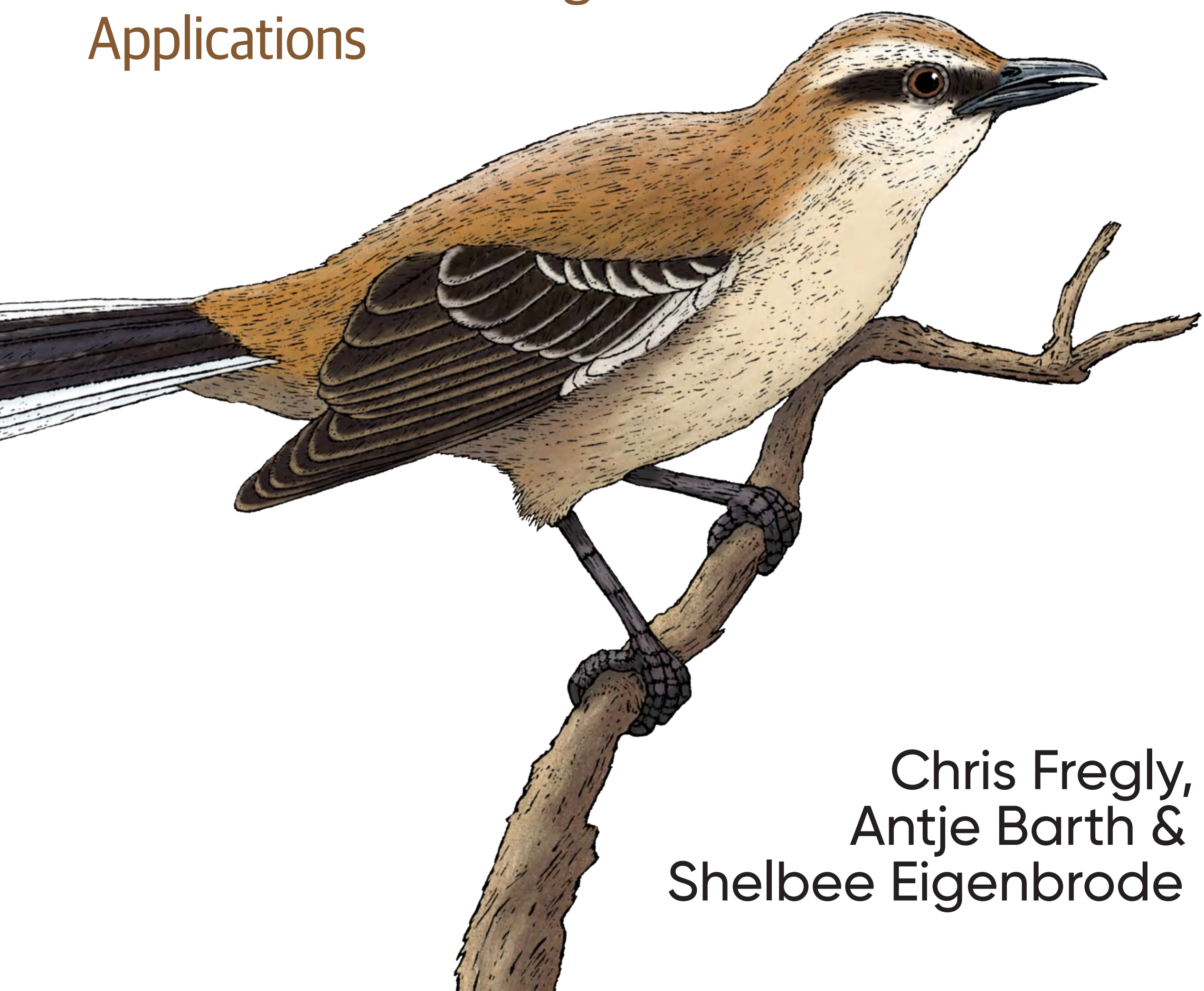
O'REILLY

Generative AI on AWS

Building Context-Aware Multimodal Reasoning Applications



Chris Fregly,
Antje Barth &
Shelbee Eigenbrode

O'REILLY®

Generative AI on AWS

Companies today are moving rapidly to integrate generative AI into their products and services. But there's a great deal of hype (and misunderstanding) about the impact and promise of this technology. With this book, Chris Fregly, Antje Barth, and Shelbee Eigenbrode from AWS help CTOs, ML practitioners, application developers, business analysts, data engineers, and data scientists find practical ways to use this exciting new technology.

You'll learn the generative AI project life cycle including use case definition, model selection, model fine-tuning, retrieval-augmented generation, reinforcement learning from human feedback, and model quantization, optimization, and deployment. And you'll explore different types of models including large language models (LLMs) and multimodal models such as Stable Diffusion for generating images and Flamingo/IDEFICS for answering questions about images.

- **Apply generative AI to your business use cases**
- **Determine which generative AI models are best suited to your task**
- **Perform prompt engineering and in-context learning**
- **Fine-tune generative AI models on your datasets with low-rank adaptation (LoRA)**
- **Align generative AI models to human values with reinforcement learning from human feedback (RLHF)**
- **Augment your model with retrieval-augmented generation (RAG)**
- **Explore libraries such as LangChain and ReAct to develop agents and actions**
- **Build generative AI applications with Amazon Bedrock**

"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!"

—Jeff Barr
VP and Chief Evangelist @ AWS

Chris Fregly is a Principal Solutions Architect for generative AI at Amazon Web Services and coauthor of *Data Science on AWS* (O'Reilly).

Antje Barth is Principal Developer Advocate for generative AI at Amazon Web Services and coauthor of *Data Science on AWS*.

Shelbee Eigenbrode is a Principal Solutions Architect for generative AI at Amazon Web Services. She holds over 35 patents across various technology domains.

DATA

US $79.99 CAN $99.99
ISBN: 978-1-098-15922-1

Twitter: @oreillymedia
linkedin.com/company/oreilly-media
youtube.com/oreillymedia

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

Generative AI on AWS

Building Context-Aware Multimodal Reasoning Applications

Chris Fregly, Antje Barth, and Shelbee Eigenbrode

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Generative AI on AWS
by Chris Fregly, Antje Barth, and Shelbee Eigenbrode

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.

978-1-098-15922-1

[LSI]

Table of Contents

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.

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.

This element signifies a tip or suggestion.

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

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!

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 multiturn 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.

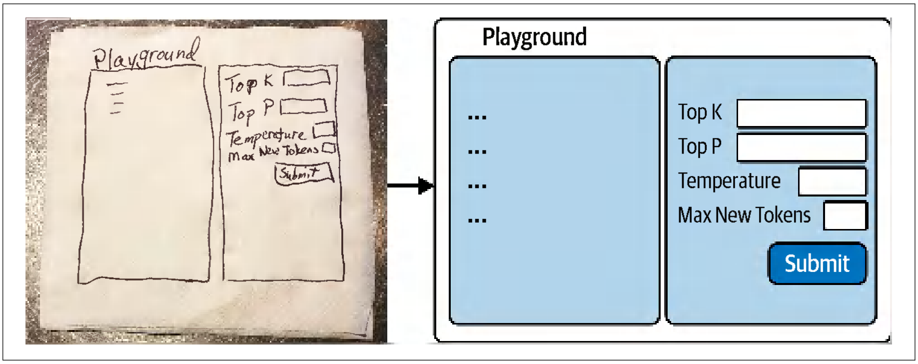


Figure 1-1. Generating UI code from hand-drawn sketch

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 Jump-Start 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.

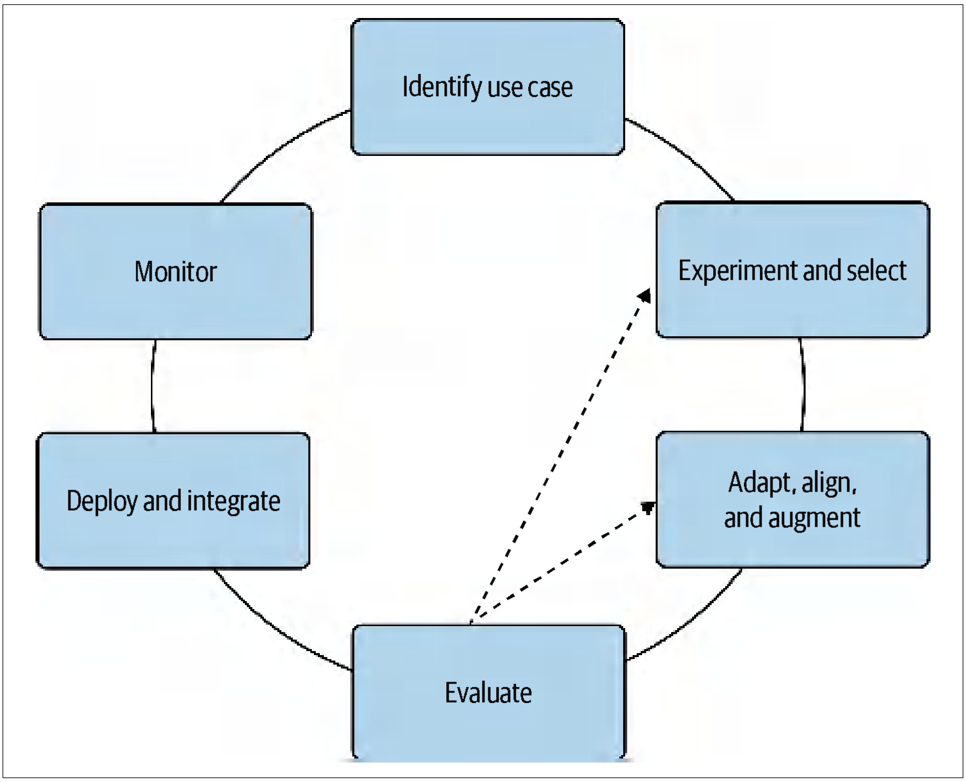


Figure 1-3. Generative AI project life cycle framework

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.

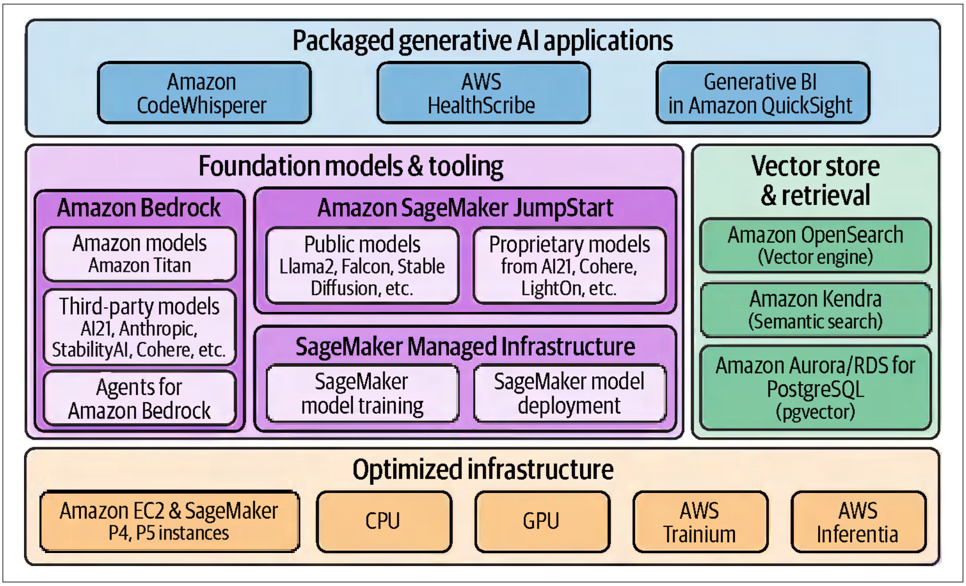


Figure 1-4. AWS services and features supporting generative AI

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" on page 5—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.

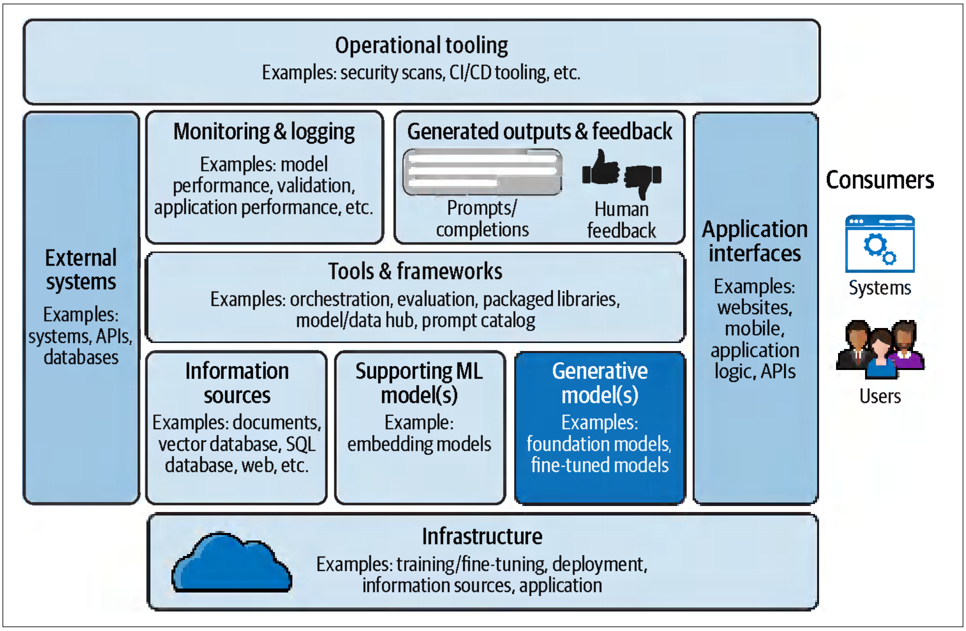


Figure 1-5. Generative AI applications include more than foundation models

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.

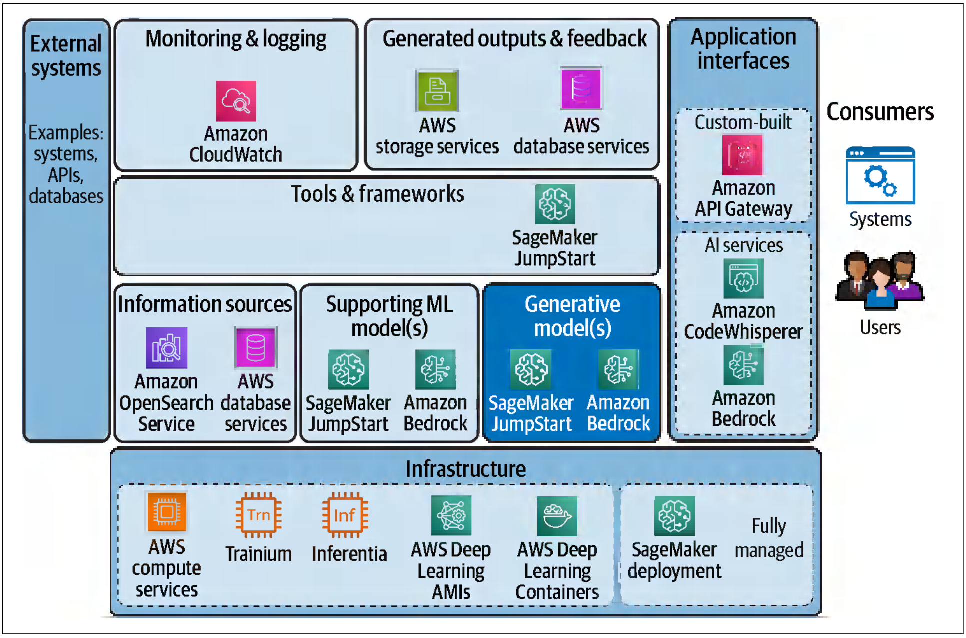


Figure 1-6. AWS breadth of service to enable customers to build generative AI applications

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).

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.

> 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.

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.”

> 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.

> 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.

> 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.

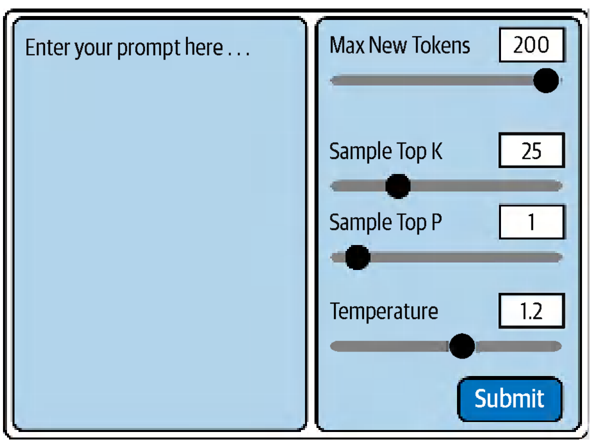


Figure 2-1. Inference configuration parameters to control model outputs

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.

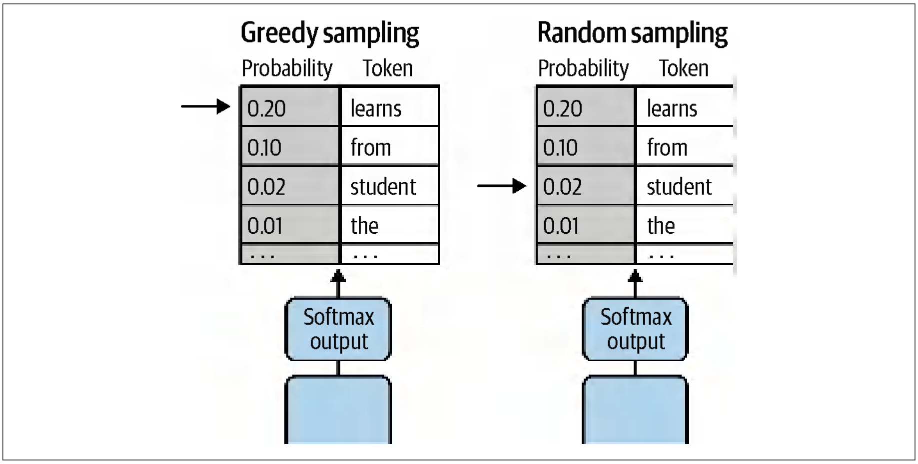


Figure 2-2. Greedy versus random sampling to predict the next token from a probability distribution

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!

> 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.

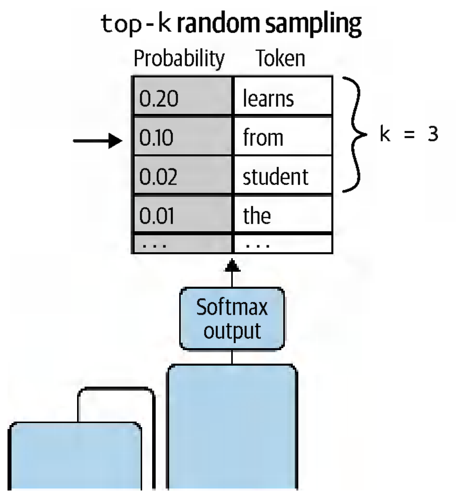


Figure 2-3. In this case, `top-k` sampling restricts the model to choosing from the top three probabilities

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.

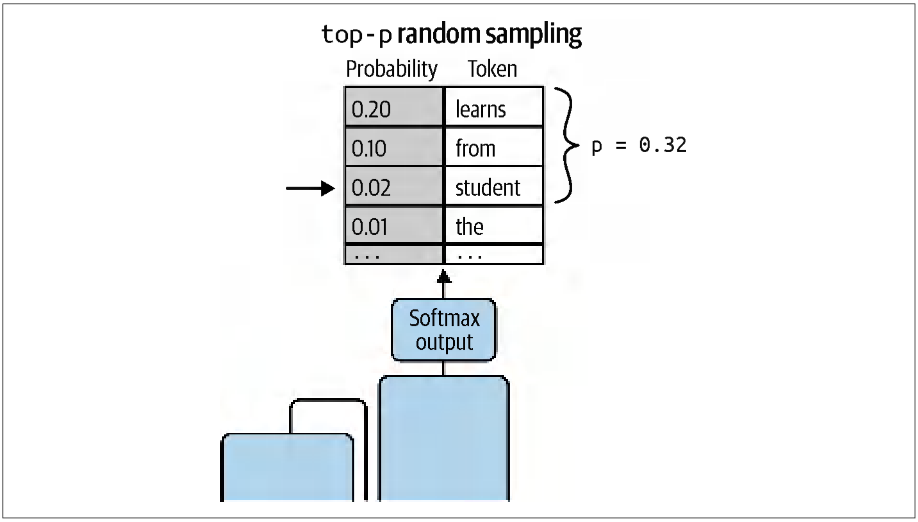


Figure 2-4. `top-p` random probability weighting

`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.

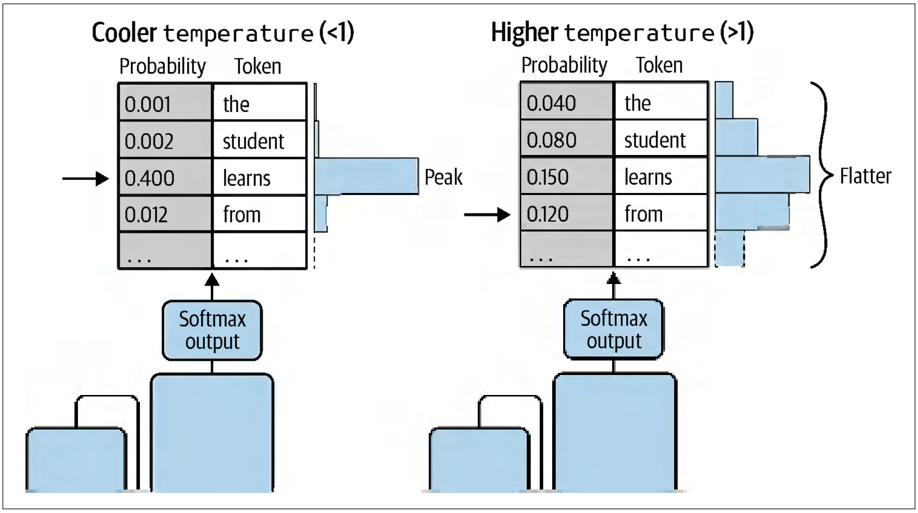


Figure 2-5. Changing the `temperature` will change the next-token probability distribution

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.

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.

> 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.

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.

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 papers[3] 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.

> 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.

3 Hugo Touvron et al., "Llama 2: Open Foundation and Fine-Tuned Chat Models", *arXiv*, 2023.

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 *(public and private)*	Web	42%
	News	5%
	Filings	2%
	Press	1%
	Bloomberg	1%
	TOTAL	**51%**
Other data *(public)*	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".

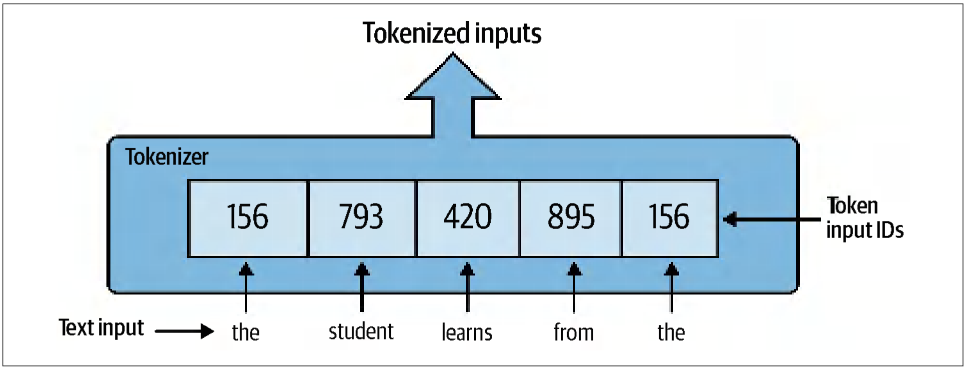


Figure 3-1. Use a tokenizer to convert text inputs into vectors for machine-readable processing

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.

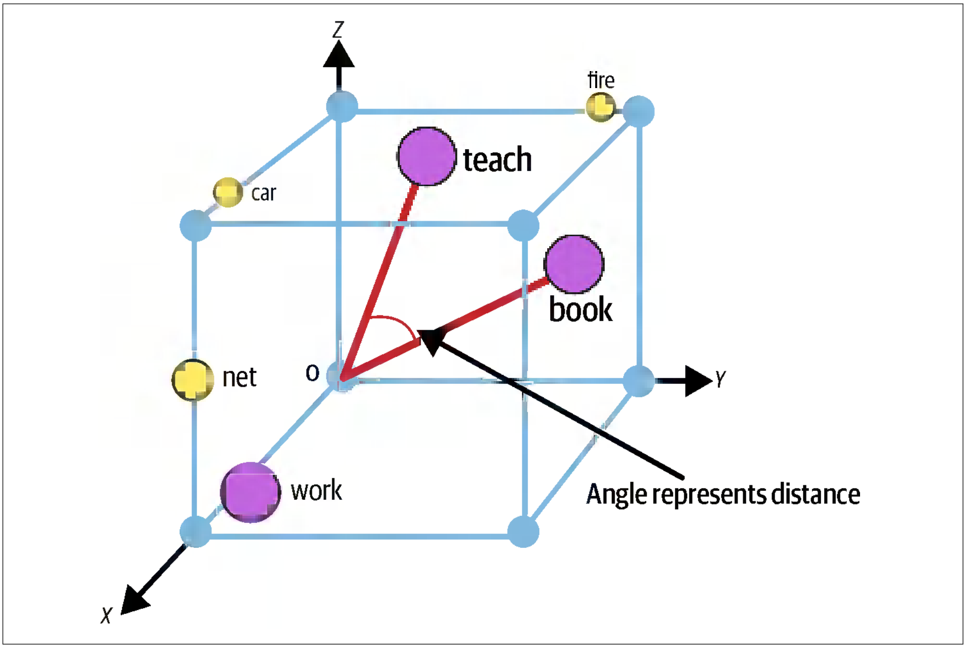


Figure 3-2. Representation of tokens in an example three-dimensional embedding space

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.

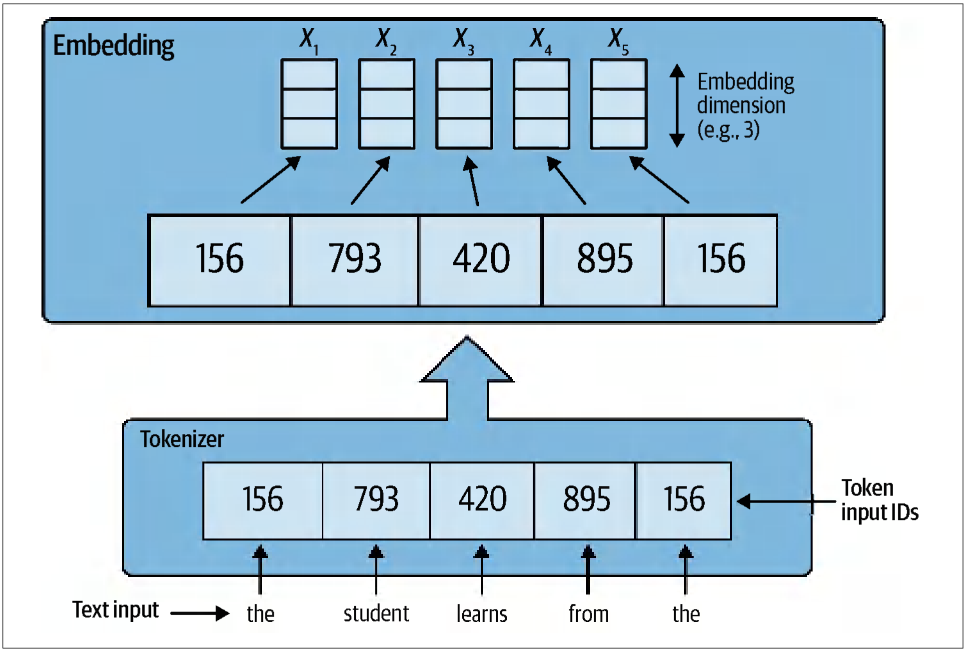


Figure 3-3. Embedding vector space of three dimensions

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.

> 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.

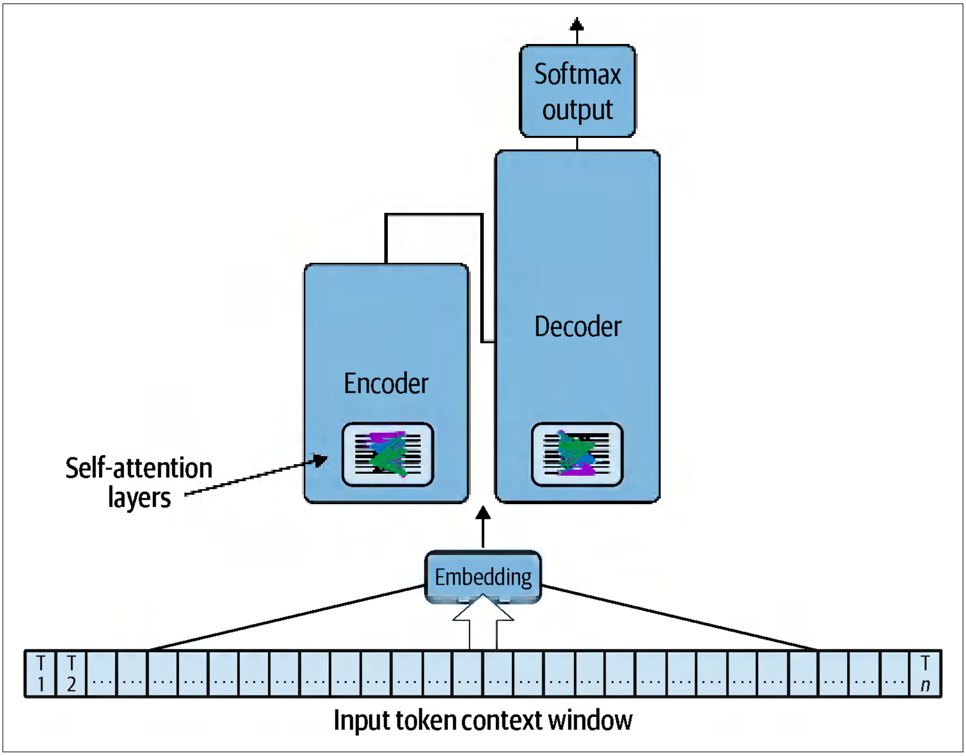


Figure 3-4. High-level Transformer architecture

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."

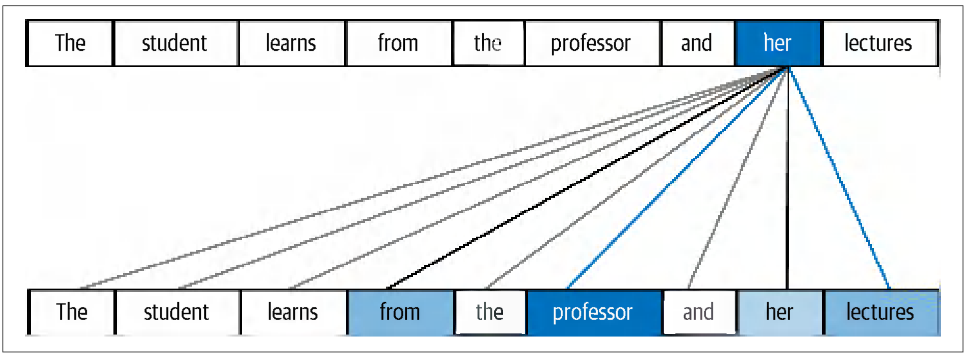


Figure 3-5. The "self-attention" mechanism attends every token in the data to all other tokens in the input sequence

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.

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.

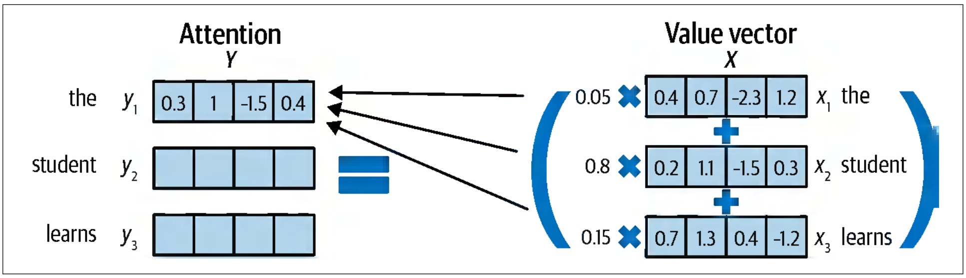


Figure 3-6. Attention is the weighted average of the input vectors

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.

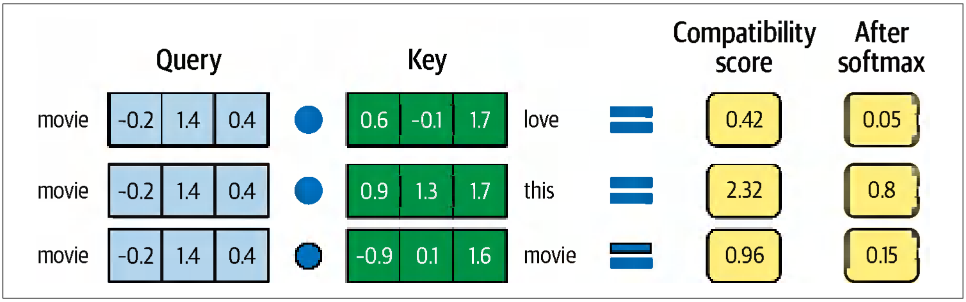


Figure 3-7. Attention weights are the normalized dot product of the query and key vectors

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.

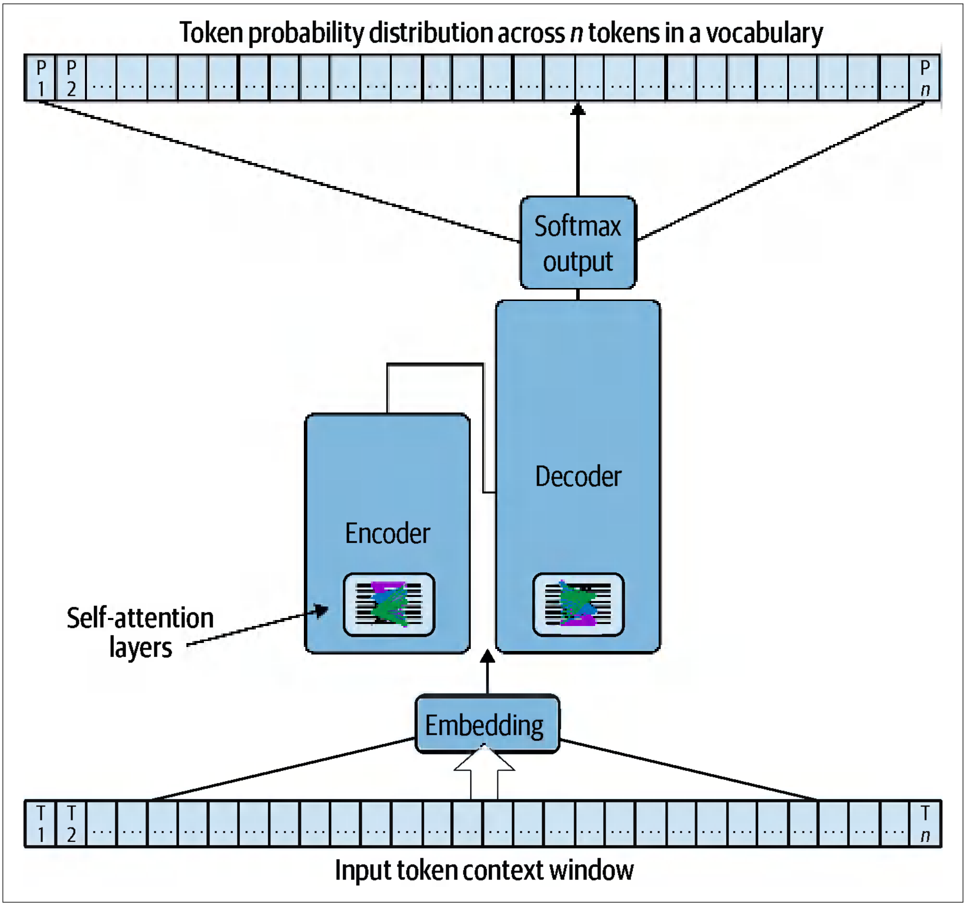


Figure 3-8. Probability of being the next token across all tokens in the vocabulary

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.

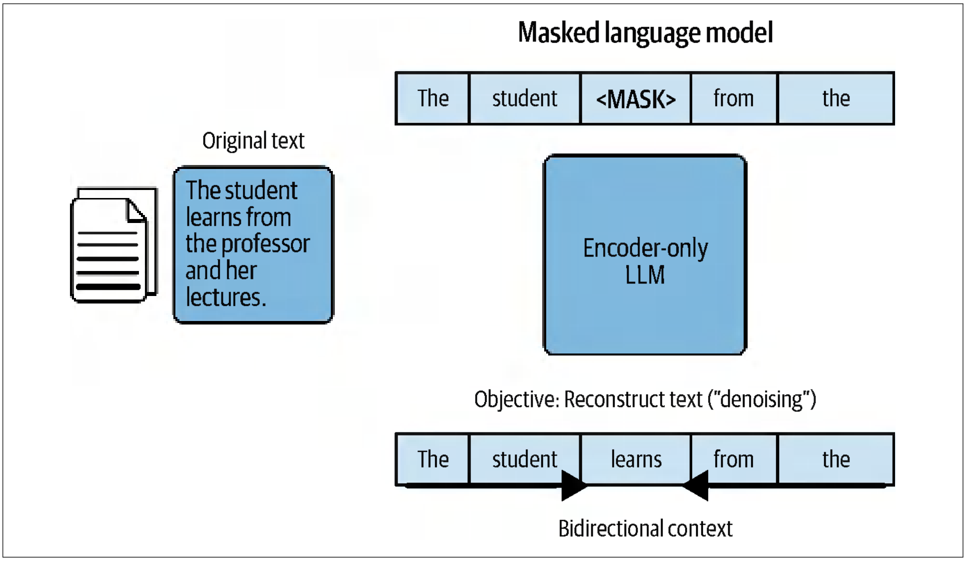


Figure 3-9. Encoder-only (autoencoder) models use a bidirectional context to reconstruct the masked input tokens

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" on page 158.

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.

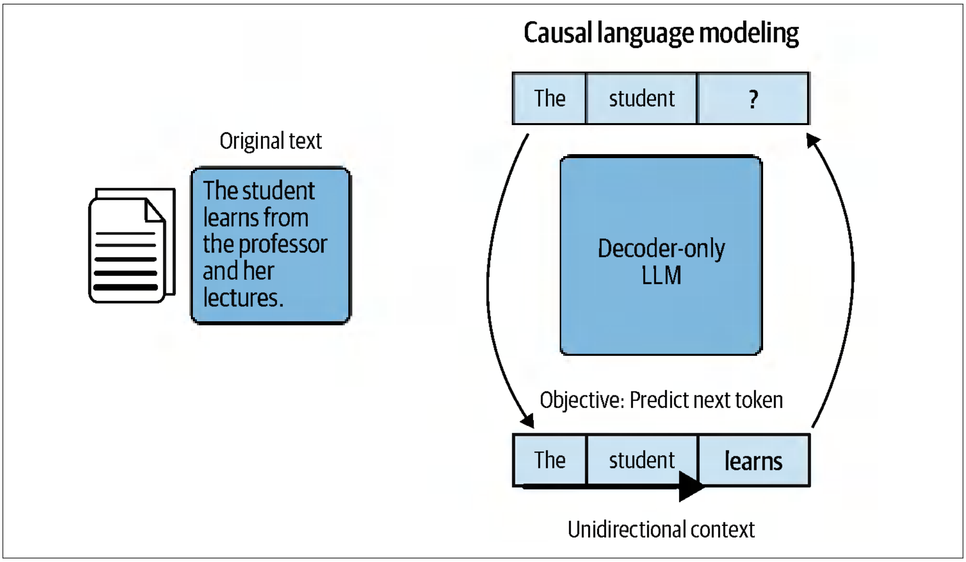


Figure 3-10. Decoder-only (autoregressive) models only reveal the tokens leading up to the token being predicted

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.

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.

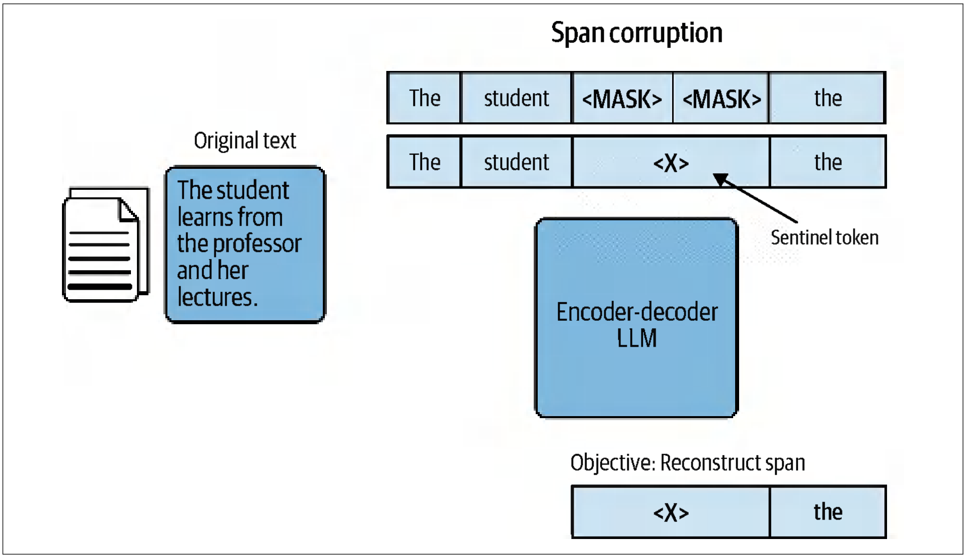


Figure 3-11. Encoder-decoder (sequence-to-sequence) models

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.

