

O'REILLY®

Generative AI on AWS

Building Context-Aware
Multimodal Reasoning
Applications



Chris Fregly,
Antje Barth &
Shelbee Eigenbrode

Praise for *Generative AI on AWS*

I am very excited about this book—it has a great mix of all-important background/theoretical info and detailed, hands-on code, scripts, and walk-throughs. I enjoyed reading it, and I know that you will too! Starting from the basics, you will learn about generative foundation models, prompt engineering, and much more. From there you will proceed to large language models (LLMs) and will see how to use them from within Amazon SageMaker. After you master the basics, you will have the opportunity to learn about multiple types of fine-tuning, and then you will get to the heart of the book and learn to build applications that have the power to perform context-aware reasoning with generative models of different modalities including text and images.

—Jeff Barr, VP and Chief Evangelist @ AWS

This book is a comprehensive resource for building generative AI-based solutions on AWS. Using real-world examples, Chris, Antje, and Shelbee have done a spectacular job explaining key concepts, pitfalls, and best practices for LLMs and multimodal models. A very timely resource to accelerate your journey for building generative AI solutions from concept to production.

—Geeta Chauhan, Applied AI Leader @ Meta

In the process of developing and deploying a generative AI application, there are many complex decision points that collectively determine whether the application will produce high quality output and can be run in a cost-efficient, scalable, and reliable manner. This book demystifies the underlying technologies and provides thoughtful guidance to help readers understand and make these decisions, and ultimately launch successful generative AI applications.

— Brent Rabowsky, Sr. Manager AI/ML Specialist SA @ AWS

It's very rare to find a book that comprehensively covers the full end-to-end process of model development and deployment! If you're an ML practitioner, this book is a must!

—Alejandro Herrera, Data Scientist @ Snowflake

This book goes deep into how GenAI models are actually built and used. And it covers the whole life cycle, not just prompt engineering or tuning. If you're thinking about using GenAI for anything nontrivial, you should read this book to understand what skill sets and tools you'll need to be successful.

—Randy DeFauw, Sr. Principal Solution Architect @ AWS

There's no better book to get started with generative AI. With all the information on the internet about the topic, it's extremely overwhelming for anyone. But this book is a clear and structured guide: it goes from the basics all the way to advanced topics like parameter-efficient fine-tuning and LLM deployment. It's also very practical and covers deployment on AWS too. This book is an extremely valuable resource for any data scientist or engineer!

—Alexey Grigorev, Principal Data Scientist @ OLX Group and Founder @ DataTalks.Club

This is by far the best book I have come across that makes building generative AI very practical. Antje, Chris, and Shelbee put together an exceptional resource that will be very valuable for years—if possible, converted to a learning resource for universities. Definitely a must-read for anyone building generative AI applications at scale on AWS.

—Olalekan Elesin, Director of Data Science Platform @ HRS Group

If you're looking for a robust learning foundation for building and deploying generative AI products or services, look no further than Generative AI on AWS. Guided by the deep expertise of authors Chris Fregly, Antje Barth, and Shelbee Eigenbrode, this book will transition you from a GenAI novice to a master of the intricate nuances involved in training, fine-tuning, and application development. This manual is an indispensable guide and true necessity for every budding AI engineer, product manager, marketer, or business leader.

—Lillian Pierson, PE, Founder @ Data-Mania

Generative AI on AWS provides an in-depth look at the innovative techniques for creating applications that comprehend diverse data types and make context-driven decisions. Readers get a comprehensive view, bridging both the theoretical aspects and practical tools needed for generative AI applications. This book is a must-read for those wanting to harness the full potential of AWS in the realm of generative AI.

—Kesha Williams, Director @ Slalom Consulting and AWS Machine Learning Hero

The generative AI landscape evolves so fast that it's incredible to see so much relevant knowledge condensed into a comprehensive book. Well done!

—Francesco Mosconi, Head of Data Science @ Catalit

OceanofPDF.com

Generative AI on AWS

Building Context-Aware Multimodal Reasoning
Applications

**Chris Fregly, Antje Barth, and Shelbee
Eigenbrode**



Beijing • Boston • Farnham • Sebastopol • Tokyo

OceanofPDF.com

Generative AI on AWS

by Chris Fregly, Antje Barth, and Shelbee Eigenbrode

Copyright © 2024 Flux Capacitor, LLC, Antje Barth, and Shelbee Eigenbrode. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

- Acquisitions Editor: Nicole Butterfield
- Development Editor: Sara Hunter
- Production Editor: Gregory Hyman
- Copyeditor: nSight, Inc.
- Proofreader: Tove Innis
- Indexer: Sue Klefstad
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea
- November 2023: First Edition

Revision History for the First Edition

- 2023-11-13: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781098159221> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Generative AI on AWS*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-15922-1

[LSI]

OceanofPDF.com

Preface

After reading this book, you will understand the most common generative AI use cases and tasks addressed by industry and academia today. You will gain in-depth knowledge of how these cutting-edge generative models are built, as well as practical experience to help you choose between reusing an existing generative model or building one from scratch. You will then learn to adapt these generative AI models to your domain-specific datasets, tasks, and use cases that support your business applications.

This book is meant for AI/ML enthusiasts, data scientists, and engineers who want to learn the technical foundations and best practices for generative AI model training, fine-tuning, and deploying into production. We assume that you are already familiar with Python and basic deep-learning components like neural networks, forward propagation, activations, gradients, and back propagations to understand the concepts used here.

A basic understanding of Python and deep learning frameworks such as TensorFlow or PyTorch should be sufficient to understand the code samples used throughout the book. Familiarity with AWS is not required to learn the concepts, but it is useful for some of the AWS-specific samples.

You will dive deep into the generative AI life cycle and learn topics such as prompt engineering, few-shot in-context learning, generative model pretraining, domain adaptation, model evaluation, parameter-efficient fine-tuning (PEFT), and reinforcement learning from human feedback (RLHF).

You will get hands-on with popular large language models such as [Llama 2](#) and [Falcon](#) as well as multimodal generative models, including [Stable Diffusion](#) and [IDEFICS](#). You will access these foundation models through the [Hugging Face Model Hub](#), [Amazon SageMaker JumpStart](#), or [Amazon Bedrock managed service for generative AI](#).

You will also learn how to implement context-aware retrieval-augmented generation (RAG)¹ and agent-based reasoning workflows.² You will explore application frameworks and libraries, including [LangChain](#), [ReAct](#),³ and Program-Aided-Language models (PAL). You can use these frameworks and libraries to access your own custom data sources and APIs or integrate with external data sources such as web search and partner data systems.

Lastly, you will explore all of these generative concepts, frameworks, and libraries in the context of multimodal generative AI use cases across different content modalities such as text, images, audio, and video.

And don't worry if you don't understand all of these concepts just yet. Throughout the book, you will dive into each of these topics in much more detail. With all of this knowledge and hands-on experience, you can start building cutting-edge generative AI applications that help delight your customers, outperform your competition, and increase your revenue!

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

Used to call attention to snippets of interest in code blocks, as well as to differentiate among multiple speakers in dialogue, or between the human user and the AI assistant.

TIP

This element signifies a tip or suggestion.

NOTE

This element signifies a general note.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at <https://oreil.ly/generative-ai-on-aws-code>.

If you have a technical question or a problem using the code examples, please send email to support@oreilly.com.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Generative AI on AWS* by Chris Fregly, Antje Barth, and Shelbee

Eigenbrode (O'Reilly). Copyright 2024 Flux Capacitor, LLC, Antje Barth, and Shelbee Eigenbrode, 978-1-098-15922-1.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

O'Reilly Online Learning

NOTE

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit <https://oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-889-8969 (in the United States or Canada)

707-829-7019 (international or local)

707-829-0104 (fax)

support@oreilly.com

<https://www.oreilly.com/about/contact.html>

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at
<https://oreil.ly/generative-ai-on-aws>.

For news and information about our books and courses, visit
<https://oreilly.com>.

Find us on LinkedIn: *<https://linkedin.com/company/oreilly-media>*

Follow us on Twitter: *<https://twitter.com/oreillymedia>*

Watch us on YouTube: *<https://youtube.com/oreillymedia>*

Acknowledgments

We'd like to thank all of our reviewers, including Brent Rabowsky, Randy DeFauw, Sean Owen, Akhil Behl, and Sireesha Muppala, PhD. Your feedback was critical to the narrative that we followed in this book. Additionally, your guidance and intuition helped us modulate the technical depth of the code examples we included.

Chris

I dedicate this book to my mom, who has always inspired me to share knowledge with others. In addition, you have always listened patiently as I navigate life, question things, and seek answers.

Antje

I would like to thank my family for providing a great education and supporting me throughout my professional endeavors. In particular, I want to thank my brother, Kai, who bought me my first laptop and made sure I had the right tools for university. This was the initial catalyst to my career in computer science.

Shelbee

To my husband, Steve, and daughter, Emily, for always being “my why” and for their continued support, especially the late nights and long weekends writing this book. I also want to thank my dog, Molly, for sitting patiently while I took pictures of her to use as input for some of the multimodal models in this book!

¹ Patrick Lewis et al., “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks”, *arXiv*, 2021.

² Jason Wei et al., “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models”, *arXiv*, 2022.

³ Shunyu Yao et al., “ReAct: Synergizing Reasoning and Acting in Language Models”, *arXiv*, 2023.

Chapter 1. Generative AI Use Cases, Fundamentals, and Project Life Cycle

In this chapter, you will see some generative AI tasks and use cases in action, gain an understanding of generative foundation models, and explore a typical generative AI project life cycle. The use cases and tasks you'll see in this chapter include intelligent search, automated customer-support chatbot, dialog summarization, not-safe-for-work (NSFW) content moderation, personalized product videos, source code generation, and others.

You will also learn a few of the generative AI service and hardware options from Amazon Web Services (AWS) including [Amazon Bedrock](#), [Amazon SageMaker](#), [Amazon CodeWhisperer](#), [AWS Trainium](#), and [AWS Inferentia](#). These service and hardware options provide great flexibility when building your end-to-end, context-aware, multimodal reasoning applications with generative AI on AWS.

Let's explore some common use cases and tasks for generative AI.

Use Cases and Tasks

Similar to deep learning, generative AI is a general-purpose technology used for multiple purposes across many industries and customer segments. There are many types of multimodal generative AI tasks. We've included a list of the most common generative tasks and associated example use cases:

Text summarization

Produce a shorter version of a piece of text while retaining the main ideas. Examples include summarizing a news

article, legal document, or financial report into a smaller number of words or paragraphs for faster consumption. Often, summarization is used on customer support conversations to provide a quick overview of the interaction between a customer and support representative.

Rewriting

Modify the wording of a piece of text to adapt to a different audience, formality, or tone. For example, you can convert a formal legal document into a less formal document using less legal terms to appeal to a nonlegal audience.

Information extraction

Extract information from documents such as names, addresses, events, or numeric data or numbers. For example, converting an email into a purchase order in an enterprise resource planning (ERP) system like SAP.

Question answering (QA) and visual question answering (VQA)

Ask questions directly against a set of documents, images, videos, or audio clips. For example, you can set up an internal, employee-facing chatbot to answer questions about human resources and benefits documents.

Detecting toxic or harmful content

An extension to the question-answer task, you can ask a generative model if a set of text, images, videos, or audio clips contains any toxicity or harmful content.

Classification and content moderation

Assign a category to a given piece of content such as a document, image, video, or audio clip. For example, deleting

email spam, filtering out inappropriate images, or labeling incoming, text-based customer-support tickets.

Conversational interface

Handle multturn conversations to accomplish tasks through a chat-like interface. Examples include chatbots for self-service customer support or mental health therapy sessions.

Translation

One of the earliest use cases for generative AI is language translation. Consider, for example, that the publisher of this book wants to release a German translation to help expand the book's reach. Or perhaps you may want to convert the Python-based examples to Java to work within your existing Java-based enterprise application.

Source code generation

Create source code from natural language code comments—or even a hand-drawn sketch, as shown in [Figure 1-1](#). Here, an HTML- and JavaScript-based website is generated from a UI sketch scribbled on the back of a restaurant napkin.

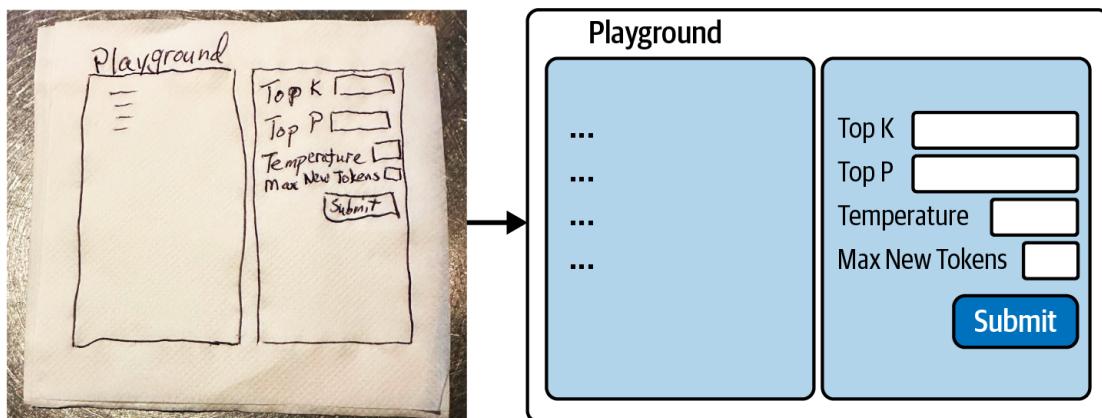


Figure 1-1. Generating UI code from hand-drawn sketch

Reasoning

Reason through a problem to discover potential new solutions, trade-offs, or latent details. For example, consider a CFO who provides an audio-based quarterly financial report to investors as well as a more-detailed written report. By reasoning through these different media formats together, the model may discover some conclusions about the company's health not directly mentioned in the audio or stated in the text.

Mask personally identifiable information (PII)

You can use generative models to mask personally identifiable information from a given corpus of text. This is useful for many use cases where you are working with sensitive data and wish to remove PII data from your workflows.

Personalized marketing and ads

Generate personalized product descriptions, videos, or ads based on user profile features. Consider an ecommerce website that wants to generate a personalized description for each product based on the logged-in user's age or family situation. You could also generate personalized product images that include mature adults, adults with children, or children themselves to better appeal to the logged-in user's demographic, as shown in [Figure 1-2](#).



Figure 1-2. Personalized marketing

In this case, each user of the service would potentially see a unique and highly personalized image and description for the same product. This could ultimately lead to more product clicks and higher sales.

In each of these generative use cases and tasks, a model creates content that approximates a human's understanding of language. This is truly amazing and is made possible by a neural network architecture called the transformer, which you will learn in [Chapter 3](#).

In the next section, you will learn how to access foundation models through model hubs.

Foundation Models and Model Hubs

Foundation models are very large and complex neural network models consisting of billions of parameters (a.k.a. weights). The model parameters are learned during the training phase—often called pretraining. Foundation models are trained on massive amounts of training data—typically over a period of many weeks and months using large, distributed clusters of CPUs and graphics processing units (GPUs). After learning billions of parameters,

these foundation models can represent complex entities such as human language, images, videos, and audio clips.

In most cases, you will start your generative AI projects with an existing foundation model from a model hub such as [Hugging Face Model Hub](#), [PyTorch Hub](#), or [Amazon SageMaker JumpStart](#). A model hub is a collection of models that typically contains detailed model descriptions including the use cases that they address.

Throughout this book, we will use Hugging Face Model Hub and SageMaker JumpStart to access foundation models like [Llama 2](#) from Meta (Facebook) and [Falcon](#) from the Technology Innovation Institute (TII) and [FLAN-T5](#) from Google. You will dive deeper into model hubs and foundation models in [Chapter 3](#).

Next, you'll see a typical generative AI project life cycle that roughly follows the outline of the rest of this book.

Generative AI Project Life Cycle

While there is no definitive project life cycle for generative AI projects, the framework shown in [Figure 1-3](#) can help guide you through the most important parts of your generative AI application journey. Throughout the book, you will gain intuition, learn to avoid potential difficulties, and improve your decision making at each step in the journey.

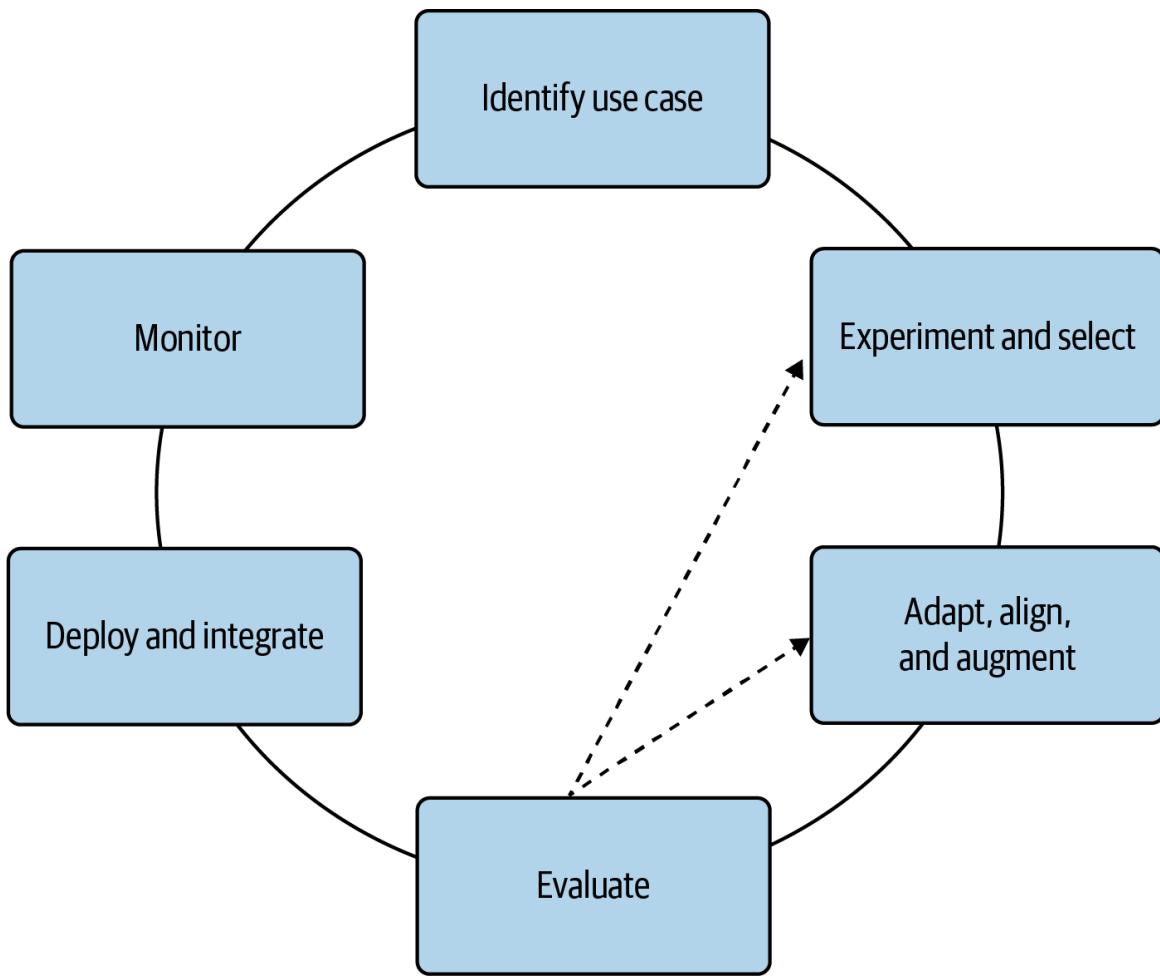


Figure 1-3. Generative AI project life cycle framework

Let's dive into each component of the life cycle shown in [Figure 1-3](#):

Identify use case.

As with any project, you first want to define your scope, including the specific generative use case and task that you plan to address with your generative AI application. We recommend that you start with a single, well-documented generative use case. This will help you get familiar with the environment and understand the power—and limitations—of these models without trying to optimize the model for different tasks at the same time. While these models are capable of carrying out multiple tasks, it's a bit more difficult

to evaluate and optimize the model across multiple tasks to start.

Experiment and select.

Generative AI models are capable of carrying out many different tasks with great success. However, you will need to decide if an existing foundation model is suitable for your application needs. In [Chapter 2](#), you will learn how to work with these existing foundation models right out of the box using techniques called *prompt engineering* and *in-context learning*.

Most commonly, you will start from an existing foundation model (as you will see in [Chapter 3](#)). This will greatly improve your time-to-market since you will avoid the pretraining step, which is extremely resource intensive and often requires trillions of words, images, videos, or audio clips to get started. Operating at this scale requires a lot of time, patience, and compute—often millions of GPU hours are required when pretraining from scratch.

You also want to consider the size of the foundation model you decide to work with as this will impact the hardware—and cost—needed to train and serve your models. While larger models tend to generalize better to more tasks, this is not always the case and depends on the dataset used during training and tuning.

