0.01)
```

```

        )(s_layer_3)
    s_layer_5 = layers.Dense(
        2, activation="tanh",
    kernel_regularizer=regularizers.l2(0.01)
    )(s_layer_4)

    t_layer_1 = layers.Dense(
        256, activation="relu",
    kernel_regularizer=regularizers.l2(0.01)
    )(input_layer)
    t_layer_2 = layers.Dense(
        256, activation="relu",
    kernel_regularizer=regularizers.l2(0.01)
    )(t_layer_1)
    t_layer_3 = layers.Dense(
        256, activation="relu",
    kernel_regularizer=regularizers.l2(0.01)
    )(t_layer_2)
    t_layer_4 = layers.Dense(
        256, activation="relu",
    kernel_regularizer=regularizers.l2(0.01)
    )(t_layer_3)
    t_layer_5 = layers.Dense(
        2, activation="linear",
    kernel_regularizer=regularizers.l2(0.01)
    )(t_layer_4)

    return models.Model(inputs=input_layer, outputs=[s_layer_5,
t_layer_5])

```

1. La entrada al bloque de capa de acoplamiento en nuestro ejemplo tiene dos dimensiones.
2. El flujo de escalado es una pila de capas densas de tamaño 256.
3. La capa de incrustación final es de tamaño 2 y tiene activación tanh.
4. El flujo de traducción es una pila de capas densas de tamaño. 256.
5. La capa de traducción final es de tamaño 2 y tiene activación lineal.
6. La capa de acoplamiento se construye como un modelo Keras con dos salidas (los factores de escala y traducción).

Observe cómo la cantidad de canales aumenta temporalmente para permitir que se aprenda una representación más compleja, antes de volver a reducirla a la misma cantidad de canales que la entrada. En el artículo original, los autores también utilizan regularizadores en cada capa para penalizar los pesos grandes.

Pasando datos a través de una capa de acoplamiento

La arquitectura de una capa de acoplamiento no es particularmente interesante; lo que la hace única es la forma en que los datos de entrada se enmascaran y transforman a medida que pasan a través de la capa, como se muestra en la Figura 6-6.

Figura 6-6. El proceso de transformar la entrada x a través de una capa de acoplamiento.

Observe cómo solo las primeras d dimensiones de los datos pasan a la primera capa de acoplamiento; las dimensiones $D - d$ restantes están completamente enmascaradas (es decir, establecidas en cero). En nuestro ejemplo simple con $D = 2$, elegir $d = 1$ significa que en lugar de que la capa de acoplamiento vea dos valores, (x_1, x_2) , la capa ve $(x_1, 0)$.

Los resultados de la capa son los factores de escala y traducción. Éstas se enmascaran nuevamente, pero esta vez con la máscara inversa a la anterior, de modo que sólo se dejan pasar las segundas mitades; es decir, en nuestro ejemplo, obtenemos $(0, s_2)$ y $(0, t_2)$. Luego, estos se aplican elemento por elemento a la segunda mitad de la entrada x_2 y la primera mitad de la entrada x_1 simplemente se pasa directamente, sin actualizarse en absoluto. En resumen, para un vector con dimensión D donde $d < D$, las ecuaciones de actualización son las siguientes:

$$\begin{aligned} z_{1:d} &= x_{1:d} \\ z_{d+1:D} &= x_{d+1:D} \quad \exp(s(x_{1:d})) + t(x_{1:d}) \end{aligned}$$

Quizás se pregunte por qué nos tomamos la molestia de crear una capa que enmascare tanta información. La respuesta es clara si investigamos la estructura de la matriz jacobiana de esta función:

$$\frac{\partial z}{\partial x} = \begin{bmatrix} I & 0 \\ \frac{\partial z_{d+1:D}}{\partial x_{1:d}} & \text{diag}(\exp[s(x_{1:d})]) \end{bmatrix}$$

La submatriz $d \times d$ superior izquierda es simplemente la matriz identidad, porque $z_{1:d} = x_{1:d}$. Estos elementos se pasan directamente sin actualizarse. Por lo tanto, la submatriz superior derecha es 0, porque $z_{1:d}$ no depende de $x_{d+1:D}$.

La submatriz inferior izquierda es compleja y no pretendemos simplificarla. La submatriz inferior derecha es simplemente una matriz diagonal, llena con los elementos de $\exp(s(x_{1:d}))$, porque $z_{d+1:D}$ depende linealmente de $x_{d+1:D}$ y el gradiente depende sólo en el factor de escala (no en el factor de traducción). La Figura 6-7 muestra un diagrama de esta forma matricial, donde sólo los elementos distintos de cero se rellenan con color.

Observe que no hay elementos distintos de cero encima de la diagonal; por esta razón, esta forma matricial se llama *triangular inferior*. Ahora vemos el beneficio de estructurar la matriz de esta manera: el determinante de una matriz triangular inferior es exactamente igual al producto de los elementos diagonales. En otras palabras, ¡el determinante no depende de ninguna de las derivadas complejas de la submatriz inferior izquierda!

Figura 6-7. La matriz jacobiana de la transformación: una matriz triangular inferior, con determinante igual al producto de los elementos a lo largo de la diagonal.

Por tanto, podemos escribir el determinante de esta matriz de la siguiente manera:

$$\det(J) = \exp \left[\sum_j s(x_{1:d})_j \right]$$

Esto es fácilmente computable, lo cual era uno de los dos objetivos originales de construir un modelo de flujo normalizado.

El otro objetivo era que la función fuera fácilmente invertible. Podemos ver que esto es cierto ya que podemos escribir la función invertible simplemente reorganizando las ecuaciones directas, de la siguiente manera:

$$x_{1:d} = z_{1:d}$$

$$x_{d+1:D} = (z_{d+1:D} - t(x_{1:d})) \quad \exp(-s(x_{1:d}))$$

El diagrama equivalente se muestra en la Figura 6-8.

Figura 6-8. La función inversa $x = g(z)$

Ahora tenemos casi todo lo que necesitamos para construir nuestro modelo RealNVP. Sin embargo, aún queda un problema: ¿cómo deberíamos actualizar los primeros d elementos de la entrada? ¡Actualmente el modelo los deja completamente sin cambios!

Apilamiento de capas de acoplamiento

Para solucionar este problema, podemos utilizar un truco realmente sencillo. Si apilamos capas de acoplamiento una encima de otra pero alternamos el patrón de enmascaramiento, las capas que una capa deja sin cambios se actualizarán en la siguiente. Esta arquitectura tiene el beneficio adicional de poder aprender representaciones más complejas de los datos, ya que es una red neuronal más profunda.

El jacobiano de esta composición de capas de acoplamiento seguirá siendo sencillo de calcular, porque el álgebra lineal nos dice que el determinante de un producto matricial es el producto de los determinantes. De manera similar, la inversa de la composición de dos funciones es simplemente la composición de las inversas, como se muestra en las siguientes ecuaciones:

$$\det(A \cdot B) = \det(A)\det(B)$$

$$(f_b \circ f_a)^{-1} = f_a^{-1} \circ f_b^{-1}$$

Por lo tanto, si apilamos capas de acoplamiento, invirtiendo la máscara cada vez, podemos construir una red neuronal que sea capaz de transformar todo el tensor de entrada, conservando al mismo tiempo las propiedades

esenciales de tener un determinante jacobiano simple y ser invertible. La Figura 6-9 muestra la estructura general.

Figura 6-9. Apilando capas de acoplamiento, alternando el enmascaramiento con cada capa.

Entrenando el modelo RealNVP

Ahora que hemos creado el modelo RealNVP, podemos entrenarlo para conocer la distribución compleja del conjunto de datos de las dos lunas. Recuerde, queremos minimizar la probabilidad logarítmica negativa de los datos según el modelo $-\log p_X(x)$. Usando la Ecuación 61, podemos escribir esto de la siguiente manera:

$$-\log p_x(x) = -\log p_z(z) - \log \left| \det \frac{\partial z}{\partial x} \right|$$

Elegimos que la distribución de salida objetivo $p_Z(z)$ del proceso directo f sea una gaussiana estándar, porque podemos tomar muestras fácilmente de esta distribución. Luego podemos transformar un punto muestreado del Gaussiano nuevamente al dominio de la imagen original aplicando el proceso inverso g , como se muestra en la Figura 6-10.

Figura 6-10. Transformación entre la distribución compleja $p_X(x)$ y una simple gaussiana $p_Z(z)$ en 1D (fila del medio) y 2D (fila inferior)

El ejemplo 6-3 muestra cómo construir una red RealNVP, como un modelo Keras personalizado.

Ejemplo 6-3. Construyendo el modelo RealNVP en Keras

```
class RealNVP(models.Model):
    def __init__(self, input_dim, coupling_layers, coupling_dim,
                 regularization):
        super(RealNVP, self).__init__()
```

```

        self.coupling_layers = coupling_layers
        self.distribution =
    tfp.distributions.MultivariateNormalDiag(
        loc=[0.0, 0.0], scale_diag=[1.0, 1.0]
    )
    self.masks = np.array(
        [[0, 1], [1, 0]] * (coupling_layers // 2),
    dtype="float32"
    )
    self.loss_tracker = metrics.Mean(name="loss")
    self.layers_list = [
        Coupling(input_dim, coupling_dim, regularization)
        for i in range(coupling_layers)
    ]

@property
def metrics(self):
    return [self.loss_tracker]

def call(self, x, training=True):
    log_det_inv = 0
    direction = 1
    if training:
        direction = -1
    for i in range(self.coupling_layers)[::direction]:
        x_masked = x * self.masks[i]
        reversed_mask = 1 - self.masks[i]
        s, t = self.layers_list[i](x_masked)
        s *= reversed_mask
        t *= reversed_mask
        gate = (direction - 1) / 2
        x = (
            reversed_mask
            * (x * tf.exp(direction * s) + direction * t *
tf.exp(gate * s))
            + x_masked
        )
        log_det_inv += gate * tf.reduce_sum(s, axis = 1)
    return x, log_det_inv

def log_loss(self, x):
    y, logdet = self(x)
    log_likelihood = self.distribution.log_prob(y) + logdet
    return -tf.reduce_mean(log_likelihood)

def train_step(self, data):
    with tf.GradientTape() as tape:
        loss = self.log_loss(data)

```

```

        g = tape.gradient(loss, self.trainable_variables)
        self.optimizer.apply_gradients(zip(g,
self.trainable_variables))
        self.loss_tracker.update_state(loss)
    return {"loss": self.loss_tracker.result()}

def test_step(self, data):
    loss = self.log_loss(data)
    self.loss_tracker.update_state(loss)
    return {"loss": self.loss_tracker.result()}

model = RealNVP(
    input_dim = 2
    , coupling_layers= 6
    , coupling_dim = 256
    , regularization = 0.01
)
model.compile(optimizer=optimizers.Adam(learning_rate=0.0001))

model.fit(
    normalized_data
    , batch_size=256
    , epochs=300
)

```

1. La distribución objetivo es una gaussiana 2D estándar.
2. Aquí creamos el patrón de máscara alterna.
3. Una lista de capas de acoplamiento que definen la red RealNVP.
4. En la función de llamada principal de la red, recorremos las capas de acoplamiento. Si entrenamiento = Verdadero, entonces avanzamos a través de las capas (es decir, de los datos al espacio latente). Si entrenamiento = Falso, entonces retrocedemos a través de las capas (es decir, del espacio latente a los datos).
5. Esta línea describe las ecuaciones hacia adelante y hacia atrás dependiendo de la dirección (intente conectar dirección = -1 y dirección = 1 para demostrarlo a ti mismo).
6. El determinante logarítmico del jacobiano, que necesitamos para calcular la función de pérdida, es simplemente la suma de los factores de escala.
7. La función de pérdida es la suma negativa de la probabilidad logarítmica de los datos transformados, según nuestra distribución gaussiana destino y el determinante logarítmico del Jacobiano.

Análisis del modelo RealNVP

Una vez que el modelo está entrenado, podemos usarlo para transformar el conjunto de entrenamiento en el espacio latente (usando la dirección hacia adelante, f) y, lo que es más importante, para transformar un punto muestreado en el espacio latente en un punto que parece que podría haberlo hecho. han sido muestreados de la distribución de datos original (usando la dirección hacia atrás, g).

La Figura 6-11 muestra la salida de la red antes de que se haya producido cualquier aprendizaje: las direcciones hacia adelante y hacia atrás simplemente pasan información directamente sin apenas ninguna transformación.

Figura 6-11. Las entradas (izquierda) y las salidas (derecha) del modelo RealNVP antes del entrenamiento, para el proceso directo (arriba) y el proceso inverso (abajo).

Después del entrenamiento (Figura 6-12), el proceso directo puede convertir los puntos del conjunto de entrenamiento en una distribución que se asemeja a una gaussiana. Asimismo, el proceso hacia atrás puede tomar puntos muestreados de una distribución gaussiana y mapearlos nuevamente a una distribución que se asemeje a los datos originales.

Figura 6-12. Las entradas (izquierda) y las salidas (derecha) del modelo RealNVP después del entrenamiento, para el proceso directo (arriba) y el proceso inverso (abajo)

La curva de pérdidas para el proceso de entrenamiento se muestra en la figura 6-13.

Figura 6-13. La curva de pérdidas del proceso de formación de RealNVP

Esto completa nuestra discusión sobre RealNVP, un caso específico de un modelo generativo de flujo normalizado. En la siguiente sección, cubriremos algunos modelos modernos de normalización de flujo que amplían las ideas presentadas en el artículo de RealNVP.

Otros modelos de flujo de normalización

Otros dos modelos de normalización de flujo exitosos e importantes son GLOW y FFJORD. Las siguientes secciones describen los avances clave que lograron.

GLOW

Presentado en NeurIPS 2018, GLOW (Brillo en inglés) fue uno de los primeros modelos en demostrar la capacidad de normalizar flujos para generar muestras de alta calidad y producir un espacio latente significativo que puede atravesarse para manipular muestras. El paso clave fue reemplazar la configuración de enmascaramiento inverso con capas convolucionales invertibles 1×1 . Por ejemplo, con RealNVP aplicado a imágenes, el orden de los canales se invierte después de cada paso, para garantizar que la red tenga la oportunidad de transformar toda la entrada. En cambio, en GLOW se aplica una convolución 1×1 , que actúa efectivamente como un método general para producir cualquier permutación de los canales que desee el modelo. Los autores muestran que incluso con esta adición, la distribución en su conjunto sigue siendo manejable, con determinantes e inversas que son fáciles de calcular a escala.

Figura 6-14. Muestras aleatorias del modelo GLOW (fuente: Kingma y Dhariwal, 2018)²

FFJORD

RealNVP y GLOW son flujos de normalización de tiempo discreto, es decir, transforman la entrada a través de un conjunto discreto de capas de

acoplamiento. FFJORD (Dinámica continua de forma libre para modelos generativos reversibles escalables), presentado en ICLR 2019 muestra cómo es posible modelar la transformación como un proceso de tiempo continuo (es decir, tomando el límite cuando el número de pasos en el flujo tiende a infinito y el tamaño del paso tiende a cero). En este caso, las dinámicas se modelan utilizando una ecuación diferencial ordinaria (ODE) cuyos parámetros son producidos por una red neuronal (f_θ).

Se utiliza un solucionador de caja negra para resolver la EDO en el tiempo t_1 , es decir, para encontrar z_1 dado algún punto inicial z_0 muestreado de una gaussiana en t_0 , como lo describen las siguientes ecuaciones:

$$z_0 \sim p(z_0)$$

$$\frac{\partial z(t)}{\partial t} = f_0(x(t), t)$$

$$x = z_1$$

Un diagrama del proceso de transformación se muestra en la figura 6-15.

Figura 6-15. FFJORD modela la transformación entre la distribución de datos y una gaussiana estándar mediante una ecuación diferencial ordinaria, parametrizada por una red neuronal (fuente: Will Grathwohl et al., 2018) [3]

Resumen

En este capítulo exploramos modelos de flujo de normalización como RealNVP, GLOW y FFJORD.

Un modelo de flujo normalizador es una función invertible definida por una red neuronal que nos permite modelar directamente la densidad de datos mediante un cambio de variables. En el caso general, la ecuación de cambio de variables requiere que calculemos un determinante jacobiano altamente

complejo, lo cual no es práctico para todos los ejemplos excepto para los más simples.

Para evitar este problema, el modelo RealNVP restringe la forma de la red neuronal, de modo que cumpla con dos criterios esenciales: es invertible y tiene un determinante jacobiano que es fácil de calcular.

Lo hace apilando capas de acoplamiento, que producen factores de escala y traducción en cada paso. Es importante destacar que la capa de acoplamiento enmascara los datos a medida que fluyen a través de la red, de una manera que garantiza que el jacobiano es triangular inferior y, por tanto, tiene un determinante fácil de calcular. La visibilidad total de los datos de entrada se logra volteando las máscaras en cada capa.

Por diseño, las operaciones de escala y traducción se pueden invertir fácilmente, de modo que una vez que se entrena el modelo, es posible ejecutar datos a través de la red a la inversa. Esto significa que podemos orientar el proceso de transformación directa hacia un gaussiano estándar, del que podemos tomar muestras fácilmente. Luego podemos ejecutar los puntos muestreados hacia atrás a través de la red para generar nuevas observaciones.

El artículo de RealNVP también muestra cómo es posible aplicar esta técnica a imágenes mediante el uso de convoluciones dentro de las capas de acoplamiento, en lugar de capas densamente conectadas. El artículo GLOW amplió esta idea para eliminar la necesidad de cualquier permutación codificada de las máscaras. El modelo FFJORD introdujo el concepto de tiempo continuo normalizando flujos, modelando el proceso de transformación como una EDO definida por una red neuronal.

En general, hemos visto cómo los flujos de normalización son una poderosa familia de modelos generativos que pueden producir muestras de alta calidad, manteniendo al mismo tiempo la capacidad de describir de manera manejable la función de densidad de datos.

1. Laurent Dinh et al., “Estimación de densidad usando NVP real”, 27 de mayo de 2016, <https://arxiv.org/abs/1605.08803v3>.

2. Diedrick P. Kingma y Prafulla Dhariwal, “Glow: Flujo generativo con convoluciones invertibles 1x1”, 10 de julio de 2018, <https://arxiv.org/abs/1807.03039>.
3. Will Grathwohl et al., “FFJORD: Dinámica continua de forma libre para modelos generativos reversibles escalables”, 22 de octubre de 2018, <https://arxiv.org/abs/1810.01367>.

Capítulo 7. Modelos basados en energía

METAS DEL CAPÍTULO

En este capítulo podrás:

- Comprender cómo formular un modelo basado en energía profunda (EBM).
- Ver cómo tomar muestras de un EBM utilizando la dinámica de Langevin.
- Entrenar su propia MBE utilizando divergencia contrastiva.
- Analizar el EBM, incluida la visualización de instantáneas del proceso de muestreo dinámico de Langevin.
- Conocer otros tipos de MBE, como las máquinas Boltzmann restringidas.

Los modelos basados en energía son una clase amplia de modelo generativo que toma prestada una idea clave del modelado de sistemas físicos, es decir, que la probabilidad de un evento se puede expresar usando una distribución de Boltzmann, una función específica que normaliza una función de energía de valor real entre 0 y 0. y 1. Esta distribución fue formulada originalmente en 1868 por Ludwig Boltzmann, quien lo utilizó para describir gases en equilibrio térmico.

En este capítulo, veremos cómo podemos usar esta idea para entrenar un modelo generativo que pueda usarse para producir imágenes de dígitos escritos a mano. Exploraremos varios conceptos nuevos, incluida la divergencia contrastiva para entrenar el EBM y dinámica de Langevin para muestreo.

Introducción

Comenzaremos con una breve historia para ilustrar los conceptos clave detrás de los modelos basados en energía.

EL CLUB DE RUNNING DE LARGO AU-VIN

Diane Mixx era entrenadora en jefe del equipo de carreras de larga distancia en la ficticia ciudad francesa de Long-au-Vin. Era bien conocida por sus excepcionales habilidades como entrenadora y se había ganado la reputación de ser capaz de convertir incluso a los atletas más mediocres en corredores de clase mundial (Figura 7-1).

Figura 7-1. Un entrenador de carreras entrenando a algunos deportistas de élite (creado con Midjourney)

Sus métodos se basaban en evaluar los niveles de energía de cada atleta. A lo largo de años de trabajar con atletas de todos los niveles, había desarrollado una idea increíblemente precisa de cuánta energía le quedaba a un atleta en particular después de una carrera, con solo mirarlo. Cuanto menor sea el nivel de energía de un atleta, mejor: ¡los atletas de élite siempre dieron todo lo que tenían durante la carrera!

Para mantener sus habilidades en forma, se entrenaba regularmente midiendo el contraste entre sus habilidades de detección de energía en atletas de élite conocidos y los mejores atletas de su club. Aseguró que la divergencia entre sus predicciones para estos dos grupos eran lo más amplias posible, por lo que la gente la tomaría en serio si decía que había encontrado un verdadero atleta de élite dentro de su club.

La verdadera magia fue su capacidad para convertir a una corredora mediocre en una corredora de primer nivel. El proceso fue simple: midió el nivel de energía actual del atleta y calculó el conjunto óptimo de ajustes que el atleta necesitaba hacer para mejorar su rendimiento la próxima vez. Luego, después de hacer estos ajustes, midió nuevamente el nivel de energía del atleta, buscando que fuera ligeramente más bajo que antes, lo que explica el mejor desempeño en la pista. Este proceso de evaluar los ajustes óptimos y dar un pequeño paso en la dirección correcta continuaría

hasta que finalmente el atleta fuera indistinguible de un corredor de clase mundial.

Después de muchos años, Diane se retiró del entrenamiento y publicó un libro sobre sus métodos para formar atletas de élite, un sistema que denominó la “Técnica Long-au-Vin” de Diane Mixx.

La historia de Diane Mixx y el club de corredores de Long-au-Vin captura las ideas clave detrás del modelado basado en energía. Exploremos ahora la teoría con más detalle, antes de implementar un ejemplo práctico usando Keras.

Modelos basados en energía

Los modelos basados en energía intentan modelar la verdadera distribución generadora de datos utilizando una *distribución de Boltzmann* (Ecuación 7-1) donde $E(x)$ se conoce como la *función de energía* (o puntuación) de una observación x .

Ecuación 7-1. Distribución de Boltzmann $-E(x)$

$$p(x) = \frac{e^{-E(x)}}{\int_{\hat{x} \in X} e^{-E(\hat{x})}}$$

En la práctica, esto equivale a entrenar una red neuronal $E(x)$ para generar puntuaciones bajas para observaciones probables (por lo que p_x está cerca de 1) y puntuaciones altas para observaciones poco probables (por lo que p_x está cerca de 0).

Hay dos desafíos al modelar los datos de esta manera. En primer lugar, no está claro cómo debemos usar nuestro modelo para muestrear nuevas observaciones; podemos usarlo para generar una puntuación dada una observación, pero ¿cómo generamos una observación que tiene una puntuación baja (es decir, una observación plausible)?

En segundo lugar, el denominador normalizador de la ecuación 7-1 contiene una integral que es intratable para todos los problemas excepto para los más

simples. Si no podemos calcular esta integral, entonces no podemos usar la estimación de máxima verosimilitud para entrenar el modelo, ya que esto requiere que p_x sea una distribución de probabilidad válida.

La idea clave detrás de un modelo basado en energía es que podemos usar técnicas de aproximación para asegurarnos de que nunca necesitemos calcular el denominador intratable. Esto contrasta, por ejemplo, con un flujo de normalización, en el que hacemos todo lo posible para garantizar que las transformaciones que aplicamos a nuestra distribución gaussiana estándar no cambien el hecho de que la salida sigue siendo una distribución de probabilidad válida.

Evitamos el complicado problema del denominador intratable utilizando una técnica llamada divergencia contrastiva (para entrenamiento) y una técnica llamada dinámica de Langevin (para muestreo), siguiendo las ideas de Du y Mordatch en su artículo de 2019 “Generación implícita y modelado con modelos basados en energía”.[1] Exploraremos estas técnicas en detalle mientras construimos nuestra propia MBE más adelante en este capítulo.

Primero, configuremos un conjunto de datos y diseñemos una red neuronal simple que represente nuestra función de energía de valor real $E(x)$.

EJECUTANDO EL CÓDIGO PARA ESTE EJEMPLO

El código de este ejemplo se puede encontrar en el cuaderno de Jupyter ubicado en `notebooks/07_ebm/01_ebm/ebm.ipynb` en el repositorio de libros.

El código está adaptado del excelente tutorial sobre modelos generativos basados en energía profunda de Phillip Lippe.

El conjunto de datos MNIST

Usaremos el conjunto de datos estándar MNIST, que consta de imágenes en escala de grises de dígitos escritos a mano. En la Figura 7-2 se muestran algunas imágenes de ejemplo del conjunto de datos.

Figura 7-2. Ejemplos de imágenes del conjunto de datos MNIST

El conjunto de datos viene empaquetado con TensorFlow, por lo que se puede descargar como se muestra en el Ejemplo 7-1.

Ejemplo 7-1. Cargando el conjunto de datos MNIST

```
from tensorflow.keras import datasets  
(x_train, _), (x_test, _) = datasets.mnist.load_data()
```

Como de costumbre, escalaremos los valores de píxeles al rango [-1, 1] y agregaremos algo de relleno para que las imágenes tengan un tamaño de 32 × 32 píxeles. También lo convertimos a un conjunto de datos de TensorFlow, como se muestra en el Ejemplo 7-2.

Ejemplo 7-2. Preprocesamiento del conjunto de datos MNIST

```
def preprocess(imgs):  
    imgs = (imgs.astype("float32") - 127.5) / 127.5  
    imgs = np.pad(imgs, ((0,0), (2,2), (2,2)), constant_values=-1.0)  
    imgs = np.expand_dims(imgs, -1)  
    return imgs  
  
x_train = preprocess(x_train)  
x_test = preprocess(x_test)  
x_train =  
tf.data.Dataset.from_tensor_slices(x_train).batch(128)  
x_test = tf.data.Dataset.from_tensor_slices(x_test).batch(128)
```

Ahora que tenemos nuestro conjunto de datos, podemos construir la red neuronal que representará nuestra función de energía $E(x)$.

La función energética

La función de energía $E_\theta(x)$ es una red neuronal con parámetros θ que puede transformar una imagen de entrada x en un valor escalar. En toda esta red, utilizamos una función de activación llamada *swish*, como se describe en la siguiente barra lateral.

ACTIVACIÓN DE CHASQUIDO (SWISH)

Chasquido (Swish) es una alternativa a ReLU que fue introducida por Google en 2017[2] y se define de la siguiente manera:

$$swish(x) = x \times \text{sigmoide}(x) = \frac{x}{e^{-x} + 1}$$

Swish es visualmente similar a ReLU, con la diferencia clave de que es suave, lo que ayuda a aliviar el problema del gradiente que desaparece. Esto es particularmente importante para los modelos basados en energía. En la Figura 7-3 se muestra un gráfico de la función swish.

Figura 7-3. La función de activación por chasquido (swish).

La red es un conjunto de capas Conv2D apiladas que reducen gradualmente el tamaño de la imagen mientras aumentan la cantidad de canales. La capa final es una única unidad completamente conectada con activación lineal, por lo que la red puede generar valores en el rango $(-\infty, \infty)$. El código para construirlo se proporciona en el Ejemplo 7-3.

Ejemplo 7-3. Construyendo la red neuronal de la función de energía E(x)

```
ebm_input = layers.Input(shape=(32, 32, 1))
x = layers.Conv2D(16, kernel_size=5, strides=2, padding="same",
activation =
activations.swish
)(ebm_input)
x = layers.Conv2D(
    32, kernel_size=3, strides=2, padding="same", activation =
activations.swish
)(x)
x = layers.Conv2D(
    64, kernel_size=3, strides=2, padding="same", activation =
activations.swish
)(x)
x = layers.Conv2D(
    64, kernel_size=3, strides=2, padding="same", activation =
activations.swish
)(x)
```

```
x = layers.Flatten()(x)
x = layers.Dense(64, activation = activations.swish)(x)
ebm_output = layers.Dense(1)(x)
model = models.Model(ebm_input, ebm_output)
```

1. La función de energía es un conjunto de capas Conv2D apiladas, con activación swish.
2. La capa final es una única unidad completamente conectada, con una función de activación lineal.
3. Un modelo de Keras que convierte la imagen de entrada en un valor de energía escalar.

Muestreo utilizando la dinámica de Langevin

La función de energía solo genera una puntuación para una entrada determinada. ¿Cómo podemos usar esta función para generar nuevas muestras que tengan una puntuación de energía baja?

Usaremos una técnica llamada *dinámica de Langevin*, que aprovecha el hecho de que podemos calcular el gradiente de la función de energía con respecto a su entrada. Si partimos de un punto aleatorio en el espacio muestral y damos pequeños pasos en la dirección opuesta al gradiente calculado, reduciremos gradualmente la función de energía. Si nuestra red neuronal está entrenada correctamente, entonces el ruido aleatorio debería transformarse ¡en una imagen que se asemeje a una observación del entrenamiento que tenemos ante nuestros ojos!

DINÁMICA DEL GRADIENTE ESTOCÁSTICO DE LANGEVIN

Es importante destacar que también debemos agregar una pequeña cantidad de ruido aleatorio a la entrada a medida que viajamos por el espacio muestral; de lo contrario, existe el riesgo de caer en mínimos locales. Por lo tanto, la técnica se conoce como dinámica de Langevin de gradiente estocástico. [3]

Podemos visualizar este descenso de gradiente como se muestra en la figura 7-4, para un espacio bidimensional con el valor de la función de energía en

la tercera dimensión. El camino es un ruidoso descenso cuesta abajo, siguiendo el gradiente negativo de la función de energía $E(x)$ con respecto a la entrada x . En el conjunto de datos de imágenes MNIST, tenemos 1024 píxeles, por lo que estamos navegando en un espacio de 1024 dimensiones, ¡pero se aplican los mismos principios!

Figura 7-4. Descenso de gradiente utilizando dinámica de Langevin.

Vale la pena señalar la diferencia entre este tipo de descenso de gradiente y el tipo de descenso de gradiente que normalmente utilizamos para entrenar una red neuronal.

Al entrenar una red neuronal, calculamos el gradiente de la función de pérdida con respecto a los *parámetros* de la red (es decir, los pesos) mediante retropropagación. Luego actualizamos los parámetros una pequeña cantidad en la dirección del gradiente negativo, de modo que, a lo largo de muchas iteraciones, minimicemos gradualmente la pérdida.

Con la dinámica de Langevin, mantenemos fijos los pesos de la red neuronal y calculamos el gradiente de la salida con respecto a la entrada. Luego actualizamos la entrada una pequeña cantidad en la dirección del gradiente negativo, de modo que a lo largo de muchas iteraciones, minimizamos gradualmente la producción (la puntuación de energía).

Ambos procesos utilizan la misma idea (descenso de gradiente), pero se aplican a diferentes funciones y con respecto a diferentes entidades.

Formalmente, la dinámica de Langevin se puede describir mediante la siguiente ecuación:

$$x^k = x^{k+1} - \eta \nabla_x E_\theta(x^{k-1}) + \omega$$

donde $\omega \sim N(0, \sigma)$ y $x^0 \sim \mathcal{U}(-1, 1)$. η es el hiperparámetro de tamaño de paso que debe ajustarse: si es demasiado grande, los pasos saltan por encima de los mínimos; si es demasiado pequeño, el algoritmo será demasiado lento para converger.

CONSEJO

$x^0 \sim \mathcal{U}(-1, 1)$ es la distribución uniforme en el rango $[-1, 1]$.

Podemos codificar nuestra función de muestreo de Langevin como se ilustra en el ejemplo 7-4.

Ejemplo 7-4. La función de muestreo de Langevin

```
def generate_samples(model, inp_imgs, steps, step_size, noise):
    imgs_per_step = []
    for _ in range(steps):
        inp_imgs += tf.random.normal(inp_imgs.shape, mean = 0,
                                     stddev = noise)
        inp_imgs = tf.clip_by_value(inp_imgs, -1.0, 1.0)
        with tf.GradientTape() as tape:
            tape.watch(inp_imgs)
            out_score = -model(inp_imgs)
        grads = tape.gradient(out_score, inp_imgs)
        grads = tf.clip_by_value(grads, -0.03, 0.03)
        inp_imgs += -step_size * grads
        inp_imgs = tf.clip_by_value(inp_imgs, -1.0, 1.0)
    return inp_imgs
```

- Recorre el número de pasos indicado.
- Agregue una pequeña cantidad de ruido a la imagen.
- Pase la imagen por el modelo para obtener la puntuación de energía.
- Calcule el gradiente de la salida con respecto a la entrada.
- Agregue una pequeña cantidad del degradado a la imagen de entrada.

Entrenamiento con divergencia contrastiva

Ahora que sabemos cómo muestrear un nuevo punto de baja energía del espacio muestral, centremos nuestra atención en entrenar el modelo.

No podemos aplicar la estimación de máxima verosimilitud porque la función de energía no genera una probabilidad; genera una puntuación que no se integra a 1 en todo el espacio muestral. En su lugar, aplicaremos una técnica propuesta por primera vez en 2002 por Geoffrey Hinton, llamado

divergencia contrastiva, para entrenar modelos de puntuación no normalizados. [4]

El valor que queremos minimizar (como siempre) es la probabilidad logarítmica negativa de los datos:

$$\mathcal{L} = -\mathbb{E}_{x \sim \text{datos}}[\log p_{\theta}(x)]$$

Cuando $p_{\theta}(x)$ tiene la forma de una distribución de Boltzmann, con función de energía $E_{\theta}(x)$, se puede demostrar que el gradiente de este valor se puede escribir de la siguiente manera (las “Notas sobre divergencia contrastiva” de Oliver Woodford para la derivación completa): [5]

$$\nabla_{\theta} \mathcal{L} = -\mathbb{E}_{x \sim \text{datos}}[\nabla_{\theta} E_{\theta}(x)] - \mathbb{E}_{x \sim \text{modelo}}[\nabla_{\theta} E_{\theta}(x)]$$

Esto intuitivamente tiene mucho sentido: queremos entrenar el modelo para que genere grandes puntuaciones de energía negativa para observaciones reales y grandes puntuaciones de energía positiva para observaciones falsas generadas para que el contraste entre estos dos extremos sea lo más grande posible.

En otras palabras, podemos calcular la diferencia entre las puntuaciones de energía de muestras reales y falsas y usarla como nuestra función de pérdida.

Para calcular las puntuaciones de energía de muestras falsas, necesitaríamos poder tomar muestras exactamente de la distribución $p_{\theta}(x)$, lo cual no es posible debido al denominador intratable. En cambio, podemos utilizar nuestro procedimiento de muestreo de Langevin para generar un conjunto de observaciones con puntuaciones de energía bajas. El proceso necesitaría ejecutarse durante una cantidad infinita de pasos para producir una muestra perfecta (lo cual obviamente no es práctico), por lo que en su lugar ejecutamos una pequeña cantidad de pasos, asumiendo que esto es lo suficientemente bueno como para producir una función de pérdida significativa.

También mantenemos un búfer de muestras de iteraciones anteriores, de modo que podamos usarlo como punto de partida para el siguiente lote, en

lugar de puro ruido aleatorio. El código para producir el búfer de muestreo se muestra en el Ejemplo 7-5.

Ejemplo 7-5. El buffer

```
class Buffer:
    def __init__(self, model):
        super().__init__()
        self.model = model
        self.examples = [
            tf.random.uniform(shape = (1, 32, 32, 1)) * 2 - 1
            for _ in range(128)
        ]

    def sample_new_exmps(self, steps, step_size, noise):
        n_new = np.random.binomial(128, 0.05)
        rand_imgs = (
            tf.random.uniform((n_new, 32, 32, 1)) * 2 - 1
        )
        old_imgs = tf.concat(
            random.choices(self.examples, k=128-n_new), axis=0
        )
        inp_imgs = tf.concat([rand_imgs, old_imgs], axis=0)
        inp_imgs = generate_samples(
            self.model, inp_imgs, steps=steps, step_size=step_size,
            noise = noise
        )
        self.examples = tf.split(inp_imgs, 128, axis = 0) +
        self.examples
        self.examples = self.examples[:8192]
        return inp_imgs
```

1. El búfer de muestreo se inicializa con un lote de ruido aleatorio.
2. En promedio, el 5% de las observaciones se generan desde cero (es decir, ruido aleatorio) cada vez.
3. El resto se extrae al azar del buffer existente.
4. Las observaciones se concatenan y recorren el muestreo de Langevin.
5. La muestra resultante se agrega al búfer, que se recorta a una longitud máxima de 8192 observaciones.

La Figura 7-5 muestra un paso de entrenamiento de divergencia contrastiva. El algoritmo reduce las puntuaciones de las observaciones reales y aumenta

las puntuaciones de las observaciones falsas, sin preocuparse por normalizar estas puntuaciones después de cada paso.

Figura 7-5. Un paso de divergencia contrastiva

Podemos codificar el paso de entrenamiento del algoritmo de divergencia contrastiva dentro de un modelo Keras personalizado como se muestra en el Ejemplo 7-6.

Ejemplo 7-6. EBM entrenado usando clase de divergencia contrastiva

```
class EBM(models.Model):
    def __init__(self):
        super(EBM, self).__init__()
        self.model = model
        self.buffer = Buffer(self.model)
        self.alpha = 0.1
        self.loss_metric = metrics.Mean(name="loss")
        self.reg_loss_metric = metrics.Mean(name="reg")
        self.cdiv_loss_metric = metrics.Mean(name="cdiv")
        self.real_out_metric = metrics.Mean(name="real")
        self.fake_out_metric = metrics.Mean(name="fake")

    @property
    def metrics(self):
        return [
            self.loss_metric,
            self.reg_loss_metric,
            self.cdiv_loss_metric,
            self.real_out_metric,
            self.fake_out_metric
        ]

    def train_step(self, real_imgs):
        real_imgs += tf.random.normal(
            shape=tf.shape(real_imgs), mean = 0, stddev = 0.005
        )
        real_imgs = tf.clip_by_value(real_imgs, -1.0, 1.0)
        fake_imgs = self.buffer.sample_new_exmps(
            steps=60, step_size=10, noise = 0.005
        )
        inp_imgs = tf.concat([real_imgs, fake_imgs], axis=0)
        with tf.GradientTape() as training_tape:
```

```

        real_out, fake_out = tf.split(self.model(inp_imgs), 2,
axis=0)
        cdiv_loss = tf.reduce_mean(fake_out, axis = 0) -
tf.reduce_mean(
            real_out, axis = 0
        )
        reg_loss = self.alpha * tf.reduce_mean(
            real_out ** 2 + fake_out ** 2, axis = 0
        )
        loss = reg_loss + cdiv_loss
        grads = training_tape.gradient(loss,
self.model.trainable_variables)
        self.optimizer.apply_gradients(
            zip(grads, self.model.trainable_variables)
        )
        self.loss_metric.update_state(loss)
        self.reg_loss_metric.update_state(reg_loss)
        self.cdiv_loss_metric.update_state(cdiv_loss)
        self.real_out_metric.update_state(tf.reduce_mean(real_out,
axis = 0))
        self.fake_out_metric.update_state(tf.reduce_mean(fake_out,
axis = 0))
    return {m.name: m.result() for m in self.metrics}

def test_step(self, real_imgs):
    batch_size = real_imgs.shape[0]
    fake_imgs = tf.random.uniform((batch_size, 32, 32, 1)) * 2
- 1
    inp_imgs = tf.concat([real_imgs, fake_imgs], axis=0)
    real_out, fake_out = tf.split(self.model(inp_imgs), 2,
axis=0)
    cdiv = tf.reduce_mean(fake_out, axis = 0) - tf.reduce_mean(
        real_out, axis = 0
    )
    self.cdiv_loss_metric.update_state(cdiv)
    self.real_out_metric.update_state(tf.reduce_mean(real_out,
axis = 0))
    self.fake_out_metric.update_state(tf.reduce_mean(fake_out,
axis = 0))
    return {m.name: m.result() for m in self.metrics[2:]}

ebm = EBM()
ebm.compile(optimizer=optimizers.Adam(learning_rate=0.0001),
run_eagerly=True)
ebm.fit(x_train, epochs=60, validation_data = x_test,)
```

1. Se agrega una pequeña cantidad de ruido aleatorio a las imágenes reales para evitar que el modelo se sobreajuste al conjunto de entrenamiento.
2. Se muestrea un conjunto de imágenes falsas del búfer.
3. Las imágenes reales y falsas se pasan por el modelo para producir puntuaciones reales y falsas.
4. La pérdida de divergencia contrastiva es simplemente la diferencia entre las puntuaciones de las observaciones reales y falsas.
5. Se agrega una pérdida de regularización para evitar que las puntuaciones sean demasiado grandes.
6. Los gradientes de la función de pérdida con respecto a los pesos de la red se calculan para la retropropagación.
7. `test_step` se utiliza durante la validación y calcula la divergencia contrastiva entre las puntuaciones de un conjunto de ruido aleatorio y los datos del conjunto de entrenamiento. Se puede utilizar como medida de qué tan bien se está entrenando el modelo (consulte la siguiente sección).

Análisis del modelo basado en energía

Las curvas de pérdida y las métricas de apoyo del proceso de capacitación se muestran en la Figura 7-6.

Figura 7-6. Curvas de pérdidas y métricas para el proceso de entrenamiento de la EBM.

En primer lugar, observe que la pérdida calculada durante el paso de entrenamiento es aproximadamente constante y pequeña en todas las épocas.

Si bien el modelo mejora constantemente, también lo hace la calidad de las imágenes generadas en el búfer que se requiere comparar con imágenes reales del conjunto de entrenamiento, por lo que no debemos esperar que la pérdida de entrenamiento disminuya significativamente.

Por lo tanto, para juzgar el rendimiento del modelo, también configuramos un proceso de validación que no toma muestras del búfer, sino que califica una muestra de ruido aleatorio y la compara con las puntuaciones de ejemplos del conjunto de entrenamiento. Si el modelo está mejorando, deberíamos ver que la divergencia contrastiva cae a lo largo de las épocas (es decir, está mejorando en la distinción del ruido aleatorio de las imágenes reales), como se puede ver en la Figura 7-6.

Generar nuevas muestras a partir del EBM es simplemente un caso de ejecutar el muestreador Langevin durante una gran cantidad de pasos, desde cero (ruido aleatorio), como se muestra en el ejemplo 7-7. La observación se fuerza cuesta abajo, siguiendo los gradientes de la función de puntuación con respecto a la entrada, de modo que fuera del ruido aparezca una observación plausible.

Ejemplo 7-7. Generando nuevas observaciones usando EBM

```
start_imgs = np.random.uniform(size = (10, 32, 32, 1)) * 2 - 1
gen_img = generate_samples(
    ebm.model,
    start_imgs,
    steps=1000,
    step_size=10,
    noise = 0.005,
    return_img_per_step=True,
)
```

En la Figura 7-7 se muestran algunos ejemplos de observaciones producidas por el muestreador después de 50 épocas de entrenamiento.

Figura 7-7. Ejemplos producidos por el muestreador Langevin utilizando el modelo EBM para dirigir el descenso del gradiente

Incluso podemos mostrar una repetición de cómo se genera una sola observación tomando instantáneas de las observaciones actuales durante el proceso de muestreo de Langevin; esto se muestra en la figura 7-8.

Figura 7-8. Instantáneas de una observación en diferentes pasos del proceso de muestreo de Langevin

Otros modelos basados en energía

En el ejemplo anterior utilizamos una EBM profunda entrenada utilizando divergencia contrastiva con un muestreador de dinámica de Langevin. Sin embargo, los primeros modelos de EBM no utilizaban el muestreo de Langevin, sino que se basaban en otras técnicas y arquitecturas.

Uno de los primeros ejemplos de EBM fue la máquina de *Boltzmann*. [6] Se trata de una red neuronal no dirigida y completamente conectada, donde las unidades binarias son *visibles* (v) u *ocultas* (h). La energía de una determinada configuración de la red se define de la siguiente manera:

$$E_\theta(v, h) = -\frac{1}{2}(v^T Lv + h^T Jh + v^T Wh)$$

donde W, L, J son las matrices de pesos aprendidas por el modelo. El entrenamiento se logra mediante divergencia contrastiva, pero utilizando el muestreo de Gibbs para alternar entre las capas visibles y ocultas hasta encontrar un equilibrio. En la práctica, esto es muy lento y no escalable a un gran número de unidades ocultas.

CONSEJO

Consulte la publicación del blog de Jessica Stringham "Gibbs Sampling in Python" para ver un excelente ejemplo simple de muestreo de Gibbs.

Una extensión de este modelo, la máquina restringida de *Boltzmann* (RBM), elimina las conexiones entre unidades del mismo tipo, creando así un gráfico bipartito de dos capas. Esto permite que los RBM se apilen en *redes de creencias profundas* para modelar distribuciones más complejas. Sin embargo, modelar datos de alta dimensión con RBM sigue siendo poco práctico, debido a que todavía se requiere el muestreo de Gibbs con tiempos de mezcla prolongados.

No fue hasta finales de la década de 2000 que se demostró que las EBM tenían potencial para modelar conjuntos de datos de mayor dimensión y se estableció un marco para construir EBM profundas. [7] La dinámica de Langevin se convirtió en el método de muestreo preferido para las EBM, que luego evolucionó hasta convertirse en una técnica de entrenamiento conocida como emparejamiento de puntuaciones. Esto se desarrolló aún más en una clase de modelo conocida como *modelo probabilístico de difusión de eliminación de ruido*, que impulsan modelos generativos de última generación como DALL.E 2 e ImageGen. Exploraremos los modelos de difusión con más detalle en el Capítulo 8.

Resumen

Los modelos basados en energía son una clase de modelo generativo que utiliza una función de puntuación de energía: una red neuronal entrenada para generar puntuaciones bajas para observaciones reales y puntuaciones altas para observaciones generadas. Calcular la distribución de probabilidad dada por esta función de puntuación requeriría normalizarla mediante un denominador intratable. Los EBM evitan este problema utilizando dos trucos: divergencia contrastiva para entrenar la red y dinámica de Langevin para muestrear nuevas observaciones.

La función de energía se entrena minimizando la diferencia entre las puntuaciones de las muestras generadas y las puntuaciones de los datos de entrenamiento, una técnica conocida como divergencia contrastiva. Se puede demostrar que esto es equivalente a minimizar la probabilidad logarítmica negativa, como lo requiere la estimación de máxima verosimilitud, pero no requiere que calculemos el denominador de normalización intratable. En la práctica, aproximamos el proceso de muestreo de las muestras falsas para garantizar que el algoritmo siga siendo eficiente.

El muestreo de EBM profundas se logra mediante la dinámica de Langevin, una técnica que utiliza el gradiente de la puntuación con respecto a la imagen de entrada para transformar gradualmente el ruido aleatorio en una observación plausible actualizando la entrada en pequeños pasos, siguiendo

el gradiente cuesta abajo. Esto mejora los métodos anteriores, como el muestreo de Gibbs, que utilizan máquinas Boltzmann restringidas.

1 Yilun Du e Igor Mordatch, “Generación implícita y modelado con modelos basados en energía”, 20 de marzo de 2019, <https://arxiv.org/abs/1903.08689>.

2 Prajit Ramachandran et al., “Buscando funciones de activación”, 16 de octubre de 2017, <https://arxiv.org/abs/1710.05941v2>.

3 Max Welling y Yee Whye Teh, “Aprendizaje bayesiano vía dinámica de gradiente estocástico de Langevin”, 2011, <https://www.stats.ox.ac.uk/~teh/research/compstats/WelTeh2011a.pdf>

4 Geoffrey E. Hinton, “Productos de formación de expertos minimizando la divergencia contrastiva”, 2002, <https://www.cs.toronto.edu/~hinton/absps/tr00-004.pdf>.

5 Oliver Woodford, “Notas sobre divergencia contrastiva”, 2006, <https://www.robots.ox.ac.uk/~ojw/files/NotesOnCD.pdf>.

6 David H. Ackley et al., "Un algoritmo de aprendizaje para máquinas Boltzmann", 1985, Ciencia cognitiva 9(1), 147-165.

7 Yann Lecun et al., “Un tutorial sobre aprendizaje basado en energía”, 2006, https://www.researchgate.net/publication/200744586_A_tutorial_on_energy-based_learning.

Capítulo 8. Modelos de difusión

METAS DEL CAPÍTULO

En este capítulo podrás:

- Conocer los principios y componentes subyacentes que definen un modelo de difusión.
- Ver cómo se utiliza el proceso de avance para agregar ruido al conjunto de imágenes de entrenamiento.
- Comprender el truco de la reparametrización y por qué es importante.
- Explorar diferentes formas de programación de difusión directa.
- Comprender el proceso de difusión inversa y cómo se relaciona con el proceso de ruido directo.
- Explorar la arquitectura de U-Net, que se utiliza para parametrizar el proceso de difusión inversa.
- Construir su propio modelo de difusión de eliminación de ruido (DDM) utilizando Keras para generar imágenes de flores.
- Mostrar nuevas imágenes de flores de tu modelo.
- Explorar el efecto del número de pasos de difusión en la calidad de la imagen e interpola entre dos imágenes en el espacio latente.

Junto con las GAN, los modelos de difusión son una de las técnicas de modelado generativo más influyentes e impactantes para la generación de imágenes que se han introducido en la última década. En muchos puntos de referencia, los modelos de difusión ahora superan a las GAN de última generación y se están convirtiendo rápidamente en la opción preferida para los profesionales del modelado generativo, particularmente para dominios visuales (por ejemplo, DALL.E 2 de OpenAI e ImageGen de Google para transferencia de texto a -generación de imágenes). Recientemente, ha habido una explosión de modelos de difusión que se aplican en una amplia gama de tareas, lo que recuerda a la proliferación de GAN que tuvo lugar entre 2017 y 2020.

Muchas de las ideas centrales que sustentan los modelos de difusión comparten similitudes con tipos anteriores de modelos generativos que ya hemos explorado en este libro (por ejemplo, codificadores automáticos con eliminación de ruido, modelos basados en energía). De hecho, el nombre difusión se inspira en la bien estudiada propiedad de la difusión termodinámica: en 2015 se estableció un

vínculo importante entre este campo puramente físico y el aprendizaje profundo. [1]

También se estaban logrando avances importantes en el campo de los modelos generativos basados en puntuaciones,^{2,3} una rama de la modelización basada en energía que estima directamente el gradiente de la distribución logarítmica (también conocida como función de puntuación) para entrenar el modelo. como alternativa al uso de divergencia contrastiva. En particular, Yang Song y Stefano Ermon utilizaron múltiples escalas de perturbaciones del ruido aplicado a los datos sin procesar para garantizar que el modelo, una red de puntuación condicional de ruido (NCSN), funcione bien en regiones de baja densidad de datos.

El revolucionario documento modelo de difusión llegó en el verano de 2020. [4] Sobre la base de trabajos anteriores, el artículo descubre una conexión profunda entre los modelos de difusión y los modelos generativos basados en puntuaciones, y los autores utilizan este hecho para entrenar un modelo de difusión que puede rivalizar con las GAN en varios conjuntos de datos, llamado *Modelo probabilístico de difusión de eliminación de ruido* (DDPM).

Este capítulo analizará los requisitos teóricos para comprender cómo funciona un modelo de difusión de eliminación de ruido. Luego aprenderá cómo construir su propio modelo de difusión de eliminación de ruido utilizando Keras.

Introducción

Para ayudar a explicar las ideas clave que sustentan los modelos de difusión, ¡comencemos con una breve historia!

DIFUSIÓN TV

Estás parado en una tienda de electrónica que vende televisores. Sin embargo, esta tienda es claramente muy diferente a las que has visitado en el pasado. En lugar de una amplia variedad de marcas diferentes, hay cientos de copias idénticas del mismo televisor conectadas en secuencia, extendiéndose hasta la parte trasera de la tienda hasta donde alcanza la vista. Es más, los primeros televisores parecen no mostrar nada más que ruido estático aleatorio (Figura 8-1).

El comerciante se acerca para preguntarle si necesita ayuda. Confundido, le preguntas sobre la extraña configuración. Ella explica que este es el nuevo modelo DiffuseTV que está destinado a revolucionar la industria del entretenimiento e

inmediatamente comienza a contarle cómo funciona, mientras se adentra en la tienda, junto a la línea de televisores.

Figura 8-1. Una larga fila de televisores conectados que se extiende a lo largo de un pasillo de una tienda (creada con Midjourney)

Ella explica que durante el proceso de fabricación, DiffuseTV está expuesto a miles de imágenes de programas de televisión anteriores, pero cada una de esas imágenes se ha corrompido gradualmente con estática aleatoria, hasta que es indistinguible del ruido aleatorio puro. Luego, los televisores se diseñan para deshacer el ruido aleatorio, en pequeños pasos, esencialmente tratando de predecir cómo se veían las imágenes antes de que se agregara el ruido. Puede ver que a medida que avanza en la tienda, las imágenes de cada televisor son un poco más claras que las anteriores.

Finalmente llegas al final de la larga fila de televisores, donde puedes ver una imagen perfecta en el último televisor. Si bien esta es ciertamente una tecnología inteligente, usted tiene curiosidad por comprender cómo es útil para el espectador. La comerciante continúa con su explicación.

En lugar de elegir un canal para mirar, el espectador elige una configuración inicial aleatoria de estática. Cada configuración conducirá a una imagen de salida diferente y, en algunos modelos, incluso puede guiarse mediante un mensaje de texto que usted elija ingresar. A diferencia de un televisor normal, con una gama limitada de canales para ver, DiffuseTV ofrece al espectador opciones ilimitadas y libertad para generar lo que quiera que aparezca en la pantalla.

Usted compra un DiffuseTV de inmediato y se siente aliviado al saber que la larga fila de televisores en la tienda es solo para fines de demostración, por lo que no tendrá que comprar también un almacén para almacenar su nuevo dispositivo.

La historia de DiffuseTV describe la idea general detrás de un modelo de difusión. Ahora profundicemos en los aspectos técnicos de cómo construimos un modelo de este tipo utilizando Keras.

Modelos de difusión de eliminación de ruido (DDM)

La idea central detrás de un modelo de difusión de eliminación de ruido es simple: entrenamos un modelo de aprendizaje profundo para eliminar el ruido de una

imagen en una serie de pasos muy pequeños. Si partimos de ruido aleatorio puro, en teoría deberíamos poder seguir aplicando el modelo hasta que obtengamos una imagen que parezca extraída del conjunto de entrenamiento. ¡Lo sorprendente es que este concepto simple funcione tan bien en la práctica!

Primero configuremos un conjunto de datos y luego repasemos los procesos de difusión hacia adelante (ruido) y hacia atrás (eliminación de ruido).

EJECUTANDO EL CÓDIGO PARA ESTE EJEMPLO

El código de este ejemplo se puede encontrar en el cuaderno de Jupyter ubicado en notebooks/08_diffusion/01_ddm/ddm.ipynb en el repositorio de libros.

El código está adaptado del excelente tutorial sobre modelos implícitos de difusión de eliminación de ruido creado por András Béres disponible en el sitio web de Keras.

El conjunto de datos de flores

Usaremos el conjunto de datos Oxford 102 de flores que está disponible a través de Kaggle. Se trata de un conjunto de más de 8,000 imágenes en color de una variedad de flores.

Puede descargar el conjunto de datos ejecutando el script de descarga del conjunto de datos de Kaggle en el repositorio de libros, como se muestra en el Ejemplo 8-1. Esto guardará las imágenes de flores en la carpeta /data.

Ejemplo 8-1. Descargando el conjunto de datos Oxford 102 de flores