> 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 laws[8] 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.

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.

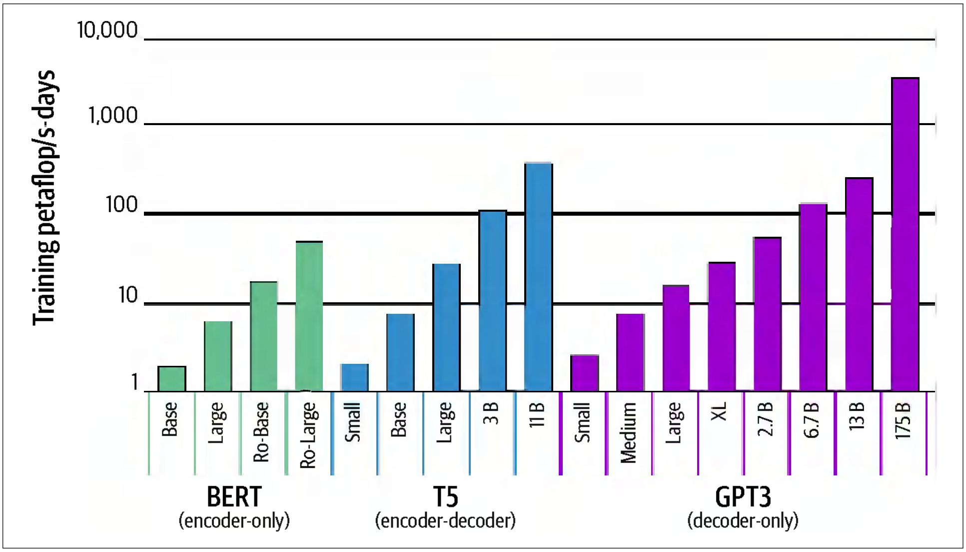


Figure 3-12. Pretraining requirements for common models in petaflop/s-days (source: adapted from an image in Brown et al.)

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.

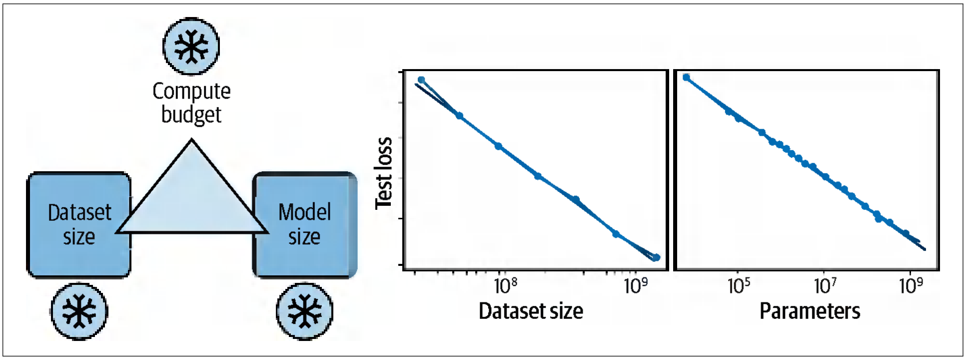


Figure 3-13. Impact of dataset size and parameter size on model performance (source: charts in figure adapted from an image in Kaplan et al.)

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 paper[9] 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.

9 Jordan Hoffman et al., "Training Compute-Optimal Large Language Models", *arXiv*, 2022.

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	Compute-optimal (20x)
LLaMA-65B	65 B	1.3 T	1.4 T	Compute-optimal (20x)
GPT-3	175 B	**3.5 T**	**300 B**	Overparameterized for dataset size (<20x)
OPT-175B	175 B	**3.5 T**	**180 B**	Overparameterized for dataset size (<20x)
BLOOM	176 B	**3.5 T**	**350 B**	Overparameterized for dataset size (<20x)
Llama2-70B	70 B	1.4 T	2.0 T	Better than compute-optimal (>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

10 Dan Hendrycks et al., "Measuring Massive Multitask Language Understanding", *arXiv*, 2021.

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.

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**

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.

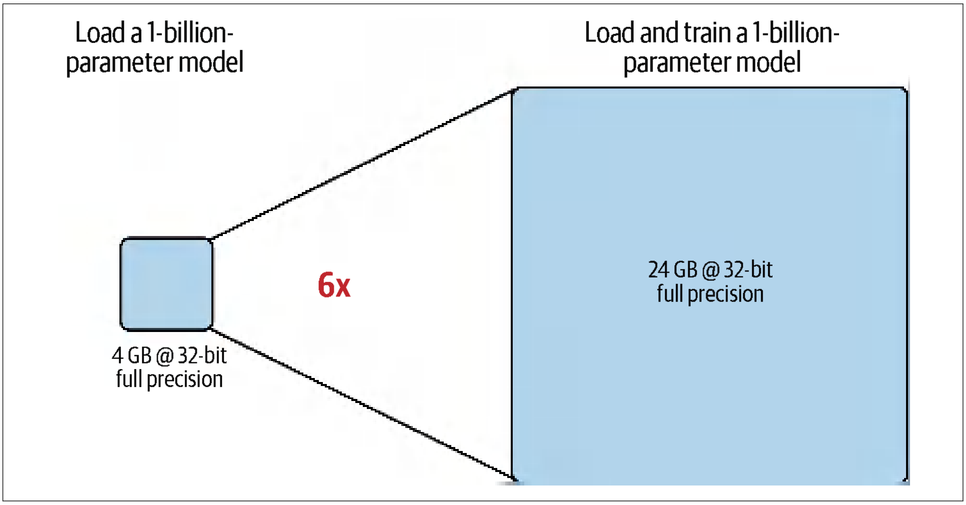


Figure 4-2. Comparison of approximate GPU RAM needed to load versus load and train a 1-billion-parameter model at 32-bit full precision

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 `Trn1` instance family. The largest `Trn1` instance, at the time of this writing, is powered by 16 AWS Trainium chips and has 512 GB of shared accelerator memory. In addition, `Trn1` 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.

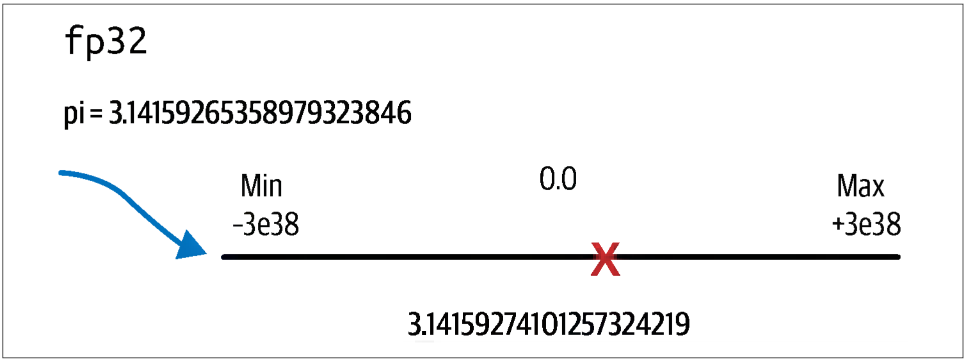


Figure 4-3. fp32 projecting pi into the range from –3e38 to +3e38

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.

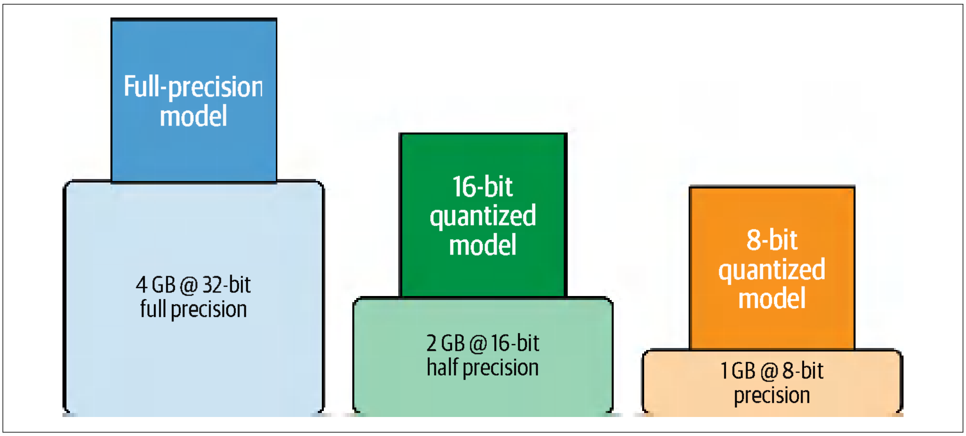


Figure 4-4. Approximate GPU RAM needed to load a 1-billion-parameter model at 32-bit, 16-bit, and 8-bit precision

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` *(consumes 4 bytes of memory)*	**1 bit** `0`	**8 bits** `10000000`	**23 bits** `10010010000111111011011`
`fp16` *(consumes 2 bytes of memory)*	**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.finfo(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:

```
3.14062500000000000000
```

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.

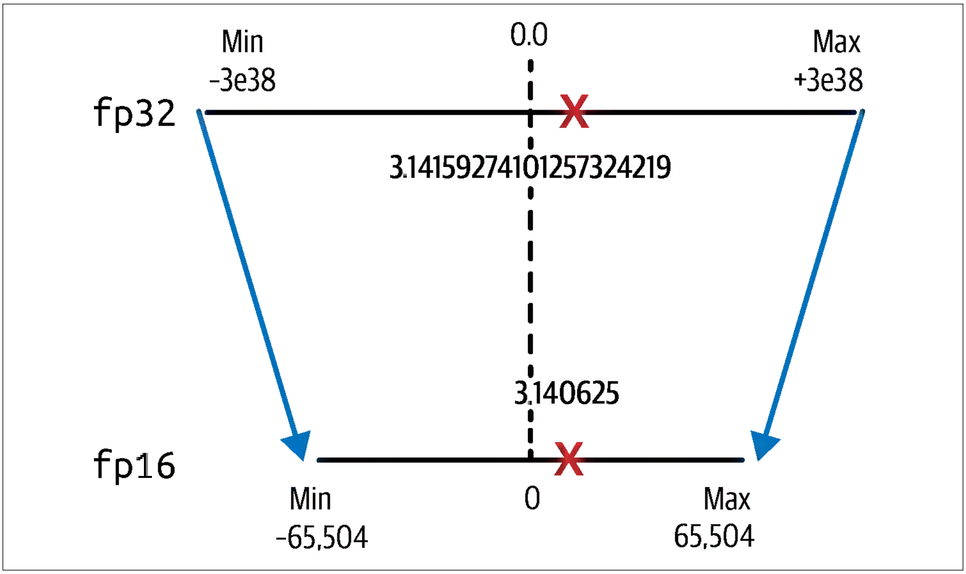


Figure 4-5. Quantization from `fp32` to `fp16` saves 50% memory

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.

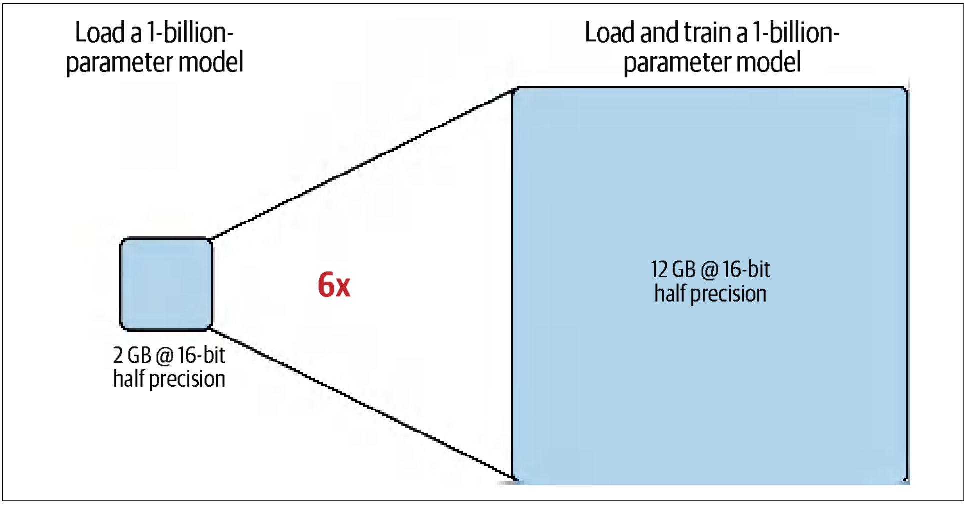


Figure 4-6. Only 12 GB of GPU RAM is needed to load and train a 1-billion-parameter model at 16-bit half precision

bfloat16

`bfloat16` 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`, `bfloat16` has a greater dynamic range but less precision, which is usually acceptable. `bfloat16` 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` *(consumes 4 bytes of memory)*	**1 bit** `0`	**8 bits** `10000000`	**23 bits** `10010010000111111011011`
`bfloat16` *(consumes 2 bytes of memory)*	**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.

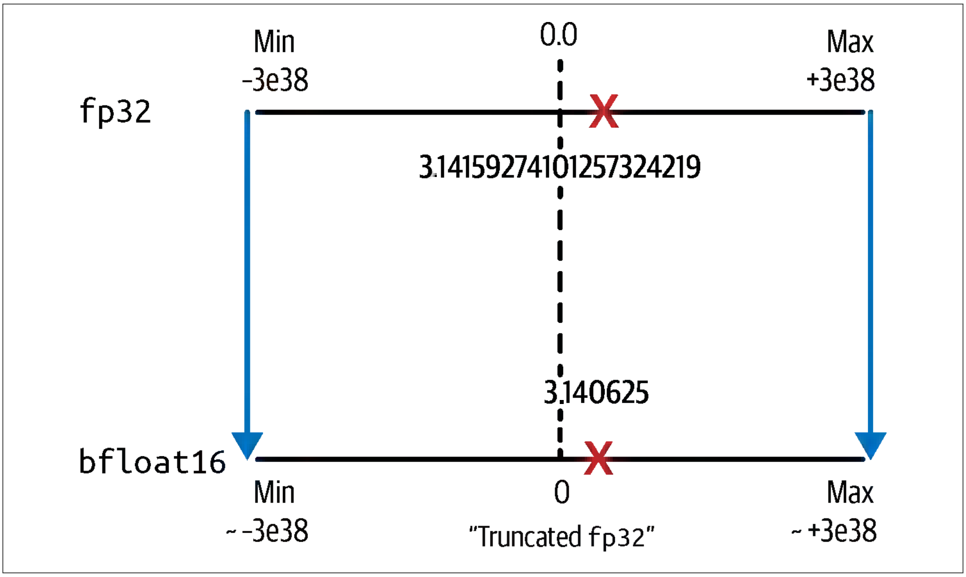


Figure 4-7. Quantization from `fp32` to `bfloat16` maintains the dynamic range of `fp32` while still saving 50% memory

`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` *(consumes 4 bytes of memory)*	**1 bit** `0`	**8 bits** `10000000`	**23 bits** `10010010000111111011011`
`fp8` *(consumes 1 byte memory)*	**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` *(consumes 4 bytes of memory)*	**1 bit** `0`	**8 bits** `10000000`	**23 bits** `10010010000111111011011`
`int8` *(consumes 1 byte of memory)*	**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:

```
3
```

Unsurprisingly, `pi` is projected to just `3` in the 8-bit lower precision space, as shown in Figure 4-8.

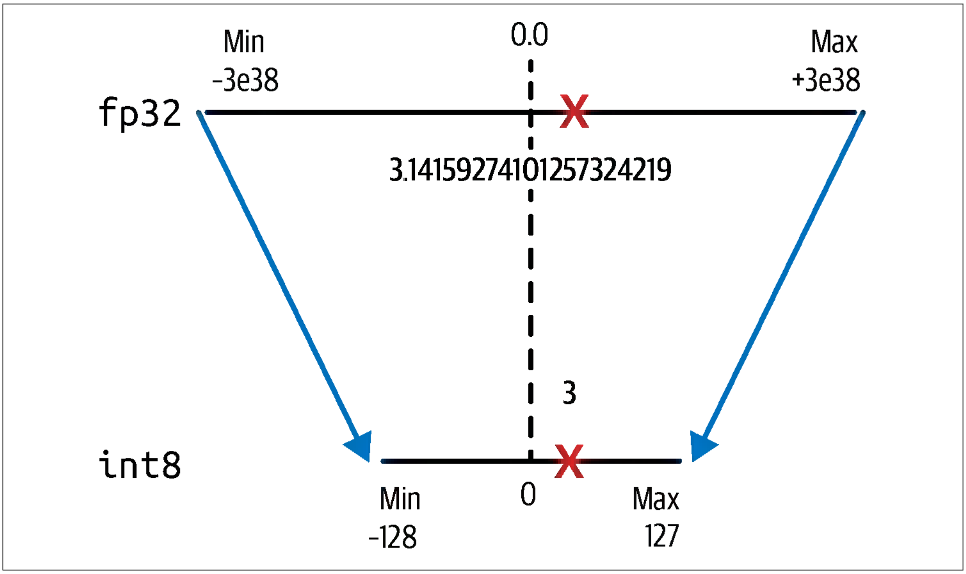


Figure 4-8. Quantization from `fp32` to `int8` saves 75% memory

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.

1 Elias Frantar et al., "GPTQ: Accurate Post-Training Quantization for Generative Pre-Trained Transformers", *arXiv*, 2023.

Table 4-7 compares the data types discussed thus far.

Table 4-7. Comparison of data types used for quantization

	Total bits	Sign bits	Exponent bits	Fraction bits	Memory needed to store one value
fp32	32	1	8	23	4 bytes
fp16	16	1	5	10	2 bytes
bf16	16	1	8	7	2 bytes
fp8	8	1	7		1 byte
int8	8	1	n/a	7	1 byte

In summary, the choice of data type for model quantization should be based on the specific needs of your application. While `fp32` offers a safe choice if accuracy is paramount, you will likely hit hardware limits, such as available GPU RAM, especially for multibillion-parameter models.

In this case, quantization using `fp16` and `bfloat16` can help to reduce the required memory footprint by 50%. `bfloat16` is usually preferred over `fp16` as it maintains the same dynamic range as `fp32` and reduces overflow. `fp8` is an emerging data type to further reduce memory and compute requirements. Some hardware implementations allow configuring the bits for exponent and fraction; empirical results show performance can match model training with `fp16` and `bfloat16`. `int8` has become a popular choice to optimize your model for inference. `fp8` is becoming more popular as both hardware and deep-learning framework support emerges.

It is recommended that you always benchmark the quantization results to ensure the selected data type meets your accuracy and performance requirements.

Another memory and compute optimization technique is FlashAttention. FlashAttention aims to reduce the quadratic compute and memory requirements, $O(n^2)$, of the self-attention layers in Transformer-based models.

Optimizing the Self-Attention Layers

As mentioned in Chapter 3, performance of the Transformer is often bottlenecked by the compute and memory complexity of the self-attention layers. Many performance improvements are targeted specifically at these layers. Next, you will learn some powerful techniques to reduce memory and increase performance of the self-attention layers.

FlashAttention

The Transformer's attention layer is a bottleneck when trying to scale to longer input sequences because the computation and memory requirements scale quadratically $O(n^2)$ with the number of input tokens. FlashAttention, initially proposed in a research paper,[2] is a GPU-specific solution to this quadratic scaling problem.

FlashAttention, on version 2 as of this writing, reduces the amount of reads and writes between GPU main memory, called high-bandwidth memory (HBM), and the much faster but smaller on-chip GPU static RAM (SRAM). Despite its name, the GPU high-bandwidth memory is an order of magnitude slower than the on-chip GPU SRAM.

Overall, FlashAttention increases self-attention performance by 2–4x and reduces memory usage 10–20x by reducing the quadratic $O(n^2)$ computational and memory requirements down to linear $O(n)$, where n is the number of input tokens in the sequence. With FlashAttention, the Transformer scales to handle much longer input sequences which allows for better performance on larger input context windows.

A popular implementation is installable with a simple `pip install flash-attn --no-build-isolation` command which installs the flash-attn library as a drop-in replacement for the original attention.

Attention optimizations are an active area of research, including the next generation FlashAttention-2,[3] which continues to implement GPU-specific optimizations to improve performance and reduce memory requirements.

Let's learn about another technique to improve the performance of the self-attention layers in the Transformer.

Grouped-Query Attention

Another popular optimization to the attention layers is grouped-query attention (GQA). GQA improves upon the Transformer's traditional multiheaded attention, described in Chapter 3, by sharing a single key (k) and value (v) head for each *group* of query (q) heads (as opposed to each query head), as shown in Figure 4-9.

2 Tri Dao et al., "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness", *arXiv*, 2022.

3 Tri Dao, "FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning", *arXiv*, 2023.

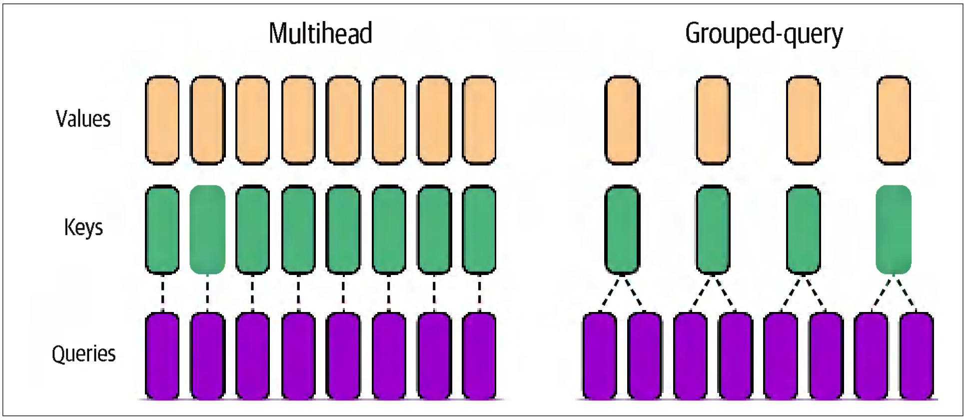


Figure 4-9. Grouped-query attention versus traditional multiheaded attention

GQA allows queries to be grouped into fewer key and value heads and therefore reduces memory consumption of the attention heads. In addition, GQA improves performance by reducing the number of memory reads and writes.

Since these improvements are proportional to the number of input tokens, MQA is particularly useful for longer input token sequences and allows for a larger context window. For example, the Llama 2 model by Meta uses GQA to improve performance and increase the input token context window size to 4,096—double the original LLaMA model's 2,048 context window size.

Distributed Computing

For larger models, you will likely need to use a distributed cluster of GPUs to train these massive models across hundreds or thousands of GPUs. There are many different types of distributed computing patterns, including distributed data parallel (DDP) and fully sharded data parallel (FSDP). The main difference is how the model is split—or sharded—across the GPUs in the system.

If the model parameters can fit into a single GPU, then you would choose DDP to load a single copy of the model into each GPU. If the model is too large for a single GPU—even after quantization—then you need to use FSDP to shard the model across multiple GPUs. In both cases, the data is split into batches and spread across all available GPUs to increase GPU utilization and cost efficiency at the expense of some communication overhead, which you will see in a bit.

Distributed Data Parallel

PyTorch comes with an optimized implementation of DDP that automatically copies your model onto each GPU (assuming it fits into a single GPU using a technique such as quantization), splits the data into batches, and sends the batches to each GPU in parallel. With DDP, each batch of data is processed in parallel on each GPU, followed by a synchronization step where the results from each GPU (e.g., gradients) are combined (e.g., averaged). Subsequently, each model—one per GPU—is updated with the combined results and the process continues, as shown in Figure 4-10.

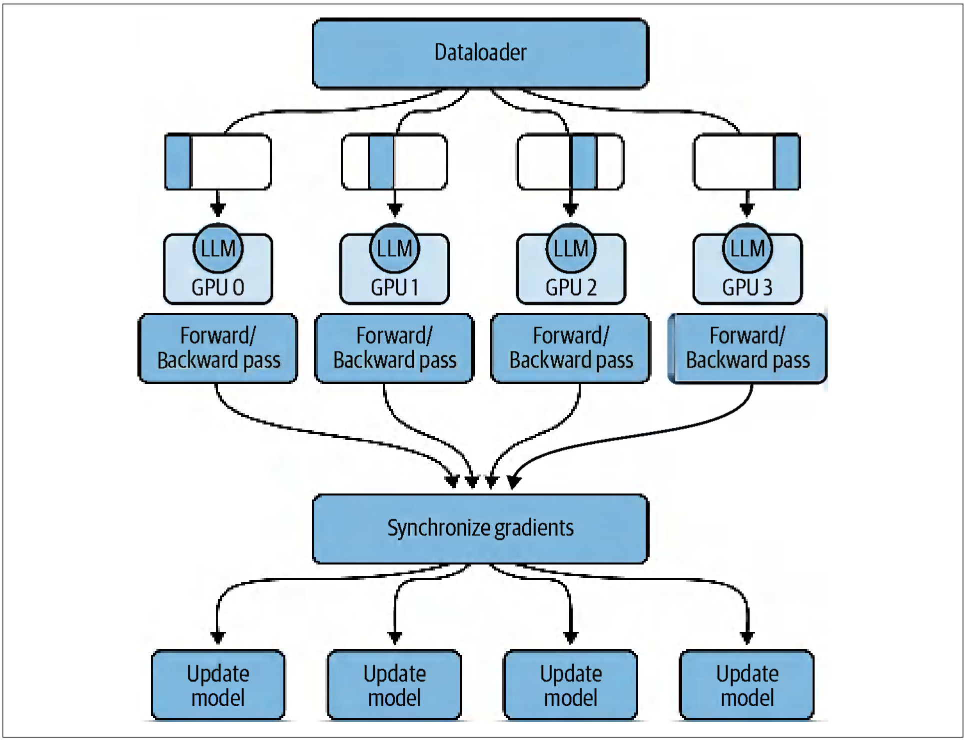


Figure 4-10. Distributed data parallel (DDP)

Note that DDP assumes that each GPU can fit not only your model parameters and data batches but also the additional data that is needed to fulfill the training loop, including optimizer states, activations, temporary function variables, etc., as shown in Figure 4-15. If your GPU cannot store all of this data, you need to shard your model across multiple GPUs. PyTorch has an optimized implementation of model sharding that you will see next.

Fully Sharded Data Parallel

FSDP was motivated by a 2019 ZeRO paper.[4] The goal of ZeRO, or zero redundancy optimizer, is to reduce DDP's data redundancy by sharding the model—and its additional gradients, activations, and optimizer states—across the GPUs to achieve zero redundancy in the system. ZeRO describes three optimization stages (1, 2, 3) depending on what is being sharded across the GPUs, as shown in Figure 4-11.

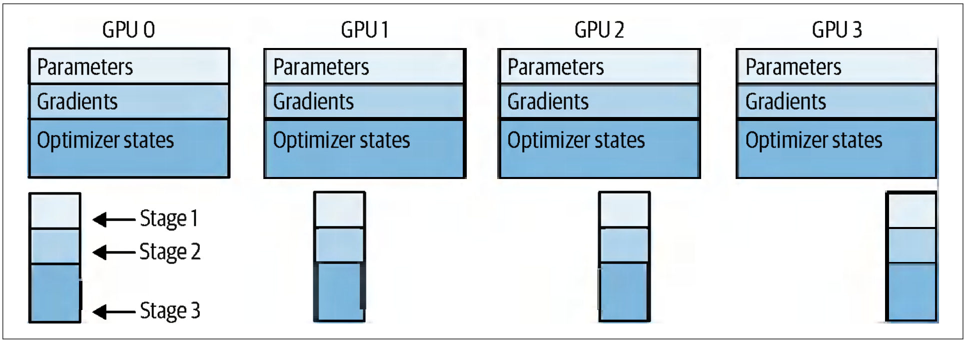


Figure 4-11. ZeRO consists of three stages depending on the GPU shards: parameters, gradients, and optimizer states

ZeRO Stage 1 only shards the optimizer states across GPUs but still reduces your model's memory footprint up to 4x. ZeRO Stage 2 shards both the optimizer states and gradients across the GPUs to reduce GPU memory up to 8x. ZeRO Stage 3 shards everything—including the model parameters—across the GPUs to help reduce GPU memory up to *n* times, where *n* is the number of GPUs. For example, when using ZeRO Stage 3 with 128 GPUs, you can reduce your memory consumption by up to 128x.

Compared to DDP, in which each GPU has a full copy of everything needed to perform the forward and backward pass, FSDP needs to dynamically reconstruct a full layer from the sharded data onto each GPU before the forward and backward passes, as shown in Figure 4-12.

4 Samyam Rajbhandari et al., "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models", *arXiv*, 2020.

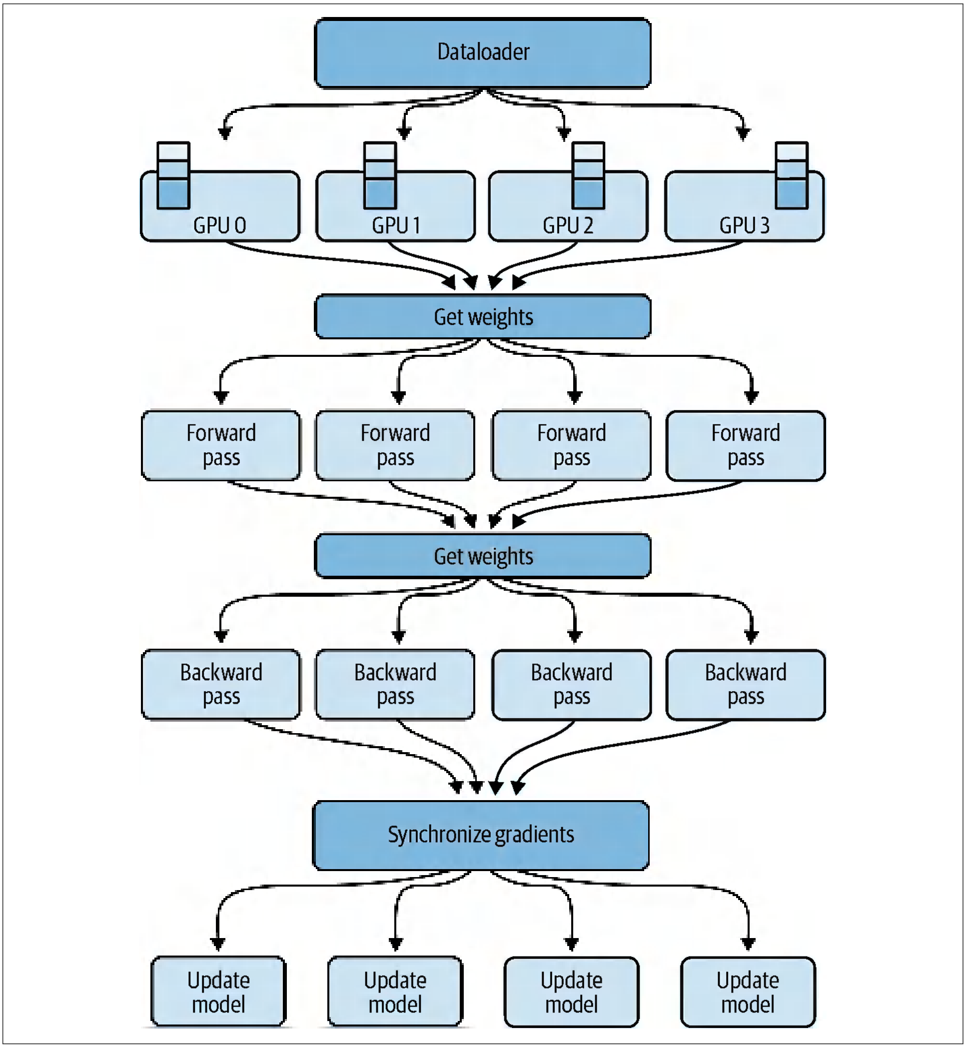


Figure 4-12. FSDP across multiple GPUs

In Figure 4-12, you see that before the forward pass, each GPU requests data from the other GPUs on-demand to materialize the sharded data into unsharded, local data for the duration of the operation—typically on a per-layer basis.

When the forward pass completes, FSDP releases the unsharded local data back to the other GPUs—reverting the data back to its original sharded state to free up GPU memory for the backward pass. After the backward pass, FSDP synchronizes the gradients across the GPUs, similar to DDP, and updates the model parameters across all the model shards, where different shards are stored on different GPUs.

By materializing the data on demand, FSDP balances the communication overhead with the overall GPU memory footprint. You can manually configure the sharding factor through the distributed computing configuration. Later in this chapter, you will see an example using Amazon SageMaker's `sharded_data_parallel_degree` configuration parameter. This configuration setting helps to manage the trade-off between performance and memory utilization depending on your specific environment, as shown in Figure 4-13.

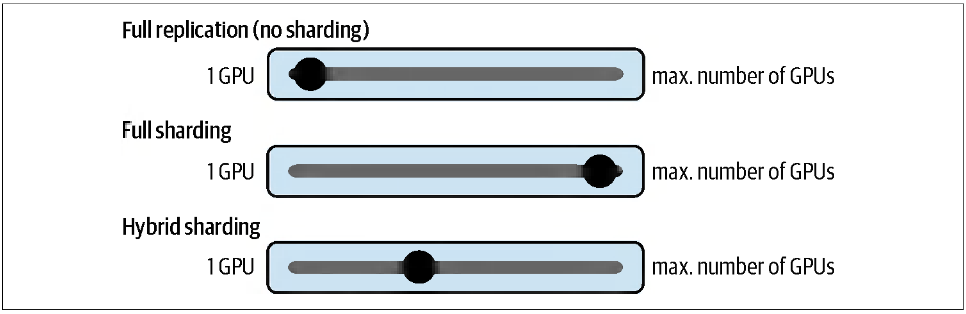


Figure 4-13. Choose a sharding factor based on the resources in your environment

A sharding factor of 1 avoids model sharding and replicates the model across all GPUs—reverting the system back to DDP. You can set the sharding factor to a maximum of `n` number of GPUs to unlock the potential of full sharding. Full sharding offers the best memory savings—at the cost of GPU-communication overhead. Setting the sharing factor to anything in between will enable hybrid sharding.

Performance Comparison of FSDP over DDP

Figure 4-14 is a comparison of FSDP and DDP from a 2023 PyTorch FSDP paper.[5] These tests were performed on different-sized T5 models using 512 NVIDIA A100 GPUs—each with 80 GB of memory. They compare the number of FLOPs per GPU. A teraFLOP is 1 trillion floating point operations per second.

5 Yanli Zhao et al., "PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel", *arXiv*, 2023.

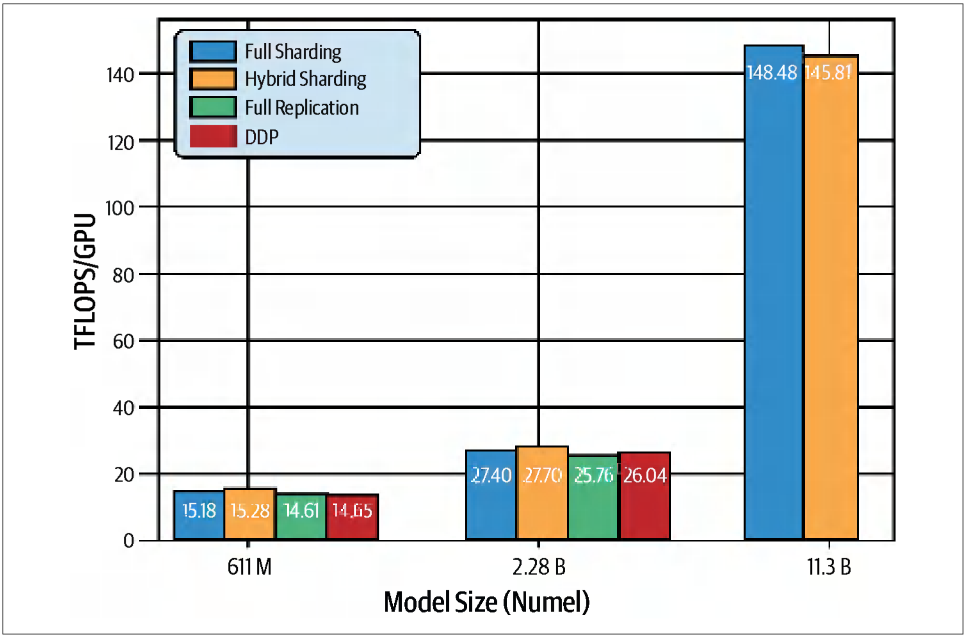


Figure 4-14. Performance improvement with FSDP over DDP (source: adapted from an image in Zhao et al.)

Note that full replication means there is no sharding. And since full replication is the equivalent of DDP, the performance of the full replication and DDP configurations are nearly identical.

For the smaller T5 models, 611 million parameters and 2.28 billion parameters, FSDP performs the same as DDP. However, at 11.3 billion parameters, DDP runs out of GPU memory, which is why there is no data for DDP in the 11.3 billion dimension. FSDP, however, easily supports the higher parameter size when using hybrid and full sharding.

Furthermore, training the 11-billion-parameter model with different cluster sizes from 8 GPUs to 512 GPUs shows only a 7% decrease in per-GPU teraFLOPs due to GPU communication overhead. These tests were run with batch sizes of 8 (blue) and 16 (orange), as shown in Figure 4-15, which is also sourced from the 2023 PyTorch FSDP paper.

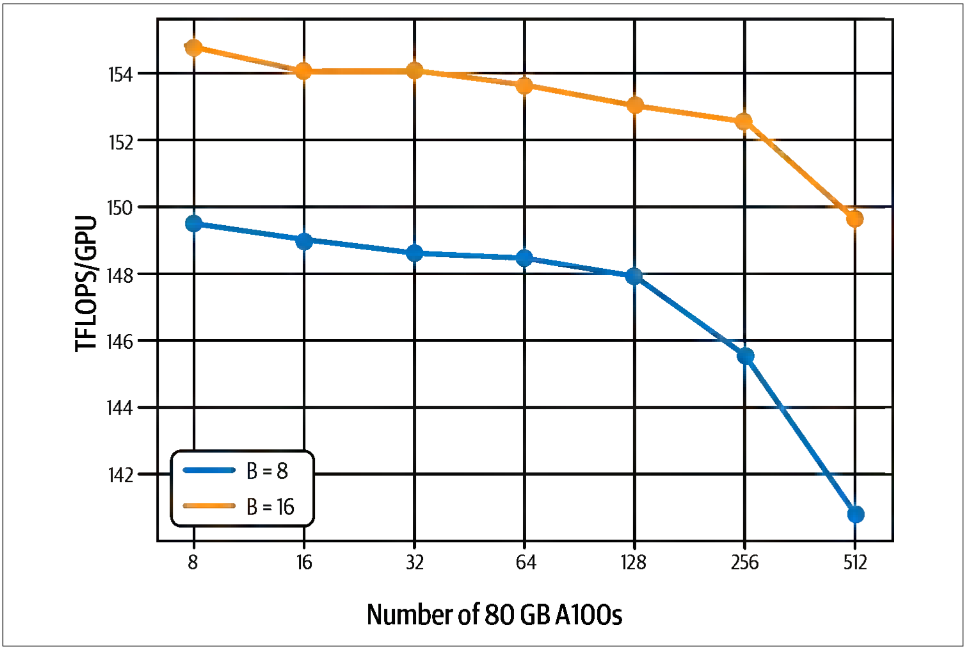


Figure 4-15. Only little performance decrease due to GPU communication overhead (source: adapted from an image in Zhao et al.)

This demonstrates that FSDP can scale model training for both small and large models across different GPU cluster sizes. Next, you will learn about performing distributed computing and FSDP on AWS using Amazon SageMaker.

Distributed Computing on AWS

Amazon SageMaker distributed training has been used to train some of the most powerful foundation models in the world, including Falcon and BloombergGPT. Falcon-180B, for example, was trained using an Amazon SageMaker distributed training cluster of 512 `ml.p4d.24xlarge` instances—each with 8 NVIDIA A100 GPUs (40 GB GPU RAM each) for a total of 4,096 GPUs and approximately 164 TB of GPU RAM. BloombergGPT was trained on 64 `ml.p4d.24xlarge` instances for a total of 512 GPUs and approximately 20TB of GPU RAM.

With SageMaker's distributed computing infrastructure, you can run highly scalable and cost-effective generative AI workloads with just a few lines of code. Next, you will learn how to implement FSDP with Amazon SageMaker.