We recommend that you try different models for your generative use case and task. Start with an existing, well-documented, relatively small (e.g., 7 billion-parameter) foundation model to iterate quickly and learn the unique ways of interacting with these generative AI models with a relatively small amount of hardware (compared to the larger 175+ billion-parameter models).

During development, you would typically start with a playground environment within either Amazon SageMaker JumpStart or Amazon Bedrock. This lets you try different prompts and models quickly, as you will see in [Chapter 2](#). Next, you might use a Jupyter notebook or Python script using an integrated development environment (IDE) like Visual Studio Code (VS Code) or Amazon SageMaker Studio notebooks to prepare your custom datasets to use when experimenting with these generative models. Once you are ready to scale your efforts to a larger distributed cluster, you would then migrate to SageMaker distributed training jobs to scale to a larger compute cluster using accelerators like the [NVIDIA GPU](#) or AWS Trainium, as you will see in [Chapter 4](#).

While you may be able to avoid accelerators initially, you will very likely need to use them for longer-term development and deployment of more complex models. The sooner you learn the unique—and sometimes obscure—aspects of developing with accelerators like NVIDIA GPUs or AWS Trainium chips, the better. Fortunately, a lot of the complexity has been abstracted by the hardware provider through the NVIDIA CUDA library and AWS Neuron SDK, respectively.

Adapt, align, and augment.

It's important to adapt generative models to your specific domain, use case, and task. Chapters [5](#), [6](#), [7](#), and [11](#) are dedicated to fine-tuning your multimodal generative AI models with your custom datasets to meet your business goals.

Additionally, as these generative models become more and more humanlike, it is important that they align with human values and preferences—and, in general, behave well.

Chapters 7 and 11 explore a technique called reinforcement learning from human feedback (RLHF) to align your multimodal generative models to be more helpful, honest, and harmless (HHH). RLHF is a key component of the much-broader field of responsible AI.

While generative models contain an enormous amount of information and knowledge, they often need to be augmented with current news or proprietary data for your business. In Chapter 9, you will explore ways to augment your generative models with external data sources or APIs.

Evaluate.

To properly implement generative AI applications, you need to iterate heavily. Therefore, it's important to establish well-defined evaluation metrics and benchmarks to help measure the effectiveness of fine-tuning. You will learn about model evaluation in Chapter 5. While not as straightforward as traditional machine learning, model evaluation helps measure improvements to your models during the adaptation and alignment phase—specifically, how well the model aligns to your business goals and human preferences.

Deploy and integrate.

When you finally have a well-tuned and aligned generative model, it's time to deploy your model for inference and integrate the model into your application. In Chapter 8, you will see how to optimize the model for inference and better utilize your compute resources, reduce inference latency, and delight your users.

You will also see how to deploy your models with the AWS Inferentia family of compute instances optimized for generative inference using Amazon SageMaker endpoints. SageMaker endpoints are a great option for serving

generative models as they are highly scalable, fault tolerant, and customizable. They offer flexible deployment and scaling options like A/B testing, shadow deployments, and autoscaling, as you will learn in [Chapter 8](#).

Monitor

As with any production system, you should set up proper metrics collection and monitoring systems for all components of your generative AI application. In Chapters [8](#) and [12](#), you will learn how to utilize Amazon CloudWatch and CloudTrail to monitor your generative AI applications running on AWS. These services are highly customizable, accessible from the AWS console or AWS software development kit (SDK), and integrated with every AWS service including Amazon Bedrock, a managed service for generative AI, which you will explore in [Chapter 12](#).

Generative AI on AWS

This section will outline the AWS stack of purpose-built generative AI services and features, as shown in [Figure 1-4](#), as well as discuss some of the benefits of using AWS for generative AI.

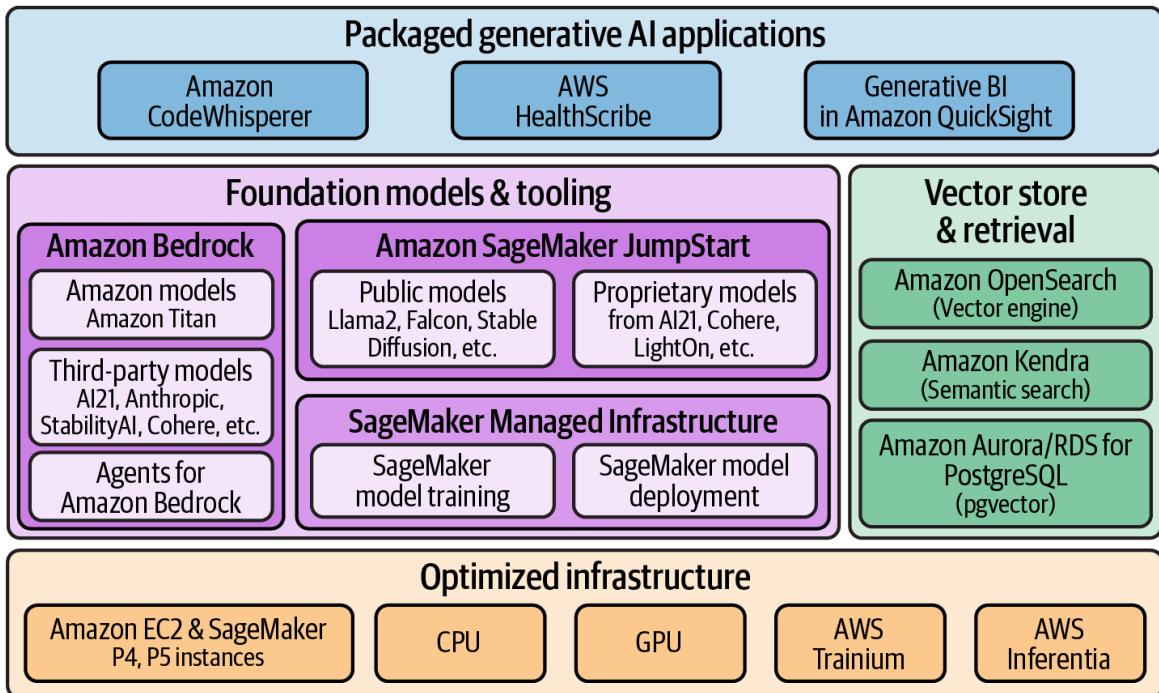


Figure 1-4. AWS services and features supporting generative AI

Model providers include those that are building or pretraining foundation models requiring access to powerful, and cost performant, compute and storage resources. For this, AWS offers a range of frameworks and infrastructure to build foundation models. This includes optimized compute instances for generative AI with self-managed options such as [Amazon EC2](#) as well as managed options like [Amazon SageMaker](#) for model training and model deployment. In addition, AWS offers its own accelerators optimized for training ([AWS Trainium](#)) and deploying generative models ([AWS Inferentia](#)).

AWS Trainium is an accelerator that is purpose-built for high-performance, low-cost training workloads. Similarly, AWS Inferentia is purpose-built for high-throughput, low-cost inference. The infrastructure options on AWS that are optimized for generative AI are used by model providers but also model tuners.

Model tuners include those that are adapting or aligning foundation models to their specific domain, use case, and task. This typically requires access to not only storage and compute resources but also tooling that helps enable these tasks through easy access to a range of foundation models while

removing the need to manage underlying infrastructure. In addition to the range of optimized infrastructure available on AWS, tuners also have access to a broad range of popular foundation models as well as tooling to adapt or align foundation models, including capabilities built into Amazon Bedrock and Amazon SageMaker JumpStart.

Amazon Bedrock is a fully managed service that provides access to models from Amazon (e.g., Titan) and popular third-party providers (e.g., AI21 Labs, Anthropic, Cohere, and Stability AI). This allows you to quickly get started experimenting with available foundation models. Bedrock also allows you to privately customize foundation models with your own data as well as integrate and deploy those models into generative AI applications. Agents for Bedrock are fully managed and allow for additional customization with the integration of proprietary external data sources and the ability to complete tasks.

Amazon SageMaker JumpStart provides access to both public and proprietary foundation models through a model hub that includes the ability to easily deploy a foundation model to Amazon SageMaker model deployment real-time endpoints. Additionally, SageMaker JumpStart provides the ability to fine-tune available models utilizing SageMaker model training. SageMaker JumpStart automatically generates notebooks with code for deploying and fine-tuning models available on the model hub.

Amazon SageMaker provides additional extensibility, through managed environments in **Amazon SageMaker Studio notebooks**, to work with any available foundation model, regardless of whether it's available in SageMaker JumpStart. As a result, you have the ability to work with any models accessible to you and are never limited in the models you can work with in Amazon SageMaker.

Adapting a model to a specific use case, task, or domain often includes augmenting the model with additional data. AWS also provides multiple implementation options for vector stores that store vector embeddings. Vector stores and embeddings are used for retrieval-augmented generation

(RAG) to efficiently retrieve relevant information from external data sources to augment the data used with a generative model.

The options available include [vector engine for Amazon OpenSearch Serverless](#) as well as the [k-NN plugin](#) available for use with [Amazon OpenSearch Service](#). In addition, both [Amazon Aurora PostgreSQL](#) and [Amazon Relational Database Services \(RDS\) for PostgreSQL](#) include vector stores capabilities through built-in [pgvector support](#).

If you are looking for a fully managed semantic search experience on domain-specific data, you can use [Amazon Kendra](#), which creates and manages the embeddings for you.

AWS offers multiple options if you want to access generative models through end-to-end generative AI applications. On AWS, you can build your own custom generative AI applications using the breadth and depth of services available; you can also take advantage of packaged, fully managed, services.

For example, [Amazon CodeWhisperer](#) provides generative coding capabilities across multiple coding languages, supporting productivity enhancements such as code generation, proactively scanning for vulnerabilities and suggesting code remediations, with automatic suggestions for code attribution.

[AWS HealthScribe](#) is another packaged generative AI service targeted toward the healthcare industry to allow for the automatic generation of clinical notes based on patient-clinician conversations.

Finally, [Amazon QuickSight Q](#) includes built-in generative capabilities allowing users to ask questions about data in natural language and receive answers as well as generated visualizations that allow users to gain more insights into their data.

This book will largely focus on the personas and tasks involved in the section “[Generative AI Project Life Cycle](#)”—as well as building generative AI applications. Many of the services highlighted in this section, such as [Amazon SageMaker JumpStart](#) and [Amazon Bedrock](#), will be referenced

throughout this book as you dive into specific areas of the generative AI project life cycle.

Now that we've introduced some core AWS services for generative AI, let's look at some of the benefits of using AWS to build generative AI applications.

Why Generative AI on AWS?

Key benefits of utilizing AWS for your generative AI workloads include increased flexibility and choice, enterprise-grade security and governance capabilities, state-of-the art generative AI capabilities, low operational overhead through fully managed services, the ability to quickly get started with ready-to-use solutions and services, and a strong history of continuous innovation. Let's dive a bit further into each of these with some specific examples:

Increased flexibility and choice

AWS provides flexibility not only in the ability to utilize a range of services and features to meet the needs of each use case, but also in terms of choice in generative models. This provides you with the ability to not only choose the right model for a use case, but to also change and continually evaluate new models to take advantage of new capabilities.

Enterprise-grade security and governance capabilities

AWS services are built with security and governance capabilities that are important to the most regulated industries. For example, SageMaker model training, SageMaker model deployment, and Amazon Bedrock support key capabilities around data protection, network isolation, controlled access and authorization, as well as threat detection.

State-of-the-art generative AI capabilities

AWS offers choice in generative AI models, from Amazon models as well as third-party provider models in Amazon Bedrock to open source and proprietary models offered through Amazon SageMaker JumpStart. Additionally, AWS has also invested in infrastructure like AWS Trainium and AWS Inferentia for training and deploying generative models at scale.

Low operational overhead

As previously discussed, many of the AWS services and features targeted toward generative AI are offered through managed infrastructure, serverless offerings, or packaged solutions. This allows you to focus on generative AI models and applications instead of managing infrastructure and to quickly get started with ready-to-use solutions and services.

Strong history of continuous innovation

AWS has an established history of rapid innovation built on years of experience in not only cloud infrastructure but artificial intelligence.

The AWS stack of services and features for supporting generative AI covers the breadth, depth, and extensibility to support every use case, whether you're a model provider, a tuner, or a consumer. In addition to the generative AI capabilities on AWS, a broader set of AWS services also supports the ability to build custom generative AI applications, which will be covered in the next section.

Building Generative AI Applications on AWS

A generative AI application includes more than generative models. It requires multiple components to build reliable, scalable, and secure

applications that are then offered to consumers of that application, whether they are end users or other systems, as shown in [Figure 1-5](#).

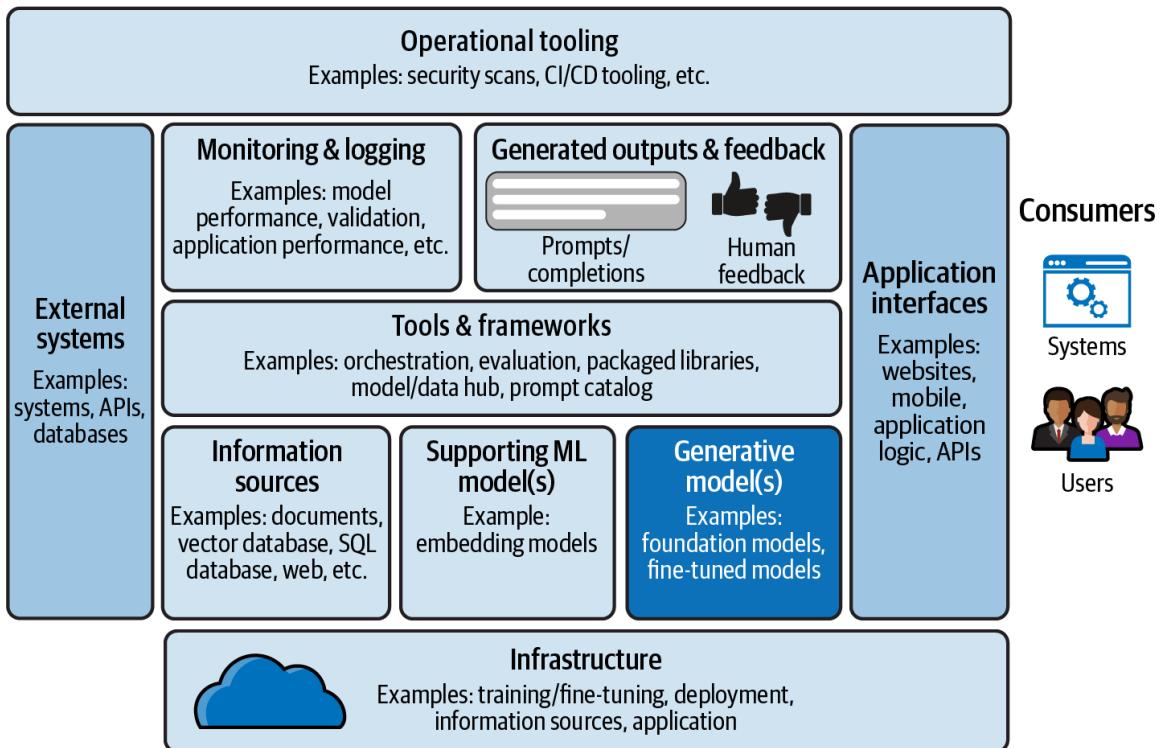


Figure 1-5. Generative AI applications include more than foundation models

When using a packaged generative AI service such as Amazon CodeWhisperer, all of this is completely abstracted and provided to the end user. However, building custom generative AI applications typically requires a range of services. AWS provides the breadth of services that are often required to build an end-to-end generative AI application. [Figure 1-6](#) shows an example of AWS services that may be used as part of a broader generative AI application.

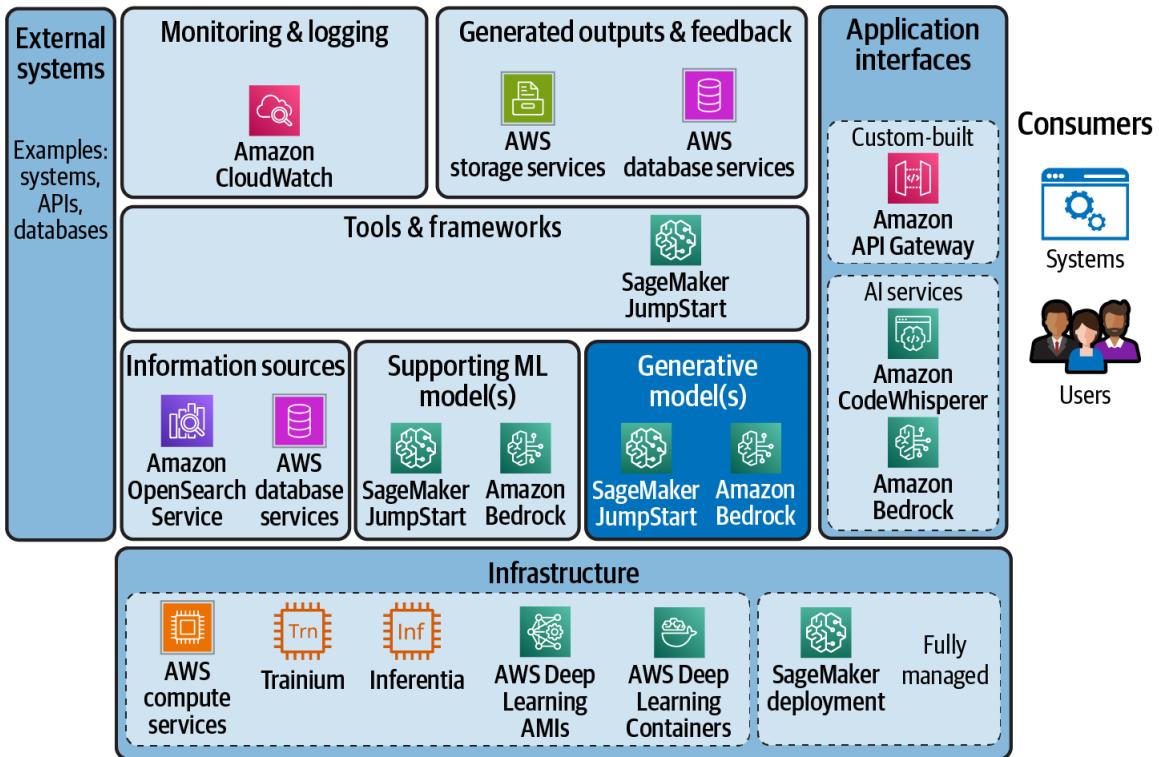


Figure 1-6. AWS breadth of service to enable customers to build generative AI applications

Summary

In this chapter, you explored some common generative AI use cases and learned some generative AI fundamentals. You also saw an example of a typical generative AI project life cycle that includes various stages, including defining a use case, prompt engineering ([Chapter 2](#)), selecting a foundation model ([Chapter 3](#)), fine-tuning (Chapters 5 and 6), aligning with human values ([Chapter 7](#)), deploying your model ([Chapter 8](#)), and integrating with external data sources and agents ([Chapter 9](#)).

The compute-intensive parts of the life cycle—including fine-tuning and human alignment—will benefit from an understanding of quantization and distributed-computing algorithms ([Chapter 4](#)). These optimizations and algorithms will speed up the iterative development cycle that is critical when developing generative AI models.

In [Chapter 2](#), you will learn some prompt engineering tips and best practices. These are useful for prompting both language-only foundation

models ([Chapter 3](#)) and multimodal foundation models (Chapters [10](#) and [11](#)) using either Amazon SageMaker JumpStart model hub ([Chapter 3](#)) or the Amazon Bedrock managed generative AI service ([Chapter 12](#)).

OceanofPDF.com

Chapter 2. Prompt Engineering and In-Context Learning

In this chapter, you will learn about low-code ways to interact with generative AI models—specifically, prompt engineering and in-context learning. You will see that writing prompts is both an art and a science that helps the model generate better and more-applicable responses. We also provide some best practices when defining prompts and prompt templates to get the most out of your generative models.

You will also learn how to use in-context-learning to pass multiple prompt-completion pairs (e.g., question-answer pairs) in the “context” along with your prompt input. This in-context learning nudges the model to respond similarly to the prompt-completion pairs in the context. This is one of the more remarkable capabilities of generative models as it temporarily alters the model’s behavior for the duration of just that single request.

Lastly, you will learn some of the most commonly configured generative parameters like `temperature` and `top k` that control the generative model’s creativity when creating content.

Language-based generative models accept prompts as input and generate a completion. These prompts and completions are made up of text-based tokens, as you will see next.

Prompts and Completions

While generative AI tasks can span multiple content modalities, they often involve a text-based input. This input is called a prompt and includes the instructions, context, and any constraints used to accomplish a given task.

Some examples of prompt instructions are “Summarize the following text” or “Who won the baseball World Series in 2016?” The model then responds

with a “completion” that returns the result of the task. This completion is often text-based, but it could be any content type the model is trained to output, such as text, image, video, or audio. You will learn how to optimize your prompts to achieve your desired completions later in this chapter on text-based prompts as well as in the coverage of multimodal prompts in [Chapter 11](#).

Tokens

It’s important to note that while text-based prompts and completions are implemented and interpreted by humans as natural language sentences, generative models convert them into sequences of *tokens*, or word fragments. By combining many of these tokens in different ways, the model is capable of representing an exponential number of words using a relatively small number of tokens—often on the order of 30,000–100,000 tokens in the model’s vocabulary.