```
bash scripts/download_kaggle_data.sh nunenuh pytorch-challange-flower-dataset
```

Como de costumbre, cargaremos las imágenes usando la función `image_dataset_from_directory` de Keras, cambiaremos el tamaño de las imágenes a 64×64 píxeles y escalaremos los valores de píxeles al rango [0, 1]. También repetiremos el conjunto de datos cinco veces para aumentar la duración de la época y agrupar los datos en grupos de 64 imágenes, como se muestra en el Ejemplo 8-2.

Ejemplo 8-2. Cargando el conjunto de datos de Oxford 102 de flores

```

train_data = utils.image_dataset_from_directory(
    "/app/data/pytorch-challange-flower-dataset/dataset",
    labels=None,
    image_size=(64, 64),
    batch_size=None,
    shuffle=True,
    seed=42,
    interpolation="bilinear",
)

def preprocess(img):
    img = tf.cast(img, "float32") / 255.0
    return img

train = train_data.map(lambda x: preprocess(x))
train = train.repeat(5)
train = train.batch(64, drop_remainder=True)

```

1. Cargue el conjunto de datos (cuando sea necesario durante el entrenamiento) utilizando la función Keras `image_dataset_from_directory`.
2. Escale los valores de píxeles al rango [0, 1].
3. Repita el conjunto de datos cinco veces.
4. Divida el conjunto de datos en grupos de 64 imágenes.

En la Figura 8-2 se muestran imágenes de ejemplo del conjunto de datos.

Figura 8-2. Imágenes de ejemplo del conjunto de datos Oxford 102 de flores

Ahora que tenemos nuestro conjunto de datos, podemos explorar cómo debemos agregar ruido a las imágenes mediante un proceso de difusión directa.

El proceso de difusión hacia adelante

Supongamos que tenemos una imagen x_0 que queremos corromper gradualmente a lo largo de un gran número de pasos (digamos, $T = 1,000$), de modo que eventualmente sea indistinguible de ruido gaussiano estándar (es decir, x_T debe tener media cero y varianza unitaria). ¿Cómo deberíamos hacer esto?

Podemos definir una función q que agrega una pequeña cantidad de ruido gaussiano con varianza β_t a una imagen x_{t-1} para generar una nueva imagen x_t . Si seguimos aplicando esta función, generaremos una secuencia de imágenes progresivamente más ruidosas (x_0, \dots, x_T), como se muestra en la Figura 8-3.

Figura 8-3. El proceso de difusión directa q

Podemos escribir este proceso de actualización matemáticamente de la siguiente manera (aquí, ϵ_{t-1} es un gaussiano estándar con media cero y varianza unitaria):

$$x_t = \sqrt{1 - \beta_t} x_{t-1} + \sqrt{\beta_t} \epsilon_{t-1}$$

Tenga en cuenta que también escalamos la imagen de entrada x_{t-1} , para garantizar que la varianza de la imagen de salida x_t permanezca constante a lo largo del tiempo. De esta manera, si normalizamos nuestra imagen original x_0 para que tenga media cero y varianza unitaria, entonces x_T se aproximará a una distribución gaussiana estándar para T lo suficientemente grande, por inducción, de la siguiente manera.

Si suponemos que x_{t-1} tiene media cero y varianza unitaria entonces $\sqrt{1 - \beta_t} x_{t-1}$ tendrá varianza $1 - \beta_t$ y $\sqrt{\beta_t} \epsilon_{t-1}$ tendrá varianza β_t , usando la regla de que $\text{Var}(aX) = a^2 \text{Var}(X)$. Sumando esto, obtenemos una nueva distribución x_t con media cero y varianza $1 - \beta_t + \beta_t = 1$, usando la regla que $\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y)$ para X e Y independientes. Por lo tanto, si x_0 se normaliza a una media cero y una varianza unitaria, entonces garantizamos que esto también es cierto para todo x_t , incluida la imagen final x_T , que se aproximará a una distribución gaussiana estándar. ¡Esto es exactamente lo que necesitamos, ya que queremos poder muestrear fácilmente x_T y luego aplicar un proceso de difusión inversa a través de nuestro modelo de red neuronal entrenado!

En otras palabras, nuestro proceso de ruido directo q también se puede escribir de la siguiente manera:

$$q(x_t | x_{t-1}) = \mathcal{N}(x_t; \sqrt{1 - \beta_t} x_{t-1}, \beta_t I)$$

El truco de la reparametrización

También sería útil poder saltar directamente desde una imagen x_0 a cualquier versión con ruido de la imagen x_T sin tener que pasar por t aplicaciones de q . Afortunadamente, existe un truco de reparametrización que podemos utilizar para hacer esto.

si definimos $\alpha_t = 1 - \beta_t$ y $\bar{\alpha}_t = \prod_{i=1}^t \alpha_i$, entonces podemos escribir lo siguiente:

$$\begin{aligned} x_t &= \sqrt{\alpha_t} x_{t-1} + \sqrt{1 - \alpha_t} \epsilon_{t-1} \\ &= \sqrt{\alpha_t \alpha_{t-1}} x_{t-2} + \sqrt{1 - \alpha_t \alpha_{t-1}} \epsilon \\ &= \dots \\ &= \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t} \epsilon \end{aligned}$$

Tenga en cuenta que la segunda línea utiliza el hecho de que podemos sumar dos gaussianos para obtener un nuevo gaussiano. Por lo tanto, tenemos una manera de saltar desde la imagen original x_0 a cualquier paso del proceso de difusión directa x_t . Además, podemos definir el programa de difusión usando los valores de $\bar{\alpha}_t$, en lugar de los valores de β_t originales, con la interpretación de que $\bar{\alpha}_t$ es la varianza debida a la señal (la imagen original, x_0) y $1 - \bar{\alpha}_t$ es la varianza debida al ruido (ϵ).

Por lo tanto, el proceso de difusión directa q también se puede escribir de la siguiente manera:

$$q(x_t | x_0) = \mathcal{N}(x_t; \sqrt{\bar{\alpha}_t} x_0, (1 - \bar{\alpha}_t)I)$$

Programas de difusión

Observe que también somos libres de elegir un β_t diferente en cada paso de tiempo; no todos tienen que ser iguales. La forma en que cambian los valores β_t (o $\bar{\alpha}_t$) se denomina *programa de difusión*.

En el artículo original (Ho et al., 2020), los autores eligieron un programa de difusión lineal para β_t , es decir, β_t aumenta linealmente con t , de $\beta_1 = 0.0001$ a $\beta_T = 0.02$. Esto asegura que en las primeras etapas del proceso de generación de ruido demos pasos de generación de ruido más pequeños que en las etapas posteriores, cuando la imagen ya tiene mucho ruido.

Podemos codificar un programa de difusión lineal como se muestra en el ejemplo 8-3.

Ejemplo 8-3. El programa de difusión lineal

```
def linear_diffusion_schedule(diffusion_times):
    min_rate = 0.0001
    max_rate = 0.02
    betas = min_rate + tf.convert_to_tensor(diffusion_times) *
(max_rate - min_rate)
    alphas = 1 - betas
    alpha_bars = tf.math.cumprod(alphas)
    signal_rates = alpha_bars
    noise_rates = 1 - alpha_bars
    return noise_rates, signal_rates

T = 1000
diffusion_times = [x/T for x in range(T)]
linear_noise_rates, linear_signal_rates = linear_diffusion_schedule(
    diffusion_times
)
```

1. Los tiempos de difusión son pasos equidistantes entre 0 y 1.
2. El programa de difusión lineal se aplica a los tiempos de difusión para producir el ruido y las tasas de señal.

En un artículo posterior se descubrió que un *programa de difusión coseno* superaba al programa lineal del artículo original. [5] Un programa coseno define los siguientes valores de $\bar{\alpha}_t$:

$$\bar{\alpha}_t = \cos^2\left(\frac{t}{T} \times \frac{\pi}{2}\right)$$

Por lo tanto, la ecuación actualizada es la siguiente (usando la identidad trigonométrica $\cos^2(x) + \sin^2(x) = 1$):

$$x_t = \cos\left(\frac{t}{T} \times \frac{\pi}{2}\right)x_0 + \sin\left(\frac{t}{T} \times \frac{\pi}{2}\right)\epsilon$$

Esta ecuación es una versión simplificada del programa de difusión del coseno real utilizado en el artículo. Los autores también agregan un término de compensación y una escala para evitar que los pasos de ruido sean demasiado pequeños al comienzo del proceso de difusión. Podemos codificar el coseno y compensar los programas de difusión del coseno como se muestra en el ejemplo 8-4.

Ejemplo 8-4. Los programas de difusión de coseno y coseno compensado

```
def cosine_diffusion_schedule(diffusion_times):
    signal_rates = tf.cos(diffusion_times * math.pi / 2)
```

```

noise_rates = tf.sin(diffusion_times * math.pi / 2)
return noise_rates, signal_rates

def offset_cosine_diffusion_schedule(diffusion_times):
    min_signal_rate = 0.02
    max_signal_rate = 0.95
    start_angle = tf.acos(max_signal_rate)
    end_angle = tf.acos(min_signal_rate)

    diffusion_angles = start_angle + diffusion_times * (end_angle -
    start_angle)

    signal_rates = tf.cos(diffusion_angles)
    noise_rates = tf.sin(diffusion_angles)

return noise_rates, signal_rates

```

1. El programa de difusión del coseno puro (sin compensación ni reescalado).
2. El programa de difusión del coseno compensado que usaremos, que ajusta el programa para garantizar que los pasos de ruido no sean demasiado pequeños al inicio del proceso de ruido.

Podemos calcular los valores $\bar{\alpha}_t$ de cada uno para mostrar cuánta señal ($\bar{\alpha}_t$) y ruido ($1 - \bar{\alpha}_t$) se deja pasar en cada etapa del proceso para las secuencias lineal, coseno, y compensar los programas de difusión del coseno, como se muestra en la Figura 8-4.

Figura 8-4. La señal y el ruido en cada paso del proceso de generación de ruido, para los programas de difusión lineal, coseno y coseno compensado.

Observe cómo el nivel de ruido aumenta más lentamente en el programa de difusión del coseno. Un programa de difusión coseno agrega ruido a la imagen de manera más gradual que un programa de difusión lineal, lo que mejora la eficiencia del entrenamiento y la calidad de la generación. Esto también se puede ver en imágenes que han sido corrompidas por los programas lineal y coseno (Figura 8-5).

Figura 8-5. Una imagen corrompida por los programas de difusión lineal (arriba) y coseno (abajo), en valores de t igualmente espaciados de 0 a T (fuente: Ho et al., 2020)

El proceso de difusión inversa

Ahora veamos el proceso de difusión inversa. En resumen, buscamos construir una red neuronal $p_\theta(x_{t-1}|x_t)$ que pueda deshacer el proceso de generación de ruido, es decir, aproximarse a la distribución inversa $q(x_{t-1}|x_t)$. Si podemos hacer esto, podemos muestrear ruido aleatorio de $\mathcal{N}(0, I)$ y luego aplicar el proceso de difusión inversa varias veces para generar una imagen novedosa. Esto se visualiza en la Figura 8-6.

Figura 8-6. El proceso de difusión inversa $p_\theta(x_{t-1}|x_t)$ intenta deshacer el ruido producido por el proceso de difusión directa

Existen muchas similitudes entre el proceso de difusión inversa y el decodificador de un autocodificador variacional. En ambos, nuestro objetivo es transformar el ruido aleatorio en una salida significativa utilizando una red neuronal. La diferencia entre los modelos de difusión y VAE es que en un VAE el proceso directo (convertir imágenes en ruido) es parte del modelo (es decir, se aprende), mientras que en un modelo de difusión no está parametrizado.

Por lo tanto, tiene sentido aplicar la misma función de pérdida que en un auto codificador variacional. El artículo original de DDPM deriva la forma exacta de esta función de pérdida y muestra que se puede optimizar entrenando una red ϵ_θ para predecir el ruido ϵ que se ha agregado a una imagen determinada x_0 en el paso de tiempo t .

En otras palabras, tomamos una muestra de una imagen x_0 y la transformamos mediante t pasos de ruido para obtener la imagen $x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon$. Le damos esta nueva imagen y la tasa de ruido $\bar{\alpha}_t$ a la red neuronal y le pedimos que prediga ϵ , tomando un paso de gradiente contra el error al cuadrado entre la predicción $\epsilon_\theta(x_t)$ y la ϵ verdadera.

Analizaremos la estructura de la red neuronal en la siguiente sección. Vale la pena señalar aquí que el modelo de difusión en realidad mantiene dos copias de la red: una que está entrenada activamente y utiliza el descenso de gradiente y otra (la red EMA) que es un promedio móvil exponencial (EMA) de los pesos de la red entrenada activamente durante los pasos de entrenamiento anteriores. La red EMA no es tan susceptible a fluctuaciones y picos a corto plazo en el proceso de capacitación, lo que la hace más robusta para la generación que la red entrenada

activamente. Por lo tanto, utilizamos la red EMA siempre que queramos producir resultados generados a partir de la red.

El proceso de entrenamiento del modelo se muestra en la Figura 8-7.

Figura 8-7. El proceso de entrenamiento para un modelo de difusión de eliminación de ruido (fuente: Ho et al., 2020)

En Keras, podemos codificar este paso de entrenamiento como se ilustra en el Ejemplo 8-5.

Ejemplo 8-5. La función train_step del modelo de difusión de Keras

```
class DiffusionModel(models.Model):
    def __init__(self):
        super().__init__()
        self.normalizer = layers.Normalization()
        self.network = unet
        self.ema_network = models.clone_model(self.network)
        self.diffusion_schedule = cosine_diffusion_schedule

    ...

    def denoise(self, noisy_images, noise_rates, signal_rates,
               training):
        if training:
            network = self.network
        else:
            network = self.ema_network
        pred_noises = network(
            [noisy_images, noise_rates**2], training=training
        )
        pred_images = (noisy_images - noise_rates * pred_noises) /
        signal_rates

    return pred_noises, pred_images

    def train_step(self, images):
        images = self.normalizer(images, training=True)
        noises = tf.random.normal(shape=tf.shape(images))
        batch_size = tf.shape(images)[0]
        diffusion_times = tf.random.uniform(
            shape=(batch_size, 1, 1, 1), minval=0.0, maxval=1.0
        )
        noise_rates, signal_rates = self.cosine_diffusion_schedule(
            diffusion_times
        )
```

```

noisy_images = signal_rates * images + noise_rates * noises
with tf.GradientTape() as tape:
    pred_noises, pred_images = self.denoise(
        noisy_images, noise_rates, signal_rates, training=True
    )
    noise_loss = self.loss(noises, pred_noises)
gradents = tape.gradient(noise_loss,
self.network.trainable_weights)
self.optimizer.apply_gradients(
    zip(gradents, self.network.trainable_weights)
)
self.noise_loss_tracker.update_state(noise_loss)

for weight, ema_weight in zip(
    self.network.weights, self.ema_network.weights
):
    ema_weight.assign(0.999 * ema_weight + (1 - 0.999) * weight)
return {m.name: m.result() for m in self.metrics}
...

```

1. Primero normalizamos el lote de imágenes para que tenga media cero y varianza unitaria.
2. A continuación, tomamos muestras de ruido para que coincida con la forma de las imágenes de entrada.
3. También tomamos muestras de tiempos de difusión aleatorios...
4. ...y los utilizamos para generar el ruido y las tasas de señal de acuerdo con el programa de difusión del coseno.
5. Luego aplicamos las ponderaciones de señal y ruido a las imágenes de entrada para generar imágenes ruidosas.
6. A continuación, eliminamos el ruido de las imágenes ruidosas pidiéndole a la red que prediga el ruido y luego deshaciendo la operación de ruido, utilizando las tasas de ruido y las tasas de señal proporcionadas.
7. Luego podemos calcular la pérdida (error absoluto medio) entre el ruido predicho y el ruido verdadero...
8. ...y dar un paso de gradiente contra esta función de pérdida.
9. Las ponderaciones de la red EMA se actualizan a un promedio ponderado de los pesos de las EMA existentes y pesos de red entrenados después del paso de gradiente.

El modelo de eliminación de ruido de U-Net

Ahora que hemos visto el tipo de red neuronal que necesitamos construir (una que prediga el ruido agregado a una imagen determinada), podemos observar la arquitectura que lo hace posible.

Los autores del artículo de DDPM utilizaron un tipo de arquitectura conocida como *U-Net*. En la Figura 8-8 se muestra un diagrama de esta red, que muestra explícitamente la forma del tensor a medida que pasa a través de la red.

Figura 8-8. Diagrama de arquitectura U-Net

De manera similar a un auto codificador variacional, una U-Net consta de dos mitades: la mitad de muestreo descendente, donde las imágenes de entrada se comprimen espacialmente pero se expanden por canales, y la mitad de muestreo de aumento, donde las representaciones se expanden espacialmente mientras el número de canales es reducido. Sin embargo, a diferencia de un VAE, también hay conexiones de salto entre capas de forma espacial equivalentes en las partes de muestreo ascendente y descendente de la red. Un VAE es secuencial; los datos fluyen a través de la red desde la entrada hasta la salida, una capa tras otra. Una U-Net es diferente, porque las conexiones de salto permiten que la información atajo partes de la red y fluya a capas posteriores.

Una U-Net es particularmente útil cuando queremos que la salida tenga la misma forma que la entrada. En nuestro ejemplo de modelo de difusión, queremos predecir el ruido agregado a una imagen, que tiene exactamente la misma forma que la imagen misma, por lo que U-Net es la elección natural para la arquitectura de red.

Primero, echemos un vistazo al código que construye esta U-Net en Keras, que se muestra en el ejemplo 8-6.

Ejemplo 8-6. Un modelo U-Net en Keras

```
noisy_images = layers.Input(shape=(64, 64, 3))
x = layers.Conv2D(32, kernel_size=1)(noisy_images)

noise_variances = layers.Input(shape=(1, 1, 1))
noise_embedding = layers.Lambda(sinusoidal_embedding)
(noise_variances)
noise_embedding = layers.UpSampling2D(size=64,
interpolation="nearest")(
    noise_embedding
)

x = layers.concatenate([x, noise_embedding])

skips = []
```

```

x = DownBlock(32, block_depth = 2)([x, skips])
x = DownBlock(64, block_depth = 2)([x, skips])
x = DownBlock(96, block_depth = 2)([x, skips])

x = ResidualBlock(128)(x)
x = ResidualBlock(128)(x)

x = UpBlock(96, block_depth = 2)([x, skips])
x = UpBlock(64, block_depth = 2)([x, skips])
x = UpBlock(32, block_depth = 2)([x, skips])

x = layers.Conv2D(3, kernel_size=1, kernel_initializer="zeros")(x)

unet = models.Model([noisy_images, noise_variances], x, name="unet")

```

1. La primera entrada a U-Net es la imagen que deseamos eliminar de ruido.
2. Esta imagen pasa a través de una capa Conv2D para aumentar la cantidad de canales.
3. La segunda entrada a U-Net es la varianza del ruido (un escalar).
4. Esto se codifica mediante una incrustación sinusoidal.
5. Esta incrustación se copia en dimensiones espaciales para que coincida con el tamaño de la imagen de entrada.
6. Los dos flujos de entrada están concatenados entre canales.
7. La lista de omisiones contendrá la salida de las capas DownBlock que deseamos conectar a las capas UpBlock aguas abajo.
8. El tensor se pasa a través de una serie de capas DownBlock que reducen el tamaño de la imagen y aumentan el número de canales.
9. Luego, el tensor pasa a través de dos capas ResidualBlock que mantienen constante el tamaño de la imagen y el número de canales.
10. A continuación, el tensor pasa a través de una serie de capas UpBlock que aumentan el tamaño de la imagen y reducen el número de canales. Las conexiones de salto incorporan salida de las capas DownBlock anteriores.
11. La capa final Conv2D reduce el número de canales a tres (RGB).
12. U-Net es un modelo de Keras que toma las imágenes ruidosas y las variaciones de ruido como entrada y genera un mapa de ruido previsto.

Para comprender U-Net en detalle, necesitamos explorar cuatro conceptos más: la incrustación sinusoidal de la varianza del ruido, ResidualBlock, DownBlock y UpBlock.

Incrustación sinusoidal

La *incrustación sinusoidal* se introdujo por primera vez en un artículo de Vaswani et al.⁶ Usaremos una adaptación de esa idea original tal como se utiliza en el artículo de Mildenhall et al. titulado “NeRF: Representing Scenes as Neural Radiance Fields for View Síntesis.” [7]

La idea es que queremos poder convertir un valor escalar (la varianza del ruido) en un vector distinto de dimensiones superiores que pueda proporcionar una representación más compleja, para su uso posterior en la red. El artículo original utilizó esta idea para codificar la posición discreta de palabras en una oración en vectores; El artículo de NeRF extiende esta idea a valores continuos.

Específicamente, un valor escalar x se codifica como se muestra en la siguiente ecuación:

$$\gamma(x) = \left(\sin(2\pi e^{\theta f} x), \dots, \sin(2\pi e^{(L-1)f} x), \cos(2\pi e^{\theta f} x), \dots, \cos(2\pi e^{(L-1)f} x) \right)$$

donde elegimos $L = 16$ para que sea la mitad del tamaño de nuestra longitud de incrustación de ruido deseada y $f = \frac{\ln(1000)}{(L-1)}$ para ser el factor de escala máximo para las frecuencias. Esto produce el patrón de incrustación que se muestra en la Figura 8-9.

Figura 8-9. El patrón de incrustaciones sinusoidales para variaciones de ruido de 0 a 1.

Podemos codificar esta función de incrustación sinusoidal como se muestra en el Ejemplo 8-7. Esto convierte un valor escalar de varianza de ruido único en un vector de longitud 32.

Ejemplo 8-7. La función `sinusoidal_embedding` que codifica la variación del ruido

```
def sinusoidal_embedding(x):
    frequencies = tf.exp(
        tf.linspace(
            tf.math.log(1.0),
            tf.math.log(1000.0),
            16,
        )
    )
    angular_speeds = 2.0 * math.pi * frequencies
    embeddings = tf.concat(
```

```

        [tf.sin(angular_speeds * x), tf.cos(angular_speeds * x)], axis=3
    )
return embeddings

```

Bloque residual

Tanto `DownBlock` como `UpBlock` contienen capas `ResidualBlock`, así que comencemos con estas. Ya exploramos los bloques residuales en el Capítulo 5, cuando construimos un PixelCNN, pero los recapitularemos aquí para que estén completos.

Un bloque residual es un grupo de capas que contiene una conexión de salto que agrega la entrada a la salida. Los bloques residuales nos ayudan a construir redes más profundas que pueden aprender patrones más complejos sin sufrir tanto por los problemas de degradación y gradiente que desaparecen. El problema del gradiente evanescente es la afirmación de que a medida que la red se hace más profunda, el gradiente que se propaga a través de capas más profundas es pequeño y, por tanto, el aprendizaje es muy lento. El problema de la degradación es el hecho de que a medida que las redes neuronales se vuelven más profundas, no son necesariamente tan precisas como sus contrapartes menos profundas: la precisión parece saturarse a cierta profundidad y luego degradarse rápidamente.

DEGRADACIÓN

El problema de la degradación es algo contradictorio, pero se observa en la práctica cuando las capas más profundas deben al menos aprender el mapeo de identidad, lo cual no es trivial, especialmente considerando otros problemas que enfrentan las redes más profundas, como el problema del gradiente que desaparece.

La solución, presentada por primera vez en el artículo de ResNet de He et al. en 2015,⁸ es muy sencillo. Al incluir una *autopista* de conexión de salto alrededor de las capas ponderadas principales, el bloque tiene la opción de omitir las actualizaciones de peso complejas y simplemente pasar por el mapeo de identidad. Esto permite entrenar la red a gran profundidad sin sacrificar el tamaño del gradiente o la precisión de la red.

En la Figura 8-10 se muestra un diagrama de un `ResidualBlock`. Tenga en cuenta que en algunos bloques residuales, también incluimos una capa `Conv2D` adicional con tamaño de kernel 1 en la conexión de omisión, para alinear la cantidad de canales con el resto del bloque.

Figura 8-10. El bloque residual en la U-Net

Podemos codificar un `ResidualBlock` en Keras como se muestra en el Ejemplo 8-8.

Ejemplo 8-8. Código para ResidualBlock en la U-Net

```
def ResidualBlock(width):
    def apply(x): input_width = x.shape[3]
        if input_width == width:
            residual = x
        else:
            residual = layers.Conv2D(width, kernel_size=1)(x)
            x = layers.BatchNormalization(center=False, scale=False)(x)
            x = layers.Conv2D(
                width, kernel_size=3, padding="same",
                activation=activations.swish
            )(x)
            x = layers.Conv2D(width, kernel_size=3, padding="same")(x)
            x = layers.Add()([x, residual])
        return x
    return apply
```

1. Comprueba si la cantidad de canales en la entrada coincide con la cantidad de canales que nos gustaría que emitía el bloque. Si no, incluye una capa `Conv2D` extra en la conexión de salto para alinear el número de canales con el resto del bloque.
2. Aplica una capa `BatchNormalization`.
3. Aplica dos capas de `Conv2D`.
4. Agrega la entrada del bloque original a la salida para proporcionar la salida final del bloque.

Bloques abajo (DownBlock) y bloques arriba (UpBlock)

Cada `DownBlock` sucesivo aumenta el número de canales a través de `block_depth` (=2 en nuestro ejemplo) `ResidualBlocks`, al mismo tiempo que aplica una capa final `AveragePooling2D` para reducir a la mitad el tamaño de la imagen. Cada `ResidualBlock` se agrega a una lista para que las capas `UpBlock` lo utilicen más adelante como conexiones de salto a través de U-Net.

Un `UpBlock` primero aplica una capa `UpSampling2D` que duplica el tamaño de la imagen, mediante interpolación bilineal. Cada `UpBlock` sucesivo disminuye el

número de canales a través de `block_depth` (=2) `ResidualBlocks`, al mismo tiempo que concatena las salidas de los `DownBlocks` a través de conexiones de salto a través de U-Net. En la Figura 8-11 se muestra un diagrama de este proceso.

Figura 8-11. El DownBlock y el UpBlock correspondiente en la U-Net

Podemos codificar `DownBlock` y `UpBlock` usando Keras como se ilustra en el ejemplo 8-9.

Ejemplo 8-9. Código para `DownBlock` y `UpBlock` en el modelo U-Net.

```
def DownBlock(width, block_depth):
    def apply(x):
        x, skips = x
        for _ in range(block_depth):
            x = ResidualBlock(width)(x)
            skips.append(x)
        x = layers.AveragePooling2D(pool_size=2)(x)
        return x

    return apply

def UpBlock(width, block_depth):
    def apply(x):
        x, skips = x
        x = layers.UpSampling2D(size=2, interpolation="bilinear")(x)
        for _ in range(block_depth):
            x = layers.Concatenate()([x, skips.pop()])
            x = ResidualBlock(width)(x)
        return x
    return apply
```

- El `DownBlock` aumenta el número de canales en la imagen usando un `ResidualBlock` de un ancho determinado...
- ...cada uno de los cuales se guarda en una lista (saltos) para que los `UpBlock` los utilicen más adelante.
- Una capa final de `AveragePooling2D` reduce la dimensionalidad de la imagen a la mitad.
- `UpBlock` comienza con una capa `UpSampling2D` que duplica el tamaño de la imagen.
- La salida de una capa `DownBlock` se pega a la salida actual usando una capa `Concatenate`.

- Se utiliza un `ResidualBlock` para reducir la cantidad de canales en la imagen a medida que pasa a través del `upBlock`.

Entrenando el modelo de difusión

¡Ahora tenemos todos los componentes instalados para entrenar nuestro modelo de difusión de eliminación de ruido! El ejemplo 8-10 crea, compila y ajusta el modelo de difusión.

Ejemplo 8-10. Código para entrenar el `DiffusionModel`

```
model = DiffusionModel()
model.compile(
    optimizer=optimizers.experimental.AdamW(learning_rate=1e-3,
    weight_decay=1e-4),
    loss=losses.mean_absolute_error,
)
model.normalizer.adapt(train)

model.fit(
    train,
    epochs=50,
)
```

1. Instancia el modelo.
2. Compila el modelo utilizando el optimizador AdamW (similar a Adam pero con caída de peso, que ayuda a estabilizar el proceso de entrenamiento) y la función de pérdida de error absoluto medio.
3. Calcula las estadísticas de normalización utilizando el conjunto de entrenamiento.

Ajusta el modelo en más de 50 épocas.

La curva de pérdida (error absoluto medio de ruido [MAE]) se muestra en la Figura 8-12.

Figura 8-12. La curva de pérdida de error absoluta media del ruido, por época.

Muestreando el modelo de difusión de eliminación de ruido

Para tomar muestras de imágenes de nuestro modelo entrenado, necesitamos aplicar el proceso de difusión inversa, es decir, debemos comenzar con ruido aleatorio y usar el modelo para deshacer gradualmente el ruido, hasta que nos quede una imagen reconocible de una flor.

Debemos tener en cuenta que nuestro modelo está entrenado para predecir la cantidad total de ruido que se ha agregado a una imagen ruidosa determinada del conjunto de entrenamiento, no solo el ruido que se agregó en el último paso del proceso de generación de ruido. Sin embargo, no queremos deshacer el ruido de una sola vez: ¡predecir una imagen a partir de ruido aleatorio puro en una sola toma claramente no va a funcionar! Preferiríamos imitar el proceso de avance y deshacer el ruido predicho gradualmente en muchos pasos pequeños, para permitir que el modelo se ajuste a sus propias predicciones.

Para lograr esto, podemos saltar de x_t a x_{t-1} en dos pasos: primero usando la predicción de ruido de nuestro modelo para calcular una estimación para la imagen original x_0 y luego volviendo a aplicar el ruido predicho a esta imagen, pero solo sobre $t - 1$ pasos de tiempo, para producir x_{t-1} . Esta idea se muestra en la Figura 8-13.

Figura 8-13. Un paso del proceso de muestreo para nuestro modelo de difusión.

Si repetimos este proceso en varios pasos, eventualmente volveremos a una estimación de x_0 que se ha guiado gradualmente a través de muchos pasos pequeños. De hecho, somos libres de elegir el número de pasos que damos y, lo que es más importante, no tiene por qué ser el mismo que el gran número de pasos del proceso de entrenamiento con ruido (es decir, 1000). Puede ser mucho más pequeño; en este ejemplo elegimos 20.

La siguiente ecuación (Song et al., 2020) explica este proceso matemáticamente:

$$(t)$$

$$x_{t-1} = \sqrt{\bar{\alpha}_{t-1}} \left(\frac{x_t - \sqrt{1 - \bar{\alpha}_t} \epsilon_\theta^{(t)}(x_t)}{\sqrt{\bar{\alpha}_t}} \right) + \sqrt{1 - \bar{\alpha}_{t-1} - \sigma_t^2} \cdot \epsilon_\theta^{(t)}(x_t) + \sigma_t \epsilon_t$$

donde:

$\left(\frac{x_t - \sqrt{1-\bar{\alpha}_t} \epsilon_\theta^{(t)}(x_t)}{\sqrt{\bar{\alpha}_t}} \right)$ es x_θ predicho

$\sqrt{1 - \bar{\alpha}_{t-1} - \sigma_t^2} \cdot \epsilon_\theta^{(t)}(x_t)$ es la dirección apuntando a x_t

y $\sigma_t \epsilon_t$ es ruido aleatorio

Analicemos esto. El primer término entre paréntesis en el lado derecho de la ecuación es la imagen estimada x_0 , calculada utilizando el ruido predicho por nuestra red $\epsilon_\theta^{(t)}$. Luego escalamos esto en $t - 1$ según la tasa de señal $\sqrt{\bar{\alpha}_{t-1}}$ y volvemos a aplicar el ruido predicho, pero esta vez escalado $t - 1$ por la tasa de ruido $\sqrt{1 - \bar{\alpha}_{t-1} - \sigma_t^2}$. También se añade ruido aleatorio gaussiano adicional $\sigma_t \epsilon_t$, con los factores σ_t que determinan qué tan aleatorio queremos que sea nuestro proceso de generación. El caso especial $\sigma_t = 0$ para todo t corresponde a un tipo de modelo conocido como *modelo implícito de difusión de eliminación de ruido* (DDIM), presentado por Song et al. en 2020. [9] Con DDIM, el proceso de generación es completamente determinista, es decir, la misma entrada de ruido aleatorio siempre dará la misma salida. Esto es deseable ya que entonces tenemos un mapeo bien definido entre las muestras del espacio latente y las salidas generadas en el espacio de píxeles.

En nuestro ejemplo, implementaremos un DDIM, haciendo así que nuestro proceso de generación sea determinista. El código para el proceso de muestreo DDIM (difusión inversa) se muestra en el Ejemplo 8-11.

Ejemplo 8-11. Muestreo del modelo de difusión

```
class DiffusionModel(models.Model):

    ...
    def reverse_diffusion(self, initial_noise, diffusion_steps):
        num_images = initial_noise.shape[0]
        step_size = 1.0 / diffusion_steps
        current_images = initial_noise
        for step in range(diffusion_steps):
            diffusion_times = tf.ones((num_images, 1, 1, 1)) - step * step_size
            noise_rates, signal_rates =
            self.diffusion_schedule(diffusion_times)
            pred_noises, pred_images = self.denoise(
                current_images, noise_rates, signal_rates, training=False
            )
```

```

        next_diffusion_times = diffusion_times - step_size
        next_noise_rates, next_signal_rates = self.diffusion_schedule(
            next_diffusion_times
        )
        current_images = (
            next_signal_rates * pred_images + next_noise_rates * 
pred_noises
        )
    return pred_images

```

1. Revise un número fijo de pasos (por ejemplo, 20).
2. Todos los tiempos de difusión están establecidos en 1 (es decir, al inicio del proceso de difusión inversa).
3. Las tasas de ruido y señal se calculan según el programa de difusión.
4. U-Net se utiliza para predecir el ruido, lo que nos permite calcular la estimación de la imagen sin ruido.
5. Los tiempos de difusión se reducen en un paso.
6. Se calculan las nuevas tasas de ruido y señal.
7. Las imágenes t-1 se calculan reaplicando el ruido predicho a la imagen predicha, de acuerdo con las tasas del programa de difusión t-1.
8. Después de 20 pasos, se devuelven las imágenes finales predichas x 0.

Análisis del modelo de difusión

Ahora veremos tres formas diferentes en las que podemos usar nuestro modelo entrenado: para generar nuevas imágenes, probar cómo el número de pasos de difusión inversa afecta la calidad e interpolar entre dos imágenes en el espacio latente.

Generando imágenes

Para producir muestras de nuestro modelo entrenado, simplemente podemos ejecutar el proceso de difusión inversa, asegurándonos de desnormalizar la salida al final (es decir, llevar los valores de píxeles nuevamente al rango [0, 1]). Podemos lograr esto usando el código del Ejemplo 8-12 dentro de la clase `DiffusionModel`.

Ejemplo 8-12. Generando imágenes usando el modelo de difusión

```

class DiffusionModel(models.Model):
    ...

```

```

def denormalize(self, images):
    images = self.normalizer.mean + images *
self.normalizer.variance**0.5
    return tf.clip_by_value(images, 0.0, 1.0)

def generate(self, num_images, diffusion_steps):
    initial_noise = tf.random.normal(shape=(num_images, 64, 64, 3))
    generated_images = self.reverse_diffusion(initial_noise,
diffusion_steps)
    generated_images = self.denormalize(generated_images)
    return generated_images

```

1. Generar algunos mapas de ruido iniciales.
2. Aplicar el proceso de difusión inversa.
3. Las imágenes generadas por la red tendrán media cero y varianza unitaria, por lo que debemos desnormalizarla volviendo a aplicar la media y la varianza calculadas a partir de los datos de entrenamiento.

En la Figura 8-14 podemos observar algunas muestras del modelo de difusión en diferentes épocas del proceso de entrenamiento.

Figura 8-14. Muestras del modelo de difusión en diferentes épocas del proceso de formación.

Ajustando el número de pasos de difusión

También podemos probar para ver cómo el ajuste del número de pasos de difusión en el proceso inverso afecta la calidad de la imagen. Intuitivamente, cuantos más pasos dé el proceso, mayor será la calidad de la generación de imágenes.

Podemos ver en la Figura 8-15 que la calidad de las generaciones mejora con el número de pasos de difusión. Con un salto de gigante desde el ruido inicial muestreado, el modelo sólo puede predecir una mancha de color nebulosa. Con más pasos, el modelo es capaz de perfeccionar y perfeccionar sus generaciones. Sin embargo, el tiempo necesario para generar las imágenes aumenta linealmente con el número de pasos de difusión, por lo que existe una compensación. Hay una mejora mínima entre 20 y 100 pasos de difusión, por lo que elegimos 20 como un compromiso razonable entre calidad y velocidad en este ejemplo.

Figura 8-15. La calidad de la imagen mejora con el número de pasos de difusión.

Interpolando entre imágenes

Por último, como hemos visto anteriormente con los codificadores automáticos variacionales, podemos interpolar entre puntos en el espacio latente gaussiano para realizar una transición suave entre imágenes en el espacio de píxeles. Aquí elegimos utilizar una forma de interpolación esférica que garantiza que la varianza permanezca constante mientras combinamos los dos mapas de ruido gaussianos. Específicamente, el mapa de ruido inicial en cada paso viene dado por $a \cdot \sin\left(\frac{\pi}{2}t\right) + b \cdot \cos\left(\frac{\pi}{2}t\right)$, donde t varía suavemente de 0 a 1 y a y b son los dos tensores de ruido gaussiano muestreados aleatoriamente entre los que deseamos interpolar.

Las imágenes resultantes se muestran en la Figura 8-16.

Figura 8-16. Interpolando entre imágenes usando el modelo de difusión de eliminación de ruido

Resumen

En este capítulo hemos explorado una de las áreas más interesantes y prometedoras del modelado generativo en los últimos tiempos: los modelos de difusión. En particular, implementamos las ideas de un artículo clave sobre modelos de difusión generativa (Ho et al., 2020) que introdujo el modelo probabilístico de difusión por eliminación de ruido (DDPM) original. Luego ampliamos esto con las ideas de Modelo implícito de difusión sin ruido (DDIM) para que el proceso de generación sea totalmente determinista.

Hemos visto cómo los modelos de difusión se forman a partir de un proceso de difusión directa y un proceso de difusión inversa. El proceso de difusión directa agrega ruido a los datos de entrenamiento a través de una serie de pequeños pasos, mientras que el proceso de difusión inversa consiste en un modelo que intenta predecir el ruido agregado.

Utilizamos un truco de reparametrización para calcular las imágenes con ruido en cualquier paso del proceso de avance sin tener que pasar por múltiples pasos con

ruido. Hemos visto cómo el programa elegido de parámetros utilizados para agregar ruido a los datos juega un papel importante en el éxito general del modelo.

El proceso de difusión inversa está parametrizado por una U-Net que intenta predecir el ruido en cada paso de tiempo, dada la imagen con ruido y la tasa de ruido en ese paso. Una U-Net consta de DownBlocks que aumentan la cantidad de canales mientras reducen el tamaño de la imagen y UpBlocks que disminuyen la cantidad de canales mientras aumentan el tamaño. La tasa de ruido se codifica mediante incrustación sinusoidal.

El muestreo del modelo de difusión se realiza en una serie de pasos. UNet se utiliza para predecir el ruido añadido a una imagen con ruido determinada, que luego se utiliza para calcular una estimación de la imagen original. Luego, el ruido previsto se vuelve a aplicar utilizando una tasa de ruido menor. Este proceso se repite en una serie de pasos (que pueden ser significativamente más pequeños que el número de pasos utilizados durante el entrenamiento), comenzando desde un punto aleatorio muestreado de una distribución de ruido gaussiana estándar, para obtener la generación final.

Vimos cómo aumentar el número de pasos de difusión en el proceso inverso mejora la calidad de generación de imágenes, a expensas de la velocidad. También realizamos aritmética del espacio latente para interpolar entre dos imágenes.

1 Jascha Sohl-Dickstein et al., "Aprendizaje profundo no supervisado utilizando termodinámica de desequilibrio", 12 de marzo de 2015, <https://arxiv.org/abs/1503.03585>

2 Yang Song y Stefano Ermon, "Modelado generativo mediante la estimación de gradientes de la distribución de datos", 12 de julio de 2019, <https://arxiv.org/abs/1907.05600>.

3 Yang Song y Stefano Ermon, "Técnicas mejoradas para entrenar modelos generativos basados en puntuaciones", 16 de junio de 2020, <https://arxiv.org/abs/2006.09011>.

4 Jonathon Ho et al., "Denoising Diffusion Probabilistic Models", 19 de junio de 2020, <https://arxiv.org/abs/2006.11239>.

5 Alex Nichol y Prafulla Dhariwal, "Modelos probabilísticos de difusión de eliminación de ruido mejorados", febrero 18 de septiembre de 2021,

<https://arxiv.org/abs/2102.09672>.

6 Ashish Vaswani et al., “La atención es todo lo que necesitas”, 12 de junio de 2017, <https://arxiv.org/abs/1706.03762>.

7 Ben Mildenhall et al., "NeRF: Representación de escenas como campos de radiación neuronal para la síntesis de vistas", 1 de marzo de 2020, <https://arxiv.org/abs/2003.08934>.

8 Kaiming He et al., “Aprendizaje residual profundo para el reconocimiento de imágenes”, 10 de diciembre de 2015, <https://arxiv.org/abs/1512.03385>.

9 Jiaming Song et al., “Denoising Diffusion Implicit Models”, 6 de octubre de 2020, <https://arxiv.org/abs/2010.02502>

Parte III. Aplicaciones

En la Parte III, exploraremos algunas de las aplicaciones clave de las técnicas de modelado generativo que hemos visto hasta ahora, en imágenes, texto, música y juegos. También veremos cómo se pueden atravesar estos dominios utilizando modelos multimodales de última generación.

En el capítulo 9 centraremos nuestra atención en Transformers, una arquitectura de última generación que impulsa la mayoría de los modelos de generación de texto modernos. En particular, exploraremos el funcionamiento interno de GPT y crearemos nuestra propia versión utilizando Keras, y veremos cómo constituye la base de herramientas como ChatGPT.

En el Capítulo 10 veremos algunas de las más importantes arquitecturas GAN que han influido en la generación de imágenes, incluidas ProGAN, StyleGAN, StyleGAN2, SAGAN, BigGAN, VQ-GAN y ViT VQ-GAN. Exploraremos las contribuciones clave de cada uno y buscaremos comprender cómo ha evolucionado la técnica con el tiempo.

El capítulo 11 analiza la generación musical, que presenta desafíos adicionales, como modelar el tono y el ritmo musical. Veremos que muchas de las técnicas que funcionan para la generación de texto (como Transformers) también se pueden aplicar en este dominio, pero también exploraremos una arquitectura de aprendizaje profundo conocida como MuseGAN que aplica un enfoque basado en GAN para generar música. .

El Capítulo 12 muestra cómo se pueden utilizar los modelos generativos en otros dominios del aprendizaje automático, como aprendizaje reforzado. Nos centraremos en los "modelos mundiales", que muestran cómo se puede utilizar un modelo generativo como entorno en el que se entrena el agente, permitiéndole entrenar dentro de una versión onírica alucinada del entorno en lugar de hacerlo en la realidad.

En el Capítulo 13 exploraremos modelos multimodales de última generación que abarcan dominios como imágenes y texto. Esto incluye modelos de texto a imagen como DALL.E 2,Imagen y Difusión Estable, así como modelos de lenguaje visual como Flamingo.

Finalmente, el Capítulo 14 resume el recorrido de la IA generativa hasta el momento, el panorama actual de la IA generativa y hacia dónde nos dirigimos en el futuro. Exploraremos cómo la IA generativa puede cambiar la forma en que vivimos y trabajamos, además de considerar si tiene el potencial de desbloquear formas más profundas de inteligencia artificial en los próximos años.

Capítulo 9. Transformadores

METAS DEL CAPÍTULO

En este capítulo podrás:

- Conocer los orígenes de GPT, un potente modelo Transformador decodificador para generación de texto.
- Aprender conceptualmente cómo un mecanismo de atención imita nuestra forma de dar más importancia a algunas palabras de una oración que a otras.
- Profundizar en cómo funciona el mecanismo de atención desde los primeros principios, incluido cómo se crean y manipulan las consultas, las claves y los valores.
- Ver la importancia del enmascaramiento causal para las tareas de generación de texto.
- Comprender cómo se pueden agrupar las cabezas de atención en una capa de atención de varias cabezas.
- Ver cómo las capas de atención de múltiples cabezales forman parte de un bloque Transformer que también incluye normalización de capas y conexiones de omisión.
- Crear codificaciones posicionales que capturen la posición de cada token, así como la incrustación del token de palabra.
- Crear un modelo GPT en Keras para generar el texto contenido en las reseñas de vinos.
- Analizar el resultado del modelo GPT, incluida la interrogación de las puntuaciones de atención para inspeccionar hacia dónde mira el modelo.
- Conocer los diferentes tipos de Transformers, incluidos ejemplos de los tipos de tareas que puede realizar cada uno y descripciones de las implementaciones más modernas y famosas.
- Comprender cómo funcionan las arquitecturas de codificador-decodificador, como el modelo T5 de Google.
- Explorar el proceso de capacitación detrás de ChatGPT de OpenAI.

Vimos en el Capítulo 5 cómo podemos construir modelos generativos sobre datos de texto utilizando redes neuronales recurrentes (RNN), como LSTM y GRU. Estos modelos autorregresivos procesan datos secuenciales, un token a la vez, actualizando constantemente un vector oculto que captura la representación latente actual de la entrada. El RNN se puede diseñar para predecir la siguiente palabra en una secuencia aplicando una capa Dense y activación softmax sobre el vector oculto. Esta se consideraba la forma más sofisticada de producir texto generativamente hasta 2017, cuando un artículo cambió para siempre el panorama de la generación de texto.

Introducción

El artículo de Google Brain, titulado “La atención es todo lo que necesitas”,¹ es famoso por popularizar el concepto de atención, un mecanismo que ahora impulsa la mayoría de los modelos de generación de texto más modernos.

Los autores muestran cómo es posible crear poderosas redes neuronales llamadas Transformadores para modelado secuencial que no requieren arquitecturas complejas recurrentes o convolucionales sino que solo dependen de mecanismos de atención. Este enfoque supera una desventaja clave del enfoque RNN, que es que es difícil paralelizar, ya que debe procesar secuencias de un token a la vez. Los transformadores son altamente paralelizables, lo que les permite entrenarse en conjuntos de datos masivos.

En este capítulo, profundizaremos en cómo los modelos modernos de generación de texto utilizan la arquitectura Transformer para alcanzar un rendimiento de vanguardia en los desafíos de generación de texto. En particular, exploraremos un tipo de modelo autorregresivo conocido como transformador generativo preentrenado (GPT), que impulsa el modelo GPT-4 de OpenAI, ampliamente considerado como el estado actual del arte para la generación de texto.

GPT

OpenAI presentó GPT en junio de 2018, en el artículo “Improving Language Understanding by Generative Pre-Training”,² casi exactamente

un año después de la aparición del artículo original del transformador.

En este artículo, los autores muestran cómo se puede entrenar una arquitectura Transformer con una gran cantidad de datos de texto para predecir la siguiente palabra en una secuencia y luego ajustarla a tareas posteriores específicas.

El proceso de preentrenamiento de GPT implica entrenar el modelo en un gran corpus de texto llamado BookCorpus (4,5 GB de texto de 7.000 libros inéditos de diferentes géneros).

Durante el entrenamiento previo, el modelo se entrena para predecir la siguiente palabra en una secuencia dadas las palabras anteriores. Este proceso se conoce como modelado del lenguaje y se utiliza para enseñar al modelo a comprender la estructura y los patrones del lenguaje natural.

Después del entrenamiento previo, el modelo GPT se puede ajustar para una tarea específica proporcionándole un conjunto de datos más pequeño y específico para la tarea. El ajuste fino implica ajustar los parámetros del modelo para que se ajuste mejor a la tarea en cuestión. Por ejemplo, el modelo se puede ajustar para tareas como clasificación, puntuación de similitud o respuesta a preguntas.

Desde entonces, OpenAI ha mejorado y ampliado la arquitectura GPT con el lanzamiento de modelos posteriores como GPT-2, GPT-3, GPT-3.5 y GPT-4. Estos modelos están entrenados en conjuntos de datos más grandes y tienen mayores capacidades, por lo que pueden generar texto más complejo y coherente. Los modelos GPT han sido ampliamente adoptados por investigadores y profesionales de la industria y han contribuido a avances significativos en las tareas de procesamiento del lenguaje natural.

En este capítulo, construiremos nuestra propia variación del modelo GPT original, entrenado con menos datos, pero aún utilizando los mismos componentes y principios subyacentes.

EJECUTANDO EL CÓDIGO PARA ESTE EJEMPLO

El código para este ejemplo se puede encontrar en el cuaderno de Jupyter ubicado en notebooks/09_transformer/01_gpt/gpt.ipynb en el repositorio de libros.

El código está adaptado del excelente tutorial GPT creado por Apoorv Nandan disponible en el sitio web de Keras.

El conjunto de datos de reseñas de vinos

Usaremos el conjunto de datos de Wine Reviews que está disponible a través de Kaggle. Se trata de un conjunto de más de 130.000 reseñas de vinos, acompañadas de metadatos como descripción y precio.

Puede descargar el conjunto de datos ejecutando el script de descarga del conjunto de datos de Kaggle en el repositorio de libros, como se muestra en el Ejemplo 9-1. Esto guardará las reseñas de vinos y los metadatos que las acompañan localmente en la carpeta /data.

Ejemplo 9-1. Descarga del conjunto de datos de Wine Reviews