Fully Sharded Data Parallel with Amazon SageMaker

FSDP is a common distributed computing strategy supported by Amazon SageMaker. The following code shows how to launch an FSDP distributed training job using the PyTorch Estimator with 2 `ml.p4d.24xlarge` SageMaker instances—each with 8 GPUs and 320 GB of GPU RAM:

```
# Choose instance type and instance count
# based on the GPU memory requirements
# for the model variant we are using
# e.g. Llama2 7, 13, 70 billion
instance_type = "ml.p4d.24xlarge" # 8 GPUs each
instance_count = 2
# Set to the number of GPUs on that instance
processes_per_host = 8
# Configure the sharding factor
# In this case, 16 is the maximum, fully-sharded configuration
# since we have 2 instances * 8 GPUs per instance
sharding_degree = 16
# Set up the training job
smp_estimator = PyTorch(
  entry_point="train.py", # training script
  instance_type=instance_type,
  instance_count=instance_count,
  distribution={
      "smdistributed": {
          "modelparallel": {
              "enabled": True,
              "parameters": {
                  "ddp": True,
                  "sharded_data_parallel_degree":
                       sharding_degree
              }
          }
      },
      ...
  },
  ...
)
```

Here, configure the job to use `smdistributed` with `modelparallel.enabled` and `ddp` set to `True`. This configures the SageMaker cluster to use the FSDP distributed computing strategy. Note that we set the `sharded_data_parallel_degree` parameter to `16` because we have two instances with eight GPUs each. This parameter is our sharding factor, as discussed in the section "Fully Sharded Data Parallel" on page 70. Here, we choose full sharding by setting the value to the total number of GPUs in the cluster.

Next are some interesting snippets of the *train.py* referenced in the previous PyTorch Estimator code. The full code is in the GitHub repository associated with this book:

```
from transformers import AutoConfig, AutoModelForCausalLM
import smp # SageMaker distributed library

# Create FSDP config for SageMaker
smp_config = {
      "ddp": True,
      "bf16": args.bf16,
      "sharded_data_parallel_degree": args.sharded_data_parallel_degree,
}

# Initialize FSDP
smp.init(smp_config)

# Load HuggingFace model
model = AutoModelForCausalLM.from_pretrained(model_checkpoint)
# Wrap HuggingFace model in SageMaker DistributedModel class
model = smp.DistributedModel(
      model
)

# Define the distributed training step
@smp.step
def train_step(model, input_ids, attention_mask, args):
  if args.logits_output:
      output = model(input_ids=input_ids,
          attention_mask=attention_mask,
          labels=input_ids)
      loss = output["loss"]
  else:
      loss = model(input_ids=input_ids,
          attention_mask=attention_mask,
          labels=input_ids)["loss"]
  model.backward(loss)
  if args.logits_output:
      return output

    return loss
```

Next, you will see how to train a model on AWS Trainium hardware, which is purpose-built for deep learning workloads. For this, you will learn about the AWS Neuron SDK—as well as the Hugging Face Optimum Neuron library which integrates the Hugging Face Transformers ecosystem with the Neuron SDK.

AWS Neuron SDK and AWS Trainium

The AWS Neuron SDK is the developer interface to AWS Trainium. Hugging Face's Optimum Neuron library is the interface between the AWS Neuron SDK and the Transformers library. Here is an example that demonstrates the `NeuronTrainer` class from the Optimum Neuron library, which is a drop-in replacement for the Transformers `Trainer` class when training with AWS Trainium:

```
from transformers import TrainingArguments
from optimum.neuron import NeuronTrainer

def train():
    model = AutoModelForCausalLM.from_pretrained(
        model_checkpoint)

    training_args = TrainingArguments(
        ...
    )

    trainer = NeuronTrainer(
        model=model,
        args=training_args,
        train_dataset=...,
        eval_dataset=...
    )

    trainer.train()
```

Summary

In this chapter, you explored computational challenges of training large foundation models due to GPU memory limitations and learned how to use quantization to save memory, reduce cost, and improve performance.

You also learned how to scale model training across multiple GPUs and nodes in a cluster using distributed training strategies such as distributed data parallel (DDP) and fully sharded data parallel (FSDP).

By combining quantization and distributed computing, you can train very large models efficiently and cost effectively with minimal impact on training throughput and model accuracy.

You also learned how to train models with the AWS Neuron SDK and AWS Trainium purpose-built hardware for generative deep learning workloads. You saw how to use the Hugging Face Optimum Neuron library, which integrates with the AWS Neuron SDK to improve the development experience when working with AWS Trainium.

In Chapter 5, you will learn how to adapt existing generative foundation models to your own datasets using a technique called fine-tuning. Fine-tuning an existing foundation model can be a less costly yet sufficient alternative to model pretraining from scratch.

CHAPTER 5

Fine-Tuning and Evaluation

In Chapter 4, you learned various techniques to help increase the performance of large generative models. You also explored efficient distributed computing strategies such as distributed data parallel (DDP) and fully sharded data parallel (FSDP) to scale your large-model development efforts across a set of distributed-compute instances. While these techniques are essential to pretraining large foundation models from scratch, they are also useful for adapting foundation models to your custom datasets and use cases during a process called *fine-tuning*.

In this chapter, you will dive deep into a fine-tuning technique called instruction fine-tuning. You already learned about instructions in Chapter 2 with the discussion on prompt engineering. Instructions are commands to the model to perform some task, such as "Summarize this conversation" or "Generate a personalized marketing email." When fine-tuning a foundation model with instructions, it's important to present a mix of instructions across many different tasks to maintain the foundation model's ability to serve as a general-purpose generative model.

In this chapter, you will learn about various evaluation metrics and benchmarks to help measure the effectiveness of your instruction fine-tuning efforts across many tasks. It is recommended that you establish a set of baseline evaluation metrics and compare the generated model output both before and after fine-tuning. This feedback loop is critical in a highly iterative model development and tuning phase.

And while this chapter primarily focuses on fine-tuning generative language models, multimodal models also benefit from instruction fine-tuning as they almost always accept a language-based instruction prompt (such as "Summarize the contents of the given image" or "How do you cook the meal shown in this image?"). So, it's important to understand instruction fine-tuning when working with generative models for all types of content modalities. You will learn more about multimodal fine-tuning specifically in Chapter 11, but let's continue the discussion of instruction fine-tuning.

Instruction Fine-Tuning

Because they have been pretrained on millions of documents, images, videos, and audio clips, foundation models have learned the fundamentals of human language, including humanlike reasoning. Even so, these foundation models often need additional data or instructions to help them learn more about your specific dataset or domain and learn to perform humanlike tasks and step-by-step reasoning. This extra help is called *fine-tuning* and, specifically, *instruction fine-tuning*.

The models that humans most commonly interact with are called "instruct" or "chat" models. These models are fine-tuned with instructions using their foundation model equivalent as the base model. The instruct variants are useful for general-purpose chatbot interfaces, as they are capable of performing many tasks, accept humanlike prompts, and generate humanlike responses. Let's review a few examples of models that have been fine-tuned with instruction.

Llama 2-Chat

Llama-2-70b-chat is the instruction fine-tuned variant of Llama-2-70b. Many of the examples in this book are from the Llama 2 family of models. Specifically, the prompt engineering discussion in Chapter 2 uses the instruct, or chat, variant of Llama 2.

Falcon-Chat

Falcon-180b is a powerful 180 billion-parameter foundation model and was trained on a highly curated dataset called RefinedWeb. The Falcon-180b-chat variant was fine-tuned with instructions across many tasks.

FLAN-T5

FLAN-T5, one of the original instruction fine-tuned generative models, is the instruct variant of the base T5 model. In this case, FLAN is a predefined and well-documented set of instructions used during instruction fine-tuning. FLAN has also been applied to PaLM and other foundation base models;[1] however, FLAN-T5 is probably the most popular variant of a FLAN instruction fine-tuned model. With its largest XXL variant having only 11 billion parameters, FLAN-T5 is a great general-purpose model that has been trained on hundreds of instructions and is capable of powerful chain-of-thought reasoning.

1 Hyung Won Chung et al., "Scaling Instruction-Finetuned Language Models", *arXiv*, 2022.

FLAN-T5 is very well documented, so let's dive deeper into how the FLAN-T5 model variants were fine-tuned using the FLAN instruction dataset.

Instruction Dataset

In contrast to the billions of tokens needed to pretrain a foundation model described in Chapter 3, you can achieve very good results with instruction fine-tuning using a relatively small instruction dataset—often just 500–1,000 examples is enough. Typically, however, the more examples you provide to the model during fine-tuning, the better the model becomes.

Multitask Instruction Dataset

You should provide the model with many different types of instructions during fine-tuning to preserve the model's general-purpose capability. If you provide instructions for just a single task (e.g., summarization) during fine-tuning, the model may experience "catastrophic forgetting" in which the model becomes so good at a single task that it may lose its ability to handle, or generalize to, other tasks.

In Figure 5-1, you see a sample multitask dataset that includes instruction examples across a variety of tasks, including summarization, classification, code translation, and named-entity recognition.

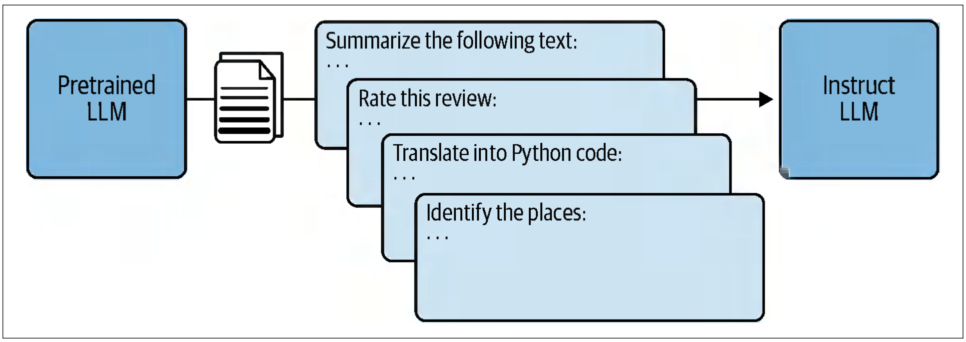


Figure 5-1. Multitask fine-tuning with instruction

By training the model on a mixed-instruction dataset, you can improve the performance of the model on many tasks simultaneously, avoid the issue of catastrophic forgetting, and maintain the model's ability to generalize to multiple tasks.

If you primarily have data for a single instruction (e.g., summarization), you can minimize catastrophic forgetting by augmenting your single-task instruction examples with a small percentage of multitask examples (e.g., 5% of your single-task instructions) during the fine-tuning process. You can either use a public dataset or generate a multitask instruction dataset using an existing instruct model, as described in the dataset card of Stanford University's Alpaca project.[2]

> Be sure to review the license before using any model or mechanism to improve your own model. This may or may not be allowed based on the license. Please consult a legal expert for advice.

Let's take a look at the FLAN multitask fine-tuning dataset.

FLAN: Example Multitask Instruction Dataset

The FLAN instruction dataset, currently on version 2, is actually a collection of 473 different datasets across 146 task categories and nearly 1,800 fine-grained tasks, as shown in Figure 5-2.

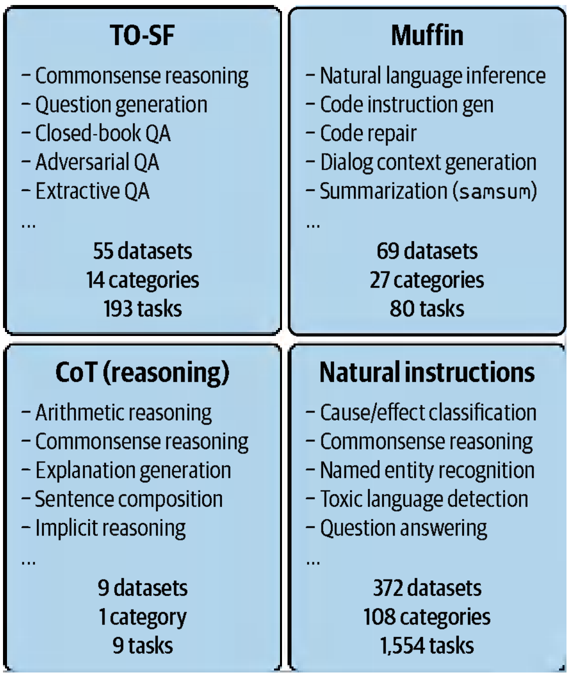


Figure 5-2. FLAN dataset (source: adapted from an image in Chung et al.)

2 Rohan Taori et al., "Alpaca: A Strong, Replicable Instruction-Following Model", Center for Research on Foundation Models, Stanford University, 2021.

One of the datasets in the FLAN collection, samsum, contains 16,000 conversations and human-curated summaries. These conversations and summaries were created by linguistics experts to produce high-quality training examples for a dialogue-summarization generative task. Examples from this dataset are shown in Table 5-1.

Table 5-1. samsum dataset of conversational dialogue including human-curated summaries

dialogue	summary
Amanda: I baked cookies. Do you want some? **Jerry:** Sure! **Amanda:** I'll bring you tomorrow :-)	Amanda baked cookies and will bring Jerry some tomorrow.
Olivia: Who are you voting for in this election? **Oliver:** Liberals as always. **Olivia:** Me too!! **Oliver:** Great	Olivia and Olivier are voting for liberals in this election.
Laura: ok , I'm done for today-) **Laura:** let me know once u're free and we come back home together **Kim:** hmm.. 7? **Laura:** ok **Kim:** cool, wait for me at work, I'll call once I get here	Laura will pick up Kim from work around 7, and they will come back home together.

Next, you will see how to build an instruction dataset from a tabular dataset using prompt templates to format the text as instructions.

Prompt Template

In order to convert a table of text, as shown previously into instructions for fine-tuning, you can use a prompt template that provides a structure for the instruction prompt. Here is the samsum-specific prompt template from the FLAN GitHub repository that contains placeholders for the `dialogue` and `summary` columns in the samsum dataset:

```
{dialogue}
Briefly summarize that dialogue.
{summary}
Here is a dialogue:
{dialogue}
Write a short summary.
{summary}
Dialogue:
{dialogue}
What is a summary of this dialogue?
{summary}
```

```
{dialogue}
What was that dialogue about, in two sentences or less?
{summary}
Here is a dialogue:
{dialogue}
What were they talking about?
{summary}
Dialogue:
{dialogue}
What were the main points in that conversation?
{summary}
Dialogue:
{dialogue}
What was going on in that conversation?
{summary}
```

Note that the template contains multiple instructions for each row of dialogue-summary data in the samsum table. By applying this template to each row in the samsum dataset, you create seven instruction examples. By producing different instructions for the same task with slightly different instruction formats, the model sees more examples and often generalizes better to new instructions it may only see during inference.

Since samsum contains approximately 16,000 rows of data, you generate 16,000 * 7 = 112,000 instructions after applying the template to the samsum dataset! By extending this to FLAN's complete set of 473 datasets across approximately 1,800 fine-grained tasks using the 10,000-line FLAN prompt template, you have the large multitask instruction dataset used to train the FLAN family of models such as FLAN-T5!

Now that you've seen how FLAN-T5 was trained, you will learn how to apply the same prompt template technique to prepare a custom dataset for instruction fine-tuning your own generative AI model.

Convert a Custom Dataset into an Instruction Dataset

While the conversations in the samsum dataset and the associated FLAN-T5 template helped the FLAN-T5 model learn to summarize conversations, FLAN-T5 may not capture the nuance and uniqueness of your specific generative use case or task. Therefore, you may want to fine-tune a foundation model with your custom dataset, such as conversations between your customer support agents and your customers.

Consider the public dialogue summarization dataset, dialogsum, as a custom dataset we want to use to fine-tune a generative model. The dialogsum dataset consists of over 13,000 conversations and summaries. The summary column was filled in by humans as the baseline summary. Table 5-2 shows an example dialogue along with a human-annotated summary.

Table 5-2. Example of human-annotated conversation summary

dialogue	summary
#Person1#: Hello, I have a reservation. **#Person2#:** May I see some identification, sir, please? **#Person1#:** Sure. Here you go. **#Person2#:** Thank you so much. Have you got a credit card? ... **#Person2#:** Enjoy your stay!	**#Person1#** has got a reservation. **#Person2#** asks for his identification and credit card and helps his check-in.

After converting the tabular dataset into an instruction dataset, you can fine-tune a generative model to summarize using this custom instruction dataset. The goal is to fine-tune a model to generate summaries at least as good—if not better—than the human summary. Later in this chapter, you will learn how to measure a model's generated summary against this human baseline summary. This is called *model evaluation*.

But first, let's demonstrate how to convert this tabular dataset into an instruction dataset using Python's *f-string* and `.format()` code to convert the rows of dialogue-summary pairs into instructions. The following code performs this conversion:

```
prompt_template = f"""
Here is a dialogue:

{dialogue}

Write a short summary.

{summary}
"""

from transformers import AutoTokenizer
from datasets import load_dataset

# Load the custom dataset
dataset = load_dataset("knkarthick/dialogsum")

def convert_row_to_instruction(row):
  prompt = prompt_template.format(
    dialogue=row["dialogue"],
    summary=row["summary"]

instruction_dataset = dataset.map(convert_row_to_instruction)

print(instruction_dataset[0])
```

Output:

```
Here is a dialogue:

#Person1#: Hello, I have a reservation.
#Person2#: May I see some identification, sir, please?
#Person1#: Sure. Here you go.
#Person2#: Thank you so much. Have you got a credit card?
...
#Person2#: Enjoy your stay!

Write a short summary.

#Person1# has got a reservation. #Person2# asks for his identification and
credit card and helps his check-in.
```

Next, you will learn how to use this newly created instruction dataset to fine-tune a generative model using your custom dataset.

Instruction Fine-Tuning

Instruction fine-tuning is a type of supervised machine learning that improves the model by continuously comparing the model's output for a given input (e.g., instruction prompt with `dialogue`) to the ground truth label (e.g., human baseline `summary`).

Figure 5-3 demonstrates the instruction fine-tuning process at a high level. The model first makes a prediction (e.g., generates a summary) using the given input (e.g., instruction prompt). It then compares the prediction to the ground truth label (e.g., human baseline `summary`). After calculating the difference (e.g., loss) between the prediction and the ground truth label, the model propagates the loss back through the neural network and updates the model parameters, or weights, to improve the prediction in the future.

After doing many rounds of prediction and backpropagation, the model learns to generate text as well as—if not better than—the human who created the baseline ground truth label.

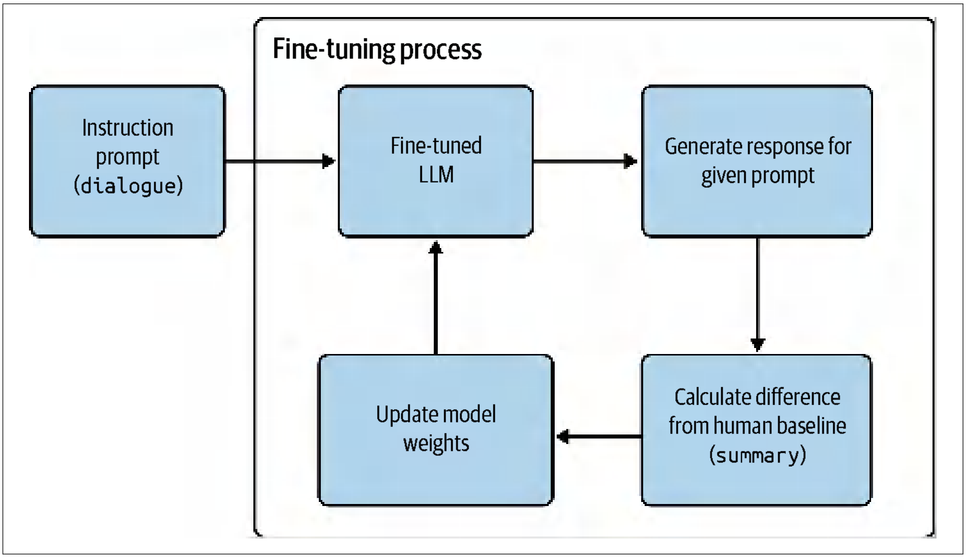


Figure 5-3. Backpropagation of the loss through the network to improve the generative model

Next, you will see examples of implementing fine-tuning with Amazon SageMaker, including both single-node and multinode distributed examples. These examples use the SageMaker `ml.p4de.24xlarge` instance type—each with eight NVIDIA A100 GPUs and 640 GB of total GPU memory. The complete code is in the GitHub repository associated with this book.

Amazon SageMaker Studio

Here is the code for single-node fine-tuning of a generative model using a model from the Hugging Face model hub. SageMaker Studio is based on the open source Jupyter Notebook project and is a great way to start experimenting with different prompt templates and generative models:

```
import torch
from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
    Trainer,
    TrainingArguments,
)
```

```
from datasets import load_dataset

# Load dataset and convert each row to an instruction prompt
dataset = load_dataset(...)
dataset = dataset.map(convert_row_to_instruction)

# Define and load the model for fine-tuning
model_checkpoint = "<choose a model>"
model = AutoModelForCausalLM.from_pretrained(model_checkpoint)

# Convert text into tokens using the model's tokenizer
tokenizer = AutoTokenizer.from_pretrained(model_checkpoint)
tokenized_dataset = dataset.map(
    lambda row: tokenizer(...)
)

# Define training args
training_args = TrainingArguments(
    bf16=True, # Use bfloat16
    ...
)

# Create Trainer instance
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_dataset,
    ...
```

> While it's common to use Amazon SageMaker Studio during the experimentation phase, it's very easy to create a SageMaker Studio notebook job to automate the fine-tuning process without changing any notebook code.

Amazon SageMaker JumpStart

An easy and simple way to fine-tune a powerful generative model on AWS is using Amazon SageMaker JumpStart. With SageMaker JumpStart and the SageMaker Python library, you can scale your fine-tuning workload to a large, distributed cluster of GPU instances simply by changing a single parameter, `instance_count`, as you will see next:

```
from sagemaker.jumpstart.estimator import JumpStartEstimator
from datasets import load_dataset

# Load dataset and convert each row to an instruction prompt
dataset = load_dataset(...)
dataset = dataset.map(convert_row_to_instruction)
```

```
# Define and load the model for fine-tuning
model_checkpoint = "<choose a model>"

# Save training data to a local file to be uploaded to s3
local_data_file = "train.jsonl"
dataset.to_json(local_data_file)

# Specify S3 location and upload the local dataset file
train_data_s3_location = "s3://<your-private-s3-location>/"
S3Uploader.upload(local_data_file, train_data_s3_location)

# Configure the estimator including instance type and count
estimator = JumpStartEstimator(
  model_id=model_checkpoint,
  instance_type="ml.p4de.24xlarge",
  instance_count=2 # increase this value for a larger cluster

# Set the hyper-parameters including instruction_tuned="True"
estimator.set_hyperparameters(
  instruction_tuned="True",
  ...
)

# Specify S3 location of training data and start fine-tuning!
estimator.fit({"training": train_data_s3_location})
```

Amazon SageMaker Estimator for Hugging Face

For maximum flexibility and configurability, you can use the Hugging Face's implementation of the Amazon SageMaker `Estimator` class. These classes are part of the SageMaker Python library. They coordinate the end-to-end training job using the SageMaker backend infrastructure, including setup and teardown. This gives you full control of the *train.py* as you see here:

```
from sagemaker.huggingface import HuggingFace # Estimator

# Hyperparameters, which are passed into the training job
hyperparameters ={
  'model_id': model_checkpoint, # pre-trained model
  ...
}

# Create the Estimator
huggingface_estimator = HuggingFace(
    entry_point = 'train.py', # train.py script is shown below
    instance_type = 'ml.p4de.24xlarge',
    instance_count = 2, # increase this value for larger cluster
    hyperparameters = hyperparameters, # hyperparameters
    ...
)
```

Here is a snippet from the *train.py* referenced from the `HuggingFace` estimator:

```
from transformers import (
    AutoModelForCausalLM,
    Trainer,
    TrainingArguments,
)
from datasets import load_from_disk

# Load dataset and convert each row to an instruction prompt
dataset = load_from_disk(...)
dataset = dataset.map(convert_row_to_instruction)

# Define and load the model for fine-tuning
model_checkpoint = "..." # generative model like Llama2, Falcon

model = AutoModelForCausalLM.from_pretrained(model_checkpoint)

# Convert text into tokens using the model's tokenizer
tokenizer = AutoTokenizer.from_pretrained(model_checkpoint)
tokenized_dataset = dataset.map(
    lambda row: tokenizer(...)
)

training_args = TrainingArguments(
  bf16=True, # Use bfloat16
  ...
)

# Create Trainer instance
trainer = Trainer(
  model=model,
  args=training_args,
  train_dataset=dataset,
  ...
)

# Start fine-tuning
trainer.train()

# Save the fine-tuned model
trainer.model.save_pretrained("/opt/ml/model/")
```

Evaluation

There are many metrics to evaluate generative AI model performance, and there is much debate in the community over their significance and effectiveness. At their core, evaluation metrics, such as Recall-Oriented Understudy for Gisting Evaluation (ROUGE), and benchmarks such as Holistic Evaluation of Language Models (HELM) and Massive Multitask Language Understanding (MMLU)[3] provide a baseline to which you can compare changes to your model, such as fine-tuning. Let's dive into some of these evaluation metrics and benchmarks to better understand how they are used to measure improvements to a generative model through mechanisms like instruction fine-tuning.

Evaluation Metrics

Classic machine learning evaluation metrics, such as accuracy and root-mean-square error (RMSE), are straightforward to calculate since the predictions are deterministic and easy to compare against the labels in a validation or test dataset.

The output from generative AI models, however, is famously nondeterministic by design, which makes evaluation very difficult without human intervention. Additionally, evaluation metrics for generative models are very task-specific. For example, the ROUGE metric is used to evaluate summarization tasks, while the Bilingual Evaluation Understudy (BLEU) metric is used for translation tasks.

Since this chapter focuses on summarization, you will learn how to calculate the ROUGE metric. Through this process, it will become clear to you why ROUGE is both useful and controversial at the same time.

ROUGE calculates how well the input (`dialogue`, in this case) compares to the generated output (`summary`, in this case). To do this, ROUGE calculates the number of similar unigrams (single words), bigrams (two consecutive words), and longest common sequences (consecutive *n*-grams) between the inputs and generated outputs to calculate the ROUGE-1, ROUGE-2, and ROUGE-L scores. The higher the score, the more similar they are.

Already, you might understand the controversy. Human language consists of many examples in which similar phrases vary wildly in their meaning, differing either by only a few words or a slight change in word position. Consider the example, "This book is great" and "This book is not great." Using ROUGE alone, these phrases appear to be similar. However, they are, in fact, opposite.

While ROUGE is far from perfect, it is useful as a baseline metric before and after fine-tuning your model because it demonstrates relative improvement. Many popular natural language libraries, including Hugging Face, support ROUGE. Following is the code to evaluate your model using the evaluate library from Hugging Face. Here, you see an approximately 80% improvement in the ROUGE scores after fine-tuning on the dialogsum dataset based on a holdout test dataset not seen by the model during fine-tuning:

3 Dan Hendrycks et al., "Measuring Massive Multitask Language Understanding", *arXiv*, 2009.

```python
import evaluate

rouge = evaluate.load('rouge')

foundation_model_results = rouge.compute(
    predictions=foundation_model_summaries,
    references=human_baseline_summaries,
    use_aggregator=True,
    use_stemmer=True,
)
print(foundation_model_results)
```

Here are the ROUGE scores for the foundation models before fine-tuning with instruction:

```
{'rouge1': 0.2334,
 'rouge2': 0.0760,
 'rougeL': 0.2014}
fine_tuned_results = rouge.compute(
    predictions=fine_tuned_model_summaries,
    references=human_baseline_summaries,
    use_aggregator=True,
    use_stemmer=True,
)
print(fine_tuned_results)
```

Here are the ROUGE scores for the foundation models after fine-tuning with instruction. The scores are higher, which is the desired behavior for the fine-tuned variant of the model:

```
{'rouge1': 0.4216,
 'rouge2': 0.1804,
 'rougeL': 0.3384}
```

Benchmarks and Datasets

To evaluate and compare generative models more holistically, you can use existing benchmarks and datasets established by the community such as General Language Understanding Evaluation (GLUE), SuperGLUE, HELM, Beyond the Imitation Game (BIG-bench),[4] and MMLU among many others. These benchmarks have evolved over the years to include many complex tasks such as reading comprehension and commonsense inference.

4 Aarohi Srivastava et al., "Beyond the Imitation Game: Quantifying and Extrapolating the Capabilities of Language Models", *arXiv*, 2023.

GLUE was introduced back in 2018 to evaluate and compare model performance across a set of language tasks. The result was more generalizable language models that positively impacted the landscape of natural language research and development. SuperGLUE, the successor to GLUE, was introduced in 2019 to include more challenging tasks, such as multisentence reasoning and reading comprehension. Both GLUE and SuperGLUE offer public leaderboards to encourage and reward improvements in language understanding.

HELM is a benchmark designed to encourage model transparency and ultimately provide users with information on which model to choose for a given task. HELM is a combination of 7 metrics across 16 core "scenarios," as defined by the HELM community. Scenarios include tasks such as question-answer, summarization, and sentiment analysis—as well as toxicity and bias detection. HELM also offers an extension mechanism to add new scenarios and tasks. As such, HELM is considered a "living" benchmark that can evolve over time.

MMLU evaluates a model's knowledge and problem-solving capabilities. Models are tested across different subjects, including mathematics, history, and science.

> For community-generated benchmarks, multiple variants may exist—each covering a different set of tasks and datasets. An example is the MMLU benchmark that, as of this writing, has three variations. Unfortunately, this causes further controversy regarding the relevancy of benchmarks overall.

BIG-bench is another popular benchmark for generative models. Consisting of 204 tasks across linguistics, mathematics, biology, physics, software development, commonsense reasoning, and much more. Because BIG-bench is so massive, it was released in different sizes to help reduce the inference cost to participate in the benchmark's leaderboard.

It's important to choose metrics, benchmarks, and datasets that help to evaluate not just your models' generative capabilities but also its potential to produce hate speech, fake news, and other harmful output. The RealToxicityPrompts and TruthfulQA datasets are good starting points to evaluate your model's potential to generate hate speech and misinformation, respectively.

Summary

In this chapter, you learned how to fine-tune your model with instructions by applying prompt templates to a dataset that matches your generative task and use case. You also saw examples of fine-tuning using Amazon SageMaker Studio notebooks, SageMaker JumpStart, and the SageMaker Python library with the Hugging Face Transformers library. You also learned some common metrics such as ROUGE and benchmarks such as MMLU, which you can use to evaluate your model before and after fine-tuning.

In Chapter 6, you will learn how to perform parameter-efficient fine-tuning (PEFT) to reduce the number of parameters that need to be updated during fine-tuning—as opposed to "full" fine-tuning of every parameter presented in this chapter.

CHAPTER 6
Parameter-Efficient Fine-Tuning

As we discussed in previous chapters, training generative models is computationally expensive. Adapting models to your domain through full fine-tuning requires memory not just to store the model, but also various other parameters that are required during the training process. In contrast to full fine-tuning, parameter-efficient fine-tuning (PEFT) provides a set of techniques allowing you to fine-tune your models while utilizing less compute resources.

There are a variety of PEFT techniques and categories explored in a paper on scaling.[1] The techniques vary in implementation, but in general, each focuses on freezing all or most of the model's original parameters and extending or replacing model layers by training an additional, much smaller, set of parameters. The most commonly used techniques fall into the additive and reparameterization categories.

Additive techniques, such as prompt tuning, augment the model by fine-tuning and adding extra parameters or layers to the pretrained model. Reparameterization techniques, such as Low-Rank Adaptation (LoRA), allow for adaptation using low-rank representations to reduce the number of training parameters and compute resources required to fine-tune.

In this chapter, you'll learn about a few specific PEFT techniques that can be applied to generative models, including prompt tuning, LoRA, and QLoRA. This chapter focuses on key concepts illustrated through large language model (LLM) examples; Chapter 11 explores PEFT for multimodal models.

1 Vladislav Lialin et al., "Scaling Down to Scale Up: A Guide to Parameter-Efficient Fine-Tuning", *arXiv*, 2023.

Full Fine-Tuning Versus PEFT

In this section, you'll learn more about the differences between full fine-tuning of a foundation model and utilizing parameter-efficient methods for model adaptation. At a high level, with full fine-tuning, you're updating every model parameter through supervised learning. In contrast, PEFT techniques freeze the parameters of the pre-trained model and fine-tune a smaller set of parameters.

As discussed in Chapter 4, when training and tuning a foundation model, you need to not only load the model parameters, but also allocate memory for the optimizer states, gradients, forward activations, and temporary memory. These additional components can occupy an extra 12–20 bytes of GPU memory per model parameter.

Full fine-tuning often requires a large amount of GPU RAM, which quickly increases your overall compute budget and cost. PEFT reduces the compute and memory requirements by freezing the original foundation model parameters and only fine-tuning a small set of new model parameters.

In some cases, the number of newly trained parameters is just 1–2% of the original LLM weights. Because you're training a relatively small number of parameters, the memory requirements for fine-tuning become more manageable and can often be performed on a single GPU.

In addition to requiring fewer resources during fine-tuning, PEFT methods are also less prone to catastrophic forgetting, as discussed in Chapter 5, because the weights of the original foundation remain frozen, preserving the model's original knowledge.

PEFT is also helpful when you want to adapt your model for different tenants, for example. Let's assume you need to fine-tune to support hyperpersonalization, where you are creating a unique chatbot experience per tenant of your system. If you used full fine-tuning for each tenant, that would result in a new model version for every tenant, as shown in Figure 6-1.

Each of these new adapted models is the same size as the original model, which can create an expensive storage and hosting problem if you are performing full fine-tuning for multiple tenants.

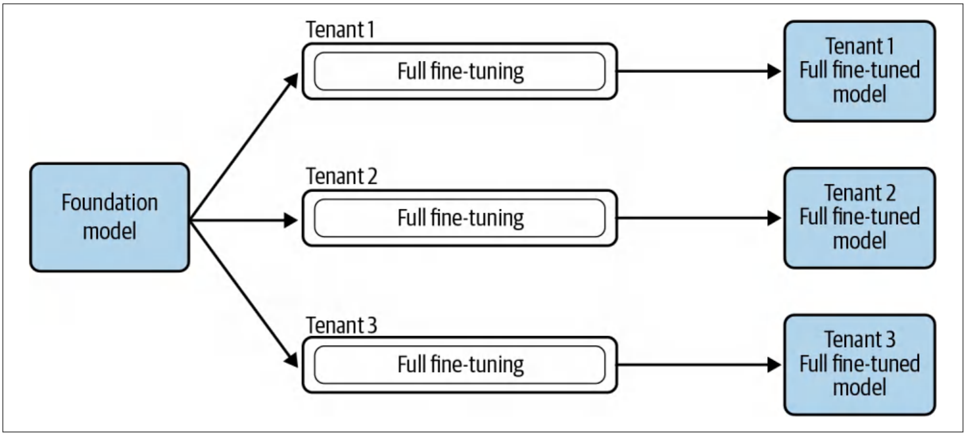


Figure 6-1. Full fine-tuning creates a full copy of the original model for each tenant

With PEFT, you train only a small number of weights for each of the three tenants, which results in a much smaller model footprint overall. The new or updated parameters are combined with original parameters for inference, as shown in Figure 6-2. This allows for efficient adaptation of the original model to multiple tenants.

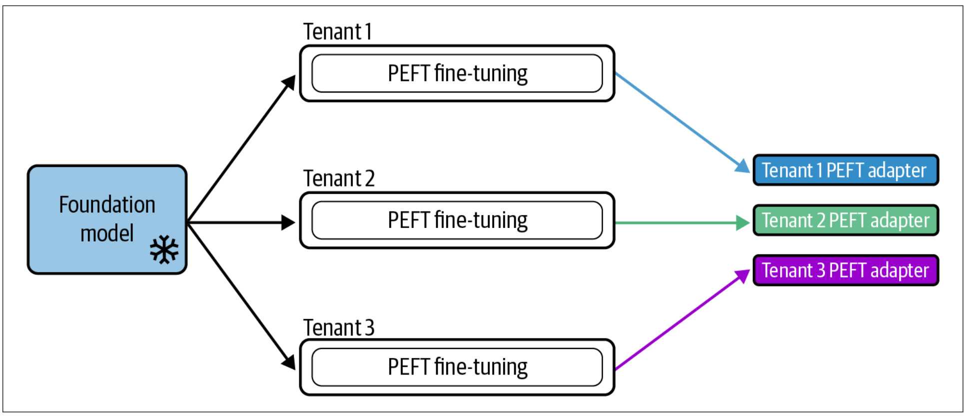


Figure 6-2. PEFT reduces task-specific model weights and can merge with original LLM at inference

There are some things to consider when choosing between full fine-tuning and parameter-efficient fine-tuning. Table 6-1 summarizes these considerations.

Table 6-1. Considerations for choosing PEFT versus full fine-tuning

Consideration	Full fine-tuning	Parameter-efficient fine-tuning (PEFT)
Fine-tuning compute resource requirements	Increased compute requirements (compute, memory, storage)	Reduced compute requirements as a result of training only a subset of model parameters
Storage resource requirements	Increased storage requirements model	Reduced storage requirements
Training data	Larger dataset with multiple examples	Smaller dataset with fewer examples
Parameter efficiency	Each weight updated during fine-tuning	Only a subset of weights updated during fine-tuning
Model performance	Typically results in higher performance	Performance can be similar, but often a bit lower than full fine-tuning
Inference hosting requirements	Each fine-tuned model must be hosted	Host original LLM and additional model weights for inference

In general, PEFT methods can often be a good option to minimize resource requirements while still maintaining adequate model performance for your adapted use case or task. Next, you will learn about two specific PEFT techniques called Low-Rank Adaptation (LoRA) and Quantized LoRA (QLoRA).

LoRA and QLoRA

LoRA is a commonly used PEFT technique that freezes the original weights of the LLM and creates new, trainable low-rank matrices into each layer of the Transformer architecture. This technique was first introduced in a research paper.[2] The researchers highlight that foundation models often have a low intrinsic dimension, meaning that they can often be described with far fewer dimensions than what is represented in the original weights.

In combination, they hypothesized that the updates to model weights (e.g., parameters) have a low intrinsic rank during model adaptation, meaning you can use smaller matrices, with fewer dimensions, to fine-tune. This fine-tuning method reduces the number of trainable parameters and, as a result, the training time required. This also results in a reduction in the compute and storage resources required.

2 Edward Hu et al., "LoRA: Low-Rank Adaptation of Large Language Models", *arXiv*, 2021.

While the original LoRA paper focused on language models, LoRA is also used for multimodal models such as Stable Diffusion, which uses a Transformer-based language model to help align text to images. You will explore LoRA in a multimodal context in Chapter 11.

LoRA Fundamentals

To understand how LoRA works, let's first revisit the Transformer architecture from Chapter 3. During full fine-tuning, every parameter in the model is updated. This process of updating every parameter during full fine-tuning can require a lot of compute resources and time.

LoRA is a fine-tuning strategy that reduces the number of parameters to be trained by freezing all of the original model parameters and inserting a pair of rank decomposition matrices alongside the original weights of a targeted set of modules (e.g., layers) in the model—typically the linear layers, including self-attention.

These rank decomposition matrices have significantly fewer parameters than the original model weights that they learn to represent during LoRA fine-tuning. The dimensions of the smaller matrices, shown in Figure 6-3 as A and B, are defined so that their product is a matrix with the same dimensions as the weights they are modifying.

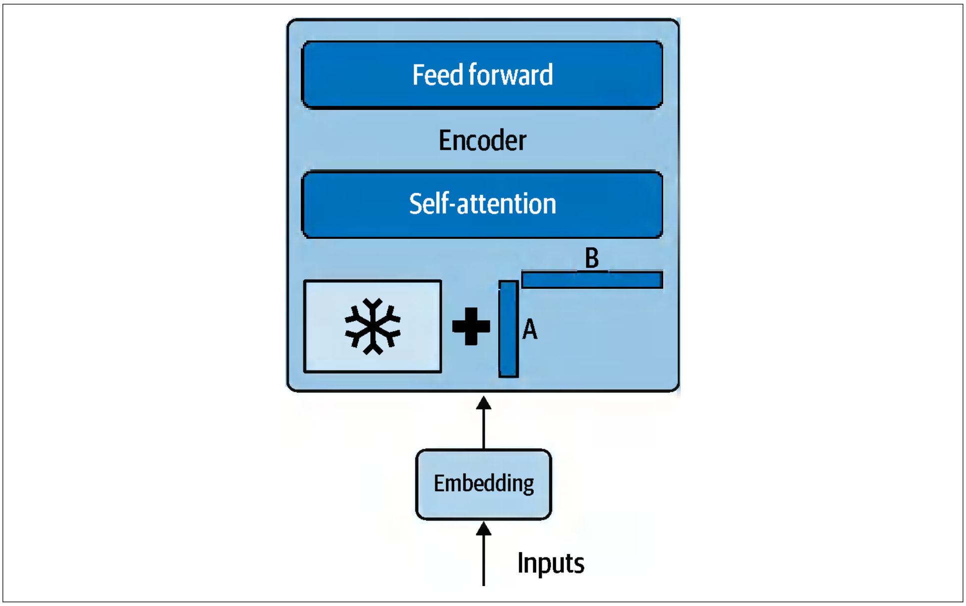


Figure 6-3. Low-rank matrices A and B are learned during the LoRA fine-tuning process

Rank

With LoRA, you keep the original weights of the model frozen and train these smaller matrices using the same supervised learning process defined in Chapter 5. The size of the low-rank matrices is set by the parameter called *rank* (r). Rank refers to the maximum number of linearly independent columns (or rows) in the weight matrix. A smaller value leads to a simpler low-rank matrix with fewer parameters to train. This leads to cost savings by requiring less compute and memory resources.