TIP

As a rule of thumb, it’s common to approximate 1.3 tokens per word, but this multiplier can vary. You would use the 1.3 multiplier when estimating the cost of services that use token-based pricing. Often, these services charge per million tokens.

Using this small vocabulary, a language model is capable of learning and understanding human language during the model pretraining phase. During pretraining, the model sees millions of documents as part of the training dataset. From the model’s standpoint, a document is simply just a sequence of tokens from the model’s vocabulary. You will learn more about the model pretraining phase in [Chapter 3](#).

Machine learning and AI models, including generative AI models, rely on statistics and linear algebra for their computations, including probability modeling, loss functions, and matrix multiplications. These calculations

power all deep learning operations, and they prefer to work with numbers and not raw text, images, or videos.

Prompt Engineering

Prompt engineering is a new and exciting skill focused on how to better understand and apply generative models to your tasks and use cases. Effective prompt engineering helps you get the most out of your generative AI models and applications.

The input that you send into a generative model is typically called the *prompt*. The prompt can include text for large language models, or other modalities, such as images, video, for multimodal models. This prompt is passed to the model during inference time to generate a “completion.”

Here is a simple example question-answer prompt and completion.

Prompt:

Who won the 2016 baseball World Series?

Completion:

The Chicago Cubs won the 2016 baseball World Series, defeating the Cleveland Indians in a thrilling seven-game series. It was the Cubs' first World Series title in 108 years, ending the longest championship drought in Major League Baseball history.

You may have to experiment with your prompt several times to get a proper and precise response, as some of these generative models are quite chatty. Prompt engineering is a learned skill that requires many iterations across many different model types and linguistic nuances. These nuances often depend on how the model was trained.

Most modern human-facing chat models have been fine-tuned using some form of human-labeled data—often with reinforcement learning, which you will explore in [Chapter 7](#). Chat models sometimes include extra markup in the prompt and completion to indicate the role and distinguish between the

user input (for example, with `User:`) and the model response (for example, with `Assistant:`). In prompt terminology, `User:` is often referred to as the input indicator and `Assistant:` as the output indicator.

Here is the same example question-answer prompt and completion, this time using a model that has been trained to act as an AI assistant and uses input and output indicators.

Prompt:

`User:` Who won the 2016 baseball World Series?

Completion:

`Assistant:` The Chicago Cubs won the 2016 baseball World Series, defeating the Cleveland Indians in a thrilling seven-game series. It was the Cubs' first World Series title in 108 years, ending the longest championship drought in Major League Baseball history.

Note that prompt structures are very model-specific. Using different input and output indicators may result in “off-distribution” and undesirable results. Therefore, always look up the prompt structure when you start experimenting with a new generative AI model. You can usually find this information in model documentation such as the model card.

Let’s explore some common prompt structures and techniques to get the most out of off-the-shelf generative AI models.

Prompt Structure

The prompt structure used in the previous example is a simple chat-assistant structure. A more complete prompt structure usually includes a section for each of the following: instruction, context, input data, and output indicator. Let’s discuss the two additional prompt elements, instruction and context.

Instruction

An instruction is a piece of text that you pass to the model that describes the task you want the model to perform. For example, you can add an instruction to our previous example to guide the model to generate a one-sentence summary.

Prompt:

User: Answer the following question in one sentence.
Who won the 2016 baseball World Series?

Completion:

Assistant: The Chicago Cubs won the 2016 baseball World Series, defeating the Cleveland Indians in a thrilling seven-game series that ended with a final score of 8-7 in extra innings.

And indeed, the model responded with a single sentence.

Writing effective instructions for generative AI models involves clarity, specificity, and context to guide the model toward the desired output. Instructions should be straightforward and easy to understand. The more specific your instruction, the better the model can fulfill your request. Providing relevant context can help the model better understand the task or topic at hand.

Context

Context refers to relevant information or details that you pass to the model, so it better understands the task or topic and responds appropriately. This context can include preceding dialogue, background information, specific user requests, or any data that provides a frame of reference for the model to generate more contextually relevant and accurate responses. Passing context allows for more coherent and meaningful interactions with the model.

A popular technique to guide the model's response toward desired output is to share example prompt-completion pairs as context information. Depending on how many examples you provide, this is called *one-shot* or

few-shot inference. The model’s ability to learn from those examples and adapt its responses accordingly is called “in-context learning.” You will explore in-context learning with few-shot inference in the next section.

Examples 2-1, 2-2, and 2-3 show a restructured version of the previous chat example using the more complete prompt structure, including an instruction, and three prompt-completion examples in the context, followed by input data and the output indicator.

Example 2-1. Instruction

User: Answer the question using the format shown in the context.

Example 2-2. Context

Who won the baseball World Series in 2022?

The Houston Astros won the World Series in 2022. They defeated the Philadelphia Phillies.

Who won the baseball World Series in 2021?

The Atlanta Braves won the World Series in 2021. They defeated the Houston Astros.

Who won the baseball World Series in 2020?

The Los Angeles Dodgers won the World Series in 2020. They defeated the Tampa Bay Rays.

Example 2-3. Input data and output indicator

Who won the baseball World Series in 2016?

Assistant:

Let’s check the completion:

The Chicago Cubs won the World Series in 2016. They defeated the Cleveland Indians.

You can see how the model learned from the examples in the context and generated a completion in the desired format. Specifically, the assistant responded with a succinct answer that does not include extra details such as the final score of the baseball game—or the number of games in the series, as in the previous example.

The ideal prompt structure may vary depending on the task as well as the size of the model’s context window. The context window refers to the number of tokens the model can take as input when generating completions. Each model has a fixed context window size—anywhere from 512 tokens for [FLAN-T5](#) to 100,000 tokens for [Anthropic’s Claude model](#). For reference, Falcon has a context window size of 2,048 and Llama 2 has a context window size of 4,096. The context window size is often due to algorithmic limitations of the underlying neural network architecture. Also, in practice, you may see the model not fully utilizing a long sequence. This is often called “forgetting.” It’s important to test longer sequences and not assume the model will process 100,000 tokens the same way it would process an input of 1,000 tokens.

TIP

Some models document a single value: the maximum number of tokens. This number represents the combined total number of input tokens and generated output tokens.

The best prompt structure depends on how the generative model was trained and fine-tuned. Therefore, it’s important to read the documentation, specifically the model card, for a given generative model to gain intuition into the prompt structure used during training and tuning. Optimizing the prompt and prompt structure is all part of prompt engineering!

Next, you will learn how to further enrich the prompt context to evoke an emergent and thought-provoking property of generative AI models called in-context learning.

In-Context Learning with Few-Shot Inference

A powerful technique to help your generative model produce better completions for your prompt is to include a few prompt-completion pairs inside the context portion of your prompt. This is called in-context learning with few-shot inference.

It's worth noting that in-context learning does not modify the model in any way. The model adjusts—or learns—on the fly for the duration of that single request using the context provided in the prompt. This is a truly remarkable characteristic of generative models that can be used in many creative ways. You actually saw this in the previous human-assistant example that included a few examples, called shots, as part of the context.

In that case, adding just a few examples to the context helped the model respond with just the winner of the baseball World Series, without all of the other details generated in the examples before.

As you may have guessed, if you pass one prompt-completion pair into the context, this is called one-shot inference; if you pass no example at all, this is called *zero-shot* inference.

Zero-shot inference is often used to evaluate a model's ability to perform a task that it hasn't been explicitly trained on or seen examples for. For zero-shot inference, the model relies on its preexisting knowledge and generalization capabilities to make inferences or generate appropriate outputs, even when it encounters tasks or questions it has never seen before.

Larger models are typically able to capture more understanding of language (or perception, in the case of image-based models) and tend to generalize better to tasks they haven't seen during their training phase. The largest models are surprisingly good at zero-shot inference and are able to infer and successfully complete many tasks that they were not specifically trained to perform.

In contrast, smaller models are generally only good at a small number of tasks, typically those that are similar to the task they were trained on. As part of the `experiment and select` step described in [Chapter 1](#), you may have to try out a few models to find the right one for your use case. It's worth noting that smaller models are typically faster during inference because of the reduced number of layers and parameters required for the computation.

Let's look at examples of zero-shot, one-shot, and few-shot inference to help you visualize their differences in both prompt context and model

response.

Zero-Shot Inference

Here's an example using zero prompts in the context.

Prompt:

User: Who won the 2016 baseball World Series?

Completion:

Assistant: The Chicago Cubs won the 2016 baseball World Series, defeating the Cleveland Indians in a thrilling seven-game series. It was the Cubs' first World Series title in 108 years, ending the longest championship drought in Major League Baseball history.

The zero-shot completion still includes additional information, because the model didn't have any example to follow.

One-Shot Inference

The following example adds an instruction and one-shot prompt in the context.

Prompt:

User: Answer the question using the format shown in the context.

Who won the baseball World Series in 2022?

The Houston Astros won the World Series in 2022. They defeated the Philadelphia Phillies.

Who won the 2016 baseball World Series?

Assistant:

Completion:

The 2016 World Series was won by the Chicago Cubs. They defeated the Cleveland Indians in seven games, becoming the first team to win the World Series after a 108-year drought.

The one-shot completion is closer to our desired output but still has a little bit too much detail.

Few-Shot Inference

The following example shows what happens if you add few-shot prompt in the context.

Prompt:

User: Answer the question using the format shown in the context.

Who won the baseball World Series in 2022?

The Houston Astros won the World Series in 2022. They defeated the Philadelphia Phillies.

Who won the baseball World Series in 2021?

The Atlanta Braves won the World Series in 2021. They defeated the Houston Astros.

Who won the baseball World Series in 2020?

The Los Angeles Dodgers won the World Series in 2020. They defeated the Tampa Bay Rays.

Who won the 2016 baseball World Series?

Assistant:

Completion:

The Chicago Cubs won the World Series in 2016. They defeated the Cleveland Indians.

Note how the model generates a more focused and relevant response for just that request as you add more in-context examples called “shots.”

TIP

2016 was a great year for one of the authors of this book, who is a lifelong Chicago Cubs fan!

There are subtle differences between the zero-shot, one-shot, and few-shot completions. With more examples, or shots, the model more closely follows the pattern of the response of the in-context prompt-completion pairs.

In-Context Learning Gone Wrong

It's worth noting that you can "trick" a model into temporarily learning an incorrect answer. For example, you can pass three in-context prompt-completion examples that demonstrate a positive customer review as a *negative* sentiment and a negative customer review as a *positive* sentiment.

Examples 2-4, 2-5, and 2-6 show a few-shot, in-context prompt with incorrect sentiment.

Example 2-4. Instruction

Classify the sentiment of the review using the format shown in the context.

Example 2-5. Context

Wrigley Field is a fun stadium to watch a baseball game on a day with a clear blue sky.

Sentiment: NEGATIVE

The hungry seagulls during the 9th inning of a home Cubs game are very aggressive and annoying.

Sentiment: POSITIVE

Murphy's is my favorite bar in Chicago on a Friday afternoon right before the start of a Cubs game.

Sentiment: NEGATIVE

Example 2-6. Input data and output indicator

Harry Caray was the best Cubs sports announcer of all time!

Sentiment:

Completion:

NEGATIVE

In this case, inference requests made to the model with this prompt are more likely to return the opposite sentiment. This is a peculiar but

interesting quality of in-context learning, so it's worth double-checking your in-context prompt-completion pairs carefully.

In-Context Learning Best Practices

When you start experimenting with a new generative model, try zero-shot inference. If the model doesn't generate the results you're looking for, try one-shot and few-shot inference. For few-shot inference, it's important to provide a consistent and appropriate mix of prompt-completion examples that represent the dataset and allow the model to properly learn from the provided context. Also, you should make sure that your context does not increase your prompt length above the input size or "context window" of the given generative model.

In-context learning is very useful, but the ability and limits for in-context learning vary across models. If you find yourself using upwards of five or six examples in your context and still not seeing the results you're looking for, you may need to choose a different model or fine-tune an existing model. In Chapters 5, 6, and 7, you will explore various methods to fine-tune a foundational model.

In [Chapter 9](#), you will see how to further augment the prompt using external data sources such as databases and knowledge stores. This is called retrieval-augmented generation (RAG) and is part of the larger generative AI ecosystem that helps augment prompts with domain knowledge. RAG improves model responses across many generative tasks and use cases.

Next, you'll explore some prompt-engineering best practices to improve the responses from your generative AI models.

Prompt-Engineering Best Practices

Constructing an effective prompt is both an art and a science. The following are some best practices to help you construct effective prompts for better generative results:

Be clear and concise.

Prompts should be simple, straightforward, and avoid ambiguity. Clear prompts lead to more coherent responses. A general rule of thumb is this: if the wording is confusing to humans, it is likely to be confusing to these generative models. Simplify when possible.

Be creative.

New and thought-provoking prompts can lead to unexpected, better, sometimes even innovative model completions.

Move the instruction to the end of the prompt for large amounts of text.

If the context and input data are long, try moving the instruction to the end, right before the output indicator, as shown in the next example.

Prompt with small amount of input data and the instruction at the beginning:

Summarize the following conversation:

Customer: How do I cancel my subscription account?

Support Representative: Click the cancel button at this link: ...

Customer: Thanks, that worked!

Summary:

Prompt with long input data and the instruction at the end of the prompt:

Customer: Where is my order? It hasn't been delivered yet.

Support Representative: Can you give me your order ID, phone number, and last 4 digits of your credit card number?

Long conversation between customer and support representative...

Summarize the conversation above.

Summary:

Clearly convey the subject.

Clearly state one of the following: who, what, where, when, why, how, etc.

Use explicit directives.

If you want the model to output in a particular format, specify that directly. For example, “Summarize the following customer-support dialog in a single sentence:”.

Avoid negative formulations.

Negative formulations, while syntactically correct, may cause confusion. For example, use “Summarize in 5 sentences or less” instead of “Summarize in no more than 5 sentences.” Avoid negative formulations if a more straightforward linguistic variation exists.

Include context and few-shot example prompts.

Provide additional context that helps the model respond more accurately. You can specify a single context across all inputs or a specific context for each input. You have already

seen examples of including additional context in this chapter.

Specify the size of the response.

Include the requested output size at the end of the prompt to focus the model. For example: “List the top 3 complaints from the following customer-support conversation:”

Provide a specific response format.

Give the response format using an example. Include brackets for clarity. For example, “Summarize this document article in 10 words or less as shown here: [New generative AI model beats X benchmark by Y %.]”

Define what to do if the model can't answer confidently.

You can often ask the model to respond with, “I don’t know,” if it cannot confidently respond to the prompt. Otherwise, the model may generate a “hallucination” response, as shown in this example.

Prompt:

Which vehicle needs more down payment based on the following information:

The total cost of vehicle A is \$40000 and it needs 30% as down payment.

The total cost of vehicle B is \$50000 and it needs 20% as down payment.

Completion with an incorrect response (hallucination):

Based on the information provided, vehicle B requires a higher down payment than vehicle A. The down payment for vehicle B is \$10000, which is \$2000 higher than the down payment for vehicle A.

However, it is important to note that the down payment is not the only factor to consider when purchasing a vehicle.

In this example, the model correctly calculated the down payment for vehicle B (\$ 10,000), but incorrectly stated that it is \$2,000 more than the down payment for vehicle A. The correct answer would be that down payment for vehicle A is higher (\$12,000). The model didn't know the right answer and just "hallucinated" a response.

NOTE

While hallucinations are often fun to share with your coworkers during development, they are not fun to share with your end users in production!

Let's see how you can fix this. Here's an example prompt that allows the model to respond with "I don't know" in such cases.

Prompt:

Which vehicle needs more down payment based on the following information.

If you are not confident, just answer "I don't know."

The total cost of vehicle A is \$40000 and it needs 30% as down payment.

The total cost of vehicle B is \$50000 and it needs 20% as down payment.

Completion:

I don't know.

A more advanced prompting technique to address such scenarios is called chain-of-thought (CoT) prompting. CoT prompting shows the model how to reason through more complex tasks step-by-step to find the correct answer.

Ask the model to “think step-by-step.”

If the model is confused about the instructions, you can ask the model to “think step-by-step,” which gives the model the freedom to break a single instruction into multiple steps.

Here’s an example of a prompt using chain-of-thought (think step-by-step).

Prompt:

Which vehicle needs more down payment based on the following information

(think step-by-step):

The total cost of vehicle A is \$40000 and it needs 30% as down payment.

The total cost of vehicle B is \$50000 and it needs 20% as down payment.

Completion:

Based on the information provided, vehicle A needs a higher down payment.

The total cost of vehicle A is \$40000 and it requires 30% down payment, which comes to \$12000.

The total cost of vehicle B is \$50000 and it requires 20% down payment, which comes to \$10000.

Therefore, vehicle A needs a higher down payment.

Note how the model now reasoned through the task and generated the correct response.

Depending on how they were trained and tuned, some models may respond to other variants of “think step-by-step” such as “divide into subtasks,” “approach the problem systematically,” “reason through the problem one step at a time,” etc.

Some models may get confused when asked to perform a complex instruction that requires multiple steps. It’s important to recognize when the model is getting confused—and when you need to break the prompt down into multiple steps manually.

Add constraints for more control.

Constrain responses by length, format, included information, excluded information, etc. For example, “Summarize this process in exactly 5 steps:”.

Evaluate the response.

This seems obvious, but it’s worth noting that you should review the models’ responses to ensure the responses are high quality and appeal to your audience. Make changes to the prompts as needed.

NOTE

Evaluating responses at scale is an open area of research. Human evaluation does not scale well, and automated evaluation may miss the nuances of human language. You will explore model evaluation in more detail in [Chapter 5](#).

Use disclaimers or avoid prompts that the model should not answer.

If your generative model is not equipped to respond to certain domains like law, medicine, or religion, you can instruct your model to respond with something like, “I am not licensed to provide medical advice. Please seek a licensed medical professional in your area.”

Use XML/HTML tags in your prompt.

Some models support XML/HTML tags like `<tag>this is important</tag>` to create structure within the prompt. For example, if you want to reference an important piece of text in your input data, you can wrap that text in a tag to indicate where the important text starts and ends. You also ask some models to tag important parts of the response so you can parse the response and extract important data in a structured way.

Selectively focus.

You can ask the model to only focus on certain parts of the input text. For example, you can ask that the model summarize only the first and last paragraph of your input data.

Mask personally identifiable information (PII).

Some models can mask PII data in the response. You can ask these models to “remove all personally identifiable information and replace it with XXX,” for instance. Here is an example.

Prompt:

Human: Here is some text inside the `<text></text>` XML tags. Please remove all personally identifiable information from this text and replace it with XXX. It is very important that names, phone numbers, and email addresses

are replaced with XXX.

<text>

Joe: Hi Hannah!

Hannah: Hi Joe! Are you coming over?

Joe: Yup! Hey I, uh, forgot where you live."

Hannah: No problem! It's 1234 Wrightwood St, Chicago IL 60614.

Joe: Got it, thanks!

</text>

Please put your sanitized version of the text with PII removed in

<response></response> XML tags

Completion:

Assistant:

<response>

XXX: Hi XXX!

XXX: Hi XXX! Are you coming over?

XXX: Yup! Hey I, uh, forgot where you live."

XXX: No problem! It's XXX XXX, XXX XXX XXX XXX.

XXX: Got it, thanks!

</response>

By trying different prompts, and combining prompt engineering techniques, you see what works and what doesn't work for your prompt, model, and use case combination. Continue to refine your prompt as needed. With more

and more experimentation, you will gain the necessary intuition to quickly create and optimize a prompt to best suit your task and use case. Prompt engineering is an iterative skill that improves with practice, but prompt optimization is not as clear or well-studied as classical numerical optimization techniques, which you may find frustrating.

Take time to explore the creative and nondeterministic side of generative AI. At a minimum, you'll enjoy a good laugh when the model surprises you with a seemingly random response to a question that you did not intend to ask.

Next, you will learn some common generative inference-specific parameters that influence the creativity of the generative model response. This is where the fun begins!

Inference Configuration Parameters

Let's examine configuration parameters to influence the way generative models generate text during inference. If you've used generative models in a "playground" such as Amazon SageMaker or Bedrock, you have likely seen slides and other numerical controls like the ones shown in [Figure 2-1](#).