```
bash scripts/download_kaggle_data.sh zynicide vino-reseñas
```

Los pasos de preparación de datos son idénticos a los pasos utilizados en el Capítulo 5 para preparar datos para ingresarlos en un LSTM, por lo que no los repetiremos en detalle aquí. Los pasos, como se muestra en la Figura 9-1, son los siguientes:

1. Cargue los datos y cree una lista de descripciones de cadenas de texto de cada vino.
1. Rellene la puntuación con espacios, de modo que cada signo de puntuación se trate como una palabra independiente.
1. Pase las cadenas a través de una capa TextVectorization que tokeniza los datos y rellena/recorta cada cadena a una longitud fija.
1. Cree un conjunto de entrenamiento donde las entradas sean las cadenas de texto tokenizadas y las salidas a predecir sean las mismas cadenas desplazadas en un token.

Figura 9-1. Procesamiento de datos para el Transformador Atención

El primer paso para comprender cómo funciona GPT es comprender cómo funciona el mecanismo de atención. Este mecanismo es lo que hace que la arquitectura Transformer sea única y distinta de los enfoques recurrentes del modelado del lenguaje. Cuando hayamos desarrollado una comprensión sólida de la atención, veremos cómo se utiliza dentro de las arquitecturas Transformer como GPT.

Cuando escribes, la elección que haces para la siguiente palabra de la oración está influenciada por otras palabras que ya has escrito. Por ejemplo, supongamos que comienza una oración de la siguiente manera:

El elefante rosa intentó subir al auto pero fue demasiado claramente, la siguiente palabra debería ser algo sinónimo de grande. Cómo sabemos esto?

Algunas otras palabras de la oración son importantes para ayudarnos a tomar nuestra decisión.

Por ejemplo, el hecho de que sea un elefante, en lugar de un perezoso, significa que preferimos lo grande a lo lento. Si fuera una piscina, en lugar de un coche, elegiríamos miedo como una posible alternativa a las grandes. Por último, la acción de subirse al coche implica que el tamaño es el problema: si el elefante intentara aplastar el coche, podríamos elegir rápido como última palabra, refiriéndose ahora al coche.

Otras palabras en la oración no son importantes en absoluto. Por ejemplo, el hecho de que el elefante sea rosa no influye en nuestra elección de la palabra final. Del mismo modo, las palabras menores de la oración (the, but, it, etc.) dan forma gramatical a la oración, pero aquí no son importantes para determinar el adjetivo requerido.

En otras palabras, prestamos atención a ciertas palabras de la oración e ignoramos en gran medida otras. ¿No sería fantástico si nuestro modelo pudiera hacer lo mismo?

Un mecanismo de atención (también conocido como cabeza de atención) en un Transformer está diseñado para hacer exactamente esto. Es capaz de

decidir de qué parte de la entrada desea extraer información, para poder extraer información útil de manera eficiente sin verse empañado por detalles irrelevantes.

Esto lo hace altamente adaptable a una variedad de circunstancias, ya que puede decidir dónde quiere buscar información en el momento de la inferencia.

Por el contrario, una capa recurrente intenta construir un estado oculto genérico que captura una representación general de la entrada en cada paso de tiempo. Una debilidad de este enfoque es que muchas de las palabras que ya se han incorporado al vector oculto no serán directamente relevantes para la tarea inmediata en cuestión (por ejemplo, predecir la siguiente palabra), como acabamos de ver. Los jefes de atención no sufren este problema, porque pueden elegir cómo combinar información de palabras cercanas, dependiendo del contexto.

Consultas, claves y valores

Entonces, ¿cómo decide una cabeza de atención dónde quiere buscar información? Antes de entrar en detalles, exploremos cómo funciona a un nivel alto, usando nuestro ejemplo del elefante rosa.

Imaginemos que también queremos predecir lo que sigue a la palabra. Para ayudar con esta tarea, otras palabras anteriores intervienen con sus opiniones, pero sus contribuciones se ponderan según la confianza que tengan en su propia experiencia para predecir las palabras que siguen también. Por ejemplo, la palabra elefante podría contribuir con confianza a que es más probable que sea una palabra relacionada con el tamaño o el volumen, mientras que la palabra era no tiene mucho que ofrecer para reducir las posibilidades.

En otras palabras, podemos pensar en una cabeza de atención como una especie de sistema de recuperación de información, donde una consulta (“¿Qué palabra sigue también?”) se convierte en un almacén de clave/valor (otras palabras en la oración) y el resultado resultante es una suma de los valores, ponderada por la resonancia entre la consulta y cada clave.

Ahora analizaremos el proceso en detalle (Figura 9-2), nuevamente con referencia a nuestra oración del elefante rosa.

Figura 9-2. La mecánica de una cabeza de atención.

La consulta (Q) se puede considerar como una representación de la tarea actual en cuestión (por ejemplo, "¿Qué palabra sigue también?"). En este ejemplo, también se deriva de la incrustación de la palabra, pasándola a través de una matriz de pesos WQ para cambiar la dimensionalidad del vector de d a dk .

Los vectores clave (K) son representaciones de cada palabra de la oración; puede considerarlos como descripciones de los tipos de tareas de predicción con las que cada palabra puede ayudar. Se derivan de manera similar a la consulta, pasando cada incorporación a través de una matriz de pesos WK para cambiar la dimensionalidad de cada vector de d a dk . Observe que las claves y la consulta tienen la misma longitud (dk).

Dentro del cabezal de atención, cada clave se compara con la consulta utilizando un producto escalar entre cada par de vectores (QKT). Es por eso que las claves y la consulta deben tener la misma longitud. Cuanto mayor sea este número para un par clave/consulta particular, más resuena la clave con la consulta, por lo que se le permite hacer una mayor contribución a la salida del cabezal de atención. El vector resultante se escala por \sqrt{dk} para mantener la varianza de la suma del vector estable (aproximadamente igual a 1) y se aplica un softmax para garantizar que las contribuciones sumen 1. Este es un vector de pesos de atención.

Los vectores de valores (V) también son representaciones de las palabras de la oración; puedes considerarlos como las contribuciones no ponderadas de cada palabra. Se obtienen pasando cada incorporación a través de una matriz de pesos WV para cambiar la dimensionalidad de cada vector de d a dv . Tenga en cuenta que los vectores de valor no necesariamente tienen que tener la misma longitud que las claves y la consulta (pero a menudo la tienen, por simplicidad).

Los vectores de valor se multiplican por los pesos de atención para dar la atención a un determinado Q , K y V , como se muestra en la ecuación 9-1.

Ecuación 9-1. ecuación de atención

qk

$$\text{Atención } (Q, K, V) = \text{suave } \text{tmax } (\sigma V \sqrt{dk})$$

Para obtener el vector de salida final del cabezal de atención, la atención se suma para obtener un vector de longitud dv. Este vector de contexto captura una opinión combinada de las palabras de la oración sobre la tarea de predecir qué palabra sigue también.

Atención multicabezal

¡No hay razón para detenerse en un solo foco de atención! En Keras, podemos construir una capa MultiHeadAttention que concatena la salida de múltiples cabezas de atención, permitiendo que cada una aprenda un mecanismo de atención distinto para que la capa en su conjunto pueda aprender relaciones más complejas.

Las salidas concatenadas se pasan a través de una matriz de pesos final WO para proyectar el vector en la dimensión de salida deseada, que en nuestro caso es la misma que la dimensión de entrada de la consulta (de), de modo que las capas se puedan apilar secuencialmente encima de entre sí.

La Figura 9-3 muestra cómo se construye la salida de una capa MultiHeadAttention. En Keras podemos simplemente escribir la línea que se muestra en el Ejemplo 9-2 para crear dicha capa.

Ejemplo 9-2. Creando una capa MultiHeadAttention en capas de Keras.
`MultiHeadAttention (num_heads = 4, key_dim = 128, value_dim = 64, output_shape = 256)`

Esta capa de atención de múltiples cabezas tiene cuatro cabezas.

Las claves (y la consulta) son vectores de longitud 128.

Los valores (y por lo tanto también la salida de cada cabeza) son vectores de longitud 64.

El vector de salida tiene una longitud de 256.

Figura 9-3. Una capa de atención multicabezal con cuatro cabezas.

Enmascaramiento causal

Hasta ahora, hemos asumido que la entrada de la consulta a nuestro cabezal de atención es un vector único.

Sin embargo, para lograr eficiencia durante el entrenamiento, lo ideal sería que la capa de atención pudiera operar con cada palabra de la entrada a la vez, prediciendo para cada una cuál será la palabra siguiente. En otras palabras, queremos que nuestro modelo GPT pueda manejar un grupo de vectores de consulta en paralelo (es decir, una matriz).

Se podría pensar que podemos simplemente agrupar los vectores en una matriz y dejar que el álgebra lineal se encargue del resto. Esto es cierto, pero necesitamos un paso adicional: debemos aplicar una máscara al producto punto clave/consulta para evitar que se filtre información de palabras futuras. Esto se conoce como enmascaramiento causal y se muestra en la Figura 9-4.

Figura 9-4. Cálculo matricial de las puntuaciones de atención para un lote de consultas de entrada, utilizando una máscara de atención causal para ocultar claves que no están disponibles para la consulta (porque aparecen más adelante en la oración)

Sin esta máscara, nuestro modelo GPT sería capaz de adivinar perfectamente la siguiente palabra de la oración, ¡porque estaría usando la clave de la palabra misma como característica! El código para crear una máscara causal se muestra en el Ejemplo 9-3, y la matriz numpy resultante (transpuesta para que coincida con el diagrama) se muestra en la Figura 9-5.

```
Ejemplo 9-3. La función de máscara causal
def causal_attention_mask(batch_size, n_dest, n_src, dtype):
    i = tf.range(n_dest)[:, None]
    j = tf.range(n_src)
    m = i >= j - n_src + n_dest
    mask = tf.cast(m, dtype)
    máscara = tf.reshape(máscara, [1, n_dest, n_src])
    mult = tf.concat([tf.expand_dims(batch_size, -1), tf.constant([1, 1],
```

```
dtype=tf.int32)], 0 return tf.tile(mask, mult)
np.transpose(causal_attention_mask(1, 10, 10, dtype = tf.int32)[0])
```

Figura 9-5. La máscara causal como una matriz numpy: 1 significa desenmascarado y 0 significa enmascarado

CONSEJO

El enmascaramiento causal solo se requiere en decodificadores Transformers como GPT, donde la tarea es generar tokens secuencialmente a partir de tokens anteriores. Por lo tanto, es esencial enmascarar tokens futuros durante el entrenamiento.

Otras versiones de Transformer (por ejemplo, el codificador Transformers) no necesitan enmascaramiento causal porque no están capacitados para predecir el siguiente token. Por ejemplo, BERT de Google predice palabras enmascaradas dentro de una oración determinada, por lo que puede usar el contexto tanto antes como después de la palabra en cuestión.³

Exploraremos los diferentes tipos de Transformers con más detalle al final del capítulo.

Con esto concluye nuestra explicación del mecanismo de atención multicabezal que está presente en todos los Transformadores. Es notable que los parámetros aprendibles de una capa tan influyente consistan en nada más que tres matrices de pesos densamente conectadas para cada cabeza de atención (WQ , WK , WV) y una matriz de pesos adicional para remodelar la salida (WO). ¡No hay convoluciones ni mecanismos recurrentes en absoluto en una capa de atención de múltiples cabezales!

A continuación, daremos un paso atrás y veremos cómo la capa de atención de múltiples cabezales forma solo una parte de un componente más grande conocido como bloque Transformer.

El bloque transformador

Un bloque Transformer es un componente único dentro de un Transformer que aplica algunas conexiones de salto, capas de avance (densas) y

normalización alrededor de la capa de atención de múltiples cabezales. En la Figura 9-6 se muestra un diagrama de un bloque transformador.

Figura 9-6. Un bloque transformador

En primer lugar, observe cómo la consulta pasa por la capa de atención de múltiples cabezales para agregarse a la salida; esta es una conexión de omisión y es común en las arquitecturas modernas de aprendizaje profundo. Significa que podemos construir redes neuronales muy profundas que no sufren tanto el problema del gradiente evanescente, porque la conexión de salto proporciona una autopista sin gradiente que permite a la red transferir información hacia adelante sin interrupciones.

En segundo lugar, la normalización de capas se utiliza en el bloque Transformer para proporcionar estabilidad al proceso de entrenamiento. Ya hemos visto la capa de normalización por lotes en acción a lo largo de este libro, donde la salida de cada canal se normaliza para tener una media de 0 y una desviación estándar de 1. Las estadísticas de normalización se calculan en las dimensiones espacial y por lotes.

Por el contrario, la normalización de capas en un bloque Transformer normaliza cada posición de cada secuencia en el lote calculando las estadísticas de normalización entre los canales. Es todo lo contrario de la normalización por lotes, en términos de cómo las estadísticas de normalización se calculan. En la Figura 9-7 se muestra un diagrama que muestra la diferencia entre la normalización por lotes y la normalización por capas.

Figura 9-7. Normalización de capas versus normalización por lotes: las estadísticas de normalización se calculan en las celdas azules (fuente: Sheng et al., 2020)⁴

NORMALIZACIÓN DE CAPAS VERSUS NORMALIZACIÓN DE LOTES

La normalización de capas se usó en el documento GPT original y se usa comúnmente para tareas basadas en texto para evitar la creación de

dependencias de normalización entre secuencias del lote. Sin embargo, trabajos recientes como el de Shen et al. cuestionan esta suposición, mostrando que con algunos ajustes aún se puede usar una forma de normalización por lotes dentro de Transformers, superando la normalización de capas más tradicional.

Por último, en el bloque Transformer se incluye un conjunto de capas de avance (es decir, densamente conectadas) para permitir que el componente extraiga características de nivel superior a medida que profundizamos en la red.

En el ejemplo 9-4 se muestra una implementación de Keras de un bloque Transformer.

Ejemplo 9-4. Una capa TransformerBlock en Keras

```
class TransformerBlock(layers.Layer):
    def __init__(self, num_heads, key_dim, embed_dim, ff_dim,
                 dropout_rate=0.1):
        super(TransformerBlock, self).__init__()
        self.num_heads = num_heads
        self.key_dim = key_dim
        self.embed_dim = embed_dim
        self.ff_dim = ff_dim
        self.dropout_rate = dropout_rate
        self.attn = layers.MultiHeadAttention( num_heads, key_dim,
                                               output_shape = embed_dim)
        self.dropout_1 = layers.Dropout(self.dropout_rate)
        self.ln_1 = layers.LayerNormalization(epsilon=1e-6)
        self.ffn_1 = layers.Dense(self.ff_dim, activation="relu")
        self.ffn_2 = layers.Dense(self.embed_dim)
        self.dropout_2 = layers.Dropout(self.dropout_rate)
        self.ln_2 = layers.LayerNormalization(epsilon=1e-6)

    def call(self, entradas):
        input_shape = tf.shape(inputs)
        lote_size = input_shape[0]
        seq_len = input_shape[1]
        causal_mask = causal_attention_mask( lote_size, seq_len,
                                             seq_len, tf.bool)
        atención_salida, atención_scores = self.attn( entradas,
                                                       entradas, atención_mask=causal_mask,
                                                       return_attention_scores=True)
        atención_salida = self.dropout_1(atención_salida)
```

```
salida1 = self.ln_1(entradas + atención_salida)
ffn_1 = self.ffn_1(salida1)
ffn_2 = self.ffn_2(ffn_1)
ffn_salida = self.dropout_2(ffn_2)
return (self.ln_2(salida1 + ffn_salida),
atención_puntuaciones)
```

Las subcapas que componen la capa TransformerBlock se definen dentro de la función de inicialización.

La máscara causal se crea para ocultar claves futuras de la consulta.

Se crea la capa de atención de múltiples cabezales, con las máscaras de atención especificadas.

La primera capa de adición y normalización.

Las capas de avance.

La segunda capa de adición y normalización.

Codificación posicional

Hay un último paso que cubrir antes de que podamos juntar todo para entrenar nuestro modelo GPT. Es posible que hayas notado que en la capa de atención de múltiples cabezales, no hay nada que se preocupe por el orden de las claves. El producto escalar entre cada clave y la consulta se calcula en paralelo, no de forma secuencial, como en una red neuronal recurrente. Esto es una fortaleza (debido a las ganancias en eficiencia de paralelización) pero también un problema, porque claramente necesitamos que la capa de atención pueda predecir diferentes resultados para las dos oraciones siguientes:

El perro miró al niño y...(¿ladró?)

El niño miró al perro y...(¿sonrió?)

Para resolver este problema, utilizamos una técnica llamada codificación posicional al crear las entradas al bloque Transformador inicial. En lugar de codificar únicamente cada token mediante una incrustación de token,

también codificamos la posición del token mediante una incrustación de posición.

La incrustación del token se crea utilizando una capa de incrustación estándar para convertir cada token en un vector aprendido. Podemos crear la incrustación posicional de la misma manera, usando una capa de incrustación estándar para convertir cada posición entera en un vector aprendido.

CONSEJO

Mientras que GPT usa una capa de Incrustación para incrustar la posición, el documento original de Transformer usaba funciones trigonométricas; cubriremos esta alternativa en el Capítulo 11, cuando exploremos la generación de música.

Para construir la codificación conjunta token-posición, la incrustación del token se agrega a la incrustación posicional, como se muestra en la Figura 9-8. De esta forma, el significado y la posición de cada palabra en la secuencia quedan capturados en un único vector.

Figura 9-8. Las incrustaciones de tokens se agregan a las incrustaciones posicionales para proporcionar la codificación de posición del token.

El código que define nuestra capa TokenAndPositionEmbedding se muestra en el Ejemplo 9-5.

Ejemplo 9-5. La capa TokenAndPositionEmbedding

```
class TokenAndPositionEmbedding(layers.Layer):
    def __init__(self, maxlen, vocab_size, embed_dim):
        super(TokenAndPositionEmbedding, self).__init__()
        self maxlen = maxlen
        self.vocab_size = vocab_size
        self.embed_dim = embed_dim
        self.token_emb = layers.Embedding(entrada_dim=vocab_size,
salida_dim=embed_dim)
        self.pos_emb = layers.Embedding(input_dim=maxlen,
salida_dim=embed_dim)
```

```

def llamada(self, x):

    maxlen = tf.shape(x)[-1]
    posiciones = tf.range(start=0, limit=maxlen, delta=1)
    posiciones = self.pos_emb(posiciones)
    x = self.token_emb(x)
    return x + posiciones

```

Los tokens se incrustan mediante una capa de incrustación.

Las posiciones de los tokens también se incrustan mediante una capa de incrustación.

La salida de la capa es la suma del token y las incrustaciones de posición.

Formación GPT

¡Ahora estamos listos para construir y entrenar nuestro modelo GPT! Para armar todo, necesitamos pasar nuestro texto de entrada a través del token y colocar la capa de incrustación, luego a través de nuestro Bloque transformador. El resultado final de la red es una capa Dense simple con activación softmax sobre la cantidad de palabras del vocabulario.

CONSEJO

Para simplificar, usaremos solo un bloque Transformer, en lugar de los 12 del documento.

La arquitectura general se muestra en la Figura 9-9 y el código equivalente se proporciona en el Ejemplo 9-6.

Figura 9-9. La arquitectura del modelo GPT simplificada

Ejemplo 9-6. Un modelo GPT en Keras