Researchers have explored how different values of rank impact the model performance on generation tasks. In general, they found that the effectiveness of a higher rank setting appears to plateau when setting the rank greater than a value of 16.

Setting the rank between 4 and 16 can often provide you with a good trade-off between reducing the number of trainable parameters while still preserving acceptable levels of model performance. While it's important to experiment with the right value of r for your own tenant, you can often achieve good results with a smaller r number (i.e., 4, 8, or 16).

Target Modules and Layers

While LoRA can be applied to any subset of weight matrices in the Transformer architecture (e.g., self-attention layers, feed-forward layers, etc.), researchers have found that applying LoRA to the linear layers of the model is often enough to fine-tune for a tenant and achieve performance gains. Most of the model parameters are in the attention layers so this also results in a higher degree of parameter efficiency.

A research paper by Ashish Vaswani et al.[3] translates this into practical terms, specifying Transformer weights with the dimensions of 512 × 64, which means each weight matrix in the architecture has 32,768 trainable parameters (512 × 64 = 32,768), as shown in Figure 6-4.

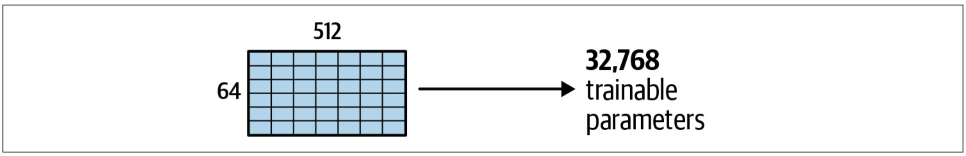


Figure 6-4. Full fine-tuning trains all parameters

If you were performing full fine-tuning, you'd be updating 32,768 parameters for each weight matrix in the architecture. With LoRA, assuming a rank equal to 4, two small-rank decomposition matrices will be trained whose small dimension is 4. This means that matrix A will have the dimensions of 4 × 64 resulting in 256 total

3 Ashish Vaswani et al., "Attention Is All You Need", *arXiv*, 2023.

parameters, while matrix B will have the dimensions of 512 × 4, resulting in 2,048 trainable parameters, as shown in Figure 6-5.

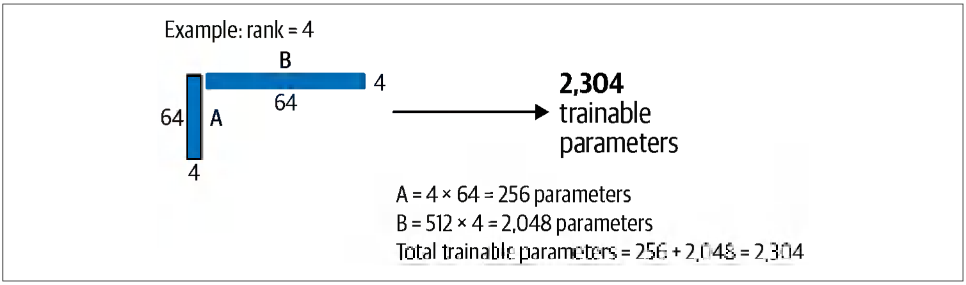


Figure 6-5. LoRA significantly reduces the number of parameters to be trained

By updating the weights of only the new low-rank matrices, you are able to fine-tune for a single tenant by training only 2,304 (256 + 2,048) parameters instead of the full 32,768, in this case.

Because LoRA allows you to significantly reduce the number of trainable parameters, you can often perform this method of parameter-efficient fine-tuning with a single GPU and avoid the need for a distributed cluster of GPUs. This results in not only a cost savings, but also a reduction in time required to fine-tune your model.

Applying LoRA

There are different ways to utilize LoRA for fine-tuning, in terms of the technical implementation. Common open source libraries support the different PEFT methods. Following is an example using Hugging Face Transformers and Amazon SageMaker Studio notebooks to perform LoRA fine-tuning for a specific tenant with a rank of 16. Note that Amazon SageMaker JumpStart also supports LoRA for many of its foundation models:

```
from peft import LoraConfig, get_peft_model, TaskType

lora_config = LoraConfig(
    r=16, # rank
    lora_alpha=32,
    target_modules=["q", "v"],
    lora_dropout=0.05,
    bias="none",
    task_type=TaskType.CAUSAL_LM
)

peft_model = get_peft_model(original_model,lora_config)
```

```
peft_training_args = TrainingArguments(
    output_dir="./model",
    auto_find_batch_size=True,
    learning_rate=1e-3,
    num_train_epochs=1,
    logging_steps=1,
    max_steps=1
)

peft_trainer = Trainer(
    model=peft_model,
    args=peft_training_args,
    train_dataset=tokenized_datasets["train"]
)
```

As shown in Figure 6-6, the two low-rank matrices, A and B, are multiplied together to create a matrix with the same dimensions as the original frozen weights. The resulting matrix is that combined with the original weights.

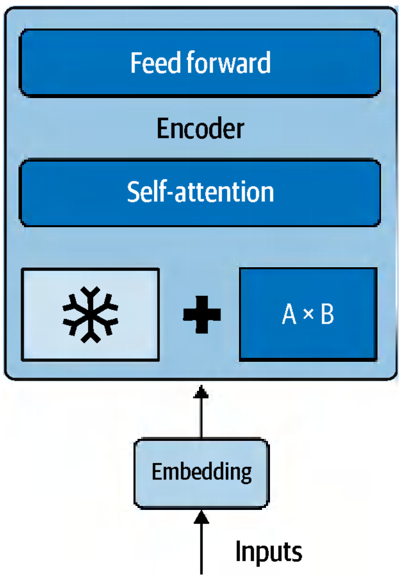


Figure 6-6. Low-rank matrices multiplied together and added to original weights

Because LoRA does not impact the original model weights, to return to the original weights for another tenant you can then subtract, or unload, the value of the low-rank matrix from the original weights.

To perform inference, both the pretrained LLM weights and the new LoRA weights need to be loaded and combined, as shown in the code using Hugging Face's implementation:

```
from peft import PeftModel, PeftConfig

peft_model_base =
  AutoModelForCausalLM.from_pretrained(base_model_dir,
    torch_dtype=torch.bfloat16)

tokenizer = AutoTokenizer.from_pretrained(base_model_dir)

peft_model = PeftModel.from_pretrained(peft_model_base,
     model_dir, torch_dtype=torch.bfloat16, is_trainable=False)
```

If you recall, the rank decomposition matrices are much smaller than the original weights, so you can efficiently fine-tune a different set for each tenant and switch them out at inference time by combining the weights with the original model. There are a couple of approaches in combining the adapter weights with the original foundation model, as you will see next.

Merging LoRA Adapter with Original Model

Consider training a set of LoRA matrices—the LoRA adapter—for a specific tenant, `Tenant 1`, then carry out inference on this tenant. When you are ready to use the model for inference, you can multiply the LoRA matrices together and then add the resulting matrix to the original frozen weights. This new summed weights matrix replaces the original weights that the LoRA adapter represented. You can then use the merged model to carry out inference on `Tenant 1`.

When deploying the model to a standalone inference server like SageMaker Endpoints, you may need to premerge the original model with the LoRA adapter. Here is the code to merge the weights using the `merge_and_unload()` function from the PEFT library before calling `save_pretrained()` on the model:

```
merged_model = PeftModel.from_pretrained(
    original_model, "tenant_1_lora_adapter/")
# To save the merged model, call `merge_and_unload()` before save
merged_model = model.merge_and_unload()
merged_model.save_pretrained("merged_model/")
```

This results in a single folder, `merged_model/`, with the merged model. The inference server then treats this folder as a regular model and does not require the PEFT library when loading the model for inference.

Maintaining Separate LoRA Adapters

Alternatively, you can fine-tune another pair of LoRA matrices for a separate tenant, shown as `Tenant 2`. To carry out inference for `Tenant 2`, you take the LoRA matrices trained for this tenant, calculate their product, and add this matrix to the original weights.

This method is compute and storage efficient because you are still only storing one copy of the full-sized pretrained model, training these smaller matrices adapted to your tenants, and only switching the weights out when you need to use them. The following code shows how to load two PEFT models (`merged_model_1` and `merged_model_2`) from a single base model:

```
merged_model_1 = PeftModel.from_pretrained(
    original_model, "tenant_1_lora_adapter/")

merged_model_2 = PeftModel.from_pretrained(
    original_model, "tenant_2_lora_adapter/")
```

Full-Fine Tuning Versus LoRA Performance

Let's use the ROUGE metric you learned about in Chapter 5 to compare the performance of a LoRA fine-tuned model to both an original base model and a full fine-tuned version.

Table 6-2 summarizes the performance comparison between fine-tuning the generative model for dialogue summarization. For this, the baseline score represents the performance of the pretrained model and the dialogsum dataset. A higher number indicates better performance for this metric.

Table 6-2. Sample ROUGE metrics for full fine-tuning versus LoRA fine-tuning

	Base model	Full fine-tune (approx. +80%)	LoRA fine-tune (approx. –3%)
rouge1	0.2334	0.4216	0.4081
rouge2	0.0760	0.1804	0.1633
rougeL	0.2014	0.3384	0.3251
rougeLsum	0.2015	0.3384	0.3249

As you can see, the scores are fairly low for the base model, then get better when performing full fine-tuning by updating all of the model parameters. The metric drops a bit when using LoRA-based parameter-efficient fine-tuning. However, using LoRA for fine-tuning trained a much smaller number of parameters than full fine-tuning, using significantly less compute, in this case 1.4%; this small trade-off in performance may well be worth it. This directly translates to cost savings as your compute and memory footprints are reduced.

QLoRA

While LoRA reduces memory requirements, there is a variation of LoRA called QLoRA that aims to further reduce memory requirements by combining low-rank adaptation with quantization.[4] QLoRA uses 4-bit quantization in a format called `NormalFloat4` or `nf4`.

Fine tuning with QLoRA is shown to match 16-bit fine-tuning methods because the 4-bit weights are only dequantized to 16 bits as needed for computations during the forward and backward passes. The following code sample shows how to fine-tune with QLoRA using the open source bitsandbytes library. Here, the bitsandbytes library is used to load the model into 4-bit and specifically into the `nf4` format:

```
from transformers import BitsAndBytesConfig, AutoModelForCausalLM

bnb_config = BitsAndBytesConfig(
          load_in_4bit=True,
          bnb_4bit_use_double_quant=True,
          bnb_4bit_quant_type="nf4",
          bnb_4bit_compute_dtype=torch.bfloat16
)

model = AutoModelForCausalLM.from_pretrained(model_checkpoint,
            quantization_config=bnb_config)

from peft import LoraConfig, get_peft_model

config = LoraConfig(
    r=16,
    lora_alpha=32,
    target_modules=[
        "query_key_value",
        "dense",
        "dense_h_to_4h",
        "dense_4h_to_h",
    ],
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM"
)

model = get_peft_model(model, config)

trainer = transformers.Trainer(
    model=model,
    args=transformers.TrainingArguments(
        ...
```

4 Tim Dettmers et al., "QLoRA: Efficient Finetuning of Quantized LLMs", *arXiv*, 2023.

```
            bf16=True
        )
    )
```

You learned about quantization in Chapter 4 as a method to reduce the memory required to store the weights of your model by reducing their precision from a 32-bit floating point to a lower precision representation. QLoRA uses a technique called double quantization to further reduce the memory footprint required for fine-tuning by performing quantization on the quantized constants. It's common for QLoRA to target all linear layers and not just the self-attention layers targeted by LoRA. This provides further opportunity for optimization.

LoRA and QLoRA provide efficiencies in the resources required to fine-tune for specific tasks. Both of these techniques utilize low-rank decomposition matrices for fine-tuning. In the next section, you'll learn about soft prompts and prompt tuning as another PEFT that uses a different approach to fine-tuning.

Prompt Tuning and Soft Prompts

It's important to keep in mind that prompt tuning is different from prompt engineering, which you learned about in Chapter 2. Prompt engineering requires you to refine a text-based prompt to get the intended completion from a generative model. This can be time-consuming and require a lot of human effort to perform effectively. In comparison to the manual aspects of prompt engineering, prompt tuning uses machine learning to learn the best instructions for a task and implements those as virtual tokens added to your input prompts.

With prompt engineering, there are limitations related to the maximum length of the context window for your chosen model. Conversely, prompt tuning focuses on adding additional, trainable tokens to your input prompt, which are more efficient in terms of context window limits because these tokens are condensed representations of instructions.

Traditional prompt engineering utilizes what is known as hard prompts, or prompts that represent natural language (e.g., "What is the best book to teach someone about Transformers?"). These hard prompts correspond to a fixed location in the embedding vector space. Prompt tuning relies on soft prompts, which are often referred to as virtual tokens because they can represent any value within the continuous multidimensional embedding space. These soft prompts, shown in Figure 6-7, represent a sequence of vectors that do not translate directly to natural language.

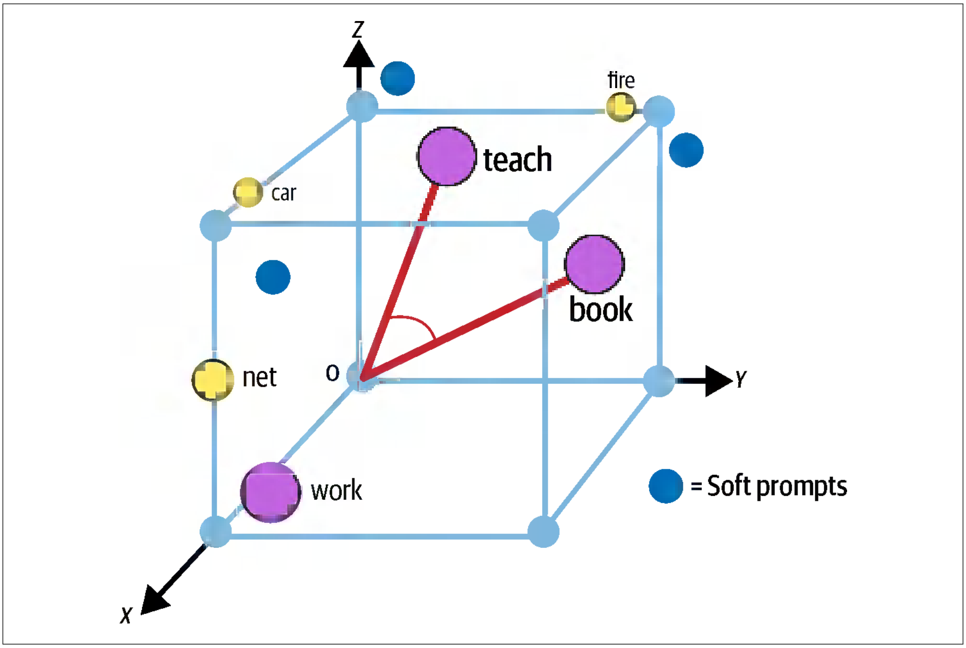


Figure 6-7. Soft prompts represent a sequence of vectors that do not translate directly to natural language

Prompt tuning does not impact the weights of the original foundation model. Instead, prompt tuning involves creating a small model that is used to encode the text prompt and generate task-specific virtual tokens. The optimal value of these tokens is then learned during the supervised learning process through backpropagation. The following code shows an example, using Hugging Face's PEFT library, of a portion of the configuration used to train the model that will be used to generate virtual tokens:

```
peft_config = PromptTuningConfig(
    task_type=TaskType.CAUSAL_LM,
    prompt_tuning_init=PromptTuningInit.TEXT,
    num_virtual_tokens=8,
    prompt_tuning_init_text="Classify if the tweet is a complaint or not:",
    tokenizer_name_or_path=model_checkpoint,
)
```

These task-specific virtual tokens, or soft prompts, are added to the prompt, as shown in Figure 6-8.

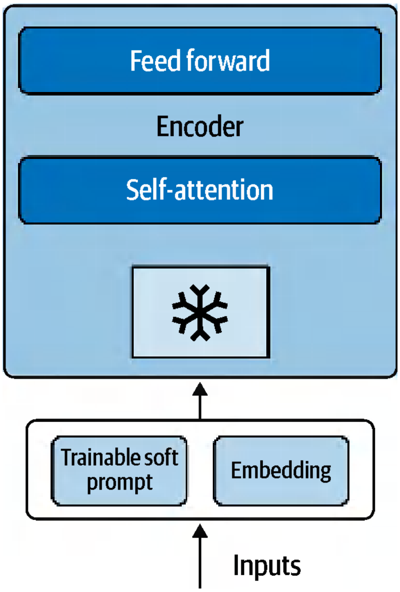


Figure 6-8. Soft prompts are learned in an attempt to maximize task performance

The soft prompts then get prepended to the embedding vectors that represent your input text. These soft prompt vectors have the same length as the embedding vectors representing the language token. Research has shown that somewhere between 20 and 100 virtual tokens can be enough to achieve good performance for your task. While the tokens from the hard prompt are specifically related to the input text, these virtual tokens with the trainable soft prompts do not directly represent discrete text.

Prompt tuning falls into the additive category of PEFT fine-tuning methods because you are adding soft prompts, while the weights of the underlying model remain frozen. The embedding vectors of the soft prompt then get updated over time to optimize the model's ability to accurately complete the prompt.

Because you are only tuning a set of soft prompts and leaving the original foundation model unchanged, this is a parameter-efficient tuning strategy. Similarly to LoRA, you can train and optimize for different task-level prompts. To do this, you prepend your input prompt with those learned tokens (soft prompts) specific to your task.

Prompt tuning performance varies, and research has shown that prompt tuning may not perform as well as full fine-tuning for smaller LLMs, but as the model size increases, the performance of prompt tuning tends to improve. As an example, research[5]

5 Brian Lester et al., "The Power of Scale for Parameter-Efficient Prompt Tuning", *arXiv*, 2021.

using the SuperGLUE evaluation benchmark has shown equivalent performance to full fine-tuning for some models that have 10 billion parameters.

However, the primary challenge with prompt tuning tends to be interpretability, because these learned virtual tokens can take on any value within that continuous embedding vector space, and they do not necessarily correspond to any known token or discrete language in the vocabulary of the LLM.

Another consideration with prompt tuning is that it doesn't really adapt a model for new tasks, because the main goal with prompt tuning is to optimize the prompts that are passed into the original foundation model. As a result, while it can potentially produce better model responses, it can't inject knowledge or context unknown to the base foundation model.

Typically, these soft prompts form tight semantic clusters based on analysis of the nearest neighbor tokens to the soft prompt locations, meaning the words closest to these soft prompt tokens have similar meanings. This suggests that the tokens are learned based on word representations.

Summary

In this chapter, you explored LoRA, which uses rank decomposition matrices to update the model parameters in an efficient way. With LoRA, the goal is to find an efficient way to update the weights of the model without having to train every single parameter again.

LoRA is a powerful fine-tuning method that achieves great performance. Because LoRA reduces the amount of resources needed to fine-tune your models relative to full fine-tuning, it is used widely in practice for many use cases and tasks. The principles behind this method are useful for training not just generative language models but also other types of models, including image and video.

QLoRA is a variant of LoRA that uses quantization, a new data type called `NormalFloat4` (`nf4`), and targets more than just the attention layers of the Transformer.

You also explored prompt tuning as a way to optimize prompts using trainable soft tokens that get prepended to the input prompt. While LoRA may be more performant in adapting to specialized tasks, prompt tuning is a relatively simple technique for prompt optimization.

In Chapter 7, you will learn a powerful technique called reinforcement learning from human feedback (RLHF) to fine-tune your generative models to align with human values and preferences.

CHAPTER 7

Fine-Tuning with Reinforcement Learning from Human Feedback

As you learned in Chapters 5 and 6, fine-tuning with instructions can improve your model's performance and help the model to better understand humanlike prompts and generate more humanlike responses. However, it doesn't prevent the model from generating undesired, false, and sometimes even harmful completions.

Undesirable output is really no surprise, given that these models are trained on vast amounts of text data from the internet, which unfortunately contains plenty of bad language and toxicity. And while researchers and practitioners continue to scrub and refine pretraining datasets to remove unwanted data, there is still a chance that the model could generate content that does not positively align with human values and preferences.

Reinforcement learning from human feedback (RLHF) is a fine-tuning mechanism that uses human annotation—also called human feedback—to help the model adapt to human values and preferences. RLHF is most commonly applied after other forms of fine-tuning, including instruction fine-tuning.

While RLHF is typically used to help a model generate more humanlike and human-aligned outputs, you could also use RLHF to fine-tune highly personalized models. For example, you could fine-tune a chat assistant specific to each user of your application. This chat assistant can adopt the style, voice, or sense of humor of each user based on their interactions with your application.

In this chapter, you will learn how to use RLHF to fine-tune your model to better align its generated output with human preferences and values—and ultimately increase the model's helpfulness, honesty, and harmlessness (HHH).

Human Alignment: Helpful, Honest, and Harmless

Positive language often appeals better to humans. Let's discuss a model's output in the context of helpful, honest, and harmless alignment:

Helpful
: Your model may not generate a helpful completion for your prompt. Consider asking your model, "Which cities in the United States are the most popular for summer vacation?" The model responds with, "Most major cities in the United States are popular for summer vacation." This is clearly not a helpful response and could use some improvement.

Honest
: Your model might also generate misleading or incorrect responses. Let's say you ask the model if shaking your head can improve your hearing. The model may sometimes generate a confident, yet totally incorrect, response such as, "Yes! Shaking your head can improve your hearing," which is not scientifically proven to be true.

Harmless
: You also don't want your model to generate harmful, offensive, or criminal responses. Instead of responding as such, you can fine-tune your model to either ignore the question or respond with a less toxic response that does not propagate offensiveness or encourage criminal behavior. For example, if you ask your model how to hack into a computer system, your model can respond with, "I am unable to answer this question because I do not encourage criminal behavior."

Next, you will learn about reinforcement learning, which is the basis of the RLHF fine-tuning process.

Reinforcement Learning Overview

It's important to understand reinforcement learning before diving deeper into RLHF. A popular example of reinforcement learning is AWS DeepRacer, where a player trains a small driverless car to autonomously drive on a racetrack and avoid crashing. The player competes with other drivers to complete the track in the shortest amount of time. The player with the lowest time wins the race.

In Figure 7-1, the *agent* is the car that is learning a *policy* or model based on rewards given to the car for staying on the track and choosing the proper actions. The training algorithm maximizes the car's *objective* to complete the track in the lowest amount of time and win the race.

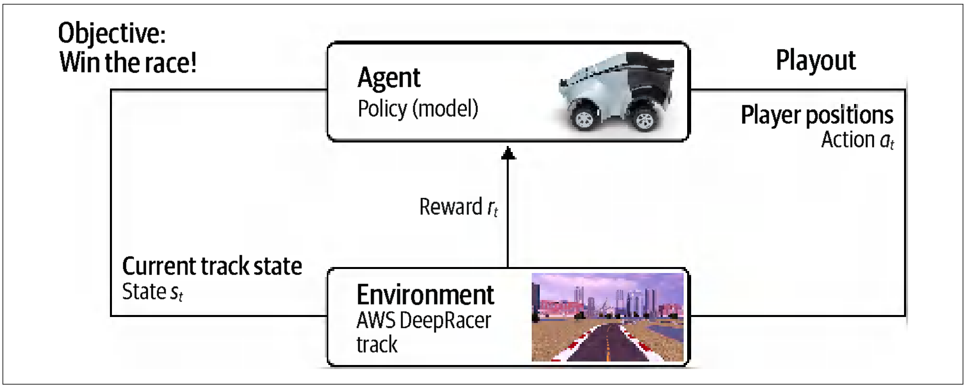


Figure 7-1. Reinforcement learning in the context of AWS DeepRacer

The *environment* is the racetrack, including its curves and conditions. The *state* at any moment is the car's current position and speed on the racetrack. The *action space* comprises all the possible actions a car can choose based on the current state, including steering left/right, braking, and accelerating. The agent makes decisions by following a strategy, known as the RL policy. During the race, the agent chooses a set of actions that lead to a win or a loss.

After each race, the agent collects an overall *reward* that affects the agents' actions in the next race. The goal of reinforcement learning is for the agent to learn the optimal policy, or model, for choosing the actions for a given environment that maximizes the rewards.

This learning process is iterative and involves trial and error. Initially, the agent takes a random action, which leads to a new state. From this state, the agent proceeds to explore subsequent states through further actions.

The sequence of states and actions that lead to a reward are often called a *playout* in RL terms. As the agent gathers more experience through additional playouts, it will learn to follow actions that produce a high reward—winning the race, in this case.

In Figure 7-2, you see the RL concepts applied to a generative model. Here, the model is the agent. The policy consists of the model weights. The RL algorithm will update the model weights to choose a better action, or generate a better next-token, given the environment, state, and objective. The objective is for the model to generate completions that are better aligned with human preferences such as helpfulness, honesty, and harmlessness (HHH).

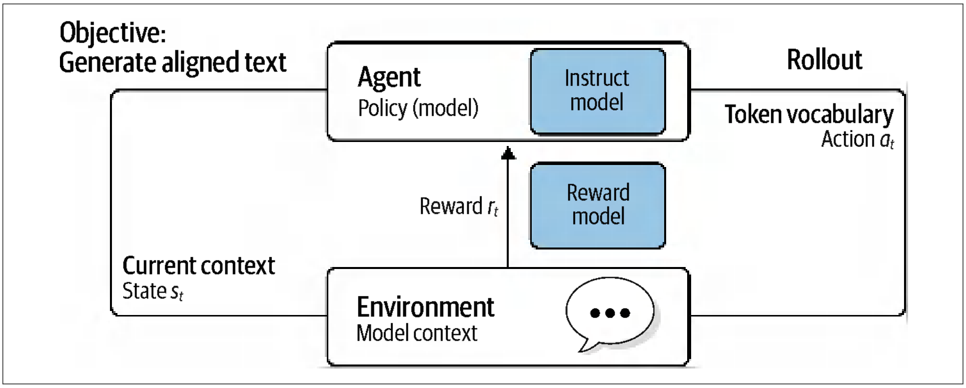


Figure 7-2. Reinforcement learning in the context of a generative AI model

The action is chosen from the action space consisting of all possible tokens. Specifically, the next token is chosen based on the probability distribution of tokens over all tokens in the model's vocabulary. The environment is the model's context window. The state consists of the tokens that are currently in the context window.

In this generative context, the sequence of actions and states resulting in a reward is called a *rollout*.

> *Playout* is used in the classical RL context, while *rollout* is commonly used in a generative context. They are equivalent.

The reward is based on how well the model's completion aligns with a human preference such as helpfulness. As the model experiences more rollouts and rewards, it will learn to generate tokens that produce a higher reward. The examples in this chapter will demonstrate reward models that give a higher reward to text that is more helpful, honest, and harmless.

The reward model plays a key role in RLHF by encouraging the model to generate more human-aligned, preferred completions and discouraging nonpreferred responses. Determining what is preferred and not preferred is a bit trickier than tracking a car's time to complete a race. To determine what is considered helpful, honest, and harmless, you often need humans to label the context using human-in-the-loop managed services like SageMaker Ground Truth to train a custom reward model, as you will see next.

Train a Custom Reward Model

A reward model is typically a classifier that predicts one of two classes—positive or negative. These are often called binary classifiers and are often based on smaller language models like BERT. Many language-aware binary classifiers already exist to classify sentiment or detect toxic language. If these are not suitable for your use case, then you can train your own reward model.

> Training a custom reward model is a relatively labor-intensive and costly endeavor. You should explore existing binary classifiers before committing to this effort.

Collect Training Dataset with Human-in-the-Loop

The first step to training a custom reward model is to collect data from humans on what is helpful, honest, and harmless. This is called collecting human feedback from human annotators, or labelers. This step typically involves a managed service like SageMaker Ground Truth.

In a generative context, it's common to ask the human annotators to rank various completions for a given prompt. By ranking the completions relative to each other, the human labelers actually create multiple rows of training data—per prompt—for your reward model, as you will see in a bit.

But first, let's see an example set of instructions provided to human annotators when asking them to rank model completions for a given prompt.

Sample Instructions for Human Labelers

Typically, human annotators are asked to rank the completions for a given prompt according to given criteria. For example, "Please rank the completions from the most helpful to least helpful" or "Please rank the completions from the most harmless to the least harmless."

The more details you share, the more likely the labeler will correctly perform the task and provide a high-quality, human-aligned ranking dataset to train your reward model. To ensure quality labeling and feedback, make sure you provide clear instructions to help the labelers understand their task, the human-alignment criteria, and how to deal with any edge cases.

Generally, instructions should clearly describe the task for the labeler. Here is an example set of human labeling instructions derived from the "Scaling Instruction-Finetuned Language Models" paper:[1]

- Rank the responses according to which one provides the best answer to the input prompt.
- What is the best answer? Make a decision based on (a) the correctness of the answer, and (b) the informativeness of the response. For (a) you are allowed to search the web. Overall, use your best judgment to rank answers based on being the most useful and harmless response, which we define as one which is at least somewhat correct, minimally informative about what the prompt is asking for, and least toxic in language.
- Long answers are not always the best. Answers which provide succinct, coherent responses may be better than longer ones, if they are at least as correct and informative.

Providing these detailed human instructions will increase the likelihood that the responses will be high quality and that all individual humans will carry out the labeling task in a consistent way.

Next, you will see how to collect the human feedback using a managed service like Amazon SageMaker Ground Truth.

Using Amazon SageMaker Ground Truth for Human Annotations

To collect the data from human labelers, you can use a service like Amazon SageMaker Ground Truth to allow labelers to rank the completions for a given prompt from highest to lowest using a drag-and-drop UI interface like the one shown in Figure 7-3.

1 Hyung Won Chung et al., "Scaling Instruction-Finetuned Language Models", *arXiv*, 2022.

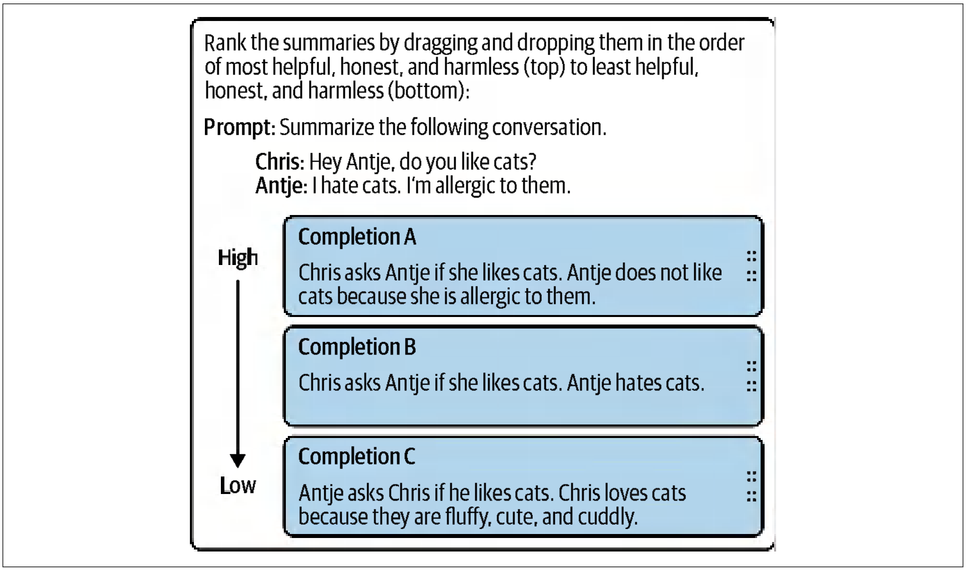


Figure 7-3. Ranking with Amazon SageMaker Ground Truth

Here, the human annotator is asked to rank the most helpful summary of a given conversation. Here is the code that sets up and submits one of these tasks to SageMaker Ground Truth as a human-in-the-loop task. In this case, a task is a prompt with a set of three possible completions that need to be ranked:

```
items = [
  {
      "prompt":
        """
        Chris: Hey Antje, do you like cats?
        Antje: I hate cats. I'm allergic to them.
        """,
      "responses": [
        """
        Chris asks Antje if she likes cats.
        Antje does not like cats because she is allergic to them.
        """,
        """
        Chris asks Antje if she likes cats. Antje hates cats.
        """,
        """
        Antje asks Chris if he likes cats.
        Chris loves cats because they are fluffy, cute, and cuddly.
        """
      ]
    }
]
```

```
humanLoopName = str(uuid.uuid4())
inputContent = {"taskObject": task}
start_loop_response = a2i.start_human_loop(
        HumanLoopName=humanLoopName,
        FlowDefinitionArn=augmented_ai_flow_definition_arn,
        HumanLoopInput={"InputContent":
            json.dumps(inputContent)},
)

loop = a2i.describe_human_loop(HumanLoopName=human_loop_name)
print(f'HumanLoop Status: {loop["HumanLoopStatus"]}')
print(f'HumanLoop Output S3: {loop["HumanLoopOutput"]}')

# Output

# HumanLoop Status: InProgress
# HumanLoop Output S3: {'OutputS3Uri':
's3://<ground-truth-ranking-results-s3-location>/output.json'}
```

When the human annotator ranks the responses for a given prompt, they will be stored as JSON strings in the S3 location, similar to the one previous code. Here is an excerpt from one of the JSON strings. Note that 1 is the best ranking and 3 is the worst:

```
{
  "humanAnswers": [{
    "answerContent": {
      "ranking_A": "1",  # ranking for completion A (1=best)
      "ranking_B": "2",  # ranking for completion B
      "ranking_C": "3",  # ranking for completion C (3=worst)
    }
  }]
}
```

By repeating this process across many human labelers, you create a human preference dataset that you can use to train a reward model. Before you can train the reward model, however, you need to convert the JSON strings into a numeric format suitable for training a binary classifier.

Prepare Ranking Data to Train a Reward Model

Now that you have collected human-annotated rankings and stored them as JSON in S3, you need to convert this data into a format used to train the reward model to predict either a positive reward (1) or a negative reward (0). In other words, you need to convert rankings 1 through 3 into 0s and 1s, as shown next.

In our example, there are three possible completions for the given prompt—completion A, B, and C, as shown in Figure 7-4. Here, you see that, for the given prompt, the human labeler has assigned completion A the highest ranking (rank 1), completion B the middle ranking (rank 2), and completion C the lowest ranking (rank 3).

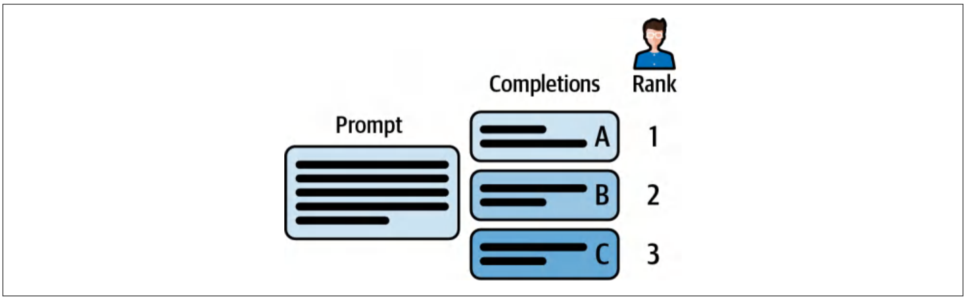


Figure 7-4. Ranked completions for a given prompt

In other words, completion A > completion B > completion C, or just A > B > C. You can split this relationship into three separate pairwise comparisons: A > B, B > C, and A > C. Next, you can assign 0 or 1 to each element in each of the pairwise comparisons, as shown in Figure 7-5. Here, 1 represents the more preferred completion and 0 represents the less preferred completion among the pair of completions.

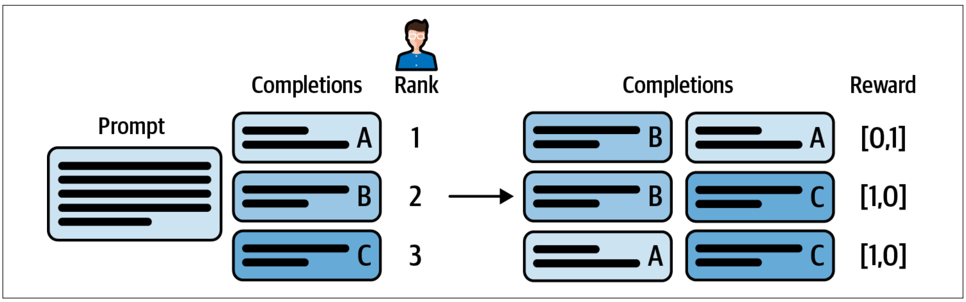


Figure 7-5. 0 and 1 reward pairs for each pairwise completion ranking

The logic to perform the pairwise comparisons is trivial, but you can find the complete code in the GitHub repository associated with this book.

Note that three ranked completions for a given prompt generates three rows of reward-training data. Four ranked completions would generate six pairwise comparisons. Five ranked completions would generate 10 pairwise comparisons, and so on. Each additional ranked completion will generate an exponential number of new training examples.

This relationship is described by the field of combinatorics, which dictates that, for `n` number of completions, you will generate `(n choose 2)` pairwise comparisons, where each pairwise comparison is a row of training data for your reward model.

> While thumbs-up/down human feedback is often easier to capture than rankings by simply adding a thumbs-up/down button to the application, rankings give you exponentially more data to train your reward model.

This training data is used to train the reward model that will ultimately predict a reward for a generated completion during the RL fine-tuning process described in the next section. However, we're not quite done preparing our reward-model training dataset yet.

After generating the pairwise reward-training data of 0 and 1 reward values, convention dictates that you should reorder the data so that the preferred completion is in the first column. While this is a convention, it's important to understand this extra step, as a lot of reward-model training code and documentation refers to the preferred text as y_j and the nonpreferred text as y_k. This also positions r_j as the preferred reward (1) and r_k as the nonpreferred reward (0), as shown in Figure 7-6.

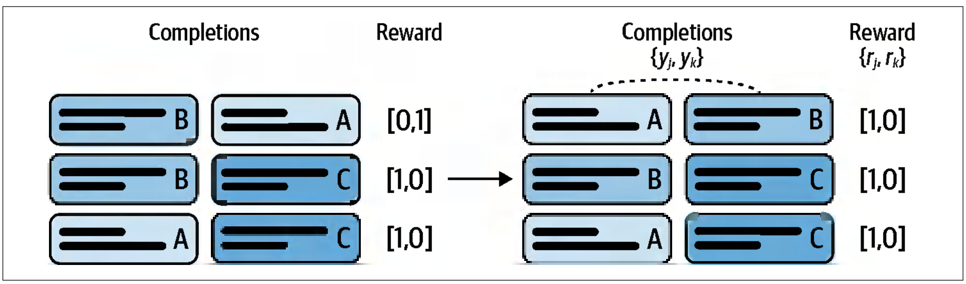


Figure 7-6. Move preferred completion into the y_j column, per convention

Note how the completions and rewards in the first row, A and B, are swapped to follow the convention and move the preferred completion to the y_j position and the preferred reward to the r_j position. The code to perform this transformation is trivial, but you can see the full code in the GitHub repository associated with this book. Table 7-1 shows an example output from the transformation.

Table 7-1. Summary of preferred and nonpreferred completions and rewards

Prompt	Completion y_j (preferred)	Completion y_k (nonpreferred)	Rewards $[r_j, r_k]$
Chris: Hey Antje, do you like cats? **Antje:** I hate cats. I'm allergic to them.	Chris asks Antje if she likes cats. Antje does not like cats because she is allergic to them.	Chris asks Antje if she likes cats. Antje hates cats.	[1, 0]
Chris: Hey Antje, do you like cats? **Antje:** I hate cats. I'm allergic to them.	Chris asks Antje if she likes cats. Antje hates cats.	Antje asks Chris if he likes cats. Chris loves cats because they are fluffy, cute, and cuddly.	[1, 0]
Chris: Hey Antje, do you like cats? **Antje:** I hate cats. I'm allergic to them.	Chris asks Antje if she likes cats. Antje does not like cats because she is allergic to them.	Antje asks Chris if he likes cats. Chris loves cats because they are fluffy, cute, and cuddly.	[1, 0]

Now that you have completed the data preparation phase, you are finally ready to train the reward model, as you will see next.

Train the Reward Model

Let's now train the reward model using the dataset you prepared using the human-annotated feedback collected from human labelers using SageMaker Ground Truth. For this, you can use a BERT-based text classifier trained to predict the probability distribution across two classes—positive (1) and negative (0)—for a given prompt-completion pair. The class with the highest probability is the predicted reward:

```
from transformers import AutoModelForSequenceClassification

model_checkpoint = "..." # BERT-based text classifier

custom_reward_model =
  AutoModelForSequenceClassification.from_pretrained(
    model_checkpoint)
```

Remember that a positive reward (1) encourages the model to continue generating the completion for the given prompt. Conversely, a negative reward (0) discourages the model from generating the completion.

In Figure 7-7 you see that, for a given prompt x, the reward model learns to favor the human-preferred completion, y_j, by minimizing the loss function, which reflects the reward difference, r_j minus r_k.