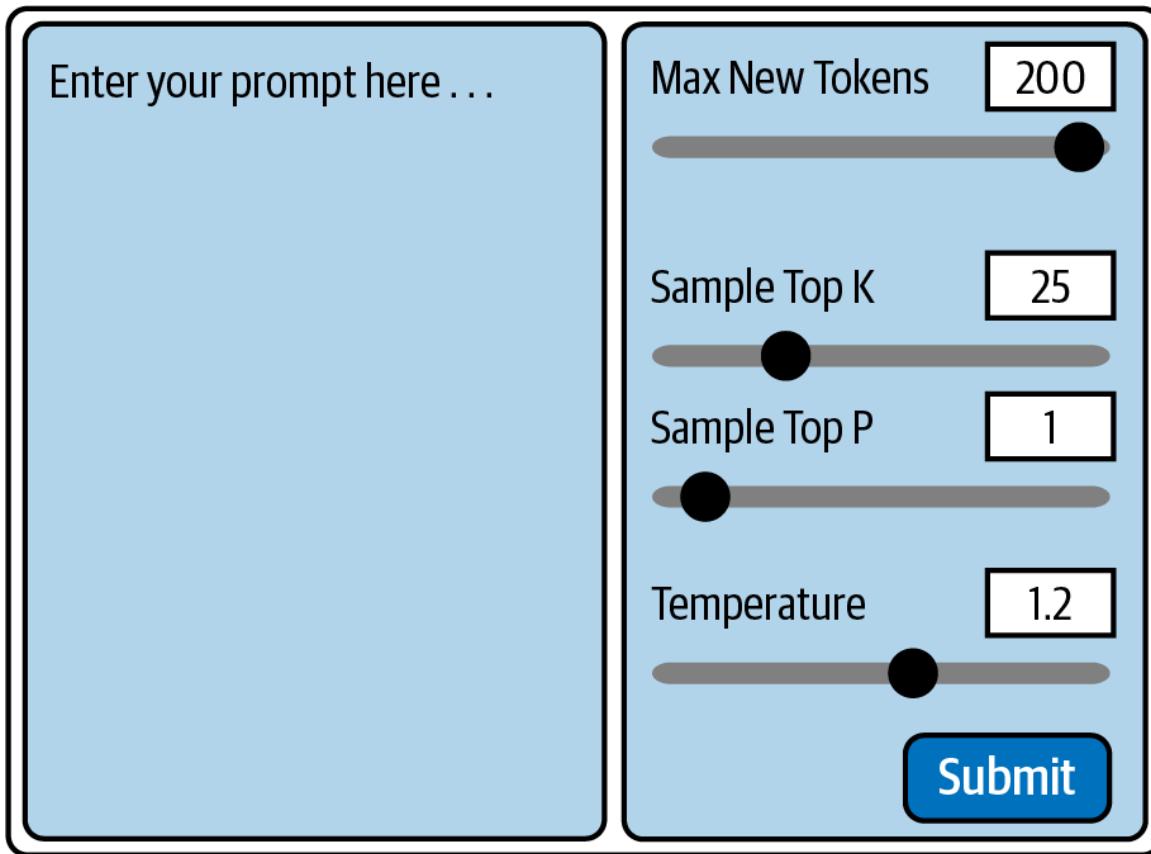


Figure 2-1. Inference configuration parameters to control model outputs

These inference configuration parameters influence the model's completion to your prompt. They give you fine-grained control over the length of the model response as well as the creativity. Each model exposes a different but often overlapping set of inference parameters. Often, these parameters are named similarly enough across models to reason through when you try out different models. Here are a few of the most common inference parameters:

Max new tokens

This is one of the most obvious and straightforward parameters to tune. Use this parameter to limit the number of new tokens generated by the model. This is a very basic mechanism to keep model responses short and prevent rambling. Note that generating more tokens generally requires more computational resources and may result in longer inference times. Also note that reducing max new

tokens is not a mechanism to prevent hallucinations; this may merely mask the hallucination by reducing its length.

Greedy versus random sampling

During model inference, the model produces a probability distribution across all tokens in the model's known vocabulary. The model chooses—or samples—a single token from this distribution as the next token to include in the response.

For each inference request, you can configure the model to choose the next token using either greedy or random sampling. For greedy sampling, the token with the highest probability is selected. With random sampling, the model selects the next token using a random-weighted strategy across all predicted token probabilities. The different sampling methods are shown in [Figure 2-2](#) for the phrase “the student learns from the professor and her lectures.”

Most generative model-inference implementations default to greedy sampling, also called greedy decoding. This is the simplest form of next-token prediction, as the model always chooses the word with the highest probability. This method works well for very short generations but may result in repeated tokens or sequences of tokens.

If you want to generate text that is more natural and minimizes repeating tokens, you can configure the model to use random sampling during inference. This will cause the model to randomly choose the next token using a weighted strategy across the probability distribution. The token `student`, as shown here, has a probability score of 0.02. With random sampling, this equates to a 2% chance that this word will be selected from the distribution.

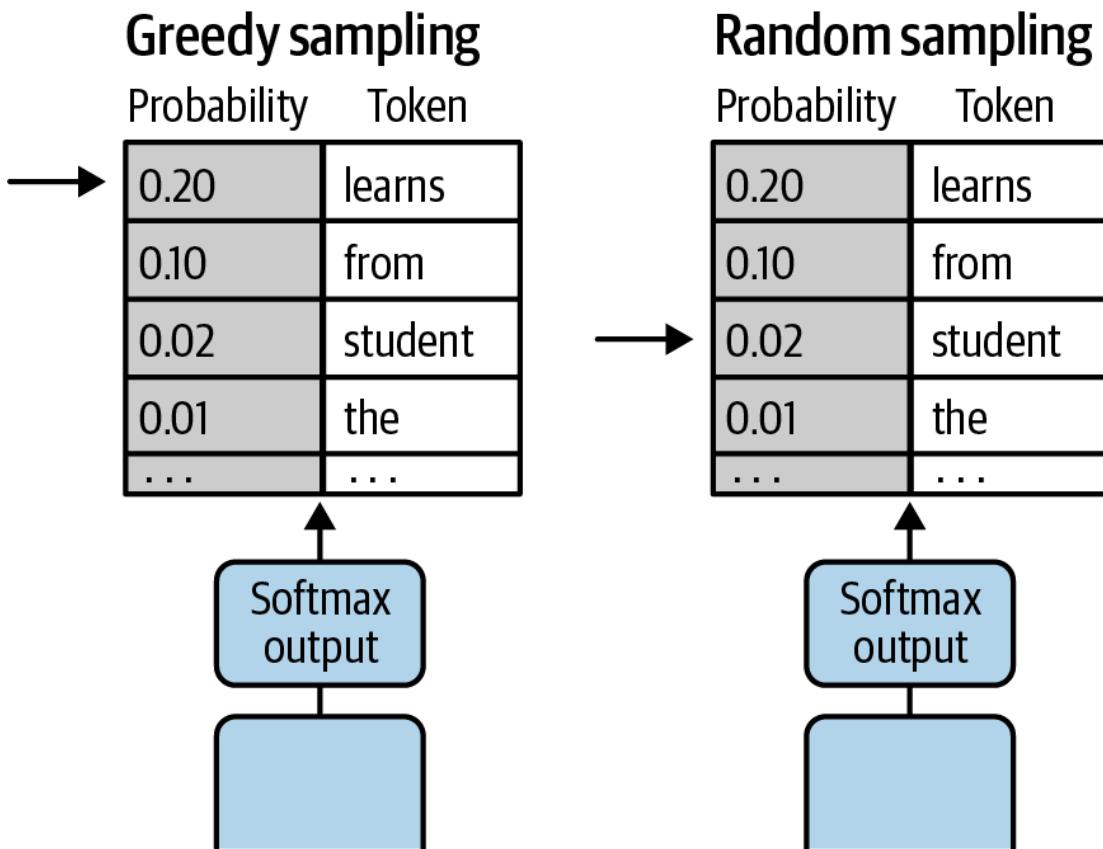


Figure 2-2. Greedy versus random sampling to predict the next token from a probability distribution

Using random sampling, you reduce the likelihood of repeated tokens in the model completion. The trade-off, however, is that the model output may be too creative and either generate an off-topic or unintelligible response. The challenge of finding this optimal setting is why this is called prompt engineering!

TIP

Some libraries like Hugging Face Transformers may require you to explicitly disable greedy sampling and manually enable random sampling using a function argument similar to `do_sample=True`.

top-p and top-k random sampling

These are the most common inference parameters when using random sampling. These parameters provide more fine-grained control for the random sample, which, if used properly, should improve the model’s response while allowing it to be creative enough to fulfill the generative task.

top-k, as you may have guessed, limits the model to choosing a token randomly from only the top- k tokens with the highest probability. For example, if k is set to 3, you are restricting the model to choose from only the top three tokens using the weighted random-sampling strategy. In this case, the model randomly chooses “from” as the next token, although it could have selected one of the other two, as shown in [Figure 2-3](#).

top-k random sampling

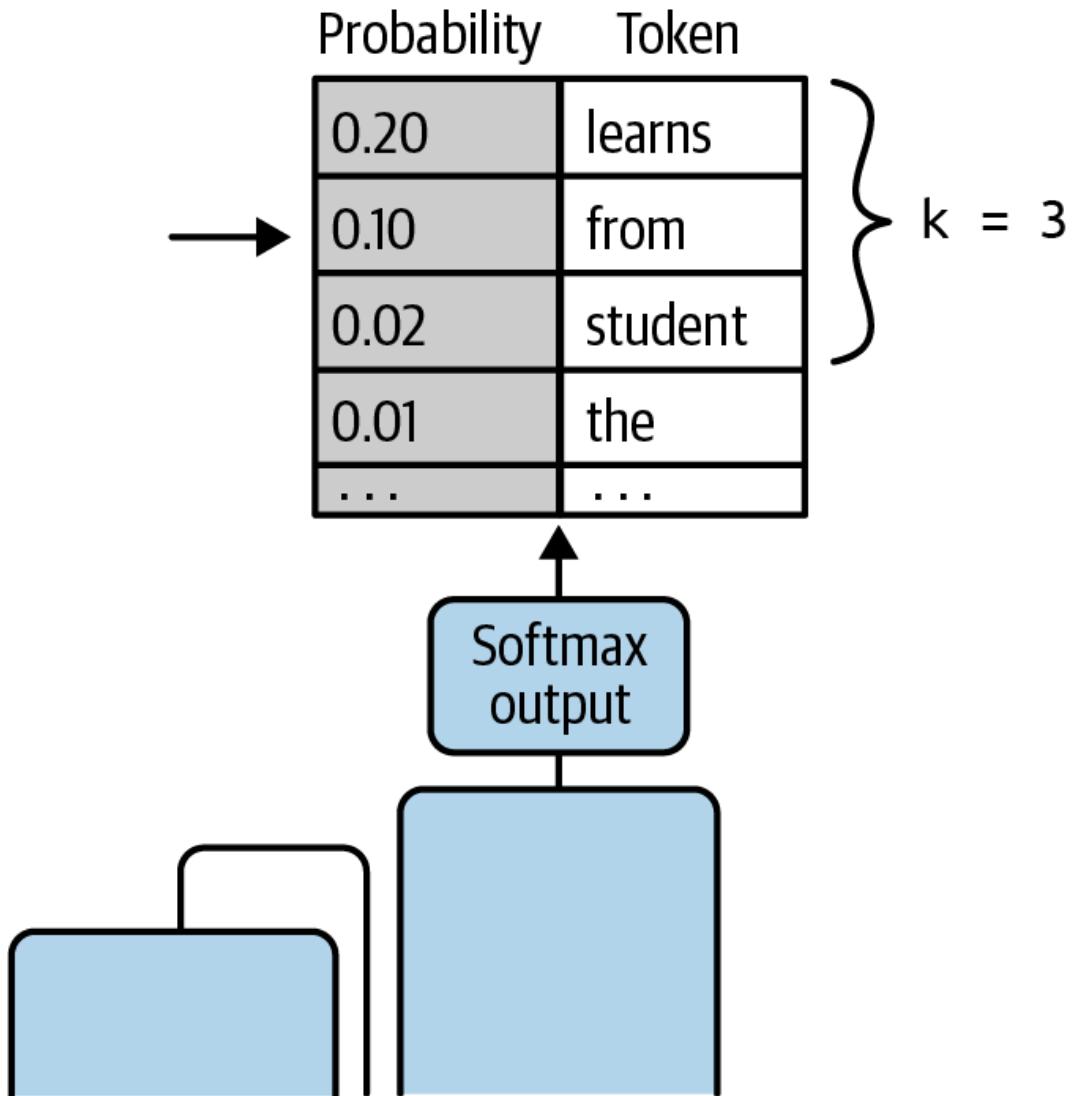


Figure 2-3. In this case, top-k sampling restricts the model to choosing from the top three probabilities

Note that setting top-k to a higher number can help reduce repetitiveness, while setting top-k to 1 basically gives you greedy decoding.

top-p limits the model to randomly sampling from the set of tokens whose cumulative probabilities do not exceed p, starting from the highest probability and working down to

the lowest probability. To illustrate this, first sort the tokens in descending order based on the probability. Then select a subset of tokens whose cumulative probability scores do not exceed p .

For example, if $p = 0.32$, the options are “learns”, “from”, and “student” since their probabilities of 0.20, 0.10, and 0.02, respectively, add up to 0.32. The model then uses the weighted random-sampling strategy to choose the next token —“student” in this case—from this subset of tokens, as shown in [Figure 2-4](#).

`top-p` can also produce greater variability and is sometimes used if it is hard to pick a good `top-k` value. `top-p` and `top-k` can also be used together.

top-p random sampling

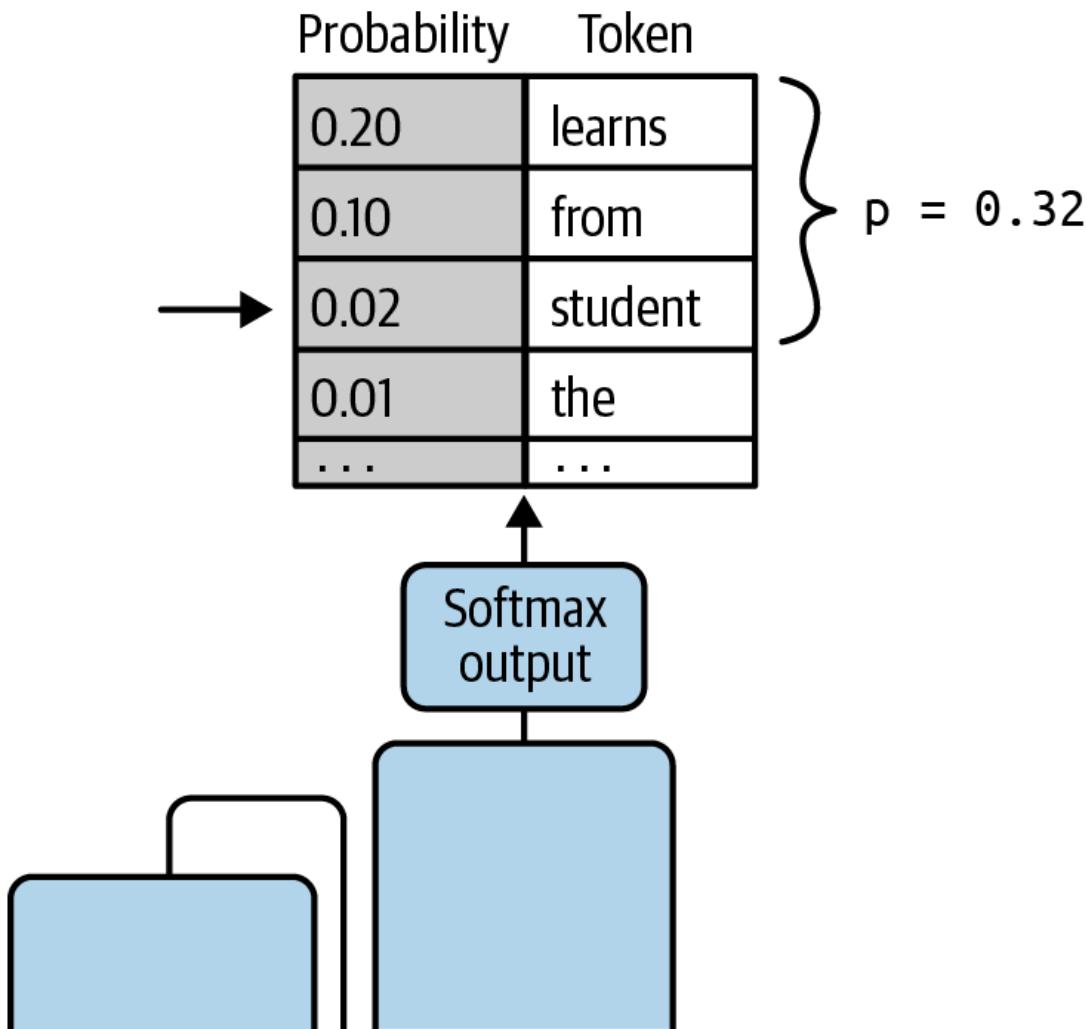


Figure 2-4. top-p random probability weighting

temperature

This parameter also helps to control the randomness of the model output by modifying the shape of the next-token probability distribution. In general, the higher the temperature, the higher the randomness; the lower the temperature, the lower the randomness.

In contrast to top-k and top-p, changing the temperature actually changes the next-token probability distribution,

which ultimately affects the next-token prediction.

A low temperature (below 1, for example) results in stronger peaks where the probabilities are concentrated among a smaller subset of tokens. A higher temperature (above 1, for example) results in a flatter next-token probability distribution where the probabilities are more evenly spread across the tokens. Setting the temperature to 1 leaves the next-token probability distribution unaltered, which represents the distribution learned during model training and tuning.

Figure 2-5 compares the low and high temperature scenarios.

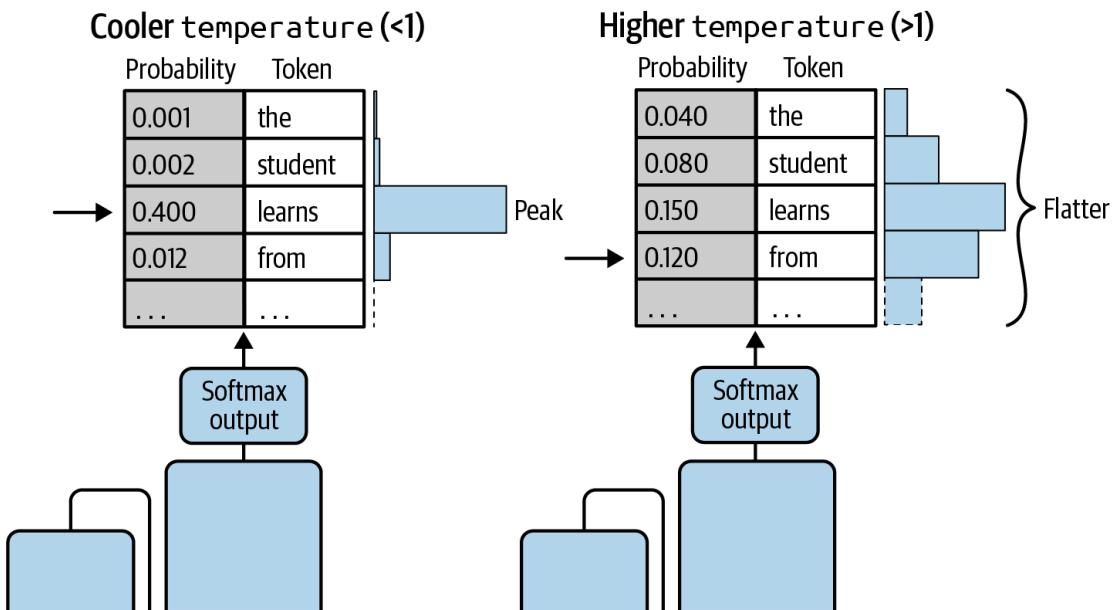


Figure 2-5. Changing the temperature will change the next-token probability distribution

In both cases, the model selects the next token from the modified probability distribution using either greedy or random sampling, which is orthogonal to the temperature parameter.

Note that if the temperature value is too low, the model may generate more repetitions; if the temperature is too high, the

model may generate nonsensical output. However, starting with a temperature value of 1 is usually a good strategy.

Summary

In this chapter, you learned techniques to help get the best possible performance from generative AI models using prompt engineering and by experimenting with different inference configuration parameters. Prompt engineering guides the generative foundation model to provide more relevant and accurate completions using various methods such as better-worded prompts, in-context learning examples, and step-by-step logical reasoning.

While you can get far with prompt engineering, in-context learning, and inference parameters, these techniques do not actually modify the generative models' weights. As such, you may need to train or fine-tune a generative model on your own datasets to help it better understand your specific domain and set of generative use cases, which you will explore in the next few chapters.

OceanofPDF.com

Chapter 3. Large-Language Foundation Models

In [Chapter 2](#), you learned how to perform prompt engineering and leverage in-context learning using an existing foundation model. In this chapter, you will explore how a foundation model is trained, including the training objectives and datasets. While it's not common to train your own foundation model from scratch, it is worth understanding how much time, effort, and complexity is required to perform this compute-intensive process.

Training a multibillion-parameter large-language model from scratch, called *pretraining*, requires millions of GPU compute hours, trillions of data tokens, and a lot of patience. In this chapter, you will learn about empirical scaling laws as described in the popular Chinchilla paper for model pretraining.¹

When training the BloombergGPT model, for example, researchers used the Chinchilla scaling laws as a starting point but still required a lot of trial and error, as explained in the BloombergGPT paper.² With a GPU compute budget of 1.3 million GPU hours, BloombergGPT was trained with a large distributed cluster of GPU instances using Amazon SageMaker.