```

MAX_LEN = 80
VOCAB_SIZE = 10000
EMBEDDING_DIM = 256
N_HEADS = 2

```

```
KEY_DIM = 256
FEED_FORWARD_DIM = 256

inputs = layers.Input(shape = (None), dtype = tf.int32)
x = TokenAndPositionEmbedding (MAX_LEN, VOCAB_SIZE,
EMBEDDING_DIM) (entradas)
x, atención_scores = TransformerBlock (N_HEADS, KEY_DIM,
EMBEDDING_DIM, FEED_FORWARD_DIM )(x)
outputs = layers.Dense(VOCAB_SIZE, activación = 'softmax')(x)
gpt = models.Model(entradas=entradas, salidas=[salidas,
atención])
gpt.compile( "adam", pérdida=
[pérdida.SparseCategoricalCrossentropy(), Ninguna])
gpt.fit(train_ds, épocas=5)
```

La entrada se rellena (con ceros).

El texto se codifica mediante una capa TokenAndPositionEmbedding.

La codificación se pasa a través de un TransformerBlock.

La salida transformada pasa a través de una capa Dense con activación softmax para predecir una distribución en la palabra siguiente.

El modelo toma una secuencia de tokens de palabras como entrada y genera la distribución de palabras posterior prevista. La salida del bloque Transformer también se devuelve para que podamos inspeccionar cómo dirige su atención el modelo.

El modelo se compila con pérdida SparseCategoricalCrossentropy sobre la distribución de palabras prevista.

Análisis de GPT

Ahora que hemos compilado y entrenado nuestro modelo GPT, podemos comenzar a usarlo para generar largas cadenas de texto. También podemos interrogar los pesos de atención que se generan desde TransformerBlock, para comprender dónde busca información el Transformer en diferentes puntos del proceso de generación.

Generando texto

Podemos generar texto nuevo aplicando el siguiente proceso:

1. Alimente la red con una secuencia existente de palabras y pídale que prediga la siguiente palabra.
1. Agregue esta palabra a la secuencia existente y repita.

La red generará un conjunto de probabilidades para cada palabra de la que podamos tomar muestras, de modo que podamos hacer que la generación de texto sea estocástica, en lugar de determinista.

Usaremos la misma clase `TextGenerator` introducida en el Capítulo 5 para la generación de texto LSTM, incluido el parámetro de temperatura que especifica qué tan determinista nos gustaría que fuera el proceso de muestreo. Echemos un vistazo a esto en acción, con dos valores de temperatura diferentes (Figura 9-10).

Figura 9-10. Salidas generadas a temperatura = 1,0 y temperatura = 0,5.

Hay algunas cosas a tener en cuenta sobre estos dos pasajes. En primer lugar, ambos son estilísticamente similares a una reseña de vinos del conjunto de formación original. Ambos comienzan con la región y el tipo de vino, y el tipo de vino se mantiene constante a lo largo del pasaje (por ejemplo, no cambia de color a la mitad). Como vimos en el Capítulo 5, el texto generado con temperatura 1,0 es más aventurero y, por tanto, menos preciso que el ejemplo con temperatura 0,5. Por lo tanto, generar múltiples muestras con temperatura 1.0 generará más variedad, ya que el modelo toma muestras de una distribución de probabilidad con mayor varianza.

Ver las puntuaciones de atención

También podemos pedirle al modelo que nos diga cuánta atención se pone en cada palabra al decidir la siguiente palabra de la oración. `TransformerBlock` genera los pesos de atención para cada encabezado, que son una distribución softmax sobre las palabras anteriores de la oración.

Para demostrar esto, la Figura 9-11 muestra las cinco fichas principales con las probabilidades más altas para tres indicaciones de entrada diferentes, así

como la atención promedio en ambas cabezas, frente a cada palabra anterior. Las palabras anteriores están coloreadas según su puntuación de atención, promediada entre las dos cabezas de atención. El azul más oscuro indica que se está prestando más atención a la palabra.

Figura 9-11. Distribución de probabilidades de palabras siguiendo varias secuencias.

En el primer ejemplo, el modelo presta atención de cerca al país (alemania) para decidir la palabra que se relaciona con la región. ¡Esto tiene sentido! Para elegir una región, es necesario tomar mucha información de las palabras relacionadas con el país para garantizar que coincidan. No es necesario prestar tanta atención a los dos primeros tokens (reseña del vino) porque no contienen ninguna información útil sobre la región.

En el segundo ejemplo, es necesario hacer referencia a la uva (riesling), por lo que presta atención a la primera vez que se mencionó. Puede extraer esta información prestando atención directamente a la palabra, sin importar qué tan atrás esté en la oración (dentro del límite superior de 80 palabras). Tenga en cuenta que esto es muy diferente de una red neuronal recurrente, que se basa en un estado oculto para mantener toda la información interesante a lo largo de la secuencia para que pueda utilizarse si es necesario; un enfoque mucho menos eficiente.

La secuencia final muestra un ejemplo de cómo nuestro modelo GPT puede elegir un adjetivo apropiado basándose en una combinación de información. Aquí la atención vuelve a centrarse en la uva (riesling), pero también en el hecho de que contiene azúcar residual. Como el Riesling suele ser un vino dulce y el azúcar ya se mencionan, tiene sentido que se describa como ligeramente dulce en lugar de ligeramente terroso, por ejemplo.

Es increíblemente informativo poder interrogar a la red de esta manera, comprender exactamente de dónde extrae información para poder tomar decisiones precisas sobre cada palabra posterior. Recomiendo encarecidamente jugar con las indicaciones de entrada para ver si puede lograr que el modelo preste atención a las palabras que se encuentran muy atrás en la oración, para convencerse del poder de los modelos basados en la atención sobre los modelos recurrentes más tradicionales.

Otros transformadores

Nuestro modelo GPT es un transformador decodificador: genera una cadena de texto, un token a la vez, y utiliza enmascaramiento causal para prestar atención únicamente a las palabras anteriores en la cadena de entrada. También hay codificadores Transformers, que no utilizan enmascaramiento causal; en cambio, atienden a toda la cadena de entrada para extraer una representación contextual significativa de la entrada.

Para otras tareas, como la traducción de idiomas, también existen transformadores codificadores-descodificadores que pueden traducir de una cadena de texto a otra; este tipo de modelo contiene tanto bloques transformadores codificadores como bloques transformadores decodificadores.

La Tabla 9-1 resume los tres tipos de transformadores, con los mejores ejemplos de cada arquitectura y casos de uso típicos.

Tipo	Ejemplos	Casos de uso
Codificador	BERT (Google)	Clasificación de oraciones, reconocimiento de entidades nombradas, respuesta a preguntas extractivas.
Codificador-decodificador	T5 (Google)	Resumen, traducción, respuesta a preguntas.
Decodificador	GPT-3 (OpenAI)	Generación de texto

Un ejemplo bien conocido de transformador codificador es el modelo de codificador bidireccional de Representaciones de Transformers (BERT), desarrollado por Google (Devlin et al. 2018) que predice las palabras que faltan en una oración, dado el contexto tanto antes como después de la palabra que falta en todas las capas.

TRANSFORMADORES CODIFICADORES

Los transformadores codificadores se utilizan normalmente para tareas que requieren una comprensión de la entrada en su conjunto, como la clasificación de oraciones, el reconocimiento de entidades nombradas y la respuesta extractiva a preguntas. No se utilizan para tareas de generación de texto, por lo que no los exploraremos en detalle en este libro; consulte Procesamiento de Natural con Transformers de Lewis Tunstall et al (O'Reilly) para obtener más información.

En las siguientes secciones exploraremos cómo funcionan los transformadores codificadores-decodificadores y discutiremos las extensiones de la arquitectura del modelo GPT original lanzada por OpenAI, incluyendo ChatGPT, que ha sido diseñado específicamente para aplicaciones conversacionales.

T5

Un ejemplo de un Transformer moderno que utiliza la estructura codificador-decodificador es el modelo T5 de Google.⁵ Este modelo reformula una variedad de tareas en un marco de texto a texto, incluida la traducción, la aceptabilidad lingüística, la similitud de oraciones y el resumen de documentos. como se muestra en la Figura 9-12.

Figura 9-12. Ejemplos de cómo T5 reformula una variedad de tareas en un marco de texto a texto, incluida la traducción, la aceptabilidad lingüística, la similitud de oraciones y el resumen de documentos (fuente: Raffel et al., 2019)

La arquitectura del modelo T5 se asemeja mucho a la arquitectura de codificador-decodificador utilizada en el artículo original de Transformer, como se muestra en la Figura 9-13. La diferencia clave es que T5 se entrena en un enorme corpus de texto de 750 GB (el Colossal Clean Crawled Corpus, o C4), mientras que el documento original de Transformer se centró únicamente en la traducción de idiomas, por lo que se entrenó en 1,4 GB de inglés-alemán pares de oraciones.

Figura 9-13. Un modelo de transformador codificador-decodificador: cada cuadro gris es un bloque de transformador (fuente: Vaswani et al., 2017)

Gran parte de este diagrama ya nos resulta familiar: podemos ver que los bloques Transformer se repiten y se utiliza la incrustación posicional para capturar el orden de las secuencias de entrada. Las dos diferencias clave entre este modelo y el modelo GPT que construimos anteriormente en este capítulo son las siguientes:

En el lado izquierdo, un conjunto de bloques codificadores Transformer codifican la secuencia a traducir. Observe que no hay ningún enmascaramiento causal en la capa de atención. Esto se debe a que no estamos generando más texto para extender la secuencia a traducir; solo queremos aprender una buena representación de la secuencia en su conjunto que pueda enviarse al decodificador. Por lo tanto, las capas de atención en el codificador se pueden desenmascarar por completo para capturar todas las dependencias cruzadas entre palabras, sin importar el orden.

En el lado derecho, un conjunto de bloques decodificadores Transformer generan el texto traducido. La capa de atención inicial es autorreferencial (es decir, la clave, el valor y la consulta provienen de la misma entrada) y se utiliza un enmascaramiento causal para garantizar que la información que llega en tokens futuros no se filtran a la palabra actual que se va a predecir. Sin embargo, luego podemos ver que la capa de atención posterior extrae la clave y el valor del codificador, dejando solo la consulta transmitida desde el decodificador. Esto se denomina atención de referencia cruzada y significa que el decodificador puede atender a la representación del codificador de la secuencia de entrada que se va a traducir. ¡Así es como el decodificador sabe qué significado debe transmitir la traducción!

La figura 9-14 muestra un ejemplo de atención referencial cruzada. Dos cabezales de atención de la capa decodificadora pueden trabajar juntos para proporcionar la traducción correcta al alemán de la palabra *the*, cuando se usa en el contexto de la calle. En alemán, hay tres artículos definidos (*der*, *die*, *das*) dependiendo del género del sustantivo, pero el Transformador sabe elegir morir porque una cabeza de atención es capaz de atender a la palabra calle (una palabra femenina en alemán), mientras otro atiende a la palabra a traducir (*el*).

Figura 9-14. Un ejemplo de cómo una cabeza de atención presta atención a la palabra "the" y otra presta atención a la palabra "street" para traducir

correctamente la palabra "the" a la palabra alemana "die" como el artículo definido femenino de "Straße"

CONSEJO

Este ejemplo es del repositorio Tensor2Tensor GitHub, que contiene un cuaderno Colab que le permite jugar con un modelo Transformer codificador-decodificador entrenado y ver cómo los mecanismos de atención del codificador y decodificador impactan la traducción de una oración determinada al alemán.

GPT-3 y GPT-4

Desde la publicación original de GPT en 2018, OpenAI ha lanzado varias versiones actualizadas que mejoran el modelo original, como se muestra en la Tabla 9-2.

todos

Tamaño de inserción de palabras	Ventana contextual	parámetro	Modelo	Fecha	Capas	Cabezas de atención	
GPT	junio de 2018	12	12	768	512	120,000	
GPT-2	febrero de 2019	48	48	1,600	1,024	1,500,000	000
GPT-3	mayo 2020	96	96	12,888	2,048	175,000	0,000
GPT-4	marzo de 2023						

La arquitectura del modelo de GPT-3 es bastante similar al modelo GPT original, excepto que es mucho más grande y está entrenado con muchos más datos. Al momento de escribir este artículo, GPT-4 se encuentra en versión beta limitada: OpenAI no ha publicado detalles de la estructura y el tamaño del modelo, aunque sí sabemos que puede aceptar imágenes como

entrada, por lo que pasa a ser un modelo multimodal para la primera vez. Los pesos de los modelos GPT-3 y GPT-4 no son de código abierto, aunque los modelos están disponibles a través de una herramienta comercial y API.

GPT-3 también se puede ajustar a sus propios datos de entrenamiento; esto le permite proporcionar múltiples ejemplos de cómo debería reaccionar ante un estilo determinado de aviso actualizando físicamente los pesos de la red. En muchos casos, esto puede no ser necesario, ya que a GPT-3 se le puede indicar cómo reaccionar ante un estilo determinado de mensaje simplemente proporcionando algunos ejemplos en el mensaje mismo (esto se conoce como aprendizaje de pocas tomas). El beneficio del ajuste fino es que no es necesario proporcionar estos ejemplos como parte de cada mensaje de entrada, lo que ahorra costos a largo plazo.

Un ejemplo del resultado de GPT-3, dada una oración de solicitud del sistema, se muestra en la Figura 9-15.

Figura 9-15. Un ejemplo de cómo GPT-3 puede ampliar un mensaje del sistema determinado

Los modelos de lenguaje como GPT se benefician enormemente del escalamiento, tanto en términos de número de pesos del modelo como de tamaño del conjunto de datos. Aún no se ha alcanzado el límite de la capacidad de los modelos de lenguajes grandes, y los investigadores continúan ampliando los límites de lo que es posible con modelos y conjuntos de datos cada vez más grandes.

ChatGPT

Unos meses antes del lanzamiento beta de GPT-4, OpenAI anunció ChatGPT, una herramienta que permite a los usuarios interactuar con su conjunto de grandes modelos de lenguaje a través de una interfaz conversacional. El lanzamiento original de noviembre de 2022 funcionaba con GPT-3.5, una versión del modelo que era más potente que GPT-3 y estaba optimizada para respuestas conversacionales.

En la Figura 9-16 se muestra un diálogo de ejemplo. Observe cómo el agente es capaz de mantener el estado entre entradas, entendiendo que la

atención mencionada en la segunda pregunta se refiere a la atención en el contexto de Transformers, más que a la capacidad de concentración de una persona.

Figura 9-16. Un ejemplo de ChatGPT respondiendo preguntas sobre Transformers

Al momento de escribir este artículo, no existe ningún documento oficial que describa cómo funciona ChatGPT en detalle, pero por la publicación oficial del blog sabemos que utiliza una técnica llamada aprendizaje por refuerzo a partir de retroalimentación humana (RLHF) para ajustar el modelo GPT-3.5. . Esta técnica también se utilizó en el artículo anterior del grupo ChatGPT6 que introdujo el modelo InstructGPT, un modelo GPT-3 optimizado que está diseñado específicamente para seguir con mayor precisión las instrucciones escritas.

El proceso de formación para ChatGPT es el siguiente:

1. Ajuste supervisado: recopile un conjunto de datos de demostración de entradas conversacionales (indicaciones) y resultados deseados que hayan sido escritos por humanos. Esto se utiliza para ajustar el modelo de lenguaje subyacente (GPT-3.5) mediante el aprendizaje supervisado.
2. Modelado de recompensas: presente a un etiquetador humano ejemplos de indicaciones y varios resultados del modelo de muestra y pídale que clasifique los resultados de mejor a peor. Entrenar un modelo de recompensa que predice la puntuación otorgada a cada salida, dado el historial de conversaciones.
3. Aprendizaje por refuerzo: trate la conversación como un entorno de aprendizaje por refuerzo donde la política es el modelo de lenguaje subyacente, inicializado en el modelo ajustado desde el paso 1. Dado el estado actual (el historial de la conversación), la política genera una acción (una secuencia de tokens).), que se califica mediante el modelo de recompensa entrenado en el paso 2. Luego se puede entrenar un algoritmo de aprendizaje por refuerzo (optimización de políticas proximales (PPO)) para maximizar la recompensa, ajustando las ponderaciones del modelo de lenguaje.

APRENDIZAJE REFORZADO

Para obtener una introducción al aprendizaje por refuerzo, consulte el Capítulo 12, donde exploramos cómo se pueden utilizar los modelos generativos en un entorno de aprendizaje por refuerzo.

El proceso RLHF se muestra en la Figura 9-17.

Figura 9-17. El aprendizaje por refuerzo a partir del proceso de ajuste de la retroalimentación humana utilizado en ChatGPT (fuente: OpenAI)

Si bien ChatGPT todavía tiene muchas limitaciones (como a veces "alucinar" información objetivamente incorrecta), es un poderoso ejemplo de cómo se pueden usar Transformers para construir modelos generativos que pueden producir resultados complejos, de largo alcance y novedosos que a menudo son indistinguibles de otros. texto generado por humanos. Los avances logrados hasta ahora por modelos como ChatGPT es un testimonio del potencial de la IA y su impacto transformador en el mundo.

Además, es evidente que la comunicación y la interacción impulsadas por la IA seguirán evolucionando rápidamente en el futuro. Proyectos como Visual ChatGPT7 ahora combinan el poder lingüístico de ChatGPT con modelos básicos visuales como Difusión Estable, lo que permite a los usuarios interactuar con ChatGPT no solo a través de texto, sino también de imágenes. La fusión de capacidades lingüísticas y visuales en proyectos como Visual ChatGPT y GPT-4 tiene el potencial de anunciar una nueva era en la interacción entre humanos y computadoras.

Resumen

En este capítulo, exploramos la arquitectura del modelo Transformer y construimos una versión del modelo GPT: un modelo para la generación de texto de última generación.

GPT utiliza un mecanismo conocido como atención, que elimina la necesidad de capas recurrentes (por ejemplo, LSTM). Funciona como un sistema de recuperación de información, utilizando consultas, claves y

valores para decidir cuánta información quiere extraer de cada token de entrada.

Las cabezas de atención se pueden agrupar para formar lo que se conoce como una capa de atención de cabezas múltiples. Luego, estos se envuelven dentro de un bloque Transformer, que incluye normalización de capas y conexiones de omisión alrededor de la capa de atención. Los bloques transformadores se pueden apilar para crear redes neuronales muy profundas.

El enmascaramiento causal se utiliza para garantizar que GPT no pueda filtrar información de los tokens posteriores a la predicción actual. Además, se utiliza una técnica conocida como codificación posicional para garantizar que el orden de la secuencia de entrada no se pierda, sino que se incluya en la entrada junto con la incrustación de palabras tradicional.

Al analizar el resultado de GPT, vimos que era posible no solo generar nuevos pasajes de texto, sino también interrogar la capa de atención de la red para comprender en qué parte de la oración busca recopilar información para mejorar su predicción. GPT puede acceder a información a distancia sin pérdida de señal, porque las puntuaciones de atención se calculan en paralelo y no dependen de un estado oculto que se transmite a través de la red de forma secuencial, como es el caso de las redes neuronales recurrentes.

Vimos cómo existen tres familias de Transformers (codificador, decodificador y codificador-decodificador) y las diferentes tareas que se pueden realizar con cada uno. Finalmente, exploramos la estructura y el proceso de capacitación de otros grandes modelos de lenguaje como T5 de Google y ChatGPT de OpenAI.

1 Ashish Vaswani et al., “La atención es todo lo que necesitas”, 12 de junio de 2017, <https://arxiv.org/abs/1706.03762>.

2 Alec Radford et al., “Improving Language Understanding by Generative Pre-Training”, 11 de junio de 2018, <https://openai.com/research/language-unsupervised>.

3 Jacob Devlin et al., "BERT: Entrenamiento previo de transformadores bidireccionales profundos para la comprensión del lenguaje", 11 de octubre de 2018, <https://arxiv.org/abs/1810.04805>.

4 Sheng Shen et al., "PowerNorm: Rethinking Batch Normalization in Transformers", 28 de junio de 2020, <https://arxiv.org/abs/2003.07845>.

5 Colin Raffel et al., "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer", octubre 23, 2019, <https://arxiv.org/abs/1910.10683>.

6 Long Ouyang et al., "Entrenamiento de modelos lingüísticos para seguir instrucciones con retroalimentación humana", 4 de marzo de 2022, <https://arxiv.org/abs/2203.02155>.

7 Chenfei Wu et al., "Visual ChatGPT: hablar, dibujar y editar con modelos de Visual Foundation", 8 de marzo de 2023, <https://arxiv.org/abs/2303.04671>.

Capítulo 10. GAN avanzadas

METAS DEL CAPÍTULO

En este capítulo podrás:

Ver cómo un modelo ProGAN entrena progresivamente a una GAN para generar imágenes de alta resolución.

Comprender cómo se adaptó ProGAN para construir una StyleGAN, una GAN de alto rendimiento para síntesis de imágenes.

Explorar cómo se ajustó StyleGAN para crear una StyleGAN2, un modelo de última generación que mejora aún más el trabajo original.

Conocer las contribuciones clave de estos modelos, incluido el entrenamiento progresivo, la normalización de instancias adaptativas, la modulación y demodulación del peso y la regularización de la longitud de la ruta.

Pasear por la arquitectura de la Autoatención GAN (SAGAN), que incorpora el mecanismo de atención en el marco de GAN.

Ver cómo BigGAN amplía las ideas del artículo de SAGAN para producir imágenes de alta calidad.

Descubra cómo VQ-GAN utiliza un libro de códigos para codificar imágenes en una secuencia discreta de tokens que se pueden modelar utilizando un Transformer.

Vea cómo ViT VQ-GAN adapta la arquitectura VQ-GAN para usar Transformers en lugar de capas convolucionales en el codificador y decodificador.

El Capítulo 4 presentó las redes generativas adversarias (GAN), una clase de modelo generativo que ha producido resultados de última generación en una amplia variedad de tareas de generación de imágenes. La flexibilidad

en la arquitectura del modelo y el proceso de capacitación ha llevado a académicos y profesionales del aprendizaje profundo a encontrar nuevas formas de diseñar y entrenar GAN, lo que ha llevado a muchas versiones avanzadas diferentes de la arquitectura que exploraremos en este capítulo.

Introducción

Explicar en detalle todos los desarrollos de GAN y sus repercusiones fácilmente podría llenar otro libro. El repositorio de GAN Zoo en GitHub contiene más de 500 ejemplos distintos de GAN con artículos vinculados, que van desde ABC-GAN hasta ZipNet-GAN!

En este capítulo cubriremos las principales GAN que han tenido influencia en el campo, incluida una explicación detallada de la arquitectura del modelo y el proceso de capacitación para cada una.

Primero exploraremos tres modelos importantes de NVIDIA que han superado los límites de la generación de imágenes:

ProGAN, StyleGAN y StyleGAN2. Analizaremos cada uno de estos modelos con suficiente detalle para comprender los conceptos fundamentales que sustentan las arquitecturas y ver cómo cada uno de ellos se ha basado en ideas de artículos anteriores.

También exploraremos otras dos arquitecturas GAN importantes que incorporan atención: la Autoatención GAN (SAGAN) y BigGAN, que se basaron en muchas de las ideas del artículo de SAGAN. Ya hemos visto el poder del mecanismo de atención en el contexto de Transformers en el Capítulo 9.

Por último, cubriremos VQ-GAN y ViT VQ-GAN, que incorporan una combinación de ideas de transformadores autocodificadores variacionales, y GAN. VQ-GAN es un componente clave del modelo de generación de texto a imagen de última generación de Google Muse. [1] Exploraremos los llamados modelos multimodales con más detalle en el Capítulo 13.

ENTRENANDO TUS PROPIOS MODELOS

Para ser conciso, he optado por no incluir código para construir directamente estos modelos en el repositorio de código de este libro, sino que señalaré implementaciones disponibles públicamente cuando sea posible, para que pueda entrenar sus propias versiones si lo desea.

ProGAN

ProGAN es una técnica desarrollada por NVIDIA Labs en 2017² para mejorar tanto la velocidad como la estabilidad del entrenamiento GAN.

En lugar de entrenar inmediatamente una GAN en imágenes de resolución completa, el artículo de ProGAN sugiere entrenar primero el generador y el discriminador en imágenes de baja resolución de, digamos, 4×4 píxeles y luego agregar capas incrementalmente a lo largo del proceso de entrenamiento para aumentar la resolución.

Veamos con más detalle el concepto de entrenamiento progresivo.

ENTRENANDO A TU PROPIO PROGÁN

Hay un excelente tutorial de Bharath K sobre cómo entrenar tu propio ProGAN usando Keras disponible en el blog de Paperspace. Tenga en cuenta que entrenar un ProGAN para lograr los resultados del artículo requiere una cantidad significativa de potencia informática.

Entrenamiento progresivo

Como siempre con las GAN, construimos dos redes independientes, el generador y el discriminador, y durante el proceso de capacitación se lleva a cabo una lucha por el dominio.

En una GAN normal, el generador siempre genera imágenes de resolución completa, incluso en las primeras etapas del entrenamiento. Es razonable pensar que esta estrategia podría no ser óptima: el generador podría tardar en aprender estructuras de alto nivel en las primeras etapas del entrenamiento, porque opera inmediatamente sobre imágenes complejas de alta resolución. ¿No sería mejor entrenar primero una GAN liviana para

generar imágenes precisas de baja resolución y luego ver si podemos aprovechar esto para aumentar gradualmente la resolución?

Esta simple idea nos lleva al entrenamiento progresivo, una de las contribuciones clave del artículo de ProGAN. ProGAN se entrena en etapas, comenzando con un conjunto de entrenamiento que se ha condensado en imágenes de 4×4 píxeles mediante interpolación, como se muestra en la Figura 10-1.

Figura 10-1. Las imágenes del conjunto de datos se pueden comprimir a una resolución más baja mediante interpolación.

Luego podemos entrenar inicialmente al generador para transformar un vector de ruido de entrada latente z (digamos, de longitud 512) en una imagen de forma $4 \times 4 \times 3$. El discriminador coincidente necesitará transformar una imagen de entrada de tamaño $4 \times 4 \times 3$. en una única predicción escalar. Las arquitecturas de red para este primer paso se muestran en la Figura 10-2.

El cuadro azul en el generador representa la capa convolucional que convierte el conjunto de mapas de características en una imagen RGB (toRGB), y el cuadro azul en el discriminador representa la capa convolucional que convierte las imágenes RGB en un conjunto de mapas de características (fromRGB) .

Figura 10-2. Las arquitecturas generadora y discriminadora para la primera etapa del proceso de capacitación de ProGAN

En el artículo, los autores entran este par de redes hasta que el discriminador haya visto 800.000 imágenes reales. Ahora necesitamos comprender cómo se expanden el generador y el discriminador para que funcionen con imágenes de 8×8 píxeles.

Para expandir el generador y el discriminador, necesitamos combinar capas adicionales. Esto se gestiona en dos fases, transición y estabilización, como se muestra en la Figura 10-3.

Figura 10-3. El proceso de entrenamiento del generador ProGAN, que expande la red de imágenes de 4×4 a 8×8 (las líneas de puntos representan el resto de la red, no se muestran)

Primero veamos el generador. Durante la fase de transición, se agregan nuevas capas convolucionales y de muestreo ascendente a la red existente, con una conexión residual configurada para mantener la salida de la capa toRGB entrenada existente. Fundamentalmente, las nuevas capas se enmascaran inicialmente utilizando un parámetro α que aumenta gradualmente de 0 a 1 a lo largo de la fase de transición para permitir que pase más salida toRGB nueva y menos de la capa toRGB existente. Esto es para evitar un impacto en la red a medida que las nuevas capas toman el control.

Al final, no hay flujo a través de la antigua capa toRGB y la red entra en la fase de estabilización: un período adicional de entrenamiento en el que la red puede ajustar la salida, sin ningún flujo a través de la antigua capa toRGB.

El discriminador utiliza un proceso similar, como se muestra en la Figura 10-4.

Figura 10-4. El proceso de entrenamiento del discriminador ProGAN, que expande la red de imágenes de 4×4 a 8×8 (las líneas de puntos representan el resto de la red, no se muestran)

Aquí, necesitamos combinar capas convolucionales y de reducción de escala adicionales. Nuevamente, las capas se inyectan en la red, esta vez al inicio de la red, justo después de la imagen de entrada. La capa fromRGB existente se conecta a través de una conexión residual y se elimina gradualmente a medida que las nuevas capas toman el control durante la fase de transición. La fase de estabilización permite al discriminador realizar ajustes utilizando las nuevas capas.

Todas las fases de transición y estabilización duran hasta que al discriminador se le han mostrado 800.000 imágenes reales. Tenga en cuenta que aunque la red se entrena progresivamente, no se congela ninguna capa.

A lo largo del proceso de formación, todas las capas siguen siendo totalmente entrenables.

Este proceso continúa, haciendo crecer la GAN de imágenes de 4×4 a 8×8 , luego 16×16 , 32×32 , y así sucesivamente, hasta que alcanza la resolución completa (1024×1024), como se muestra en la Figura 10-5.

Figura 10-5. El mecanismo de entrenamiento de ProGAN y algunos ejemplos de caras generadas (fuente: Karras et al., 2017)

La estructura general del generador y discriminador una vez completado el proceso de entrenamiento progresivo completo se muestra en la Figura 10-6.

Figura 10-6. El generador y discriminador ProGAN utilizado para generar caras CelebA de 1024×1024 píxeles (fuente: Karras et al., 2018)

El artículo también hace otras contribuciones importantes, a saber, la desviación estándar de minibatch, las tasas de aprendizaje ecualizadas y la normalización por píxeles, que se describen brevemente en las siguientes secciones.

Desviación estándar del minibatch

La capa de desviación estándar del minibatch es una capa adicional en el discriminador que agrega la desviación estándar de los valores de las características, promediada en todos los píxeles y en el minibatch como una característica adicional (constante). Esto ayuda a garantizar que el generador cree más variedad en su producción: si la variedad es baja en todo el minibatch, entonces la desviación estándar será pequeña y el discriminador puede usar esta característica para distinguir los lotes falsos de los lotes reales. Por lo tanto, se incentiva al generador para garantizar que genere una cantidad similar de variedad a la que está presente en los datos de entrenamiento reales.

Tasas de aprendizaje igualadas

Todas las capas densas y convolucionales de ProGAN utilizan tasas de aprendizaje ecualizadas. Por lo general, los pesos en una red neuronal se inicializan utilizando un método como la inicialización He, una distribución gaussiana donde la desviación estándar se escala para que sea inversamente proporcional a la raíz cuadrada del número de entradas a la capa. De esta manera, las capas con una mayor cantidad de entradas se inicializarán con pesos que tengan una desviación menor de cero, lo que generalmente mejora la estabilidad del proceso de entrenamiento.

Los autores del artículo de ProGAN descubrieron que esto causaba problemas cuando se usaba en combinación con optimizadores modernos como Adam o RMSProp. Estos métodos normalizan la actualización de gradiente para cada peso, de modo que el tamaño de la actualización sea independiente de la escala (magnitud) del peso. Sin embargo, esto significa que los pesos con un rango dinámico mayor (es decir, capas con menos entradas) tardarán comparativamente más en ajustarse que los pesos con un rango dinámico más pequeño (es decir, capas con más entradas). Se descubrió que esto provoca un desequilibrio entre la velocidad de entrenamiento de las diferentes capas del generador y discriminador en ProGAN, por lo que utilizaron tasas de aprendizaje igualadas para resolver este problema.

En ProGAN, los pesos se inicializan utilizando un estándar simple gaussiano, independientemente del número de entradas a la capa.

La normalización se aplica dinámicamente, como parte de la llamada a la capa, en lugar de solo durante la inicialización. De esta manera, el optimizador considera que cada peso tiene aproximadamente el mismo rango dinámico, por lo que aplica la misma tasa de aprendizaje. Sólo cuando se llama a la capa el peso se escala según el factor del inicializador He.

Normalización por píxeles

Por último, en ProGAN se utiliza la normalización por píxeles en el generador, en lugar de la normalización por lotes. Esto normaliza el vector de características en cada píxel a una unidad de longitud y ayuda a evitar

que la señal se salga de control a medida que se propaga a través de la red. La capa de normalización por píxeles no tiene pesos entrenables.

Salidas

Además del conjunto de datos de CelebA, ProGAN también se aplicó a imágenes del conjunto de datos de comprensión de escenas a gran escala (LSUN) con excelentes resultados, como se muestra en la Figura 10-7. Esto demostró el poder de ProGAN sobre arquitecturas GAN anteriores y allanó el camino para futuras iteraciones como StyleGAN y StyleGAN2, que exploraremos en las siguientes secciones.

Figura 10-7. Ejemplos generados a partir de un ProGAN capacitado progresivamente en el conjunto de datos LSUN con resolución de 256×256 (fuente: Karras et al., 2017)

EstiloGAN

StyleGAN3 es una arquitectura GAN de 2018 que se basa en las ideas anteriores del artículo de ProGAN. De hecho, el discriminador es idéntico; Sólo se cambia el generador.

A menudo, cuando se entrena GAN, es difícil separar los vectores en el espacio latente correspondientes a atributos de alto nivel; con frecuencia están entrelazados, lo que significa que ajustar una imagen en el espacio latente para darle a una cara más pecas, por ejemplo, también podría cambiar inadvertidamente el color de fondo. Si bien ProGAN genera imágenes increíblemente realistas, no es una excepción a esta regla general. Lo ideal sería tener control total del estilo de la imagen, y esto requiere una separación clara de características en el espacio latente.

StyleGAN logra esto inyectando explícitamente vectores de estilo en la red en diferentes puntos: algunos que controlan características de alto nivel (por ejemplo, la orientación de la cara) y otros que controlan detalles de bajo nivel (por ejemplo, la forma en que el cabello cae sobre la frente).

La arquitectura general del generador StyleGAN se muestra en la Figura 10-8. Repasemos esta arquitectura paso a paso, comenzando con la red de

mapeo.

Figura 10-8. La arquitectura del generador StyleGAN (fuente: Karras et al., 2018)

ENTRENANDO TU PROPIO STYLEGAN

Hay un excelente tutorial de Soon-Yau Cheong sobre cómo entrenar su propio StyleGAN usando Keras disponible en el sitio web de Keras. Tenga en cuenta que entrenar un StyleGAN para lograr los resultados del artículo requiere una cantidad significativa de potencia informática.

La red cartográfica

La red de mapeo f es una red de retroalimentación simple que convierte el ruido de entrada $z \in Z$ en un espacio latente diferente $w \in W$. Esto le da al generador la oportunidad de desenredar el vector de entrada ruidoso en distintos factores de variación, que pueden ser fácilmente captados por las capas generadoras de estilo posteriores.

El objetivo de hacer esto es separar el proceso de elegir un estilo para la imagen (la red de mapeo) de la generación de una imagen con un estilo determinado (la red de síntesis).

La red de síntesis

La red de síntesis es el generador de la imagen real con un estilo determinado, proporcionado por la red de mapeo. Como se puede ver en la Figura 10-8, el vector de estilo w se inyecta en la red de síntesis en diferentes puntos, cada vez a través de una capa A_i densamente conectada de manera diferente, lo que genera dos vectores: un vector de polarización b, i y un vector de escala $s, y o$.

Estos vectores definen el estilo específico que se debe inyectar en este punto de la red; es decir, le dicen a la red de síntesis cómo ajustar los mapas de características para mover la imagen generada en la dirección del estilo especificado.

Este ajuste se logra mediante capas de normalización de instancias adaptativas (AdaIN).

Normalización de instancia adaptativa

Una capa AdaIN es un tipo de capa de red neuronal que ajusta la media y la varianza de cada mapa de características x_i con un sesgo de estilo de referencia $y_{b,i}$ y una escala $s_{i,i}$, respectivamente.⁴ Ambos vectores tienen una longitud igual al número de canales de salida. de la capa convolucional anterior en la red de síntesis. La ecuación para la normalización de instancias adaptativas es la siguiente:

$$\text{AdaIN}(x_i, y) = y s_i$$

$$x_i - \mu(x_i) \sigma(x_i)$$

- $y b_i, y o$

Las capas de normalización de instancias adaptables garantizan que los vectores de estilo que se inyectan en cada capa solo afecten a las características de esa capa, evitando que cualquier información de estilo se filtre entre capas. Los autores muestran que esto da como resultado que los vectores latentes w estén significativamente más desenredados que los vectores z originales.

Dado que la red de síntesis se basa en la arquitectura ProGAN, se entrena progresivamente. Los vectores de estilo en las capas anteriores de la red de síntesis (cuando la resolución de la imagen es más baja: 4×4 , 8×8) afectarán a características más generales que las posteriores en la red (64×64 a $1024 \times$

Resolución de 1.024 píxeles). Esto significa que no solo tenemos control total sobre la imagen generada a través del vector latente w , sino que también podemos cambiar el vector w en diferentes puntos de la red de síntesis para cambiar el estilo en una variedad de niveles de detalle.

Mezcla de estilos

Los autores utilizan un truco conocido como mezcla de estilos para garantizar que el generador no pueda utilizar correlaciones entre estilos adyacentes durante el entrenamiento (es decir, los estilos inyectados en cada capa estén lo más desenredados posible). En lugar de muestrear sólo un único vector latente z , se muestran dos (z_1, z_2), correspondientes a dos vectores de estilo (w_1, w_2). Luego, en cada capa, se elige (w_1 o w_2) al azar, para romper cualquier posible correlación entre los vectores.

Variación estocástica

La red del sintetizador agrega ruido (pasado a través de una capa de transmisión B aprendida) después de cada convolución para tener en cuenta detalles estocásticos como la ubicación de los pelos individuales o el fondo detrás de la cara. Nuevamente, la profundidad a la que se inyecta el ruido afecta la tosiedad del impacto en la imagen.

Esto también significa que la entrada inicial a la red de síntesis puede ser simplemente una constante aprendida, en lugar de ruido adicional. Ya hay suficiente estocasticidad presente en las entradas de estilo y de ruido para generar suficiente variación en las imágenes.

Salidas de StyleGAN

La Figura 10-9 muestra StyleGAN en acción.

Figura 10-9. Fusionar estilos entre dos imágenes generadas con diferentes niveles de detalle (fuente: Karras et al., 2018)

Aquí, se generan dos imágenes, fuente A y fuente B, a partir de dos vectores w diferentes. Para generar una imagen fusionada, el vector fuente A w pasa a través de la red de síntesis pero, en algún momento, se cambia por el vector fuente B w . Si este cambio ocurre desde el principio (resolución de 4×4 u 8×8), los estilos burdos como la pose, la forma de la cara y las gafas de la fuente B se trasladan a la fuente A.

Sin embargo, si el cambio ocurre más tarde, solo se transmiten detalles finos de la fuente B, como los colores y la microestructura de la cara, mientras que se conservan los rasgos toscos de la fuente A.

EstiloGAN2

La contribución final en esta cadena de importantes artículos sobre GAN es StyleGAN2.5. Esto se basa aún más en la arquitectura StyleGAN, con algunos cambios clave que mejoran la calidad de la salida generada. En particular, las generaciones StyleGAN2 no sufren tanto por artefactos (áreas de la imagen similares a gotas de agua que se encontró que fueron causadas por las capas de normalización de instancias adaptativas en StyleGAN, como se muestra en la Figura 10-10).

Figura 10-10. Un artefacto en una imagen de una cara generada por StyleGAN (fuente:

Karras et al., 2019)

Tanto el generador como el discriminador en StyleGAN2 son diferentes de StyleGAN. En las siguientes secciones exploraremos las diferencias clave entre las arquitecturas.

ENTRENANDO TU PROPIO ESTILOGAN2

El código oficial para entrenar tu propio StyleGAN usando TensorFlow está disponible en GitHub. Tenga en cuenta que entrenar un StyleGAN2 para lograr los resultados del artículo requiere una cantidad significativa de potencia informática.

Modulación y demodulación de peso

El problema de los artefactos se resuelve eliminando las capas AdaIN en el generador y reemplazándolas con pasos de modulación y demodulación de peso, como se muestra en la Figura 1011. w representa los pesos de la capa convolucional, que se actualizan directamente mediante los pasos de modulación y demodulación en StyleGAN2. en tiempo de ejecución. En comparación, las capas AdaIN de StyleGAN operan sobre el tensor de imagen a medida que fluye a través de la red.

La capa AdaIN en StyleGAN es simplemente una normalización de instancia seguida de modulación de estilo (escalado y sesgo). La idea en

StyleGAN2 es aplicar modulación y normalización (demodulación) de estilo directamente a los pesos de las capas convolucionales en tiempo de ejecución, en lugar de la salida de las capas convolucionales, como se muestra en la Figura 10-11.

Los autores muestran cómo esto elimina el problema de los artefactos manteniendo el control del estilo de la imagen.

Figura 10-11. Una comparación entre los bloques de estilo StyleGAN y StyleGAN2

En StyleGAN2, cada capa Dense A genera un único vector de estilo s_i , donde i indexa el número de canales de entrada en la capa convolucional correspondiente. Luego, este vector de estilo se aplica a los pesos de la capa convolucional de la siguiente manera:

$$i,j,k$$

$$= s_i \cdot w_{i,j,k}$$

Aquí, j indexa los canales de salida de la capa y k indexa las dimensiones espaciales. Este es el paso de modulación del proceso.

Luego, necesitamos normalizar los pesos para que nuevamente tengan una desviación estándar unitaria, para garantizar la estabilidad en el proceso de entrenamiento. Este es el paso de demodulación: "

$$i,j,k \sqrt{\sum}$$

$$i, k$$

$$i,j,k$$

$$i,j,k + \epsilon$$

donde ϵ es un valor constante pequeño que evita la división por cero.

En el artículo, los autores muestran cómo este simple cambio es suficiente para evitar artefactos de gotas de agua, manteniendo al mismo tiempo el

control sobre las imágenes generadas a través de los vectores de estilo y garantizando que la calidad de la salida se mantenga alta.

Regularización de la longitud del camino

Otro cambio realizado en la arquitectura StyleGAN es la inclusión de un término de penalización adicional en la función de pérdida; esto se conoce como regularización de la longitud de la ruta.

Nos gustaría que el espacio latente fuera lo más suave y uniforme posible, de modo que un paso de tamaño fijo en el espacio latente en cualquier dirección dé como resultado un cambio de magnitud fija en la imagen.

Para fomentar esta propiedad, StyleGAN2 pretende minimizar el siguiente término, junto con la habitual pérdida de Wasserstein con penalización de gradiente:

$$E_{w,y} (\|J$$

$$y\|$$

- una)

Aquí, w es un conjunto de vectores de estilo creados por la red de mapeo, y es un conjunto de imágenes ruidosas extraídas de $N(0, I)$ y ∂g es el jacobiano de la red generadora con respecto a los vectores de estilo.

$$J_w =$$

$$\partial_w$$

El término $\|J^T y\|$ mide la magnitud de las imágenes y después de la transformación por los gradientes dados en el jacobiano.

Queremos que esto esté cerca de una constante a , que se calcula dinámicamente como el promedio móvil exponencial de $\|J^T y\|$ a medida que avanza el entrenamiento.

Los autores encuentran que este término adicional hace que la exploración del espacio latente sea más confiable y consistente. Además, los términos de regularización en la función de pérdida solo se aplican una vez cada 16 minibatches, por motivos de eficiencia. Esta técnica, llamada regularización diferida, no provoca una caída mensurable en el rendimiento.

Sin crecimiento progresivo

Otra actualización importante es la forma en que se entrena StyleGAN2.

En lugar de adoptar el mecanismo de entrenamiento progresivo habitual, StyleGAN2 utiliza conexiones de salto en el generador y conexiones residuales en el discriminador para entrenar a toda la red como una sola. Ya no es necesario entrenar diferentes resoluciones de forma independiente y combinarlas como parte del proceso de capacitación.

La Figura 10-12 muestra los bloques generador y discriminador en StyleGAN2.

Figura 10-12. Los bloques generador y discriminador en StyleGAN2

La propiedad crucial que nos gustaría poder preservar es que StyleGAN2 comienza aprendiendo características de baja resolución y refina gradualmente el resultado a medida que avanza el entrenamiento. Los autores demuestran que esta propiedad efectivamente se conserva utilizando esta arquitectura. Cada red se beneficia del refinamiento de los pesos convolucionales en las capas de menor resolución en las primeras etapas del entrenamiento, y las conexiones de salto y residuales utilizadas para pasar la salida a través de las capas de mayor resolución prácticamente no se ven afectadas. A medida que avanza el entrenamiento, las capas de mayor resolución comienzan a dominar, a medida que el generador descubre formas más complejas de mejorar el realismo de las imágenes para engañar al discriminador. Este proceso se demuestra en la Figura 10-13.

Figura 10-13. La contribución de cada capa de resolución a la salida del generador, por tiempo de entrenamiento (adaptado de Karras et al., 2019)

Salidas de StyleGAN2

Algunos ejemplos de salida de StyleGAN2 se muestran en la Figura 10-14. Hasta la fecha, la arquitectura StyleGAN2 (y variaciones escaladas como StyleGAN-XL6) siguen siendo lo último en generación de imágenes en conjuntos de datos como Flickr-FacesHQ (FFHQ) y CIFAR-10, según el sitio web de evaluación comparativa Papers with Code.

Figura 10-14. Salida StyleGAN2 no seleccionada para el conjunto de datos faciales FFHQ y el conjunto de datos de automóviles LSUN (fuente: Karras et al., 2019)

Otras GAN importantes

En esta sección, exploraremos dos arquitecturas más que también han contribuido significativamente al desarrollo de GAN: SAGAN y BigGAN.

GAN de autoatención (SAGAN)

La Self-Attention GAN (SAGAN)⁷ es un desarrollo clave para GAN, ya que muestra cómo el mecanismo de atención que impulsa los modelos secuenciales como el Transformer también se puede incorporar en modelos basados en GAN para la generación de imágenes. La Figura 10-15 muestra el mecanismo de autoatención del artículo que presenta esta arquitectura.

Figura 10-15. El mecanismo de autoatención dentro del modelo SAGAN (fuente:

Zhang et al., 2018)

El problema con los modelos basados en GAN que no incorporan atención es que los mapas de características convolucionales solo pueden procesar información localmente. Conectar información de píxeles de un lado de una imagen al otro requiere múltiples capas convolucionales que reducen el tamaño de la imagen y al mismo tiempo aumentan la cantidad de canales. La información posicional precisa se reduce a lo largo de este proceso a favor de capturar características de nivel superior, lo que hace que sea computacionalmente ineficiente para el modelo aprender dependencias de largo alcance entre píxeles conectados distantes.

SAGAN resuelve este problema incorporando el mecanismo de atención que exploramos anteriormente en este capítulo en la GAN. El efecto de esta inclusión se muestra en la Figura 10-16.

Figura 10-16. Una imagen generada por SAGAN de un pájaro (celda más a la izquierda) y los mapas de atención de la capa generadora final basada en la atención para los píxeles cubiertos por los tres puntos de colores (celdas más a la derecha) (fuente: Zhang et al., 2018)

El punto rojo es un píxel que forma parte del cuerpo del ave, por lo que la atención se centra naturalmente en las células del cuerpo circundante. El punto verde es parte del fondo, y aquí la atención realmente recae en el otro lado de la cabeza del pájaro, en otros píxeles del fondo. El punto azul es parte de la larga cola del ave, por lo que la atención se centra en otros píxeles de la cola, algunos de los cuales están alejados del punto azul. Sería difícil mantener esta dependencia de largo alcance para los píxeles sin atención, especialmente para las estructuras largas y delgadas de la imagen (como la cola en este caso).

ENTRENANDO TU PROPIO SAGAN

El código oficial para entrenar tu propio SAGAN usando TensorFlow está disponible en GitHub. Tenga en cuenta que entrenar un SAGAN para lograr los resultados del artículo requiere una cantidad significativa de potencia informática.

GranGAN

BigGAN8, desarrollado en DeepMind, amplía las ideas del artículo de SAGAN. La Figura 10-17 muestra algunas de las imágenes generadas por BigGAN, entrenadas en el conjunto de datos ImageNet en resolución 128×128.

Figura 10-17. Ejemplos de imágenes generadas por BigGAN (fuente: Brock et al., 2018)

Además de algunos cambios incrementales en el modelo SAGAN básico, también se describen varias innovaciones en el documento que llevan el

modelo al siguiente nivel de sofisticación. Una de esas innovaciones es el llamado truco de truncamiento. Aquí es donde la distribución latente utilizada para el muestreo es diferente de la distribución $z \sim N(0, I)$ utilizada durante el entrenamiento. Específicamente, la distribución utilizada durante el muestreo es una distribución normal truncada (valores de remuestreo de z que tienen una magnitud mayor que cierto umbral). Cuanto menor sea el umbral de truncamiento, mayor será la credibilidad de las muestras generadas, a expensas de una variabilidad reducida. Este concepto se muestra en la Figura 10-18.

Figura 10-18. El truco del truncamiento: de izquierda a derecha, el umbral se establece en 2, 1, 0,5 y 0,04 (fuente: Brock et al., 2018)

Además, como sugiere el nombre, BigGAN es una mejora con respecto a SAGAN en parte simplemente por ser más grande. BigGAN utiliza un tamaño de lote de 2048, 8 veces mayor que el tamaño de lote de 256 utilizado en SAGAN y un tamaño de canal que aumenta en un 50% en cada capa. Sin embargo, BigGAN muestra además que SAGAN se puede mejorar estructuralmente mediante la inclusión de una incrustación compartida, mediante regularización ortogonal e incorporando el vector latente z en cada capa del generador, en lugar de solo la capa inicial.

Para una descripción completa de las innovaciones introducidas por BigGAN, recomiendo leer el artículo original y el material de presentación que lo acompaña.

USANDO BIGGAN

En el sitio web de TensorFlow hay disponible un tutorial para generar imágenes utilizando un BigGAN previamente entrenado.

VQ-GAN

Otro tipo importante de GAN es el Vector Quantized

GAN (VQ-GAN), presentado en 2020.⁹ Esta arquitectura de modelo se basa en una idea presentada en el artículo de 2017 “Neural Discrete Representation Learning”¹⁰, es decir, que las representaciones aprendidas

por un VAE pueden ser discretas, en lugar de continuas. Se demostró que este nuevo tipo de modelo, Vector Quantized VAE (VQ-VAE), genera imágenes de alta calidad y al mismo tiempo evita algunos de los problemas que se observan a menudo con el espacio latente continuo tradicional VAE, como el colapso posterior (donde el espacio latente aprendido deja de ser informativo debido a un decodificador demasiado potente).

CONSEJO

La primera versión de DALL.E, un modelo de texto a imagen lanzado por OpenAI en 2021 (ver Capítulo 13) utilizó un VAE con un espacio latente discreto, similar a VQ-VAE.

Por espacio latente discreto nos referimos a una lista aprendida de vectores (el libro de códigos), cada uno asociado con un índice correspondiente. El trabajo del codificador en un VQ-VAE es colapsar la imagen de entrada en una cuadrícula más pequeña de vectores que luego se puede comparar con el libro de códigos. El vector de libro de códigos más cercano a cada vector cuadrado de cuadrícula (por distancia euclídea) se lleva adelante para ser decodificado por el decodificador, como se muestra en la Figura 10-19. El libro de códigos es una lista de vectores aprendidos de longitud d (el tamaño de incrustación) que coincide con el número de canales en la salida del codificador y la entrada al decodificador. Por ejemplo, e_1 es un vector que puede interpretarse como fondo.

Figura 10-19. Un diagrama de un VQ-VAE

El libro de códigos puede considerarse como un conjunto de conceptos discretos aprendidos que comparten el codificador y el decodificador para describir el contenido de una imagen determinada. El VQ VAE debe encontrar una manera de hacer que este conjunto de conceptos discretos sea lo más informativo posible para que el codificador pueda etiquetar con precisión cada cuadrado de la cuadrícula con un vector de código particular que sea significativo para el decodificador. Por lo tanto, la función de pérdida para un VQ-VAE es la pérdida de reconstrucción agregada a dos términos (pérdida de alineación y compromiso) que garantizan que los vectores de salida del codificador estén lo más cerca posible de los vectores

en el libro de códigos. Estos términos reemplazan el término de divergencia KL entre la distribución codificada y el estándar gaussiano previo en una típica VAE.

Sin embargo, esta arquitectura plantea una pregunta: ¿cómo tomamos muestras de nuevas cuadrículas de código para pasárlas al decodificador y generar nuevas imágenes? Claramente, usar una priorización uniforme (elegir cada código con la misma probabilidad para cada cuadrado de la cuadrícula) no funcionará. Por ejemplo, en el conjunto de datos MNIST, es muy probable que el cuadrado de la cuadrícula superior izquierda se codifique como fondo, mientras que es poco probable que los cuadrados de la cuadrícula hacia el centro de la imagen se codifiquen como tal. Para resolver este problema, los autores utilizaron otro modelo, un PixelCNN autorregresivo (consulte el Capítulo 5), para predecir el siguiente vector de código en la cuadrícula, dados los vectores de código anteriores. En otras palabras, el modelo aprende el prior, en lugar de ser estático como en el caso del VAE básico.

ENTRENANDO TU PROPIO VQ-VAE

Hay un excelente tutorial de Sayak Paul sobre cómo entrenar su propio VQVAE usando Keras disponible en el sitio web de Keras.

El artículo VQ-GAN detalla varios cambios clave en la arquitectura VQVAE, como se muestra en la Figura 10-20.

Figura 10-20. Un diagrama de un VQ-GAN: el discriminador de GAN ayuda a alentar al VAE a generar imágenes menos borrosas a través de un término de pérdida adversa adicional

En primer lugar, como sugiere el nombre, los autores incluyen un discriminador GAN que intenta distinguir entre la salida del decodificador VAE y las imágenes reales, acompañado de un término contradictorio en la función de pérdida. Se sabe que las GAN producen imágenes más nítidas que las VAE, por lo que esta adición mejora la calidad general de la imagen. Tenga en cuenta que, a pesar del nombre, el VAE todavía está presente en un modelo VQ-GAN; el discriminador GAN es un componente adicional en

lugar de un reemplazo del VAE. La idea de combinar un VAE con un discriminador GAN (VAE-GAN) fue introducida por primera vez por Larsen et al. en su artículo de 2015.¹¹

En segundo lugar, el discriminador GAN predice si pequeñas partes de las imágenes son reales o falsas, en lugar de la imagen completa a la vez. Esta idea (PatchGAN) se aplicó en el exitoso modelo de imagen a imagen pix2pix presentado en 2016 por Isola et al.¹² y también se aplicó con éxito como parte de CycleGAN,¹³ otro modelo de transferencia de estilo de imagen a imagen.

El discriminador PatchGAN genera un vector de predicción (una predicción para cada parche), en lugar de una única predicción para la imagen general. El beneficio de utilizar un discriminador PatchGAN es que la función de pérdida puede medir qué tan bueno es el discriminador para distinguir imágenes en función de su estilo, en lugar de su contenido. Dado que cada elemento individual de la predicción del discriminador se basa en un pequeño cuadrado de la imagen, debe utilizar el estilo del parche, en lugar de su contenido, para tomar su decisión. Esto es útil porque sabemos que los VAE producen imágenes que son estilísticamente más borrosas que las imágenes reales, por lo que el discriminador PatchGAN puede alentar al decodificador VAE a generar imágenes más nítidas de las que produciría naturalmente.

En tercer lugar, en lugar de utilizar una única pérdida de reconstrucción MSE que compara los píxeles de la imagen de entrada con los píxeles de salida del decodificador VAE, VQ-GAN utiliza un término de pérdida de percepción >que calcula la diferencia entre mapas de características en las capas intermedias del codificador y las capas correspondientes del decodificador. Esta idea es del artículo de 2016 de Hou et al.¹⁴, donde los autores muestran que este cambio en la función de pérdida da como resultado generaciones de imágenes más realistas.

Por último, en lugar de PixelCNN, se utiliza un Transformer como parte autorregresiva del modelo, entrenado para generar secuencias de códigos. El Transformer se entrena en una fase separada, después de que VQ-GAN se haya entrenado por completo.

En lugar de utilizar todos los tokens anteriores de forma totalmente autorregresiva, los autores optan por utilizar únicamente tokens que se encuentren dentro de una ventana deslizante alrededor del token que se va a predecir. Esto garantiza que el modelo se escala a imágenes más grandes, lo que requiere un tamaño de cuadrícula latente más grande y, por lo tanto, el Transformer generará más tokens.

ViT VQ-GAN

Yu et al. realizaron una extensión final del VQ-GAN en su artículo de 2021 titulado “Modelado de imagen cuantificada vectorial con VQGAN mejorado”.¹⁵ Aquí, los autores muestran cómo el codificador y decodificador convolucional del VQ-GAN se puede reemplazar con Transformers como se muestra en la Figura 10-21.

Para el codificador, los autores utilizan un Vision Transformer (ViT).¹⁶ Un ViT es una arquitectura de red neuronal que aplica el modelo Transformer, originalmente diseñado para el procesamiento del lenguaje natural, a datos de imágenes. En lugar de utilizar capas convolucionales para extraer características de una imagen, una ViT divide la imagen en una secuencia de parches, que se tokenizan y luego se envían como entrada a un codificador Transformer.

Específicamente, en ViT VQ-GAN, los parches de entrada que no se superponen (cada uno de tamaño 8×8) primero se aplana y luego se proyectan en un espacio de incrustación de baja dimensión, donde se agregan incrustaciones posicionales. Luego, esta secuencia se envía a un codificador estándar Transformer y las incrustaciones resultantes se cuantifican de acuerdo con un libro de códigos aprendido. Estos códigos enteros luego son procesados por un decodificador modelo Transformer, siendo el resultado general una secuencia de parches que se pueden volver a unir para formar la imagen original. El modelo general codificador-decodificador se entrena de un extremo a otro como un auto codificador.

Figura 10-21. Un diagrama de un ViT VQ-GAN: el discriminador de GAN ayuda a alentar al VAE a generar imágenes menos borrosas a través de un término de pérdida adversa adicional (fuente: Yu y Koh, 2022)¹⁷

Al igual que con el modelo VQ-GAN original, la segunda fase de entrenamiento implica el uso de un transformador decodificador autorregresivo para generar secuencias de códigos. Por lo tanto, en total, hay tres transformadores en un ViT VQ-GAN, además del discriminador GAN y el libro de códigos aprendido.

En la Figura 10-22 se muestran ejemplos de imágenes generadas por ViT VQ-GAN a partir del artículo.

Figura 10-22. Imágenes de ejemplo generadas por un ViT VQ-GAN entrenado en ImageNet (fuente: Yu et al., 2021)

Resumen

En este capítulo, hemos realizado un recorrido por algunos de los artículos GAN más importantes e influyentes desde 2017. En particular, hemos explorado ProGAN, StyleGAN, StyleGAN2, SAGAN, BigGAN, VQ-GAN y ViT VQ-GAN.

Comenzamos explorando el concepto de entrenamiento progresivo que fue pionero en el artículo ProGAN de 2017. Se introdujeron varios cambios clave en el artículo StyleGAN de 2018 que brindaron un mayor control sobre la salida de la imagen, como la red de mapeo para crear un vector de estilo específico y una red de síntesis que permitió inyectar el estilo en diferentes resoluciones. Finalmente, StyleGAN2 reemplazó la normalización de instancia adaptativa de StyleGAN con modulación de peso y pasos de demodulación, junto con mejoras adicionales como la regularización de ruta. El artículo también mostró cómo se podría conservar la propiedad deseable del refinamiento gradual de la resolución sin tener que entrenar la red progresivamente.

También vimos cómo el concepto de atención podría incorporarse a una GAN, con la introducción de SAGAN en 2018. Esto permite a la red capturar dependencias de largo alcance, como colores de fondo similares en lados opuestos de una imagen, sin depender de profundidad. mapas convolucionales para difundir la información sobre las dimensiones espaciales de la imagen. BigGAN fue una extensión de esta idea que realizó

varios cambios clave y capacitó a una red más grande para mejorar aún más la calidad de la imagen.

En el artículo de VQ-GAN, los autores muestran cómo se pueden combinar varios tipos diferentes de modelos generativos con gran efecto. Sobre la base del artículo original de VQ-VAE que introdujo el concepto de un VAE con un espacio latente discreto, VQ-GAN incluye además un discriminador que alienta al VAE a generar imágenes menos borrosas a través de un término de pérdida adversario adicional. Un transformador autorregresivo se utiliza para construir una secuencia novedosa de tokens de código que el decodificador VAE puede decodificar para producir imágenes novedosas. El artículo de ViT VQ-GAN amplía esta idea aún más, reemplazando el codificador y decodificador convolucional de VQ-GAN con Transformers.

1 Huiwen Chang et al., “Muse: generación de texto a imagen a través de transformadores generativos enmasacarados”, 2 de enero de 2023, <https://arxiv.org/abs/2301.00704>.

2 Tero Karras et al., “Crecimiento progresivo de GAN para mejorar la calidad”, estabilidad y variación”, 27 de octubre de 2017, <https://arxiv.org/abs/1710.10196>.

3 Tero Karras et al., “Una arquitectura generadora basada en estilos para redes adversarias generativas”, 12 de diciembre de 2018, <https://arxiv.org/abs/1812.04948>.

4 Xun Huang y Serge Belongie, “Transferencia de estilo arbitrario en tiempo real con normalización de instancia adaptativa”, 20 de marzo de 2017, <https://arxiv.org/abs/1703.06868>.

5 Tero Karras et al., “Analizar y mejorar la calidad de imagen de StyleGAN”, 3 de diciembre de 2019, <https://arxiv.org/abs/1912.04958>.

6 Axel Sauer et al., “StyleGAN-XL: Escalando StyleGAN a conjuntos de datos diversos grandes”, 1 de febrero de 2022, <https://arxiv.org/abs/2202.00273v2>.

7 Han Zhang et al., “Self-Attention Generative Adversarial Networks”, mayo 21, 2018, <https://arxiv.org/abs/1805.08318>.

8 Andrew Brock et al., “Entrenamiento GAN a gran escala para sistemas naturales síntesis de imágenes de alta fidelidad”, 28 de septiembre de 2018, <https://arxiv.org/abs/1809.11096>.

9 Patrick Esser et al., “Domar transformadores para síntesis de imágenes de alta resolución”, 17 de diciembre de 2020, <https://arxiv.org/abs/2012.09841>.

10 Aaron van den Oord et al., “Aprendizaje de representación discreta neuronal”, 2 de noviembre de 2017, <https://arxiv.org/abs/1711.00937v2>.

11 Anders Boesen Lindbo Larsen et al., “Autocodificación más allá de los píxeles utilizando una métrica de similitud aprendida”, 31 de diciembre de 2015, <https://arxiv.org/abs/1512.09300>.

12 Phillip Isola et al., “Traducción de imagen a imagen con redes adversarias condicionales”, 21 de noviembre de 2016, <https://arxiv.org/abs/1611.07004v3>.

13 Jun-Yan Zhu et al., “Traducción de imagen a imagen no emparejada usando CycleConsistent Adversarial Networks”, 30 de marzo de 2017, <https://arxiv.org/abs/1703.10593>.

14 Xianxu Hou et al., “auto codificador variacional consistente de características profundas”, 2 de octubre de 2016, <https://arxiv.org/abs/1610.00291>.

15 Jiahui Yu et al., “Modelado de imágenes cuantificadas por vectores con VQGAN”, 9 de octubre de 2021, <https://arxiv.org/abs/2110.04627>.

16 Alexey Dosovitskiy et al., “Una imagen vale 16 x 16 palabras: transformadores para el reconocimiento de imágenes a escala”, 22 de octubre de 2020, <https://arxiv.org/abs/2010.11929v2>.

17 Jiahui Yu y Jing Yu Koh, “Modelado de imágenes cuantificadas por vectores con VQGAN mejorado”, 18 de mayo de 2022,

<https://ai.googleblog.com/2022/05/vector-quantized-image-modelingwith.html>.

Capítulo 11. Generación de música

METAS DEL CAPÍTULO

En este capítulo podrás:

Comprender cómo podemos tratar la generación de música como un problema de predicción de secuencia, de modo que podamos aplicar modelos autorregresivos como Transformers.

Vea cómo analizar y tokenizar archivos MIDI usando el paquete music21 para crear un conjunto de entrenamiento.

Aprenda a utilizar la codificación posicional sinusoidal.

Entrene un Transformer generador de música, con múltiples entradas y salidas para manejar notas y duración.

Comprenda cómo manejar la música polifónica, incluida la tokenización de grillas y la tokenización basada en eventos.

Entrene un modelo MuseGAN para generar música multipista.

Utilice MuseGAN para ajustar diferentes propiedades de las barras generadas.

La composición musical es un proceso complejo y creativo que implica combinar diferentes elementos musicales como la melodía, la armonía, el ritmo y el timbre. Si bien esto se considera tradicionalmente como una actividad exclusivamente humana, los avances recientes han hecho posible generar música que sea agradable al oído y tenga una estructura a largo plazo.

Una de las técnicas más populares para la generación de música es el Transformer, ya que la música puede considerarse como un problema de predicción de secuencias. Estos modelos se han adaptado para generar música tratando las notas musicales como una secuencia de símbolos,

similar a las palabras de una oración. El modelo transformador aprende a predecir la siguiente nota de la secuencia basándose en las notas anteriores, lo que da como resultado una pieza musical generada.

MuseGAN adopta un enfoque totalmente diferente a la hora de generar música. A diferencia de Transformers, que genera música nota por nota, MuseGAN genera pistas musicales completas a la vez tratando la música como una imagen, que consta de un eje de tono y un eje de tiempo. Además, MuseGAN separa diferentes componentes musicales como acordes, estilo, melodía y ritmo para que puedan controlarse de forma independiente.

En este capítulo, aprenderemos cómo procesar datos musicales y aplicar Transformer y MuseGAN para generar música que sea estilísticamente similar a un conjunto de entrenamiento determinado.

Introducción

Para que una máquina pueda componer música que sea agradable a nuestros oídos, debe superar muchos de los mismos desafíos técnicos que vimos en el capítulo 9 en relación con el texto. En particular, nuestro modelo debe poder aprender y recrear la estructura secuencial de la música y poder elegir entre un conjunto discreto de posibilidades para notas posteriores.

Sin embargo, la generación de música presenta desafíos adicionales que no están presentes para la generación de texto, a saber, el tono y el ritmo. La música suele ser polifónica, es decir, hay varias corrientes de notas tocadas simultáneamente en diferentes instrumentos, que se combinan para crear armonías que son disonantes (chocantes) o consonantes (armoniosas).

La generación de texto solo requiere que manejemos un único flujo de texto, en contraste con los flujos paralelos de acordes que están presentes en la música.

Además, la generación de texto se puede manejar palabra por palabra.

A diferencia de los datos textuales, la música es un tapiz de sonidos entrelazados y de varias partes que no necesariamente se emiten al mismo

tiempo; gran parte del interés que surge de escuchar música está en la interacción entre los diferentes ritmos del conjunto. Por ejemplo, un guitarrista puede tocar una ráfaga de notas más rápidas mientras el pianista mantiene un acorde sostenido más largo. Por lo tanto, generar música nota a nota es complejo, porque muchas veces no queremos que todos los instrumentos cambien de nota simultáneamente.

Comenzaremos este capítulo simplificando el problema para centrarnos en la generación de música para una única línea musical (monofónica). Muchas de las técnicas del Capítulo 9 para la generación de texto también se pueden utilizar para la generación de música, ya que las dos tareas comparten muchos temas comunes. Comenzaremos entrenando a un Transformer para generar música al estilo de las Suites para violonchelo de J. S. Bach y vea cómo el mecanismo de atención permite al modelo centrarse en las notas anteriores para determinar la nota siguiente más natural. Luego abordaremos la tarea de generación de música polifónica y exploraremos cómo podemos implementar una arquitectura basada en GAN para crear música para múltiples voces.

Transformadores para la generación musical

El modelo que construiremos aquí es un transformador decodificador, inspirado en MuseNet de OpenAI, que también utiliza un decodificador Transformer (similar a GPT3) entrenado para predecir la siguiente nota dada una secuencia de notas anteriores.

En las tareas de generación de música, la longitud de la secuencia N crece a medida que avanza la música, y esto significa que la matriz de atención $N \times N$ para cada cabeza se vuelve caro de almacenar y calcular. Idealmente, no queremos recortar la secuencia de entrada a una cantidad corta de tokens, ya que nos gustaría que el modelo construyera la pieza alrededor de una estructura a largo plazo y repitiera motivos y frases de hace varios minutos, como lo haría un compositor humano.

Para abordar este problema, MuseNet utiliza una forma de transformador conocido como Transformador Sparse. Cada posición de salida en la matriz de atención solo calcula pesos para un subconjunto de posiciones de entrada, lo que reduce la complejidad computacional y la memoria

requerida para entrenar el modelo. Por lo tanto, MuseNet puede operar con total atención sobre 4.096 tokens y puede aprender estructuras a largo plazo y estructuras melódicas en una variedad de estilos. (Véase, por ejemplo, las grabaciones de Chopin y Mozart de OpenAI en Nube de sonido.)

Para ver cómo la continuación de una frase musical suele verse influenciada por notas de varios compases anteriores, eche un vistazo a los primeros compases del Preludio de la Suite para violonchelo n.^o 1 de Bach (Figura 11-1).

Figura 11-1. La apertura de la Suite para violonchelo n.^o 1 de Bach (Preludio)

BARRAS

Los compases (o compases) son pequeñas unidades de música que contienen un número pequeño y fijo de tiempos y están marcadas por líneas verticales que cruzan el pentagrama. Si puedes contar 1, 2, 1, 2 al son de una pieza musical, entonces hay dos tiempos en cada compás y probablemente estés escuchando una marcha. Si puedes contar 1, 2, 3, 1, 2, 3, entonces hay tres tiempos en cada compás y es posible que estés escuchando un vals.

¿Qué nota crees que viene después? Incluso si no tienes formación musical, es posible que puedas adivinar. Si tú dijiste Sol (G) (igual que la primera nota de la pieza), entonces estarías en lo cierto. ¿Cómo supiste esto? Es posible que haya podido ver que cada compás y medio compás comienza con la misma nota y haya utilizado esta información para fundamentar su decisión. Queremos que nuestro modelo pueda realizar el mismo truco; en particular, queremos que preste atención a una nota particular del medio compás anterior, cuando se registró el G bajo anterior. Un modelo basado en la atención como un

Transformer podrá incorporar esta mirada retrospectiva a largo plazo sin tener que mantener un estado oculto en muchas barras, como es el caso de una red neuronal recurrente.

Cualquiera que se dedique a la tarea de generar música debe tener primero un conocimiento básico de la teoría musical. En la siguiente sección, analizaremos los conocimientos esenciales necesarios para leer música y cómo podemos representarlos numéricamente para transformar la música en los datos de entrada necesarios para entrenar nuestro Transformer.

EJECUTANDO EL CÓDIGO PARA ESTE EJEMPLO

El código para este ejemplo se puede encontrar en el cuaderno de Jupyter ubicado en notebooks/11_music/01_transformer/transformer.ipynb en el repositorio de libros.

El conjunto de datos de la suite para violonchelo de Bach

El conjunto de datos sin procesar que usaremos es un conjunto de archivos MIDI para Cello Suites de J.S. Llevar una vida de soltero. Puede descargar el conjunto de datos ejecutando el script de descarga del conjunto de datos en el repositorio de libros, como se muestra en el Ejemplo 11-1. Esto guardará los archivos MIDI localmente en la carpeta /data.

Ejemplo 11-1. Descargando el conjunto de datos de las suites de cello de J.S. Bach

```
bash scripts/download_music_data.sh
```

Para ver y escuchar la música generada por el modelo, necesitará algún software que pueda producir notación musical. MuseScore es una gran herramienta para este propósito y se puede descargar de forma gratuita.

Análisis de archivos MIDI

Usaremos la biblioteca de Python music21 para cargar los archivos MIDI en Python para procesarlos. El ejemplo 11-2 muestra cómo cargar un archivo MIDI y visualizarlo (Figura 11-2), tanto como una partitura como como datos estructurados.

Figura 11-2. Notación musical

Ejemplo 11-2. Importar un archivo MIDI

```
import music21  
  
file = "/app/data/bach-cello/cs1-2all.mid"  
example_score = music21.converter.parse(file).chordify()
```

OCTAVAS

El número después del nombre de cada nota indica la octava en la que se encuentra la nota; dado que los nombres de las notas (A a G) se repiten, esto es necesario para identificar de forma única el tono de la nota. Por ejemplo, G2 está una octava por debajo de G3.

¡Ahora es el momento de convertir las partituras en algo que se parezca más a texto! Comenzamos recorriendo cada partitura y extrayendo la nota y la duración de cada elemento de la pieza en dos cadenas de texto separadas, con elementos separados por espacios. Codificamos la clave y el compás de la pieza como símbolos especiales, con duración cero.

MÚSICA MONOFÓNICA VERSUS POLIFÓNICA

En este primer ejemplo, trataremos la música como monofónica (una sola línea), tomando solo la nota más alta de cualquier acorde. A veces es posible que deseemos mantener las partes separadas para generar música de naturaleza polifónica. Esto presenta desafíos adicionales que abordaremos más adelante en este capítulo.

El resultado de este proceso se muestra en la Figura 11-3; compárela con la Figura 11-2 para que pueda ver cómo los datos musicales sin procesar se han transformado en las dos cuerdas.

Figura 11-3. Ejemplos de la cadena de texto de notas y la cadena de texto de duración, correspondientes a la Figura 11-2

Esto se parece mucho más a los datos de texto que hemos tratado anteriormente. Las palabras son las combinaciones nota-duración, y deberíamos intentar construir un modelo que prediga la siguiente nota y

duración, dada una secuencia de notas y duraciones anteriores. Una diferencia clave entre la generación de música y texto es que necesitamos construir un modelo que pueda manejar la predicción de nota y duración simultáneamente; es decir, hay dos flujos de información que debemos manejar, en comparación con los flujos de texto únicos que vimos. en el Capítulo 9.

Tokenización

Para crear el conjunto de datos que entrenará el modelo, primero debemos tokenizar cada nota y duración, exactamente como lo hicimos anteriormente para cada palabra en un corpus de texto. Podemos lograr esto usando una capa TextVectorization, aplicada a las notas y duraciones por separado, como se muestra en el Ejemplo 11-3.

```
Ejemplo 11-3. Tokenización de las notas y duraciones def  
create_dataset(elements): ds = (tf.data.Dataset.from_tensor_slices(elements).batch(BATCH_SIZE,  
drop_remainder = True).shuffle(1000)  
  
vectorize_layer = capas.TextVectorization(estandarizar = Ninguno,  
salida_mode="int") vectorize_layer.adapt(ds) vocab = vectorize_layer.get_vocabulary()  
return ds, vectorize_layer, vocab  
notes_seq_ds, notes_vectorize_layer, notes_vocab = create_dataset(notas)  
duraciones_seq_ds, duraciones_vectorize_layer, duraciones_vocab = create_dataset(duraciones)  
duraciones_seq_ds = tf.data.Dataset.zip((notes_seq_ds, duraciones_seq_ds))
```

El proceso completo de análisis y tokenización se muestra en la Figura 11-4.

Figura 11-4. Analizar los archivos MIDI y tokenizar las notas y duraciones

Creando el conjunto de entrenamiento

El último paso del preprocesamiento es crear el conjunto de entrenamiento que alimentaremos a nuestro Transformer.

Hacemos esto dividiendo las cadenas de notas y de duración en trozos de 50 elementos, utilizando una técnica de ventana deslizante. La salida es simplemente la ventana de entrada desplazada una nota, de modo que el transformador está entrenado para predecir la nota y la duración del elemento en un paso de tiempo hacia el futuro, dados los elementos anteriores en la ventana. En la Figura 11-5 se muestra un ejemplo de esto (usando una ventana deslizante de sólo cuatro elementos con fines de demostración).

Figura 11-5. Las entradas y salidas para el modelo musical Transformer; en este ejemplo, se utiliza una ventana deslizante de ancho 4 para crear fragmentos de entrada, que luego se desplazan un elemento para crear la salida de destino.

La arquitectura que usaremos para nuestro Transformer es la misma que usamos para la generación de texto en el Capítulo 9, con algunas diferencias clave.

Codificación de posición sinusoidal

En primer lugar, introduciremos un tipo diferente de codificación para las posiciones de los tokens. En el Capítulo 9 utilizamos una sencilla capa de incrustación para codificar la posición de cada token, asignando efectivamente cada posición entera a un vector distinto que fue aprendido por el modelo. Por lo tanto, necesitábamos definir una longitud máxima (N) que podría tener la secuencia y entrenar en esta longitud de secuencia. La desventaja de este enfoque es que es imposible extrapolar a secuencias que sean más largas que esta longitud máxima. Tendrías que recortar la entrada a los últimos N tokens, lo cual no es ideal si intentas generar contenido de formato largo.

Para evitar este problema, podemos pasar a utilizar un tipo diferente de incrustación llamada incrustación de posición sinusoidal. Esto es similar a la incorporación que utilizamos en el Capítulo 8 para codificar las variaciones de ruido del modelo de difusión. Específicamente, la siguiente función se utiliza para convertir la posición de la palabra (pos) en la secuencia de entrada en un vector único de longitud d :

pos

P Epos,2i

$\sin(10,000)$

2i/d

pos

P Épos,2i+1

$\cos(10,000)$

(2i+1)/día

Para i pequeña, la longitud de onda de esta función es corta y, por lo tanto, el valor de la función cambia rápidamente a lo largo del eje de posición. Los valores más grandes de i crean una longitud de onda más larga.

Por tanto, cada posición tiene su propia codificación única, que es una combinación específica de las diferentes longitudes de onda.

CONSEJO

Observe que esta incrustación está definida para todos los valores de posición posibles.

Es una función determinista (es decir, el modelo no la aprende) que utiliza funciones trigonométricas para definir una codificación única para cada posición posible.

El módulo Keras NLP tiene una capa incorporada que implementa esta incorporación para nosotros; por lo tanto, podemos definir nuestra capa TokenAndPositionEmbedding como se muestra en el Ejemplo 11-4.

Ejemplo 11-4. Tokenizando las notas y duraciones