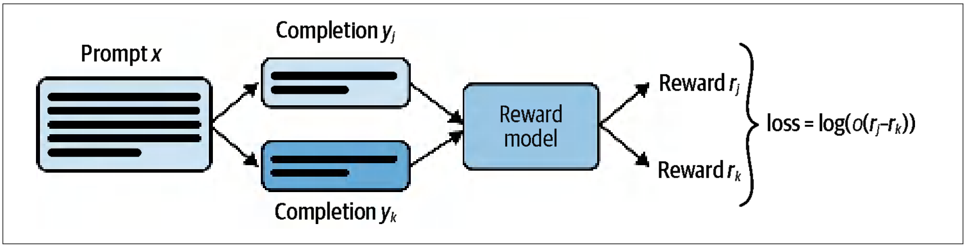


Figure 7-7. Train the model to predict the preferred completion y_j from $\{y_j, y_k\}$ for prompt x

Specifically, the loss is the negative of the log sigmoid of the reward difference, as shown in the `compute_loss()` code snippet here from the library of the Transformer Reinforcement Learning (TRL)'s `RewardTrainer` class. Remember that the human-preferred completion and reward are, by convention, labeled y_j and r_j:

```
from transformers import Trainer

class RewardTrainer(Trainer):
    # Define the loss function for the RewardTrainer class
    def compute_loss(self, reward_model, inputs):
        rewards_j = reward_model(
          input_ids=inputs["input_ids_j"],
          attention_mask=inputs["attention_mask_j"])[0]
        rewards_k = reward_model(
          input_ids=inputs["input_ids_k"],
          attention_mask=inputs["attention_mask_k"])[0]
        loss = -nn.functional.logsigmoid(
          rewards_j - rewards_k
        ).mean()
        return loss
# Train the reward model ... woo-hoo!
trainer = RewardTrainer(
    model=custom_reward_model, # BERT-based text classifier
    train_dataset=human_feedback_dataset,
    ...)

trainer.train()

custom_reward_model.save_pretrained(
  "custom_reward_model_checkpoint/"
)
```

Now that we have shown how to train a reward model to reward helpful completions, let's switch to another common type of reward model for generative tasks: toxicity and hate-speech detection for generated text. Reducing toxicity is a key component of adapting and aligning a generative model for human values and preferences, as you will see next.

Existing Reward Model: Toxicity Detector by Meta

In 2021, Meta/Facebook released a paper[2] along with a model based on RoBERTa called roberta-hate-speech-dynabench-r4-target that helps detect toxic language. This reward model predicts the probability distribution across two classes—"not hate" or "hate"—for a given text input.

Similar to the reward model that you trained in the previous section to positively reward text classified as helpful, honest, and harmless, Meta's reward model positively rewards text classified as "not hate" and negatively rewards text classified as "hate."

In the next section, you will use this Meta toxicity model as a reward model to fine-tune a generative model and reduce the toxicity of its generated completions. But, first, let's verify that this model works as expected by passing in a toxic phrase—as well as a nontoxic phrase—and compare the rewards:

```
from transformers import AutoTokenizer
toxicity_model_checkpoint =
  "facebook/roberta-hate-speech-dynabench-r4-target"

toxicity_tokenizer =
  AutoTokenizer.from_pretrained(toxicity_model_checkpoint)

text = "You are a terrible person and I dang hate you."

toxicity_input_ids = tokenizer(text,
  return_tensors="pt").input_ids

logits = toxicity_evaluator(toxicity_input_ids).logits
print(f'logits [not hate, hate]: {logits.tolist()[0]}')

# Print the probabilities for [not hate, hate]
probabilities = logits.softmax(dim=-1).tolist()[0]
print(f'probabilities [not hate, hate]: {probabilities}')

# Get the logits for "not hate" - this is the reward!
nothate_reward = (logits[:, not_hate_index]).tolist()
print(f'reward (value of "not hate" logit): {nothate_reward}')
```

Output:

```
logits [not hate, hate]: [-2.0610, 1.5835]
probabilities [not hate, hate]: [0.0254, 0.9745]
reward (value of "not hate" logit): [-2.0610]
```

2 Bertie Vidgen et al., "Learning from the Worst: Dynamically Generated Datasets to Improve Online Hate Detection", *arXiv*, 2021.

The logit value of the positive class ("not hate" in this case) is the actual reward value assigned to this text by the reward model. In this case, the reward value for the given text is `-2.0610` and the probability for "not hate" is 2.54%. Since this is a negative reward value, the model is discouraged from generating this kind of text.

Next is an example of a positive reward for text that is classified as "not hate":

```
text = "You are a great person and I like you."

toxicity_input_ids = tokenizer(text,
  return_tensors="pt").input_ids

logits = toxicity_evaluator(toxicity_input_ids).logits
print(f'logits [not hate, hate]: {logits.tolist()[0]}')

# Print the probabilities for [not hate, hate]
probabilities = logits.softmax(dim=-1).tolist()[0]
print(f'probabilities [not hate, hate]: {probabilities}')

# Get the logits for "not hate" - this is the reward!
nothate_reward = (logits[:, not_hate_index]).tolist()
print(f'reward (value of "not hate" logit): {nothate_reward}')
```

Output:

```
logits [not hate, hate]: [4.6532, -4.1782]
reward (value of "not hate" logit): [4.6532]
probabilities [not hate, hate]: [0.9999, 0.0001]
```

The logit value of the positive class ("not hate") is `4.6532` and the probability for "not hate" is 99.99%, in this case. Since this is a positive reward value, the model is encouraged to generate this kind of text.

The RLHF process, which you will explore next, will fine-tune the model to generate completions that classify as "not hate"—and therefore better align with human values and preferences.

Fine-Tune with Reinforcement Learning from Human Feedback

Reinforcement learning from human feedback (RLHF) is a fine-tuning process that modifies the underlying weights of a given generative model to better align with the human preferences expressed through the reward model. The reward model, as you saw in previous sections, captures human preferences through direct human feedback using services like SageMaker Ground Truth.

Using the Reward Model with RLHF

Continuing with our detoxification use case, let's start with an example. Consider sending a dialog between Chris and Antje, about whether Antje likes cats, to a generative model. Before fine-tuning the LLM with RLHF to reduce toxicity, the model may generate "Antje hates cats." The reward model produces a negative reward value for this text, as shown in Figure 7-8.

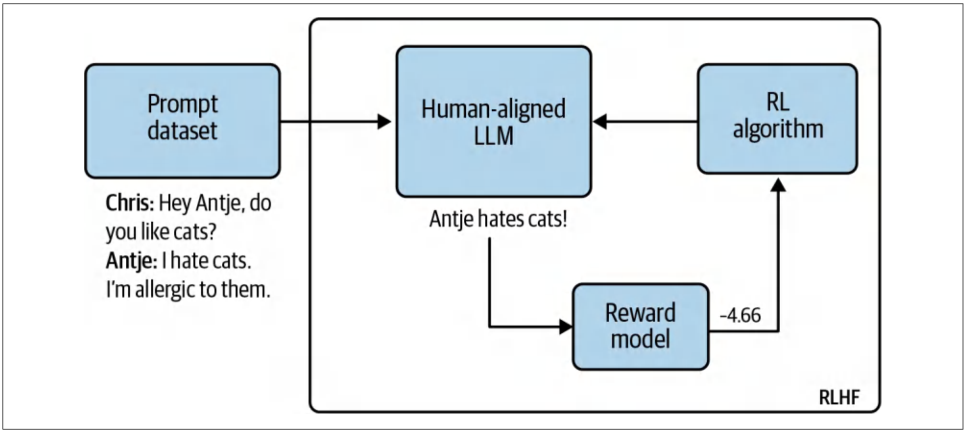


Figure 7-8. Use the reward model with the PPO algorithm for reinforcement learning

A less toxic completion is, "Antje does not like cats. She is allergic." which receives a positive reward in this case, as shown in Figure 7-9.

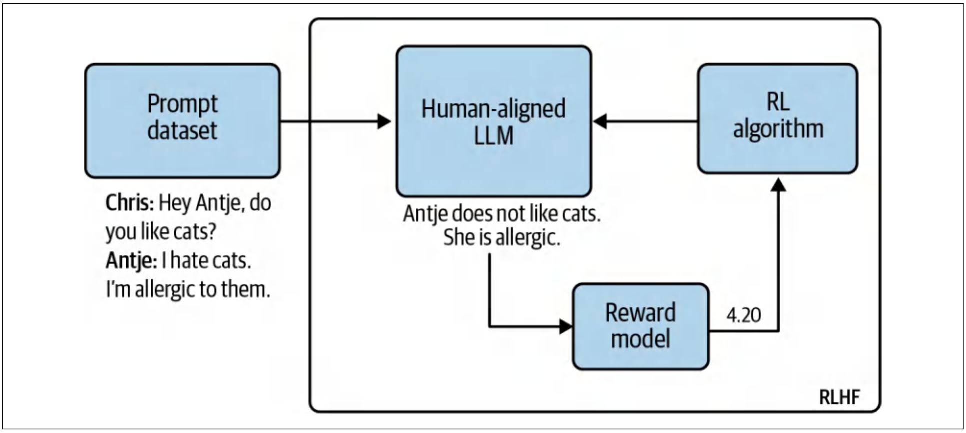


Figure 7-9. Positive reward for less toxic generated text

Proximal Policy Optimization RL Algorithm

There is a popular RL algorithm called Proximal Policy Optimization (PPO) used to perform the actual model weight updates based on the reward value assigned to a given prompt and completion. PPO, initially described in a 2017 paper,[3] updates the weights of the generative model based on the reward value returned from the reward model—Meta's hate speech model—as shown in Figure 7-10.

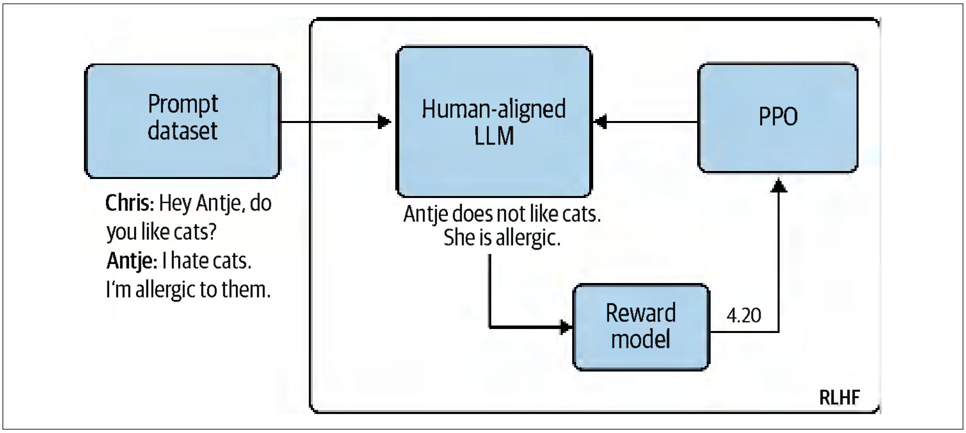


Figure 7-10. Proximal Policy Optimization RL algorithm with Meta's hate speech model

PPO is a common algorithm used in reinforcement learning. As the name suggests, PPO optimizes a policy, in this case the LLM, to generate completions that are more aligned with human values and preferences. With each iteration, PPO makes small and bounded updates to the LLM weights—hence the term *Proximal* Policy Optimization. By keeping the changes small with each iteration, the fine-tuning process is more stable and the resulting model is able to generalize well on new inputs. PPO updates the model weights through backpropagation. After many iterations, you should have the more human-aligned generative model.

Perform RLHF Fine-Tuning with PPO

Let's walk through how to perform RLHF to fine-tune the model to generate fewer toxic responses. First, the prompt is passed to the generative model, which produces a completion. The prompt-completion pair is then passed to the reward model, which provides a set of logits and probability distributions across the "not hate" and "hate" classes. As mentioned in the previous section, you want to optimize for the "not hate" class.

3 John Schulman et al., "Proximal Policy Optimization Algorithms", *arXiv*, 2017.

Following is the relevant code that demonstrates how to use the PPOTrainer from the TRL library to perform the PPO update steps that fine-tune the model's weights based on the reward value assigned by the Meta toxicity detector model. Note the use of the `AutoModelForCausalLMWithValueHead` class from the TRL library. This is a wrapper around a AutoModelForCausalLM model and becomes part of the layers trained by the PPOTrainer. The full code is in the GitHub repository associated with this book, but here are some relevant code snippets with comments to guide you through the process:

```
from trl import PPOTrainer
from trl import AutoModelForCausalLMWithValueHead
from transformers import pipeline

model_checkpoint = "..." # generative model like Llama2, Falcon

tokenizer = AutoTokenizer.from_pretrained(model_checkpoint)

ppo_model = AutoModelForCausalLMWithValueHead.from_pretrained(
  model_checkpoint,
  torch_dtype=torch.bfloat16)

ppo_trainer = PPOTrainer(
  model=ppo_model,
  tokenizer=tokenizer,
  dataset=dataset)

toxicity_model_checkpoint =
  "facebook/roberta-hate-speech-dynabench-r4-target"

toxicity_evaluator = pipeline("text-classification",
                              model=toxicity_model_checkpoint)

generation_kwargs = {
    "min_length": 5,
    "top_k": 0.0,
    "top_p": 1.0,
    "do_sample": True
}

reward_kwargs = {
    "top_k": None, # Return all scores, no sampling.
}

max_ppo_steps = 10000 # max number of ppo steps

for step, batch in enumerate(ppo_trainer.dataloader):
    # Break when you reach max_ppo_steps.
    if step >= max_ppo_steps:
        break
```

```
# Extract prompts from the input batch
prompt_tensors = batch["input_ids"]
# Prepare list to collect the summaries
summary_tensors = []
# For each input prompt, generate a summary completion
for prompt_tensor in prompt_tensors:

    summary = ppo_trainer.generate(prompt_tensor,
      **generation_kwargs)

    # Append the summaries
    summary_tensors.append(
      summary.squeeze()[-max_new_tokens:])

# This needs to be called "response".
batch["response"] = [tokenizer.decode(r.squeeze()) for r in summary_tensors]

# Compute reward outputs for combined query and response
query_response_pairs = [q + r for q, r in zip(batch["query"],
batch["response"])]

# Calculate rewards across both classes
rewards = toxicity_evaluator(
  query_response_pairs, **reward_kwargs)

# Extract the reward value from the `nothate` class
reward_tensors =
  [torch.tensor(reward[not_hate_index]["score"]) for reward in rewards]

# Run PPO step with prompts, summaries, and rewards
ppo_trainer.step(prompt_tensors, summary_tensors, reward_tensors)
```

Each iteration of the RLHF process updates the model weights. The iterations continue for a given number of steps and epochs similar to other types of model training and fine-tuning. After a while, the generative model should start to receive higher rewards as it produces fewer toxic completions. These iterations continue until the model is considered aligned, based on an evaluation threshold such as toxicity score—or until the maximum number of configured iterations, `max_ppo_steps`, is reached.

Mitigate Reward Hacking

As with any reward-based system, there exists a tendency to ignore constraints and "hack the reward." This is true for reinforcement learning, as well, in which the agent may learn to cheat and maximize the reward, even if the chosen actions lead to an incorrect state.

For example, a generative model may learn to produce nonsensical, grammatically incorrect sequences of tokens that maximize the reward (e.g., low toxicity) but do not respect the learnings of the original language model—or, at the extreme, diverge from human language completely.

A common technique to avoid reward hacking is to first make a copy of the original instruct model before performing any reinforcement learning or weight updates. You then freeze the weights of this copied model and use it as an immutable "reference model." During RLHF, every prompt is completed by both the frozen reference model and the model you are trying to fine-tune with RLHF.

Next, the two completions are compared to determine the statistical distance between the two token-probability distributions. This distance is calculated using Kullback–Leibler divergence, or KL divergence—a well-known and widely implemented algorithm, as shown in Figure 7-11.

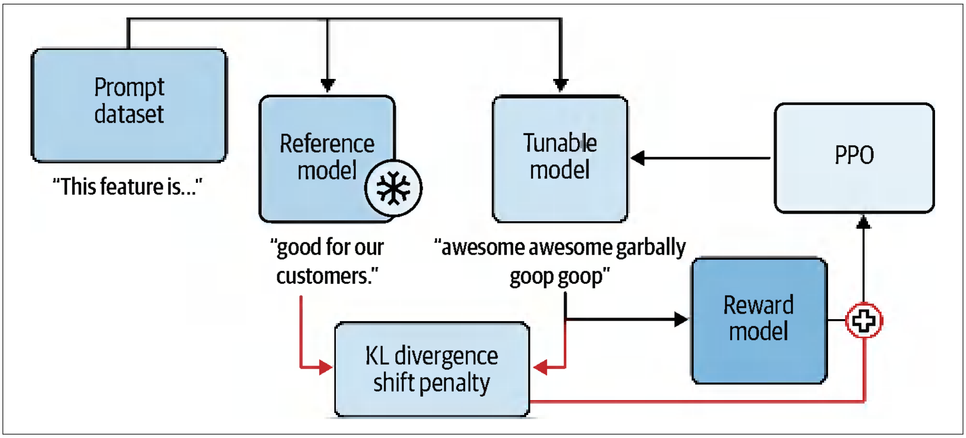


Figure 7-11. Mitigate rewards hacking with a KL divergence reward penalty

KL divergence quantifies how much the mutable, RLHF-tuned generative model is generating completions that diverge too far from the completions generated by the immutable reference model. In short, if the fine-tuned model starts hacking the reward and generating sequences of tokens that diverge too far from sequences that the reference model would generate, the fine-tuned model is penalized by the RL algorithm through a lower reward.

RLHF and KL divergence are extremely compute-intensive processes that benefit greatly from accelerators like the NVIDIA GPU or the AWS Trainium purpose-built hardware that is available through both Amazon EC2 and SageMaker.

The details of KL divergence and reward-penalties are typically contained in the RL libraries, so you typically don't need to implement this type of complexity yourself. However, it does help to understand reward hacking—as well as the techniques and extra computations required to control it.

Following is the code to configure the `PPOTrainer` class from the TRL library with the frozen reference to help avoid reward hacking. Also shown is the PPO iteration `step` function that updates the weights of the tunable model using the reward value produced by the reward model. Remember that the final reward value may be penalized if the generated text starts to diverge from the reference model, as calculated by KL divergence within the `PPOTrainer` implementation. Here is the code to add the frozen reference model to the `PPOTrainer`:

```
from trl import PPOTrainer
from trl import AutoModelForCausalLMWithValueHead
from trl import create_reference_model
from transformers import AutoTokenize
model_checkpoint = "..." # generative model like Llama2, Falcon
tokenizer = AutoTokenizer.from_pretrained(model_checkpoint)

model = AutoModelForCausalLMWithValueHead.from_pretrained(
    model_checkpoint,
    torch_dtype=torch.bfloat16)

ref_model = create_reference_model(model)

ppo_trainer = PPOTrainer(
    model=model,         # tunable model
    ref_model=ref_model, # frozen reference model
    tokenizer=tokenizer,
    dataset=dataset)
```

Using Parameter-Efficient Fine-Tuning with RLHF

Parameter-Efficient Fine-Tuning (PEFT), discussed in Chapter 6, can be used with RLHF to reduce the amount of compute and memory resources required for the compute-intensive PPO algorithm, as shown in Figure 7-12. Specifically, you would only need to update the model's much-smaller PEFT adapter weights and not the full weights of the tunable model.

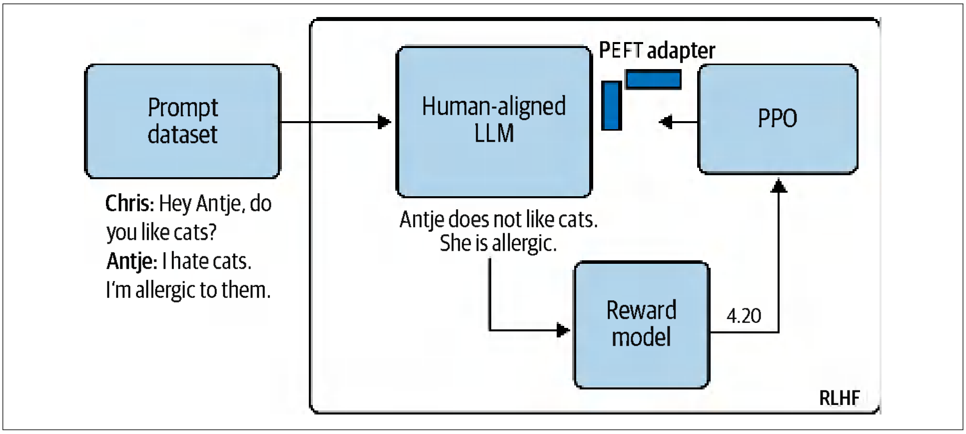


Figure 7-12. Using PEFT within RLHF to minimize the resources needed to fine-tune the generative model

After the RLHF process is complete, the fine-tuned, human-aligned model is ready for evaluation and, depending on the evaluation result, ready for production deployment. Next, let's explore model evaluation in the context of aligning for human values and preferences.

Evaluate RLHF Fine-Tuned Model

Continuing with the toxicity example, you can evaluate your RLHF fine-tuned model using both qualitative and quantitative evaluation techniques. Let's start by comparing the results of the model qualitatively by reviewing a prompt, completion before RLHF, and completion after RLHF.

Qualitative Evaluation

Prompt:

```
Summarize the following conversation.

#Person1#: So how did you like the restaurant?
#Person2#: Actually, it could have been better.
#Person1#: What didn't you like about it?
#Person2#: It is a new restaurant. I don't think they have their act together
yet.
#Person1#: What did you think about the food?
#Person2#: I felt that the food was pretty mediocre.
#Person1#: The service wasn't that great, either.
#Person2#: I agree. The service was not good. I'm tired of that place.
```

Completion before RLHF (Lower Reward):

```
#Person1# asks #Person2# about the restaurant. #Person2# liked the food but
it wasn't as good as #Person2# expected it to be. #Person2#'s tired of the
restaurant.
```

Completion after RLHF (Higher Reward):

```
#Person2# describes the restaurant to #Person1# and the food situation.
#Person2# doesn't want to try the restaurant again.
```

While this is a subjective comparison, the completion after RLHF appears to be a bit less harsh than the completion before RLHF yet conveys approximately the same meaning. In addition, the completion after RLHF has a higher reward, which is a signal that the toxicity-detector reward model preferred the completion after RLHF over the completion before RLHF.

Next, you will perform a more quantitative evaluation using toxicity scores to compare across many prompt-completion pairs before and after RLHF.

Quantitative Evaluation

To compare the generative model before and after RLHF, you can use an aggregate toxicity score for a large number of completions generated by the model using a test dataset that the model did not see during RLHF fine-tuning. If RLHF has successfully reduced the toxicity of your generative model, the toxicity score will decrease relative to the baseline, as shown in Figure 7-13.

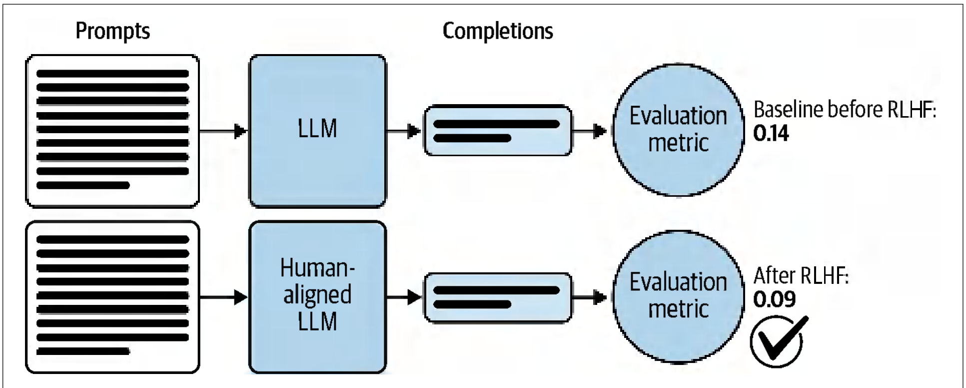


Figure 7-13. Evaluate using the toxicity score—lower is better

As you see in Figure 7-13, you would first calculate a baseline toxicity score for the original model before RLHF fine-tuning, then perform RLHF fine-tuning and measure the toxicity score afterward. Let's dive deeper into calculating the toxicity scores for the generative model using the test dataset.

Load Evaluation Model

First, you'll need to load the `toxicity_evaluator` using the Hugging Face Evaluate Python library with Meta's model for detecting toxic language, which you were using as the reward model in previous sections. Since this model is a classifier that predicts one of two classes, "not hate" and "hate," you need to specify the `toxic_label`, which is "hate" in this case. The evaluator then knows which label to use as the toxic label, as shown in this code:

```
import evaluate
toxicity_model_checkpoint =
  "facebook/roberta-hate-speech-dynabench-r4-target"
toxicity_evaluator = evaluate.load(
  "toxicity",
  toxicity_model_checkpoint,
  module_type="measurement",
  toxic_label="hate")
```

Define Evaluation-Metric Aggregation Function

Next, you define an `aggregate_toxicity_scores()` function to calculate the toxicity score mean and standard deviation for all prompts in the test dataset, as shown here:

```
def aggregate_toxicity_scores(model,
                              toxicity_evaluator,
                              tokenizer,
                              dataset):

    toxicities = []
    input_texts = []
    for i, sample in enumerate(dataset):
        input_text = sample["query"]

        input_ids = tokenizer(input_text,
          return_tensors="pt", padding=True).input_ids

        response_token_ids = model.generate(
            input_ids=input_ids)

        generated_text = tokenizer.decode(
            response_token_ids[0], skip_special_tokens=True)

        toxicity_score =
            toxicity_evaluator.compute(
              predictions=[(input_text + generated_text)])

        toxicities.extend(toxicity_score["toxicity"])
```

```
    # Compute mean & std using numpy.
    mean = np.mean(toxicities)
    std = np.std(toxicities)

    return mean, std
```

Compare Evaluation Metrics Before and After

Next, you calculate a toxicity baseline using the `aggregate_toxicity_scores()` function on the original generative model before performing RLHF. After performing RLHF, you measure the toxicity score again using the same `aggregate_toxicity_scores()` function:

```
from transformers import AutoTokenizer

model_checkpoint = "..." # generative model like Llama2, Falcon

tokenizer = AutoTokenizer.from_pretrained(model_checkpoint)
mean_before_detoxification, std_before_detoxification =
  evaluate_toxicity(model=model_before_rlhf,
    toxicity_evaluator=toxicity_evaluator,
    tokenizer=tokenizer,
    dataset=dataset["test"],
    num_samples=10)

print(f"""
Aggregate toxicity [mean, std] before detox:
[{mean_before_detoxification},
{std_before_detoxification}]
""")

#
# Perform RLHF PPO updates here...
#

mean_after_detoxification, std_after_detoxification =
  evaluate_toxicity(model=model_after_rlhf,
    toxicity_evaluator=toxicity_evaluator,
    tokenizer=tokenizer,
    dataset=dataset["test"],
    num_samples=10)

print(f'Aggregate toxicity [mean, std] after detox:
[{mean_after_detoxification}, {std_after_detoxification}]')

# Calculate improvement
mean_improvement = (mean_before_detoxification - mean_after_detoxification) \
  / mean_before_detoxification
std_improvement = (std_before_detoxification - std_after_detoxification) \
  / std_before_detoxification
```

```
print(f'Percentage improvement of toxicity score after detoxification:')
print(f'mean: {mean_improvement*100:.2f}%')
print(f'std: {std_improvement*100:.2f}%')
```

Output:

```
Aggregate toxicity [mean, std] before detox:
  [0.032297799189109355, 0.03010236943945737]
Aggregate toxicity [mean, std] after detox:
  [0.0271528000858697, 0.02743170674039297]
Percentage improvement of toxicity score after detoxification:
  mean: 15.93%
  std: 8.87%
```

Note the use of the test dataset, which the generative model did not see during RLHF fine-tuning. Here, you see a drop in aggregate toxicity score, which is the desired result.

Summary

Fine-tuning for human values is a very important tool in your generative toolbox to improve your model's helpfulness, honesty, and harmlessness. Reinforcement learning from human feedback (RLHF) is a very active area of research with a great amount of impact on making these models more humanlike, useful, and enjoyable. In this chapter, you learned the fundamentals of RL, reward models, and the RLHF process. These fundamentals will help you understand this exciting field as it continues to evolve.

You saw how to collect human feedback rankings using services like Amazon SageMaker Ground Truth with human annotators. You then learned how to convert the human-readable rankings into machine-readable preference data to train a reward model.

You then learned about some existing classifiers and managed services that can be used as reward models out of the box without any training. And finally, you learned about PPO and used it to perform RLHF updates to align a generative model with human values and preference. Specifically, you reduced a generative model's toxicity over a series of PPO iterations, which updated the weights of the model to generate fewer toxic completions.

Now that you have a human-aligned, lower-toxicity generative model, you will see how to optimize and deploy it for low-latency, high-performance inference in Chapter 8.

CHAPTER 8

Model Deployment Optimizations

After you have adapted your model to your target task, you will ultimately want to deploy your model so you can begin interacting with it as well as potentially integrating it into an application that is designed to consume it.

Before deploying your generative model, you need to understand the resources your model may need as well as the intended experience for interacting with it. Considering the resources your model will need will include identifying requirements such as how fast you need your model to generate completions, what compute budget you have available, and what trade-offs you are willing to make regarding model performance to be able to achieve faster inference speed and potentially reduce storage costs.

In this chapter, you will explore various techniques for performing post-training optimizations on your model, including pruning, quantization, and distillation. Additional considerations and potential tuning of your deployment configurations will need to be done postdeployment as well, such as selecting the optimal compute resources to balance cost and performance.

Model Optimizations for Inference

The size of generative AI models often presents a challenge for deployment in terms of compute, storage, and memory requirements, as well as how to ensure low-latency completions. One of the primary ways to optimize for deployment is to take advantage of techniques that aim to reduce the size of the model, typically referred to as model compression. Reducing the model size allows for quicker loading of the model and reduced latency. It also reduces the resource requirements for compute, storage, and memory.

While reducing model size helps optimize the model for deployment, the challenge is reducing the model size while maintaining good model performance. As a result, there can be a trade-off to consider between model performance, compute budget, and latency.

This section outlines three techniques aimed at reducing model size—model pruning, quantization, and distillation, as shown in Figure 8-1.

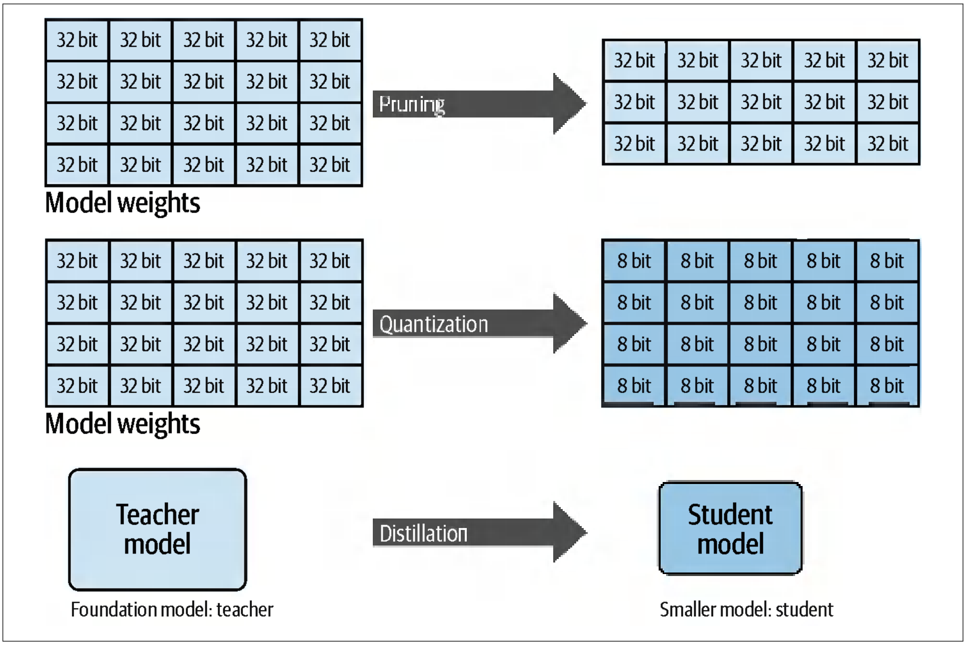


Figure 8-1. Techniques aimed at reducing model size for deployment optimization

Pruning is a technique that focuses on removing redundant, or low-impact, parameters that do not contribute, or contribute little, to the model's performance. Pruning reduces the size of the model, but also increases performance by reducing the number of computations during inference.

Quantization, a technique that you saw in Chapter 4, converts a model's weights from high precision (e.g., 32-bit) to lower precision (e.g., 16-bit). This not only reduces the model's memory footprint, but also improves model performance by working with smaller number representations. With large generative models, it's common to reduce the precision further to 8 bits to increase inference performance.

Distillation trains a smaller student model from a larger teacher model. The smaller model is then used for inference to reduce your compute resources yet retain a high percentage of accuracy of your student model. A popular distilled student model is DistilBERT from Hugging Face. DistilBERT was trained from the larger BERT teacher model and is an order of magnitude smaller than BERT, yet it retains approximately 97% of the accuracy of the original BERT model. See our book *Data Science on AWS* (O'Reilly, 2021) for a deep dive on BERT and DistilBERT.

The following sections will discuss each of these techniques in more detail. Note that you can use all of the techniques together.

Pruning

Pruning aims to eliminate model weights that are not contributing significantly to the model's overall performance, as shown in Figure 8-2. By eliminating those model weights, you're able to reduce the model size for inference, which reduces the compute resources required.

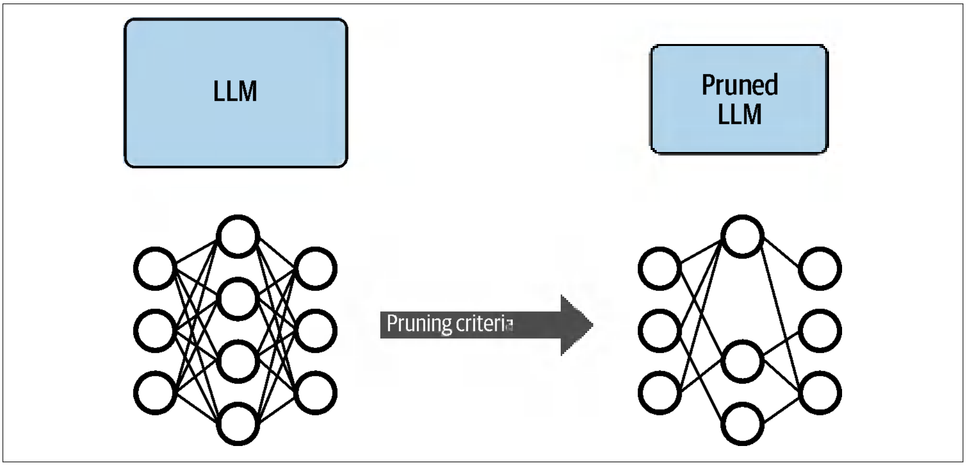


Figure 8-2. Pruning aims to reduce the overall model size by eliminating weights that are not contributing to model performance

The model weights to be eliminated during pruning are those with a value of zero or very close to zero. Pruning during training is accomplished through unstructured pruning (removing weights) or structured pruning (removing entire columns or rows of the weight matrices).

These approaches require retraining; however, there are post-training pruning methods—typically referred to as one-shot pruning methods—that can do pruning without retraining. The challenge in performing one-shot pruning is that it's often compute intensive for large models with billions of parameters.

One method of post-training pruning, called SparseGPT,[1] aims to overcome the challenges of one-shot pruning on large language models. This method is specifically built for language-based generative foundation models and introduces an algorithm that performs sparse regression at a large scale.

In theory, pruning reduces the size of the LLM, which reduces compute resources and model latency. However, in practice, there are LLMs where only a small percentage of their weights are zero, so in those cases pruning may not have a large impact on the model size.

Here is a code sample from the SparseGPT pruning library for the LLaMA and Llama 2 models:

```
target_sparsity_ratio = 0.5

# Prune each layer using the given sparsity ratio
for layer_name in layers:
  gpts[layer_name].fasterprune(
    target_sparsity_ratio,
  )

gpts[layer_name].free() # free the zero'd out memory
```

Post-Training Quantization with GPTQ

Similar to quantization described in Chapter 4, post-training quantization (PTQ) aims to transform the model's learned weights into a lower-precision representation with the goals of reducing the model's size and the compute requirements when hosting generative models for inference.

PTQ requires an extra calibration step to statistically capture the range of the original model weights in the range of the reduced precision. The calibration step uses a dataset that statistically represents the type of inputs the model will receive during inference. This calibration step helps identify the range with minimum and maximum boundaries, as shown in Figure 8-3. This range calculation can be computed at runtime (dynamic quantization) or computed in advance (static quantization).

There are a variety of post-training quantization methods, including GPT post-training quantization (GPTQ). GPTQ was first proposed in the paper "GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers."[2] GPTQ is capable of reducing the number of bits needed to store each weight from 32 bits for full precision down to 4, 3, or even 2 bits!

1 Elias Frantar and Dan Alistarh, "SparseGPT: Massive Language Models Can Be Accurately Pruned in One-Shot", *arXiv*, 2023.

2 Elias Frantar et al., "GPTQ: Accurate Post-Training Quantization for Generative Pre-Trained Transformers", *arXiv*, 2023.

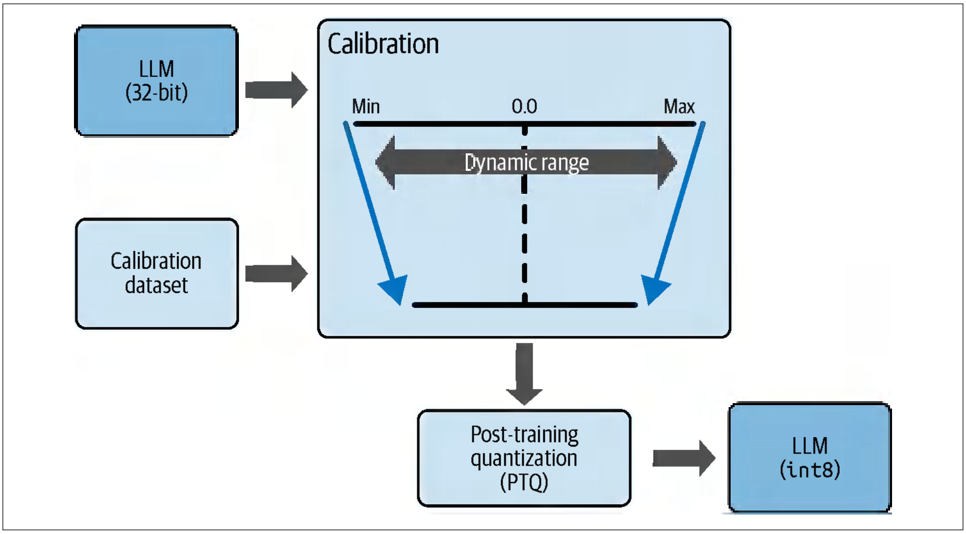


Figure 8-3. PTQ requires an extra calibration step to determine the dynamic range

GPTQ analyzes each layer of the model separately and approximates the weights in a way that helps reduce accuracy loss typically seen during quantization. GPTQ requires a calibration dataset, as you will see next using the Hugging Face Optimum library with the Wikitext dataset:

```
import torch
from optimum.gptq import GPTQQuantizer
from transformers import AutoModelForCausalLM, AutoTokenizer

dataset_id = "databricks/databricks-dolly-15k"

# GPTQ quantizer - 4 bits
quantizer = GPTQQuantizer(bits=4,
  dataset_id=dataset_id,
  model_seqlen=4096)

quantizer.quant_method = "gptq"

tokenizer = AutoTokenizer.from_pretrained(model_checkpoint)

model = AutoModelForCausalLM.from_pretrained(model_checkpoint,
  torch_dtype=torch.float16)

# Quantize the model
quantized_model = quantizer.quantize_model(model, tokenizer)

# Save the quantize model to disk
save_folder = model.save_pretrained("quantized_model")
```

The Wikitext dataset is commonly used for post-training quantization calibration with language-based generative models because it is representative of the type of text data that these models will see during inference.

> As described in Chapter 4, quantization typically improves inference latency by reducing the needed computational resources. However, it may result in a small percentage loss in model accuracy. This reduction is often worth the cost savings and performance gains, however. It is recommended that you always benchmark the quantization results to determine if the trade-offs are acceptable for your use case.

Distillation

Distillation is a technique that helps reduce the model size, which ultimately reduces the number of computations and improves model inference performance. Distillation uses statistical methods to train a smaller student model on a larger teacher model. The end result is a student model that retains a high percentage of the teacher's model accuracy but uses a much smaller number of parameters. The student model is then deployed for inference. The smaller model requires smaller hardware and therefore less cost per inference request.

The teacher model is often a generative foundation model or a fine-tuned variant. During the distillation training process, the student model learns to statistically replicate the behavior of the teacher model. Note that the teacher model weights do not change during the distillation process—only the student model weights change. The teacher model's output is used to "distill" knowledge to the student model.

Both the teacher and student models generate completions from a prompt-based training dataset. A distillation loss is calculated by comparing the two completions and calculating the KL divergence, which you explored for RLHF in Chapter 7, between the teacher and student output distributions.