NOTE

This chapter dives deep into pretraining generative foundation models, which may overwhelm some readers. It's important to note that you do not need to fully understand this chapter to effectively build generative AI applications. You may find this chapter useful as a reference for some advanced concepts later in this book.

Large-Language Foundation Models

At the start of any generative AI project, you should first explore the vast number of publicly available, pretrained foundation models that exist today, including the [Llama 2](#) model variants from Meta, which are used throughout this book. Many of these generative models have been trained on public data from the internet across many different languages and topics. As such, these models have built a solid understanding of human language as well as a massive amount of knowledge across many domains. This is often called *parametric memory*, as the knowledge is captured in the models' parameters.

You can find these foundation models in a model hub such as Hugging Face Model Hub, PyTorch Hub, or Amazon SageMaker JumpStart. Model hubs offer a model card for each model. Model cards typically contain important information about the model, including training details, context window size, prompt information, and known limitations.

For example, the Hugging Face Model Hub contains a [model card](#) for the 70 billion-parameter variant of Llama 2 from Meta. This model card includes useful details, including the context window length (4,096 tokens), the languages supported (English only, in this case), [sample code to construct the prompt](#), and any research papers³ associated with the model.

Often, the model hubs contain the same models. So just pick a model hub that best fits your security and infrastructure needs. For example, with the SageMaker JumpStart model hub, you can deploy a private copy of a foundation model directly into your AWS account with just a few clicks, as described in the Amazon SageMaker JumpStart [documentation](#). This lets you start generating new content within minutes!

Some models may use slight variations of the original Transformer architecture to optimize for specific language tasks. This may cause issues if you try to swap out models during development, so it's important to conduct enough research before you begin development to prevent this from happening.

TIP

Fear of missing out (FOMO) may tempt you to swap out a newer generative model before completing your evaluation of the current model. Try to avoid this temptation and complete your testing with a single model—or set of models—before chasing the latest and greatest leaderboard winner.

In your evaluations, you will notice that some pretrained foundation models may not have seen enough public text to learn the nuances of your specific domain. For example, the vocabulary of the public foundation models, often measured in tens of thousands or hundreds of thousands of tokens, may not include the terms commonly used by your business.

Additionally, public foundation models and datasets may have been scrubbed to avoid providing medical, legal, or financial advice due to the sensitive nature of these domains. To remedy this, one financial company, Bloomberg, chose to pretrain their own foundation model from scratch called BloombergGPT. BloombergGPT was trained with both public and private **financial data**, as shown in [Table 3-1](#).

Table 3-1. Breakdown of BloombergGPT training data

	Source	Approx. %
Financial data <i>(public and private)</i>	Web	42%
	News	5%
	Filings	2%
	Press	1%
Other data <i>(public)</i>	Bloomberg	1%
	TOTAL	51%
	The Pile	26%
	C4	20%
	Wikipedia	3%
	TOTAL	49%

Let's learn more about the fundamentals of large language models, starting with tokenizers that convert natural language text into word parts, or tokens, as you learned about in [Chapter 2](#).

Tokenizers

Every language-based generative AI model has a tokenizer that converts human-readable text (e.g., prompts) into a vector containing `token_ids` or `input_ids`. Each `input_id` represents a token in the model's vocabulary.

You will see `input_ids` in a lot of generative AI application source code, as these are the numeric representations of each token. A list of `input_ids` represents a larger piece of text like a phrase, sentence, or paragraph, as shown in [Figure 3-1](#) for the phrase, “The student learns from the”.

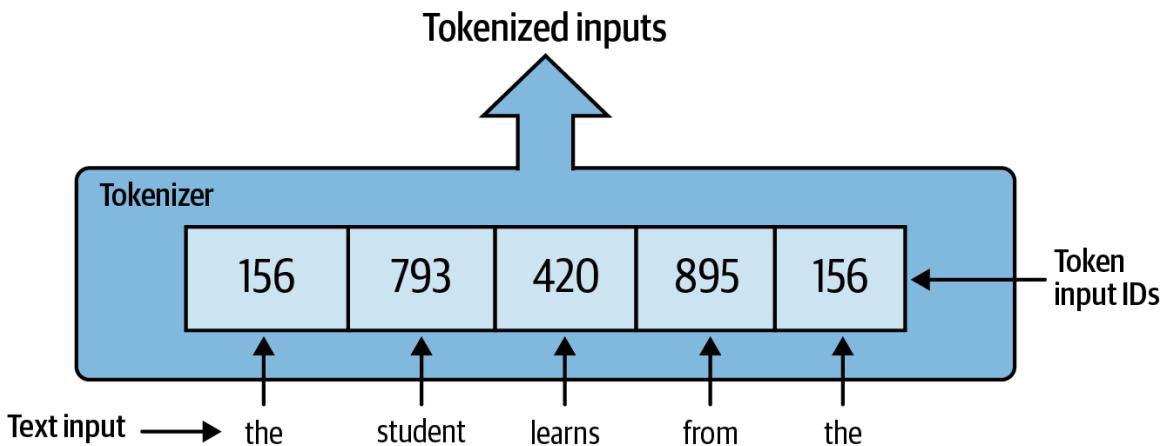


Figure 3-1. Use a tokenizer to convert text inputs into vectors for machine-readable processing

Once the model converts the input text into a vector of `input_ids`, it needs to perform one more step to retrieve each token’s high-dimensional representation, called an *embedding vector*, which is learned during the model pretraining phase. An embedding vector is a key component to language-based generative models.

Embedding Vectors

Embedding vectors, often called the “embeddings,” have been used in machine learning, information retrieval, and search use cases for decades. Embeddings are a numerical, vectorized representation of any entity of any type, including text, images, videos, and audio clips, projected into very high-dimensional vector spaces.

For simplicity, let’s use a simple three-dimensional vector space in which each embedding is a vector of three values projected in the three-dimensional space (shown in [Figure 3-2](#)). Here, you can see that tokens like “teach” and “book” are closely related, while other tokens like “car” and “fire” are farther apart.

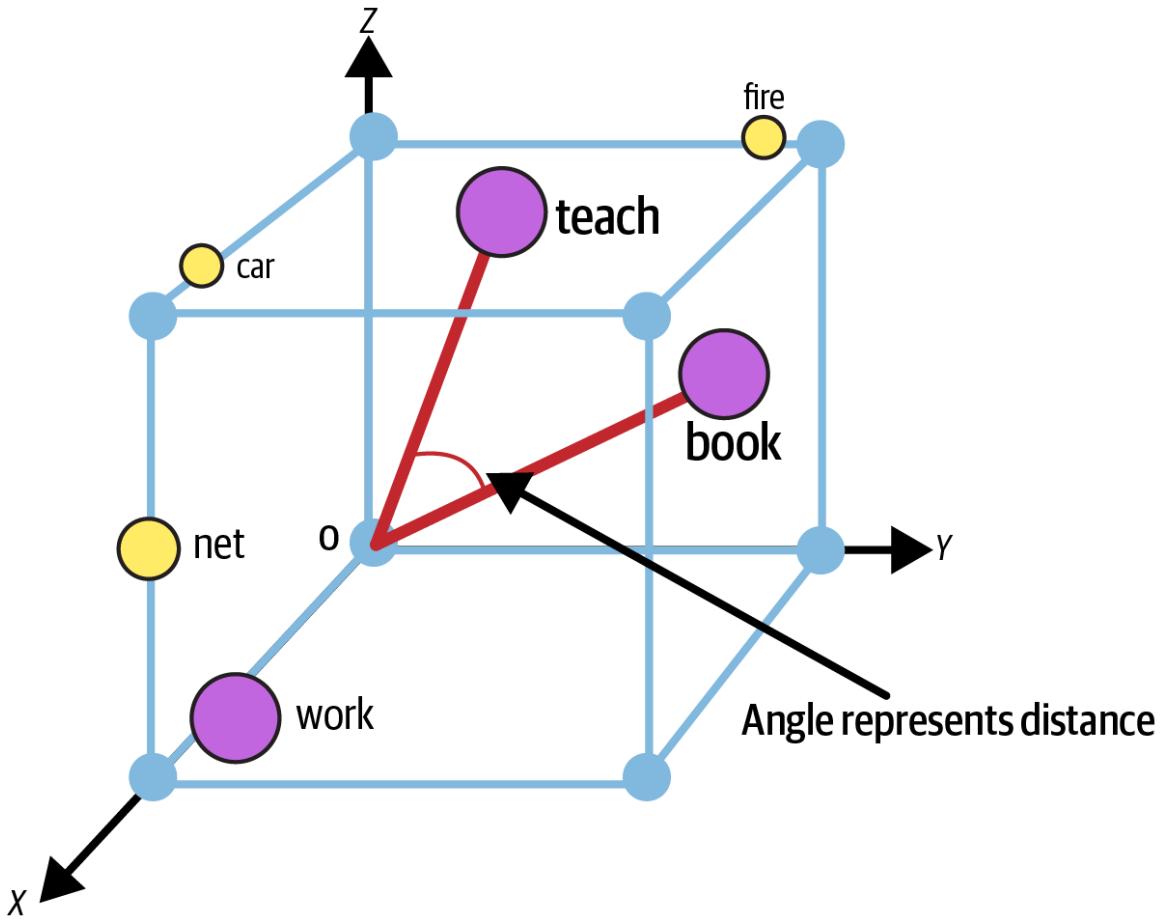


Figure 3-2. Representation of tokens in an example three-dimensional embedding space

Since these vectors encode the meaning and context of tokens within a larger corpus of text, they allow the model to statistically represent and understand human language. The closer these tokens are to each other in the vector space, the more similar they are in semantic meaning.

Figure 3-3 shows how each token in the phrase, “the student learns from the,” maps to a vector in a three-dimensional embedding space. While the examples here show only a few dimensions, a typical embedding space is often between 512 and 4,096 dimensions.

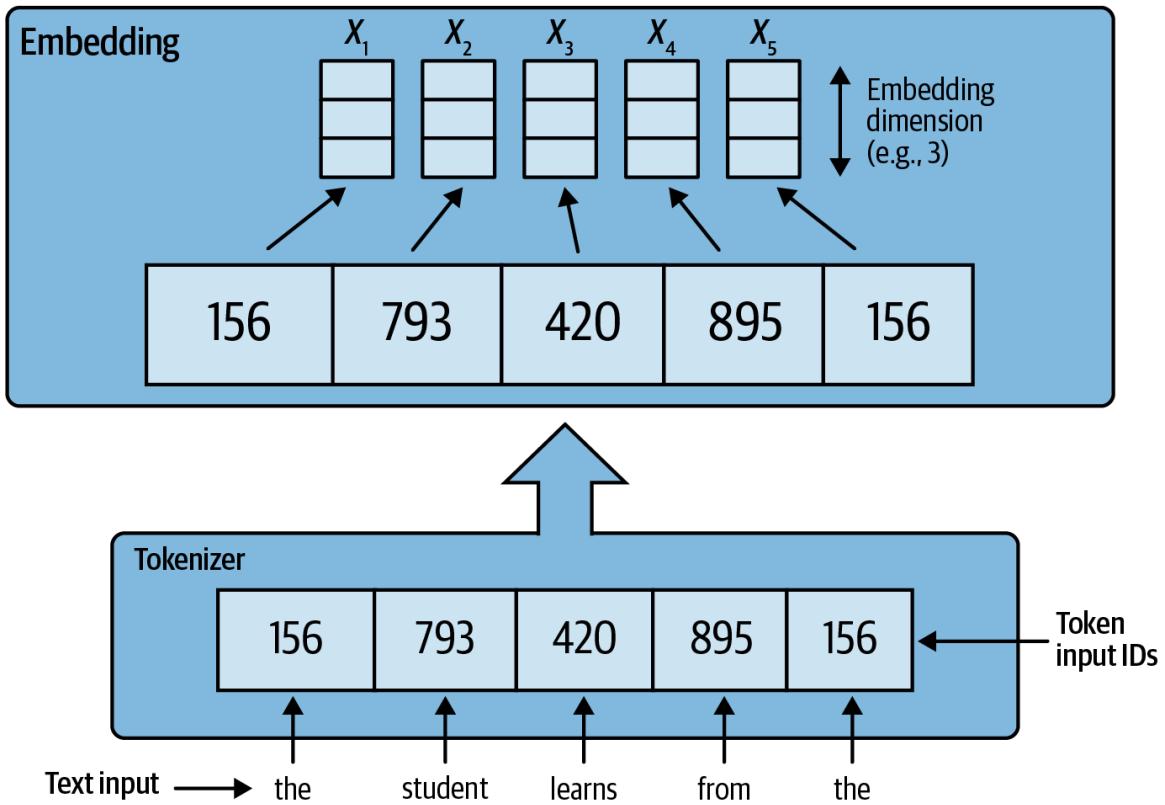


Figure 3-3. Embedding vector space of three dimensions

Now that you are familiar with embedding vectors, you are ready to explore the Transformer architecture. The embeddings are passed to the self-attention layers, which are a key component of the Transformer, as you will see next.

Transformer Architecture

Released in 2017, Transformers are at the core of most modern language models. In fact, the “T” in BERT and GPT, two popular language architectures, stands for Transformer. The Transformer serves a slightly different purpose depending on how it’s being used.

During model inference, as you saw in [Chapter 2](#), the Transformer is primarily focused on helping the model generate a completion to a given input prompt. During model pretraining and fine-tuning (Chapters [5](#), [6](#), and

7), the Transformer is helping the model gain contextual understanding of the language from the input training/tuning corpus.

TIP

It's important to remember that you don't need to understand the low-level details of the Transformer architecture to be successful with generative AI. While it always helps to understand your environment, the complex implementation details have been abstracted away into libraries such as the [Hugging Face Transformers Python library](#) used throughout the examples in this book.

Figure 3-4 includes a visual representation of the Transformer that we focus on in this book. Roughly from bottom to top, the input token context window contains the prompt-input tokens (e.g., max 4,096 input tokens), embeddings, encoder, self-attention layers, decoder, and the softmax output which helps the model choose the next token to generate from a probability distribution across the entire token vocabulary (e.g., 30,000–50,000 tokens). Next, let's walk through each of the components.

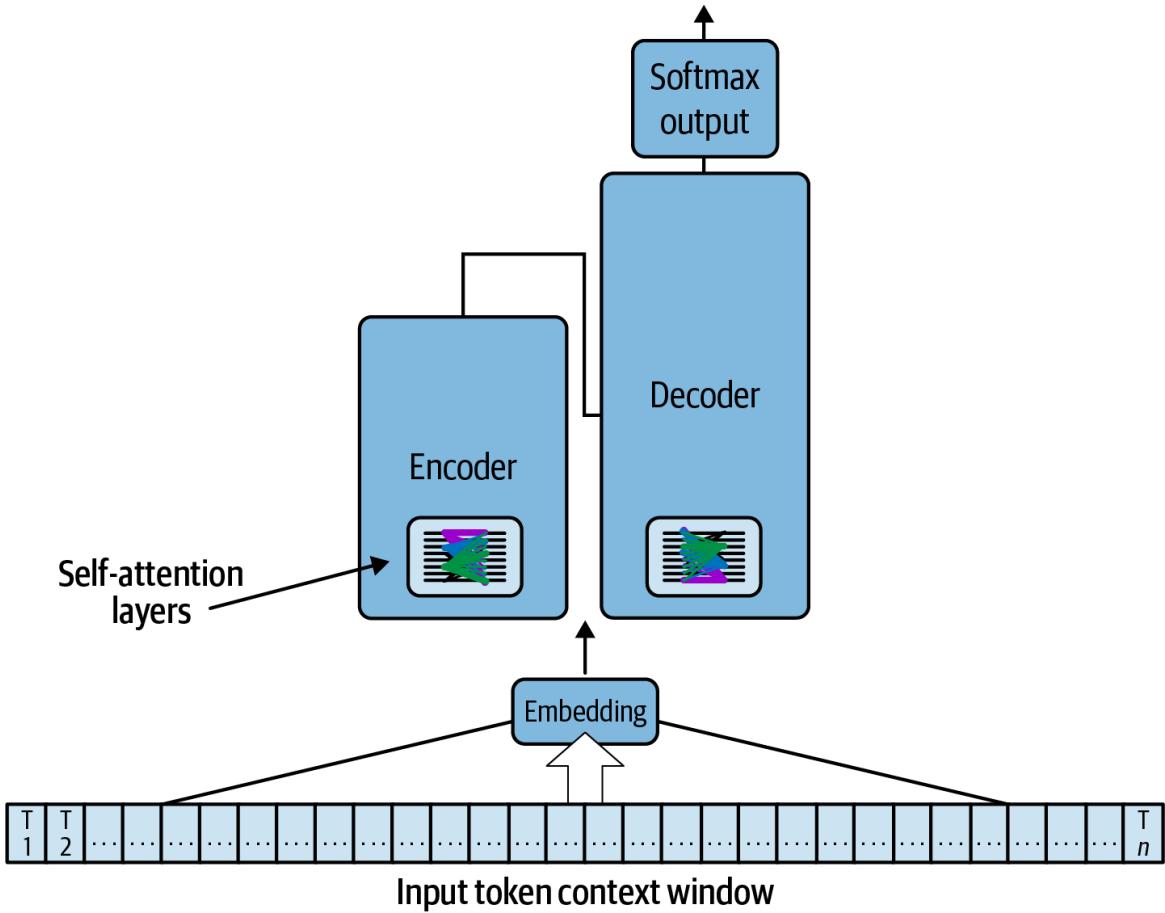


Figure 3-4. High-level Transformer architecture

Inputs and Context Window

The input prompt is stored in a construct called the input “context window.” It’s measured by the number of tokens it holds. The size of the context window varies widely from model to model. Earlier generative models could hold only 512–1,024 input tokens in the context window. However, more recent models can hold upwards of 10,000 and even 100,000 tokens (at the time of this writing). The model’s input context window size is defined during model design and pretraining.

Embedding Layer

You learned about embeddings previously; however, it’s worth reminding you that they are learned during model pretraining and are actually part of

the larger Transformer architecture. Each input token in the input context window is mapped to an embedding. These embeddings are used throughout the rest of the Transformer neural network, including the self-attention layers.

Encoder

The encoder, at a high level, encodes—or projects—sequences of input tokens into a vector space that represents the structure and meaning of the input. The vector space representation is learned during model pretraining.

Self-Attention

The Transformer architecture uses a mechanism called self-attention to “pay attention” to interesting tokens as it traverses the inputs. Specifically, self-attention is used to attend every token in the input data to all other tokens in the input sequence. An example of self-attention is shown in [Figure 3-5](#), where the word “her” attends highly to the word “professor” as well as the word “lectures,” though to a lesser degree than the word “professor.”

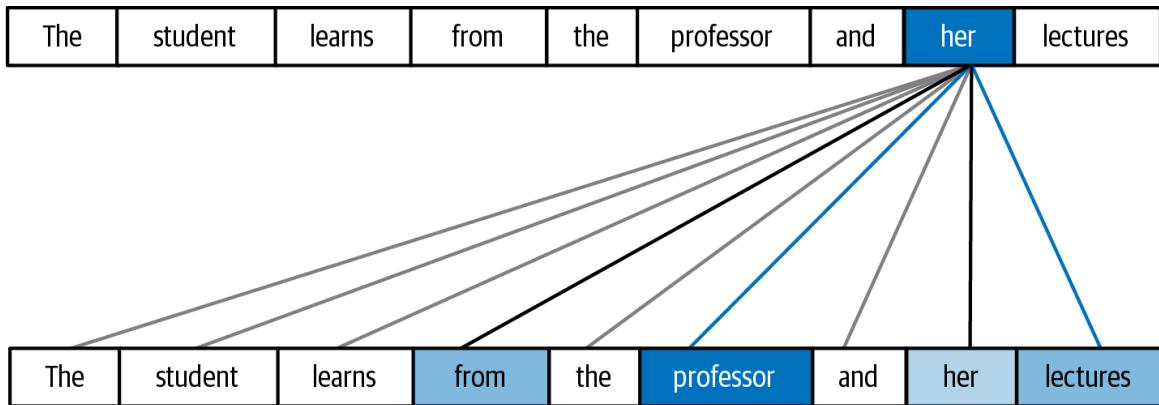


Figure 3-5. The “self-attention” mechanism attends every token in the data to all other tokens in the input sequence

This pairwise attention lets the model learn the contextual dependencies, or contextual understanding, of the input data during model pretraining. By paying attention to the whole input, the Transformer unlocks the model’s ability to learn and represent language from the training documents provided.

In practice, the Transformer actually learns multiple sets of self-attention weights through *multiheaded attention*. Each head runs in parallel over the same input and learns different aspects of the language. For example, one head may attend to the relationships between entities in the input while another head attends to a specific set of activities described in the input.

Note that the parameters, or weights, of each head are initialized randomly at the start, so it's difficult to predict which aspects each head will attend to. The number of heads varies from model to model, but it is typically in the range of 12–100 heads.

NOTE

Self-attention is very computationally expensive as it calculates n^2 pairwise attention scores between every token in the input with every other token. In fact, a lot of generative performance improvements are targeted at the attention layers such as FlashAttention and grouped-query attention (GQA) described in [Chapter 4](#).