```

class TokenAndPositionEmbedding(layers.Layer):
    def __init__(self, vocab_size, embed_dim):
        super(TokenAndPositionEmbedding, self).__init__()
        self.vocab_size = vocab_size
        self.embed_dim = embed_dim
        self.token_emb = Layers.Embedding (input_dim=vocab_size,
output_dim=embed_dim)
        self.pos_emb = keras_nlp.layers.SinePositionEncoding()

    def call(self, x):
        incrustación = self.token_emb(x)
        posiciones = self.pos_emb(incrustación)
        return incrustación + posiciones

```

La Figura 11-6 muestra cómo se agregan las dos incrustaciones (token y posición) para producir la incrustación general de la secuencia.

Figura 11-6. La capa TokenAndPositionEmbedding agrega las incrustaciones de tokens a las incrustaciones de posición sinusoidal para producir la incrustación general de la secuencia.

Múltiples entradas y salidas

Ahora tenemos dos flujos de entrada (notas y duraciones) y dos flujos de salida (notas y duraciones previstas). Por lo tanto, necesitamos adaptar la arquitectura de nuestro transformador para atender esto.

Hay muchas maneras de manejar el doble flujo de entradas.

Podríamos crear tokens que representen cada par nota-duración y luego tratar la secuencia como un único flujo de tokens. Sin embargo, esto tiene la desventaja de no poder representar pares nota-duración que no se hayan visto en el conjunto de entrenamiento (por ejemplo, es posible que hayamos visto una nota G#2 y una duración de 1/3 de forma independiente, pero nunca juntas, entonces no habría ningún token para G#2:1/3).

En su lugar, elegimos incrustar los tokens de nota y duración por separado y luego usar una capa de concatenación para crear una representación única de la entrada que puede ser utilizada por el bloque Transformer descendente. De manera similar, la salida del bloque Transformer se pasa a dos capas densas separadas, que representan las probabilidades de duración

y nota predichas. La arquitectura general se muestra en la Figura 11-7. Las formas de salida de las capas se muestran con un tamaño de lote b y una longitud de secuencia l .

Figura 11-7. La arquitectura del Transformer generador de música.

Un enfoque alternativo sería intercalar los tokens de nota y duración en un único flujo de entrada y dejar que el modelo aprenda que la salida debe ser un flujo único donde los tokens de nota y duración se alternan. Esto viene con la complejidad adicional de garantizar que la salida aún pueda analizarse cuando el modelo aún no ha aprendido a intercalar los tokens correctamente.

CONSEJO

No existe una forma correcta o incorrecta de diseñar tu modelo; parte de la diversión es experimentar con diferentes configuraciones y ver cuál funciona mejor para ti.

Análisis del transformador generador de música

Comenzaremos generando algo de música desde cero, sembrando la red con un token de nota INICIO y un token de duración 0.0 (es decir, le estamos diciendo al modelo que asuma que está comenzando desde el principio de la pieza). Luego podemos generar un pasaje musical usando la misma técnica iterativa que usamos en el Capítulo 9 para generar secuencias de texto, de la siguiente manera:

1. Dada la secuencia actual (de notas y duraciones), el modelo predice dos distribuciones, una para la siguiente nota y otra para la siguiente duración.
1. Tomamos muestras de ambas distribuciones, usando un parámetro de temperatura para controlar cuánta variación nos gustaría en el proceso de muestreo.
1. La nota y la duración elegidas se añaden a las respectivas secuencias de entrada.

1. El proceso se repite con las nuevas secuencias de entrada para tantos elementos como queramos generar.

La Figura 11-8 muestra ejemplos de música generada desde cero por el modelo en varias épocas del proceso de entrenamiento. Usamos una temperatura de 0,5 para las notas y duraciones.

Figura 11-8. Algunos ejemplos de pasajes generados por el modelo cuando se siembra solo con un token de nota de INICIO y un token de duración de 0.0

La mayor parte de nuestro análisis en esta sección se centrará en las predicciones de notas, más que en las duraciones, como en el caso de las suites de violonchelo de Bach con complejidades armónicas son más difíciles de capturar y, por lo tanto, más dignas de investigación.

Sin embargo, también puedes aplicar el mismo análisis a las predicciones rítmicas del modelo, lo que puede ser particularmente relevante para otros estilos de música que podrías usar para entrenar este modelo (como una pista de batería).

Hay varios puntos a tener en cuenta sobre los pasajes generados en la Figura 11-8. En primer lugar, observa cómo la música se va volviendo más sofisticada a medida que avanza el entrenamiento. Para empezar, el modelo va a lo seguro apegándose al mismo grupo de notas y ritmos. En la época 10, el modelo ha comenzado a generar pequeñas series de notas, y en la época 20 está produciendo ritmos interesantes y está firmemente establecido en una tonalidad establecida (E \flat mayor).

En segundo lugar, podemos analizar la distribución de notas a lo largo del tiempo trazando la distribución prevista en cada paso de tiempo como un mapa de calor. La Figura 11-9 muestra este mapa de calor para el ejemplo de la época 20 en la Figura 11-8.

Figura 11-9. La distribución de posibles siguientes notas a lo largo del tiempo (en la época 20): cuanto más oscuro es el cuadrado, más seguro es el modelo de que la siguiente nota está en este tono

Un punto interesante a tener en cuenta aquí es que el modelo ha aprendido claramente qué notas pertenecen a claves particulares, ya que hay espacios en la distribución en las notas que no pertenecen a la clave. Por ejemplo, hay un espacio gris a lo largo de la fila de la nota 54 (correspondiente a G \flat /F \sharp). Es muy poco probable que esta nota aparezca en una pieza musical en clave de mi \flat mayor. El modelo establece la clave al principio del proceso de generación y, a medida que avanza la pieza, el modelo elige notas que tienen más probabilidades de aparecer en esa clave al prestar atención al token que la representa.

También vale la pena señalar que el modelo ha aprendido el estilo característico de Bach de bajar a una nota baja en el violonchelo para terminar una frase y volver a subir para comenzar la siguiente. Mira como alrededor de la nota 20, la frase termina en Mi \flat bajo: es común en las Suites para violonchelo de Bach volver a un rango más alto y sonoro del instrumento para el comienzo de la siguiente frase, que es exactamente lo que predice el modelo. Hay una gran brecha gris entre el Mi \flat bajo (tono número 39) y la siguiente nota, que se predice que estará alrededor del tono número 50, en lugar de continuar retumbando en las profundidades del instrumento.

Por último, debemos comprobar si nuestro mecanismo de atención funciona como se esperaba. El eje horizontal de la Figura 11-10 muestra la secuencia de notas generada; el eje vertical muestra hacia dónde se dirigió la atención de la red al predecir cada nota a lo largo del eje horizontal. El color de cada cuadrado muestra el peso máximo de atención en todas las cabezas en cada punto de la secuencia generada. Cuanto más oscuro es el cuadrado, más atención se presta a esta posición en la secuencia. Por simplicidad, sólo mostramos las notas en este diagrama, pero la red también se ocupa de las duraciones de cada nota.

Podemos ver que para la armadura inicial, el compás y el descanso, la red optó por poner casi toda su atención en el token START. Esto tiene sentido, ya que estos artefactos siempre aparecen al comienzo de una pieza musical; una vez que las notas comienzan a fluir, el token START esencialmente deja de ser atendido.

A medida que avanzamos más allá de las pocas notas iniciales, podemos ver que la red pone más atención en aproximadamente las últimas dos o cuatro notas y rara vez otorga un peso significativo a las notas de hace más de cuatro notas. Nuevamente, esto tiene sentido; Probablemente haya suficiente información contenida en las cuatro notas anteriores para comprender cómo podría continuar la frase. Además, algunas notas se remontan con más fuerza a la armadura de re menor, por ejemplo, E3 (séptima nota de la pieza) y B-2 (B \flat –14^a nota de la pieza). Esto es fascinante, porque estas son las notas exactas que dependen de la clave de re menor para aliviar cualquier ambigüedad. La red debe volver a mirar la armadura para saber que hay un B \flat en la armadura (en lugar de un B natural) pero no hay un E \flat en la armadura (en su lugar se debe usar E natural).

Figura 11-10. El color de cada cuadrado en la matriz indica la cantidad de atención prestada a cada posición en el eje vertical, en el punto de predecir la nota en el eje horizontal.

También hay ejemplos en los que la red ha optado por ignorar una determinada nota o permanecer cerca, ya que no añade ninguna información adicional a su comprensión de la frase. Por ejemplo, la penúltima nota (A2) no está particularmente atenta a las tres notas de B-2, pero está un poco más atenta a las cuatro notas de A2. Es más interesante para el modelo mirar el La2 que cae en el tiempo, en lugar del Si-2 fuera del tiempo, que es solo una nota pasajera.

Recuerde que no le hemos dicho nada al modelo sobre qué notas están relacionadas o qué notas pertenecen a qué armaduras; lo ha resuelto por sí mismo simplemente estudiando la música de J.S. Llevar una vida de soltero.

Tokenización de la música polifónica

El Transformer que hemos estado explorando en esta sección funciona bien para música de una sola línea (monofónica), pero ¿podría adaptarse a música de varias líneas (polifónica)?

El desafío radica en cómo representar las diferentes líneas musicales como una única secuencia de fichas. En la sección anterior decidimos dividir las notas y duraciones de las notas en dos entradas y salidas distintas de la red, pero también vimos que podríamos haber intercalado estos tokens en una sola secuencia. Podemos utilizar la misma idea para manejar la música polifónica. Aquí se introducirán dos enfoques diferentes: tokenización de red y tokenización basada en eventos, como se analiza en el artículo de 2018 “Transformador Musical: Generando música con estructura a largo plazo” [1]

Tokenización de rejilla

Considere los dos compases de música de J.S. Coral de Bach en la Figura 11-11. Hay cuatro partes distintas (soprano [S], alto [A], tenor [T], bajo [B]), escritas en diferentes pentagramas.

Figura 11-11. Los dos primeros compases de un J.S. coral de bach

Podemos imaginarnos dibujando esta música en una cuadrícula, donde el eje y representa el tono de la nota y el eje x representa el número de semicorcheas (semicorcheas) que han pasado desde el comienzo de la pieza. Si el cuadrado de la cuadrícula está lleno, entonces hay una nota sonando en ese momento.

Las cuatro partes están dibujadas en la misma cuadrícula. Esta cuadrícula se conoce como rollo de piano porque se asemeja a un rollo de papel físico con agujeros perforados, que se utilizaba como mecanismo de grabación antes de que se inventaran los sistemas digitales.

Podemos serializar la cuadrícula en una secuencia de tokens moviéndonos primero a través de las cuatro voces y luego a lo largo de los pasos de tiempo en secuencia. Esto produce una secuencia de tokens.

S₁, A₁, T₁, B₁, S₂, A₂, T₂, B₂, ...,

donde el subíndice denota el paso de tiempo, como se muestra en la Figura 11-12.

Figura 11-12. Creando la tokenización de la cuadrícula para los dos primeros compases del coral de Bach

Luego entrenaríamos a nuestro Transformer en esta secuencia de tokens, para predecir el siguiente token dados los tokens anteriores.

Podemos decodificar la secuencia generada nuevamente en una estructura de cuadrícula haciendo retroceder la secuencia a lo largo del tiempo en grupos de cuatro notas (una para cada voz). Esta técnica funciona sorprendentemente bien, a pesar de que la misma nota a menudo se divide en varias fichas con fichas de otras voces en el medio.

Sin embargo, existen algunas desventajas. En primer lugar, observe que el modelo no tiene forma de distinguir entre una nota larga y dos notas adyacentes más cortas del mismo tono. Esto se debe a que la tokenización no codifica explícitamente la duración de las notas, solo si una nota está presente en cada paso de tiempo.

En segundo lugar, este método requiere que la música tenga un ritmo regular que sea divisible en fragmentos de tamaño razonable. Por ejemplo, con el sistema actual, no podemos codificar trillizos (un grupo de tres notas tocadas en un solo tiempo).

Podríamos dividir la música en 12 pasos por negra (entrepierna) en lugar de 4, lo que triplicaría la cantidad de fichas necesarias para representar el mismo pasaje de música, lo que agregaría gastos generales al proceso de entrenamiento y afectaría la capacidad de retrospectiva del modelo.

Por último, no es obvio cómo podríamos agregar otros componentes a la tokenización, como la dinámica (qué tan alta o baja es la música en cada parte) o cambios de tempo.

Estamos atrapados en la estructura de cuadrícula bidimensional del rollo de piano, que proporciona una manera conveniente de representar el tono y la sincronización, pero no necesariamente una manera fácil de incorporar otros componentes que hagan que la música sea interesante de escuchar.

Tokenización basada en eventos

Un enfoque más flexible es utilizar la tokenización basada en eventos. Esto puede considerarse como un vocabulario que describe literalmente cómo se crea la música como una secuencia de eventos, utilizando un rico conjunto de elementos.

Por ejemplo, en la Figura 11-13, utilizamos tres tipos de tokens:

NOTA_ON (empieza a tocar una nota de un tono determinado)

NOTA_OFF (deja de tocar una nota de un tono determinado)

TIME_SHIFT (avanza en el tiempo un paso determinado)

Este vocabulario se puede utilizar para crear una secuencia que describa la construcción de la música como un conjunto de instrucciones.

Figura 11-13. Una tokenización de evento para el primer compás del coral de Bach

Podríamos incorporar fácilmente otros tipos de tokens a este vocabulario, para representar cambios dinámicos y de tempo para notas posteriores. Este método también proporciona una manera de generar tresillos con un fondo de negras, separando las notas de los tresillos con TIME_SHIFT<0.33>

fichas. En general, es un marco más expresivo para la tokenización, aunque también es potencialmente más complejo para el Transformer aprender patrones inherentes en la música del conjunto de entrenamiento, ya que, por definición, está menos estructurado que el método de cuadrícula.

CONSEJO

Le animo a que intente implementar estas técnicas polifónicas y entrenar a un Transformer en el nuevo conjunto de datos tokenizados utilizando todo el conocimiento que ha adquirido hasta ahora en este libro. También recomendaría consultar nuestra guía del Dr. Tristan Behrens para la investigación de la generación musical, disponible en GitHub, que proporciona una descripción general completa de diferentes artículos sobre el tema de la generación musical mediante el aprendizaje profundo.

En la siguiente sección adoptaremos un enfoque completamente diferente para la generación de música, utilizando GAN.

musaGAN

Quizás hayas pensado que el rollo de piano que se muestra en la figura 11-12 se parece un poco a una obra de arte moderno. Esto plantea la pregunta: ¿podríamos de hecho tratar este rollo de piano como una imagen y utilizar métodos de generación de imágenes en lugar de técnicas de generación de secuencias?

Como veremos, la respuesta a esta pregunta es sí, podemos tratar la generación de música directamente como un problema de generación de imágenes. Esto significa que en lugar de utilizar Transformers, podemos aplicar las mismas técnicas basadas en convolución que funcionan tan bien para problemas de generación de imágenes; en particular, GAN.

MuseGAN se presentó en el artículo de 2017 “MuseGAN: Redes adversarias generativas secuenciales de múltiples vías para la generación y acompañamiento de música simbólica”.² Los autores muestran cómo es posible entrenar un modelo para generar música polifónica, multipista y multibarra a través de un marco GAN novedoso. Además, muestran cómo, al dividir las responsabilidades de los vectores de ruido que alimentan el generador, son capaces de mantener un control detallado sobre las características temporales y basadas en pistas de alto nivel de la música.

Comencemos presentando al conjunto de datos corales de J.S. Bach.

EJECUTANDO EL CÓDIGO PARA ESTE EJEMPLO

El código de este ejemplo se puede encontrar en el cuaderno de Jupyter ubicado en notebooks/11_music/02_musegan/musegan.ipynb en el repositorio de libros.

El conjunto de datos corales de Bach

Para comenzar este proyecto, primero deberá descargar los archivos MIDI que usaremos para entrenar MuseGAN. Usaremos un conjunto de datos de

229 corales de J. S. Bach a cuatro voces.

Puede descargar el conjunto de datos ejecutando el script de descarga del conjunto de datos corales de Bach en el repositorio de libros, como se muestra en el Ejemplo 11-5. Esto guardará los archivos MIDI localmente en la carpeta /data.

Ejemplo 11-5. Descarga del conjunto de datos corales de Bach

```
bash scripts/download_bach_chorale_data.sh
```

El conjunto de datos consta de una matriz de cuatro números para cada paso de tiempo: los tonos de notas MIDI de cada una de las cuatro voces.

Un paso de tiempo en este conjunto de datos es igual a una semicorchea (una semicorchea). Entonces, por ejemplo, en un solo compás de 4 tiempos de cuarto (entrepierna), hay 16 pasos de tiempo. El conjunto de datos se divide automáticamente en conjuntos de entrenamiento, validación y prueba. Usaremos el conjunto de datos del tren para entrenar MuseGAN.

Para comenzar, necesitamos darle a los datos la forma correcta para alimentar la GAN. En este ejemplo generaremos dos compases de música, por lo que extraeremos solo los dos primeros compases de cada coral. Cada compás consta de 16 pasos de tiempo y hay 84 tonos potenciales en las 4 voces.

CONSEJO

A partir de ahora nos referiremos a las voces como pistas, para mantener la terminología acorde con el documento original.

Por tanto, los datos transformados tendrán la siguiente forma:
[BATCH_SIZE, N_BARS, N_STEPS_PER_BAR, N_PITCHES, N_TRACKS]

dónde:

BATCH_SIZE = 64

N_BARAS = 2

N_STEPS_PER_BAR = 16

N_PITCHES = 84

N_PIESTAS = 4

Para darle esta forma a los datos, codificamos en caliente los números de tono en un vector de longitud 84 y dividimos cada secuencia de notas en dos compases de 16 pasos de tiempo cada uno. Aquí asumimos que cada coral en el conjunto de datos tiene cuatro tiempos en cada compás, lo cual es razonable, e incluso si este no fuera el caso, no afectaría negativamente el entrenamiento del modelo.

La Figura 11-14 muestra cómo dos barras de datos sin procesar se convierten en el conjunto de datos de pianola transformado que usaremos para entrenar la GAN.

Figura 11-14. Procesar dos barras de datos sin procesar en datos de pianola que podemos usar para entrenar la GAN

El generador MuseGAN

Como todas las GAN, MuseGAN consta de un generador y un crítico. El generador intenta engañar al crítico con sus creaciones musicales, y el crítico intenta evitar que esto suceda asegurándose de que sea capaz de distinguir entre los corales de Bach falsificados del generador y los reales.

En lo que MuseGAN se diferencia es en el hecho de que el generador no solo acepta un único vector de ruido como entrada, sino que tiene cuatro entradas separadas, que corresponden a cuatro características diferentes de la música: acordes, estilo, melodía y ritmo. Al manipular cada una de estas entradas de forma independiente, podemos cambiar las propiedades de alto nivel de la música generada.

En la Figura 11-15 se muestra una vista de alto nivel del generador.

Figura 11-15. Diagrama de alto nivel del generador MuseGAN

El diagrama muestra cómo las entradas de acordes y melodías pasan primero a través de una red temporal que genera un tensor con una de las dimensiones igual al número de compases que se generarán. Las entradas de estilo y ritmo no se estiran temporalmente de esta manera, ya que permanecen constantes a lo largo de la pieza.

Luego, para generar un compás particular para una pista particular, las salidas relevantes de las partes de acordes, estilo, melodía y ritmo de la red se concatenan para formar un vector más largo. Luego, esto se pasa a un generador de barras, que finalmente genera la barra especificada para la pista especificada.

Al concatenar los compases generados para todas las pistas, creamos una partitura que el crítico puede comparar con partituras reales.

Primero echemos un vistazo a cómo construir una red temporal.

La red temporal

El trabajo de una red temporal (una red neuronal que consta de capas convolucionales transpuestas) es transformar un único vector de ruido de entrada de longitud $Z_DIM = 32$ en un vector de ruido diferente para cada barra (también de longitud 32). El código Keras para construir esto se muestra en el Ejemplo 11-6.

Ejemplo 11-6. Construyendo la red temporal

```
def conv_t(x, f, k, s, a, p, bn):
    x = Layers.Conv2DTranspose( filters = f , kernel_size = k ,
padding = p , strides = s , kernel_initializer = initializer )
(x)
    if bn:
        x = capas.BatchNormalization(momento = 0,9)(x)

x = capas.Activación(a)(x) return x def TemporalNetwork(): input_layer =
capas.Input(forma=(Z_DIM,), nombre='temporal_input') x =
capas.Reshape([1,1,Z_DIM] )(capa_entrada) x = conv_t( x, f=1024, k=
(2,1), s=(1,1), a = 'relu', p = 'válido', bn =
```

Verdadero $x = \text{conv_t}(x, f=Z_DIM, k=(N_BARS - 1, 1), s=(1, 1), a = \text{'relu}', p = \text{'válido}'$, $bn = \text{Verdadero}$ capa_salida = capas.Reformar([N_BARS, Z_DIM])(x) modelos de retorno. Modelo (capa_entrada, capa_salida)

La entrada a la red temporal es un vector de longitud 32 (Z_DIM).

Transformamos este vector en un tensor de 1×1 con 32 canales, de modo que podamos aplicarle operaciones de transposición convolucionales 2D.

Aplicamos capas Conv2DTranspose para expandir el tamaño del tensor a lo largo de un eje, de modo que tenga la misma longitud que N_BARS.

Eliminamos la dimensión extra innecesaria con una capa reformadora.

La razón por la que utilizamos operaciones convolucionales en lugar de requerir dos vectores independientes en la red es porque nos gustaría que la red aprenda cómo una barra debe seguir a otra de manera consistente. El uso de una red neuronal para expandir el vector de entrada a lo largo del eje temporal significa que el modelo tiene la oportunidad de aprender cómo fluye la música a través de los compases, en lugar de tratar cada compás como completamente independiente del anterior.

Acordes, estilo, melodía y ritmo.

Ahora echemos un vistazo más de cerca a las cuatro entradas diferentes que alimentan el generador:

Acordes

La entrada de acordes es un único vector de ruido de longitud Z_DIM.

El trabajo de este vector es controlar la progresión general de la música a lo largo del tiempo, compartida entre pistas, por lo que utilizamos un TemporalNetwork para transformar este único vector en un vector latente diferente para cada barra. Tenga en cuenta que, si bien llamamos a esta entrada acordes, realmente podría controlar cualquier aspecto de la música que cambie por compás, como el estilo rítmico general, sin ser específico de ninguna pista en particular.

Estilo

La entrada de estilo también es un vector de longitud Z_DIM. Esto se traslada sin transformación, por lo que es el mismo en todos los compases y pistas. Se puede considerar como el vector que controla el estilo general de la pieza (es decir, afecta a todos los compases y pistas de manera consistente).

Melodía

La entrada de melodía es una matriz de forma [N_TRACKS, Z_DIM], es decir, proporcionamos al modelo un vector de ruido aleatorio de longitud Z_DIM para cada pista.

Cada uno de estos vectores pasa a través de una TemporalNetwork de ruta específica, donde los pesos no se comparten entre pistas. La salida es un vector de longitud Z_DIM para cada compás de cada pista. Por lo tanto, el modelo puede utilizar estos vectores de entrada para ajustar el contenido de cada barra y pista de forma independiente.

Ranura

La entrada del groove también es una matriz de formas [N_TRACKS, Z_DIM]: un vector de ruido aleatorio de longitud Z_DIM para cada pista. A diferencia de la entrada de melodía, estos vectores no pasan a través de la red temporal sino que pasan directamente, al igual que el vector de estilo.

Por lo tanto, cada vector de ritmo afectará las propiedades generales de una pista, en todos los compases.

Podemos resumir las responsabilidades de cada componente del generador MuseGAN como se muestra en la Tabla 11-1.

¿La salida difiere entre compases?

Estilo

Ranura

Acordes

Melodía

¿La salida difiere entre las piezas?

La última pieza del generador MuseGAN es el generador de barras; veamos cómo podemos usarlo para unir las salidas de los componentes de acordes, estilos, melodías y ritmos.

El generador de compases

El generador de compases recibe cuatro vectores latentes, uno de cada uno de los componentes de acorde, estilo, melodía y ritmo.

Estos se concatenan para producir un vector de longitud $4 * Z_DIM$ como entrada. El resultado es una representación de pianola de una sola barra para una sola pista, es decir, un tensor de forma $[1, n_steps_per_bar, n_pitches, 1]$.

El generador de barras es solo una red neuronal que utiliza capas de transposición convolucionales para expandir las dimensiones de tiempo y tono del vector de entrada. Creamos un generador de barras para cada pista y los pesos no se comparten entre pistas. El código Keras para construir un BarGenerator se proporciona en el Ejemplo 11-7.

Ejemplo 11-7. Construyendo BarGenerator
def BarGenerator():
 input_layer = capas.Input(forma=(Z_DIM * 4,), nombre='bar_generator_input')
 x = capas.Dense(1024)(input_layer)
 x = capas.BatchNormalization(momentum=0.9)(x)
 x = capas.Activación('relu')(x)
 x = capas.Reshape([2,1,512])(x)
 x = conv_t(x, f=512, k=(2,1), s=(2,1), a='relu', p='mismo', bn=True)
 x = conv_t(x, f=256, k=(2,1), s=(2,1), a='relu', p='igual', bn=True)
 x = conv_t(x, f=256, k=(2,1), s=(2,1), a='relu', p='igual', bn=True)
 x = conv_t(x, f=256, k=(1,7), s=(1,7), a='relu', p='igual', bn=True)
 x = conv_t(x, f=1, k=(1,12), s=(1,12), a='tanh', p='mismo', bn=False)
 capa_salida = capas.Reformar([1, N_STEPS_PER_BAR, N_PITCHES, 1])
(x) modelos de retorno. Modelo (capa_entrada, capa_salida)

La entrada al generador de barras es un vector de longitud 4 * Z_DIM.

Después de pasarlo por una capa Dense, remodelamos el tensor para prepararlo para las operaciones de transposición convolucional.

Primero expandimos el tensor a lo largo del eje de paso de tiempo... ...luego a lo largo del eje de tono.

A la capa final se le aplica una activación tanh, ya que usaremos un WGAN-GP (que requiere activación de salida tanh) para entrenar la red.

Se reforma el tensor para agregar dos dimensiones adicionales de tamaño 1, para prepararlo para la concatenación con otras barras y pistas.

Poniendolo todo junto

En última instancia, el generador MuseGAN toma los cuatro tensores de ruido de entrada (acordes, estilo, melodía y ritmo) y los convierte en una partitura multipista y multibarra. El código Keras para construir el generador MuseGAN se proporciona en el Ejemplo 11-8.

Ejemplo 11-8. Construyendo el generador MuseGAN

```
def Generator():
    chords_input = layers.Input(form=(Z_DIM, ),
name='chords_input')
    style_input = layers.Input(form=(Z_DIM, ), name='style_input')
    melody_input = layers.Input(form=(N_TRACKS, Z_DIM), name
='melody_input')
    groove_input = layers.Input(form=(N_TRACKS, Z_DIM),
name='groove_input')
    chords_tempNetwork = TemporalNetwork()
    chords_over_time = chords_tempNetwork(chords_input)
    melody_over_time = [None] * N_TRACKS
    melody_tempNetwork = [None] * N_TRACKS
    for track in range(N_TRACKS):
        melody_tempNetwork[track] = TemporalNetwork()
        melody_track = Layers.Lambda(lambda x, track = track:
x[:,track,:]) ( melody_input
```

```

melody_over_time[track] = melody_tempNetwork[track]
(melody_track)
barGen = [None] * N_TRACKS
for track in range(N_TRACKS):
    barGen[track] = BarGenerator()
    bars_output = [None] * N_BARS
    c = [None] * N_BARS
    for bar in range(N_BARS):
        track_output = [None] * N_TRACKS
        c[bar] = layers.Lambda(lambda x, bar = bar: x[:,bar,:])
(chords_over_time)
s = style_input
for track in range(N_TRACKS):
    m = layers.Lambda(lambda x, barra = barra:
x[:,bar,:])( melodía_sobre_tiempo[pista]
                g = capas.Lambda(lambda x, pista = pista:
x[:,pista,:])( groove_input
                    z_input = layers.concatenate( eje = 1, nombre =
'total_input_bar_{}track{}'.format(bar, track) )
([c[bar],s,m,g]) track_output[track] = barGentrack
bars_output[bar] = capas.concatenate(eje = -1) (pista_salida)
generador_salida = layers.concatenate(eje = 1, nombre
= 'concat_bars')( bares_salida devuelve modelos.Model(
[chords_input, style_input, melody_input, groove_input],
generador_output generador ()
```

Definir las entradas al generador.

Pasar la entrada de acordes a través de la red temporal.

Pasar la entrada de melodía a través de la red temporal.

Crea una red generadora de barras independiente para cada pista.

Recorre las pistas y los compases, creando un compás generado para cada combinación.

Concatena todo para formar un único tensor de salida.

El modelo MuseGAN toma cuatro tensores de ruido distintos como entrada y genera una puntuación multipista y multibarra generada.

El crítico de MuseGAN

En comparación con el generador, la arquitectura crítica es mucho más sencilla (como suele ser el caso con GAN).

El crítico intenta distinguir las partituras multipistas y multibarras creadas por el generador de extractos reales de los corales de Bach. Es una red neuronal convolucional, que consta principalmente de capas Conv3D que colapsan la puntuación en una única predicción de salida.

CAPAS CONV3D

Hasta ahora en este libro, solo hemos trabajado con capas Conv2D, aplicables a imágenes de entrada tridimensionales (ancho, alto, canales). Aquí tenemos que usar capas Conv3D, que son análogas a las capas Conv2D pero aceptan tensores de entrada de cuatro dimensiones (n_bars, n_steps_per_bar, n_pitches, n_tracks).

No utilizamos capas de normalización por lotes en la crítica, ya que usaremos el marco WGAN-GP para entrenar la GAN, que prohíbe esto.

El código Keras para construir la crítica se muestra en el Ejemplo 11-9.

```
Ejemplo 11-9. Construyendo el crítico de MuseGAN
def conv(x, f, k, s, p):
    x = Layers.Conv3D(filters = f, kernel_size = k, padding = p, strides = s,
                      kernel_initializer = inicializador)(x)
    x = Layers.LeakyReLU()(x)
    return x
def Critic():
    entrada_critica = capas.Input(shape=(N_BARS,
                                           N_STEPS_PER_BAR,
                                           N_PITCHES,
                                           N_TRACKS),
                                    name='entrada_critica')
    x = entrada_critica
    x = conv(x, f=128, k = (2,1,1), s = (1,1,1), p = 'válido')
    x = conv(x, f=128, k = (N_BARS - 1,1,1), s = (1,1,1), p = 'válido')
    x = conv(x, f=128, k = (1,1,12), s = (1,1,12), p = 'igual')
    x = conv(x, f=128, k = (1,1,7), s = (1,1,7), p = 'igual')
    x = conv(x, f=128, k = (1,2,1), s = (1,2,1), p = 'igual')
    x = conv(x, f=128, k = (1,4,1), s = (1,2,1), p = 'igual')
    x = conv(x, f=512, k = (1,3,1), s = (1,2,1), p = 'igual')
    x = capas.Flatten()(x)
    x = capas.Dense(1024, kernel_initializer = inicializador)(x)
    x = capas.LeakyReLU()(x)
    salida_critica = capas.Dense(1,
```

activación = Ninguna, kernel_initializer = inicializador (x) devuelve modelos. Modelo (entrada_crítica, salida_crítica) crítico = Crítico ()

La entrada al crítico es una serie de partituras multipista y multibarra, cada una con la forma [N_BARS, N_STEPS_PER_BAR, N_PITCHES, N_TRACKS].

Primero, colapsamos el tensor a lo largo del eje de la barra. Aplicamos capas Conv3D en todo el crítico mientras trabajamos con tensores 4D.

A continuación, colapsamos el tensor a lo largo del eje de tono.

Finalmente, colapsamos el tensor a lo largo del eje de pasos de tiempo.

La salida es una capa Dense con una sola unidad y sin función de activación, como lo requiere el marco WGAN-GP.

Análisis de MuseGAN

Podemos realizar algunos experimentos con nuestro MuseGAN generando una puntuación y luego ajustando algunos de los parámetros de ruido de entrada para ver el efecto en la salida.

La salida del generador es una matriz de valores en el rango $[-1, 1]$ (debido a la función de activación tanh de la capa final). Para convertir esto en una sola nota para cada pista, elegimos la nota con el valor máximo entre los 84 tonos para cada paso de tiempo. En el artículo original de MuseGAN, los autores utilizan un umbral de 0, ya que cada pista puede contener varias notas; sin embargo, en esta configuración podemos simplemente tomar el máximo para garantizar exactamente una nota por paso de tiempo por pista, como es el caso de los corales de Bach.

La Figura 11-16 muestra una puntuación generada por el modelo a partir de vectores de ruido aleatorios distribuidos normalmente (arriba a la izquierda). Podemos encontrar la puntuación más cercana en el conjunto de datos (por distancia euclídea) y verifique que nuestra partitura generada no sea una copia de una pieza musical que ya existe en el conjunto de datos; la partitura más cercana se muestra justo debajo y podemos ver que no se parece a nuestra partitura generada.

Figura 11-16. Ejemplo de una puntuación predicha de MuseGAN, que muestra la puntuación real más cercana en los datos de entrenamiento y cómo la puntuación generada se ve afectada al cambiar el ruido de entrada.

Ahora juguemos con el ruido de entrada para modificar nuestra partitura generada. Primero, podemos intentar cambiar el vector de ruido de las cuerdas; la partitura inferior izquierda de la Figura 11-16 muestra el resultado. Podemos ver que cada pista ha cambiado, como se esperaba, y también que las dos barras presentan propiedades diferentes. En el segundo compás, la línea de base es más dinámica y la línea superior tiene un tono más alto que en el primer compás. Esto se debe a que los vectores latentes que afectan a las dos barras son diferentes, ya que el vector de cuerda de entrada pasó a través de una red temporal.

Cuando cambiamos el vector de estilo (arriba a la derecha), ambas barras cambian de forma similar. Todo el pasaje ha cambiado de estilo con respecto a la partitura generada original, de manera consistente (es decir, se usa el mismo vector latente para ajustar todas las pistas y compases).

También podemos alterar pistas individualmente, a través de las entradas de melodía y ritmo. En la partitura central derecha de la Figura 11-16 podemos ver el efecto de cambiar sólo la entrada de ruido de la melodía para la línea superior de la música. Todas las demás partes no se ven afectadas, pero las notas principales cambian significativamente.

Además, podemos ver un cambio rítmico entre los dos compases en la línea superior: el segundo compás es más dinámico y contiene notas más rápidas que el primer compás.

Por último, la puntuación de la parte inferior derecha del diagrama muestra la puntuación prevista cuando modificamos el parámetro de entrada del surco solo para la línea de base. Nuevamente, todas las demás partes no se ven afectadas, pero la línea de base es diferente. Además, el patrón general de la línea de base sigue siendo similar entre barras, como era de esperar.

Esto muestra cómo cada uno de los parámetros de entrada se puede utilizar para influir directamente en las características de alto nivel de la secuencia musical generada, de la misma manera que pudimos ajustar los vectores

latentes de VAE y GAN en capítulos anteriores para alterar la apariencia de una imagen generada. Un inconveniente del modelo es que el número de barras a generar debe especificarse por adelantado. Para abordar esto, los autores muestran una extensión del modelo que permite que las barras anteriores se introduzcan como entrada, lo que permite que el modelo genere puntuaciones de formato largo al introducir continuamente las barras predichas más recientes como entrada adicional.

Resumen

En este capítulo hemos explorado dos tipos diferentes de modelos para la generación de música: un Transformer y una MusaGAN.

El Transformer tiene un diseño similar a las redes que vimos en el Capítulo 9 para la generación de texto. La generación de música y texto comparten muchas características en común y, a menudo, se pueden utilizar técnicas similares para ambas. Ampliamos la arquitectura del transformador al incorporar dos flujos de entrada y salida, para nota y duración. Vimos cómo el modelo fue capaz de aprender conceptos como claves y escalas, simplemente aprendiendo a generar con precisión la música de Bach.

También exploramos cómo podemos adaptar el proceso de tokenización para manejar la generación de música polifónica (multipista). La tokenización de cuadrícula serializa una representación de pianola de la partitura, lo que nos permite entrenar un Transformador en un único flujo de tokens que describen qué nota está presente en cada voz, en intervalos de tiempo discretos e igualmente espaciados. Tokenización basada en eventos

produce una receta que describe cómo crear múltiples líneas de música de forma secuencial, a través de un único flujo de instrucciones. Ambos métodos tienen ventajas y desventajas: el éxito o el fracaso de un enfoque de generación de música basado en Transformer a menudo depende en gran medida de la elección del método de tokenización.

También vimos que generar música no siempre requiere un enfoque secuencial: MuseGAN utiliza convoluciones para generar partituras musicales polifónicas con múltiples pistas, tratando la partitura como una imagen donde las pistas son canales individuales de la imagen. La novedad

de MuseGAN radica en la forma en que se organizan los cuatro vectores de ruido de entrada (acordes, estilo, melodía y ritmo) de modo que sea posible mantener un control total sobre las características de alto nivel de la música.

Si bien la armonización subyacente aún no es tan perfecta ni tan variada como la de Bach, es un buen intento de abordar un problema extremadamente difícil de dominar y destaca el poder de las GAN para abordar una amplia variedad de problemas.

1 Cheng-Zhi Anna Huang et al., “Music Transformer: Generating Music with Long-Term Structure”, 12 de septiembre de 2018, <https://arxiv.org/abs/1809.04281>.

2 Hao-Wen Dong et al., “MuseGAN: Redes adversarias para la generación simbólica de música y acompañamiento”, 19 de septiembre de 2017, <https://arxiv.org/abs/1809.04281>.

Capítulo 12. Modelos del mundo

METAS DEL CAPÍTULO

En este capítulo podrás:

Recorra los conceptos básicos del aprendizaje por refuerzo (RL).

Comprender cómo se puede utilizar el modelado generativo dentro de un enfoque de modelo mundial para RL.

Vea cómo entrenar un auto codificador variacional (VAE) para capturar observaciones ambientales en un espacio latente de baja dimensión.

Recorra el proceso de entrenamiento de una red neuronal recurrente y de densidad mixta (MDNRNN) que predice la variable latente.

Utilice la estrategia de evolución de adaptación de matriz de covarianza (CMA-ES) para entrenar un controlador que pueda tomar acciones inteligentes en el entorno.

Comprenda cómo el MDN-RNN entrenado puede usarse como entorno, permitiendo al agente entrenar al controlador dentro de sus propios sueños alucinados, en lugar del entorno real.

Este capítulo presenta una de las aplicaciones más interesantes de los modelos generativos en los últimos años, es decir, su uso dentro de los llamados modelos mundiales.

Introducción

En marzo de 2018, David Ha y Jürgen Schmidhuber publicaron su artículo “World Models”.¹ El artículo mostró cómo es posible entrenar un modelo que pueda aprender a realizar una tarea particular a través de la experimentación dentro de su propio entorno onírico generado, en lugar de dentro del entorno real. Es un excelente ejemplo de cómo se puede utilizar el modelado generativo para resolver problemas prácticos, cuando se aplica

junto con otras técnicas de aprendizaje automático, como el aprendizaje por refuerzo.

Un componente clave de la arquitectura es un modelo generativo que puede construir una distribución de probabilidad para el siguiente estado posible, dado el estado y la acción actuales. Una vez que ha adquirido una comprensión de la física subyacente del entorno a través de movimientos aleatorios, el modelo puede entrenarse desde cero en una nueva tarea, enteramente dentro de su propia representación interna del entorno.

Este enfoque condujo a las mejores puntuaciones del mundo en las dos tareas en las que se probó.

En este capítulo exploraremos el modelo del artículo en detalle, con especial atención en una tarea que requiere que el agente aprenda a conducir un automóvil por una pista de carreras virtual lo más rápido posible. Si bien usaremos una simulación por computadora 2D como nuestro entorno, la misma técnica también podría aplicarse a escenarios del mundo real donde probar estrategias en el entorno real es costoso o inviable.

CONSEJO

En este capítulo haremos referencia a la excelente implementación de TensorFlow del documento "World Models" disponible públicamente en GitHub, que te animo a clonar y ejecutar tú mismo.

Antes de comenzar a explorar el modelo, debemos analizar más de cerca el concepto de aprendizaje por refuerzo.

Aprendizaje reforzado

El aprendizaje por refuerzo se puede definir de la siguiente manera:

El aprendizaje por refuerzo (RL) es un campo del aprendizaje automático que tiene como objetivo entrenar a un agente para que se desempeñe de manera óptima en un entorno determinado, con respecto a un objetivo particular.

Si bien tanto el modelado discriminativo como el generativo tienen como objetivo minimizar una función de pérdida en un conjunto de datos de observaciones, el aprendizaje por refuerzo tiene como objetivo maximizar la recompensa a largo plazo de un agente en un entorno determinado. A menudo se describe como una de las tres ramas principales del aprendizaje automático, junto con el aprendizaje supervisado (predicción utilizando datos etiquetados) y el aprendizaje no supervisado (estructura de aprendizaje a partir de datos no etiquetados).

Primero introduzcamos alguna terminología clave relacionada con el aprendizaje por refuerzo:

Ambiente

El mundo en el que opera el agente. Define el conjunto de reglas que rigen el proceso de actualización del estado del juego y la asignación de recompensas, dada la acción anterior del agente y el estado actual del juego. Por ejemplo, si estuviéramos enseñando un algoritmo de aprendizaje por refuerzo para jugar al ajedrez, el entorno consistiría en las reglas que rigen cómo una acción dada (por ejemplo, el movimiento del peón e2e4) afecta el siguiente estado del juego (las nuevas posiciones de las piezas en el tablero). tablero) y también especificaría cómo evaluar si una posición determinada es jaque mate y asignaría al jugador ganador una recompensa de 1 después del movimiento ganador.

Agente

La entidad que realiza acciones en el medio ambiente.

Estado del juego

Los datos que representan una situación particular que el agente puede encontrar (también llamado simplemente estado). Por ejemplo, una configuración particular de tablero de ajedrez acompañada de información del juego, como qué jugador realizará el siguiente movimiento.

Acción

Un movimiento factible que un agente puede hacer.

Premio

El valor que el entorno devuelve al agente después de que se ha realizado una acción. El agente pretende maximizar la suma a largo plazo de sus recompensas. Por ejemplo, en un juego de ajedrez, hacer jaque mate al rey del oponente tiene una recompensa de 1 y cada dos movimientos tiene una recompensa de 0. Otros juegos tienen recompensas otorgadas constantemente a lo largo del episodio (por ejemplo, puntos en un juego de Space Invaders).

Episodio

Una ejecución de un agente en el entorno; esto también se llama implementación.

Hora de caminar

Para un entorno de evento discreto, todos los estados, acciones y recompensas están subíndices para mostrar su valor en el paso de tiempo t .

La relación entre estos conceptos se muestra en la figura 12-1.

Figura 12-1. Diagrama de aprendizaje por refuerzo

Primero se inicializa el entorno con un estado actual del juego, s_0 . En el paso de tiempo t , el agente recibe el estado actual del juego s_t y lo utiliza para decidir cuál es su siguiente mejor acción, que luego realiza. Dada esta acción, el entorno calcula el siguiente estado s_{t+1} y la recompensa r_{t+1} y los devuelve al agente, para que el ciclo comience de nuevo.

El ciclo continúa hasta que se cumple el criterio final del episodio (por ejemplo, transcurre un número determinado de pasos de tiempo o el agente gana/pierde).

¿Cómo podemos diseñar un agente para maximizar la suma de recompensas en un entorno determinado? Podríamos crear un agente que contenga un conjunto de reglas sobre cómo responder a cualquier estado del juego. Sin

embargo, esto rápidamente se vuelve inviable a medida que el entorno se vuelve más complejo y nunca nos permite crear un agente que tenga una habilidad sobrehumana en una tarea particular, ya que estamos codificando las reglas.

El aprendizaje por refuerzo implica la creación de un agente que pueda aprender estrategias óptimas por sí mismo en entornos complejos mediante juegos repetidos.

Echemos ahora un vistazo al entorno CarRacing que simula un coche conduciendo por una pista.

El entorno de las carreras de coches

CarRacing es un entorno que está disponible a través del paquete Gymnasium. Gymnasium es una biblioteca de Python para desarrollar algoritmos de aprendizaje por refuerzo que contiene varios entornos clásicos de aprendizaje por refuerzo, como CartPole y Pong, así como entornos que presentan desafíos más complejos, como entrenar a un agente para caminar sobre terreno irregular o ganar un juego de Atari.

GIMNASIO

Gymnasium es una bifurcación mantenida de la biblioteca Gym de OpenAI desde 2021, el desarrollo adicional de Gym se trasladará a Gymnasium. Por lo tanto, en este libro nos referimos a los entornos Gymnasium como entornos Gym.

Todos los entornos proporcionan un método paso a paso mediante el cual puede enviar una acción determinada; el entorno devolverá el siguiente estado y la recompensa. Al llamar repetidamente al método de pasos con las acciones elegidas por el agente, puedes reproducir un episodio en el entorno. También hay un método de reinicio para devolver el entorno a su estado inicial y un método de renderizado que le permite observar el desempeño de su agente en un entorno determinado. Esto es útil para depurar y encontrar áreas donde su agente podría mejorar.

Veamos cómo se definen el estado, la acción, la recompensa y el episodio del juego para el entorno CarRacing:

Estado del juego

Una imagen RGB de 64×64 píxeles que muestra una vista aérea de la pista y el auto.

Acción

Un conjunto de tres valores: dirección de dirección (-1 a 1), aceleración (0 a 1) y frenado (0 a 1). El agente debe establecer los tres valores en cada paso de tiempo.

Recompensa

Una penalización negativa de -0,1 por cada paso de tiempo realizado y una recompensa positiva de $1000/N$ si se visita una nueva ficha de pista, donde N es el número total de fichas que componen la pista.

Episodio

El episodio termina cuando el coche completa la pista o se sale del borde del entorno, o cuando han transcurrido 3.000 pasos de tiempo.

Estos conceptos se muestran en una representación gráfica del estado de un juego en la Figura 12-2.

Figura 12-2. Una representación gráfica del estado de un juego en el entorno CarRacing.

PERSPECTIVA

Deberíamos imaginar al agente flotando sobre la pista y controlando el coche a vista de pájaro, en lugar de ver la pista desde la perspectiva del conductor.

Descripción general del modelo mundial

Ahora cubriremos una descripción general de alto nivel de toda la arquitectura del modelo mundial y el proceso de capacitación, antes de profundizar en cada componente con más detalle.

Arquitectura

La solución consta de tres partes distintas, como se muestra en la Figura 12-3, que se entrenan por separado:

Un auto codificador variacional (VAE)

Una red neuronal recurrente con una red de densidad mixta (MDN-RNN)

un controlador

Figura 12-3. Diagrama de arquitectura del modelo mundial.

El VAE

Cuando toma decisiones mientras conduce, no analiza activamente cada píxel de su vista; en cambio, condensa la información visual en un número menor de entidades latentes, como la rectitud de la carretera, las próximas curvas y su posición relativa, a la carretera, para informar su próxima acción.

Vimos en el Capítulo 3 cómo un VAE puede tomar una imagen de entrada de alta dimensión y condensarla en una variable aleatoria latente que sigue aproximadamente una distribución gaussiana estándar, mediante minimización del error de reconstrucción y divergencia KL. Esto garantiza que el espacio latente sea continuo y que podamos tomar muestras fácilmente de él para generar nuevas observaciones significativas.

En el ejemplo de las carreras de autos, el VAE condensa los $64 \times 64 \times$

3 (RGB) en una variable aleatoria de 32 dimensiones distribuida normalmente, parametrizada por dos variables, μ y $\log\sigma$. Aquí, $\log\sigma$ es el logaritmo de la varianza de la distribución. Podemos tomar muestras de esta distribución para producir un vector latente z que representa el estado actual. Esto se pasa a la siguiente parte de la red, el MDN-RNN.

El MDN-RNN

Mientras conduce, cada observación posterior no es una completa sorpresa para usted. Si la observación actual sugiere un giro a la izquierda en el camino que tiene delante y usted gira el volante hacia la izquierda, espera que la siguiente observación muestre que todavía está en línea con el camino.

Si no tuvieras esta habilidad, tu auto probablemente serpentearía por toda la carretera ya que no serías capaz de ver que una ligera desviación del centro será peor en el siguiente paso a menos que hagas algo al respecto ahora. .

Esta visión de futuro es trabajo de MDN-RNN, una red que intenta predecir la distribución del siguiente estado latente en función del estado latente anterior y la acción anterior.

Específicamente, MDN-RNN es una capa LSTM con 256 unidades ocultas seguida de una capa de salida de red de densidad mixta (MDN) que permite que el siguiente estado latente en realidad pueda extraerse de cualquiera de varias distribuciones normales.

La misma técnica fue aplicada por uno de los autores del artículo “World Models”, David Ha, a una tarea de generación de escritura a mano, como se muestra en la Figura 12-4, para describir el hecho de que la siguiente punta del bolígrafo podría aterrizar en cualquiera de las distintas áreas rojas.

Figura 12-4. MDN para generación de escritura a mano

En el ejemplo de las carreras de autos, permitimos que cada elemento del siguiente estado latente observado se extraiga de cualquiera de las cinco distribuciones normales.

El controlador

Hasta este punto, no hemos mencionado nada sobre la elección de una acción. Esa responsabilidad recae en el responsable del tratamiento. El controlador es una red neuronal densamente conectada, donde la entrada es una concatenación de z (el estado latente actual muestreado de la

distribución codificada por el VAE) y el estado oculto del RNN. Las tres neuronas de salida corresponden a las tres acciones (girar, acelerar, frenar) y están escaladas para caer en los rangos apropiados.

El controlador se entrena mediante aprendizaje por refuerzo, ya que no existe un conjunto de datos de entrenamiento que nos diga que una determinada acción es buena y otra es mala. En cambio, el agente descubre esto por sí mismo mediante experimentación repetida.

Como veremos más adelante en el capítulo, el quid de la cuestión del artículo de Modelos mundiales es que demuestra cómo este aprendizaje por refuerzo puede tener lugar dentro del propio modelo generativo del entorno del agente, en lugar del ambiente de gimnasio. En otras palabras, tiene lugar en la versión alucinada del agente de cómo se comporta el entorno, en lugar de en la realidad.

Para comprender los diferentes roles de los tres componentes y cómo funcionan juntos, podemos imaginar un diálogo entre ellos:

VAE (observando la última observación de $64 \times 64 \times 3$): parece una carretera recta, con una ligera curva a la izquierda acercándose, con el coche mirando en la dirección de la carretera (z).

RNN: Según esa descripción (z) y el hecho de que el controlador eligió acelerar con fuerza en el último paso de tiempo (acción), actualizare mi estado oculto (h) para que se prediga que la próxima observación seguirá siendo un camino recto, pero con un poco más de giro a la izquierda a la vista.

Controlador: según la descripción de VAE (z) y el estado oculto actual de RNN (h), mi red neuronal genera [0.34, 0.8, 0] como siguiente acción.

La acción del controlador luego pasa al entorno, que devuelve una observación actualizada y el ciclo comienza de nuevo.

Capacitación

El proceso de capacitación consta de cinco pasos, ejecutados en secuencia, que se describen a continuación:

1. Recopilar datos de implementación aleatorios. Aquí, al agente no le importa la tarea asignada, sino que simplemente explora el entorno mediante acciones aleatorias. Se simulan múltiples episodios y se almacenan los estados, acciones y recompensas observados en cada paso de tiempo. La idea es construir un conjunto de datos sobre cómo funciona la física del entorno funciona, del cual el VAE puede aprender para capturar los estados de manera eficiente como vectores latentes. El MDN-RNN puede posteriormente aprender cómo evolucionan los vectores latentes con el tiempo.
2. Entrenar al VAE. Utilizando los datos recopilados aleatoriamente, entrenamos un VAE en las imágenes de observación.
3. Recopilar datos para entrenar el MDN-RNN. Una vez que tenemos un VAE entrenado, lo usamos para codificar cada una de las observaciones recopiladas en vectores mu y logvar, que se guardan junto con la acción y la recompensa actuales.
4. Entrenar al MDN-RNN. Tomamos lotes de episodios y cargamos las variables mu, logvar, acción y recompensa correspondientes en cada paso de tiempo que se generaron en el paso 3. Luego tomamos muestras de un vector z a partir de los vectores mu y logvar. Dado el vector z, la acción y la recompensa actuales, el MDN-RNN se entrena para predecir el vector z y la recompensa posteriores.
5. Entrenar al controlador. Con un VAE y un RNN entrenados, ahora podemos entrenar al controlador para que genere una acción dada la z actual y el estado oculto, h, del RNN. El controlador utiliza un algoritmo evolutivo, CMA-ES, como optimizador. El algoritmo recompensa las ponderaciones de la matriz que generan acciones que conducen a puntuaciones generales altas en la tarea, de modo que es probable que las generaciones futuras también hereden este comportamiento deseado.

Veamos ahora cada uno de estos pasos con más detalle.

Recopilación de datos de implementación aleatorios

El primer paso es recopilar datos de implementación del entorno mediante un agente que realiza acciones aleatorias. Esto puede parecer extraño, dado que en última instancia queremos que nuestro agente aprenda a realizar acciones inteligentes, pero este paso proporcionará los datos que el agente utilizará para aprender cómo funciona el mundo y cómo sus acciones (aunque aleatorias al principio) influyen en las observaciones posteriores.

Podemos capturar múltiples episodios en paralelo activando múltiples procesos de Python, cada uno ejecutando una instancia separada del entorno. Cada proceso se ejecutará en un núcleo independiente, por lo que si su máquina tiene muchos núcleos, podrá recopilar datos mucho más rápido que si solo tuviera unos pocos núcleos.

Los hiperparámetros utilizados en este paso son los siguientes:

`procesos_paralelos`

La cantidad de procesos paralelos que se ejecutarán (por ejemplo, 8 si su máquina tiene ≥ 8 núcleos)

`max_trials`

Cuántos episodios debe ejecutar cada proceso en total (por ejemplo, 125, por lo que 8 procesos crearían 1000 episodios en total)

`max_frames`

El número máximo de pasos de tiempo por episodio (p. ej., 300)

La Figura 12-5 muestra un extracto de los fotogramas 40 a 59 de un episodio, cuando el coche se acerca a una esquina, junto con la acción y la recompensa elegidas al azar. Observe cómo la recompensa cambia a 3,22 cuando el automóvil pasa sobre nuevos mosaicos de pista, pero por lo demás es -0,1.

Figura 12-5. Fotogramas 40 a 59 de un episodio

Entrenando al VAE

Ahora construimos un modelo generativo (un VAE) a partir de estos datos recopilados. Recuerde, el objetivo del VAE es permitirnos colapsar una imagen de $64 \times 64 \times 3$ en una variable aleatoria z normalmente distribuida, cuya distribución está parametrizada por dos vectores, μ y $\log\sigma$. Cada uno de estos vectores tiene una longitud de 32. Los hiperparámetros de este paso son los siguientes:

`vae_batch_size`

El tamaño de lote que se utilizará al entrenar el VAE (cuántas observaciones por lote) (por ejemplo, 100)

`z_size`

La longitud del vector z latente (y por lo tanto de las variables μ y $\log\sigma$) (por ejemplo, 32)

`vae_num_epoch`

El número de épocas de entrenamiento (por ejemplo, 10)

La arquitectura VAE

Como hemos visto anteriormente, Keras nos permite no solo definir el modelo VAE que se entrenará de un extremo a otro, sino también submodelos adicionales que definen el codificador y decodificador de la red entrenada por separado. Estos serán útiles cuando queramos codificar una imagen específica o decodificar un vector z determinado, por ejemplo. Definiremos el modelo VAE y tres submodelos, de la siguiente manera:

`vae`

Este es el VAE de extremo a extremo que está capacitado. Acepta una imagen de $64 \times 64 \times 3$ como entrada y saca una imagen reconstruida de $64 \times 64 \times 3$.

`encode_mu_logvar`

Esto acepta una imagen de $64 \times 64 \times 3$ como entrada y genera los vectores mu y logvar correspondientes a esta entrada.

Ejecutar la misma imagen de entrada a través de este modelo varias veces producirá los mismos vectores mu y logvar cada vez.

encode

Esto acepta una imagen de $64 \times 64 \times 3$ como entrada y genera un vector z muestreado. Ejecutar la misma imagen de entrada a través de este modelo varias veces producirá un vector z diferente cada vez, utilizando los valores mu y logvar calculados para definir la distribución de muestreo.

decode

Esto acepta un vector z como entrada y devuelve la imagen reconstruida de $64 \times 64 \times 3$.

Un diagrama del modelo y submodelos se muestra en la Figura 12-6.

Figura 12-6. La arquitectura VAE del artículo "World Models"

Explorando el VAE

Ahora veremos el resultado del VAE y cada submodelo y luego veremos cómo se puede utilizar el VAE para generar observaciones de seguimiento completamente nuevas.

El modelo VAE

Si alimentamos al VAE con una observación, es capaz de reconstruir con precisión la imagen original, como se muestra en la Figura 12-7. Esto es útil para comprobar visualmente que el VAE está funcionando correctamente.

Figura 12-7. La entrada y salida del modelo VAE.

Los modelos de codificador.

Si alimentamos el modelo encode_mu_logvar con una observación, el resultado son los vectores mu y logvar generados que describen una distribución normal multivariada. El modelo de codificación va un paso más allá al muestrear un vector z particular de esta distribución. El diagrama que muestra la salida de los dos modelos de codificador se muestra en la Figura 12-8.

Figura 12-8. La salida de los modelos de codificador.

La variable latente z se toma como muestra del gaussiano definido por mu y logvar tomando un muestreo de un gaussiano estándar y luego escalando y desplazando el vector muestreado (Ejemplo 12-1).

Ejemplo 12-1. Muestreo de z de la distribución normal multivariada definida por mu y logvar