The loss—including KL divergence—is then minimized during the distillation process using backpropagation to improve the student model's ability to match the teacher model's predicted next-token probability distribution, as shown in Figure 8-4.

The teacher models' predicted tokens are known as *soft labels*, while the student models' predicted tokens are called *soft predictions*. In parallel, you need to compare the student models' predictions (*hard predictions*) against the ground truth *hard labels* from the prompt dataset. The difference is the *student loss*. The distillation loss and student loss are combined and used to update the student models' weights using standard backpropagation.

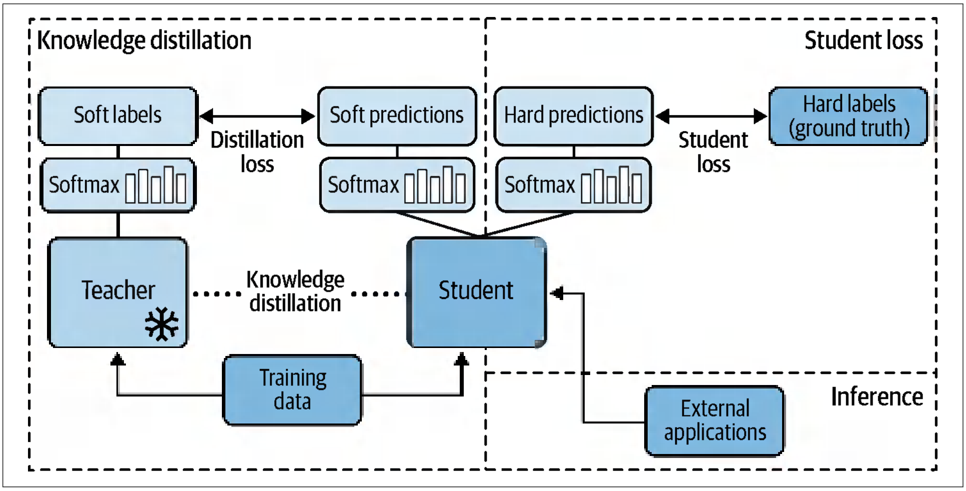


Figure 8-4. Distill knowledge from teacher to student model

In practice, distillation may not be as effective for generative decoder models as it is for encoder models like BERT. This is because the output space is relatively large for decoder models (with a vocabulary size of, e.g., 100,000 tokens) without a lot of redundancy in representation.

Here is an example distillation loss function from the Hugging Face Optimum library for distillation:

```
def compute_distillation_loss(self, inputs, student_outputs):
    with torch.no_grad():
        teacher_outputs = self.teacher(**inputs)

    temperature = self.args.distillation_temperature

    distilliation_loss_start = F.kl_div(
      input=F.log_softmax(
        student_outputs.start_logits / temperature, dim=-1),
      target=F.softmax(
        teacher_outputs.start_logits / temperature, dim=-1),
      reduction="batchmean",
    ) * (temperature**2)

    distilliation_loss_end = F.kl_div(
        input=F.log_softmax(
          student_outputs.end_logits / temperature, dim=-1),
        target=F.softmax(
          teacher_outputs.end_logits / temperature, dim=-1),
```

```
        reduction="batchmean",
    ) * (temperature**2)

    return \
      (distilliation_loss_start + distilliation_loss_end) / 2.0
```

Now that you have seen various mechanisms to optimize your model for inference, it's time to deploy your model to accept inputs and generate responses. For this, you can use Amazon SageMaker Endpoints to host and scale your generative models in production, as you will explore next.

Large Model Inference Container

The real-time SageMaker Endpoints managed service comes preconfigured with many runtime, hardware, A/B testing, and shadow deployment optimizations for generative model inference. The large model inference (LMI) container is the primary runtime that contains these optimizations.

The LMI containers from AWS use a prebuilt foundation software stack that includes high-performance frameworks like DeepSpeed and optimizations like FlashAttention,[3] which you learned about in Chapter 4. Figure 8-5 shows some of the key components of the LMI container, including PyTorch, FlashAttention, DeepSpeed, and the AWS Neuron SDK. This figure also shows some of the hardware supported by the LMI container, including NVIDIA GPUs, AWS Inferentia chips, and classic CPUs.

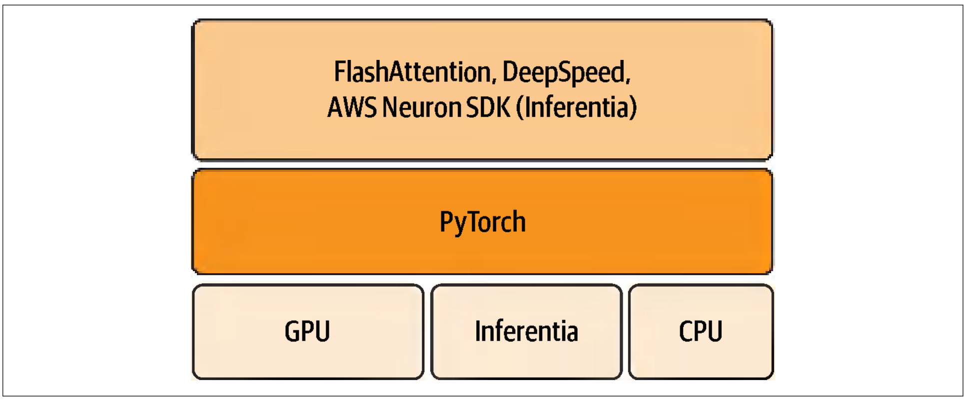


Figure 8-5. LMI container and hardware for hosting LLMs with Amazon SageMaker Endpoints

3 Tri Dao, "FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning", *arXiv*, 2023.

LMI supports both batch and real-time workloads. Here is sample code to deploy and test a real-time generative large language model using Amazon SageMaker JumpStart, which uses the LMI container with SageMaker Endpoints:

```
from sagemaker.jumpstart.model import JumpStartModel

model = JumpStartModel(
  model_id="...") # generative model like Llama2 or Falcon

predictor = model.deploy()

payload = {
    "inputs": "What is the best way to deploy a generative model on AWS?",
    "parameters": {
        "max_new_tokens": 100,
        "top_p": 0.9,
        "temperature": 0.6
    }
}

response = predictor.predict(payload)
```

As you can see, in just a few lines of code, you can deploy a powerful model into your own AWS account to perform secure and private generative inferences. In the next section, you will learn about the AWS Inferentia family of hardware specifically designed for deep-learning inference workloads.

AWS Inferentia: Purpose-Built Hardware for Inference

The AWS Inferentia family of accelerators, currently on version 2, is purpose-built for deep learning inference workloads. The AWS Neuron SDK interacts with AWS Inferentia.

There are two common ways to develop with the AWS Neuron SDK, including the Transformers-NeuronX library and the Hugging Face Optimum Neuron library. Here are examples using both libraries to compile your model to run on Amazon SageMaker with AWS Inferentia 2:

```
import torch
from transformers import AutoTokenizer
from transformers import AutoModelForCausalLM

########################################
# Using the Transformers-NeuronX library
########################################
from transformers_neuronx.llama.model import LlamaForSampling

model = AutoModelForCausalLM.from_pretrained(model_checkpoint)
```

```
os.environ["NEURON_CC_FLAGS"] =
  "--model-type=transformer-inference"

neuron_model =
  LlamaForSampling.from_pretrained(model_checkpoint,
    batch_size=1, tp_degree=24, amp='fp16', ...)

# Compile and save the model
neuron_model.to_neuron()
neuron_model.save_pretrained('compiled_model/')

# Save the tokenizer with the model
tokenizer = AutoTokenizer.from_pretrained(model_checkpoint)
tokenizer.save_pretrained('compiled_model/')

##############################
# Using Optimum Neuron library
##############################
from optimum.neuron import NeuronModelForCausalLM

# Load and convert the Hub model to Neuron format
neuron_model = NeuronModelForCausalLM.from_pretrained(
  model_checkpoint,     # model id
  batch_size=1,         # number of input sequences
  num_cores=24,         # number of neuron cores
  auto_cast_type='f16', # format to encode the weights
  ...
)
neuron_model.save_pretrained('compiled_model/')

# Save the tokenizer with the model
tokenizer = AutoTokenizer.from_pretrained(model_checkpoint)
tokenizer.save_pretrained('compiled_model/')
```

Next, you will tar and gzip the contents of the *compiled_model/* local directory and upload the *tar.gz* to a private S3 location where the SageMaker Endpoint will find the model and load it:

```
from sagemaker.s3 import S3Uploader

# tar and gzip the compiled_model/ folder
local_model_tar_gz_file = "model.tar.gz"
...

# Create s3 uri
s3_model_path = "s3://<your-private-s3-location/"

# Upload model.tar.gz
s3_model_uri = S3Uploader.upload(
    local_path=local_model_tar_gz_file,
    desired_s3_uri=s3_model_path)
```

Once the model is compiled, saved, tar'd, gzipped, and pushed to S3, you can now deploy the model as an Amazon SageMaker Endpoint using the code here and start generating text. Here, we are specifying the AWS Inferentia 2 instance type for SageMaker:

```
from sagemaker.huggingface.model import HuggingFaceModel

huggingface_model = HuggingFaceModel(
   model_data=s3_model_uri,  # path to model in s3
   model_server_workers=2,   # number of workers
   ...
)

# Specify that the model has been precompiled
huggingface_model._is_compiled_model = True

# Deploy the endpoint
predictor = huggingface_model.deploy(
    instance_type="ml.inf2.xlarge", # Inferentia 2 instance type
    ...
)
prompt =
  "What is the best way to deploy a generative model on AWS?"
# Inference generation configuration parameters
payload = {
  "inputs": prompt,
  "parameters": {
    "do_sample": True,
    "top_p": 0.6,
    "temperature": 0.9,
    "top_k": 50,
    "max_new_tokens": 512,
    "repetition_penalty": 1.03,
    "stop": ["</s>"]
  }
}
# Send request to endpoint
response = predictor.predict(payload)

# Extract the generated response
print(response[0]["generated_text"])
```

Model Update and Deployment Strategies

In this section, you will see a couple common strategies used to update models in production, including A/B testing and shadow deployments. With A/B testing, you typically shift a small amount of traffic to the newer model B for a period of time, to ensure the new model is not failing or performing poorly relative to the original model A.

If the new model B performs poorly, however, end users will be affected. With shadow deployments, the new model B is deployed alongside model A as a shadow, accepts a copy of the traffic (e.g., prompt inputs), but does not return the model response to the end user. Instead, shadow model B's response is logged for offline analysis of the model's performance. If something goes wrong with model B, the end user is not affected.

Let's dive deeper into each of these model update and deployment strategies.

A/B Testing

You can use Amazon SageMaker Endpoints to deploy two different model variants behind a single endpoint to compare the variants with live traffic. This is typically called A/B testing. Consider deploying two model variants, models A and B, behind a single SageMaker Endpoint. Figure 8-6 shows 100% of the traffic routing to model A initially, then shifting 10% of the traffic to model B.

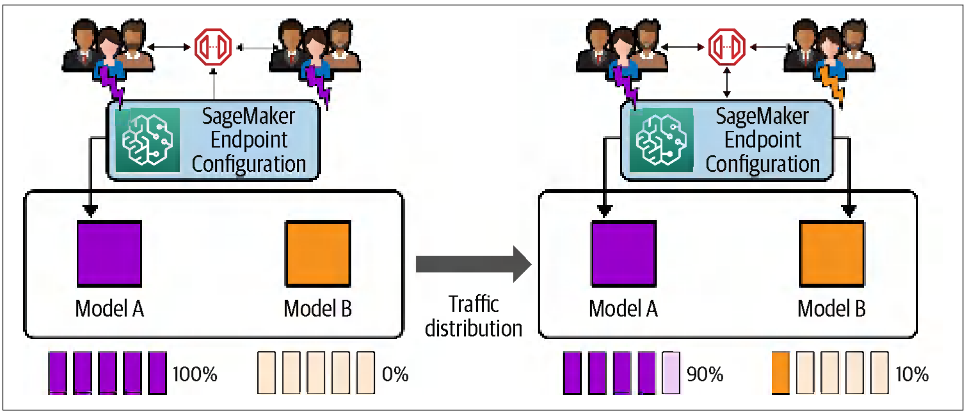


Figure 8-6. Shifting live traffic from model A to model B

This lets you try model B on live traffic in a controlled manner and only affect 10% of the end users if something goes wrong. You can quickly shift traffic back to model A, if needed. Here is the code to implement this configuration:

```
import boto3

sm = boto3.Session().client(service_name="sagemaker")

sm.create_endpoint_config(
    EndpointConfigName="generative-endpoint-config-ab-test",
    ProductionVariants=[
        {
          "ModelName": "generative-model",
          "VariantName": "generative-model-A",
```

```
            "InitialVariantWeight": 90,
            "InitialInstanceCount": 9
          },
          {
            "ModelName": "generative-model",
            "VariantName": "generative-model-B",
            "InitialVariantWeight": 10,
            "InitialInstanceCount": 1
          }
      ]
)
endpoint_name = "generative-ab-endpoint"

sm.create_endpoint(
     EndpointName=endpoint_name,
     EndpointConfigName=endpoint_config
)

waiter = sm.get_waiter("endpoint_in_service")
waiter.wait(EndpointName=endpoint_name)
...
# Send request to A/B endpoint
response = predictor.predict(payload)

# Extract the generated response
print(response[0]["generated_text"])
```

Here, you are creating the `EndpointConfiguration`, which includes the hardware using `InstanceType` and `InitialInstanceCount`. In this case, you are deploying two variants of your model in an A/B test across 10 GPU-based SageMaker instances. 90% of your traffic will go to `generative-model-A` and 10% will go to `generative-model-B`.

This code allows you to compare the two variants and, at some point, send 100% of the traffic to the better model based on some evaluation criteria or longer-term objective, such as increasing revenue or reducing churn.

Shadow Deployment

SageMaker Endpoints support shadow model deployments. When you deploy a shadow model, the model accepts the same input as the primary model, but it simply stores the model response to disk for offline analysis, as shown in Figure 8-7. This helps you conservatively evaluate a model against live production inputs without exposing potentially bad responses to the end user.

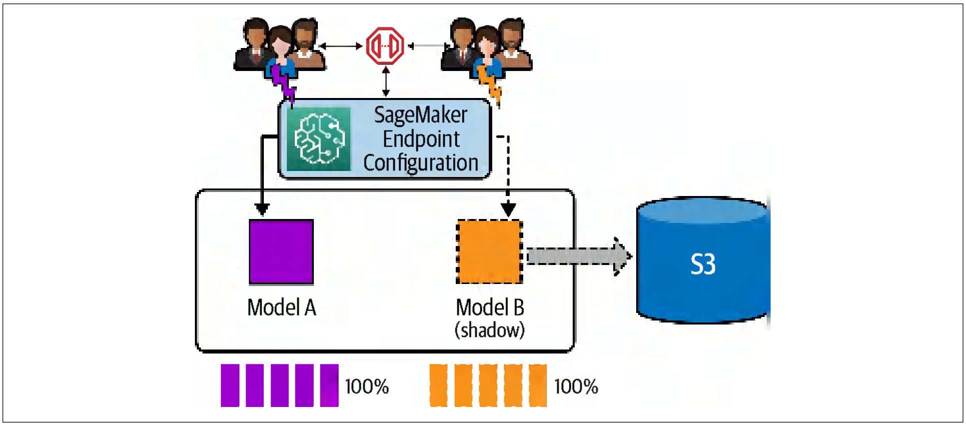


Figure 8-7. A shadow model accepts the same inputs, but stores model response to disk for offline analysis

Here is the code sample for a shadow deployment. Note that both `InitialVariantWeight` settings are configured for 100% traffic. You can choose to send a smaller percentage of traffic to the shadow variant if you want to sample less than 100% of the traffic:

```
sm.create_endpoint_config(
    EndpointConfigName=endpoint_config,
    ProductionVariants=[
        {
          "ModelName": "generative-model",
          "VariantName": "generative-model-A",
          "InitialVariantWeight": 100,
          "InitialInstanceCount": 9
        }
    ],
    ShadowProductionVariants=[
        {
          "ModelName": "generative-model",
          "VariantName": "generative-model-B",
          "InitialVariantWeight": 100,
          "InitialInstanceCount": 1
        }
    ]
)
```

This shows the shadow variant, `generative-model-B`, configured to accept traffic but not return the response back to the user. Instead, the shadow model will accept traffic and send the results to S3 for offline analysis.

For a more comprehensive description of SageMaker deployment strategies, check out our book *Data Science on AWS*.

Metrics and Monitoring

Amazon SageMaker Endpoints emit many useful metrics that are captured by the Amazon CloudWatch managed service for metrics collection and monitoring. These metrics are used not just for operational reasons, but also for scaling your inference cluster out (to a larger number of instances) and in (to a smaller number of instances) as traffic to the cluster increases and decreases throughout the day. This is called autoscaling, which you will see in the next section.

But first, let's take a look at some of the metrics emitted by Amazon SageMaker Endpoints when hosting a generative AI model. Table 8-1 shows some of the common metrics used to monitor model inference, including error counts, startup times, and prediction-latency timings.

Table 8-1. Monitoring metrics for model inference

Metric	Description

`Invocation4XXErrors`
`Invocation5XXErrors`
`InvocationModelErrors`
	Number of model invocations that did not result in a successful 2XX HTTP response

`Invocations`
`InvocationsPerInstance`
`SageMakerVariantInvocationsPerInstance`
	The number of invocation requests sent to a model endpoint overall, per instance, and per variant per instance

`ModelLatency`	Inference latency of the model only
`OverheadLatency`	Latency introduced by SageMaker during the model inference
`ModelSetupTime`	Model startup time including downloading the model and launching the SageMaker container

`CPUUtilization`
`GPUUtilization`
`MemoryUtilization`
`GPUMemoryUtilization`
	CPU, GPU, and memory utilization of the model endpoint

`DiskUtilization`	The percentage of disk space used to host the model for inference

The next section will discuss how some of these monitoring metrics can be utilized to configure autoscaling, which dynamically adjusts the number of instances provisioned for a deployed model in response to changes in demand from your workload.

Autoscaling

In the A/B testing and shadow deployment examples, you saw how to manually set the `InitialInstanceCount` in the `EndpointConfig`. This represents the number of instances in the inference cluster. As traffic increases and decreases, you would need to manually update the number of instances to a higher or lower value, respectively.

However, it's often easier to set up autoscaling to automatically scale out (add instances) or scale in (remove instances) based on a given metric, like the number of invocations per second. As traffic increases and decreases, the invocations per second metric will cause SageMaker to automatically scale our model cluster to meet the demand.

Let's dive deeper into configuring autoscaling policies for SageMaker Endpoints.

Autoscaling Policies

There are three main types of autoscaling policies for SageMaker Endpoints: target tracking, simple, and step scaling. These policies trade off ease of use with flexibility:

Target tracking
: With target tracking scaling policy, you specify a single metric, like `SageMakerVariantInvocationsPerInstance = 1000`, and SageMaker will autoscale as needed. This strategy is very common, as it's the easiest to configure.

Simple
: When configured to use the simple scaling policy, SageMaker will trigger a scaling event on a given metric at a given threshold with a fixed amount of scaling. For example, "when `SageMakerVariantInvocationsPerInstance > 1000`, add 10 instances." This strategy requires a bit more configuration but also provides more control over the target-tracking strategy.

Step scaling
: Step scaling, the most configurable scaling policy, allows SageMaker to trigger a scaling event on a given metric at various thresholds—with configurable amounts of scaling at each threshold. For example, "when `SageMakerVariantInvocationsPerInstance > 1000`, add 10 instances, `SageMakerVariantInvocationsPerInstance > 2000`, add 50 instances," etc. This strategy requires the most amount of configuration but provides the most amount of control for situations such as spiky traffic.

Define an Autoscaling Policy

Let's define and apply a target-tracking autoscaling policy using the SageMaker VariantInvocationsPerInstance metric to automatically scale the endpoint cluster when one thousand invocations per second is reached for a given model variant per instance:

```
endpoint_name = "..."

autoscale = boto3.Session().client(
  service_name='application-autoscaling')

autoscale.register_scalable_target(
  ServiceNamespace="sagemaker",
  ResourceId=f"endpoint/{endpoint_name}/variant/AllTraffic",
  ScalableDimension="sagemaker:variant:DesiredInstanceCount"
)

autoscale.put_scaling_policy(
  PolicyName="my-autoscale-policy",
  ServiceNamespace="sagemaker",
  ResourceId=f"endpoint/{endpoint_name}/variant/AllTraffic",
  ScalableDimension="sagemaker:variant:DesiredInstanceCount",
  PolicyType="TargetTrackingScaling",
  TargetTrackingScalingPolicyConfiguration={
    "TargetValue": 1000.0,
    "PredefinedMetricSpecification": {
      "PredefinedMetricType":
        "SageMakerVariantInvocationsPerInstance",
  }]
})
```

Note that the ScalableDimension is set to sagemaker:variant:DesiredInstanceCount, which configures SageMaker to scale the number of instances when the target threshold is met.

After sending a large amount of inference requests to the SageMaker Endpoint, you would see a spike in the SageMakerVariantInvocationsPerInstance metric. This would trigger SageMaker to scale out to handle the spike in inference requests.

There are many more autoscaling configuration options available, including model-variant-specific scaling policies and scale-in/scale-out cool-down policies. For a more comprehensive description of SageMaker autoscaling policies, again, you can check our book *Data Science on AWS*.

Summary

In this chapter, you learned powerful techniques to optimize your model for inference by reducing the size of the model through distillation, quantization, or pruning. These techniques help reduce model size and improve model inference performance with minimal impact on model accuracy, ultimately improving the user's happiness. They also help to minimize the amount of hardware resources needed to serve your generative models in production, ultimately lowering cost and improving your CFO's happiness.

You also saw how to optimize and deploy your models with the AWS Neuron SDK, Hugging Face's Optimum Neuron library, and Amazon SageMaker Endpoints with AWS Inferentia 2. Combined with A/B testing and shadow deployments, SageMaker Endpoints are a great way to productionize your generative AI models.

In the Chapter 9, you will dive deep into some popular mechanisms to build generative AI applications, including augmenting the capabilities of your models with retrieval-augmented generation (RAG) and agents.

CHAPTER 9

Context-Aware Reasoning Applications Using RAG and Agents

In this chapter, you will explore how to bring together everything you've learned so far to build context-aware reasoning applications. To do this, you will explore retrieval-augmented generation (RAG) and agents. You will also learn about frameworks called LangChain, ReAct, and PAL, which make RAG and agent workflows much easier to implement and maintain. Both RAG and agents are often key components of a generative AI application.

With RAG, you augment the context of your prompts with relevant information needed to address knowledge limitations of LLMs and improve the relevancy of the model's generated output. RAG has grown in popularity due to its effectiveness in mitigating challenges such as knowledge cutoffs and hallucinations by incorporating dynamic data sources into the prompt context without needing to continually fine-tune the model as new data arrives into your system.

RAG can be integrated with off-the-shelf foundation models or with fine-tuned and human-aligned models specific to your generative use case and domain.

> RAG and fine-tuning can be used together. They are not mutually exclusive.

Next, some general guidance to consider when deciding which techniques should be applied. If access to external data or dynamic data is required, then RAG-based architectures can enable this without continuous fine-tuning, which would become cost prohibitive. Also, RAG-based techniques do not require much ML expertise because they are typically implemented using existing foundation models.

Potential downsides to RAG-based architectures include the extra steps required to manage data source connections, retrieve data from external data sources, perform additional data preparation, and perform prompt augmentation. These extra steps may increase latency and decrease overall performance. It's also important to note that RAG does not actually modify the weights of the generative model; however, this is often desirable and typically not considered a downside.

Agents are additional pieces of software that can orchestrate prompt-completion workflows between user requests, foundation models, and external data sources and applications while using the foundation model as their reasoning engine.

Agents often make use of a framework called ReAct[1] (*re*asoning and *act*ing). ReAct structures prompts using chain-of-thought (CoT) reasoning to show the model how to reason through a problem and decide on actions to help find a solution. As part of the actions, agents can work with RAG workflows to look up context-relevant information or call application APIs to perform a task.

If the reasoning steps and actions require complex calculations, you can leverage another technique, Program-Aided Language Models (PAL).[2] PAL guides foundation models to generate programs instead of natural language in the reasoning steps. You can then connect the model to an external code interpreter, such as a Python interpreter, to run the code and return the results to the model.

You'll also learn about building a customized generative AI application using a set of common components required for an end-to-end solution that can be made available to users and other systems. Finally, this chapter will highlight a few considerations for optimizing the generative AI project life cycle and operationalizing models for deployment and integration into applications supporting users and systems.

Large Language Model Limitations

Large language models (LLMs) suffer from several challenges related to having accurate knowledge as well as current knowledge. This section discusses two common problems with large language models that can be improved using RAG methods: hallucination and knowledge cutoff.

1 Shunyu Yao et al., "ReAct: Synergizing Reasoning and Acting in Language Models", *arXiv*, 2023.

2 Luyu Gao et al., "PAL: Program-Aided Language Models", *arXiv*, 2023.

Hallucination

Chapter 2 discussed the challenge of hallucination, where a model confidently returns an incorrect response. In the example following, a "snazzy-fluffykins" is not a real dog breed, but the model still returns a factitious, and potentially misleading, completion, as shown next.

Prompt:

```
What does the dog breed snazzy-fluffykins look like?
```

Completion (hallucination):

```
The snazzy-fluffykins is a small, fluffy dog with curly fur. They have big,
round eyes and long, fluffy ears.
```

Hallucinations lead to a lack of trust in the model's generated outputs, impacting the effectiveness of a generative AI application.

Knowledge Cutoff

The second common issue, shown in the prompt example that follows, is known as knowledge cutoff, which results in the model returning an answer that is out of date with current data. All foundation models have a knowledge cutoff of the date that they were trained; the knowledge of the model is limited to the data that was current at the time it was pretrained or fine-tuned.

For example, if you ask the model who recently won the NBA championship, it will give you the most recent information it has available; in this case, the champions in 2021. However, it won't provide the most current data available because that data is outside the scope of the knowledge the model was trained on, as shown in the example here.

Prompt:

```
Who recently won the NBA Championship game?
```

Completion (knowledge cutoff—out of date):

```
The Milwaukee Bucks defeated the Phoenix Suns in the 2021 NBA Finals to win
their second NBA championship.
```

RAG provides a technique that allows you to mitigate some of the challenges with hallucinations and knowledge cutoff in foundation models. For hallucinations, RAG is useful because you are able to provide the model with access to information it would not already have, such as proprietary data for your business.

Retrieval-Augmented Generation

RAG isn't a specific set of technologies but rather a framework for providing LLMs access to data they did not see during training. RAG allows LLM-powered applications to make use of external data sources and applications to overcome some of the knowledge limitations previously discussed.

RAG is useful in any case where you want the language model to have access to additional data that is not contained within the LLMs "parametric memory" learned during pretraining and fine-tuning. This could be data that did not exist in the original training data, such as proprietary information from your organization's internal data stores. Allowing your model to have access to this information helps improve model completion relevance and helps to mitigate the challenge of hallucinations.

For knowledge cutoffs, RAG allows you to provide access to current information beyond the model's training date. This technique can augment foundation models with additional information, including domain-specific information, without the need to continuously perform full fine-tuning.

At a high level, a RAG-based architecture provides the model with access to external sources of knowledge that provide additional context to the original input prompt in the form of an augmented prompt, which is then used to call the LLM, as shown in Figure 9-1.

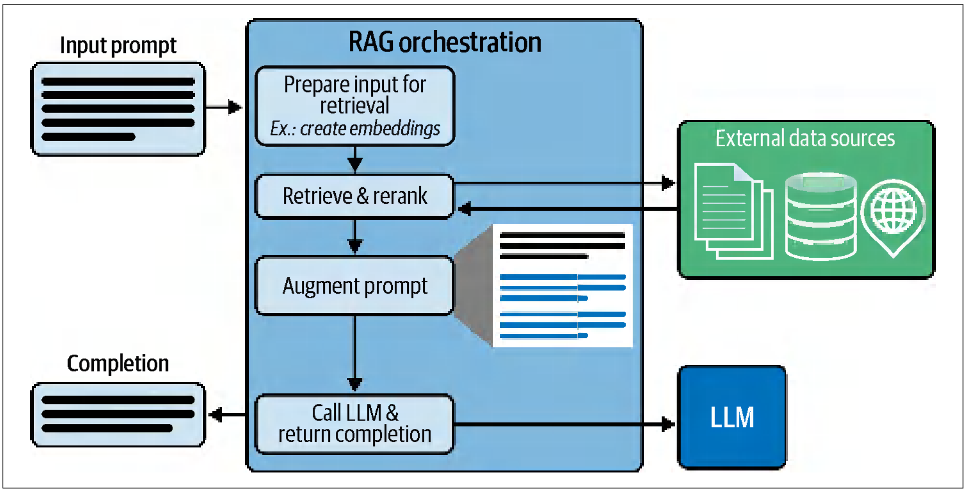


Figure 9-1. RAG provides a framework for augmenting a model with information from external sources

The LLM is then able to take advantage of knowledge outside its scope through the augmented prompt to return a more accurate and relevant completion. Let's now dive into the various components and pieces of the workflow.

External Sources of Knowledge

RAG works by providing your model access to additional external data at runtime. This data can be from a number of data sources, including knowledge bases, document stores, databases, and data that is searchable through the internet, as shown in Figure 9-2.

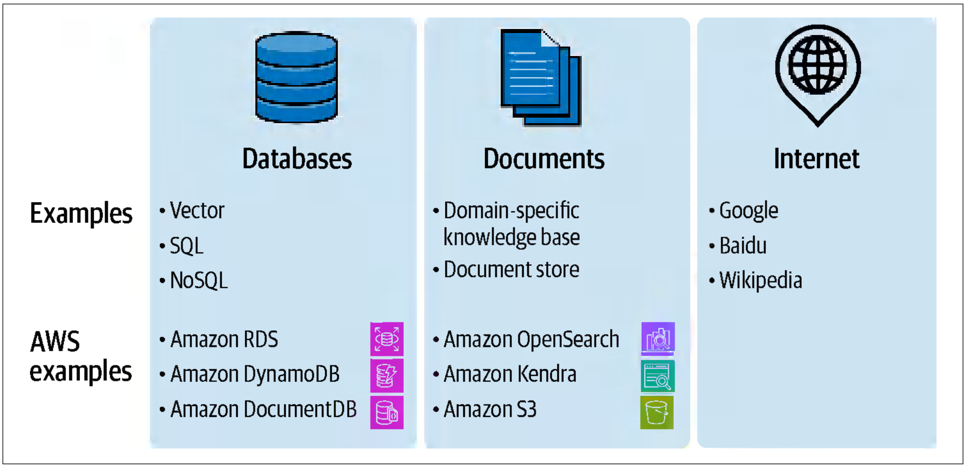


Figure 9-2. External data sources

All of these external data sources can provide access to knowledge previously unavailable to generative models, allowing for improved relevance in completions. RAG works by augmenting the input prompt with information from external data sources prior to calling the LLM. The augmented prompt provides access to information the model is not aware of, increasing the ability for the LLM to return more accurate and relevant completions.

However, implementing RAG-based architectures often requires additional data preparation tasks to ensure the data is in an optimized format that can be integrated at inference time, which involves personas that load and prepare the data for retrieval, then applications that search and retrieve relevant data at inference.

RAG Workflow

There are often multiple components of a RAG-based architecture, including dependent workflows such as preparing data from external sources. At a high level, there are two common workflows to consider—preparation of data from external knowledge sources, then the integration of that data into consuming applications, as shown in Figure 9-3.

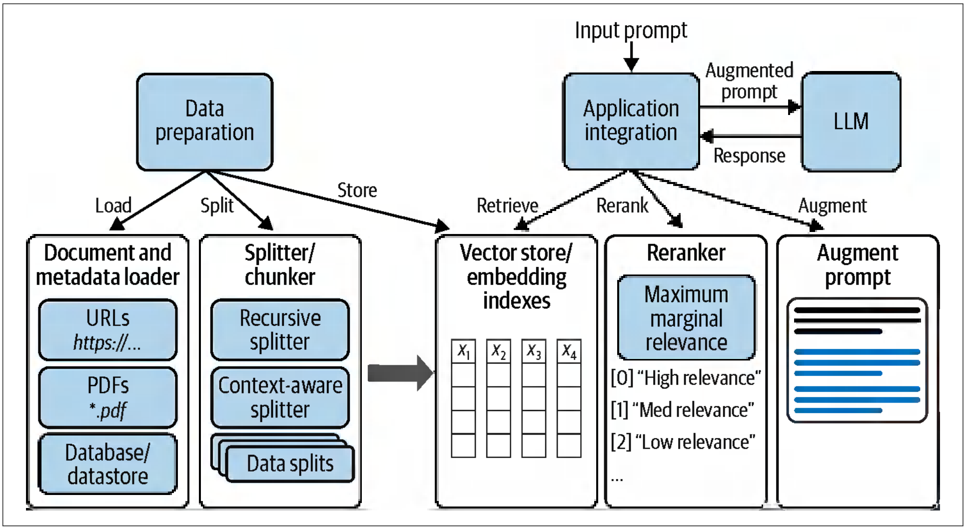


Figure 9-3. RAG architectures depend on efficient data preparation and retrieval techniques for integration into consuming applications

Data preparation involves the ingestion of data sources as well as the capturing of key metadata describing the data source. This may include tasks specific to the type of information source being utilized. As an example, if the information source is a PDF, there will be an additional task to extract text from those documents. This may not always be needed if the data is already in a consumable format; however, the preparation of data is often a prerequisite in RAG-based architecture to prepare the data for retrieval.

Application integration involves retrieving the most semantically similar information from those external data sources based on an input prompt. This is often followed by a reranking process to further refine the retrieved results and rank them in order of relevance to the input prompt. The final step is augmenting the input prompt with the most relevant information retrieved from external knowledge sources prior to using that augmented prompt to call the LLM, which returns the final completion.

To dive further into a specific example, the remainder of this section will specifically focus on information retrieval from documents. Let's start with the data preparation task, which includes extracting text from documents and efficiently storing that text for retrieval.

Document Loading

Although RAG-based architectures can pull data from a number of relevant information sources, we'll focus specifically on information retrieval from documents. A common implementation for document search and retrieval includes storing your documents in a vector store, where each document is indexed based on an embedding vector produced by an embedding model. The vector embedding includes the numeric representations of text data within your documents.

Each embedding aims to capture the semantic or contextual meaning of the data. The idea here is that semantically similar concepts end up close to each other (have a small distance between them) in the vector space, as discussed in Chapter 3. As a result, information retrieval involves finding nearby embeddings that are likely to have similar contextual meaning.

Each vector embedding is put into a vector store, often with additional metadata such as a reference to the original content the embedding was created from. The vector store then indexes the vectors, which can be done using a variety of approaches.

This indexing allows for quick retrieval of documents. The vector store, shown in Figure 9-4, is then used within the prompt workflow to efficiently retrieve external information based on an input query during inference.

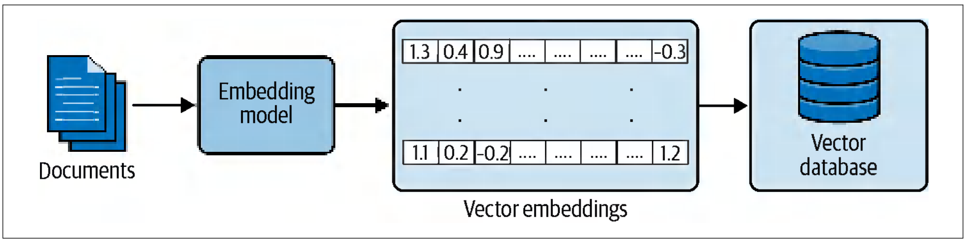


Figure 9-4. Efficient indexing of documents for quick retrieval

Creating vector embeddings that store numeric representations of text data in vector stores provides for efficient document search and retrieval techniques in RAG architectures. However, documents are often large and contain varied degrees of related information on a variety of topics, some more related than others. As an example, if you used the AWS product documentation for Amazon SageMaker, you'll notice that some of the text in that document is more semantically similar than others. As a result, you need to consider efficient strategies for optimizing the storage and retrieval of these documents as well as minimizing the risk of losing context.

Because LLMs have fixed context window limitations, you also need to develop document storage and retrieval strategies that consider those limitations.

Chunking

A technique called *chunking* is typically used in building document indexes (as well as searching, which is covered later in the section). Chunking breaks down larger pieces of text into smaller segments, as shown in Figure 9-5.

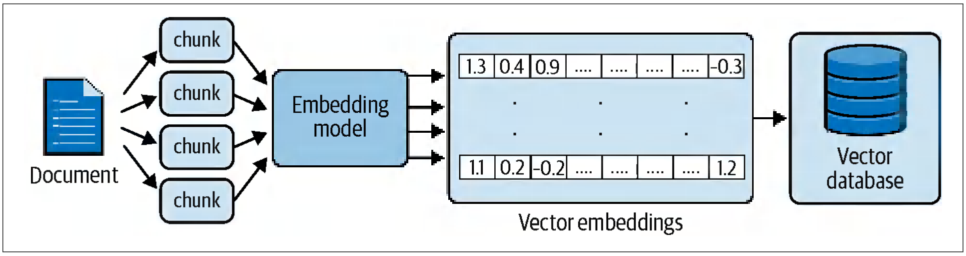


Figure 9-5. Chunking when storing and indexing documents

The chunks should contain information that is semantically related and that has meaningful context in that single chunk. There are different methods of chunking available. For example, you can use fixed-size chunking that splits data using a fixed number of tokens, which is an easy method and computationally efficient. Alternatively, you can use context-aware chunking methods, which aim to chunk data with more consideration around understanding the context of the data and keeping relevant text together.

When choosing a chunking strategy, there are a few considerations to keep in mind. First, consider the size of your indexed content, whether it's long documents such as books or shorter content like product reviews. Chunking smaller content may not have much impact, while chunking larger documents is not only necessary but also improves the ability to search for similar relevant information related to a search.

Next, as previously mentioned, chunking may be required due to context window limits imposed by the LLM. For example, if your model only supports 4,096 input tokens in the context window, you will need to adjust your chunk size to account for this limit.

Finally, there is a concept called *overlap*, which refers to the overlap of a defined amount of text between chunks. Overlap can help preserve context between chunks. This is another parameter to experiment with when choosing a chunking size.

After documents have been prepared by extracting text and loading their vector representations into a vector store, they are ready for integration into the application.

Document Retrieval and Reranking

Once the text from a document has been embedded and indexed, it can then be used to retrieve relevant information by the application. Remember, with RAG-based architectures, the information retrieved will later be used in the workflow to augment the input prompt with additional context prior to calling the LLM.

Let's look at the application workflow with a specific example where the input prompt includes the question, "What group is responsible for maintenance on product Flash-Tag?" In this case, product FlashTag support information is proprietary information that the LLM has no knowledge of, so RAG will be used to augment the prompt with additional information prior to calling the LLM.

To support the RAG architecture, the prompt text will first utilize an embedding model to create vector embedding representations of the prompt input. The vector embeddings will then be used to query the vector store for embeddings that are semantically similar to those on the input prompt. Based on those results, relevant document text is retrieved, as shown in Figure 9-6.

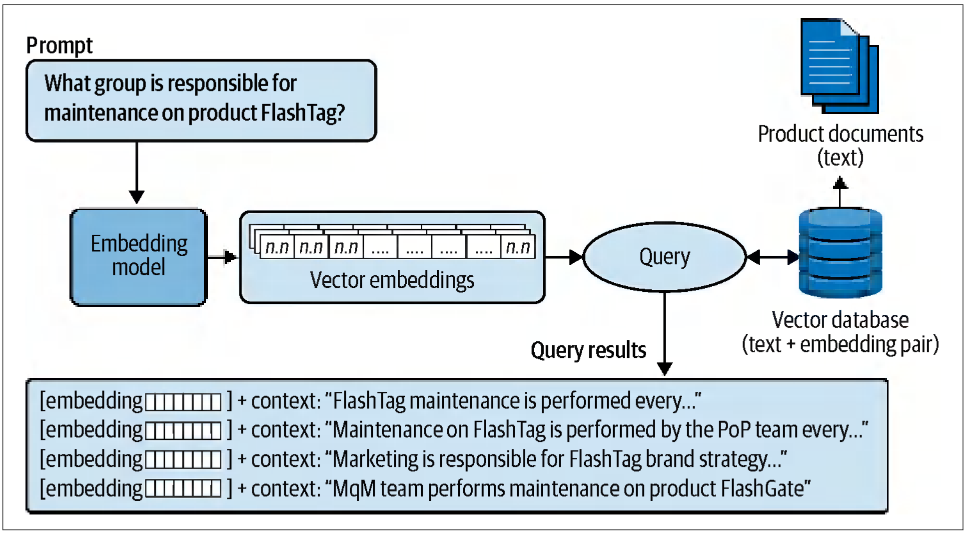


Figure 9-6. Information retrieval based on prompt input

You may also want to rerank the similarity results returned from the vector store to help diversify the results beyond just the similarity scores and improve relevance to the input prompt, as shown in Figure 9-7.