Let's have a closer look at how the Transformer implements the self-attention mechanism. Attention assigns a weight to the input tokens based on their importance relative to the generative task. Consider attention as a function that takes input sequence X and returns output sequence Y , where X and Y are the same length vector. Each vector in Y is a weighted average of the vectors in X , as shown in [Figure 3-6](#).

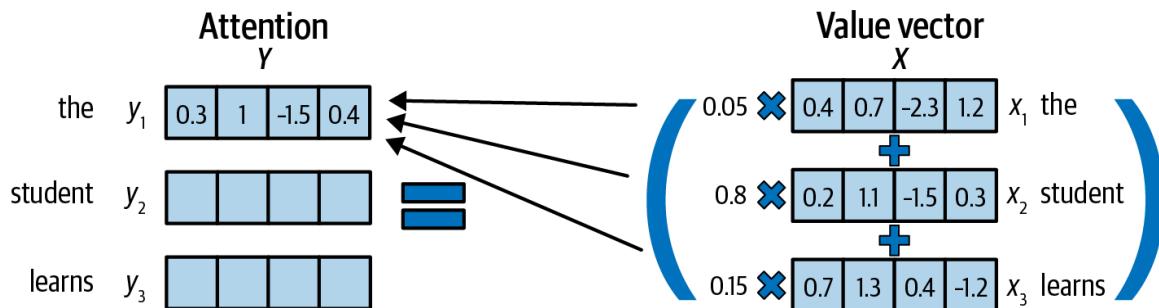


Figure 3-6. Attention is the weighted average of the input vectors

These weights express how much the model is attending to each input vector in X when computing the weighted average. To calculate the

attention weights, a compatibility function assigns a score to each pair of words indicating how compatible they are—or rather, how strongly they attend to each other. Let's dive deeper into the compatibility function and score shown in [Figure 3-7](#).

	Query	Key		Compatibility score	After softmax							
movie	<table border="1"><tr><td>-0.2</td><td>1.4</td><td>0.4</td></tr></table>	-0.2	1.4	0.4	<table border="1"><tr><td>0.6</td><td>-0.1</td><td>1.7</td></tr></table>	0.6	-0.1	1.7	love		<table border="1"><tr><td>0.42</td></tr></table>	0.42
-0.2	1.4	0.4										
0.6	-0.1	1.7										
0.42												
movie	<table border="1"><tr><td>-0.2</td><td>1.4</td><td>0.4</td></tr></table>	-0.2	1.4	0.4	<table border="1"><tr><td>0.9</td><td>1.3</td><td>1.7</td></tr></table>	0.9	1.3	1.7	this		<table border="1"><tr><td>2.32</td></tr></table>	2.32
-0.2	1.4	0.4										
0.9	1.3	1.7										
2.32												
movie	<table border="1"><tr><td>-0.2</td><td>1.4</td><td>0.4</td></tr></table>	-0.2	1.4	0.4	<table border="1"><tr><td>-0.9</td><td>0.1</td><td>1.6</td></tr></table>	-0.9	0.1	1.6	movie		<table border="1"><tr><td>0.96</td></tr></table>	0.96
-0.2	1.4	0.4										
-0.9	0.1	1.6										
0.96												

Figure 3-7. Attention weights are the normalized dot product of the query and key vectors

First, the model creates a query (q) vector for the word that is paying attention and a key (k) vector for the word being paid attention to. These are linear transformations (matrix multiplications) of the original input vectors with learned weight matrices for each query and key vector.

Next, the compatibility score is calculated as the dot product of the query vector of one word and the key vector of the other. Last, the score is then normalized by applying the softmax function. The result is the attention weight after the softmax is applied.

Decoder

The attention weights are passed through the rest of the Transformer neural network, including the decoder. The decoder uses the attention-based contextual understanding of the input tokens to help generate new tokens, which ultimately “completes” the provided input. This is why the model’s response is often called a completion.

Softmax Output

The softmax output layer generates a probability distribution across the entire token vocabulary in which each token is assigned a probability that it

will be selected next. Typically, the token with the highest probability will be generated as the next token, but as you saw in [Chapter 2](#), there are mechanisms like `temperature` to modify next-token selection to make the model more or less creative, for example.

The softmax layer produces a vector of probabilities that represent each token's likelihood of being chosen next. In other words, if the vocabulary is 100,000 tokens, this layer produces a vector of 100,000 probabilities, as shown in [Figure 3-8](#).

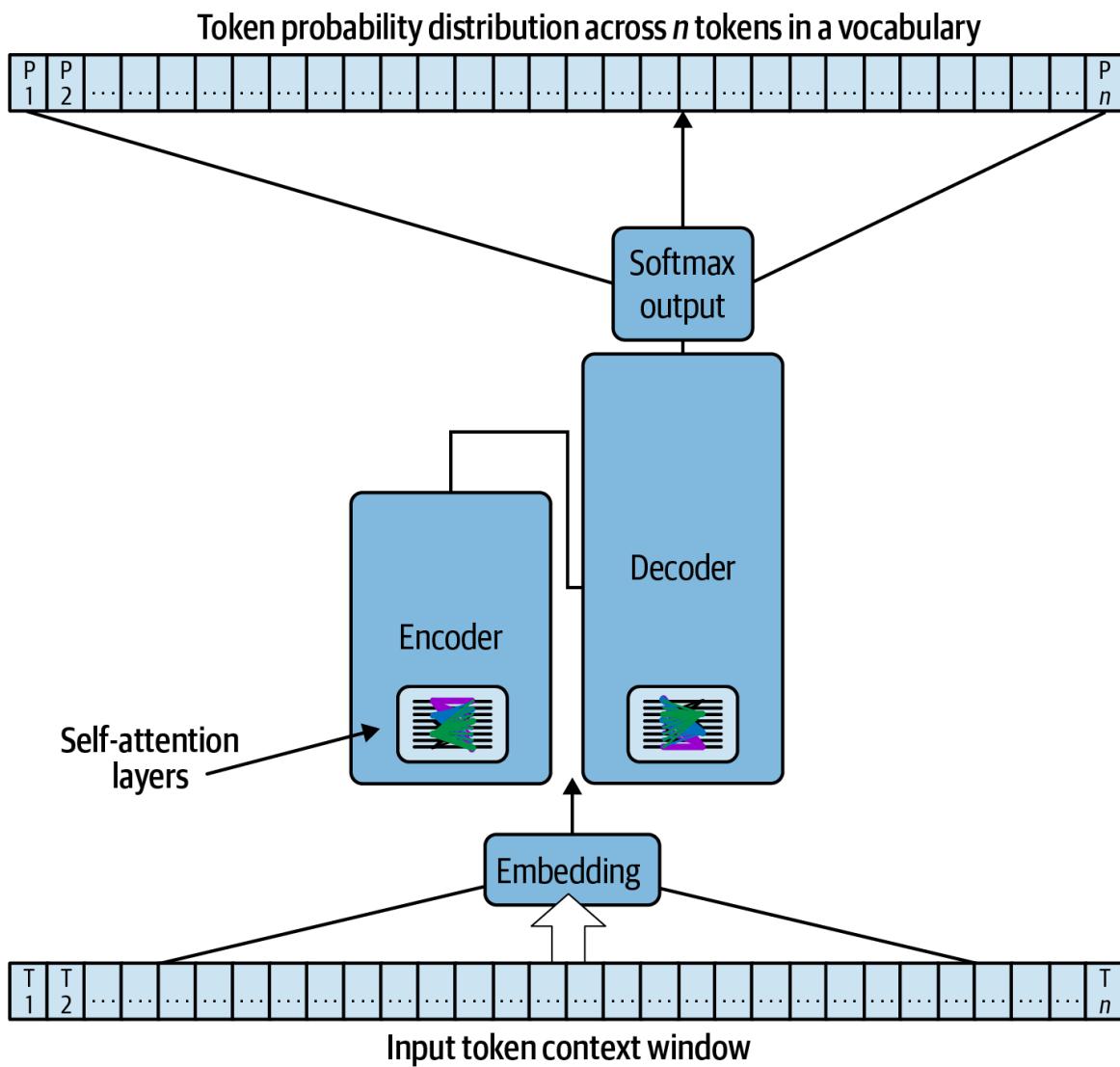


Figure 3-8. Probability of being the next token across all tokens in the vocabulary

The model continues to generate new tokens in a loop until a stop condition is reached—typically when an end-of-sequence (EOS) token is generated. Similar to the token vocabulary and input context window size, the EOS token is often model-specific and should be defined by the model creator.

At this point, you've learned the key components of the Transformer architecture. This sets the foundation for the rest of the book, including the chapters on multimodal generative models. The Transformer is a key component to almost all generative models since the primary way to interact with these models is through language.

Types of Transformer-Based Foundation Models

There are three variants of generative transformer-based models overall: encoder-only, decoder-only, and encoder-decoder. Each variant is trained with a different training objective and, during pretraining, the model weights are updated to minimize the loss of the training objectives described next for each variation. Each variant is capable of addressing different types of generative tasks, as you will see next.

Encoder-only models, or autoencoders, are pretrained using a technique called masked language modeling (MLM), which randomly mask input tokens and try to predict the masked tokens. This is sometimes called a *denoising* objective. Autoencoding models use bidirectional representations of the input to better understand the full context of a token—not just the previous tokens in the sequence, as shown in [Figure 3-9](#).

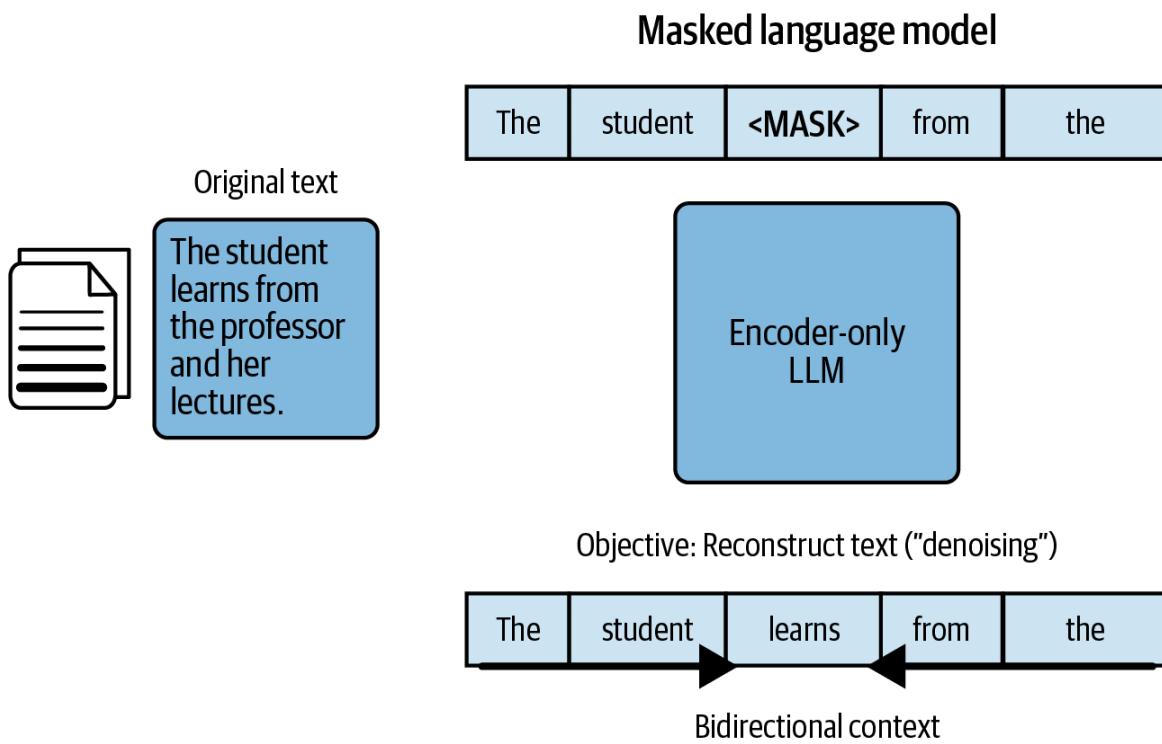


Figure 3-9. Encoder-only (autoencoder) models use a bidirectional context to reconstruct the masked input tokens

Encoder-only models are best suited for language tasks that utilize the embeddings generated by the encoder, such as text classification. They are not particularly useful for generative tasks that continue to generate more text. A well-known encoder-only model is BERT, which is covered extensively in *Data Science on AWS* (O'Reilly).

The embedding outputs are also useful for semantic similarity search—an advanced document-search algorithm beyond simple keyword search. You will explore semantic similarity search more in [“Retrieval-Augmented Generation”](#).

Decoder-only models, or autoregressive models, are pretrained using unidirectional causal language modeling (CLM), which predicts the next token using only the previous tokens—every other token is masked, as shown in [Figure 3-10](#).

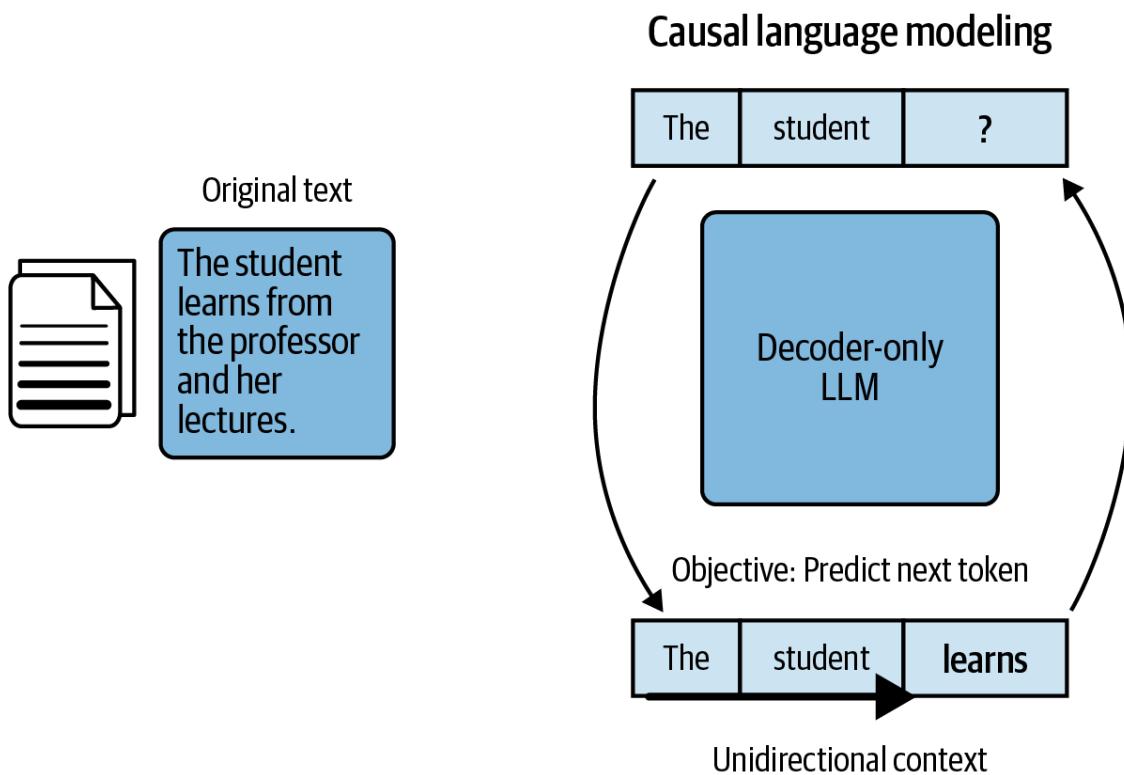


Figure 3-10. Decoder-only (autoregressive) models only reveal the tokens leading up to the token being predicted

Decoder-only, autoregressive models use millions of text examples to learn a statistical language representation by continuously predicting the next token from the previous tokens. These models are the standard for generative tasks, including question-answer. The families of GPT-3, Falcon, and LLaMA models are well-known autoregressive models.

NOTE

In case you're wondering, Meta changed the case of the Llama model name when they released Llama 2. The first version uses mixed case (LLaMA), which is an acronym for Large Language Model Meta AI. The second version uses title case (Llama 2).

Encoder-decoder models, often called sequence-to-sequence models, use both the Transformer encoder and decoder. While the pretraining objectives vary from model to model, the popular T5 foundation model (e.g., FLAN-T5) was pretrained using consecutive multitoken masking called span

corruption. The decoder then attempts to reconstruct the masked sequence of tokens, <X>, as shown in [Figure 3-11](#).

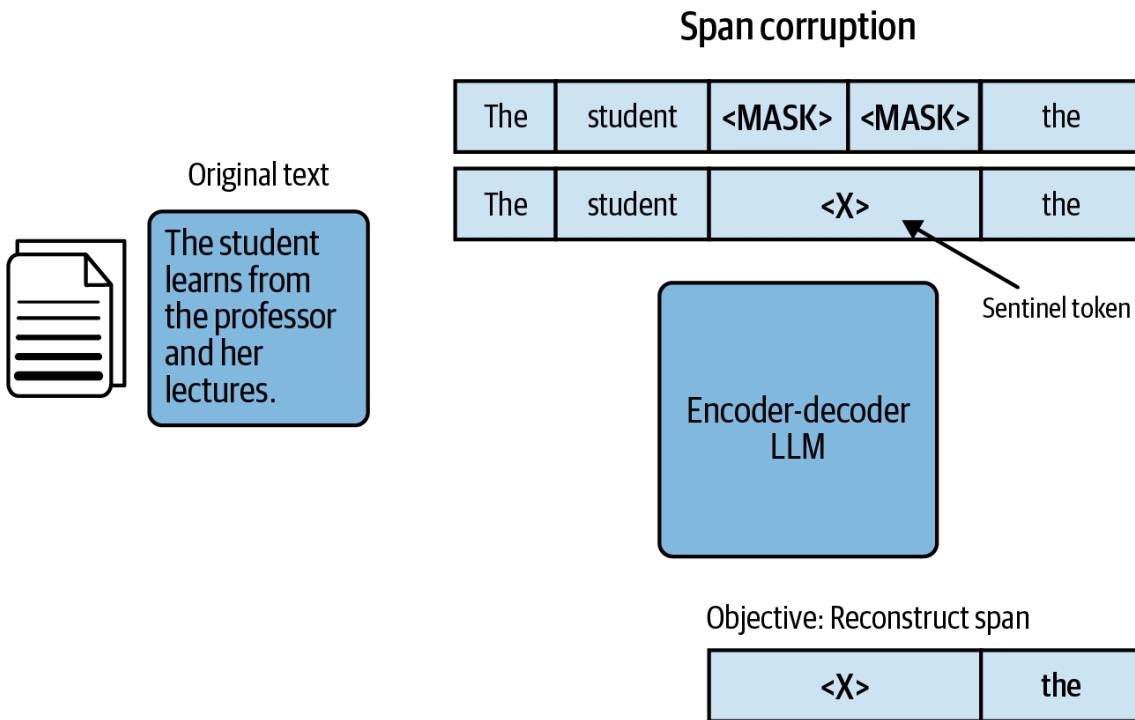


Figure 3-11. Encoder-decoder (sequence-to-sequence) models

Sequence-to-sequence models, originally designed for translation, are also very useful for text-summarization tasks. [T5](#) and its fine-tuned sibling, [FLAN-T5](#), are well-known encoder-decoder, sequence-to-sequence models used across a wide number of generative language tasks.

Now that you've seen the three main types of transformer-based foundation models, let's explore some of the most common publicly available datasets used to pretrain foundation models.

Pretraining Datasets

A generative model learns its capabilities during the pretraining phase when it sees a large amount of training data—often on the scale of terabytes and petabytes. The datasets are often sourced from the public internet but can also include proprietary data from your private Amazon S3 buckets or databases.

Two of the most popular datasets to pretrain large language models are Wikipedia and Common Crawl. Wikipedia offers a multilingual extract of its contents from 2022, while Common Crawl is a monthly dump of text found on the whole of the internet.

As you can imagine, this type of free-form internet data is very messy. As such, there are variants of these datasets, such as Wiki-40B,⁴ Colossal Clean Crawled Corpus (C4),⁵ The Pile,⁶ and RefinedWeb,⁷ that attempt to clean the data for higher-quality model training. RefinedWeb, in particular, attempts to filter out machine-generated text using statistical methods to determine if the text is human-generated versus machine-generated.

NOTE

The Falcon family of models was trained on 1.5 trillion tokens of data called RefinedWeb. The data was processed on a cluster of 257 `ml.c5.18xlarge` SageMaker instances consisting of 18,504 CPUs and 37TB of CPU RAM.

Next, you'll learn about scaling laws, which describe the relationship between model size, dataset size, and compute budget.

Scaling Laws

For generative models, a set of *scaling laws* have emerged that describe the trade-offs between model size and dataset size for a fixed compute budget (e.g., number of GPU hours). These scaling laws⁸ state that you can achieve better generative model performance by either increasing the number of tokens or the number of model parameters.

Scaling up both will typically require a higher compute budget, which is typically defined in terms of floating point operations per second (FLOPs). Figure 3-12 is a comparison of compute budgets required to pretrain different variations and sizes of BERT, T5, and GPT-3. Remember that

BERT is an encoder-only model, T5 is an encoder-decoder model, and GPT-3 is a decoder-only model. Note that the y-axis is logarithmic.