```
eps = tf.random_normal(shape=tf.shape(mu))
sigma = tf.exp(logvar * 0.5)
z = mu + eps * sigma
```

El modelo decodificador

El modelo de decodificación acepta un vector z como entrada y reconstruye la imagen original. En la Figura 12-9 interpolamos linealmente dos de las dimensiones de z para mostrar cómo cada dimensión parece codificar un aspecto particular de la pista; en este ejemplo, z[4] controla la dirección inmediata izquierda/derecha de la pista más cercana al automóvil y z[7] controla la nitidez del giro a la izquierda que se aproxima.

Esto muestra que el espacio latente que el VAE ha aprendido es continuo y puede usarse para generar nuevos segmentos de seguimiento que el agente nunca antes había observado.

Figura 12-9. Una interpolación lineal de dos dimensiones de z.

Recopilación de datos para entrenar el MDN-RNN

Ahora que tenemos un VAE entrenado, podemos usarlo para generar datos de entrenamiento para nuestro MDN-RNN.

En este paso, pasamos todas las observaciones de implementación aleatoria a través del modelo encode_mu_logvar y almacenamos los vectores mu y logvar correspondientes a cada observación. Estos datos codificados, junto con la acción, la recompensa y las variables realizadas ya recopiladas, se utilizarán para entrenar el MDNRNN. Este proceso se muestra en la Figura 12-10.

Figura 12-10. Creando el conjunto de datos de entrenamiento MDN-RNN

Entrenamiento del MDN-RNN

Ahora podemos entrenar el MDN-RNN para predecir la distribución del siguiente vector z y recompensar un paso adelante hacia el futuro, dado el vector z actual, la acción actual y la recompensa anterior. Luego podemos usar el estado oculto interno del RNN (que puede considerarse como la comprensión actual del modelo de la dinámica del entorno) como parte de la entrada en el controlador, que en última instancia decidirá cuál es la mejor acción a tomar a continuación.

Los hiperparámetros de este paso del proceso son los siguientes:

rnn_batch_size

El tamaño del lote que se utilizará al entrenar MDN-RNN (cuántas secuencias por lote) (por ejemplo, 100)

rnn_num_steps

El número total de iteraciones de entrenamiento (por ejemplo, 4000)

La arquitectura MDN-RNN

La arquitectura de MDN-RNN se muestra en la Figura 12-11.

Figura 12-11. La arquitectura MDN-RNN

El MDN-RNN consta de una capa LSTM (el RNN), seguida de una capa Densemente conectada (el MDN) que transforma el estado oculto del LSTM en los parámetros de una distribución mixta. Repasemos la red paso a paso.

La entrada a la capa LSTM es un vector de longitud 36, una concatenación del vector z codificado (longitud 32) de la VAE, la acción actual (duración 3) y la recompensa anterior (duración 1).

La salida de la capa LSTM es un vector de longitud 256: un valor para cada celda LSTM en la capa. Esto se pasa al MDN, que es simplemente una capa Densemente conectada que transforma el vector de longitud 256 en un vector de longitud 481.

¿Por qué 481? La Figura 12-12 explica la composición de la salida de MDN-RNN. El objetivo de una red de densidad mixta es modelar el hecho de que nuestra próxima z podría extraerse de una de varias distribuciones posibles con una cierta probabilidad. En el ejemplo de las carreras de autos, elegimos cinco distribuciones normales. ¿Cuántos parámetros necesitamos para definir estas distribuciones? Para cada una de las 5 mezclas, necesitamos un mu y un logvar (para definir la distribución) y una probabilidad logarítmica de que se elija esta mezcla (logpi), para cada una de las 32 dimensiones de z. Esto hace $5 \times 3 \times 32 =$

480 parámetros. El único parámetro adicional es para la predicción de recompensa.

Figura 12-12. La salida de la red de densidad de mezcla.

Muestreo del MDN-RNN

Podemos tomar muestras de la salida de MDN para generar una predicción para la próxima z y recompensar en el siguiente paso de tiempo, mediante el siguiente proceso:

1. Divida el vector de salida de 481 dimensiones en las 3 variables (logpi, mu, logvar) y el valor de recompensa.

1. Exponenciar y escalar logpi de modo que pueda interpretarse como 32 distribuciones de probabilidad sobre los 5 índices de mezcla.
1. Para cada una de las 32 dimensiones de z , tome una muestra de las distribuciones creadas a partir de logpi (es decir, elija cuál de las 5 distribuciones debe usarse para cada dimensión de z).
1. Obtenga los valores correspondientes de mu y logvar para esta distribución.
1. Muestreo de un valor para cada dimensión de z de la distribución normal parametrizada por los parámetros elegidos de mu y logvar para esta dimensión.

La función de pérdida para MDN-RNN es la suma de la pérdida de reconstrucción del vector z y la pérdida de recompensa. La pérdida de reconstrucción del vector z es la probabilidad logarítmica negativa de la distribución predicha por MDN-RNN, dado el valor verdadero de z , y la pérdida de recompensa es el error cuadrático medio entre la recompensa predicha y la recompensa verdadera.

Entrenando al controlador

El último paso es entrenar al controlador (la red que genera la acción elegida) utilizando un algoritmo evolutivo llamado estrategia de evolución de adaptación de matriz de covarianza (CMA-ES).

Los hiperparámetros de este paso del proceso son los siguientes:

`controller_num_worker`

La cantidad de trabajadores que probarán soluciones en paralelo

`controller_num_worker_trial`

La cantidad de soluciones que cada trabajador recibirá para probar en cada generación.

`controller_num_episode`

La cantidad de episodios con los que se probará cada solución para calcular la recompensa promedio

controlador_eval_steps

El número de generaciones entre evaluaciones del mejor conjunto de parámetros actual.

La arquitectura del controlador

La arquitectura del controlador es muy simple. Es una red neuronal densamente conectada sin capas ocultas. Conecta el vector de entrada directamente al vector de acción.

El vector de entrada es una concatenación del vector z actual (longitud 32) y el estado oculto actual del LSTM (longitud 256), lo que da un vector de longitud 288. Dado que estamos conectando cada unidad de entrada directamente a las 3 unidades de acción de salida, el número total de pesos a ajustar es $288 \times 3 = 864$, más 3 pesos de sesgo, lo que da 867 en total.

¿Cómo debemos entrenar esta red? Tenga en cuenta que este no es un problema de aprendizaje supervisado; no estamos tratando de predecir la acción correcta. No existe un conjunto de entrenamiento de acciones correctas, ya que no sabemos cuál es la acción óptima para un estado dado del medio ambiente. Esto es lo que lo distingue como un problema de aprendizaje por refuerzo. Necesitamos que el agente descubra los valores óptimos para los pesos experimentando dentro del entorno y actualizando sus pesos en función de los comentarios recibidos.

Las estrategias evolutivas son una opción popular para resolver problemas de aprendizaje por refuerzo debido a su simplicidad, eficiencia y escalabilidad. Usaremos una estrategia particular, conocida como CMA-ES.

CMA-ES

Las estrategias evolutivas generalmente se adhieren al siguiente proceso:

1. Cree una población de agentes e inicialice aleatoriamente los parámetros que se optimizarán para cada agente.
1. Realice un bucle sobre lo siguiente:
 - a. Evalúe cada agente en el entorno y devuelva la recompensa promedio en varios episodios.
 - b. Cría a los agentes con las mejores puntuaciones para crear nuevos miembros de la población.
- C. Añade aleatoriedad a los parámetros de los nuevos miembros.
- d. Actualice el grupo de población agregando los agentes recién creados y eliminando los agentes con bajo rendimiento.

Esto es similar al proceso mediante el cual los animales evolucionan en la naturaleza, de ahí el nombre de estrategias evolutivas.

“Cría” en este contexto simplemente significa combinar los agentes existentes con mejor puntuación de manera que la próxima generación tenga más probabilidades de producir resultados de alta calidad, similares a los de sus padres. Como ocurre con todas las soluciones de aprendizaje por refuerzo, se debe encontrar un equilibrio entre buscar con avidez soluciones localmente óptimas y explorar áreas desconocidas del espacio de parámetros para soluciones potencialmente mejores. Por eso es importante agregar aleatoriedad a la población, para garantizar que no seamos demasiado limitados en nuestro campo de búsqueda.

CMA-ES es sólo una forma de estrategia evolutiva. En resumen, funciona manteniendo una distribución normal a partir de la cual puede tomar muestras de los parámetros de nuevos agentes. En cada generación, actualiza la media de la distribución para maximizar la probabilidad de muestrear los agentes con puntuación más alta del paso de tiempo anterior. Al mismo tiempo, actualiza la matriz de covarianza de la distribución para maximizar la probabilidad de muestrear los agentes con puntuación alta, dada la media anterior. Se puede considerar como una forma de descenso de gradiente que surge naturalmente, pero con el beneficio adicional de que no tiene derivados, lo que significa que no necesitamos calcular ni estimar gradientes costosos.

En la Figura 12-13 se muestra una generación del algoritmo demostrada en un ejemplo de juguete. Aquí estamos tratando de encontrar el punto mínimo de una función altamente no lineal en dos dimensiones: el valor de la función en las áreas roja/negra de la imagen es mayor que el valor de la función en las partes blanca/amarilla de la imagen.

Figura 12-13. Un paso de actualización del algoritmo CMA-ES (fuente: Ha, 2017)2

Los pasos son los siguientes:

1. Comenzamos con una distribución normal 2D generada aleatoriamente y tomamos muestras de una población de candidatos, que se muestra en azul en la Figura 12-13.
2. Luego calculamos el valor de la función para cada candidato y aislamos el mejor 25%, que se muestra en violeta en la Figura 12-13: llamaremos a este conjunto de puntos P.
3. Establecemos la media de la nueva distribución normal como la media de los puntos en P. Esto puede considerarse como la etapa de reproducción, en la que solo utilizamos los mejores candidatos para generar una nueva media para la distribución.

También configuramos la matriz de covarianza de la nueva distribución normal como la matriz de covarianza de los puntos en P, pero utilice la media existente en el cálculo de la covarianza en lugar de la media actual de los puntos en P. Cuanto mayor sea la diferencia entre las medias existentes y la media de los puntos en P, más amplia será la varianza de la siguiente distribución normal. Esto tiene el efecto de crear naturalmente un impulso en la búsqueda de los parámetros óptimos.

4. Luego podemos muestrear una nueva población de candidatos de nuestra nueva distribución normal con una matriz de media y covarianza actualizada.

La Figura 12-14 muestra varias generaciones del proceso. Observe cómo la covarianza se amplía a medida que la media se mueve en grandes pasos

hacia el mínimo, pero se estrecha a medida que la media se estabiliza en el mínimo verdadero.

Figura 12-14. CMA-ES (fuente: Wikipedia)

Para la tarea de carreras de autos, no tenemos una función bien definida para maximizar, sino un entorno donde los 867 parámetros a optimizar determinan qué tan bien puntúa el agente. Inicialmente, algunos conjuntos de parámetros, por casualidad, generarán puntuaciones más altas que otros y el algoritmo moverá gradualmente la distribución normal en la dirección de aquellos parámetros que tengan la puntuación más alta en el entorno.

Paralelizando CMA-ES

Uno de los grandes beneficios de CMA-ES es que se puede parallelizar fácilmente. La parte del algoritmo que consume más tiempo es calcular la puntuación para un conjunto determinado de parámetros, ya que necesita simular un agente con estos parámetros en el entorno. Sin embargo, este proceso se puede parallelizar, ya que no existen dependencias entre simulaciones individuales. Hay un proceso orquestador que envía conjuntos de parámetros para ser probados a muchos procesos de nodo en paralelo. Los nodos devuelven los resultados al orquestador, que los acumula y luego pasa el resultado general de la generación al objeto CMA-ES. Este objeto actualiza la matriz de media y covarianza de la distribución normal según la Figura 12-13 y proporciona al orquestador una nueva población para probar. Luego el bucle comienza de nuevo. La figura 12-15 explica esto en un diagrama.

Figura 12-15. Paralelización de CMA-ES: aquí hay un tamaño de población de ocho y cuatro nodos (por lo que $t = 2$, el número de ensayos de los que cada nodo es responsable)

El orquestador solicita a los objetos CMA-ES un conjunto de parámetros para probar.

El orquestador divide los parámetros en la cantidad de nodos disponibles. Aquí, cada uno de los cuatro procesos de nodo pone a prueba dos conjuntos de parámetros.

Los nodos ejecutan un proceso de trabajo que recorre cada conjunto de parámetros y ejecuta varios episodios para cada uno. Aquí ejecutamos tres episodios para cada conjunto de parámetros.

Las recompensas de cada episodio se promedian para dar una puntuación única para cada conjunto de parámetros.

Cada nodo devuelve su lista de partituras al orquestador.

El orquestador agrupa todas las partituras y envía esta lista al objeto es.

El objeto es utiliza esta lista de recompensas para calcular la nueva distribución normal según la Figura 12-13.

Después de alrededor de 200 generaciones, el proceso de entrenamiento alcanza una puntuación de recompensa promedio de alrededor de 840 para la tarea de carrera de autos, como se muestra en la Figura 12-16.

Figura 12-16. Recompensa promedio por episodio del proceso de capacitación de controladores, por generación (fuente: Zac Wellmer, “World Models”)

Entrenamiento en sueños

Hasta el momento, la capacitación del controlador se ha realizado utilizando el entorno Gym CarRacing para implementar el método de pasos que mueve la simulación de un estado al siguiente. Esta función calcula el siguiente estado y la recompensa, dado el estado actual del entorno y la acción elegida.

Observe cómo el método de pasos realiza una función muy similar al MDN-RNN en nuestro modelo. Muestreo de la MDN-RNN genera una predicción para la próxima z y recompensa, dada la z actual y la acción elegida.

De hecho, se puede considerar el MDN-RNN como un entorno en sí mismo, pero que opera en el espacio z en lugar de en el espacio de la imagen original. Increíblemente, esto significa que podemos sustituir el

entorno real por una copia de la MDN-RNN y entrenar al controlador completamente dentro de un sueño inspirado en MDNRNN sobre cómo debería comportarse el medio ambiente.

En otras palabras, MDN-RNN ha aprendido lo suficiente sobre la física general del entorno real a partir del conjunto de datos de movimiento aleatorio original como para poder utilizarlo como proxy del entorno real al entrenar al controlador. Esto es bastante notable: significa que el agente puede entrenarse para aprender una nueva tarea pensando en cómo puede maximizar la recompensa en el entorno de sus sueños, sin tener que probar estrategias en el mundo real.

Entonces puede desempeñarse bien en la tarea la primera vez, sin haberla intentado nunca en realidad.

A continuación se presenta una comparación de las arquitecturas para el entrenamiento en el entorno real y en el entorno onírico: la arquitectura del mundo real se muestra en la Figura 12-17 y la configuración del entrenamiento en el sueño se ilustra en la Figura 12-18.

Figura 12-17. Entrenando al controlador en el ambiente del Gimnasio

Observe cómo en la arquitectura del sueño, el entrenamiento del controlador se realiza completamente en el espacio z sin la necesidad de decodificar los vectores z nuevamente en imágenes de seguimiento reconocibles. Por supuesto, podemos hacerlo para inspeccionar visualmente el desempeño del agente, pero no es necesario para la capacitación.

Figura 12-18. Entrenando al controlador en el entorno de ensueño MDN-RNN

Uno de los retos de formar agentes íntegramente dentro del entorno de ensueño de MDN-RNN está sobreajustado. Esto ocurre cuando el agente encuentra una estrategia que es gratificante en el entorno del sueño pero que no se generaliza bien al entorno real, debido a que MDN-RNN no captura completamente cómo se comporta el entorno real bajo ciertas condiciones.

Los autores del artículo original destacan este desafío y muestran cómo incluir un parámetro de temperatura para controlar la incertidumbre del modelo puede ayudar a aliviar el problema.

El aumento de este parámetro magnifica la varianza cuando se muestrea z a través de MDN-RNN, lo que genera implementaciones más volátiles cuando se entrena en el entorno de los sueños. El controlador recibe mayores recompensas por estrategias más seguras que encuentran estados bien comprendidos y, por lo tanto, tienden a generalizarse mejor al entorno real. Sin embargo, es necesario equilibrar el aumento de temperatura para no hacer que el entorno sea tan volátil que el controlador no pueda aprender ninguna estrategia, ya que no hay suficiente coherencia en cómo evoluciona el entorno del sueño con el tiempo.

En el artículo original, los autores muestran esta técnica aplicada con éxito a un entorno diferente:

DoomTakeCover, basado en el juego de ordenador Doom.

La Figura 12-19 muestra cómo el cambio del parámetro de temperatura afecta tanto a la puntuación virtual (sueño) como a la puntuación real en el entorno real.

Figura 12-19. Uso de la temperatura para controlar la volatilidad del entorno de los sueños (fuente: Ha y Schmidhuber, 2018)

El ajuste de temperatura óptimo de 1,15 logra una puntuación de 1.092 en el entorno real, superando al actual líder de Gimnasio al momento de publicación. Este es un logro sorprendente; recuerde, el controlador nunca ha intentado la tarea en el entorno real. Sólo ha dado pasos aleatorios en el entorno real (para entrenar al modelo de sueño VAE y MDN-RNN) y luego usó el entorno del sueño para entrenar al controlador.

Un beneficio clave del uso de modelos mundiales generativos como enfoque para el aprendizaje por refuerzo es que cada generación de entrenamiento en el entorno onírico es mucho más rápida que el entrenamiento en el entorno real. Esto se debe a que la predicción de z y

recompensa por parte de MDN-RNN es más rápida que el cálculo de z y recompensa por parte del entorno Gym.

Resumen

En este capítulo hemos visto cómo un modelo generativo (una VAE) se puede utilizar dentro de un entorno de aprendizaje por refuerzo para permitir que un agente aprenda una estrategia efectiva probando políticas dentro de sus propios sueños generados, en lugar de dentro del entorno real.

El VAE está entrenado para aprender una representación latente del entorno, que luego se utiliza como entrada para una red neuronal recurrente que pronostica trayectorias futuras dentro del espacio latente. Sorprendentemente, el agente puede utilizar este modelo generativo como un pseudoentorno para probar políticas de forma iterativa, utilizando una metodología evolutiva, que se generaliza bien al entorno real.

Para obtener más información sobre el modelo, hay una excelente explicación interactiva disponible en línea, escrita por los autores del artículo original.

1 David Ha y Jürgen Schmidhuber, “World Models”, 27 de marzo de 2018, <https://arxiv.org/abs/1803.10122>.

2 David Ha, “Una guía visual de estrategias de evolución”, 29 de octubre de 2017, <https://blog.otoro.net/2017/10/29/visual-evolution-strategies>.

Capítulo 13. Modelos multimodales

METAS DEL CAPÍTULO

En este capítulo podrás:

Aprender qué se entiende por modelo multimodal.

Explorar el funcionamiento interno de DALL.E 2, un modelo de conversión de texto a imagen a gran escala de OpenAI.

Comprender cómo CLIP y los modelos de difusión como GLIDE juega un papel integral en la arquitectura general de DALL.E 2.

Analizar las limitaciones de DALL.E 2, como destacan los autores del artículo.

Explorar la arquitectura de Imagen, un modelo de conversión de texto a imagen a gran escala de Google Brain.

Conocer el proceso de difusión latente utilizado por Stable Diffusion, un modelo de texto a imagen de código abierto.

Comprender las similitudes y diferencias entre DALL.E 2, Imagen y Difusión Estable.

Investigar DrawBench, una suite de evaluación comparativa para evaluar modelos de texto a imagen.

Conocer el diseño arquitectónico de Flamingo, un novedoso modelo de lenguaje visual de DeepMind.

Descubrir los diferentes componentes de Flamingo y aprenda cómo cada uno de ellos contribuye al modelo en su conjunto.

Explorar algunas de las capacidades de Flamingo, incluidas las indicaciones conversacionales.

Hasta ahora, hemos analizado problemas de aprendizaje generativo que se centran únicamente en una modalidad de datos: texto, imágenes o música. Hemos visto cómo las GAN y los modelos de difusión pueden generar imágenes de última generación y cómo los Transformers son pioneros en el camino para la generación de texto e imágenes.

Sin embargo, como seres humanos, no tenemos dificultades para cruzar modalidades: por ejemplo, escribir una descripción de lo que está sucediendo en una fotografía determinada, crear arte digital para representar un mundo de fantasía ficticio en un libro o hacer coincidir la banda sonora de una película con las emociones de una persona. escena dada. ¿Podemos entrenar máquinas para que hagan lo mismo?

Introducción

El aprendizaje multimodal implica entrenar modelos generativos para convertir entre dos o más tipos diferentes de datos. Algunos de los modelos generativos más impresionantes introducidos en los últimos dos años han sido de naturaleza multimodal. En este capítulo exploraremos cómo funcionan en detalle y consideraremos cómo el futuro del modelado generativo estará determinado por grandes modelos multimodales.

Exploraremos cuatro modelos diferentes de visión y lenguaje: DALL.E 2 de OpenAI; Imagen de Google Brain; Difusión Estable de Stability AI, CompVis y Runway; y Flamenco de DeepMind.

CONSEJO

El objetivo de este capítulo es explicar de manera concisa cómo funciona cada modelo, sin entrar en los detalles de cada decisión de diseño. Para obtener más información, consulte los artículos individuales de cada modelo, que explican en detalle todas las opciones de diseño y decisiones arquitectónicas.

La generación de texto a imagen se centra en producir imágenes de última generación a partir de un mensaje de texto determinado. Por ejemplo, dada la entrada "Una cabeza de brócoli hecha de plastilina, sonriendo al sol", nos

gustaría que el modelo pudiera generar una imagen que coincida con precisión con el mensaje de texto, como se muestra en la Figura 13-1.

Este es claramente un problema muy desafiante. La comprensión del texto y la generación de imágenes son difíciles de resolver por sí solas, como hemos visto en capítulos anteriores de este libro. El modelado multimodal como este presenta un desafío adicional, porque el modelo también debe aprender cómo cruzar el puente entre los dos dominios y aprender una representación compartida que le permita convertir con precisión de un bloque de texto a una imagen de alta fidelidad sin pérdida de información.

Figura 13-1. Un ejemplo de generación de texto a imagen por DALL.E 2

Además, para tener éxito, el modelo debe poder combinar conceptos y estilos que quizás nunca antes haya visto. Por ejemplo, no hay frescos de Miguel Ángel que contengan personas con cascos de realidad virtual, pero nos gustaría que nuestro modelo pudiera crear esa imagen si se lo pedimos. Del mismo modo, sería deseable que el modelo infiriera con precisión cómo se relacionan entre sí los objetos de la imagen generada, basándose en el mensaje de texto. Por ejemplo, una imagen de “un astronauta montando un donut a través del espacio” debería verse muy diferente de la de “un astronauta comiendo un donut en un espacio lleno de gente”. El modelo debe aprender cómo se les da significado a las palabras a través del contexto y cómo convertir relaciones textuales explícitas entre entidades en imágenes que implican el mismo significado.

DALL.E 2

El primer modelo que exploraremos es DALL.E 2, un modelo diseñado por OpenAI para la generación de texto a imagen. La primera versión de este modelo, DALL.E.1 fue lanzada en febrero de 2021 y provocó una nueva ola de interés en los modelos multimodales generativos. En esta sección, investigaremos el funcionamiento de la segunda iteración del modelo, DALL.E 2.2, lanzado poco más de un año después, en abril de 2022.

DALL.E 2 es un modelo extremadamente impresionante que ha ampliado nuestra comprensión de la capacidad de la IA para resolver este tipo de problemas multimodales. No sólo tiene ramificaciones académicas, sino

que también nos obliga a plantearnos grandes preguntas relacionadas con el papel de la IA en procesos creativos que antes se pensaba que eran exclusivos de los humanos. Comenzaremos explorando cómo funciona DALL.E 2, basándose en ideas fundamentales clave que ya exploramos anteriormente en este libro.

Arquitectura

Para comprender cómo funciona DALL.E 2, primero debemos examinar su arquitectura general, como se muestra en la Figura 13-2.

Figura 13-2. La arquitectura DALL.E 2

Hay tres partes distintas a considerar: el codificador de texto, el anterior y el decodificador. Primero, el texto pasa a través del codificador de texto para producir un vector de incrustación de texto. Luego, este vector es transformado por el anterior para producir un vector de incrustación de imagen. Finalmente, esto pasa por el decodificador, junto con el texto original, para producir la imagen generada. Revisaremos cada componente por separado para obtener una imagen completa de cómo funciona DALL.E 2 en la práctica.

El codificador de texto

El objetivo del codificador de texto es convertir el mensaje de texto en un vector de incrustación que represente el significado conceptual del mensaje de texto dentro de un espacio latente. Como hemos visto en capítulos anteriores, convertir texto discreto a un vector espacial latente continuo es esencial para todas las tareas posteriores, porque podemos continuar manipulando el vector más dependiendo de nuestro objetivo particular.

En DALL.E 2, los autores no entrena n el codificador de texto desde cero, sino que utilizan un modelo existente llamado Preentrenamiento de imágenes y lenguaje contrastivo (CLIP), también producido por OpenAI. Por lo tanto, para comprender el codificador de texto, primero debemos comprender cómo funciona CLIP.

ACORTAR

CLIP3 fue presentado en un artículo publicado por OpenAI en febrero de 2021 (solo unos días después del primer artículo de DALL.E) que lo describió como "una red neuronal que aprende de manera eficiente conceptos visuales a partir de la supervisión del lenguaje natural".

Utiliza una técnica llamada aprendizaje contrastivo para unir imágenes con descripciones de texto. El modelo se entrena con un conjunto de datos de 400 millones de pares de texto e imagen extraídos de Internet; en la Figura 13-3 se muestran algunos pares de ejemplo.

A modo de comparación, hay 14 millones de imágenes anotadas a mano en ImageNet. Dada una imagen y una lista de posibles descripciones de texto, su tarea es encontrar la que realmente coincide con la imagen.

Figura 13-3. Ejemplos de pares texto-imagen

La idea clave detrás del aprendizaje contrastivo es simple. Entrenamos dos redes neuronales: un codificador de texto que convierte texto en una incrustación de texto y un codificador de imágenes que convierte una imagen en una incrustación de imagen. Luego, dado un lote de pares de texto-imagen, comparamos todas las combinaciones de incrustación de texto e imagen usando similitud de coseno y entrenamos las redes para maximizar la puntuación entre pares de texto-imagen coincidentes y minimizar la puntuación entre pares de texto-imagen incorrectos.

Este proceso se muestra en la Figura 13-4.

EL CLIP NO ES GENERATIVO

Tenga en cuenta que CLIP no es en sí mismo un modelo generativo: no puede producir imágenes ni texto. ¿Está más cerca de un modelo discriminativo, porque el resultado final es una predicción sobre qué descripción de texto de un conjunto determinado coincide más con una imagen determinada (o al revés, qué imagen coincide más con una descripción de texto determinada)?

Figura 13-4. El proceso de formación CLIP

Tanto el codificador de texto como el codificador de imágenes son Transformadores: el codificador de imágenes es un Vision Transformer (ViT), introducido en “ViT VQ-GAN”, que aplica el mismo concepto de atención a las imágenes. Los autores probaron otras arquitecturas de modelos, pero descubrieron que esta combinación produce los mejores resultados.

Lo que hace que CLIP sea especialmente interesante es la forma en que se puede utilizar para realizar predicciones cero en tareas a las que nunca ha estado expuesto. Por ejemplo, supongamos que queremos usar CLIP para predecir la etiqueta de una imagen determinada en el conjunto de datos de ImageNet. Primero podemos convertir las etiquetas de ImageNet en oraciones usando una plantilla (por ejemplo, “una foto de una”), como se muestra en la Figura 13-5.

Figura 13-5. Convertir etiquetas en un nuevo conjunto de datos en títulos para producir incrustaciones de texto CLIP

Para predecir la etiqueta de una imagen determinada, podemos pasarla a través del codificador de imágenes CLIP y calcular la similitud del coseno entre la imagen incrustada y todas las posibles incrustaciones de texto para encontrar la etiqueta con la puntuación máxima, como se muestra en la Figura 13-6.

Figura 13-6. Usando CLIP para predecir el contenido de una imagen

Tenga en cuenta que no es necesario volver a entrenar ninguna de las redes neuronales CLIP para que sea fácilmente aplicable a nuevas tareas.

Utiliza el lenguaje como dominio común a través del cual se puede expresar cualquier conjunto de etiquetas.

Con este enfoque, es posible demostrar que CLIP funciona bien en una amplia gama de desafíos de etiquetado de conjuntos de datos de imágenes (Figura 13-7). Otros modelos que han sido entrenados en un conjunto de datos específico para predecir un conjunto de etiquetas determinado a menudo fallan cuando se aplican a diferentes conjuntos de datos con las mismas etiquetas porque están altamente optimizados para los conjuntos de

datos individuales en los que fueron entrenados. CLIP es mucho más robusto, ya que ha adquirido una profunda comprensión conceptual de imágenes y descripciones de texto completo, en lugar de simplemente sobresalir en la estrecha tarea de asignar una única etiqueta a una imagen determinada en un conjunto de datos.

Figura 13-7. CLIP funciona bien en una amplia gama de conjuntos de datos de etiquetado de imágenes (fuente: Radford et al., 2021)

Como se mencionó, CLIP se mide por su capacidad discriminativa, entonces, ¿cómo nos ayuda a construir modelos generativos como DALL.E 2?

La respuesta es que podemos tomar el codificador de texto entrenado y usarlo como parte de un modelo más grande como DALL.E 2, con pesos congelados. El codificador entrenado es simplemente un modelo generalizado para convertir texto en una incrustación de texto, lo que debería ser útil para tareas posteriores, como generar imágenes. El codificador de texto es capaz de capturar una rica comprensión conceptual del texto, ya que ha sido entrenado para ser lo más similar posible a su contraparte de incrustación de imágenes coincidentes, que se produce solo a partir de la imagen emparejada. Por lo tanto, es la primera parte del puente que debemos poder cruzar del dominio del texto al dominio de la imagen.

El anterior

La siguiente etapa del proceso consiste en convertir el texto incrustado en una incrustación de imagen CLIP. Los autores de DALL.E 2 probaron dos métodos diferentes para entrenar el modelo anterior:

Un modelo autorregresivo

Un modelo de difusión

Descubrieron que el enfoque de difusión superaba al modelo autorregresivo y era más eficiente desde el punto de vista computacional. En esta sección, veremos ambos y veremos en qué se diferencian.

Autoregresivo previo

Un modelo autorregresivo genera resultados de forma secuencial, ordenando los tokens de salida (por ejemplo, palabras, píxeles) y condicionando el siguiente token a los tokens anteriores.

Hemos visto en capítulos anteriores cómo se usa esto en redes neuronales recurrentes (por ejemplo, LSTM), Transformers y PixelCNN.

El anterior autorregresivo de DALL.E 2 es un transformador codificador-decodificador. Está capacitado para reproducir la incrustación de imágenes CLIP dada una incrustación de texto CLIP, como se muestra en la Figura 13-8. Tenga en cuenta que hay algunos componentes adicionales al modelo autorregresivo mencionado en el artículo original que omitimos aquí por motivos de concisión.

Figura 13-8. Un diagrama simplificado del prior autorregresivo de DALL.E 2

El modelo se entrena en el conjunto de datos de pares de texto e imagen CLIP.

Puedes considerarlo como la segunda parte del puente que necesitamos para saltar del dominio de texto al dominio de imagen: estamos convirtiendo un vector del espacio latente que incorpora texto al espacio latente que incorpora imágenes.

El codificador del Transformer procesa la incrustación de texto de entrada para producir otra representación que se envía al decodificador, junto con la incrustación de imagen de salida generada actualmente. La salida se genera un elemento a la vez, utilizando la fuerza del profesor para comparar el siguiente elemento previsto con la incrustación de imagen CLIP real.

La naturaleza secuencial de la generación significa que el modelo autorregresivo es menos eficiente desde el punto de vista computacional que el otro método probado por los autores, que veremos a continuación.

Difusión previa

Como vimos en el Capítulo 8, los modelos de difusión se están convirtiendo rápidamente en la opción preferida por los profesionales del modelado generativo, junto con los Transformers. En DALL.E 2 solo un decodificador transformador se utiliza como previo, entrenado mediante un proceso de difusión.

El proceso de capacitación y generación se muestra en la Figura 139. Nuevamente, esta es una versión simplificada; El artículo original contiene todos los detalles de cómo está estructurado el modelo de difusión.

Figura 13-9. Un diagrama simplificado del proceso de difusión, formación previa y generación de DALL.E 2

Durante el entrenamiento, cada par de incrustación de imágenes y texto CLIP se concatena primero en un único vector. Luego, se aplica ruido a la incrustación de la imagen en más de 1000 pasos de tiempo hasta que no se puede distinguir del ruido aleatorio. Luego, la difusión previa se entrena para predecir la incrustación de la imagen sin ruido en el paso de tiempo anterior. El prior tiene acceso al texto incrustado en todas partes, por lo que es capaz de condicionar sus predicciones a esta información, transformando gradualmente el ruido aleatorio en una incrustación de imagen CLIP predicha. La función de pérdida es el error cuadrático medio promedio en los pasos de eliminación de ruido.

Para generar nuevas incrustaciones de imágenes, tomamos muestras de un vector aleatorio, anteponemos la incrustación de texto relevante y lo pasamos a través de la difusión entrenada varias veces.

El decodificador

La parte final de DALL.E 2 es el decodificador. Esta es la parte del modelo que genera la imagen final condicionada al mensaje de texto y la salida de incrustación de imagen predicha por el anterior.

La arquitectura y el proceso de entrenamiento del decodificador se basan en un artículo anterior de OpenAI, publicado en diciembre de 2021, que presentó un modelo generativo llamado Lenguaje guiado a la difusión de imágenes para generación y Edición (GLIDE). [4]

GLIDE es capaz de generar imágenes realistas a partir de indicaciones de texto, de forma muy parecida a como lo hace DALL.E 2. La diferencia es que GLIDE no utiliza incrustaciones CLIP, sino que trabaja directamente con el mensaje de texto sin formato, entrenando todo el modelo desde cero, como se muestra en la figura 13-10.

Figura 13-10. Una comparación entre DALL.E 2 y GLIDE: GLIDE entrena todo el modelo generativo desde cero, mientras que DALL.E 2 utiliza incrustaciones CLIP para transmitir información desde el mensaje de texto inicial

Veamos primero cómo funciona GLIDE.

GLIDE

GLIDE está entrenado como modelo de difusión, con arquitectura U-Net para el eliminador de ruido y arquitectura Transformer para el codificador de texto. Aprende a deshacer el ruido agregado a una imagen, guiado por el mensaje de texto. Finalmente, se entrena un Upsampler para escalar la imagen generada a 1024×1024 píxeles.

GLIDE entrena el modelo de 3.500 millones de parámetros (B) desde cero: 2.300 millones de parámetros para la parte visual del modelo (U-Net y Upsampler) y 1.200 millones para Transformer. Está entrenado en 250 millones de pares de texto e imagen.

El proceso de difusión se muestra en la Figura 13-11. A

Transformer se utiliza para crear una incrustación del mensaje de texto de entrada, que luego se utiliza para guiar a U-Net durante todo el proceso de eliminación de ruido. Exploramos la arquitectura U-Net en el Capítulo 8; es una elección de modelo perfecta cuando el tamaño total de la imagen debe permanecer igual (por ejemplo, para transferencia de estilo, eliminación de ruido, etc.).

Figura 13-11. El proceso de difusión GLIDE

El decodificador DALL.E 2 todavía utiliza el eliminador de ruido U-Net y las arquitecturas de codificador de texto Transformer, pero además tiene las incrustaciones de imágenes CLIP previstas para condicionar.

Esta es la diferencia clave entre GLIDE y DALL.E 2, como se muestra en la Figura 13-12.

Figura 13-12. El decodificador DALL.E 2 condiciona adicionalmente la incrustación de imagen producida por el anterior

Como ocurre con todos los modelos de difusión, para generar una nueva imagen, simplemente tomamos muestras de un poco de ruido aleatorio y lo pasamos por el eliminador de ruido U-Net varias veces, condicionado a la codificación de texto transformador e incrustación de imágenes. El resultado es una imagen de 64×64 píxeles.

Sobremuestreador

La parte final del decodificador es el Upsampler (dos modelos de difusión separados). El primer modelo de difusión transforma la imagen de 64×64 a 256×256 píxeles. El segundo lo transforma nuevamente, de 256×256 a $1.024 \times$

1.024 píxeles, como se muestra en la Figura 13-13.

El muestreo ascendente es útil porque significa que no tenemos que crear grandes modelos ascendentes para manejar imágenes de alta dimensión. Podemos trabajar con imágenes pequeñas hasta las etapas finales del proceso, cuando aplicamos los Upsamplers. Esto ahorra parámetros del modelo y garantiza un proceso de capacitación ascendente más eficiente.

Figura 13-13. El primer modelo de difusión de Upsampler convierte la imagen de 64×64 píxeles a 256×256 píxeles, mientras que el segundo convierte de 256×256 píxeles a 1.024×1.024 píxeles.

¡Con esto concluye la explicación del modelo DALL.E 2! En resumen, DALL.E 2 utiliza el modelo CLIP previamente entrenado para producir inmediatamente una incrustación de texto del mensaje de entrada. Luego lo

convierte en una imagen incrustada utilizando un modelo de difusión llamado anterior. Por último, implementa un modelo de difusión estilo GLIDE para generar la imagen de salida, condicionado a la incrustación de la imagen prevista y al mensaje de entrada codificado por Transformer.

Ejemplos de DALL.E 2

Se pueden encontrar ejemplos de más imágenes generadas por DALL.E 2 en el sitio web oficial. La forma en que el modelo es capaz de combinar conceptos complejos y dispares de una manera realista y creíble es asombrosa y representa un importante avance para la IA y el modelado generativo.

En el artículo, los autores muestran cómo el modelo se puede utilizar para fines adicionales además de la generación de texto a imagen. Una de estas aplicaciones es la creación de variaciones de una imagen determinada, que exploramos en la siguiente sección.

Variaciones de imagen

Como se discutió anteriormente, para generar imágenes usando el decodificador DALL.E 2 tomamos muestras de una imagen que consiste en ruido aleatorio puro y luego reducimos gradualmente la cantidad de ruido usando el modelo de difusión de eliminación de ruido, condicionado a la incrustación de la imagen proporcionada. La selección de diferentes muestras de ruido aleatorio iniciales dará como resultado imágenes diferentes.

Por lo tanto, para generar variaciones de una imagen determinada, solo necesitamos establecer su incrustación de imagen para alimentar al decodificador. Podemos obtener esto usando el codificador de imágenes CLIP original, que está diseñado explícitamente para convertir una imagen en su incrustación de imagen CLIP. Este proceso se muestra en la Figura 13-14.

Figura 13-14. DALL.E 2 se puede utilizar para generar variaciones de una imagen determinada.

Importancia de lo previo

Otra vía explorada por los autores es establecer la importancia del prior. El propósito de lo anterior es proporcionar al decodificador una representación útil de la imagen que se generará, haciendo uso del modelo CLIP previamente entrenado. Sin embargo, es factible que este paso no sea necesario; tal vez podríamos simplemente pasar el texto incrustado directamente al decodificador en lugar de la incrustación de la imagen, o ignorar las incrustaciones CLIP completamente y condición solo en el mensaje de texto. ¿Esto impactaría la calidad de las generaciones?

Para probar esto, los autores probaron tres enfoques diferentes:

1. Alimente el decodificador solo con el mensaje de texto (y un vector cero para la incrustación de la imagen).
1. Alimente el decodificador con el mensaje de texto y la incrustación de texto (como si fuera una incrustación de imagen).
1. Alimente el decodificador con el mensaje de texto y la imagen incrustada (es decir, el modelo completo).

Los resultados de ejemplo se muestran en la Figura 13-15. Podemos ver que cuando el decodificador carece de información de incrustación de imágenes, solo puede producir una aproximación aproximada del mensaje de texto, faltando información clave como la calculadora. Usar la incrustación de texto como si fuera una incrustación de imágenes funciona ligeramente mejor, aunque no es capaz de capturar la relación entre el erizo y la calculadora. Solo el modelo completo con el anterior produce una imagen que refleja con precisión toda la información contenida en el mensaje.

Figura 13-15. Lo anterior proporciona al modelo contexto adicional y ayuda al decodificador a producir generaciones más precisas (fuente: Ramesh et al., 2022)

Limitaciones

En el artículo DALL.E 2, los autores también destacan varias limitaciones conocidas del modelo. Dos de ellos (vinculación de atributos y generación de texto) se muestran en la Figura 13-16.

Figura 13-16. Dos limitaciones de DALL.E 2 residen en su capacidad para vincular atributos a objetos y reproducir información textual: mensaje superior: "Un cubo rojo encima de un cubo azul"; Mensaje inferior: "Un letrero que dice aprendizaje profundo" (fuente: Ramesh et al., 2022)

La vinculación de atributos es la capacidad de un modelo para comprender la relación entre las palabras en un mensaje de texto determinado y, en particular, cómo se relacionan los atributos con los objetos. Por ejemplo, el mensaje "Un cubo rojo encima de un cubo azul" debe aparecer visualmente distinto de "Un cubo azul encima de un cubo rojo".

DALL.E tiene algunas dificultades con esto, en comparación con modelos anteriores como GLIDE, aunque la calidad general de las generaciones es mejor y más diversa.

Además, DALL.E 2 no puede reproducir texto con precisión; esto probablemente se deba al hecho de que las incrustaciones de CLIP no capturan la ortografía, sino que solo contienen una representación de nivel superior del texto. Estas representaciones se pueden decodificar en texto con éxito parcial (p. ej., las letras individuales son en su mayoría correctas), pero no con suficiente comprensión de la composición para formar palabras completas.

Imagen

Poco más de un mes después de que OpenAI lanzara DALL.E 2, el equipo de Google Brain lanzó su propio modelo de conversión de texto a imagen llamado Imagen. [5] Muchos de los temas centrales que ya hemos explorado en este capítulo también son relevantes para Imagen: por ejemplo, utiliza un codificador de texto y un decodificador de modelo de difusión.

En la siguiente sección, exploraremos la arquitectura general de Imagen y vamos a compararla con DALL.E 2.

Arquitectura

Una descripción general de la arquitectura de Imagen se muestra en la Figura 13-17.

Figura 13-17. La arquitectura Imagen (fuente: Saharia et al., 2022)

El codificador de texto congelado es el modelo T5-XXL previamente entrenado, un transformador codificador-decodificador grande. A diferencia de CLIP, este se entrenó solo con texto y no con imágenes, por lo que no es un modelo multimodal. Sin embargo, los autores descubrieron que todavía funciona extremadamente bien como codificador de texto para Imagen y que escalar este modelo tiene más impacto en el rendimiento general que escalar el decodificador del modelo de difusión.

Al igual que el de DALL.E 2, el modelo de difusión de decodificación de Imagen se basa en una arquitectura U-Net, condicionada a la incrustación de texto. Se han realizado varias mejoras arquitectónicas en la arquitectura U-Net estándar, para producir lo que los autores llaman Efficient U-Net. Este modelo utiliza menos memoria, converge más rápido y tiene una mejor calidad de muestra que los modelos U-Net anteriores.

Los modelos de superresolución de Upsampler que toman la imagen generada de 64×64 a 1024×1024 píxeles también son modelos de difusión que continúan utilizando las incrustaciones de texto para guiar el proceso de muestreo ascendente.

DrawBench (banco de dibujo)

Una contribución adicional del artículo de Imagen es DrawBench: un conjunto de 200 indicaciones de texto para la evaluación de texto a imagen. Las indicaciones de texto cubren 11 categorías, como Conteo (capacidad de generar un número específico de objetos), Descripción (capacidad de generar mensajes de texto largos y complejos que describen objetos) y Texto (capacidad de generar texto citado). Para comparar dos modelos, las indicaciones de texto de DrawBench se pasan a través de cada modelo y los resultados se entregan a un panel de evaluadores humanos para su evaluación según dos métricas:

Alineación

¿Qué imagen describe con mayor precisión el título?

Fidelidad

¿Qué imagen es más fotorrealista (parece más real)?

Los resultados de la evaluación humana de DrawBench se muestran en la Figura 13-18.

Tanto DALL.E 2 como Imagen son modelos notables que han hecho contribuciones significativas al campo de la generación de texto a imagen. Si bien Imagen supera a DALL.E 2 en muchos de los puntos de referencia de DrawBench, DALL.E 2 proporciona funcionalidades adicionales que no están presentes en Imagen.

Por ejemplo, debido a que DALL.E 2 utiliza CLIP (un modelo multimodal de texto-imagen), puede aceptar imágenes como entrada para generar incrustaciones de imágenes. Esto significa que DALL.E 2 puede proporcionar capacidades de edición y variación de imágenes.

Esto no es posible con Imagen; el codificador de texto es un modelo de texto puro, por lo que no hay forma de ingresar una imagen.

Figura 13-18. Comparación de Imagen y DALL.E 2 en DrawBench en cuanto a alineación y fidelidad de la imagen (fuente: Saharia et al., 2022)

Ejemplos de Imagen

En la Figura 13-19 se muestran ejemplos de generaciones de Imagen.

Figura 13-19. Ejemplo Imagen generaciones (fuente: Saharia et al., 2022)

Difusión estable

El último modelo de difusión de texto a imagen que exploraremos es Stable Diffusion, lanzado en agosto de 2022 por Stability AI, en colaboración con

el grupo de investigación de Aprendizaje y Visión por Computadora de la Universidad Ludwig Maximilian de Munich y Runway. Se diferencia de DALL.E 2 e Imagen en que su código y los pesos del modelo se han hecho públicos a través de Hugging Face. Esto significa que cualquiera puede interactuar con el modelo en su propio hardware, sin tener que utilizar API propietarias.

Arquitectura

La principal diferencia arquitectónica entre la difusión estable y los modelos de texto a imagen discutidos anteriormente es que utiliza la difusión latente como modelo generativo subyacente.

Los modelos de difusión latente (LDM) fueron introducidos por Rombach et al. en diciembre de 2021, en el artículo “Síntesis de imágenes de alta resolución con modelos de difusión latente”. [6]

La idea clave del artículo es envolver el modelo de difusión dentro de un auto codificador, de modo que el proceso de difusión opere en una representación espacial latente de la imagen en lugar de la imagen misma, como se muestra en la Figura 13-20.

Figura 13-20. La arquitectura de difusión estable

Este avance significa que el modelo U-Net de eliminación de ruido se puede mantener relativamente liviano, en comparación con U-Net.

Modelos que operan con imágenes completas. El auto codificador maneja el trabajo pesado de codificar los detalles de la imagen en el espacio latente y decodificar el espacio latente nuevamente a una imagen de alta resolución, dejando que el modelo de difusión funcione puramente en un espacio conceptual latente. Esto proporciona un importante impulso de velocidad y rendimiento al proceso de entrenamiento.

Opcionalmente, el proceso de eliminación de ruido también puede guiarse mediante un mensaje de texto que ha pasado a través de un codificador de texto.

La primera versión de Stable Diffusion utilizó el modelo CLIP pre-entrenado de OpenAI (igual que en DALL.E 2), pero Difusión Estable 2 tiene un modelo CLIP entrenado personalizado llamado OpenCLIP, que ha sido entrenado desde cero.

Ejemplos de difusión estable

La Figura 13-21 muestra algunos ejemplos de salidas de Difusión Estable 2.1: puedes probar tus propias indicaciones a través del modelo alojado en Hugging Face.

Figura 13-21. Ejemplos de resultados de Stable Diffusion 2.1

EXPLORANDO EL ESPACIO LATENTE

Si desea explorar el espacio latente del modelo de Difusión Estable, le recomiendo encarecidamente el tutorial en el sitio web de Keras.

Flamenco

Hasta ahora hemos analizado tres tipos diferentes de modelos de texto a imagen. En esta sección, exploraremos un modelo multimodal que genera texto dado un flujo de texto y datos visuales. Flamingo, presentado en un artículo de DeepMind en abril del 2022,⁷ es una familia de modelos de lenguaje visual (VLM) que actúan como un puente entre los modelos previamente entrenados de solo visión y de solo lenguaje.

En esta sección, repasaremos la arquitectura de modelos de Flamingo y compárelos con los modelos de texto a imagen que hemos visto hasta ahora.

Arquitectura

La arquitectura general de Flamingo se muestra en la Figura 1322. Para ser concisos, exploraremos los componentes centrales de este modelo: el Codificador de visión, el Perceptor Remuestreador y Modo Lenguaje, con el detalle suficiente para resaltar las ideas clave que hacen que Flamingo

sea único. Recomiendo encarecidamente leer el artículo de investigación original para obtener una revisión exhaustiva de cada parte del modelo.

Figura 13-22. La arquitectura Flamingo (fuente: Alayrac et al., 2022)

El codificador de visión

La primera diferencia entre un modelo Flamingo y los modelos puros de texto a imagen como DALL.E 2 e Imagen es que Flamingo puede aceptar una combinación de texto y datos visuales intercalados. Aquí, los datos visuales incluyen videos e imágenes.

El trabajo del Vision Encoder es convertir los datos de visión dentro de la entrada en vectores de incrustación (similar al codificador de imágenes en CLIP). Vision Encoder en Flamingo es una ResNet sin normalizador (NFNet) previamente entrenada, como lo presentaron Brock et al. en 20218 —en particular, una NFNet-F6 (los modelos NFNet van desde F0 a F6, aumentando en tamaño y potencia). Esta es una diferencia clave entre el codificador de imágenes CLIP y Codificador Flamingo Vision: el primero utiliza una arquitectura ViT, mientras que el segundo utiliza una arquitectura ResNet.

Vision Encoder se entrena en pares de imagen y texto utilizando el mismo objetivo contrastivo que se presenta en el artículo CLIP.

Después del entrenamiento, los pesos se congelan para que cualquier entrenamiento adicional del modelo Flamingo no afecte los pesos del Vision Encoder.

La salida del Vision Encoder es una cuadrícula 2D de características que luego se aplana a un vector 1D antes de pasar al Perceiver Resampler. El vídeo se maneja muestreando 1 fotograma por segundo y pasando cada instantánea a través del Vision Encoder de forma independiente para producir varias cuadrículas de funciones; Luego se agregan las codificaciones temporales aprendidas antes de aplanar las características y concatenar los resultados en un solo vector.

El remuestreador del perceptor

Los requisitos de memoria en un codificador tradicional Transformer (por ejemplo, BERT) escalan cuadráticamente con la longitud de la secuencia de entrada, razón por la cual las secuencias de entrada normalmente están limitadas a un número determinado de tokens (por ejemplo, 512 en BERT). Sin embargo, la salida del Vision Encoder es un vector de longitud variable (debido a la resolución variable de la imagen de entrada y al número variable de fotogramas de vídeo) y, por lo tanto, es potencialmente muy larga.

La arquitectura Perceiver está diseñada específicamente para manejar de manera eficiente secuencias de entrada largas. En lugar de realizar la autoatención en la secuencia de entrada completa, funciona con un vector latente de longitud fija y solo utiliza la secuencia de entrada para atención cruzada. En concreto, en el Perceptor Remuestreador Flamingo Perceiver, la clave y el valor son una concatenación de la secuencia de entrada y el vector latente y la consulta es solo el vector latente. Un diagrama del proceso de codificador y percepto remuestreador Visión para datos de video se muestra en la Figura 13-23.

Figura 13-23. El Perceiver Resampler aplicado a la entrada de vídeo (fuente: Alayrac et al., 2022)

La salida del Perceiver Resampler es un vector latente de longitud fija que se pasa al modelo de lenguaje.

El modelo del lenguaje