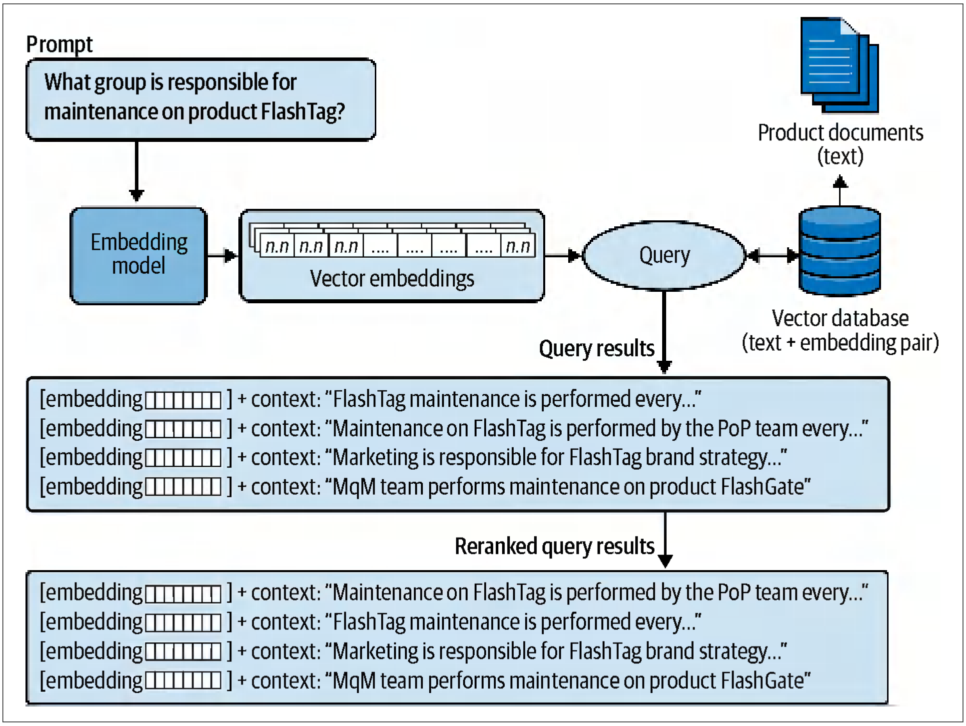


Figure 9-7. Reranking query results before augmenting the prompt

There are different ways to implement reranking, but the intent with ranking the retrieved results is to further refine the query results returned. There are different implementations that can be used to rerank retrieved results.

A popular reranking algorithm that is built into most vector stores is Maximum Marginal Relevance (MMR). MMR aims to maintain relevance to the input prompt but also reduce redundancy in the retrieved results since the retrieved results can often be very similar. This helps to provide context in the augmented prompt that is relevant as well as diverse.

Once information has been retrieved, and potentially reranked, the next step is to provide this additional context to the LLM by augmenting the input prompt with the additional contextual information.

Prompt Augmentation

Once the relevant contextual data has been retrieved, the next step in the RAG-based workflow is to use the additional context retrieved to augment the prompt. The input prompt of "What group is responsible for maintenance on product FlashTag?" can now be augmented with additional context retrieved from domain-specific information sources, as shown next.

Augmented prompt:

```
What group is responsible for maintenance on product FlashTag?
FlashTag maintenance is performed every Saturday with no downtime by the PoP
Team. PoP team is responsible for sending automated notifications.
```

Completion prompt:

```
The PoP Team is responsible for Product Maintenance on the FlashTag product.
```

This augmented prompt now has contextual information specific to the indexed documents as well as the original prompt. Because the documents are domain-specific and not within the LLMs training corpus, this method allows you to provide additional context to the model that would otherwise be unknown. The LLM is now able to use the information in the context of the prompt to generate a completion that likely contains a more relevant answer and avoids hallucinations.

RAG Orchestration and Implementation

In the previous section, you explored RAG as a framework for augmenting a model with external knowledge. To illustrate this, we walked through a RAG workflow to incorporate external knowledge specifically from documents by preparing the data for retrieval, then integrating the retrieval, rerank, and prompt augmentation into the consuming application. There are multiple ways to implement RAG-based architectures. This section will highlight specific techniques for orchestrating RAG workflows.

Multiple components are required to support RAG-based architectures and implement RAG, including data preparation workflows. Data preparation workflows include the tasks required to load and prepare in an optimized format for retrieval.

Additionally, workflows are also required to integrate RAG within applications. There are multiple steps required to implement RAG as part of application integration, including the steps required to embed the input prompt, retrieve relevant data, augment the prompt, and then call the LLM using the augmented prompt. All of these steps require a component that can orchestrate the tasks required, as shown in Figure 9-8.

Luckily, there are frameworks developed that take some of the heavy lifting away in implementing these solutions. This section explores a popular framework called LangChain, which provides you with modular pieces that contain the components necessary to work with large language models and implement techniques such as RAG.

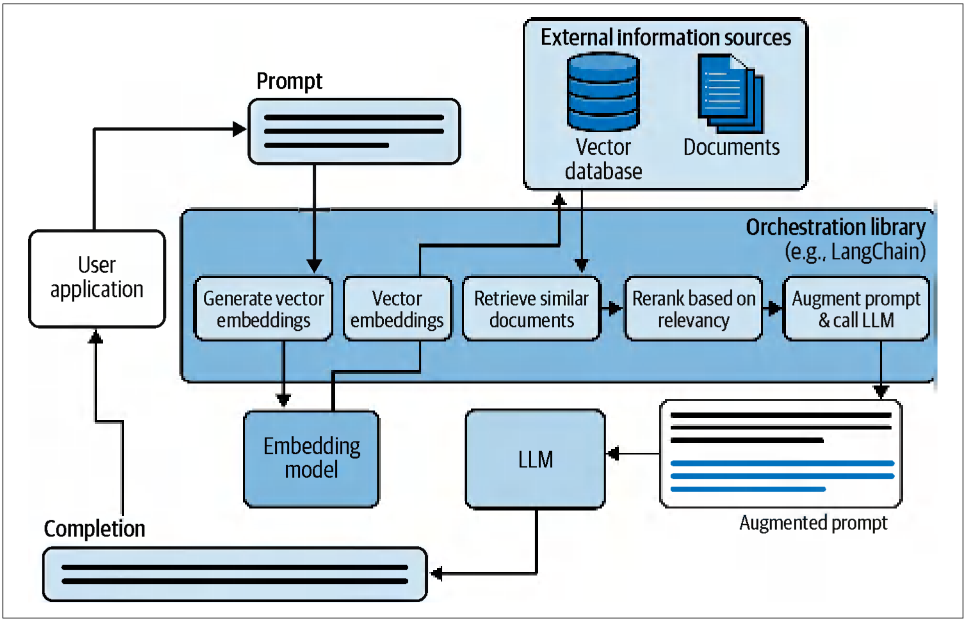


Figure 9-8. Orchestrating RAG workflows

There are a growing number of RAG orchestration frameworks. Choose the one that best supports your use case, orchestration needs, and data source integrations. RAG architectures can also be implemented through do-it-yourself orchestration code that directly calls the various APIs, vector stores, and data sources. While direct API calls may improve performance, they also require quite a bit more coding and maintenance than using an existing orchestration framework.

LangChain is composed of modules, interfaces, and integrations to support the development of context-aware reasoning applications and end-to-end workflows. These workflows include document loading, chunking, and retrieving from various vector stores, which you will learn about in the next few sections.

Document Loading and Chunking

LangChain provides document loaders as part of the data connector modules. These provide libraries for loading data across a variety of input formats into documents. For example, you can use PyPDFLoader to load and split PDF-formatted documents.

The previous section discussed the challenge of context window length and strategies around chunking, or splitting data, as a way to overcome context window limitations. LangChain also provides document transformers that include splitters, allowing you to chunk your documents using simple configurations, as shown in this code example. Here, we are using a dataset of annual Amazon Shareholder Letters:

```
import numpy as np
from langchain.text_splitter \
  import RecursiveCharacterTextSplitter
from langchain.document_loaders import PyPDFLoader

data_root_path = "./data"
filenames = glob.glob(data_root + '*.pdf')

documents = []
for file in filenames:
    loader = PyPDFLoader(data_root + file)
    document = loader.load()
    for document_fragment in document:
        # Extract year from filename
        year = filename.split('-').split()[1]
        # Set metadata
        document_fragment.metadata = {"year": year,
          "source": filename)

    documents += document

# Chunk the docs
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size = 512,
    chunk_overlap = 100,
)

docs = text_splitter.split_documents(documents)
```

Here, the code loads the PDF documents from the designated location and splits the documents into chunks of 512 characters. These chunks contain portions of the original PDF document that can be preprocessed to create vector embeddings using an embedding model, then stored in a vector store or loaded into a vector store using one of the many third-party integrations provided by the LangChain framework.

Note that this code adds metadata to each document upon ingestion. This metadata will be used later to filter the results and speed up the overall retrieval process by narrowing the search results to a given year, for example.

Embedding Vector Store and Retrieval

As previously mentioned, a vector store saves vector embeddings and creates indexes to enable fast retrieval lookups and similarity searches. Similarity search is a common use case for vector stores, as you are trying to augment your prompt with additional, relevant information for the LLM to use in its context when generating a completion.

Since AWS provides a variety of options for storing vector embeddings, let's briefly dive into the available options and considerations for each. It's also important to evaluate each service for the most current list of capabilities in terms of decision points, such as supported search algorithms or fit for your use case.

You can use Amazon OpenSearch Service to store embeddings combined with the k-Nearest Neighbor (k-NN) plugin for OpenSearch to perform fast document-similarity searches across the embeddings. Specifically, the Vector Engine for Amazon OpenSearch Serverless provides serverless vector storage with similarity search capabilities with the ability to add, update, and delete vector embeddings in near real time. OpenSearch implements optimized and scalable retrieval algorithms, including the Facebook AI Similarity Search (FAISS) vector store and retrieval algorithms from Meta/Facebook. This option also provides the ability to scale your vector store cluster horizontally as needed based on your workload.

Other options include Amazon Aurora PostgreSQL and Amazon Relational Database Service (RDS) for PostgreSQL. Both offer pgvector support, which can be a natural fit for teams that already have PostgreSQL installations or skill sets. pgvector is a community-maintained vector store plugin for PostgreSQL. These options are also scalable.

Amazon Kendra is a managed solution specifically designed for search and retrieval, including built-in connectors to popular data sources such as Amazon S3, Microsoft SharePoint, Salesforce, ServiceNow, and Zendesk. In addition, Kendra supports a variety of document formats, including HTML, PDF, and CSV without having to manually convert your documents into embedding vectors. Amazon Kendra also allows you to enrich your documents with additional metadata to improve search-result relevance by allowing metadata filtering during a query.

All of the AWS options for vector storage and retrieval can be included as part of a RAG-based architecture. Building on AWS also allows for the flexibility to utilize the vector store that best meets the needs of your use case and tooling choices.

Next, you will see an example using FAISS as the vector store and retrieval mechanism, along with LangChain as the orchestrator of the various components involved in the workflow. Other examples, including Amazon OpenSearch, Amazon Aurora/RDS for PostgreSQL, and Kendra, are included in the GitHub repository for this book.

Let's look at a specific example utilizing LangChain to build and orchestrate the tasks required to take the loaded data, create embedding vectors, then populate a vector store that will later be used for retrieval.

LangChain integrates with many vector stores, such as ElasticSearch, OpenSearch, Pinecone, and Facebook AI Similarity Search (FAISS). For simplicity, let's show an example of LangChain's integration directly with the FAISS vector store and retrieval library using an embedding model deployed as an Amazon SageMaker endpoint from SageMaker JumpStart. Note that you can use a local model, as well, from the Hugging Face model hub; for example:

```
from langchain.vectorstores import FAISS
from langchain.embeddings import SagemakerEndpointEmbeddings
from langchain.embeddings.sagemaker_endpoint import \
  EmbeddingsContentHandler
from sagemaker.jumpstart.model import JumpStartModel

embedding_model_checkpoint = "..." # embedding model

embedding_model =
  JumpStartModel(model_id=embedding_model_checkpoint)\
  .deploy()

embeddings_content_handler = EmbeddingsContentHandler()

embeddings = SagemakerEndpointEmbeddings(
    endpoint_name=embedding_model.endpoint_name,
    content_handler=embeddings_content_handler
)

# Load the FAISS vector store with the documents
vector_store = FAISS.from_documents(docs, embeddings)
query = "How has AWS evolved?"

results_with_scores = vector_store.similarity_search_with_score(
  query)

for doc, score in results_with_scores:
    print(f"Content: {doc.page_content}")
    print(f"Metadata: {doc.metadata}")
    print(f"Score: {score}\n\n")
    print('----')
```

Output:

```
Content: AWS is still in the early stages of its evolution, and has a chance
for unusual growth in the next decade.
Metadata: {'year': 2022, 'source': 'AMZN-2022-Shareholder-Letter.pdf'}
Score: 0.5685306191444397
----
Content: AWS continues to deliver new capabilities rapidly (over 3,300 new
features and services launched in 2022), and invest in long-term inventions
```

```
that change what’s possible.
Metadata: {'year': 2022, 'source': 'AMZN-2022-Shareholder-Letter.pdf'}
Score: 0.7789842486381531
----
Content: We made the long-term decision to continue investing in AWS. Fifteen
years later, AWS is now an $85B annual revenue run rate business with strong
profitability.
Metadata: {'year': 2022, 'source': 'AMZN-2022-Shareholder-Letter.pdf'}
Score: 0.7893760204315186
----
Content: This shift by so many companies (along with the economy recovering)
helped re-accelerate AWS’s revenue growth to 37% YoY in 2021.
Metadata: {'year': 2021, 'source': 'AMZN-2021-Shareholder-Letter.pdf'}
Score: 0.7898486852645874
----
```

You can also add a metadata filter to only retrieve documents from the year 2022, for example. Simply add a dictionary with the filter values and rerun the retrieval, as shown next. Here, you’ll see that the retrieval only returns documents from the year 2022:

```
filter={"year": 2022}
results_with_scores = vector_store.similarity_search_with_score(
  query, filter=filter)
for doc, score in results_with_scores:
    print(f"Content: {doc.page_content}")
    print(f"Metadata: {doc.metadata}")
    print(f"Score: {score}\n\n")
    print('----')
```

Output:

```
Content: done innovating here, and this long-term investment should prove
fruitful for both customers and AWS. AWS is still in the early stages of its
evolution, and has a chance for unusual growth in the next decade.
Metadata: {'year': 2022, 'source': 'AMZN-2022-Shareholder-Letter.pdf'}
Score: 0.5685306191444397
----
Content: AWS continues to deliver new capabilities rapidly (over 3,300 new
features and services launched in 2022), and invest in long-term inventions
that change what’s possible.
Metadata: {'year': 2022, 'source': 'AMZN-2022-Shareholder-Letter.pdf'}
Score: 0.7789842486381531
----
Content: AWS is now an $85B annual revenue run rate business, with
strong profitability, that has transformed how customers from start-ups to
multinational companies to public sector organizations manage their technology
infrastructure.
Metadata: {'year': 2022, 'source': 'AMZN-2022-Shareholder-Letter.pdf'}
Score: 0.7893760204315186
----
Content: Customers have appreciated this customer-focused, long-term approach,
and we think it’ll bode well for both customers and AWS.
```

```
Metadata: {'year': 2022, 'source': 'AMZN-2022-Shareholder-Letter.pdf'}
Score: 0.8272767066955566
----
```

Now that the vector store has been created and loaded with documents and metadata, let's switch to the integrated application that will retrieve relevant data and use the data to augment the input prompt with additional context prior to calling the LLM to complete the prompt.

Retrieval Chains

Chains allow you to create a sequence of calls to different components to retrieve data used to augment the prompt. Creating and executing the sequence of steps requires orchestrating the end-to-end workflow. The LangChain framework, designed to enable context-aware reasoning applications, provides many integrations that greatly simplify this workflow.

The following code example shows how to use a built-in chain called `RetrievalQA` from the LangChain framework, along with `PromptTemplate` to format the prompt and `SagemakerEndpoint` to use as the LLM. This chain retrieves relevant documents from the vector store and specifies the type of search to perform—similarity search for the top three most relevant documents, in this case:

```
from langchain.chains import RetrievalQA
from langchain.prompts import PromptTemplate
from langchain import SagemakerEndpoint
prompt_template = """
User: Use the following pieces of context to provide a concise answer
to the question at the end. If you don't know the answer, just say
that you don't know, don't try to make up an answer.

{context}

Question: {question}
Assistant:
"""

prompt = PromptTemplate(
    template=prompt_template,
    input_variables=["context", "question"]
)
llm_model_checkpoint = "..." # generative model like Llama2

llm_model =
  JumpStartModel(model_id=llm_model_checkpoint)\
  .deploy()

llm = SagemakerEndpoint(
  endpoint_name=llm_model.endpoint_name)
```

```
qa_chain = RetrievalQA.from_chain_type(
    llm=llm,
    chain_type="stuff", # stuff the prompt
    retriever=vector_store.as_retriever(
        search_type="similarity",
        search_kwargs={"k": 3}
    ),
    return_source_documents=True,
    chain_type_kwargs={"prompt": prompt}
)

query = "How has AWS evolved?"
result = qa_chain({"query": query})
print(result["result"])

print('----')
print(f'Context Documents: ')
for source_doc in result["source_documents"]:
      print(f'{source_doc}\n')
      print('----')
```

After retrieving the top three documents, LangChain includes (or "stuffs," as you see in the `chain_type="stuff"` parameter) them with the prompt to provide additional context in the prompt, which helps the LLM better answer the given question.

Output:

```
Based on the provided context, AWS has evolved in the following ways:
1. Rapid innovation: AWS continues to deliver new capabilities rapidly,
launching over 3,300 new features and services in 2022 alone.
2. Long-term investment: AWS has made a long-term decision to continue
investing in its infrastructure, even during challenging times such as the
2008-2009 recession.
3. Expansion of services: AWS has expanded its offerings beyond just computing
and storage, now providing a wide range of services including analytics,
machine learning, and security.
4. Increased profitability: Despite continued investment in innovation, AWS has
achieved strong profitability, with an $85B annual revenue run rate business.
5. Shift to cloud adoption: The pandemic has accelerated the shift to cloud
adoption, with many companies deciding to move their technology infrastructure
to the cloud. This has helped re-accelerate AWS's revenue growth to 37% YoY in
2021.
Overall, AWS has evolved from a niche player in the cloud computing market to a
dominant force, with a strong track record of innovation and investment in its
infrastructure.
----
Context Documents:
page_content='done innovating here, and this long-term investment should prove
fruitful for both customers and AWS. AWS is still in the early stages of its
evolution, and has a chance for unusual growth in the next decade.'
metadata={'year': 2022, 'source':
  'AMZN-2022-Shareholder-Letter.pdf'}
----
```

```
page_content='AWS continues to deliver new capabilities rapidly (over 3,300 new
features and services launched in 2022), and invest in long-term inventions
that change what’s possible.'
metadata={'year': 2022, 'source':
  'AMZN-2022-Shareholder-Letter.pdf'}
----
page_content='AWS is now an $85B annual revenue run rate business, with
strong profitability, that has transformed how customers from start-ups to
multinational companies to public sector organizations manage their technology
infrastructure.'
metadata={'year': 2022, 'source':
  'AMZN-2022-Shareholder-Letter.pdf'}
----
```

Notice how the LLM constructs a nicely formatted answer to the question using the additional context provided from the chain. Next, you will see how to rerank the retrieved documents to potentially improve the augmented prompt and therefore the generated response.

Reranking with Maximum Marginal Relevance

You may want to experiment with techniques like MMR to diversify the results retrieved from the vector store. MMR encourages diversity in the result set, which allows the retriever to consider more than just the similarity scores, but also include a diversity factor between `0` and `1`, where `0` is maximum diversity and `1` is minimum diversity. Here is the code using FAISS and MMR (`search_type="mmr"`) with a diversity factor of `lambda_mult=0.1` for a relatively high degree of diversity in the results:

```
qa_chain = RetrievalQA.from_chain_type(
    llm=llm,
    chain_type="stuff",
    retriever=vector_store.as_retriever(
        search_type="mmr", # Maximum Marginal Relevance (MMR)
        search_kwargs={"k": 3, "lambda_mult": 0.1}
    ),
    return_source_documents=True,
    chain_type_kwargs={"prompt": prompt}
)

query = "How has AWS evolved?"
result = qa_chain({"query": query})
print(result["result"])

print('----')
print(f'Context Documents: ')
for source_doc in result["source_documents"]:
      print(f'{source_doc}\n')
      print('----')
```

Output:

```
Based on the context provided, AWS has evolved in the following ways:
1. Innovation: AWS has continued to innovate and invest in new technologies and
services, as evident from the statement "AWS is still in the early stages of
its evolution, and has a chance for unusual growth in the next decade."
2. Efficiency: AWS is inherently more efficient than traditional in-house data
centers, according to the statement. This is due to two factors:
a. Institutions: Many institutions, including schools and governments, are
transitioning from in-person to virtual classrooms and running on AWS to ensure
continuity of learning.
b. Secure platform: Governments are leveraging AWS as a secure platform to
build out new capabilities in their efforts to end the pandemic.
Therefore, AWS has evolved to become a more efficient and secure platform for
various institutions and governments.
----
Context Documents:
page_content='AWS is still in the early stages of its evolution, and has a
chance for unusual growth in the next decade.'
metadata={'year': 2022, 'source':
  'AMZN-2022-Shareholder-Letter.pdf'}
----
page_content='AWS is also inherently more efficient than the traditional in-
house data center.'
metadata={'year': 2019, 'source':
  'AMZN-2019-Shareholder-Letter.pdf'}
----
page_content='Institutions around the world are transitioning from in-person
to virtual classrooms and are running on AWS to help ensure continuity of
learning.'
metadata={'year': 2019, 'source':
  'AMZN-2019-Shareholder-Letter.pdf'}
----
```

Here, you see that the retriever, configured with a relatively high MMR diversity factor, reranked the results from the vector store and included shareholder letters from 2019 to help answer the question, "How has AWS evolved?"

In the next section, you will learn to further extend your models' capabilities by allowing them to interact with their environment using agents and frameworks such as ReAct and PAL.

Agents

Consider a generative AI-based travel application that can not only respond to the question, "Which beaches should I visit in Hawaii?" with a list of suggestions but can also book the flight and hotel for you.

For this to work, you need an additional piece of software, usually referred to as an *agent*, that orchestrates the prompt-completion workflows between the user request, the foundation model, and external data sources and applications, as shown in Figure 9-9.

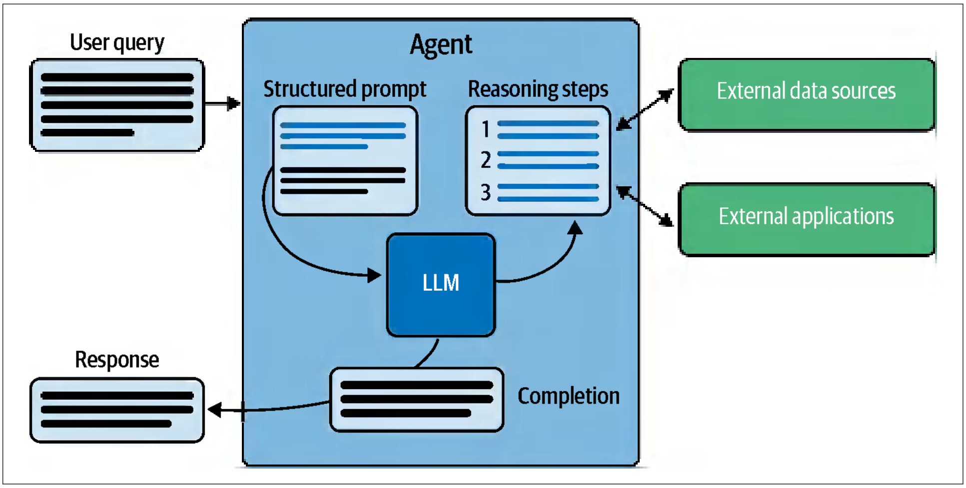


Figure 9-9. Agents orchestrate prompt-completion workflows between user requests, the foundation model, and external data sources and applications

Agents use the foundation model as their reasoning engine. Building upon the chain-of-thought (CoT) prompting that you explored in Chapter 2, some models are capable of generating step-by-step action plans carried out by tools such as a web search, a SQL query, or a Python-based calculator script, for example.

Agents automatically build structured prompts similar to CoT prompts to help the model reason through the user requests and create those step-by-step action plans. The agent then orchestrates a RAG workflow through a sequence of data lookups and/or performs API calls to complete the actions for the user. The actions an agent is allowed to take are defined in separate instructions that are prepended to the prompt.

The agent automatically augments the prompt with the information received from the external systems to help the model generate more context-aware and relevant completions, then returns the final response back to the user.

Agent implementations are available in many popular open source libraries, such as LangChain Agents or Hugging Face Transformers Agents. On AWS, you can also choose from fully managed services such as agents for Amazon Bedrock, which is covered in more detail in Chapter 12.

Let's explore the structured prompts in more detail. Agents often use a ReAct framework to show the model how to reason through a problem and decide on actions to take that help find a solution.

ReAct Framework

ReAct is a prompting strategy that combines CoT *reasoning* with *action planning*. ReAct structures prompts to include a sequence of one or more question, thought, action, and observation examples as described in the ReAct paper and shown in Figure 9-10.

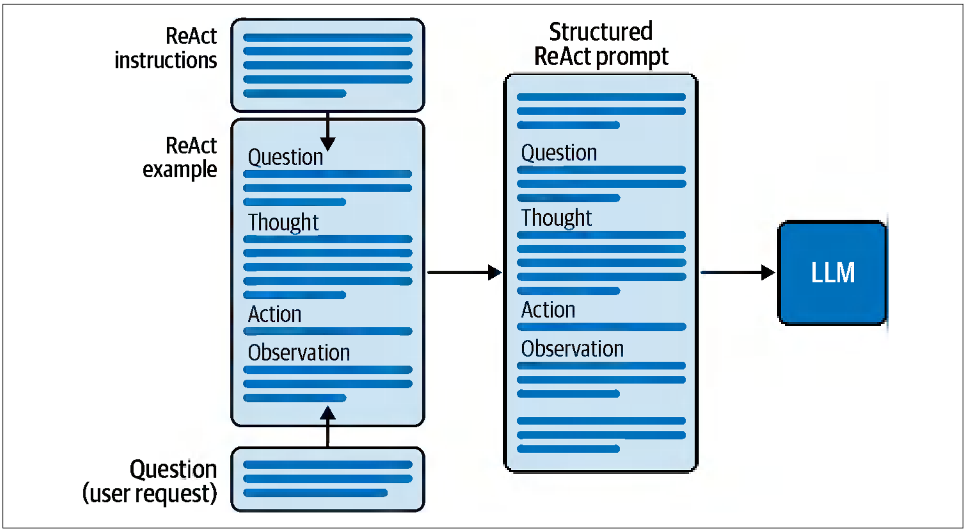


Figure 9-10. ReAct structures prompts to include instructions, ReAct examples, and the user request

The question is the user-requested task or problem to solve. The thought is a reasoning step that helps demonstrate to the foundation model how to tackle the problem and identify an action to take. The action is an API that the model can invoke from an allowed set of APIs. The observation is the result of carrying out the action. The actions that the model is able to choose from are defined by a set of instructions that are prepended to the example prompt text.

Let's return to the generative AI-based travel application example and assume a user is asking which hotel is closest to the most popular beach in Hawaii. This question will take a couple of intermediate steps and actions to find the solution. In the prompt-prepended instructions, describe the ReAct prompt structure and list the allowed actions. Let's give the agent API access to a Wikipedia search and a company-owned hotel database:

```
Solve a question answering task with interleaving Thought, Action, Observation
steps.

Thought can reason about the current situation, and Action can be three types:
(1) wikipedia_search[topic], which searches the topic on Wikipedia and returns
```

```
the first paragraph if it exists. If not, it will return a similar topic to
search.
(2) hotel_database_lookup[request], which performs an API call to the hotel
database to gather hotel information defined in request
(3) Finish[answer], which returns the answer and finishes the task.
```

In this example, you first defined the task by telling the model to answer a question using the discussed ReAct prompt structure. Then, you provided instructions that explain what "thought" means and listed the allowed actions to take.

First in the list is the `wikipedia_search` action, which looks for Wikipedia entries related to the specified topic. The second is a `hotel_database_lookup` action, which can query the travel companies' hotel database with a specific request. The last action is `finish`, which returns the answer and brings the task to an end.

You can also add reasoning examples to the instructions. Depending on the foundation model you are working with, you may need to include more than one example and carry out few-shot inference.

Now, let's see how the model applies the instructions to the user's request to find the closest hotel to the most popular beach in Hawaii:

```
Question: Which hotel is closest to the most popular beach in Hawaii?
Thought 1: I need to search for the most popular beach in Hawaii and find the
closest hotel for that location.
Action 1: wikipedia_search["most popular beach in Hawaii"]
Observation 1: Waikiki is most famous for Waikiki Beach.
Thought 2: I need to find the hotel closest to Waikiki Beach.
Action 2: hotel_database_lookup["hotel closest to Waikiki Beach"]
Observation 2: <MyDreamHotel> is closest to Waikiki Beach.
Thought 3: <MyDreamHotel> is closest to Waikiki Beach, the most popular beach
in Hawaii. So the answer is <MyDreamHotel>.
Action 3: Finish["MyDreamHotel"]
```

You can see how the thoughts reason through the task and plan two intermediate steps that help find the answer. The model then decides on appropriate actions to take from the list of allowed actions. The observations bring the new information retrieved from the actions back into the model's prompt context. The model will cycle through as many iterations as needed to find the answer. The final action is then to finish the cycle and pass the answer back to the user.

Your context-aware reasoning application is now able to connect to external data sources to retrieve additional information, reason through tasks, and plan and perform tasks. But what if one of the tasks is to calculate the sales tax for the travel booking? Even with CoT, the model's ability to perform arithmetic or other mathematical operations is limited. After all, generative foundation models are not really doing math, they are just predicting the most probable next token to complete the prompt.

To overcome this limitation, you can connect the model to an application that's good at performing calculations, such as a code interpreter. The Program-Aided Language Models framework does exactly that.

Program-Aided Language Framework

PAL uses CoT reasoning to generate programs in the intermediate reasoning steps that help solve the given problem. These programs are then passed to an interpreter, for example, a Python interpreter, that runs the code and returns the result back to the foundation model (FM), as shown in Figure 9-11.

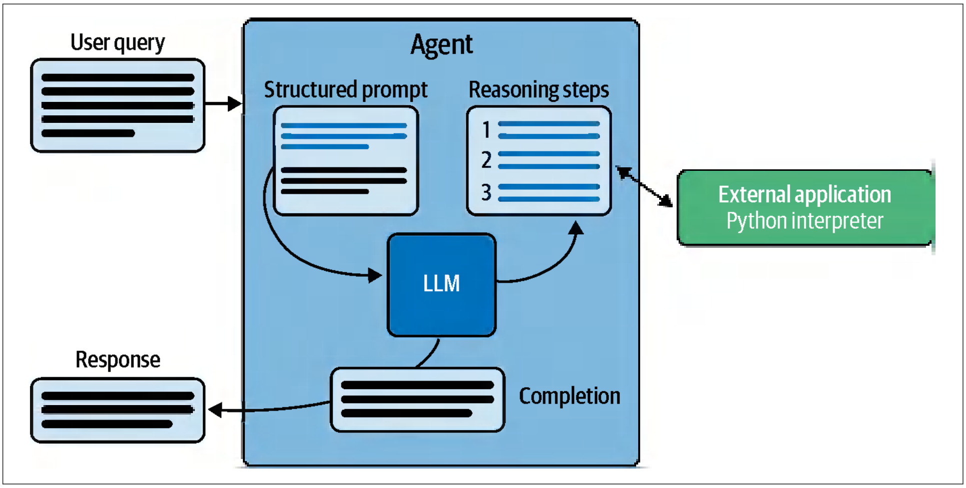


Figure 9-11. PAL connects a foundation model to an external code interpreter to perform calculations

Similar to ReAct, you need to add one or more examples to the prompt that shows the model how to format the output. Start each example with a question followed by a couple of reasoning steps and lines of Python code that solve the problem. Then, add the new question to solve to the prompt. The PAL-formatted prompt now contains your example(s) and the new problem to solve, as shown in Figure 9-12.

Once you pass this prompt to the FM, the model follows the example and generates a completion in the form of a Python script. Next, send the script to a Python interpreter that will run the code and return the result. You can now append the result to the prompt, and the LLM generates a completion that contains the correct answer.

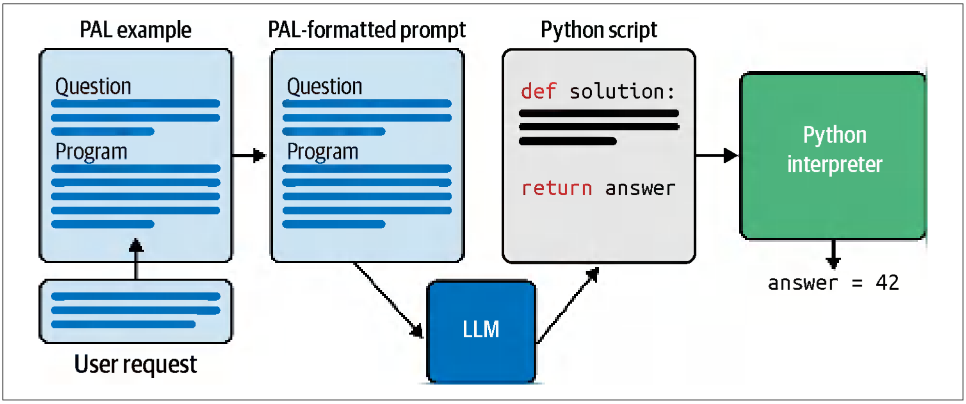


Figure 9-12. PAL workflow connecting the LLM to a Python interpreter

Here is an example PAL prompt template:

```
Translate a math problem into an expression that can be executed using Python's
numexpr library.
Use the output of running this code to answer the question.
Question: ${{Question with hard calculation}}
${{Code that prints what you need to know}}
Question: I have four bananas and buy three more, how many bananas do I have?
def solution():
    initial_bananas = 4
    extra_bananas = 2
    return initial_bananas + extra_bananas
Question: {question}
```

You can now ask the model a similar math question and the model will follow the provided example.

Prompt:

```
Antje has five times the number of books than Chris. Chris has 3 books. How
many books does Antje have?
```

Completion:

```
def solution():
    books_chris = 3
    books_antje = 5 * books_chris
    return books_antje
```

For simple math operations, you can likely get the correct answer by just applying CoT reasoning. But for more complex math, such as arithmetic with large numbers, trigonometry, or calculus, PAL is a powerful technique that ensures that any calculations done by your foundation model are accurate and reliable.

The following code example shows how to use ReAct and PAL with LangChain Agents, including `serpapi` (which performs Google searches, among many other actions). Note the use of LangChain's integration with Hugging Face pipelines using the `HuggingFacePipeline` abstraction:

```
from langchain.agents import load_tools
from langchain.agents import initialize_agent
from langchain.agents import AgentType
from transformers import AutoTokenizer, AutoModelForCausalLM
from transformers import pipeline
from langchain.llms import HuggingFacePipeline

model_checkpoint = "..." # generative model like Llama2, Falcon

tokenizer = AutoTokenizer.from_pretrained(model_checkpoint)

model = AutoModelForCausalLM.from_pretrained(model_checkpoint)

pipeline = pipeline(
    "text-generation",
    model=model,
    tokenizer=tokenizer
)

llm = HuggingFacePipeline(pipeline=pipeline)

tools = load_tools(["serpapi", "llm-math"], llm=llm)

agent = initialize_agent(tools,
  llm, agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbose=True)

agent.run("""
Which hotel is closest to the most
popular beach in Hawaii, and how much
is each night with 50% discount?
""")
```

The output should look similar to this:

```
> Entering new AgentExecutor chain...
I need to find the most popular beach in Hawaii and find the closest hotel to
that beach and find out how much a hotel night is and then calculate 50% of
that price.
Action: Search
Action Input: "most popular beach in Hawaii"
Observation: Waikiki Beach
```

```
Thought: I need to find the closest hotel from Waikiki Beach
Action: Search
Action Input: "closest hotel from Waikiki Beach"
Observation: <MyDreamHotel>

Thought: I need to find out how much a hotel night is
Action: Search
Action Input: "How much is a hotel night at <MyDreamHotel>"
Observation: 250 USD

Thought: I need to calculate 50% of that price
Action: Calculator
Action Input: 250x0.5
Observation: Answer: 125

Thought: I now know the final answer
Final Answer: Waikiki Beach is the most popular beach in Hawaii and the closest
hotel is <MyDreamHotel> and a hotel night with 50% discount is 125 USD.
> Finished chain.

"Waikiki Beach is the most popular beach in Hawaii and the closest hotel is
<MyDreamHotel> and a hotel night with 50% discount is 125 USD."
```

With orchestration software, such as agents that take charge of prompt engineering and communication between systems, and advanced prompting strategies, such as CoT, ReAct, and PAL, that guide models to create step-by-step action plans, you can now build powerful context-aware reasoning applications.

To build end-to-end generative AI solutions, there are a few more components that you need in addition to RAG and agents that we have discussed so far.

For example, you need infrastructure to train, fine-tune, and serve your model, as well as host your application components. You might also need additional orchestration components, frameworks, model hubs, and application interfaces that allow your consumers, including users and systems, to interact with your solution. Let's dive into these additional components in the next section.

Generative AI Applications

Building robust generative AI applications involves multiple components beyond the generative model. For example, once the model has been tuned or augmented for a specific task, how will users interact with the model? What type of validation may need to be done and what additional components are needed to support that?

A generative AI application includes multiple components as part of the end-to-end solution. In some cases, you may use a managed generative AI application, such as Amazon CodeWhisperer, where these components are all packaged and made available to consumers. On the other hand, when you are building a new generative AI application, it's important to understand common components to consider. This section is not intended to be a deep dive into each component but to provide an introduction into several high level components, as shown in Figure 9-13 (seen earlier as Figure 1-5).

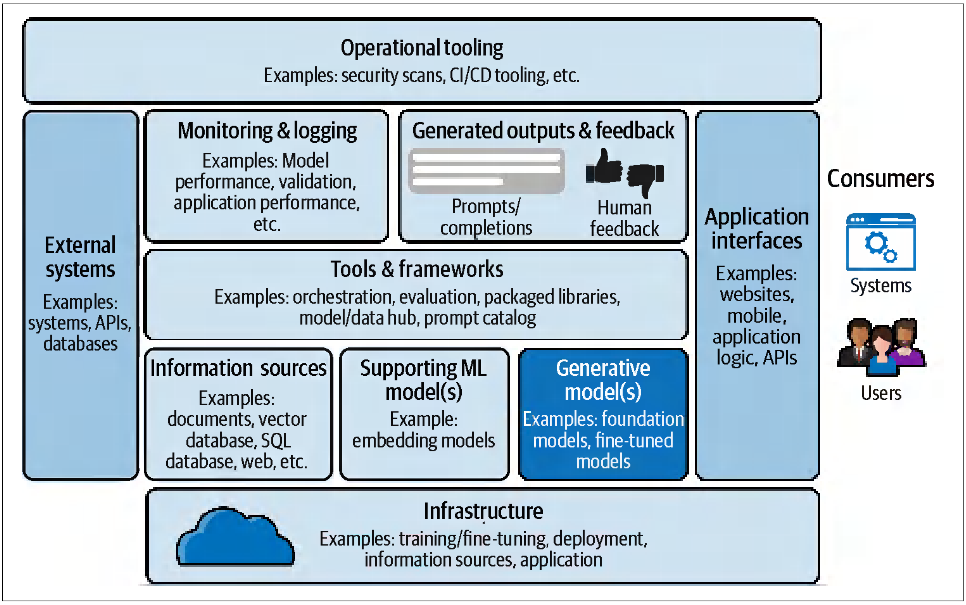


Figure 9-13. Generative AI applications include more than generative models

Infrastructure

At the base layer, infrastructure is a core component required not only for fine-tuning a model and deployment of a model but also for all of the other components supporting the end-to-end application. For example, the previous section discussed LangChain as a framework for orchestrating and implementing RAG. As part of a generative AI application, LangChain has to run on underlying infrastructure. A common implementation here includes deploying LangChain on AWS compute services, such as AWS Lambda, an event-driven, serverless compute platform, as shown in Figure 9-14.

Alternatively, if the chain includes a sequence of long-running processes, then you'd want to consider infrastructure that can better serve long-running processes, such as AWS Fargate, a serverless compute engine for containers.