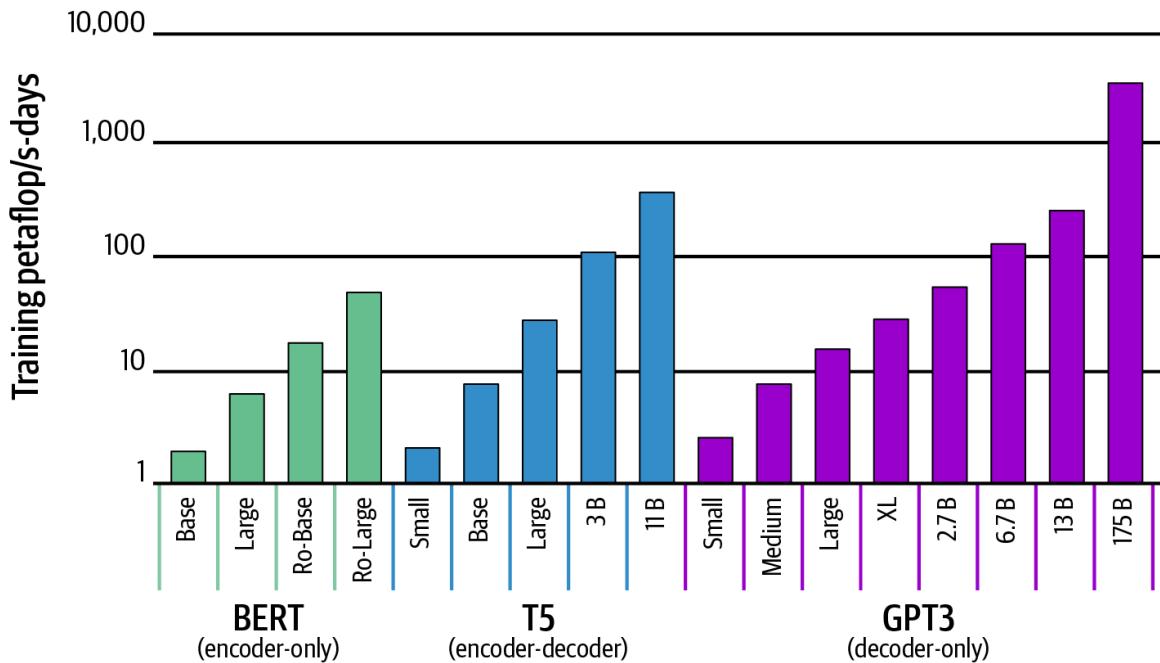


Figure 3-12. Pretraining requirements for common models in petaflop/s-days (source: adapted from an image in [Brown et al.](#))

While the 175 billion GPT-3 model outperforms the T5 and BERT models on generative tasks, according to various benchmarks, the larger models require a larger compute budget. You might wonder if it's possible to get 175 billion-parameter performance from a smaller model. In fact, you can!

Researchers have found that by increasing the training dataset size instead of the model size, you can get state-of-the-art performance that exceeds the 175 billion-parameter models with a much smaller set of weights. In fact, the “[Scaling Laws for Neural Language Models](#)” paper shows that if you hold the compute budget constant, model performance may increase when you either increase the training dataset size (and hold model parameter size constant) or increase the number of model parameters (and hold the dataset size constant). See [Figure 3-13](#) to see how the loss decreases as either dataset size or parameter size increases.

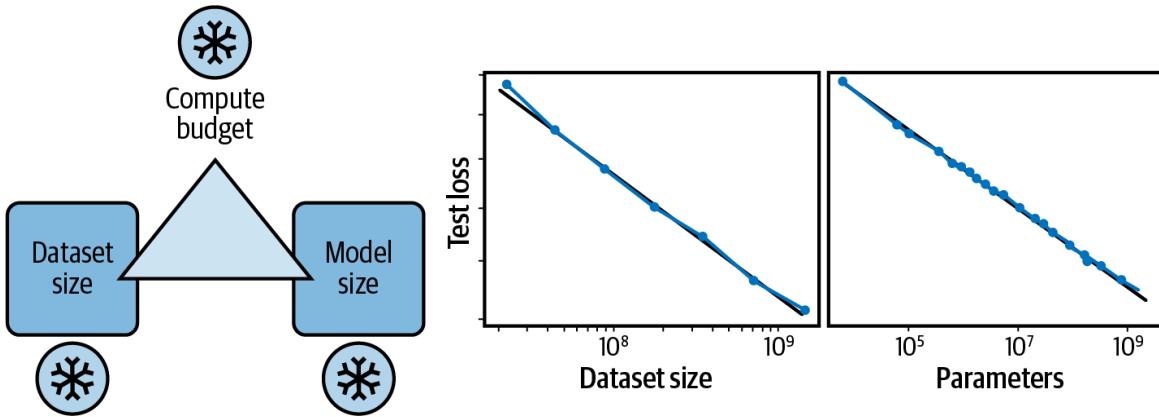


Figure 3-13. Impact of dataset size and parameter size on model performance (source: charts in figure adapted from an image in Kaplan et al.)

This also hints that you can improve performance for smaller models by just training them on more data. This is the exciting field of compute-optimal model research that you will learn about next.

Compute-Optimal Models

In 2022, a group of researchers released a paper⁹ that compared model performance of various model and dataset size combinations. Since the authors named their final compute-optimal model Chinchilla, this paper is famously called the Chinchilla paper.

The Chinchilla paper implies that the massive 100 billion-plus parameter models like GPT-3 may be overparameterized and undertrained. Additionally, they hypothesize that you could achieve 100 billion-plus parameter model performance with a small model by simply providing more training data to the smaller model.

To be more specific, the authors of the Chinchilla paper claim that the optimal training dataset size (measured in tokens) is 20x the number of model parameters and that anything below that 20x ratio is potentially overparameterized and undertrained. **Table 3-2** compares the compute-optimal Chinchilla and LLaMA models with the 175 billion-parameter variants of GPT-3, OPT, and BLOOM.

Table 3-2. Chinchilla scaling laws for given model size and dataset size

Model	Model size (parameters)	Optimal dataset size (tokens)	Actual dataset size (tokens)	Hypothesis
Chinchilla	70 B	1.4 T	1.4 T	Compliant (20x)
LLaMA-65B	65 B	1.3 T	1.4 T	Compliant (20x)
GPT-3	175 B	3.5 T	300 B	Overfitted for data (<20x)
OPT-175B	175 B	3.5 T	180 B	Overfitted for data (<20x)
BLOOM	176 B	3.5 T	350 B	Overfitted for data (<20x)
Llama2-70B	70 B	1.4 T	2.0 T	Better compression (>20x)

Here, you see that according to the Chinchilla scaling laws, these 175+ billion-parameter models should be trained on 3.5 trillion tokens. Instead, they were trained with 180–350 billion tokens—an order of magnitude smaller than recommended. As such, the paper hints that these 175+ billion-parameter models could have been trained with much more data—or could have been an order-of-magnitude smaller.

In fact, the more recent [Llama 2](#) 70 billion-parameter model, which was released after the Chinchilla paper, was trained with 2 trillion tokens—greater than the 20-to-1 token-to-parameter ratio described by the paper. Llama 2 outperformed the original LLaMA model based on various benchmarks,¹⁰ including Massive Multitask Language Understanding (MMLU). This demonstrates the recent trend to increase the amount of pretraining data while keeping the number of parameters relatively fixed.

Summary

In this chapter, you saw how foundation models are trained using vast amounts of text during their initial training phase, called pretraining. This is where the model develops its understanding of language.

You also learned three different types of transformer-based language models: encoder-only (autoencoding), decoder-only (autoregressive), and encoder-decoder (sequence-to-sequence).

Additionally, you learned some empirical scaling laws that have been discovered for pretraining generative AI models. These scaling laws help researchers choose the number of model parameters (1 billion, 7 billion, 70 billion, etc.) and dataset size (700 billion tokens, 1.4 trillion tokens, 2 trillion tokens, etc.) for a given compute budget when pretraining a foundation model from scratch.

You also saw how adding more training data—beyond the 20x ratio defined by the Chinchilla scaling laws—can improve model performance while keeping the model size relatively fixed.

Remember that pretraining a foundation model is not common, as it requires a large amount of GPU compute hours and data. It is more common that you would fine-tune your model on your dataset using a much smaller GPU compute cluster, as you will see in the coming chapters. However, before we explore fine-tuning, let's better understand the computational and memory challenges of working with large generative

models. Such challenges include GPU memory limitations and distributed-computing overhead.

In [Chapter 4](#), you will learn how to use quantization to reduce the memory requirements of your training job. You will also learn how to efficiently scale model training across multiple GPUs using distributed computing strategies such as fully sharded data parallel (FSDP), including optimizations for AWS.

- ¹ Jordan Hoffmann et al., “Training Compute-Optimal Large Language Models”, *arXiv*, 2022.
- ² Shijie Wu et al., “BloombergGPT: A Large Language Model for Finance”, *arXiv*, 2023.
- ³ Hugo Touvron et al., “Llama 2: Open Foundation and Fine-Tuned Chat Models”, *arXiv*, 2023.
- ⁴ Mandy Guo et al., “Wiki-40B: Multilingual Language Model Dataset”, *arXiv*, 2020.
- ⁵ Colin Raffel et al., “Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer”, *arXiv*, 2020.
- ⁶ Leo Gao et al., “The Pile: An 800GB Dataset of Diverse Text for Language Modeling”, *arXiv*, 2020.
- ⁷ Guilherme Penedo et al., “The RefinedWeb Dataset for Falcon LLM: Outperforming Curated Corpora with Web Data, and Web Data Only”, *arXiv*, 2023.
- ⁸ Jared Kaplan et al., “Scaling Laws for Neural Language Models”, *arXiv*, 2020.
- ⁹ Jordan Hoffman et al., “Training Compute-Optimal Large Language Models”, *arXiv*, 2022.
- ¹⁰ Dan Hendrycks et al., “Measuring Massive Multitask Language Understanding”, *arXiv*, 2021.

Chapter 4. Memory and Compute Optimizations

In [Chapter 3](#), you explored best practices for experimenting with and selecting a foundation model for your use case. The next step is usually to customize the model to your specific needs and datasets. This could include adapting the model to your datasets using a technique called *fine-tuning*, which you will explore in more detail in [Chapter 5](#). When training or fine-tuning large foundation models, you often face compute challenges—in particular, how to fit large models into GPU memory.

In this chapter, you will explore techniques that help overcome memory limitations. You will learn how to apply quantization and distributed training to minimize the required GPU RAM, and how to scale model training horizontally across multiple GPUs for larger models.

For example, the original 40 billion-parameter [Falcon model](#) was trained on a cluster of 48 `ml.p4d.24xlarge` Amazon SageMaker instances consisting of 384 NVIDIA A100 GPUs, 15TB of GPU RAM, and 55TB of CPU RAM. A more recent version of Falcon was trained on a cluster of 392 `ml.p4d.24xlarge` SageMaker instances consisting of 3,136 NVIDIA A100 GPUs, 125TB of GPU RAM, and 450TB of CPU RAM. The size and complexity of the Falcon model requires a cluster of GPUs, but also benefits from quantization, as you will see next.

Memory Challenges

One of the most common issues you'll encounter when you try to train or fine-tune foundation models is running out of memory. If you've ever tried training or even just loading your model on NVIDIA GPUs, the error message in [Figure 4-1](#) might look familiar.

OutOfMemoryError: CUDA out of memory.

Figure 4-1. CUDA out-of-memory error

CUDA, short for Compute Unified Device Architecture, is a collection of libraries and tools developed for NVIDIA GPUs to boost performance on common deep-learning operations, including matrix multiplication, among many others. Deep-learning libraries such as PyTorch and TensorFlow use CUDA extensively to handle the low-level, hardware-specific details, including data movement between CPU and GPU memory. As modern generative models contain multiple billions of parameters, you have likely encountered this out-of-memory error during development while loading and testing a model in your research environment.

A single-model parameter, at full 32-bit precision, is represented by 4 bytes. Therefore, a 1-billion-parameter model requires 4 GB of GPU RAM just to load the model into GPU RAM at full precision. If you want to also train the model, you need more GPU memory to store the states of the numerical optimizer, gradients, and activations, as well as any temporary variables used by your functions, as shown in [Table 4-1](#).

Table 4-1. Additional RAM needed to train a model

States	Bytes per parameter
Model parameters (weights)	4 bytes per parameter
Adam optimizer (2 states)	8 bytes per parameter
Gradients	4 bytes per parameter
Activations and temp memory (variable size)	8 bytes per parameter (high-end estimate)
TOTAL	= 4 + 20 bytes per parameter

TIP

When you experiment with training a model, it's recommended that you start with `batch_size=1` to find the memory boundaries of the model with just a single training example. You can then incrementally increase the batch size until you hit the CUDA out-of-memory error. This will determine the maximum batch size for the model and dataset. A larger batch size can often speed up your model training.

These additional components lead to approximately 12–20 extra bytes of GPU memory per model parameter. For example, to train a 1-billion-parameter model, you will need approximately 24 GB of GPU RAM at 32-bit full precision, six times the memory compared to just 4 GB of GPU RAM for loading the model, as shown in [Figure 4-2](#).

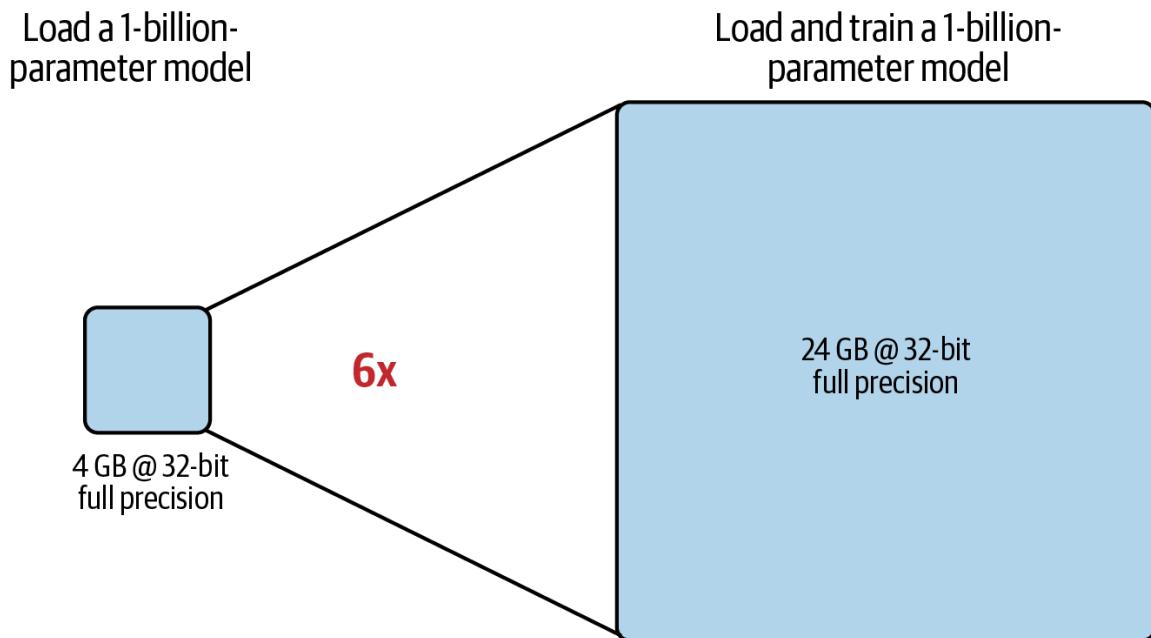


Figure 4-2. Comparison of approximate GPU RAM needed to load versus load and train a 1-billion-parameter model at 32-bit full precision

It's worth noting that the NVIDIA A100 and H100, used at the time of this writing, only support up to 80 GB of GPU RAM. And since you likely want to train models larger than 1 billion parameters, you'll need to find a workaround, such as quantizing your model.

AWS has also developed purpose-built ML accelerators, AWS Trainium, for high-performance and cost-efficient training of 100B+ parameter generative AI models. You can leverage AWS Trainium chips through the `Tn1` instance family. The largest `Tn1` instance, at the time of this writing, is powered by 16 AWS Trainium chips and has 512 GB of shared accelerator memory. In addition, `Tn1` instances are optimized for quantization and distributed model training, and they support a wide range of data types.

Quantization is a popular way to convert your model parameters from 32-bit precision down to 16-bit precision—or even 8-bit or 4-bit. By quantizing your model weights from 32-bit full precision down to 16-bit half precision, you can quickly reduce your 1-billion-parameter-model memory requirement down 50% to only 2 GB for loading and 40 GB for training.

But before we dive into quantization, let's explore common data types for model training and discuss numerical precision.

Data Types and Numerical Precision

The following are the various data types used by PyTorch and TensorFlow: `fp32` for 32-bit full precision, `fp16` for 16-bit half-precision, and `int8` for 8-bit integer precision.

More recently, `bfloat16` has become a popular alternative to `fp16` for 16-bit precision in more-modern generative AI models. `bfloat16` (or `bf16`) is short for “brain floating point 16” as it was developed at Google Brain. Compared to `fp16`, `bfloat16` has a greater dynamic range with 8 bits for the exponent and can therefore represent a wide range of values that we find in generative AI models.

Let's discuss how these data types compare and why `bfloat16` is a popular choice for 16-bit quantization.

Suppose you want to store `pi` to 20 decimal places (3.14159265358979323846) using full 32-bit precision. Remember that

floating point numbers are stored as a series of bits consisting of only 0s and 1s. Numbers are stored in 32-bits using 1 bit for the sign (negative or positive), 8 bits for the exponent (representing the dynamic range), and 23 bits for the fraction, also called the mantissa or significand, which represents the precision of the number. **Table 4-2** shows how fp32 represents the value of pi.

Table 4-2. fp32 representing pi

Sign	Exponent	Fraction (mantissa/significand)
1 bit	8 bits	23 bits
0	10000000	10010010000111111011011

fp32 can represent numbers in a range from $-3e38$ to $+3e38$. The following PyTorch code shows how to print the data type information for fp32:

```
import torch
torch.finfo(torch.float32)
```

The output is:

```
finfo(resolution=1e-06, min=-3.40282e+38, max=3.40282e+38, eps=1.19209e-07,
smallest_normal=1.17549e-38, tiny=1.17549e-38, dtype=float32)
```

Storing a real number in 32 bits will actually cause a slight loss in precision. You can see this by storing pi as an fp32 data type and then printing the value of the tensor to 20 decimal places using `Tensor.item()`:

```
pi = 3.14159265358979323846
pi_fp32 = torch.tensor(pi, dtype=torch.float32)
print('%.20f' % pi_fp32.item())
```

The output is:

```
3.14159274101257324219
```

You can see the slight loss in precision if you compare this value to the real value of pi, which starts with 3.14159265358979323846. This slight loss in precision is due to the conversion into the fp32 number range, as depicted in [Figure 4-3](#).

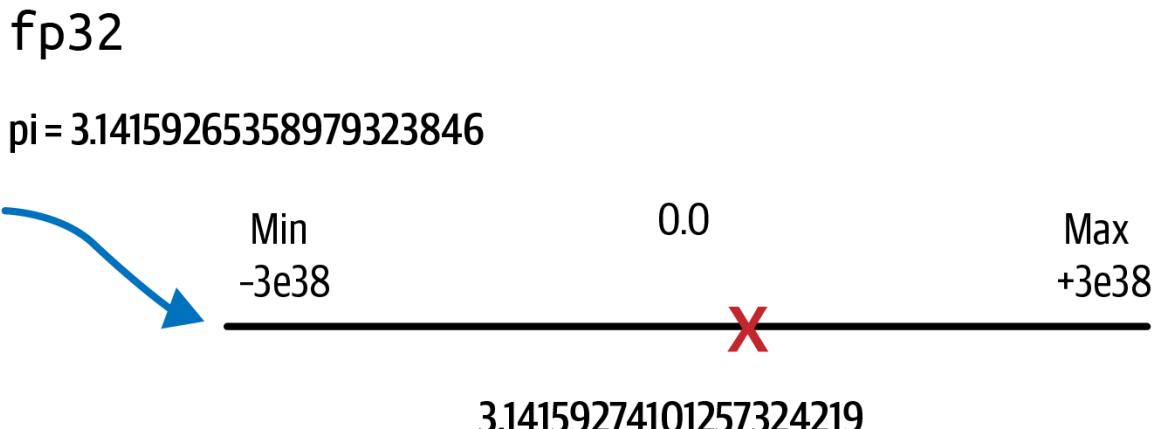


Figure 4-3. fp32 projecting pi into the range from –3e38 to +3e38

You can also print the memory consumption:

```
def show_memory_comsumption(tensor):
    memory_bytes = tensor.element_size() * tensor.numel()
    print("Tensor memory consumption:", memory_bytes, "bytes")
show_memory_comsumption(pi_fp32)
```

The output is:

```
Tensor memory consumption: 4 bytes
```

Now that you've explored data types and numerical representations, let's move on and discuss how quantization can help you reduce the memory footprint required to load and train your multibillion-parameter model.