El modelo de lenguaje consta de varios bloques apilados, al estilo de un decodificador Transformer, que genera una continuación de texto predicha. De hecho, la mayoría del modelo del lenguaje proviene de un modelo DeepMind previamente entrenado llamado Chinchilla. El artículo de Chinchilla, publicado en marzo de 2022,⁹ muestra un modelo de lenguaje que está diseñado para ser considerablemente más pequeño que sus pares (por ejemplo, 70 mil millones de parámetros para Chinchilla en comparación con 170 mil millones para GPT-3), al tiempo que utiliza significativamente más tokens para el entrenamiento. Los autores muestran que el modelo supera a los modelos más grandes en una variedad de tareas, destacando la importancia de optimizar el equilibrio entre entrenar un

modelo más grande y usar una mayor cantidad de tokens durante el entrenamiento.

Una contribución clave del artículo de Flamingo es mostrar cómo Chinchilla se puede adaptar para trabajar con datos de visión adicionales (X) que se intercalan con los datos del lenguaje (Y).

Primero exploraremos cómo se combinan las entradas del lenguaje y la visión para producir la entrada al modelo de lenguaje (Figura 13-24).

Primero, el texto se procesa reemplazando los datos de visión (por ejemplo, imágenes) con una etiqueta y el texto se divide en fragmentos usando la etiqueta (fin de fragmento). Cada fragmento contiene como máximo una imagen, que siempre está al comienzo del fragmento; es decir, se supone que el texto posterior se relaciona únicamente con esa imagen. El comienzo de la secuencia también está marcado con la etiqueta (comienzo de oración).

A continuación, se tokeniza la secuencia y a cada token se le asigna un índice (ϕ) correspondiente al índice de la imagen anterior (o 0 si no hay ninguna imagen anterior en el fragmento). De esta manera, se puede forzar a los tokens de texto (Y) a que solo atiendan de forma cruzada los tokens de imagen (X) que corresponden a su fragmento particular, mediante enmascaramiento. Por ejemplo, en la Figura 13-24 el primer fragmento no contiene imágenes, por lo que todos los tokens de imagen del Perceptor Remuestreador están enmascarados. El segundo fragmento contiene la imagen 1, por lo que estos tokens pueden interactuar con los tokens de imagen de la imagen 1. Del mismo modo, el fragmento final contiene la imagen 2, por lo que estos tokens pueden interactuar con los tokens de imagen de la imagen 2.

Figura 13-24. Atención cruzada enmascarada (XATTN), que combina datos de visión y texto: las entradas de color azul claro están enmascaradas y las entradas de color azul oscuro no están enmascaradas (fuente:

Alayrac et al., 2022)

Ahora podemos ver cómo este componente de atención cruzada enmascarada encaja en la arquitectura general del modelo de lenguaje (Figura 13-25).

Los componentes de la capa LM azul son capas congeladas de Chinchilla: estos no se actualizan durante el proceso de capacitación. Las capas violetas Gated XATTN-DENSE se entrenan como parte de Flamingo e incluyen los componentes de atención cruzada enmascarados que combinan la información del lenguaje y la visión, así como capas posteriores de alimentación directa (densa).

La capa está cerrada porque pasa la salida de los componentes de atención cruzada y retroalimentación a través de dos puertas tanh distintas, ambas inicializadas a cero.

Por lo tanto, cuando se inicializa la red, no hay ninguna contribución de las capas Gated XATTN-DENSE: la información del idioma simplemente pasa directamente. La red aprende los parámetros de activación alfa para combinar gradualmente la información de los datos de visión a medida que avanza el entrenamiento.

Figura 13-25. Un bloque de modelo de lenguaje Flamingo, que comprende una capa de modelo de lenguaje congelado de Chinchilla y una capa Gated XATTN-DENSE (fuente: Alayrac et al., 2022)

Ejemplos de flamingo

Flamingo se puede utilizar para una variedad de propósitos, incluida la comprensión de imágenes y videos, indicaciones conversacionales y diálogo visual. En la Figura 13-26 podemos ver algunos ejemplos de lo que Flamingo es capaz de hacer.

Figura 13-26. Ejemplos de entradas y salidas obtenidas del parámetro 80B del modelo Flamingo (fuente: Alayrac et al., 2022)

Observe cómo en cada ejemplo, Flamingo combina información del texto y las imágenes en un verdadero estilo multimodal. El primer ejemplo utiliza imágenes en lugar de palabras y puede sugerir un libro apropiado para

continuar con la indicación. El segundo ejemplo muestra fotogramas de un vídeo y Flamingo identifica correctamente la consecuencia de la acción. Los últimos tres ejemplos demuestran cómo se puede utilizar Flamingo de forma interactiva, para proporcionar información adicional a través del diálogo o sondear con preguntas adicionales.

Es sorprendente ver que una máquina sea capaz de responder preguntas complejas en una gama tan amplia de modalidades y tareas de entrada. En el artículo, los autores cuantifican la capacidad de Flamingo en un conjunto de tareas de referencia y descubrimos que en muchas pruebas, Flamingo es capaz de superar el rendimiento de los modelos que han sido diseñados para abordar específicamente la tarea en cuestión. Esto destaca cómo los grandes modelos multimodales pueden adaptarse rápidamente a una amplia gama de tareas y allana el camino para el desarrollo de agentes de IA que no estén vinculados simplemente a una sola tarea, sino que sean agentes verdaderamente generales que pueden ser guiados por el usuario. en el momento de la inferencia.

Resumen

En este capítulo hemos explorado cuatro modelos multimodales de última generación diferentes: DALL.E 2, Imagen, Difusión estable y Flamingo.

DALL.E 2 es un modelo de conversión de texto a imagen a gran escala de OpenAI que puede generar imágenes realistas en una variedad de estilos mediante un mensaje de texto. Funciona combinando modelos previamente entrenados (por ejemplo, CLIP) con arquitecturas de modelos de difusión de trabajos anteriores (GLIDE). También tiene capacidades adicionales, como poder editar imágenes mediante mensajes de texto y proporcionar variaciones de una imagen determinada. Si bien tiene algunas limitaciones, como la representación de texto inconsistente y la vinculación de atributos, DALL.E 2 es un modelo de IA increíblemente poderoso que ha ayudado a impulsar el campo del modelado generativo hacia una nueva era.

Otro modelo que ha superado los puntos de referencia anteriores es Imagen de Google Brain. Este modelo comparte muchas similitudes con DALL.E 2, como un codificador de texto y un decodificador de modelo de difusión. Una de las diferencias clave entre los dos modelos es que el codificador de

texto Imagen se entrena con datos de texto puro, mientras que el proceso de entrenamiento para el codificador de texto DALL.E 2 involucra datos de imagen (a través del objetivo de aprendizaje CLIP contrastivo). Los autores muestran que este enfoque conduce a un rendimiento de vanguardia en una variedad de tareas, a través de su conjunto de evaluación DrawBench.

Stable Diffusion es una oferta de código abierto de Stability IA, CompVis y Runway. Es un modelo de texto a imagen cuyos pesos y código de modelo están disponibles gratuitamente, por lo que puede ejecutarlo en su propio hardware. La difusión estable es particularmente rápida y liviana debido al uso de un modelo de difusión latente que opera en el espacio latente de un auto codificador, en lugar de en las imágenes mismas.

Finalmente, Flamingo de DeepMind es un modelo de lenguaje visual, es decir, acepta un flujo de texto y datos visuales entrelazados (imágenes y videos) y es capaz de continuar el mensaje con texto adicional, al estilo de un decodificador Transformer.

La contribución clave es mostrar cómo la información visual se puede enviar al transformador a través de un codificador visual y el preceptor remuestreador que codifica las características de entrada visual en una pequeña cantidad de tokens visuales. El modelo de lenguaje en sí es una extensión del modelo Chinchilla anterior de DeepMind, adaptado para combinar información visual.

Los cuatro son ejemplos notables del poder de los modelos multimodales. En el futuro, es muy probable que el modelado generativo se vuelva más multimodal y los modelos de IA puedan cruzar fácilmente modalidades y tareas mediante indicaciones de lenguaje interactivo.

1 Aditya Ramesh et al., “Zero-Shot Text-to-Image Generation”, febrero 24, 2021, <https://arxiv.org/abs/2102.12092>.

2 Aditya Ramesh et al., “Generación de imágenes condicionales de texto jerárquico con latentes CLIP”, 13 de abril de 2022, <https://arxiv.org/abs/2204.06125>.

3 Alec Radford et al., “Aprendizaje de modelos visuales transferibles a partir de supervisión del lenguaje”, 26 de febrero de 2021, <https://arxiv.org/abs/2103.00020>.

4 Alex Nichol et al., “GLIDE: Hacia la generación de imágenes fotorrealistas y edición con modelos de difusión guiada por texto”, 20 de diciembre de 2021, <https://arxiv.org/abs/2112.10741>.

5 Chitwan Saharia et al., “Modelos fotorrealistas de difusión de texto a imagen con comprensión profunda del lenguaje”, 23 de mayo de 2022, <https://arxiv.org/abs/2205.11487>.

6 Robin Rombach et al., “Síntesis de imágenes de alta resolución con información de modelos de difusión latente”, 20 de diciembre de 2021, <https://arxiv.org/abs/2112.10752>.

7 Jean-Baptiste Alayrac et al., “Flamingo: un modelo de lenguaje visual para aprendizaje con pocas posibilidades”, 29 de abril de 2022, <https://arxiv.org/abs/2204.14198>.

8 Andrew Brock et al., “Reconocimiento de imágenes a gran escala de alto rendimiento sin normalización”, 11 de febrero de 2021, <https://arxiv.org/abs/2102.06171>.

9 Jordan Hoffmann et al., “Entrenando grandes modelos de lenguaje computacionalmente óptimos”, 29 de marzo de 2022, <https://arxiv.org/abs/2203.15556v1>.

Capítulo 14. Conclusión

METAS DEL CAPÍTULO

En este capítulo podrás:

Revise la historia de la IA generativa desde 2014 hasta la actualidad, incluida una cronología de modelos y desarrollos clave.

Comprender el estado actual de la IA generativa, incluidos los temas generales que dominan el panorama.

Vea mis predicciones para el futuro de la IA generativa y cómo afectará la vida cotidiana, el lugar de trabajo y la educación.

Conozca los importantes desafíos éticos y prácticos que enfrentará la IA generativa en el futuro.

Lea mis pensamientos finales sobre el significado más profundo de la IA generativa y cómo tiene el potencial de revolucionar nuestra búsqueda de la inteligencia artificial general.

En mayo de 2018 comencé a trabajar en la primera edición de este libro. Cinco años después, estoy más entusiasmado que nunca con las infinitas posibilidades y el impacto potencial de la IA generativa.

Durante este tiempo hemos visto un progreso increíble en este campo, con un potencial aparentemente ilimitado para aplicaciones del mundo real. Me llena una sensación de asombro y asombro por lo que hemos podido lograr hasta ahora y anticipo con impaciencia ser testigo del efecto que la IA generativa tendrá en el mundo en los próximos años.

El aprendizaje profundo generativo tiene el poder de dar forma al futuro en formas que ni siquiera podemos empezar a imaginar.

Es más, a medida que he ido investigando el contenido de este libro, me ha quedado cada vez más claro que este campo no se trata sólo de crear

imágenes, texto o música. Creo que en el centro del aprendizaje profundo generativo se encuentra el secreto de la inteligencia misma.

La primera sección de este capítulo resume cómo hemos llegado a este punto en nuestro viaje de IA generativa. Recorremos una línea de tiempo de los desarrollos de la IA generativa desde 2014 en orden cronológico, para que pueda ver dónde encaja cada técnica en la historia de la IA generativa hasta la fecha. La segunda sección explica dónde nos encontramos actualmente en términos de IA generativa de última generación. Discutiremos las tendencias actuales en el enfoque del aprendizaje profundo generativo y los modelos disponibles actualmente disponibles para el público en general.

A continuación, exploraremos el futuro de la IA generativa y las oportunidades y desafíos que tenemos por delante. Consideraremos cómo podría ser la IA generativa dentro de cinco años y su impacto potencial en la sociedad y los negocios, y abordaremos algunas de las principales preocupaciones éticas y prácticas.

Cronología de la IA generativa

La figura 14-1 es una cronología de los desarrollos clave en el modelado generativo que hemos explorado juntos en este libro. Los colores representan diferentes tipos de modelos.

El campo de la IA generativa se apoya en avances anteriores en el aprendizaje profundo, como la retropropagación y las redes neuronales convolucionales, que abrieron la posibilidad de que los modelos aprendan relaciones complejas a través de grandes conjuntos de datos a escala. En esta sección, estudiaremos la historia moderna de la IA generativa, desde 2014 en adelante, que ha avanzado a una velocidad impresionante.

Para ayudarnos a comprender cómo encaja todo, podemos dividir esta historia en tres épocas principales:

1. 2014-2017: la era VAE y GAN
2. 2018-2019: la era de los transformadores
3. 2020-2022: la era del gran modelo

Figura 14-1. Una breve historia de la IA generativa de 2014 a 2023 (nota: algunos desarrollos importantes, como los LSTM y los primeros modelos basados en energía [por ejemplo, las máquinas Boltzmann] preceden a esta línea de tiempo)

2014-2017: la era VAE y GAN

La invención del VAE en diciembre de 2013 quizás pueda considerarse como la chispa que encendió el papel táctil de la IA generativa. Este artículo mostró cómo era posible generar no sólo imágenes simples, como dígitos MNIST, sino también imágenes más complejas, como rostros, en un espacio latente que podía atravesarse sin problemas. En 2014 le siguió la introducción de GAN, un marco adversario completamente nuevo para abordar problemas de modelado generativo.

Los tres años siguientes estuvieron dominados por ampliaciones cada vez más impresionantes de la cartera de GAN. Además de los cambios fundamentales en la arquitectura del modelo GAN (DCGAN, 2015), función de pérdida (Wasserstein GAN, 2017) y proceso de entrenamiento (ProGAN, 2017), se abordaron nuevos dominios utilizando GAN, como la traducción de imagen a imagen (pix2pix, 2016 y CycleGAN, 2017) y la generación de música (MuseGAN, 2017).

Durante esta era también se introdujeron importantes mejoras en VAE, como VAE-GAN (2015) y posteriormente VQ-VAE (2017), y se vieron aplicaciones al aprendizaje por refuerzo en los “modelos mundiales” (2018).

Los modelos autorregresivos establecidos, como LSTM y GRU, siguieron siendo la fuerza dominante en la generación de texto durante este tiempo. Las mismas ideas autorregresivas también se utilizaron para generar imágenes, con PixelRNN (2016) y PixelCNN (2016) introducidas como nuevas formas de pensar sobre la generación de imágenes. También se estaban probando otros enfoques para la generación de imágenes, como el modelo RealNVP (2016), que allanó el camino para tipos posteriores de modelos de normalización de flujo.

En junio de 2017, se publicó un artículo innovador titulado “La atención es todo lo que usted necesita” que marcaría el comienzo de la próxima era de la IA generativa, centrada en transformadores.

2018-2019: la era de los transformadores

En el corazón de un Transformer se encuentra el mecanismo de atención que niega la necesidad de las capas recurrentes presentes en modelos autorregresivos más antiguos, como los LSTM. El Transformer rápidamente saltó a la fama con la introducción de GPT (un transformador sólo decodificador) y BERT (un transformador sólo codificador) en 2018.

El año siguiente se construyeron modelos de lenguaje progresivamente más grandes que sobresalieron en una amplia gama de tareas al tratarlas como problemas puros de generación de texto a texto, con GPT-2 (2018, parámetros 1.5B) y T5 (2019, parámetros 11B). siendo ejemplos destacados.

Los transformadores también comenzaron a aplicarse con éxito a la generación de música, con la introducción, por ejemplo, de los modelos Transformador Musical (2018) y MuseNet (2019).

A lo largo de estos dos años, también se lanzaron varias GAN impresionantes que consolidaron el lugar de la técnica como enfoque de última generación para la generación de imágenes. En particular, SAGAN (2018) y el BigGAN más grande (2018) incorporaron el mecanismo de atención en el marco GAN con resultados increíbles, y StyleGAN (2018) y posteriormente StyleGAN2 (2019) mostraron cómo se pueden generar imágenes con un control sorprendente y detallado sobre el estilo y el contenido de una imagen en particular.

Otro campo de la IA generativa que estaba cobrando impulso eran los modelos basados en puntuaciones (NCSN, 2019), que eventualmente allanarían el camino para el próximo cambio sísmico en el panorama de la IA generativa: los modelos de difusión.

2020-2022: la era de los grandes modelos

Esta era vio la introducción de varios modelos que fusionaron ideas de diferentes familias de modelos generativos y potenciaron las arquitecturas existentes. Por ejemplo, VQ-GAN (2020) incorporó el discriminador GAN a la arquitectura VQ-VAE y Transformador Vision (2020) mostró cómo era posible entrenar un transformador para operar sobre imágenes. 2022 vio el lanzamiento de StyleGAN-XL, una actualización adicional de la arquitectura StyleGAN que permite generar imágenes de 1.024×1.024 píxeles.

En 2020 se introdujeron dos modelos que sentarían las bases de todos los futuros modelos de generación de imágenes de gran tamaño: DDPM y DDIM.

De repente, los modelos de difusión rivalizaron con las GAN en términos de calidad de generación de imágenes, como se indica explícitamente en el título del artículo de 2021 "Los modelos de difusión superan a las GAN en síntesis de imágenes". La calidad de imagen de los modelos de difusión es increíblemente buena y solo requieren que se entrene una única red U-Net, en lugar de la configuración de red dual de una GAN, lo que hace que el proceso de capacitación sea mucho más estable.

Casi al mismo tiempo, se lanzó GPT-3 (2020), un enorme transformador de parámetro 175B que puede generar texto sobre casi cualquier tema de una manera que parece casi imposible de comprender.

El modelo se lanzó a través de una aplicación web y una API, lo que permitió a las empresas crear productos y servicios sobre él.

ChatGPT (2022) es una aplicación web y un contenedor de API para la última versión de GPT de OpenAI que permite a los usuarios tener conversaciones naturales con la IA sobre cualquier tema.

Durante 2021 y 2022, se lanzó una serie de otros modelos de lenguajes importantes para rivalizar con GPT-3, incluido Megatron-Turing NLG (2021) de Microsoft y NVIDIA, Gopher (2021) y Chinchilla de DeepMind (2022), LaMDA (2022) y PaLM (2022) de Google, y Luminous (2022) de Aleph Alpha. También se lanzaron algunos modelos de código abierto, como GPT-Neo (2021), GPT-J (2021) y GPT-NeoX (2022) de EleutherAI;

el modelo OPT del parámetro 66B (2022) por Meta; el modelo Flan-T5 perfeccionado (2022) de Google, BLOOM (2022) de Hugging Face; y otros. Cada uno de estos modelos es una variación de un Transformer, entrenado con un enorme corpus de datos.

El rápido aumento de potentes transformadores para la generación de texto y de modelos de difusión de última generación para la generación de imágenes ha significado que gran parte del enfoque de los últimos dos años de desarrollo de la IA generativa se haya centrado en modelos multimodales, es decir, modelos que operan en más de un dominio (por ejemplo, modelos de texto a imagen).

Esta tendencia se estableció en 2021 cuando OpenAI lanzó DALL.E, un modelo de texto a imagen basado en un VAE discreto (similar a VQVAE) y CLIP (un modelo Transformer que predice pares de imagen/texto). A esto le siguieron GLIDE (2021) y DALL.E 2 (2022), que actualizaron la parte generativa del modelo para utilizar un modelo de difusión en lugar de un VAE discreto, con resultados realmente impresionantes.

Esta era también vio el lanzamiento de tres modelos de conversión de texto a imagen de Google: Imagen (2022, usando modelos Transformer y de difusión), Parti (2022, usando Transformers y un modelo ViT-VQGAN), y más tarde MUSE (2023, usando Transformers y VQ-GAN). DeepMind también lanzó Flamingo (2022), un modelo de lenguaje visual que se basa en su modelo de lenguaje grande Chinchilla al permitir el uso de imágenes como parte de los datos de solicitud.

Otro avance de difusión importante introducido en 2021 fue la difusión latente, donde se entrena un modelo de difusión dentro del espacio latente de un auto codificador. Esta técnica impulsa el Modelo de difusión estable, lanzado como colaboración conjunta entre Stability AI, CompVis y Runway en 2022. A diferencia de DALL.E 2, Imagen y Flamingo, el código y los pesos del modelo de Stable Diffusion son de código abierto, lo que significa que cualquiera puede ejecutar el modelo en su propio hardware.

El estado actual de la IA generativa

Al llegar al final de nuestro viaje a través de la historia de la IA generativa, es importante reflexionar ahora sobre nuestra situación en términos de aplicaciones y modelos actuales de última generación. Tomémonos un momento para evaluar nuestro progreso y logros clave en el campo hasta la fecha.

Modelos de lenguaje grandes

La IA generativa para texto ahora se centra casi por completo en la construcción de grandes modelos de lenguaje (LLM), cuyo único propósito es modelar directamente el lenguaje a partir de un enorme corpus de texto; es decir, están entrenados para predecir la siguiente palabra, al estilo de un transformador descifrador

El enfoque del modelo de lenguaje grande se ha adoptado ampliamente debido a su flexibilidad y capacidad para sobresalir en una amplia gama de tareas. El mismo modelo se puede utilizar para responder preguntas, resumir textos, crear contenido y muchos otros ejemplos porque, en última instancia, cada caso de uso se puede enmarcar como un problema de texto a texto, donde las instrucciones de la tarea específica (el mensaje) se dan como parte de la entrada al modelo.

Tomemos GPT-3 como ejemplo. La Figura 14-2 muestra cómo se puede utilizar el mismo modelo para el resumen de texto y la creación de contenido.

Figura 14-2. Salida de GPT-3: el texto no resaltado es el mensaje y el texto resaltado en verde es la salida de GPT-3

Observe cómo en ambos casos el mensaje contiene las instrucciones relevantes. El trabajo de GPT-3 es simplemente continuar con el mensaje, un token a la vez. No tiene una base de datos de hechos en la que pueda buscar información, ni fragmentos de texto que pueda copiar en sus respuestas. Solo se le pide que prediga qué token es más probable que siga a los tokens existentes y luego agregue esta predicción al mensaje para generar el siguiente token, y así sucesivamente.

Increíblemente, este diseño simple es suficiente para que el modelo de lenguaje sobresalga en una variedad de tareas, como se muestra en la Figura 14-2. Además, le da al modelo de lenguaje una flexibilidad increíble para generar texto realista como respuesta a cualquier mensaje: ¡la imaginación es a menudo el factor limitante!

La Figura 14-3 muestra cómo los modelos de lenguaje grandes han crecido en tamaño desde que se publicó el modelo GPT original en 2018. La cantidad de parámetros creció exponencialmente hasta finales de 2021, y MegatronTuring NLG alcanzó 530B de parámetros. Recientemente, se ha puesto más énfasis en la creación de modelos de lenguaje más eficientes que utilicen menos parámetros, ya que los modelos más grandes son más costosos y más lentos de utilizar en un entorno de producción.

Figura 14-3. El tamaño de los modelos de lenguaje grandes (naranja) y los modelos multimodales (rosa) en número de parámetros a lo largo del tiempo.

Muchos todavía consideran que la colección GPT de OpenAI (GPT-3, GPT-3.5, GPT-4, etc.) es la suite de modelos de lenguaje de última generación más poderosa disponibles para uso personal y comercial. Cada uno de ellos está disponible a través de una aplicación web y API.

Otra incorporación reciente a la gran familia de modelos de lenguaje es Large Language Model Meta AI (LLaMA) de Meta,¹ un conjunto de modelos que van desde 7B a 65B de parámetros de tamaño y que se entrena exclusivamente en conjuntos de datos disponibles públicamente.

En la Tabla 14-1 se muestra un resumen de algunos de los LLM más poderosos que existen en la actualidad. Algunos, como LLaMA, son familias de modelos de diferentes tamaños; en este caso, aquí se muestra el tamaño del modelo más grande. Los pesos previamente entrenados son completamente de código abierto para algunos de los modelos, lo que significa que cualquiera puede usarlos y desarrollarlos de forma gratuita.

Modelo	Fecha	Desarrollador	parámetro	Fuente abierta

GPT-3	mayo 2020	OpenAI	175,000,000,000	No
GPT-Neo	marzo de 2021	EleutherAI	2,700,000,000	Sí
GPT-J	junio de 2021	EleutherAI	6,000,000,000	Sí
MegatrónTuring NLG	octubre de 2021	Microsoft y NVIDIA	530,000,000,000	No
Gopher	diciembre de 2021	DeepMind	280,000,000,000	No
LaMDA	enero de 2022	Google	137,000,000,000	No
GPT-NeoX	febrero de 2022	EleutherAI	20,000,000,000	Sí
Chinchilla	marzo de 2022	DeepMind	70,000,000 ,000	No
Palmera	abril de 2022	Google	540,000,000,000	No
Luminous	abril de 2022	Aleph Alfa	70,000,000,000	No
OPTAR	mayo 2022	Meta	175,000,000,000	Sí (66B)
BLOOM	julio de 2022	Hugging Face	175,000,000,000	Sí
collaboraci on Flan-T5	octubre de 2022	Google	11,000,000,000	Sí
GPT-3.5	noviembre de 2022	OpenAI	Desconocido	No
LlaMA	febrero de 2023	Meta	65,000,000,000	No
GPT-4	marzo de 2023	OpenAI	Desconocido	No

A pesar de las impresionantes aplicaciones de los grandes modelos de lenguaje, aún quedan importantes desafíos por superar. En particular, son propensos a inventar hechos y no pueden aplicar de manera confiable procesos de pensamiento lógico, como se muestra en la Figura 14-4.

Figura 14-4. Si bien los modelos de lenguaje grandes sobresalen en algunas tareas, también son propensos a cometer errores relacionados con el razonamiento fáctico o lógico (se muestra el resultado GPT-3)

Es importante recordar que los LLM están capacitados únicamente para predecir la siguiente palabra. No tienen otra conexión con la realidad que les permita identificar de manera confiable falacias fácticas o lógicas. Por lo tanto, debemos ser extremadamente cautelosos acerca de cómo utilizamos estos poderosos modelos de predicción de texto en producción; todavía no se pueden utilizar de manera confiable para nada que requiera un razonamiento preciso.

Modelos de texto a código

Otra aplicación de los modelos de lenguaje grandes es la generación de código. En julio de 2021, OpenAI presentó un modelo llamado Codex, un modelo de lenguaje GPT que se había perfeccionado en código de GitHub. [2]

El modelo pudo escribir con éxito soluciones codificadas novedosas para una variedad de problemas, con solo un comentario sobre el problema a resolver o el nombre de una función. La tecnología hoy impulsa GitHub Copilot, un programador de pares de IA que se puede utilizar para sugerir código en tiempo real mientras escribe. Copilot es un servicio de suscripción paga, con un período de prueba gratuito.

La Figura 14-5 muestra dos ejemplos de terminaciones generadas automáticamente. El primer ejemplo es una función que recupera tweets de un usuario determinado, utilizando la API de Twitter. Dado el nombre de la función y el parámetro, Copilot puede autocompletar el resto de la definición de la función.

El segundo ejemplo le pide a Copilot que analice una lista de gastos, incluyendo además una descripción de texto libre en la cadena de documentación que explica el formato del parámetro de entrada e instrucciones específicas relacionadas con la tarea. Copilot puede autocompletar toda la función solo a partir de la descripción.

Esta extraordinaria tecnología ya está empezando a cambiar la forma en que los programadores abordan una tarea determinada. Una proporción significativa del tiempo de un programador generalmente se dedica a buscar ejemplos de soluciones existentes, leer foros de preguntas y respuestas de la comunidad como Stack Overflow y búsqueda de sintaxis en la documentación del paquete. Esto significa abandonar el entorno de desarrollo interactivo (IDE) a través del cual está codificando, cambiar a un navegador web y copiar y pegar fragmentos de código de la web para ver si resuelven su problema específico. Copilot elimina la necesidad de hacer esto en muchos casos, porque simplemente puede navegar por las posibles soluciones generadas por la IA desde el IDE, después de escribir una breve descripción de lo que busca lograr.

Figura 14-5. Dos ejemplos de capacidades de GitHub Copilot (fuente: GitHub Copilot)

Modelos de texto a imagen

La generación de imágenes de última generación está actualmente dominada por grandes modelos multimodales que convierten un mensaje de texto determinado en una imagen.

Los modelos de texto a imagen son muy útiles ya que permiten a los usuarios manipular fácilmente las imágenes generadas mediante lenguaje natural. Esto contrasta con modelos como StyleGAN, que, si bien son extremadamente impresionantes, no tienen una interfaz de texto a través de la cual puedas describir la imagen que deseas generar.

Tres modelos importantes de generación de texto a imagen que están actualmente disponibles para uso comercial y personal son DALL.E 2, Midjourney y difusión estable.

DALL.E 2 de OpenAI es un servicio de pago por uso que está disponible a través de una aplicación web y API. Midjourney ofrece un servicio de conversión de texto a imagen por suscripción a través de su canal Discord. Tanto DALL.E 2 como Midjourney ofrecen créditos gratuitos a quienes se unan a la plataforma para realizar una experimentación temprana.

MIDJOURNEY

¡Midjourney es el servicio utilizado para crear las ilustraciones de las historias en la Parte II de este libro!

Difusión Estable es diferente porque es completamente de código abierto. Los pesos del modelo y el código para entrenar el modelo están disponibles en GitHub, por lo que cualquiera puede ejecutar el modelo en su propio hardware. El conjunto de datos utilizado para entrenar Stable Diffusion también es de código abierto. Este conjunto de datos, llamado LAION-5B, contiene 5.850 millones de pares de imágenes y texto y actualmente es el conjunto de datos de imágenes y texto de acceso abierto más grande del mundo.

Un corolario importante de este enfoque es que la línea base del modelo de difusión de la Difusión Estable se puede desarrollar y adaptar a diferentes casos de uso. Una excelente demostración de esto es ControlNet, una estructura de red neuronal que permite un control detallado de la salida de Stable Diffusion agregando condiciones adicionales.³ Por ejemplo, las imágenes de salida se pueden condicionar en un mapa de borde Canny de una imagen de entrada determinada, como se muestra en la Figura 14-6.

Figura 14-6. Acondicionar la salida de Difusión Estable usando un mapa de borde Canny y ControlNet (fuente: Lvmin Zhang, ControlNet)

ControlNet contiene una copia entrenable del codificador Stable Diffusion, junto con una copia bloqueada del modelo Stable Diffusion completo.

El trabajo de este codificador entrenable es aprender a manejar la condición de entrada (por ejemplo, el mapa de borde de Canny), mientras que la copia bloqueada conserva el poder del modelo original. De esta manera, la difusión estable se puede ajustar utilizando solo una pequeña cantidad de pares de imágenes. Las convoluciones cero son simplemente convoluciones 1×1 donde todos los pesos y sesgos son cero, de modo que antes del entrenamiento, ControlNet no tiene ningún efecto.

Figura 14-7. La arquitectura ControlNet, con las copias entrenables de los bloques codificadores de Difusión Estable resaltados en azul (fuente: Lvmin

Zhang, ControlNet)

Otra ventaja de Stable Diffusion es que puede ejecutarse en una única GPU de tamaño modesto con sólo 8 GB de VRAM, lo que la convierte en posible de ejecutarse en dispositivos perimetrales, en lugar de mediante llamadas a un servicio en la nube. A medida que los servicios de conversión de texto a imagen se incluyen en productos posteriores, la velocidad de generación se vuelve cada vez más importante. Esta es una de las razones por las que el tamaño de los modelos multimodales generalmente tiene una tendencia a la baja (ver Figura 14-3).

En la Figura 14-8 se pueden ver ejemplos de resultados para los tres modelos. Todos estos modelos son excepcionales y pueden capturar el contenido y el estilo de la descripción dada.

Figura 14-8. Salidas de Stable Diffusion v2.1, Midjourney y DALL.E 2 para el mismo mensaje

En la Tabla 14-2 se muestra un resumen de algunos de los modelos de conversión de texto a imagen más potentes que existen en la actualidad.

Modelo	Fecha	Desarrollador	parámetros	Fuente abierta
DALL.E 2	abril de 2022	OpenAI	3,500,000,000	No
Imagen	mayo 2022	Google	4,600,000,000	No
Parti	junio de 2022	Google	20,000,000 ,000	No
Stable Difusión	agosto de 2022	Stability AI, CompVis y Runway	890,000,00	Sí
MUSA	enero de 2023	Google	3,000,000,000	No

Parte de la habilidad de trabajar con modelos de texto a imagen es crear un mensaje que describa el contenido de la imagen que desea generar y utilice palabras clave que alienten al modelo a producir un estilo o tipo de imagen en particular. Por ejemplo, adjetivos como deslumbrante o premiado a

menudo pueden usarse para mejorar la calidad de la generación. Sin embargo, no siempre se da el caso de que el mismo mensaje funcione bien en diferentes modelos; depende del contenido del conjunto de datos de texto e imagen específico utilizado para entrenar el modelo.

El arte de descubrir indicaciones que funcionan bien para un modelo particular se conoce como ingeniería de indicaciones.

Otras aplicaciones

La IA generativa está encontrando rápidamente aplicaciones en una variedad de dominios novedosos, desde el aprendizaje por refuerzo hasta otros tipos de modelos multimodales de texto a X.

Por ejemplo, en noviembre de 2022 Meta publicó un artículo sobre CICERO, un agente de IA entrenado para jugar al juego de mesa Diplomacy. En este juego, los jugadores representan diferentes países de Europa antes de la I Guerra Mundial y deben negociar y engañarse unos a otros para hacerse con el control del continente. Es un juego muy complejo de dominar para un agente de IA, sobre todo porque hay un elemento comunicativo en el que los jugadores deben discutir sus planes con otros jugadores para ganar aliados, coordinar maniobras y sugerir objetivos estratégicos. Para lograr esto, CICERO contiene un modelo de lenguaje que es capaz de iniciar un diálogo y responder a mensajes de otros jugadores. Fundamentalmente, el diálogo es coherente con los planes estratégicos del agente, que son generados por otra parte del modelo para adaptarse a un escenario en constante evolución. Esto incluye la capacidad del agente de farolear cuando conversa con otros jugadores, es decir, convencer a otro jugador de que coopere con los planes del agente, solo para luego realizar una maniobra agresiva contra el jugador en un turno posterior. Sorprendentemente, en una liga de Diplomacia anónima en línea que contó con 40 juegos, la puntuación de CICERO fue más del doble del promedio de los jugadores humanos y se ubicó entre el 10% superior de los participantes que jugaron múltiples juegos. Este es un excelente ejemplo de cómo la IA generativa se puede combinar con éxito con el aprendizaje por refuerzo.

El desarrollo de grandes modelos de lenguaje incorporados es un área de investigación apasionante, ejemplificada aún más por el PaLM-E de Google. Este modelo combina el potente modelo de lenguaje PaLM con un Transformador Visión para convertir datos visuales y sensoriales en tokens que se pueden intercalar con instrucciones de texto, lo que permite a los robots ejecutar tareas basadas en indicaciones de texto y retroalimentación continua de otras modalidades sensoriales. El sitio web PaLM-E muestra las capacidades del modelo, incluido el control de un robot para organizar bloques y buscar objetos basándose en descripciones de texto.

Los modelos de texto a video implican la creación de videos a partir de la entrada de texto.

Este campo, que se basa en el concepto de modelado de texto a imagen, tiene el desafío adicional de incorporar una dimensión temporal. Por ejemplo, en septiembre de 2022, Meta publicó Make-A-Video, un modelo generativo que puede crear un video corto con solo un mensaje de texto como entrada. El modelo también puede agregar movimiento entre dos imágenes estáticas y producir variaciones de un video de entrada determinado.

Curiosamente, se entrena sólo con datos emparejados de texto e imagen y secuencias de vídeo no supervisadas, en lugar de pares de texto y vídeo directamente.

Los datos de vídeo no supervisados son suficientes para que el modelo aprenda cómo el mundo se mueve; luego utiliza los pares texto-imagen para aprender a mapear entre modalidades de imágenes de texto, que luego se animan. El modelo Dreamix puede realizar edición de video, donde un video de entrada se transforma en función de un mensaje de texto determinado conservando otros atributos estilísticos. Por ejemplo, un vídeo de un vaso de leche que se sirve se podría convertir en una taza de café que se sirve, conservando al mismo tiempo el ángulo de la cámara, el fondo y los elementos de iluminación del vídeo original.

De manera similar, los modelos de texto a 3D extienden los enfoques tradicionales de texto a imagen a una tercera dimensión. En septiembre de 2022, Google publicó DreamFusion, un modelo de difusión que genera

activos 3D al recibir un mensaje de texto. Fundamentalmente, el modelo no requiere recursos 3D etiquetados para entrenar. En cambio, los autores utilizan un modelo de texto a imagen 2D (Imagen) previamente entrenado como previo y luego entrenar un Campo de radiación neuronal 3D (NeRF), de modo que es capaz de producir buenas imágenes cuando se renderizan desde ángulos aleatorios. Otro ejemplo es Point-E de OpenAI, publicado en diciembre de 2022. Point-E es un sistema basado puramente en difusión que es capaz de generar una nube de puntos 3D a partir de un mensaje de texto determinado. Si bien el resultado producido no es de tan alta calidad como el de DreamFusion, la ventaja de este enfoque es que es mucho más rápido que los métodos basados en NeRF: puede producir resultados en solo uno o dos minutos en una sola GPU, en lugar de requerir varias horas de GPU.

Dadas las similitudes entre texto y música, no sorprende que también haya habido intentos de crear modelos de texto a música.

MusicLM, lanzado por Google en enero de 2023, es un modelo de lenguaje que puede convertir una descripción de texto de una pieza musical (por ejemplo, “una relajante melodía de violín respaldada por un riff de guitarra distorsionado”) en audio que abarca varios minutos y refleja con precisión la descripción. Se basa en el trabajo anterior AudioLM al agregar la capacidad de que el modelo sea guiado por un mensaje de texto; Los ejemplos que puede escuchar están disponibles en el sitio web de Google Research.

El futuro de la IA generativa

En esta sección final, exploraremos el impacto potencial que los poderosos sistemas de IA generativa pueden tener en el mundo en el que vivimos: en nuestra vida cotidiana, en el lugar de trabajo y en el campo de la educación. También estableceremos los desafíos prácticos y éticos clave que enfrentará la IA generativa si quiere convertirse en una herramienta ubicua que haga una contribución neta positiva significativa a la sociedad.

IA generativa en la vida cotidiana

No hay duda de que en el futuro la IA generativa desempeñará un papel cada vez más importante en la vida cotidiana de las personas, en particular en los grandes modelos lingüísticos. Con ChatGPT de OpenAI, ya es posible generar una carta de presentación perfecta para una solicitud de empleo, una respuesta profesional por correo electrónico a un colega o una publicación divertida en las redes sociales sobre un tema determinado utilizando IA generativa. Esta tecnología es verdaderamente interactiva: puede incluir detalles específicos que usted solicite, responder a los comentarios y formular sus propias preguntas si algo no está claro. Este estilo de IA de asistente personal debería ser cosa de ciencia ficción, pero no lo es: está aquí ahora mismo, para cualquiera que decida utilizarlo.

¿Cuáles son las repercusiones de que este tipo de aplicaciones se generalicen? Es probable que el efecto más inmediato sea un aumento en la calidad de la comunicación escrita. El acceso a grandes modelos de lenguaje con una interfaz fácil de usar permitirá a las personas traducir el esbozo de una idea en párrafos coherentes y de alta calidad en segundos. Esta tecnología transformará la redacción de correos electrónicos, las publicaciones en las redes sociales e incluso la mensajería instantánea de formato corto. Va más allá de eliminar las barreras comunes asociadas con la ortografía, la gramática y la legibilidad: vincula directamente nuestros procesos de pensamiento con resultados utilizables, eliminando a menudo la necesidad de involucrarnos en el proceso de construcción de oraciones.

La producción de textos bien formados es sólo uno de los usos de los grandes modelos lingüísticos. La gente empezará a utilizar estos modelos para generar ideas, dar consejos y recuperar información. Creo que podemos ver esto como la cuarta etapa de nuestra capacidad como especie para adquirir, compartir, recuperar y sintetizar información. Comenzamos adquiriendo información de quienes nos rodeaban o viajando físicamente a nuevos lugares para transferir conocimientos. La invención de la imprenta permitió que el libro se convirtiera en el principal medio a través del cual se compartían ideas. Finalmente, el nacimiento de Internet nos permitió buscar y recuperar información instantáneamente con solo tocar un botón. La IA generativa abre una nueva era de síntesis de información que creo que reemplazará muchos de los usos actuales de los motores de búsqueda actuales.

Por ejemplo, el conjunto de modelos GPT de OpenAI puede proporcionar recomendaciones personalizadas de destinos de vacaciones, como se muestra en la Figura 14-9, o consejos sobre cómo responder a una situación difícil, o una explicación detallada de un concepto oscuro. Usar esta tecnología se siente más como preguntarle a un amigo que escribir una consulta en un motor de búsqueda y, por esa razón, la gente acude a ella extremadamente rápido.

ChatGPT es la plataforma tecnológica de más rápido crecimiento hasta la fecha; adquirió 1 millón de usuarios dentro de los 5 días posteriores a su lanzamiento. Para el contexto, le tomó a Instagram 2.5 meses para llegar al mismo número de usuarios y a Facebook 10 meses.

Figura 14-9. Salida de GPT-3, que ofrece recomendaciones de vacaciones personalizadas

IA generativa en el lugar de trabajo

Además del uso general, la IA generativa encontrará aplicaciones en trabajos específicos donde se requiere creatividad. Una lista no exhaustiva de las ocupaciones que pueden beneficiarse son las siguientes:

Publicidad

La IA generativa se puede utilizar para crear campañas publicitarias personalizadas dirigidas a datos demográficos específicos en función de su historial de navegación y compras.

Producción musical

La IA generativa se puede utilizar para componer y producir pistas musicales originales, lo que permite una gama ilimitada de posibilidades.

Arquitectura

La IA generativa se puede utilizar para diseñar edificios y estructuras, teniendo en cuenta factores como el estilo y las limitaciones de la distribución.

Diseño de moda

La IA generativa se puede utilizar para crear diseños de ropa únicos y diversos, teniendo en cuenta las tendencias y las preferencias del usuario.

Diseño automotriz

La IA generativa se puede utilizar para diseñar y desarrollar nuevos modelos de vehículos y encontrar automáticamente variaciones interesantes sobre un diseño particular.

Producción de cine y vídeo.

La IA generativa se puede utilizar para crear efectos especiales y animaciones, así como para generar diálogos para escenas o historias completas.

Investigación farmacéutica

La IA generativa se puede utilizar para generar nuevos compuestos farmacológicos, que pueden ayudar en el desarrollo de nuevos tratamientos.

Escritura creativa

La IA generativa se puede utilizar para generar contenido escrito, como historias de ficción, poesía, artículos de noticias y más.

Diseño de juegos

La IA generativa se puede utilizar para diseñar y desarrollar nuevos niveles y contenidos de juegos, creando una variedad infinita de experiencias de juego.

Diseño digital

La IA generativa se puede utilizar para crear animaciones y arte digital original, así como para diseñar y desarrollar nuevas interfaces de usuario y diseños web.

A menudo se dice que la IA representa una amenaza existencial para empleos en campos como estos, pero no creo que sea así. Para mí, la IA es simplemente otra herramienta más en la caja de herramientas de estos roles creativos (aunque muy poderosa), en lugar de un reemplazo del rol en sí.

Aquellos que opten por adoptar esta nueva tecnología descubrirán que pueden explorar nuevas ideas mucho más rápido e iterar sobre conceptos de una manera que antes no era posible.

IA generativa en la educación

Una última área de la vida cotidiana que creo que se verá significativamente afectada es la educación. La IA generativa desafía los axiomas fundamentales de la educación de una manera que no hemos visto desde los albores de Internet. Internet brindó a los estudiantes la capacidad de recuperar información de manera instantánea y sin ambigüedades, lo que hizo que los exámenes que simplemente evaluaban la memorización y el recuerdo parecieran anticuados e irrelevantes. Esto impulsó un cambio de enfoque, centrado en evaluar la capacidad de los estudiantes para sintetizar ideas de una manera novedosa en lugar de evaluar únicamente el conocimiento fáctico.

Creo que la IA generativa provocará otro cambio transformador en el campo de la educación, lo que requerirá una reevaluación y un ajuste de los métodos de enseñanza y los criterios de evaluación actuales. Si ahora cada estudiante tiene acceso a una máquina de escribir ensayos en su bolsillo que puede generar respuestas novedosas a las preguntas, ¿cuál es el propósito de los cursos basados en ensayos?

Muchos pedirían que se prohibiera el uso de este tipo de herramientas de inteligencia artificial, del mismo modo que se prohíbe el plagio. Sin embargo, no es tan simple, ya que detectar texto generado por IA es mucho más difícil que detectar plagio y aún más difícil de demostrar más allá de toda duda. Además, los estudiantes podrían utilizar herramientas de inteligencia artificial para generar un borrador básico para el ensayo y luego agregar detalles adicionales o actualizar información objetivamente incorrecta según sea necesario. En este caso, ¿es el trabajo original del estudiante o de la IA?

Claramente, estas son preguntas enormes que deben abordarse para que la educación y las certificaciones mantengan su integridad. En mi opinión, no tiene sentido resistirse a la proliferación de herramientas de inteligencia artificial en la educación; cualquier enfoque de este tipo está condenado al fracaso, ya que se generalizarán tanto en la vida cotidiana que tratar de restringir su uso será inútil. En cambio, debemos encontrar formas de adoptar la tecnología y preguntarnos cómo podemos diseñar cursos con IA abierta, de la misma manera que permitimos cursos con libros abiertos y animamos a los estudiantes a investigar abiertamente material utilizando Internet y herramientas de IA.

El potencial de la IA generativa para ayudar en el proceso de aprendizaje en sí también es inmenso y profundamente profundo. Un tutor con tecnología de inteligencia artificial podría ayudar a un estudiante a aprender un tema nuevo (como se muestra en la Figura 14-10), superar un malentendido o generar un plan de estudio completamente personalizado. El desafío de filtrar la verdad de la ficción generada no es diferente de lo que tenemos actualmente con la información disponible en Internet y es una habilidad para la vida que necesita mayor atención en todo el plan de estudios.

Figura 14-10. Resultado de GPT-3: un ejemplo de cómo se pueden utilizar modelos de lenguaje grandes para el aprendizaje

La IA generativa puede ser una herramienta increíblemente poderosa para nivelar el campo de juego entre quienes tienen acceso a excelentes maestros y los mejores materiales de aprendizaje y quienes no. Estoy emocionado de ver el progreso en este espacio, ya que creo que podría desbloquear enormes cantidades de potencial en todo el mundo.

Ética y desafíos de la IA generativa

A pesar del increíble progreso que se ha logrado en el campo de la IA generativa, aún quedan muchos desafíos por superar. Algunos de estos desafíos son prácticos y otros éticos.

Por ejemplo, una crítica importante a los modelos de lenguaje grandes es que son propensos a generar información errónea cuando se les pregunta sobre un tema desconocido o contradictorio, como se muestra en la Figura 14-4.

El peligro de esto es que es difícil saber si la información contenida en una respuesta generada es realmente precisa.

Incluso si le pide al LLM que explique su razonamiento o cite fuentes, podría inventar referencias o emitir una serie de declaraciones que no se suceden lógicamente unas a otras. Este no es un problema fácil de resolver, ya que el LLM no es más que un conjunto de pesos que capturan con precisión la siguiente palabra más probable dado un conjunto de tokens de entrada; no tiene un banco de información verdadera que pueda usar como referencia.

Una posible solución a este problema es proporcionar modelos de lenguaje grandes con la capacidad de recurrir a herramientas estructuradas como calculadoras, compiladores de código y fuentes de información en línea para tareas que requieren una ejecución o hechos precisos. Por ejemplo, la Figura 14-11 muestra el resultado de un modelo llamado Toolformer, publicado por Meta en febrero de 2023. [4]

Figura 14-11. Un ejemplo de cómo Toolformer puede llamar de forma autónoma a diferentes API para obtener información precisa cuando sea necesario (fuente: Schick et al., 2023)

Toolformer puede llamar explícitamente a las API para obtener información, como parte de su respuesta generativa. Por ejemplo, podría utilizar la API de Wikipedia para recuperar información sobre una persona en particular, en lugar de depender de que esta información esté integrada en los pesos de su modelo.

Este enfoque es particularmente útil para operaciones matemáticas precisas, donde Toolformer puede indicar qué operaciones le gustaría ingresar en la API de la calculadora en lugar de intentar generar la respuesta de forma autorregresiva de la manera útil.

Otra preocupación ética destacada con la IA generativa se centra en el hecho de que las grandes empresas han utilizado enormes cantidades de datos extraídos de la web para entrenar sus modelos, cuando los creadores originales no dieron su consentimiento explícito para hacerlo. A menudo, estos datos ni siquiera se publican, por lo que es imposible saber si sus datos se utilizan para entrenar modelos de lenguaje grandes o modelos multimodales de texto a imagen. Claramente esta es una preocupación válida, particularmente para artistas, quienes pueden argumentar que es el uso de su obra de arte por el cual no se les paga ninguna regalía o comisión. Además, el nombre de un artista puede usarse como estímulo para generar más obras de arte que sean similares en estilo a los originales, degradando así la singularidad del contenido y mercantilizando el estilo.

Stability AI está siendo pionera en una solución a este problema, cuyo modelo multimodal Stable Diffusion se entrena en un subconjunto del conjunto de datos de código abierto LAION-5B. También han puesto en marcha la web ¿Me han formado? donde cualquiera puede buscar una imagen o un pasaje de texto en particular dentro del conjunto de datos de entrenamiento y optar por no incluirse en el futuro en el proceso de entrenamiento del modelo. Esto devuelve el control a los creadores originales y garantiza que haya transparencia en los datos que se utilizan para crear herramientas poderosas como esta. Sin embargo, esta práctica no es común y muchos modelos de IA generativa disponibles comercialmente

no hacen que sus conjuntos de datos o ponderaciones de modelos sean de código abierto ni brindan ninguna opción para optar por no participar en el proceso de capacitación.

En conclusión, si bien la IA generativa es una herramienta poderosa para la comunicación, la productividad y el aprendizaje en la vida cotidiana, en el lugar de trabajo y en el campo de la educación, su uso generalizado tiene ventajas y desventajas. Es importante ser consciente de los riesgos potenciales de utilizar los resultados de un modelo de IA generativa y asegurarse siempre de utilizarlos de forma responsable.

Sin embargo, sigo siendo optimista sobre el futuro de la IA generativa y estoy ansioso por ver cómo las empresas y las personas se adaptan a esta nueva y apasionante tecnología.

Pensamientos finales

En este libro hemos realizado un viaje a través de la última década de investigación en modelos generativos, comenzando con las ideas básicas detrás de VAE, GAN, modelos autorregresivos, modelos de flujo normalizador, modelos basados en energía y modelos de difusión, y construyendo sobre estos fundamentos para comprender cómo las técnicas de vanguardia como VQ-GAN, Transformers, modelos mundiales y modelos multimodales ahora están superando los límites de lo que los modelos generativos son capaces de lograr, en una variedad de tareas.

Creo que en el futuro, el modelado generativo puede ser la clave para una forma más profunda de inteligencia artificial que trascienda cualquier tarea particular y permita a las máquinas formular orgánicamente sus propias recompensas, estrategias y tal vez conciencia dentro de su entorno. Mis creencias están estrechamente alineadas con el principio de inferencia activa, originalmente iniciado por Karl Friston. La teoría detrás de la inferencia activa fácilmente podría llenar otro libro entero, y lo hace, en el excelente *Inferencia activa: el principio de la energía libre en mente, cerebro y comportamiento*, de Thomas Parr et al. (Prensa del MIT) que recomiendo ampliamente, así que sólo daré una breve explicación aquí.