Quantization

When you try to train a multibillion-parameter model at 32-bit full precision, you will quickly hit the limit of a single NVIDIA A100 or H100

GPU with only 80 GB of GPU RAM. Therefore, you will almost always need to use quantization when using a single GPU.

Quantization reduces the memory needed to load and train a model by reducing the precision of the model weights. Quantization converts your model parameters from 32-bit precision down to 16-bit precision—or even 8-bit or 4-bit.

By quantizing your model weights from 32-bit full-precision down to 16-bit or 8-bit precision, you can quickly reduce your 1-billion-parameter-model memory requirement down 50% to only 2 GB, or even down 75% to just 1 GB for loading, as shown in [Figure 4-4](#).

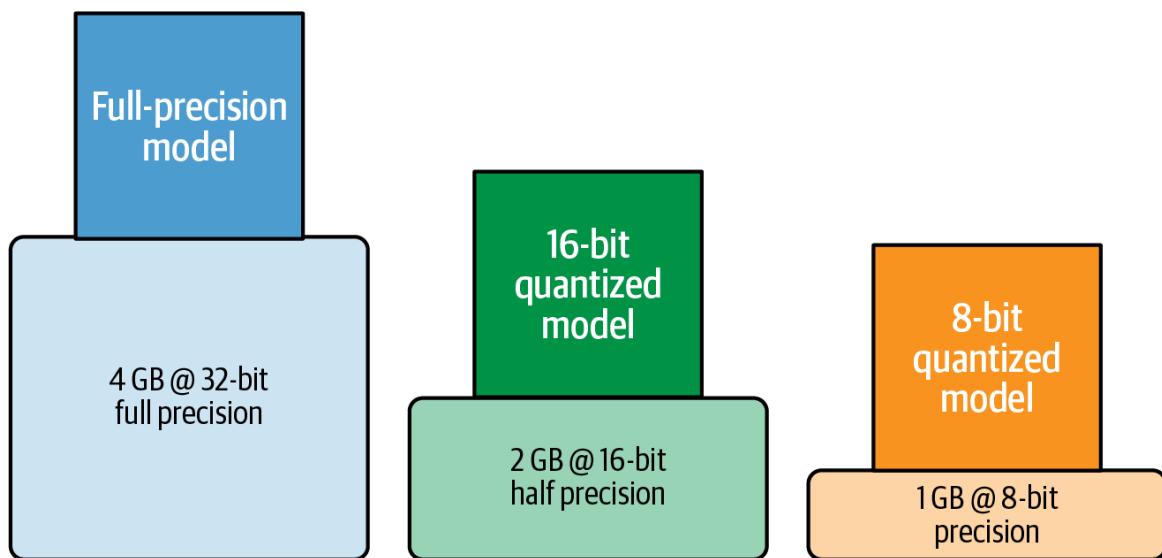


Figure 4-4. Approximate GPU RAM needed to load a 1-billion-parameter model at 32-bit, 16-bit, and 8-bit precision

Quantization projects a source set of higher-precision floating-point numbers into a lower-precision target set of numbers. Using the source and target ranges, the mechanism of quantization first calculates a scaling factor, makes the projection, then stores the results in reduced precision, which requires less memory and ultimately improves training performance and reduces cost.

fp16

With fp16, the 16 bits consist of 1 bit for the sign but only 5 bits for the exponent and 10 bits for the fraction, as shown in [Table 4-3](#).

Table 4-3. fp32 versus fp16

	Sign	Exponent	Fraction (mantissa/significand)
fp32 <i>(consumes 4 bytes of memory)</i>	1 bit 0	8 bits 10000000	23 bits 1001001000011111011011
fp16 <i>(consumes 2 bytes of memory)</i>	1 bit 0	5 bits 10000	10 bits 1001001000

With the reduced number of bits for the exponent and fraction, the range of representable fp16 numbers is only from -65,504 to +65,504. You can also see this when you print the data type information for fp16:

```
torch.info(torch.float16)
```

The output is:

```
finfo(resolution=0.001, min=-65504, max=65504, eps=0.000976562,  
smallest_normal=6.10352e-05, tiny=6.10352e-05, dtype=float16)
```

Let's store pi with 20 decimal places again in fp16 and compare the values:

```
pi = 3.14159265358979323846  
pi_fp16 = torch.tensor(pi, dtype=torch.float16)  
print('%.20f' % pi_fp16.item())
```

The output is:

3.1406250000000000000

Note the loss in precision after this projection, as there are only six places after the decimal point now. The fp16 value of pi is now 3.140625.

Remember that you already lost precision just by storing the value in fp32, as shown in [Figure 4-5](#).

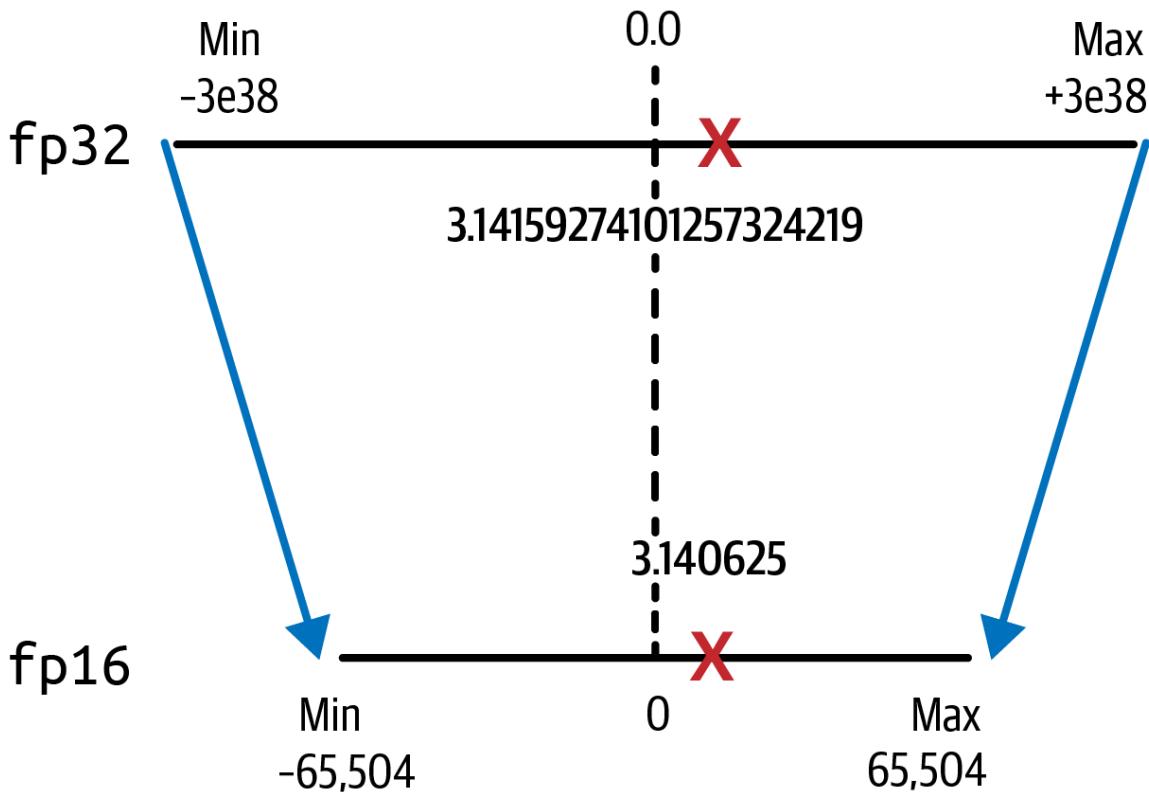


Figure 4-5. Quantization from fp32 to fp16 saves 50% memory

The loss in precision is acceptable in most cases, however. The benefits of a 50% reduction in GPU memory for fp16 compared to fp32 is typically worth the trade-off since fp16 only requires 2 bytes of memory versus 4 bytes of fp32.

Loading a 1-billion-parameter model now only requires 2 GB of GPU RAM, with 12 GB of GPU RAM needed for training the model, as shown in [Figure 4-6](#).

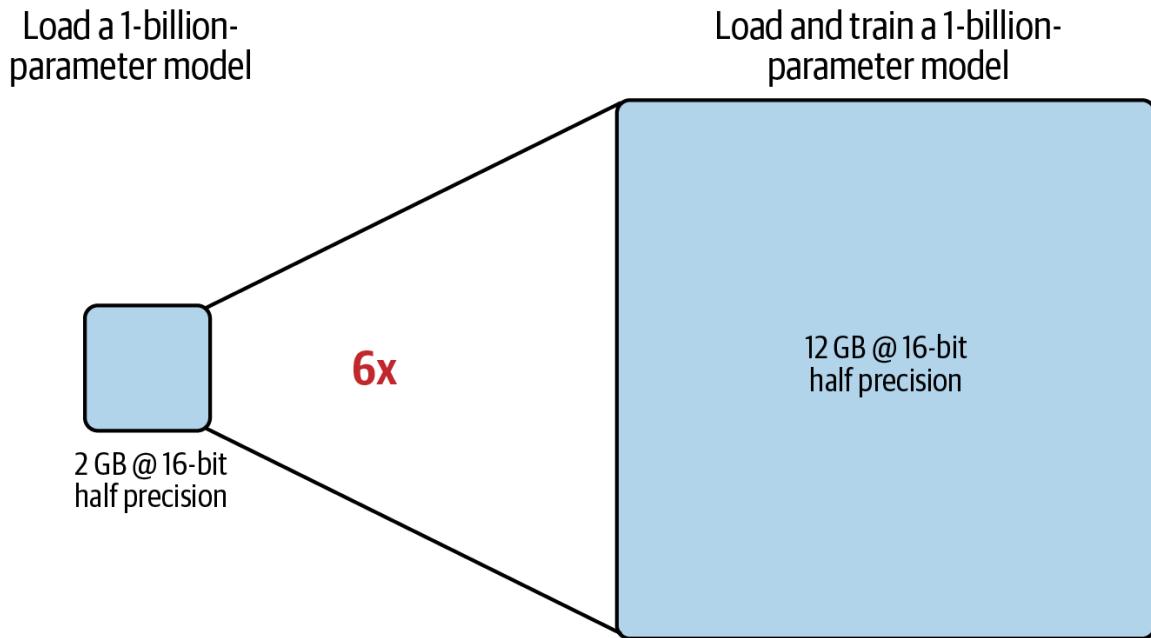


Figure 4-6. Only 12 GB of GPU RAM is needed to load and train a 1-billion-parameter model at 16-bit half precision

bfloating16

bfloating16 has become a popular alternative to fp16 as it captures the full range of fp32 with only 16-bits. This reduces numerical instabilities during model training caused by overflow. Overflow happens when numbers flow outside of the range of representation when converting them from a high-precision to a lower-precision space, causing NaN (not a number) errors.

Compared to fp16, bfloating16 has a greater dynamic range but less precision, which is usually acceptable. bfloating16 uses a single bit for the sign and the full 8 bits for the exponent. However, it truncates the fraction to just 7 bits, which is why it's often called the “truncated 32-bit float,” as shown in [Table 4-4](#).

Table 4-4. fp32 versus bfloat16

	Sign	Exponent	Fraction (mantissa/significand)
fp32 <i>(consumes 4 bytes of memory)</i>	1 bit 0	8 bits 10000000	23 bits 1001001000011111011011
bfloat16 <i>(consumes 2 bytes of memory)</i>	1 bit 0	8 bits 10000000	7 bits 1001001

The range of representable bfloat16 numbers is identical to fp32. Let's print the data type information for bfloat16:

```
torch.finfo(torch.bfloat16)
```

The output is:

```
finfo(resolution=0.01, min=-3.38953e+38, max=3.38953e+38, eps=0.0078125,
smallest_normal=1.17549e-38, tiny=1.17549e-38, dtype=bfloat16)
```

Let's store pi with 20 decimal places again in bfloat16 and compare the values:

```
pi = 3.14159265358979323846
pi_bfloat16 = torch.tensor(pi, dtype=torch.bfloat16)
print('%.20f' % pi_bfloat16.item())
```

The output is:

```
3.14062500000000000000
```

Similar to fp16, bfloat16 comes with a minimal loss in precision. The bfloat16 value of pi is 3.140625. However, the benefits of maintaining the dynamic range of fp32 (shown in [Figure 4-7](#)) and thereby reducing overflow, usually outweighs the loss in precision.

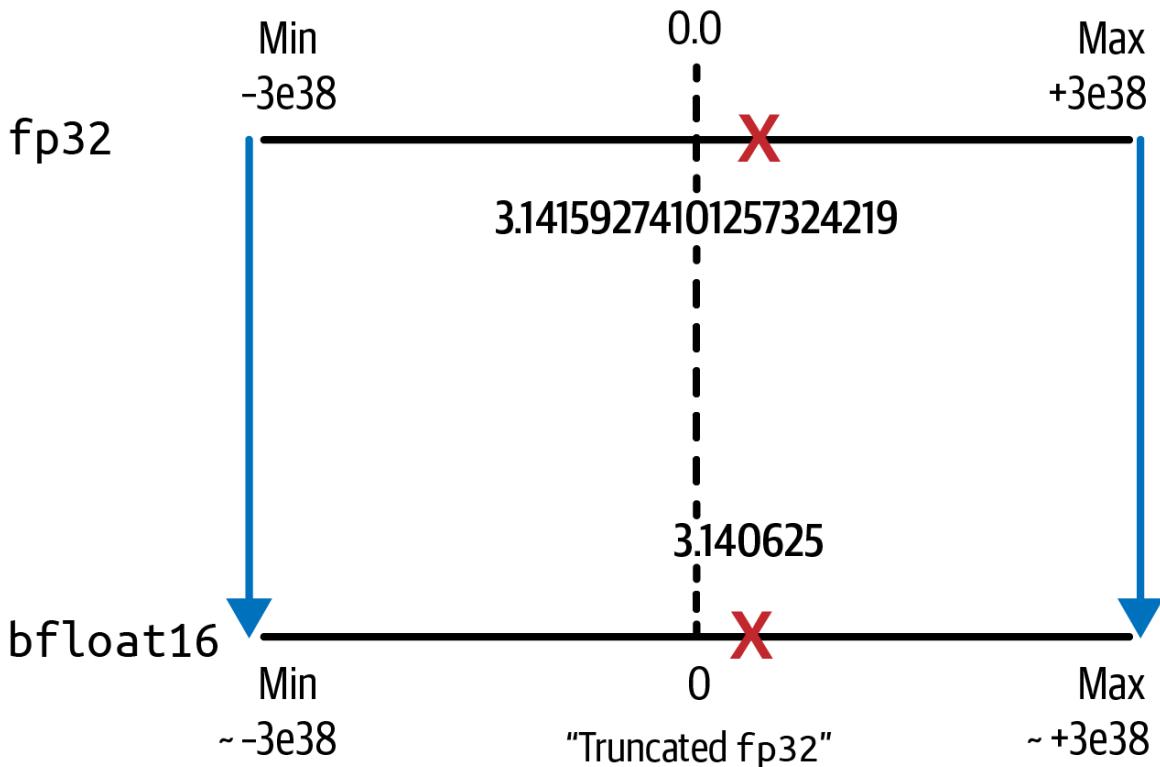


Figure 4-7. Quantization from fp32 to bfloat16 maintains the dynamic range of fp32 while still saving 50% memory

bfloat16 is natively supported by newer GPUs such as NVIDIA's A100 and H100. Many modern generative AI models were pretrained with bfloat16, including FLAN-T5, Falcon, and Llama 2.

fp8

fp8 is a newer data type and natural progression from fp16 and bfloat16 to further reduce memory and compute footprint for multibillion-parameter models.

fp8 allows the user to configure the number of bits assigned to the exponent and fraction depending on the task, such as training, inference, or post-

training quantization. NVIDIA GPUs started supporting fp8 with the H100 chip. AWS Trainium also supports fp8, called configurable fp8, or just cfp8. With cfp8, 1 bit is used for the sign, and the remaining 7 bits are configurable between the exponent and fraction, as shown in [Table 4-5](#).

Table 4-5. fp32 versus fp8

	Sign	Exponent	Fraction (mantissa/significand)
fp32 <i>(consumes 4 bytes of memory)</i>	1 bit 0	8 bits 10000000	23 bits 1001001000011111011011
fp8 <i>(consumes 1 byte memory)</i>	1 bit 0	7 bits 0000011 (configurable)	

Empirical results show that fp8 can match model training performance of fp16 and bfloat16 while reducing memory footprint by another 50% and speeding up model training.

int8

Another quantization option is int8 8-bit quantization. Using 1 bit for the sign, int8 values are represented by the remaining 7 bits, as shown in [Table 4-6](#).

Table 4-6. *fp32* versus *int8*

	Sign	Exponent	Fraction (mantissa/significand)
fp32 <i>(consumes 4 bytes of memory)</i>	1 bit 0	8 bits 10000000	23 bits 1001001000011111011011
int8 <i>(consumes 1 byte of memory)</i>	1 bit 0	n/a	7 bits 0000011

The range of representable int8 numbers is –128 to +127. Here’s the data type information for int8:

```
torch.iinfo(torch.int8)
```

The output is:

```
iinfo(min=-128, max=127, dtype=int8)
```

Let’s store pi with 20 decimal places again in int8 and see what happens:

```
pi = 3.14159265358979323846
pi_int8 = torch.tensor(pi, dtype=torch.int8)
print(pi_int8.item())
```

The output is:

Unsurprisingly, π is projected to just 3 in the 8-bit lower precision space, as shown in [Figure 4-8](#).

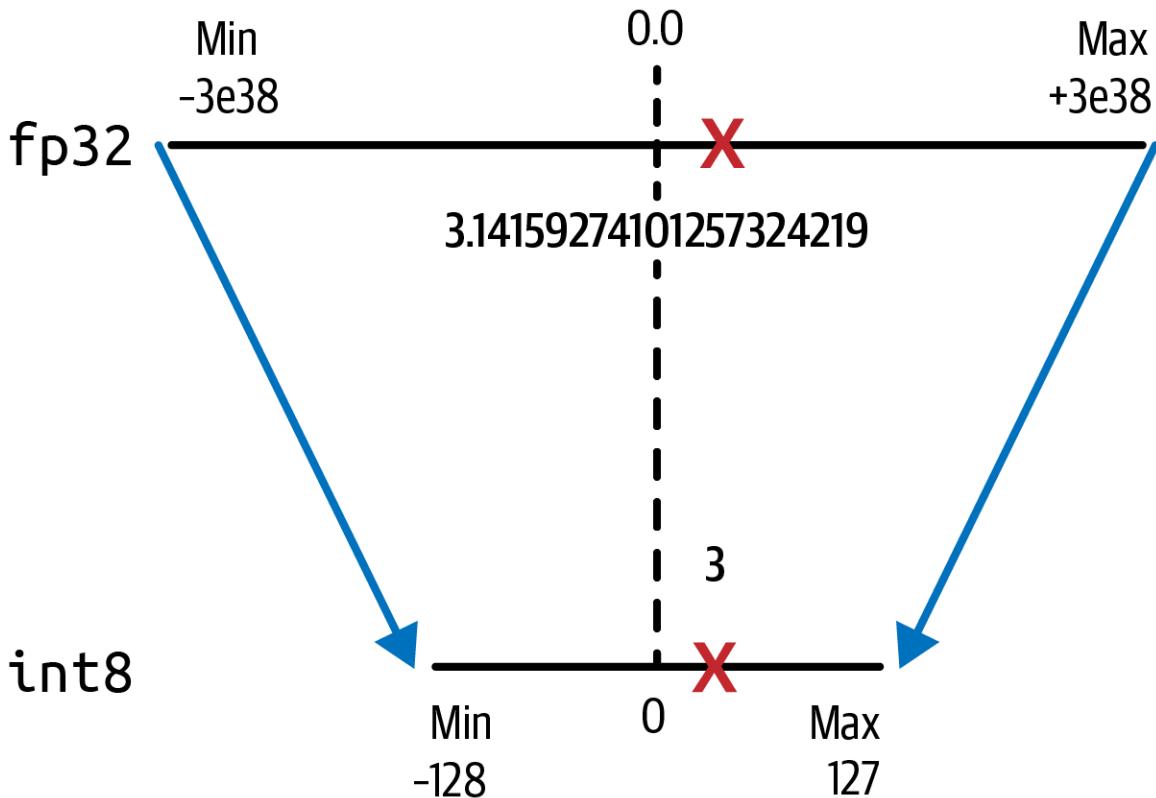


Figure 4-8. Quantization from fp32 to int8 saves 75% memory

This brings the memory requirement down from originally 4 bytes to just 1 byte, but results in a bigger loss of precision due to the conversion from a floating point representation to an integer value.

Reducing the memory footprint of large foundation models is not only helpful for loading and training models, but also for inference. Despite the loss in precision, 8-bit quantization is often used to improve inference throughput and latency for deployed models. Optimized implementations for `int8` quantization such as Hugging Face's [bitsandbytes integration of LLM.int8\(\)](#), have shown to minimize quantization impact on model performance. You will learn about post-training quantization (PTQ) and the technique GPT post-training quantization (GPTQ)¹ in more detail when you prepare the model for deployment in [Chapter 8](#).

[Table 4-7](#) compares the data types discussed thus far.