Cuando somos bebés, exploramos constantemente nuestro entorno, construyendo un modelo mental de futuros posibles sin otro objetivo aparente que el de desarrollar una comprensión más profunda del mundo. No hay etiquetas en los datos que recibimos: un flujo aparentemente aleatorio de ondas de luz y sonido que bombardean nuestros sentidos desde el momento en que nacemos. Incluso cuando alguien señala una manzana y dice manzana, no hay razón para que nuestros cerebros jóvenes asocien las dos entradas y aprendan que la forma en que la luz entró en nuestro ojo en ese momento en particular está relacionada de alguna manera con la forma en que se escuchó el sonido. Las ondas entraron en nuestro oído. No existe un conjunto de entrenamiento de sonidos e imágenes, ni un conjunto de entrenamiento de olores y sabores, ni un conjunto de entrenamiento de

acciones y recompensas; simplemente hay un flujo interminable de datos extremadamente ruidosos.

Y, sin embargo, aquí estás, leyendo esta frase, tal vez disfrutando del sabor de una taza de café en una cafetería ruidosa. No prestas atención al ruido de fondo mientras te concentras en convertir la ausencia de luz en una pequeña porción de tu retina en una secuencia de conceptos abstractos que casi no transmiten ningún significado individualmente pero que, cuando se combinan, desencadenan una ola de representaciones paralelas en tu mente. ojo: imágenes, emociones, ideas, creencias y acciones potenciales inundan tu conciencia, esperando tu reconocimiento. El mismo ruidoso flujo de datos que esencialmente no tenía sentido para el cerebro infantil ya no lo es tanto. Todo tiene sentido para ti. Ves estructura en todas partes. Nunca te sorprende la física de la vida cotidiana. El mundo es como es porque tu cerebro decidió que debería ser así. En este sentido, su cerebro es un modelo generativo extremadamente sofisticado, equipado con la capacidad de prestar atención a partes particulares de los datos de entrada, formar representaciones de conceptos dentro de un espacio latente de vías neuronales y procesar datos secuenciales a lo largo del tiempo.

La inferencia activa es un marco que se basa en esta idea para explicar cómo el cerebro procesa e integra información sensorial para tomar decisiones y acciones. Afirma que un organismo tiene un modelo generativo del mundo que habita y utiliza este modelo para hacer predicciones sobre eventos futuros. Para reducir la sorpresa provocada por las discrepancias entre el modelo y la realidad, el organismo ajusta sus acciones y creencias en consecuencia. La idea clave de Friston es que la optimización de la acción y la percepción se pueden enmarcar como dos caras de la misma moneda, y ambas buscan minimizar una única cantidad conocida como energía libre.

En el centro de este marco se encuentra un modelo generativo del entorno (capturado dentro del cerebro) que se compara constantemente con la realidad. Fundamentalmente, el cerebro no es un observador pasivo de los acontecimientos. En los humanos, está unido a un cuello y a un par de patas que pueden colocar sus sensores de entrada centrales en innumerables posiciones en relación con la fuente de datos de entrada. Por lo tanto, la

secuencia generada de futuros posibles no sólo depende de su comprensión de la física del medio ambiente, sino también de su comprensión de sí mismo y de cómo actúa. Este circuito de retroalimentación de acción y percepción es extremadamente interesante para mí, y creo que sólo hemos Arañado la superficie de lo que es posible con modelos generativos encarnados que son capaces de tomar acciones dentro de un entorno determinado de acuerdo con los principios de la inferencia activa.

Esta es la idea central que creo que seguirá impulsando el modelado generativo hacia el centro de atención en la próxima década, como una de las claves para desbloquear la inteligencia artificial general.

Con eso en mente, lo animo a que continúe aprendiendo más sobre los modelos generativos a partir de todo el excelente material disponible en línea y en otros libros. Gracias por tomarse el tiempo de leer hasta el final de este libro. ¡Espero que haya disfrutado leyéndolo tanto como he disfrutado generándolo!

1 Hugo Touvron et al., “LLaMA: Modelos de lenguaje básico abiertos y eficientes”, 27 de febrero de 2023, <https://arxiv.org/abs/2302.13971>.

2 Mark Chen et al., “Evaluación de modelos de lenguaje grandes entrenados en código”, 7 de julio de 2021, <https://arxiv.org/abs/2302.13971>.

3 Lvmin Zhang y Maneesh Agrawala, “Aregar control condicional a los modelos de difusión para la conversión de texto a imagen”, 10 de febrero de 2023, <https://arxiv.org/abs/2302.05543>.

4 Timo Schick et al., “Toolformer: Los modelos de lenguaje pueden enseñarse a usar herramientas”, 9 de febrero de 2023, <https://arxiv.org/abs/2302.04761>.

Índice

Símbolos

1-Función continua de Lipschitz, La restricción de Lipschitz de precisión, determinación, El Discriminador, ChatGPT, Grandes Modelos de lenguaje, acción de desafíos y ética de la IA generativa, en el aprendizaje por refuerzo, funciones de activación del aprendizaje por refuerzo, inferencia activa de capas, reflexiones finales

Optimizador Adam (Estimación de momento adaptativo), Normalización de instancia adaptativa de optimizadores (AdaIN), Normalización de instancia adaptativa, Agente de modulación y demodulación de peso, en aprendizaje por refuerzo, Aprendizaje por refuerzo

IA (inteligencia artificial), Modelado Generativo e IA,

Ética de la IA, Ética de la IA generativa y Desafíos generativos

Modelos de densidad aproximada de ética y desafíos de la IA, artefactos de taxonomía de modelos generativos, el generador, inteligencia artificial (IA) StyleGAN2, modelado generativo e IA,

Redes neuronales artificiales (RNA), Deep Neural Networks arXiv, Otros recursos “La atención es todo lo que necesitas” (Vaswani), Introducción, 2014–

2017: Los mecanismos de atención de la era VAE y GAN, ecuación de atención, consultas, claves y valores, cabeza de atención, puntuaciones de atención, visualización de las puntuaciones de atención, ponderaciones de atención, consultas, claves y valores que generan música polifónica, tokenización de

Popularización del artículo sobre música polifónica, Introducción auto-referencial frente a referencia cruzada, Comprensión T5, Enlace de atributos de atención, Atributos de limitaciones, enredados, StyleGAN

AudioLM, otras aplicaciones “Bayes variacionales de codificación automática” (Kingma y Welling),

Autocodificadores variacionales codificadores automáticos (ver también codificadores automáticos variacionales) arquitectura de, arquitectura del decodificador de la arquitectura Autoencoder, diagrama de proceso del Decodificador-El Decodificador, Autocodificadores

arquitectura del codificador, The Encoder

Conjunto de datos Fashion-MNIST, El conjunto de datos Fashion-MNIST generando nuevas imágenes, Generando nuevas imágenesGenerando nuevas imágenes uniendo el codificador al decodificador, Uniendo el codificador al

Decodificador que reconstruye imágenes, Usos de reconstrucción de imágenes para, La arquitectura del Autoencoder que visualiza el espacio latente, Visualización de los modelos autorregresivos del espacio latente, autorregresivo anterior de DALL.E 2, Autoregresivo anterior,

Importancia de las celdas bidireccionales anteriores, descripción de las celdas bidireccionales, modelos autorregresivos, unidades recurrentes cerradas (GRU), taxonomía del modelo generativo de unidades recurrentes cerradas, modelo generativo

Historia de la taxonomía, 2014-2017: La era VAE y GAN, cómo funcionan los LSTM, Introducción a las redes de memoria a corto plazo (LSTM), Red de memoria a corto plazo (LSTM): análisis de las capas convolucionales enmascaradas de LSTM, Convolucional enmascarado

Capas convolucionales enmascaradas por capas

redes recurrentes apiladas, recurrentes apiladas

Redes recurrentes apiladas en redes

Conjunto de datos corales de Bach, Retropropagación del conjunto de datos corales de Bach, ¿Qué es una red neuronal? normalización por lotes,

Normalización por lotes: predicción mediante normalización por lotes, Abandono, Entrenamiento de los lotes WGAN-GP, Entrenamiento del modelo

BERT (representaciones de codificador bidireccional de transformadores), Otros Transformadores, 2018-2019: las células bidireccionales de la era de los transformadores, era del gran modelo de células bidireccionales, 2020-2022: la era del gran modelo BigGAN, BigGAN, 2018-2019: pérdida de entropía cruzada binaria de la era Transformer, funciones de pérdida

BLOOM, 2020-2022: La era de los grandes modelos, el lenguaje amplio

Modelos

Distribución Boltzmann, Modelos Basados en Energía

Máquina Boltzmann, Otros modelos basados en energía

LibroCorpus, GPT

Conjunto de datos de ladrillos, The Bricks Dataset

Mapas de borde Canny, modelos de texto a imagen

pérdida categórica de entropía cruzada, funciones de pérdida de enmascaramiento causal, enmascaramiento causal-enmascaramiento causal

Conjunto de datos de atributos de CelebFaces (CelebA), The CelebA

Conjunto de datos, entrenamiento progresivo

Desafíos de CGAN (ver GAN condicional), de IA generativa, ChatGPT, lenguaje grande

Modelos, ética y desafíos de la IA generativa-IA generativa

Ética y Retos ecuación del cambio de variables, El Cambio de Variables

Técnica de cambio de ecuación de variables, Cambio de Variables-Cambio de variables tokens de caracteres, Tokenización

ChatGPT, ChatGPT-ChatGPT, 2020-2022: el gran modelo

Era, IA generativa en la vida cotidiana

Chinchilla, 2020-2022: la era de los grandes modelos

CICERON, Otras Aplicaciones

Conjunto de datos CIFAR-10, Preparación de los datos

Descripción de CLIP (preentrenamiento de imagen y lenguaje contrastante), historia de CLIP, 2020-2022: conceptos clave detrás de la era del gran modelo, proceso de capacitación de CLIP, CLIP-CLIP

CMA-ES (estrategia de evolución de adaptación de matriz de covarianza),

Entrenamiento del controlador-paralelización CMA-ES

Ejemplos de código CNN (ver redes neuronales convolucionales), obtención y uso, base de código, clonación del

Libro de códigos del repositorio, VQ-GAN

Codex, comentarios y preguntas sobre modelos de texto a código, método de compilación Cómo contactarnos, análisis de GAN condicional (CGAN) de optimizadores, análisis de la arquitectura CGAN, capacitación en arquitectura CGAN, capacitación de las “redes adversas generativas condicionales” de CGAN (Mirza y

Osindero), vector de contexto GAN condicional (CGAN), consultas, claves y valores de divergencia contrastiva, modelos basados en energía, entrenamiento con divergencia contrastiva-entrenamiento con contraste

Divergencia

Preentrenamiento de lenguaje contrastivo-imagen (ver CLIP) Aprendizaje contrastivo, CLIP

ControlNet, modelos de texto a imagen, redes neuronales convolucionales (CNN)

normalización por lotes, normalización por lotes: predicción utilizando los beneficios de la normalización por lotes, construcción de redes neuronales convolucionales (CNN), construcción de CNN: construcción de capas convolucionales de CNN, capas convolucionales: inspección de la deserción del modelo, deserción-deserción de capas convolucionales enmascaradas, convolucional enmascarado

Capacitación y evaluación de capas convolucionales enmascaradas, Capacitación y evaluación de

Capas de transposición convolucional de CNN, The Decoder

Copiloto, Programa de difusión de coseno de modelos de texto a código, Programación de difusión de similitud de coseno, Capas de acoplamiento CLIP, Capas de acoplamiento-Capas de acoplamiento, Apilamiento de capas de acoplamiento desplazamiento de covariable, Atención referencial cruzada de desplazamiento de covariables, T5

CycleGAN, VQ-GAN, 2014-2017: la era VAE y GAN

DALL.E, VQ-GAN, DALL.E 2, Limitaciones, 2020-2022: El

Gran era del modelo

DALL.E 2

arquitectura, Disponibilidad de arquitectura de, decodificador de modelos de texto a imagen, Ejemplos de Decoder-Upsampler generados por, Ejemplos de DALL.E 2Importancia del historial anterior de modelos de texto a imagen de, DALL.E 2, 2020–2022: Las limitaciones de la era del gran modelo, las limitaciones del codificador de texto, el codificador de texto-CLIP que entrena el modelo anterior, la difusión previa anterior

DCGAN (ver GAN convolucional profunda)

DDIM (ver Modelo implícito de difusión de eliminación de ruido)

DDM (ver modelos de difusión con eliminación de ruido)

Decodificador DDPM (ver Modelo probabilístico de difusión de eliminación de ruido) Transformadores, enmascaramiento causal, otros

Análisis de GAN convolucional profundo (DCGAN) de Transformers, Análisis del conjunto de datos DCGAN utilizado, Discriminador del conjunto de datos Bricks en, Generador Discriminator-The Discriminator en, Historia del Generator-The Generator, 2014-2017: artículo publicado en la era VAE y GAN en, Entrenamiento de GAN convolucional profundo (DCGAN), Entrenamiento del DCGAN-Análisis del DCGAN

consejos y trucos de capacitación, Capacitación GAN: consejos y trucosAbordar los desafíos de GAN aprendizaje profundo redes neuronales profundas, Aprendizaje de redes neuronales profundas

Funciones de alto nivel, resumen definido, aprendizaje profundo

Keras y TensorFlow para, creación de modelos TensorFlow y Keras, TensorFlow y Keras: evaluación de

Mejora del modelo, Red neuronal convolucional (CNN): entrenamiento y evaluación de datos estructurados versus no estructurados de CNN, Datos para profundidad

Paso de demodulación de aprendizaje, modulación de peso y demodulación.

Descripción del modelo implícito de difusión de eliminación de ruido (DDIM), muestreo a partir de la difusión de eliminación de ruido

Historia del modelo de, 2020-2022: análisis de los modelos de difusión de eliminación de ruido (DDM) de la era de los grandes modelos, análisis del modelo de difusión, interpolación entre el conjunto de datos de imágenes utilizado, descripción del conjunto de datos de las flores: descripción del

conjunto de datos de las flores, programas de difusión de los modelos de difusión de eliminación de ruido (DDM), Horarios de difusión

proceso de difusión directa, The Forward Diffusion

Truco de reparametrización de procesos, El truco de reparametrización
Proceso de difusión inversa, La difusión inversa

Proceso: muestreo del proceso de difusión inversa, muestreo de la difusión de eliminación de ruido

Muestreo de modelos a partir del entrenamiento del modelo de difusión de eliminación de ruido, Entrenamiento del modelo de difusión: entrenamiento del

Modelo de difusión

Modelo de eliminación de ruido de U-Net, modelo de eliminación de ruido de U-NetDownBlocks y UpBlocks, GLIDE

Desarrollo del modelo probabilístico de difusión de eliminación de ruido (DDPM), Historia de otros modelos basados en energía, 2020-2022: modelos de eliminación de ruido de la era de los grandes modelos, La arquitectura del auto codificador (ver también modelos de difusión de eliminación de ruido), capas densas, ¿Qué es una red neuronal? función de densidad, teoría central de la probabilidad, generativa

Determinantes de la taxonomía modelo, el determinante jacobiano

Dhariwal, Prafulla, 2020-2022: Descripción de los modelos de difusión de la Big Model Era (ver también modelos de difusión con eliminación de ruido), difusión previa de los modelos de DALL.E 2, difusión previa

taxonomía del modelo generativo, Modelo Generativo

Historia de la taxonomía, 2020-2022: Las ideas clave que sustentan la era de los grandes modelos, Introducción-Introducción, difusión latente, 2020-2022: La era de los grandes modelos “Los modelos de difusión superan a

las GAN en la síntesis de imágenes” (Dhariwal y Nichol), 2020-2022: La Horarios de difusión de Big Model Era, Horarios de difusión

Dinh, Laurent, RealNVP

Juego de mesa de diplomacia, Otras aplicaciones, espacio latente discreto, Modelado discriminativo VQ-GAN, Generativo versus discriminativo

Modelando discriminadores, Introducción, El Discriminador-El

Discriminador, Discriminador domina al generador.

Dong, Hae-Wen, MuseGAN

Banco de dibujo, Banco de dibujo

DreamFusion, otras aplicaciones

Dreamix, Capas de abandono de otras aplicaciones, Abandono

Du, Yilun, modelos basados en energía

aplicaciones educativas, IA generativa en la educación

EMA (media móvil exponencial), la difusión inversa

Espacio de incrustación de procesos, la arquitectura de Autoencoder,

Visualización del espacio latente, exploración del espacio latente
Transformación entre caras, incrustación sinusoidal de modelos de lenguaje
grandes incorporados, otras aplicaciones codificador Transformadores,
enmascaramiento causal, otros

Codificador-decodificador de transformadores Transformadores, otros
codificadores de transformadores, función de energía/puntuación de energía
del codificador, modelos basados en energía,

Análisis de modelos basados en energía (EBM) de función energética,
Análisis del modelo basado en energíaAnálisis del modelo basado en

energía

Distribución Boltzmann, Modelos Basados en Energía

Máquina de Boltzmann, Otros conjuntos de datos de modelos basados en energía utilizados, Descripción del conjunto de datos MNIST de, Función de energía de los modelos basados en energía, Taxonomía del modelo generativo de función de energía, Modelo generativo

Conceptos clave de taxonomía detrás, Introducción

RBM (máquina Boltzmann restringida), otros modelos basados en energía

muestreo usando dinámica de Langevin, muestreo usando

Dinámica de Langevin: muestreo con Langevin

Modelos generativos dinámicos basados en puntuaciones, Entrenamiento de modelos de difusión con divergencia contrastiva, Entrenamiento con

Entrenamiento de divergencia contrastiva con contrastiva

Atributos entrelazados de divergencia, entorno StyleGAN, en aprendizaje por refuerzo, refuerzo

Episodio de aprendizaje, en aprendizaje por refuerzo, Refuerzo

Épocas de aprendizaje, Entrenamiento del modelo, tasas de aprendizaje igualadas, Tasas de aprendizaje igualadas, preocupaciones éticas, de la IA generativa, Ética de la IA generativa y

Método de evaluación de desafíos y ética de la IA generativa, evaluación del modelo de tokenización basada en eventos, estrategias evolutivas de tokenización basada en eventos, modelos de densidad explícitos de la arquitectura del controlador, problema de gradiente explosivo de la taxonomía del modelo generativo, promedio móvil exponencial de normalización por lotes (EMA), lo inverso Difusión

Proceso

conjunto de datos de generación de imágenes faciales utilizado, el conjunto de datos CelebA que genera nuevas caras, generación de nuevas caras aritmética del espacio latente, transformación aritmética del espacio latente entre caras, progreso de la transformación entre caras, el auge del modelado generativo

Análisis VAE, Análisis del Autocodificador Variacional

Entrenamiento VAE, Entrenamiento del Autocodificador Variacional

Conjunto de datos Fashion-MNIST, El conjunto de datos Fashion-MNIST,

Visualización de la ruta rápida del espacio latente, ingeniería de funciones de bloques residuales, descripción de funciones de aprendizaje de funciones de alto nivel, ¿Qué es el modelado generativo? aprendizaje de alto nivel, aprendizaje de características de alto nivel

FFJORD (Dinámica continua de forma libre para escalable

Modelos generativos reversibles), filtros FFJORD, método de ajuste de capas convolucionales, entrenamiento del modelo

Arquitectura Flamingo, Arquitectura-El modelo de lenguaje ejemplos generados por, Ejemplos de FlamingoEjemplos de Flamingo

Historia de, 2020-2022: La era de los grandes modelos

Flan-T5, 2020-2022: la era de los grandes modelos

Conjunto de datos de Flowers, proceso de difusión hacia adelante del conjunto de datos de Flowers, paso hacia adelante del proceso de difusión hacia adelante, ¿Qué es una red neuronal?

Dinámica continua de forma libre para reversible escalable

Modelos generativos (FFJORD), capas totalmente conectadas de FFJORD, ¿Qué es una red neuronal? API funcional (Keras), construcción del modelo-inspección del modelo

estado del juego, en aprendizaje por refuerzo, Refuerzo

Aprendiendo

Unidades recurrentes cerradas (GRU) GAN (ver redes generativas adversarias), memoria a corto plazo y larga

Network (LSTM), unidades recurrentes cerradas, 2014-2017: el

Era VAE y GAN

Distribución gaussiana (distribución normal), The Encoder

GenAI (ver IA generativa) “Redes antagónicas generativas” (Goodfellow), Generativa

Redes adversarias Redes generativas adversarias (GAN)

BigGAN, BigGAN

Desafíos de la capacitación GAN: consejos y trucos, abordaje

GAN desafía la GAN condicional (CGAN), GAN condicional (CGAN) Análisis de las GAN convolucionales profundas (DCGAN) de CGAN, Deep

GAN convolucional (DCGAN): abordar los desafíos de GAN conceptos de entrenamiento fundamentales, Introducción a la taxonomía del modelo generativo, Modelo generativo

Historia de la taxonomía de redes generativas adversarias, 2014–

2017: La era VAE y GAN

ProGAN, salidas ProGAN

StyleGAN, StyleGAN-Salidas de StyleGAN

StyleGAN2, StyleGAN2-Salidas de StyleGAN2

Wasserstein GAN con penalización de gradiente (WGAN-GP),

Wasserstein GAN con penalización de gradiente (WGAN-GP) Análisis de WGAN-GP versus WGAN-GP, Entrenamiento del estado actual de la IA generativa (GenAI) de WGAN-GP, El estado actual de la IA generativa, Otras aplicaciones, ética y desafíos relacionados con ChatGPT, Generativo

Ética y desafíos de la IA en el futuro, El futuro de la IA generativa: la IA generativa en

Educación, reflexiones finales

Historia de, Cronología de la IA generativa-2020-2022: El

Recursos adicionales de aprendizaje profundo generativo de Big Model Era (ver también modelos), otros recursos, teoría de probabilidad central, teoría de probabilidad central-núcleo

Teoría de la probabilidad creando algo que es creativo, Prefacio marco de modelado generativo, El generativo

Introducción al marco de modelado, ¿Qué es el modelado generativo? Objetivos y enfoque de aprendizaje del modelado generativo e IA, objetivo y

Enfoque de requisitos previos para el aprendizaje, Requisitos previos Modelado generativo (ver modelos) Transformador generativo preentrenado (ver GPT) Generadores basados en la atención, Generador de barras Self-Attention GAN (SAGAN), El generador de barras

Generador DCGAN, El Generador-El Generador,

El discriminador domina al generador en GAN, Introducción, El generador-El generador

Generador MuseGAN, El generador MuseGAN

Generador StyleGAN, StyleGAN

Copiloto de GitHub, modelos de texto a código
GLIDE (Difusión guiada de lenguaje a imagen para
Generación y edición), The Decoder-Upsampler, 2020–
2022: La era de los grandes modelos

Modelo GLOW, GLOW

Goodfellow, Ian, Redes generativas adversarias

Gopher, 2020-2022: la era de los grandes modelos

Análisis de GPT (transformador generativo preentrenado), análisis de aplicaciones de visualización de puntuaciones de atención de GPT en la vida cotidiana, IA generativa en todos los días

Mecanismo de atención a la vida, enmascaramiento causal de atención, conjunto de datos de enmascaramiento causal-enmascaramiento causal utilizado, descripción del conjunto de datos de The Wine Reviews, evolución de GPT, historia de GPT-3 y GPT-4, GPT, 2018-2019: mejoras de la era Transformer a GPT atención multicabezal, codificación posicional de atención multicabezal, codificación posicional-posicional

Codificación de consultas, claves y valores, Consultas, claves y valores
Consultas, claves y valores

Bloque transformador, El bloque transformador-El

Bloque transformador

GPT-2, GPT, 2018-2019: La era de los transformadores

Disponibilidad de GPT-3, beneficios de modelos de lenguaje grandes, evolución de GPT de, ejemplo de GPT-3 y GPT-4 generado por, GPT-3 y GPT-4, grande

Modelos de lenguaje, IA generativa en la historia de la educación, 2020-2022: la era de los grandes modelos

GPT-3.5, GPT, ChatGPT

Descenso de gradiente GPT-4, GPT, GPT-3 y GPT-4 usando dinámica de Langevin, muestreo usando

Pérdida de penalización por gradiente de Langevin Dynamics, Pérdida por penalización por gradiente

Cinta de gradiente, entrenamiento de tokenización de cuadrícula del auto codificador variacional, tokenización de cuadrícula-tokenización de cuadrícula

GRU (ver unidades recurrentes cerradas)

Difusión guiada de lenguaje a imagen para generación y

Edición (GLIDE), El Decodificador-Upsampler

Ja, David, Introducción, MDN-RNN, 2014–2017: El

Era VAE y GAN

alucinaciones, ChatGPT

Inicialización, capas ocultas de tasas de aprendizaje igualadas, estado oculto de funciones de alto nivel de aprendizaje, capa LSTM, celda LSTM

Hinton, Geoffrey, Abandono escolar

Hochreiter, Sepp, Red de memoria a corto plazo (LSTM)

Huang, Cheng-Zhi Anna, Tokenización de la música polifónica

Hui, Jonathan, Los hiperparámetros de restricción de Lipschitz, Hiperparámetros

generación de imágenes (ver también generación de imágenes faciales; PixelCNN) beneficios de los modelos de difusión para, Modelos de difusión BigGAN, BigGAN

Conjunto de datos CIFAR-10 para preparar los datos

Análisis DDM, Generación de imágenes-Interpolación entre imágenes generando nuevas imágenes, Generación de nuevas imágenesProceso de modelado generativo de generación de nuevas imágenes, ¿Qué es generativo?

¿Modelado? modelado generativo versus discriminativo, generativo versus modelos discriminativos

Historia de, 2020-2022: La era de los grandes modelos

ProGAN, ProGAN-Outputs avanza en la generación de imágenes faciales, The Rise of

Modelado generativo, reconstrucción de imágenes, Reconstrucción de imágenes, aprendizaje de representación para, Aprendizaje de representación

Autoatención GAN (SAGAN), Autoatención GAN (SAGAN)

StyleGAN2, StyleGAN2-Salidas de StyleGAN2 que visualizan el espacio latente, Visualización de los modelos de imagen a imagen del espacio latente, VQ-GAN, 2014-2017: VAE y

Era GAN

Imagen arquitectura, Arquitectura

DrawBench, ejemplos de DrawBench generados por, Ejemplos de la historia de Imagen de, Imagen, 2020-2022: descripción general de la era del gran modelo de los modelos de texto a imagen

Imágenes del conjunto de datos de LEGO Bricks, modelos de densidad implícita de The Bricks Dataset, Taxonomía de modelos generativos “Generación implícita y modelado con tecnología basada en energía

Models” (Du y Mordatch), Modelos basados en energía “Improving Language Understanding by Generative PreTraining” (Radford), GPT

entrenamiento en sueños, entrenamiento en sueños-entrenamiento en sueños

Modelo InstructGPT, distribuciones normales multivariadas isotrópicas ChatGPT, The Encoder

Determinante jacobiano, codificación conjunta de token/posición del determinante jacobiano, codificación posicional

Kaggle, el conjunto de datos de CelebA, el conjunto de datos de reseñas de vinos

Creación de auto codificador de Keras (ver también modelos), beneficios de The Encoder, TensorFlow y Keras

Capa Conv2DTranspose, The Decoder creando nuevas capas en, Construyendo la carga de datos del codificador VAE, Preparando la creación del conjunto de datos de datos, Creación del decodificador The Bricks Dataset en, Documentación de The Decoder, Optimizadores

Creación del discriminador GAN en, Construcción del modelo Discriminador, Construcción del modelo: inspección del modelo, compilación del modelo, Compilación del modelo.

evaluación del modelo, Evaluación del modelo-Evaluación del

Mejora del modelo, Red neuronal convolucional (CNN): entrenamiento y evaluación del entrenamiento del modelo CNN, Entrenamiento del modelo

Generador MuseGAN, Reuniendo todos los recursos, La base de código de aprendizaje profundo generativo

Tutorial de StyleGAN, StyleGAN

Creación de VAE en, Construyendo el codificador VAE
capas de keras

Activación, construcción de la CNN

Normalización por lotes, Normalización por lotes

Células bidireccionales, bidireccionales.

Conv2D, capas convolucionales

Conv2DTranspose, el decodificador, el generador

Conv3D, el crítico de MuseGAN

Denso, capas

Abandono, abandono

Incrustación, la capa de incrustación

Aplanar, Capas

GRU, Red de memoria a corto plazo (LSTM)

Entrada, Capas

LeakyReLU, construyendo la CNN

LSTM, la capa LSTM

Atención MultiHead, Atención Multihead

UpSampling2D, el generador

Módulo Keras NLP, núcleos de codificación de posición sinusoidal,
vectores clave de capas convolucionales, consultas, claves y valores

Kingma, Diederik, codificadores automáticos variacionales

Divergencia Kullback-Leibler (KL), la función de pérdida suavizado de etiquetas, Entrenamiento del DCGAN

Conjunto de datos LAION-5B, modelos de texto a imagen

LaMDA, 2020-2022: la era de los grandes modelos

Dinámica de Langevin, modelos basados en energía, uso de muestreo

Langevin Dynamics-Sampling utilizando el modelado de lenguaje Langevin Dynamics, GPT

Modelo de lenguaje grande Meta AI (LLaMA), lenguaje grande

Modela modelos de lenguajes grandes (LLM), modelos de lenguajes grandes Modelos de lenguajes grandes

Conjunto de datos de comprensión de escenas a gran escala (LSUN), difusión latente de resultados, arquitectura, 2020-2022: el gran modelo

Era

espacio latente, La arquitectura Autoencoder, Visualizando el

Espacio latente, explorando la transformación del espacio latente

Normalización de la capa entre caras, Las capas del bloque transformador, ¿Qué es una red neuronal?, Construcción del codificador VAE (consulte también Capas Keras), regularización diferida, Regularización de la longitud de la ruta.

LeakyReLU, tasa de aprendizaje de funciones de activación, probabilidad de optimizadores, programa de difusión lineal de la teoría de la probabilidad central, programas de difusión

Restricción de Lipschitz, La restricción de Lipschitz, Aplicación de la restricción de Lipschitz

LLaMA (Meta AI del modelo de lenguaje grande), lenguaje grande

Modelos

LLM (modelos de lenguaje grande), modelos de lenguaje grande, logaritmo de la varianza de modelos de lenguaje grande, funciones de pérdida de redes de memoria a corto plazo del codificador (consulte redes LSTM), funciones de pérdida, unión del codificador al

Decodificador, función de pérdida, muestreo con Langevin

Matriz de triángulo inferior dinámica, pasando datos a través de una capa de acoplamiento

Redes LSTM (memoria a largo plazo)

Capa de incrustación, La capa de incrustación genera conjuntos de datos, Creación del conjunto de entrenamiento que genera texto nuevo, Análisis del historial de LSTM, Red de memoria a corto plazo (LSTM),

2014-2017: la era VAE y GAN

Arquitectura LSTM, La arquitectura LSTM

Célula LSTM, La célula LSTM: entrenamiento del LSTM

Capa LSTM, The LSTM Layer: artículo publicado sobre la capa LSTM, Red de memoria a corto plazo (LSTM) que tokeniza el texto, Tokenización

Conjunto de datos LSUN (comprensión de escenas a gran escala), resultados

beneficios del aprendizaje automático, datos de aprendizaje de representación para, datos para aprendizaje profundo-datos para profundo

Principio de abandono del aprendizaje, modelado generativo del abandono y qué es generativo

¿Modelado?: El auge de las bibliotecas de modelado generativo para las ramas principales de TensorFlow y Keras de aprendizaje profundo y multicapa.

Perceptrón (MLP), aprendizaje por refuerzo

recursos, otros recursos

Make-A-Video, Otras aplicaciones red de mapeo f, The Mapping Network capas convolucionales enmascaradas, capas convolucionales enmascaradas Capas convolucionales enmascaradas enmascaramiento, causal, determinantes de la matriz de enmascaramiento causal-enmascaramiento causal, estimación de máxima verosimilitud del determinante jacobiano, teoría de la probabilidad central

MDN (red de densidad de mezcla), El MDN-RNN, Recolección

Datos para entrenar la pérdida de error cuadrático medio de MDN-RNN, funciones de pérdida

Megatron-Turing NLG, 2020-2022: la era del gran modelo,

Parámetro de métricas de modelos de lenguaje grandes, optimizadores

Archivos MIDI, Análisis de archivos MIDI

Midjourney, Introducción, Introducción, Texto a imagen

Modelos, modelos de texto a imagen

Mildenhall, Ben, Capa de desviación estándar de minibatch de incrustación sinusoidal, Desviación estándar de minibatch

Mirza, Mehdi, Distribuciones de mezclas de GAN condicional (CGAN), Distribuciones de mezclas-Mezcla

Distribuciones

MLP (ver perceptrones multicapa)

Conjunto de datos MNIST, colapso del modo del conjunto de datos MNIST, el generador domina el método discriminador model.summary(), inspeccionando el modelo, inspeccionando los modelos (ver también aprendizaje profundo generativo; Keras) teoría de probabilidad central, teoría de probabilidad central-núcleo

Teoría de la probabilidad de redes neuronales profundas, redes neuronales profundas, taxonomía de modelos generativos TensorFlow y Keras, modelo generativo

Modelado generativo de taxonomía, ¿Qué es el modelado generativo? modelado generativo versus discriminativo, generativo

Historia del modelado discriminativo versus, 2020-2022: mejora de la era de los grandes modelos, capacitación y evaluación de la red neuronal convolucional (CNN) del modelado paramétrico de CNN, teoría central de la probabilidad probabilística versus determinista, qué es generativo

¿Modelado? codificadores automáticos variacionales (VAE), variacionales codificadores automáticos

Arquitectura del modelo mundial, descripción general del modelo mundialResumen del paso de modulación, modulación y demodulación de peso Mordatch

Igor, Atención multicabezal de modelos basados en energía, Preparación de datos de perceptrones multicapa (MLP) de Atención multicabezal, Ejemplo de preparación de datos de ¿Qué es una red neuronal? construcción de modelos, construcción del modelo: inspección del modelo, aprendizaje supervisado y RNN multicapa de perceptrón multicapa (MLP), redes recurrentes apiladas, desafíos de modelos multimodales de generación de texto a imagen, Introducción

DALL.E 2, DALL.E 2-Limitaciones

Flamingo, Flamingo: ejemplos de la historia de Flamingo, 2020-2022: la gran era del modelo

Imagen, Difusión Imagen-Estable

Difusión estable, distribución normal multivariada de difusión estable, The Encoder

MUSE, 2020-2022: La era de los grandes modelos, texto a imagen

Modelos

MuseGAN, MuseGAN: análisis del análisis de MuseGAN, análisis del conjunto de datos de MuseGAN utilizado, conjunto de datos de Bach Chorale

Crítico de MuseGAN, El crítico de MuseGAN

Generador MuseGAN, El generador MuseGAN “MuseGAN: Adversario generativo secuencial multipista

Redes de Generación de Música Simbólica y

Acompañamiento” (Dong), MuseGAN

MuseNet, 2018-2019: La era de los transformadores

MuseScore, The Bach Cello Suite Dataset análisis de generación musical de la generación musical Transformer, análisis del transformador generador de música: análisis del

Conjunto de datos de Transformer generador de música utilizado, Conjunto de datos de Bach Cello Suite que genera música polifónica, Tokenización de

Música polifónica importando archivos MIDI, analizando entradas y salidas de archivos MIDI, múltiples entradas y salidas

MuseGAN, MuseGAN: análisis de la generación de música versus texto de MuseGAN, requisitos previos de introducción a la codificación de posición sinusoidal de Transformers for Music Generation, codificación de posición sinusoidal

Tokenización de codificación de posición, Conjunto de entrenamiento de tokenización para, Creación del conjunto de entrenamiento

Transformadores aplicados, 2018-2019: El transformador

Era

Music Transformer, 2018-2019: La era Transformer “Music Transformer: generando música con perspectivas a largo plazo

Structure” (Huang), Tokenización de la biblioteca de música polifónica music21, Análisis de archivos MIDI

MusicLM, Otras Aplicaciones

Nain, Aakash Kumar, Wasserstein GAN con gradiente

Penalización (WGAN-GP), procesamiento del lenguaje natural (NLP), otros transformadores

NCSN (Red de puntuación condicional de ruido), 2018-2019: La

Transformer Era “NeRF: Representación de escenas como campos de radiación neuronal para

View Synthesis” (Mildenhall), incrustación sinusoidal “Neural Discrete Representation Learning” (van den Oord),

Redes neuronales VQ-GAN (ver también redes neuronales convolucionales; aprendizaje profundo) redes neuronales profundas, Resumen definido, ¿Qué es una red neuronal? Funciones de pérdida de características de alto nivel de aprendizaje y papel de la teoría central de la probabilidad en el aprendizaje profundo, Taxonomía del modelo generativo

usando Keras para construir, TensorFlow y Keras, construyendo el modelo-inspeccionando el modelo

Nichol, Alex, 2020-2022: La era de los grandes modelos

PNL (procesamiento del lenguaje natural), Otros transformadores

Red de puntuación condicional de ruido (NCSN), 2018-2019: la

Ruido de la era del transformador, adición a etiquetas, entrenamiento de los parámetros no entrenables de DCGAN, predicción utilizando la distribución normal de normalización por lotes (distribución gaussiana), ecuación de cambio de variables del codificador que normaliza los modelos de flujo, cambio de variables

Técnica de cambio de ecuación de variables, Cambio de Variables, Descripción del Cambio de Variables, Normalización de Modelos de Flujo

FFJORD (Dinámica continua de forma libre para escalable

Modelos Generativos Reversibles), taxonomía del modelo generativo FFJORD, Modelo Generativo

Taxonomía

BRILLO, BRILLO

Determinante jacobiano, Conceptos clave detrás del determinante jacobiano, Introducción a la motivación, Normalización de flujos

Modelo RealNVP, RealNVP-Análisis del RealNVP

Modelo

observaciones, ¿Qué es el modelado generativo?

OPT, 2020-2022: la era de los grandes modelos, el lenguaje amplio

Optimizadores de modelos, compilando el modelo.

Osindero, Simon, sobreajuste de GAN condicional (CGAN), abandono

Conjunto de datos de flores de Oxford 102, conjunto de datos de flores

acolchado, acolchado

PaLM-E, otras aplicaciones

Artículos con código, parámetros de otros recursos, entrenables y no entrenables, predicción mediante modelado paramétrico de normalización por lotes, teoría de la probabilidad central

Partido, 2020-2022: la era de los grandes modelos, texto a imagen

Modelos

PatchGAN, regularización de longitud de ruta VQ-GAN, término de pérdida de percepción de regularización de longitud de ruta, VQ-GAN

asistentes personales, IA generativa en la cuadrícula de piano roll de la vida cotidiana, tokenización de cuadrícula pix2pix, VQ-GAN, 2014-2017: la era VAE y GAN

Análisis de PixelCNN, Análisis de PixelCNN-Análisis del

Historia de PixelCNN, PixelCNN, 2014-2017: VAE y GAN

Era de capas convolucionales enmascaradas, convolucional enmascarado

Distribuciones de mezcla de capas convolucionales enmascaradas, bloques residuales de distribuciones de mezcla, entrenamiento de bloques residuales-bloques residuales, entrenamiento de PixelCNN

PixelRNN, 2014-2017: normalización de píxeles de la era VAE y GAN, poesía de normalización de píxeles, modelos de lenguaje grandes

Point-E, Otras aplicaciones incrustación posicional, codificación posicional codificación posicional, codificación posicional-posicional

Codificación del colapso posterior, predicción VQ-GAN, mediante normalización por lotes, predicción mediante normalización por lotes

función de densidad de probabilidad, teoría central de la probabilidad,

Taxonomía del modelo generativo, distribuciones de probabilidad del codificador, teoría de probabilidad de distribuciones de mezcla, teoría central de la probabilidad-núcleo

Teoría de probabilidad

Concepto de entrenamiento progresivo de ProGAN, descripción de la normalización de Pixelwise del entrenamiento progresivo, historia de ProGAN, 2014-2017: resultados de la era VAE y GAN, entrenamiento progresivo de resultados, ingeniería de indicaciones de salidas de entrenamiento progresivo, indicaciones de modelos de texto a imagen, modelos de lenguaje grandes

consulta, Consultas, Claves y Valores preguntas y comentarios, Cómo Contactarnos

Radford, Alec, GAN convolucional profunda (DCGAN), elementos aleatorios (estocásticos) GPT, ¿Qué es generativo?

¿Modelado? ruido aleatorio, Entrenamiento del DCGAN, Uso de muestreo

Dinámica Langevin

RBM (máquina Boltzmann restringida), Otros basados en energía

Modelos

Análisis de RealNVP, Análisis de las capas de acoplamiento del modelo RealNVP, Conjunto de datos de capas de acoplamiento-capas de acoplamiento utilizado, Descripción del conjunto de datos de las dos lunas, Historia de RealNVP, 2014-2017: La era VAE y GAN pasando datos a través de capas de acoplamiento, Pasando datos a través de un capa de acoplamiento: pasar datos a través de una capa de acoplamiento, apilar capas de acoplamiento, apilar capas de acoplamiento, entrenar, entrenar el modelo RealNVP: entrenar el

Modelo RealNVP

Conjunto de datos de recetas, técnicas de regularización de redes neuronales recurrentes del conjunto de datos de recetas (ver RNN), aprendizaje por refuerzo de abandono (RL)

ChatGPT y ChatGPT definido, terminología clave de aprendizaje por refuerzo, proceso de aprendizaje por refuerzo, aprendizaje por refuerzo: The CarRacing

Ambiente

Aprendizaje reforzado a partir de la retroalimentación humana (RLHF),

ChatGPT

ReLU (unidad lineal rectificada), Truco de reparametrización de funciones de activación, Construcción del codificador VAE, El

Truco de reparametrización Aprendizaje de representación, Aprendizaje de representación Representación Aprendizaje de bloques residuales, Bloques residuales-Bloques residuales,

Máquina Boltzmann (RBM) restringida ResidualBlock-ResidualBlock, otras basadas en energía

Modelado del proceso de difusión inversa, El proceso de difusión inversa Modelado de recompensa del proceso de difusión inversa, recompensa ChatGPT, en aprendizaje por refuerzo, Aprendizaje por refuerzo

RLHF (Aprendizaje reforzado a partir de la retroalimentación humana),

ChatGPT

RMSE (error cuadrático medio), unión del codificador al decodificador

Optimizador RMSProp (propagación cuadrática media),

Optimizadores

Células bidireccionales RNN (redes neuronales recurrentes), unidades recurrentes cerradas (GRU) de células bidireccionales, historial de unidades recurrentes cerradas, red de memoria a corto plazo (LSTM)

Redes LSTM (memoria a corto plazo), Red de memoria a corto plazo (LSTM): análisis de LSTM

Arquitectura del modelo mundial MDN-RNN, El multicapa MDN-RNN, Redes recurrentes apiladas Redes recurrentes apiladas, Recurrentes apiladas

Error cuadrático medio de redes (RMSE), uniendo el codificador al decodificador

Optimizador de propagación cuadrática media (RMSProp),

Optimizadores

SAGAN (GAN de autoatención), GAN de autoatención (SAGAN),

2018-2019: espacio muestral de la era de los transformadores, flujos de escala de la teoría de la probabilidad central, capas de acoplamiento

Schmidhuber, Jurgen, Red de memoria a corto plazo (LSTM), Introducción, 2014-2017: técnica de comparación de puntuaciones de la era VAE y GAN, otros modelos basados en energía, modelos generativos basados en puntuaciones, modelos de difusión

Autoatención GAN (SAGAN), Autoatención GAN (SAGAN),

2018-2019: Las capas autorreferenciales de la era Transformer, T5

Modelos secuenciales (Keras), construcción del modelo-inspección del modelo

activación sigmoidea, funciones de activación de incrustación de posición sinusoidal, codificación de posición sinusoidal de incrustación sinusoidal, conexiones de salto de incrustación sinusoidal, bloques residuales, eliminación de ruido de U-Net

Modelo, El Bloque Transformador, Activación softmax sin crecimiento progresivo, Funciones de activación

Sparse Transformers, fase de estabilización de Transformers for Music Generation, entrenamiento progresivo

Ventajas de la difusión estable de los modelos de texto a imagen: texto a imagen

Arquitectura de modelos, Ejemplos de arquitectura generados por, Ejemplos de la historia de Difusión estable de, Difusión estable, 2020-2022: El gran modelo

Era de redes recurrentes apiladas, desviación estándar de redes recurrentes apiladas, curvas normales estándar del codificador, derivación del codificador, elementos estocásticos (aleatorios) de tokenización, qué es generativo

¿Modelado? gradiente estocástico dinámica de Langevin, muestreo usando

Variación estocástica de Langevin Dynamics, Variación estocástica

parámetro strides (Keras), datos estructurados de Stride, datos para mezcla de estilos de aprendizaje profundo, mezcla de estilos

StyleGAN, StyleGAN-Resultados de StyleGAN, 2018-2019:

La era de los transformadores

StyleGAN-XL, resultados de StyleGAN2, 2020-2022: el

Gran era del modelo

StyleGAN2, StyleGAN2-Salidas de StyleGAN2, 2018–

2019: Capas de subclases de Transformer Era, Creación del método de resumen del codificador VAE, Inspección del ajuste fino supervisado del modelo, Aprendizaje supervisado ChatGPT, Perceptrón multicapa (MLP),

Aprendizaje por refuerzo activación swish, La red de síntesis de la función energética, La red de síntesis

T5, T5-T5, 2018–2019: El método `tape.gradient()` de Transformer Era, Entrenamiento del variacional

Taxonomía del auto codificador, parámetro de temperatura de la taxonomía del modelo generativo, análisis de las redes temporales LSTM, generador MuseGAN

Generación de datos de texto TensorFlow, TensorFlow y Keras (ver también GPT)

Redes LSTM (memoria a corto plazo), Red de memoria a corto plazo (LSTM): análisis de LSTM

Extensiones RNN (red neuronal recurrente), Recurrente

Extensiones de red neuronal (RNN): ejemplo de generación de historias cortas de células bidireccionales, Introducción a datos de texto versus datos de imágenes, Trabajo con datos de texto, generación de texto versus música, Introducción a modelos de texto a 3D, Otras aplicaciones, modelos de texto a código, Texto a código Modelos de código: texto a código

Modelos modelos de texto a imagen, Introducción, Modelos de texto a imagen Modelos de texto a imagen Modelos de texto a música, Otras aplicaciones Modelos de texto a video, Otras aplicaciones Modelos multimodales de texto a X, Otras aplicaciones termodinámicas difusión, Línea de tiempo de los modelos de difusión de la IA, Línea de tiempo del paso de tiempo de la IA generativa, en aprendizaje por refuerzo, Refuerzo

Incorporación de tokens de aprendizaje, tokenización de codificación posicional basada en eventos, tokenización basada en eventos

grid, Tokenización de grid-Tokenización de notas para generación musical, Proceso de tokenización de, Tokenización-Tokenización

Toolformer, Modelos manejables de ética y desafíos de la IA generativa, Parámetros entrenables de la taxonomía del modelo generativo, Predicción utilizando datos de entrenamiento de normalización por lotes, ¿Qué es el modelado generativo? Proceso de formación, ¿Qué es una red neuronal?

Bloque transformador, El bloque transformador-El

Bloque transformador

Arquitecturas de Transformers (ver también GPT; generación de música) para Otros Transformers

BERT (representaciones de codificador bidireccional de

Transformadores), Otros Transformadores

ChatGPT, decodificador versus codificador ChatGPT-ChatGPT, descripción del enmascaramiento causal, Introducción

Historia de GPT-3 y GPT-4, GPT-3 y GPT-4, 2018-2019: la era de los transformadores

Transformers dispersos, transformadores para la música

Generación

T5, fase de transición T5-T5, Entrenamiento Progresivo

flujos de traducción, distribución normal truncada de capas de acoplamiento, truco de truncamiento de BigGAN, verdad de BigGAN, filtrado de ficción generada, The Discriminator,

ChatGPT, modelos de lenguaje grandes, IA generativa en

Conjunto de datos educativos sobre las dos lunas, El conjunto de datos de las dos lunas

Modelo de eliminación de ruido de U-Net, DownBlocks y UpBlocks del modelo de eliminación de ruido de U-Net, Pérdida no informativa GLIDE,

Curvas normales de unidades de pérdida no informativa, Las unidades codificadoras, ¿Qué es una red neuronal?, Datos no estructurados de la capa LSTM, Datos para aprendizaje profundo, aprendizaje no supervisado, Refuerzo Aprendizaje “Aprendizaje de representación no supervisado con profundidad

Red Adversaria Generativa Convolutional” (Radford),

Muestreo ascendente de GAN convolucional profundo (DCGAN), The Generator, Upsampler

VAE (ver codificadores automáticos variacionales)

VAE con un discriminador GAN (VAE-GAN), 2014-2017: El

Era VAE y GAN

vectores de valor, consultas, claves y valores van den Oord, Aaron, PixelCNN, problema de gradiente de fuga VQ-GAN, memoria larga a corto plazo

Variación de red (LSTM), análisis de los codificadores automáticos variacionales (VAE) del codificador (ver también codificadores automáticos), análisis de la arquitectura del auto codificador del auto codificador variacional, generación de codificadores automáticos

Nuevos decodificadores de imágenes, ajustes del codificador The Encoder, uso de generación de imágenes faciales The Encoder-The Encoder, taxonomía del modelo generativo The CelebA DatasetMorphing Between Faces, modelo generativo

Historia de la taxonomía, 2014-2017: Introducción a la era VAE y GAN, Introducción, artículo publicado sobre codificadores automáticos variacionales, Entrenamiento de codificadores automáticos variacionales, Entrenamiento del auto codificador variacional

Construcción de VAE en Keras, construcción del codificador VAE

Función de pérdida VAE, la función de pérdida

Resumen del modelo VAE, Creación del codificador VAE

Arquitectura del modelo mundial, el VAE

Entrenamiento del modelo mundial, Entrenamiento del modelo VAE-El decodificador

Vaswani, Ashish, incrustación sinusoidal, Introducción,

2014-2017: la era VAE y GAN

Red adversarial generativa cuantificada por vectores (VQGAN), VQ-GAN-VQ-GAN, 2020-2022: la era del gran modelo

VAE cuantificado vectorial (VQ-VAE), 2014-2017: VAE y

Era GAN “Modelado de imágenes cuantificadas por vectores con VQGAN mejorado” (Yu), ViT VQ-GAN

Transformador de visión (ViT), ViT VQ-GAN, CLIP, 2020-2022:

La era de los grandes modelos

Visual ChatGPT, Vocabulario ChatGPT, Tokenización

VQ-GAN (Adversario Generativo Cuantizado Vectorial)

Network), VQ-GAN-VQ-GAN, 2020-2022: El gran modelo

Era

VQ-VAE (VAE cuantificado por vectores), 2014-2017: El VAE y

Era GAN

Wasserstein GAN con análisis de penalización de gradiente (WGAN-GP), Análisis de la pérdida de penalización de gradiente de WGAN-GP, Pérdida de penalización de gradiente

Restricción de Lipschitz, La restricción de Lipschitz versus GAN estándar, Entrenamiento del WGAN-GP

entrenamiento, Tutorial de entrenamiento de WGAN-GP sobre Wasserstein GAN con penalización de gradiente (WGAN-GP)

Pérdida de Wasserstein, Recorte de peso de la pérdida de Wasserstein, Aplicación de la restricción de Lipschitz

Beneficios de Wasserstein GAN (WGAN), historia de Wasserstein GAN con penalización de gradiente (WGAN-GP), 2014-2017: recorte de peso de la era VAE y GAN, aplicación de la modulación y demodulación de peso de la restricción de Lipschitz, modulación de peso y demodulación-modulación de peso y Pesos de demodulación, ¿Qué es una red neuronal?

Welling, Max, codificadores automáticos variacionales

WGAN (ver GAN de Wasserstein)

Conjunto de datos de Wine Reviews, aplicaciones de trabajo del conjunto de datos de Wine Reviews, IA generativa en el lugar de trabajo

Arquitectura de modelos mundiales, Arquitectura: el controlador que recopila datos de entrenamiento MDN-RNN, Recopilación de datos para

Entrene el MDN-RNN recopilando datos de implementación aleatorios, recopilando datos aleatorios

Datos de implementación

artículo publicado sobre Introducción, MDN-RNN,

2014-2017: Entrenamiento en el sueño de la era VAE y GAN, Entrenamiento en el sueño en el sueño

Entrenamiento del proceso de entrenamiento, Entrenamiento del controlador, Entrenamiento del controlador Paralelización de CMA-ES, entrenamiento del MDN-RNN, Entrenamiento del MDN-RNN Sampleo

desde el MDN-RNN, entrenamiento del VAE, Entrenamiento del VAE-El modelo de decodificador

Arquitectura del modelo mundial, descripción general del modelo mundialResumen

Yu, Jiahui, ViT VQ-GAN

predicción de tiro cero, CLIP

Sobre el Autor

David Foster es un científico de datos, emprendedor y educador especializado en aplicaciones de IA en ámbitos creativos. Como cofundador de Applied Data Science Partners (ADSP), inspira y capacita a las organizaciones para aprovechar el poder transformador de los datos y la IA. Tiene una maestría en Matemáticas del Trinity College, Cambridge, una maestría en Investigación de Operaciones de la Universidad de Warwick y es miembro del cuerpo docente del Machine Learning Institute, con especialización en las aplicaciones prácticas de la IA y la resolución de problemas del mundo real. Sus intereses de investigación incluyen mejorar la transparencia y la interpretabilidad de los algoritmos de IA, y ha publicado literatura sobre aprendizaje automático explicable en la atención médica.

Colofón

El animal de la portada de Generative Deep Learning es un periquito pintado (*Pyrrhura picta*). El género *Pyrrhura* pertenece a la familia Psittacidae, una de las tres familias de loros. Dentro de su subfamilia Arinae se encuentran varias especies de guacamayos y periquitos del hemisferio occidental. El periquito pintado habita en los bosques costeros y montañas del noreste de Sudamérica.

Las plumas de color verde brillante cubren la mayor parte de un periquito pintado, pero son azules encima del pico, marrones en la cara y rojizas en el pecho y la cola. Lo más sorprendente es que las plumas del cuello del periquito pintado parecen escamas; el centro marrón está delineado en color blanquecino. Esta combinación de colores camufla a las aves en la selva tropical.

Los periquitos pintados tienden a alimentarse en el dosel del bosque, donde su plumaje verde los enmascara mejor. Se alimentan en bandadas de 5 a 12 aves en busca de una amplia variedad de frutas, semillas y flores. Ocasionalmente, cuando se alimentan debajo del dosel, los periquitos pintados comen algas de los estanques del bosque. Crecen hasta aproximadamente 9 pulgadas de largo y viven de 13 a 15 años.

Una nidada de polluelos de periquito pintado, cada uno de los cuales mide menos de una pulgada de ancho al nacer, suele tener alrededor de cinco huevos.

Muchos de los animales que aparecen en las portadas de O'Reilly están en peligro de extinción; Todos ellos son importantes para el mundo.

La ilustración de la portada es de Karen Montgomery, basada en un grabado en blanco y negro de Shaw's Zoology. Las fuentes de portada son Gilroy Semibold y Guardian Sans. La fuente del texto es Adobe Minion Pro; la fuente del encabezado es Adobe Myriad Condensada; y la fuente del código es Ubuntu Mono de Dalton Maag.