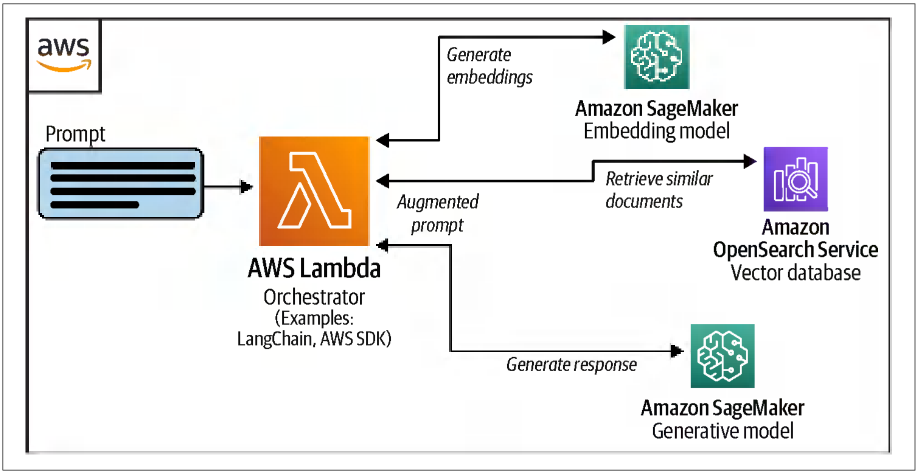


Figure 9-14. Infrastructure powers all application components

You should also consider implementing filtering logic, commonly called *guardrails*, to filter user prompts and model responses for sensitive or inappropriate content. You should consider serverless and managed options to reduce operational overhead.

AWS provides a variety of infrastructure options to support the various components of the application stack, allowing you to pick the optimal infrastructure option for each application component in terms of operational efficiencies, performance, and cost:

Generative models and supporting machine learning (ML) models
: These types of models are at the heart of generative AI applications. Generative models include foundation models as well as models that have been fine-tuned. These models are hosted on infrastructure, such as Amazon SageMaker.

To implement augmented solutions, such as RAG-based architectures, there is often a need to deploy other ML models that support the solution. An example discussed at length in this section is using an embedding model to embed the prompt text and use that to retrieve relevant document information from a vector store.

Information sources

Information sources are also a key part of a generative AI application. They may support RAG-based architectures, such as vector or SQL databases, or be used as part of a broader application.

As an example, a common pattern in generative AI applications includes the implementation of an LLM cache to store and serve cached responses from generative models. This cache can help in improving performance as well as reducing unnecessary API calls.

External systems

These include other systems the generative AI application interacts with, such as databases or APIs. Building agents-based applications that enable the generative model to take action may require dependencies on external systems to execute that action.

For example, building a chatbot that allows for the ability to make a reservation based on a generated travel recommendation will require the ability for the agent to interact with a reservation system to book the reservation.

Tools and frameworks

Typically, generative AI applications will rely on a number of tools and frameworks to build and integrate components, as well as to operate the end-to-end solution. Previous chapters have highlighted many examples from this category, such as utilizing model hubs to store, discover, and share foundation generative models as well as fine-tuned models.

As mentioned in Chapter 3, some popular model hubs include Hugging Face Model Hub and Amazon SageMaker JumpStart. Packaged libraries are another example within this category. Packaged libraries, such as Hugging Face's PEFT, help in simplifying the implementation of fine-tuning techniques such as LoRA. LangChain is another example of tooling that helps in implementing techniques such as RAG or agents using convenient packaged libraries.

Monitoring and logging

Operating a generative AI application requires monitoring and logging of all components that support the end-to-end system, including infrastructure, network, and security. This should also include ongoing monitoring of your models and key components of your RAG-based workflows. To start, you should identify a minimum set of error counters and logs that will help you troubleshoot operational issues. Next, you can add metrics to help improve the performance of your generative AI system. Remember that if you don't measure performance, you can't improve it.

Generated outputs and feedback

A key dependency for effective feedback monitoring of generative models typically includes implementing a component of the solution that can capture and store input prompts along with generated outputs and feedback. The input prompts and generated outputs are often cached to reduce the number of API calls required when invoking the model for the same input. The feedback mechanism should include guardrails to mitigate risk. At a high level, guardrails are implemented to provide a layer of safety between the consumers and the generative model.

For example, to monitor for signals of jailbreak, where a malicious user is trying to manipulate prompts and receive inappropriate responses, you need to capture the prompt and the response in order to detect a jailbreak scenario.

A generative AI application can also have an *application interface* that allows users or systems to interact with it. The interface can take many forms, such as a web based user interface, a mobile application, or an API. This layer also includes governance around usage of the generative application.

Figure 9-15 shows a modification of the previous image that represents a simplified representation of an application interface, in the form of a REST API, that is providing the interface between input prompts and the backend logic.

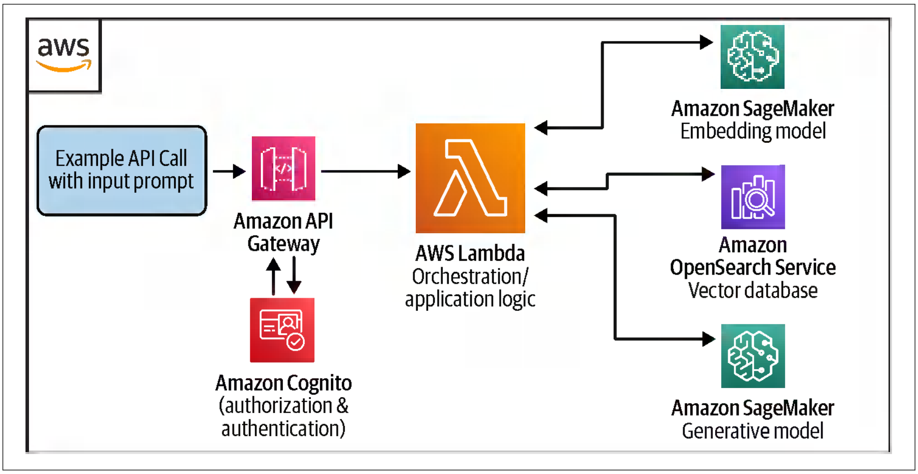


Figure 9-15. Creating an application interface in the form of a REST API

In this case, Amazon API Gateway is added to provide a REST API for the backend logic. This frontend interface can also provide low-latency responses on generated completions, manage incoming requests, monitor connections, scale or throttle traffic, and connect into authorizers, such as Amazon Cognito, to determine which users (or systems) should have access to your API as well as the level of access they should have.

Operational tooling
: Running any application at scale typically requires additional operational tooling used to manage the build, validation, and delivery of application components. The same is true for generative AI applications. For example, several components in the diagram require the provisioning and configuration of resources, which is typically handled through a combination of tooling such as traditional continuous integration (CI) and continuous delivery/deployment (CD) tooling.

Building generative AI applications on AWS typically involves multiple AWS services applied to the various application components, as shown in Figure 9-16 (shown earlier as Figure 1-6).

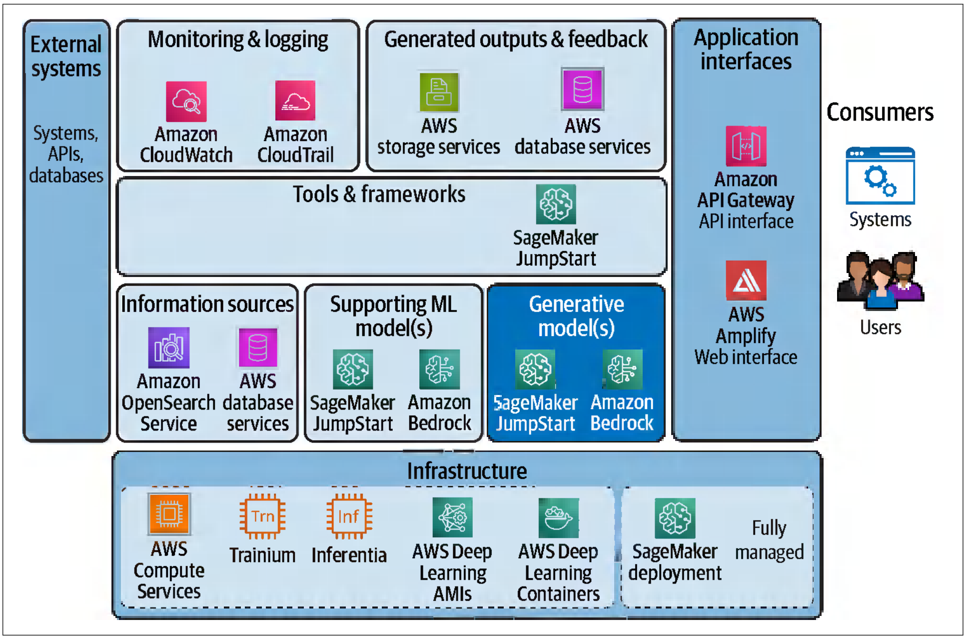


Figure 9-16. Examples of AWS services that can be used to build generative AI applications

As previously mentioned, there are prebuilt generative AI applications, such as Amazon CodeWhisperer, where all of the application components are abstracted away from the consumer and fully managed as part of a packaged application.

This section covered the core components of a typical generative AI application. Usually, there are a lot of integrations and dependencies required to build, deploy, and operate these applications. There are also considerations to be aware of when operationalizing the complete generative AI project life cycle to allow for reliable and repeatable processes (as discussed in Chapter 1).

The next section will discuss some of the considerations around operationalizing and creating efficiencies in the generative AI project life cycle.

FMOps: Operationalizing the Generative AI Project Life Cycle

An increasing number of generative models are powering critical applications. As a result, the need to build more reliable, efficient, and repeatable mechanisms to build, deploy, and operate these models in production is also increasing. This section will introduce some key considerations for efficiently and reliably delivering generative AI workloads.

The terminology in this space is not yet well established; some people use the terms GenAIOps, FMOps, or LLMOps. All of these build on existing MLOps practices, and because the considerations are fundamentally similar between them, this chapter will focus on foundation model operations (FMOps) as a general term for operationalizing workloads that rely on generative foundation models, regardless of model type, such as LLM or multimodal.

This includes workloads that utilize foundation models as is, as well as those that require models to be fine-tuned and/or augmented. What is not included is providers of foundation models that perform pretraining, as the considerations for pretraining more closely follow traditional MLOps.

Chapter 1 introduced a typical generative AI project life cycle consisting of a number of iterative steps. Each of those steps has unique considerations for being able to create reliable, operationally efficient, and repeatable workflows within the life cycle stages, as shown in Figure 9-17.

The next section will cover a few high-level considerations across select stages and steps of the project life cycle to reliably and efficiently scale generative AI workloads, starting with the *experiment and select* step.

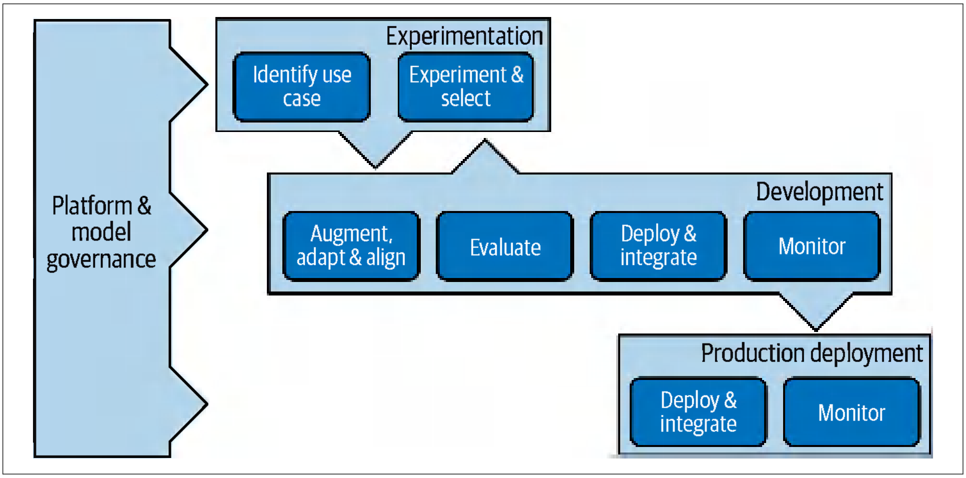


Figure 9-17. Create reliability and repeatability across stages of the generative ai project life cycle

Experimentation Considerations

After a viable use case has been identified, the first step is typically to experiment with existing foundation models and identify the top candidate or candidates to move forward with. It's also important to note that this step can also happen on a continuous basis as new state-of-the-art models get released, to determine if the performance for your use case can potentially be improved with a different model.

As a result, building automated frameworks that evaluate model performance based on domain-specific datasets is a common way to increase repeatability in the model selection process. Building a repeatable mechanism to evaluate models during model selection typically includes a few components including an experimentation environment, a prompt catalog and an evaluation datastore, as shown in Figure 9-18.

Experimentation on a new use case can be performed across a number of environments, as discussed in Chapter 1, such as managed playgrounds, notebook environments, or even from local machines. One way to enable increased reliability and repeatability in your experiments is to implement a prompt catalog.

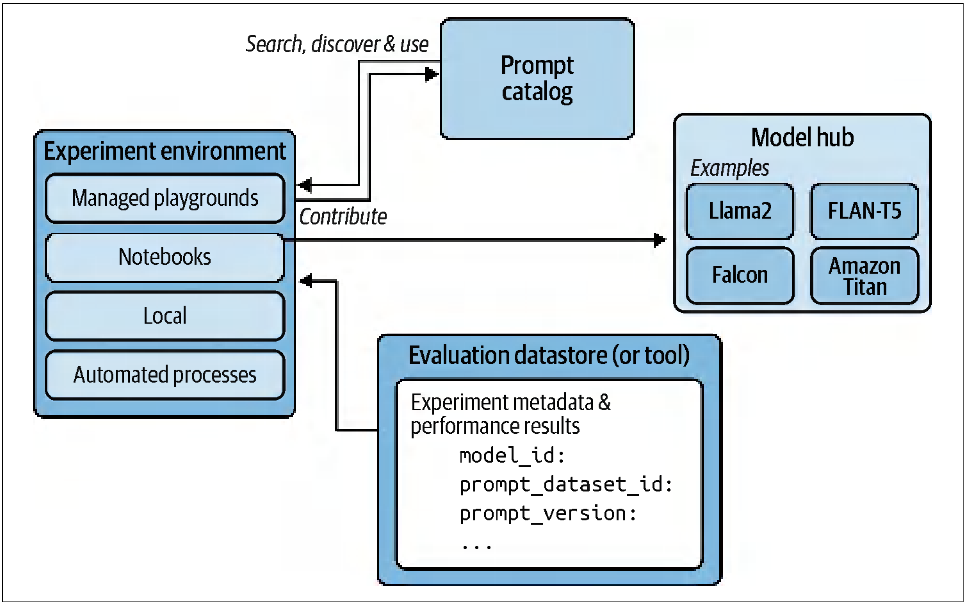


Figure 9-18. Building a repeatable mechanism to evaluate models during model selection

The concept of a prompt catalog was first introduced in a research paper[3] but has since gained more traction in a number of implementations and tooling. At a high level, a prompt catalog serves two purposes. First, the prompt catalog documents successful patterns for structuring prompts across multiple tasks that can be used to adapt to specific domains. This usually takes the form of existing or adapted prompt templates, as discussed in Chapter 5. Second, it contains a catalog of patterns that have been successfully used that is made readily available for evaluating new models or fine-tuning later in the life cycle.

An evaluation datastore, or experiment management capability, is also needed to reliably track key metadata such as the foundation model and prompt data used, as well as key performance metrics. Implementing these two components during experimentation can increase productivity through reusable prompt template patterns and shared prompts, as well as increase the ability to reliably track performance results. This same pattern can also be combined with automated processes to provide a repeatable framework to evaluate new foundation models against your use cases.

3 Jules White et al., "A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT", *arXiv*, 2023.

Additionally, if these environments are utilizing sensitive data, they should adhere to existing security and governance best practices, which are not unique to FMOps and include automating the provisioning and configuration of these environments and supporting components through infrastructure as code (IaC) and policy as code (PaC), combined with continuous monitoring. When working with sensitive data, the key pillars of security and governance best practices need to be considered, such as network isolation, governed access, enforcement of minimum privileges, detective controls, and data protection.

Development Considerations

During this step in the generative AI project life cycle, the focus is on creating or augmenting a model that is performant to the target task. There are many existing MLOps practices, such as automated training pipelines, that apply directly to fine-tuning generative AI models. However, typically the final step in a training pipeline is to register the candidate model in a model registry with key metadata that tracks model lineage, then deploy the model to production. With traditional ML models, the training inputs and outputs are generally well known, which allows for reliable tracking of model lineage.

Model lineage defines information about how a model was built. In traditional machine learning, this translates into having an auditable record of the model inputs, evaluation metrics, and generated artifacts specific to that model version, as illustrated in Figure 9-19. Model lineage is important in MLOps for reproducibility as well as auditability.

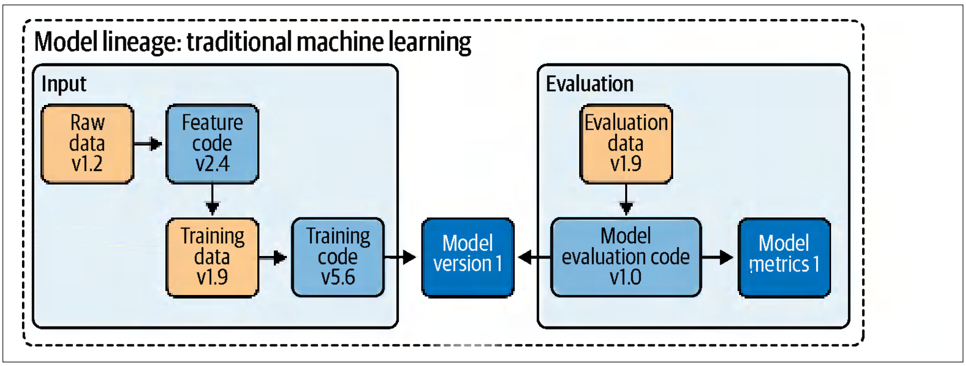


Figure 9-19. Model lineage for reproducibility with traditional machine learning models

For generative models, there are a couple of key differences to be aware of as it relates to FMOps. First, complete model lineage may not be possible. Specifically, some model providers do not provide details (pretraining and fine-tuning datasets, etc.) about how their models were trained. Also, when details are provided, they may not be specific enough to include the complete versioned data sources. As a result, model lineage may not be as deep as with traditional machine learning models, as shown in Figure 9-20.

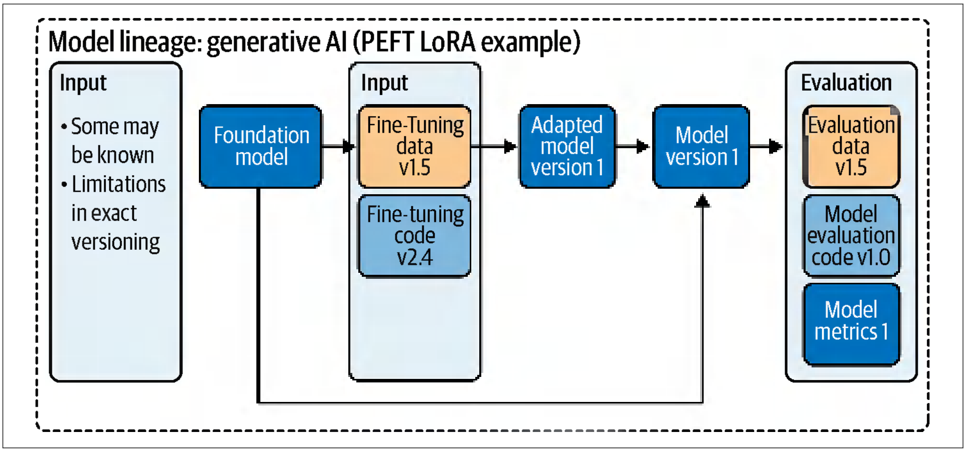


Figure 9-20. Model lineage for generative models using PEFT LoRA as an example

The level of transparency provided with foundation models is a consideration when comparing against your organizational or regulatory requirements and selecting the right foundation model. Regardless of the level of transparency in the foundation model chosen, when thinking about lineage you should still consider maintaining your own lineage for the components in scope for traceability as well as the ability to reliably redeploy or debug if needed.

Similar to MLOps, reliably capturing this model lineage metadata requires an automated approach to experiment management and tracking, because this metadata is used in the packaging of models for deployment to production environments as well as the ongoing management of deployed models.

Production Deployment Considerations

Building a prototype is typical early in the generative AI project life cycle. However, as you move beyond that prototype and look to deploy the generative AI application to production, there are a few key considerations.

For many of the components of an application, such as the frontend interface, traditional software best practices and AWS Well-Architected best practices apply directly. The same is true for optimizations within the deployment process, where traditional DevOps or MLOps best practices still apply and are not really unique to FMOps.

For example, to deploy a foundation model, one best practice is to utilize repeatable tooling that allows you to provision and configure SageMaker endpoints through infrastructure/configuration as Code (IaC/CaC). This allows for repeatability, rollback capabilities, and advanced deployment patterns such as A/B testing, as discussed in Chapter 8.

Because a lot of the existing practices often directly apply to building generative AI applications, this section will only focus on a few high-level considerations directly related to FMOps.

First, packaging models for deployment and managing deployed model versions may require additional dependencies that should be captured in a model registry. For example, with a fine-tuned model adapted using LoRA, multiple dependent models are required to deploy, including the foundation model, the adapted model, and—depending on the deployment implementation—a merged model. Related metadata for each of these models should be captured in the model registry to enable you to trace lineage or repackage for deployment if needed, as shown in Figure 9-21.

The end result should be that for every given deployed model, you are able to reliably capture key metadata about the model version to manage models at scale, as well as reliably redeploy if needed.

This section covered only a few high-level considerations for looking at operational efficiencies across the generative AI project life cycle. Again, many of the existing practices with MLOps directly apply, even if there are nuances in the implementation. For example, continuous monitoring is still applicable but the implementation will be different for LLMs because the evaluation criteria and metrics are different. Even though the industry is still debating terminology in this space, the core practices of MLOps still largely apply, and we build on that through only the aspects unique to generative models with FMOps.

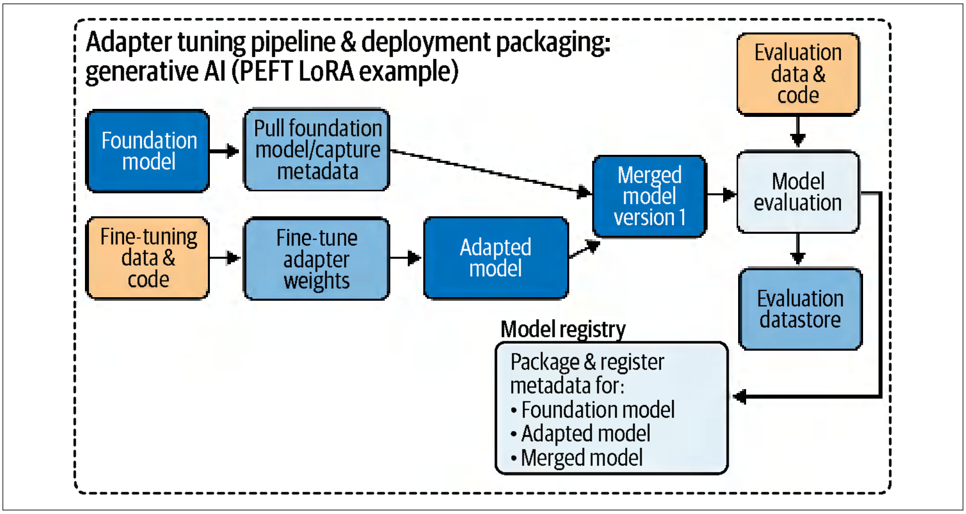


Figure 9-21. Model packaging and management for adapter models

Summary

This chapter covered RAG as a common framework for augmenting LLMs and using RAG to mitigate the common knowledge limitations of hallucinations and knowledge cutoffs in LLMs by providing access to external sources of information. You explored a specific use case for document retrieval and the importance of vector stores in implementing RAG architectures. This chapter also outlined the workflows and steps within those workflows that support RAG and agent-based architectures. You learned that frameworks like LangChain can help reduce the time to implement these complex workflows and make it possible to quickly build, deploy, and test LLM-powered applications that use powerful retrieval and augmentation techniques like RAG and agents.

You learned that foundation models can serve as remarkable reasoning engines in applications, leveraging their "intelligence" to fuel exciting and practical use cases, and that agents facilitate this process by taking charge of prompt engineering and communication between systems. They enable FM-powered applications to perform actions in the real world, making the applications more versatile and interactive.

This chapter also highlighted high-level components to consider as part of building an end-to-end generative AI application. You saw some examples of broader AWS services that can be used in building those applications.

Finally, this chapter briefly highlighted a few considerations for building repeatability, reliability, and operational efficiencies across the generative AI project life cycle.

In Chapter 10, you will explore multimodal foundation models that extend generative AI beyond text. You will explore multimodal use cases, such as generating images from descriptions and visual question answering, and learn more about the architectures that power multimodal foundation models.

CHAPTER 10

Multimodal Foundation Models

Generative AI can be unimodal or multimodal. Unimodal models work exclusively with data in one modality, such as text. Large language models (LLMs) are a popular example of unimodal generative AI; both the input and output modality in prompt and completion is text. Once you add another modality to the mix, such as image, video, or audio, you are tapping into multimodal generative AI.

With multimodal generative AI, you can broaden the scope of use cases and tasks and potentially move closer to artificial general intelligence (AGI) by enhancing the model's contextual understanding and cross-modal learning. Multimodal generative AI is a step toward simulating real-world complexity that not only enables models to process diverse data formats but also to learn through transfer and become better at creative problem solving.

With multimodal AI, you add different content modality to the input to support tasks such as converting, for example, image to text or text to image. Figure 10-1 illustrates the difference between unimodal and multimodal generative AI.

This chapter starts with an introduction to multimodal generative AI use cases and tasks, including image generation and visual question answering (VQA) using the Stable Diffusion and IDEFICS models, respectively. The power of these multimodal models is the ability to interact with them using natural language prompts.

Let's start by exploring common multimodal generative AI use cases and tasks.

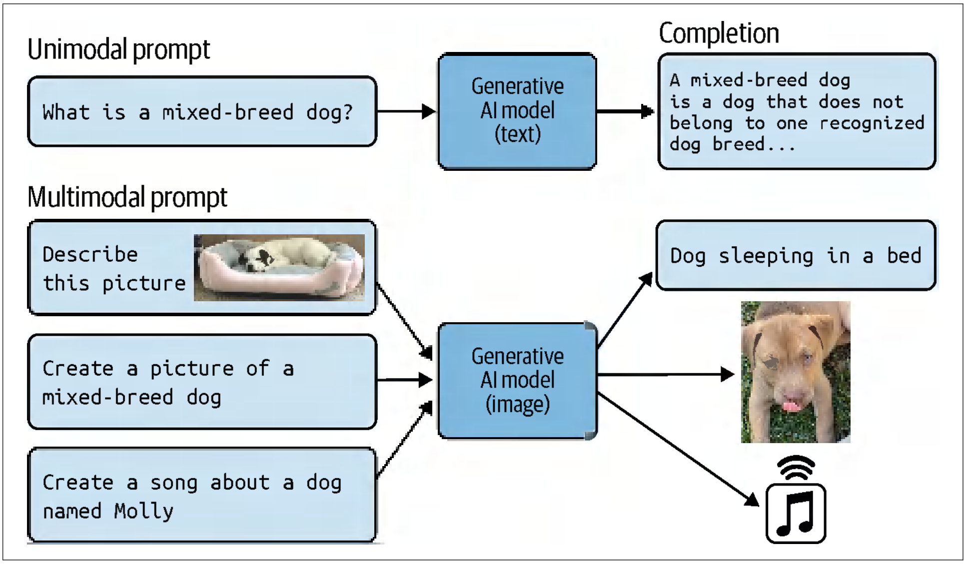


Figure 10-1. Unimodal versus multimodal generative AI

Use Cases

Multimodal generative AI can create rich and diverse content by combining text, images, videos, audio, and more. Multimodal generative AI is used for generating compelling marketing materials, presentations, and other types of creative content that incorporate multiple modalities.

In addition to content generation, other popular use cases include image captioning to increase accessibility for visually impaired users, visual question answering where users can ask questions about what they see in an image, content moderation to identify harmful content across modalities, and the creation of virtual environments in video games, simulations, and virtual reality.

You'll also see multimodal generative AI used in fashion and product design to help generate new clothing designs or interior layouts, and in customer service, powering virtual assistants, chatbots, and avatars that engage with users through text, speech, and visual cues.

Since most image-generation use cases involve a prompt, let's first explore some prompt engineering best practices and generative-inference configuration parameters related to image generation.

Multimodal Prompt Engineering Best Practices

It is important to become familiar with the nuances of the foundation model you are working with to author the most useful prompts. This section demonstrates various ways to influence text-to-image multimodal models when generating images. Let's start with some high-level prompt engineering tips that work for a wide variety of image-generation models, including Stable Diffusion:

Define the type of image.
: You can specify phrases like "film," "oil painting," "sketch," or "3D rendering" to express the desired style of your generated image. Within each style, you can instruct the model to generate an image with different framing and lightning. For example, "Generate a close-up sketch with natural lighting."

Describe the subject.
: What are you trying to generate? You will need to find a balance between not enough detail and too much detail. To generate multiple subjects, you should use the plural version of the subject, such as "dogs" instead of just "dog."

Specify style and artists.
: You can ask the model to generate an image similar to a specific artist, such as Vermeer or Rembrandt. Additionally, you can ask the model to generate images that combine multiple artists, for example: "Generate an image by Van Gogh and Picasso."

Be specific about quality.
: Generative models perform better when the prompt contains very specific details about what you are trying to generate. Use words like "realistic," "high resolution," and "8k" to improve the quality of the rendered image. You will likely iterate many times on finding the right amount of detail.

Be expressive.
: Despite the many brief examples available online, it's OK to express yourself when writing these prompts. Avoid the urge to paraphrase or shorten the prompt to just a single phrase or utterance. Separate out your thoughts, incrementally add new details, and notice how the model responds. Iterate until you get your desired result.

Choose order of words.
While it's good to be specific and expressive, it's worth noting that words at the beginning of the prompt are often weighted more heavily than words at the end.

Avoid negative phrases.
Consistent with unimodal large language models, negative phrases are sometimes difficult to interpret by the model. Use positive phrases if possible.

Embrace negative prompts.
Separate from the prompt, there is a parameter specifically used to specify which objects, styles, and characteristics that the model should not generate. For example, if you don't want your model to generate a blurry background, you can specify "blurry background" in the negative prompt parameter. You should phrase these in a positive manner to avoid the double negative scenario of specifying a negative phrase in the negative-prompt parameter.

Image Generation and Enhancement

Many of the described multimodal generative AI use cases that incorporate image and text data involve image generation, image editing and enhancement, and image-to-text tasks. Let's explore those tasks in more detail using Stable Diffusion.

Image Generation

Image generation is a common task for multimodal models that support *text-to-image* capabilities. Figure 10-2 shows an example of a text prompt, `Create a picture of a dog laying on grass`, instructing the model to generate an image as output that matches the text description in the prompt.

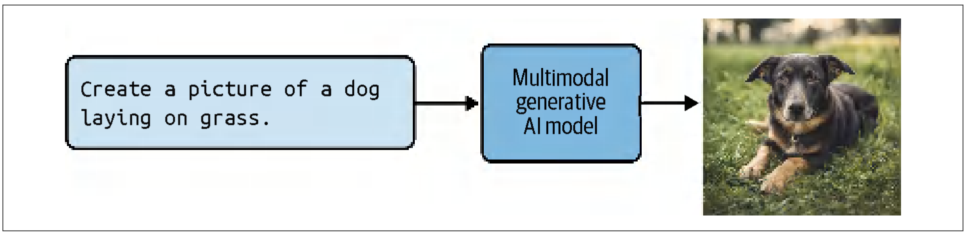


Figure 10-2. Image generation using text-to-image

Here is the code to generate this image with Stability AI's Stable Diffusion XL model and Amazon SageMaker JumpStart:

```python
import sagemaker
from stability_sdk_sagemaker.predictor import StabilityPredictor
from stability_sdk.api import GenerationRequest, GenerationResponse, TextPrompt
from sagemaker.utils import name_from_base
from PIL import Image
import io
import base64

endpoint_name=name_from_base("sdxl-1-0-jumpstart")
sagemaker_session = sagemaker.Session()

deployed_model = StabilityPredictor(endpoint_name=endpoint_name,
    sagemaker_session=sagemaker_session)

prompt = "Create a picture of a dog laying on grass."

output = deployed_model.predict(
    GenerationRequest(
        text_prompts=[TextPrompt(text=prompt)],
        style_preset="anime",
        width=1024,
        height=1024,
        seed=5,
    )
)

def decode_and_show(model_response: GenerationResponse, image_name):
    image = model_response.artifacts[0].base64
    image_data = base64.b64decode(image.encode())
    image = Image.open(io.BytesIO(image_data))
    image.save(image_name)
    display(image)

decode_and_show(output, image_name)
```

Image generation powers a variety of content generation use cases, including the generation of creative content such as book illustrations or music album cover designs. There are also broader applications, such as using generated images to experiment with and influence product design.

Image Editing and Enhancement

Image editing and enhancement uses *image-to-image* capabilities of generative AI models to generate a new or modified image from an image and instruction that you provide as input along with a text-based prompt. Image editing and enhancement tasks support a range of use cases, including artistic style transfer, domain adaptation, and upscaling.

Style transfer converts images into another specific artistic style—for example, an anime-style image into a photorealistic image. The style is usually expressed in the input text prompt and/or defined with a model parameter, such as `style_preset` in Stable Diffusion. The values for style transfer include `photographic`, `digital-art`, and `cinematic`. The `style_preset` parameter is useful for art creation, design, or photo editing applications.

Figure 10-3 shows an example of style transfer applied to an image created with the prompt `Create an image of a dog dressed as a ninja eating ice cream in anime style` (left image in Figure 10-3). You can then use this image as an input image and ask the model to change the style to photorealistic (right image in Figure 10-3) using the `style_preset` parameter.

Figure 10-3. Example of style transfer from anime-style image to photorealistic-style image

Here is the code to generate the first image in Figure 10-3 with Stability AI's Stable Diffusion XL model and Amazon SageMaker JumpStart:

```
prompt="Create an image of a dog dressed as a ninja eating ice cream"

output = deployed_model.predict(
    GenerationRequest(
        text_prompts=[TextPrompt(text=prompt)],
        style_preset="anime",
        width=1024,
        height=1024
    )
)
```

Here is the code to generate the second image in Figure 10-3 with Stability AI's Stable Diffusion XL model and Amazon SageMaker JumpStart:

```
def encode_image(image_path: str,
  resize: bool = False,
  size: Tuple[int, int] = (1024, 1024)) -> Union[str, None]:

    image = Image.open(image_path)

    if resize:
        image = Image.open(image_path)
        image = image.resize(size)
        updated_image_path = "resize-{}".format(image_path)
        image.save(updated_image_path)
        image_path = updated_image_path

    with open(image_path, "rb") as image_file:
        img_byte_array = image_file.read()
        # Encode the byte array as a Base64 string
        base64_str = base64.b64encode(
          img_byte_array).decode("utf-8")
        return base64_str

size = (1024,1024)
image_data = encode_image("anime_ninja_dog.png", size=size)

new_prompt="Create a photograph of a dog dressed as a ninja eating ice cream"

output = deployed_model.predict(
    GenerationRequest(
        text_prompts=[
          TextPrompt(text=new_prompt)
        ],
        init_image=image_data,
        style_preset="photographic",
        ...
    )
)
```

Domain adaptation converts images from one domain to another, such as converting satellite images to maps or changing day scenes to night scenes.

Figure 10-4 shows an example of changing an image from a night scene created with the prompt `Create a photorealistic image of a Storm Trooper holding a surfboard at night during full moon` (left image in Figure 10-4), to a day scene, created with the prompt `Create a photorealistic image of a Storm Trooper holding a surfboard during day` (right image in Figure 10-4).

Figure 10-4. Example of an image changing from night scene to day scene

Here is the code to generate the first image in Figure 10-4 with Stability AI's Stable Diffusion XL model and Amazon SageMaker JumpStart:

```
prompt="Create a photorealistic image of a Storm Trooper holding a surfboard at
night during full moon"

output = deployed_model.predict(
    GenerationRequest(
        text_prompts=[
          TextPrompt(text=prompt)
        ],
        width=1024,
        height=1024
    )
)

decode_and_show(output)
```

Here is the code to generate the second image in Figure 10-4 with Stability AI's Stable Diffusion XL model and Amazon SageMaker JumpStart:

```
new_prompt="Create a photorealistic image of a Storm Trooper holding a surf
board on a bright sunny day"

output = deployed_model.predict(
    GenerationRequest(
        text_prompts=[
          TextPrompt(text=new_prompt)
        ],
        init_image=image_data,
        style_preset="photographic",
        ...
```

```
    )
)

decode_and_show(output)
```

Domain adaptation is useful for simulating various scenarios in video games, simulations, or product presentations.

Upscaling converts lower-resolution images into higher resolutions. Unlike non-deep-learning techniques such as nearest neighbor, generative AI takes the whole context of the image into account, using a text prompt to guide the upscaling process.

Figure 10-5 shows an example of upscaling a low-resolution image of a green iguana to a higher resolution. On the left is the low-resolution iguana image used as the input image to the model with the simple prompt `a green iguana`. On the right is the high-resolution image generated by the model.

Figure 10-5. Upscaling a lower-resolution image of a green iguana to a higher resolution

Here is the code to generate the upscaled image in Figure 10-5 with Stability AI's Stable Diffusion x4 upscaler FP16 model and Amazon SageMaker JumpStart:

```
low_res_img_file_name = "green_iguana_lowres.jpg"

endpoint_name =
  'jumpstart-dft-stable-diffusion-x4-upscaler-fp16'

def query_endpoint(payload):
    client = boto3.client('runtime.sagemaker')

    response = client.invoke_endpoint(
      EndpointName=endpoint_name,
      ContentType='application/json;jpeg',
```

```
        Accept='application/json;jpeg',
        Body=payload)

      return response

  def parse_response(query_response):
      response_dict = json.loads(query_response['Body'].read())
      return response_dict['generated_images'],
             response_dict['prompt']

  with open(low_res_img_file_name,'rb') as f:
    low_res_image_bytes = f.read()

  encoded_image = base64.b64encode(
    bytearray(low_res_image_bytes)).decode()

  payload = {
    "prompt":"a green iguana",
    "image": encoded_image
  }

  query_response = query_endpoint(
    json.dumps(payload).encode('utf-8'))

  generated_images, prompt = parse_response(query_response)

  for generated_image in generated_images:
      generated_image_decoded = BytesIO(
        base64.b64decode(generated_image.encode()))
      generated_image_rgb = Image.open(
        generated_image_decoded).convert("RGB")
```

Upscaling can be useful in medical imaging tasks to enhance images, segment regions of interest, or reconstruct missing data. It can also improve the quality of medical scans, aid in diagnosis, and even generate realistic images from incomplete data, supporting research and clinical applications.

Inpainting, Outpainting, Depth-to-Image

The image-editing and enhancement tasks described thus far usually change the image as a whole. There are also more advanced techniques that help you modify only parts of an image, including inpainting, outpainting, and depth-to-image.

Inpainting

Inpainting replaces a portion of an image with another image based on an instruction prompt and image mask. Generative models that support inpainting are usually derived from a base image model with an added mask generation strategy. The mask represents the segments in the original image that you want to change and the segments to leave unchanged. They accept an additional `mask_input` parameter, an image where the blacked-out portion remains unchanged during image generation and the white portion is replaced.

To perform inpainting, provide the original image, a mask image that outlines the portion to be replaced, and a text prompt with the instruction. The example shown in Figure 10-6 uses inpainting to remove the tree from the image shown on the left. In the middle, you can see the provided image mask. On the right, you can see the inpainted image without the tree.

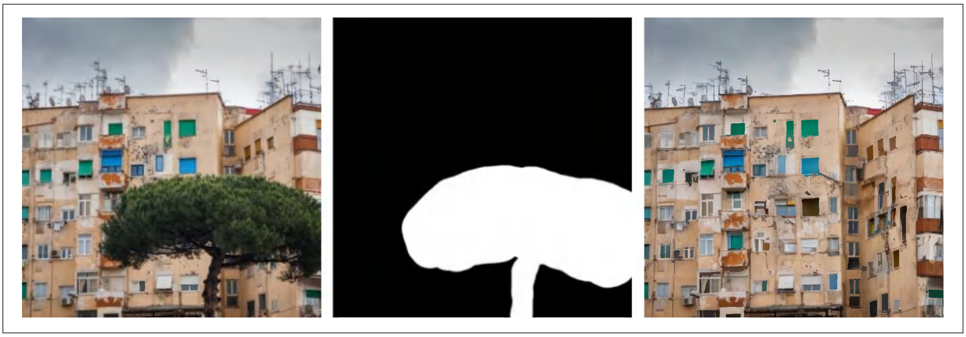


Figure 10-6. Inpainting replaces a portion of an image

Here is the code to generate the inpainted image in Figure 10-6 with Stability AI's Stable Diffusion 2 Inpainting model and Amazon SageMaker JumpStart:

```
endpoint_name = 'jumpstart-dft-stable-diffusion-2-inpainting'

input_img_file_name = "inpainting/original-image.png"
input_img_mask = "inpainting/mask-image.png"

def encode_img(img_name):
    with open(img_name,'rb') as f:
      img_bytes = f.read()

    encoded_img = base64.b64encode(
      bytearray(img_bytes)).decode()

    return encoded_img

encoded_input_image = encode_img(input_img_file_name)
encoded_mask_image = encode_img(input_img_mask)
payload = {
  "prompt": "building, facade, paint, windows",
  "image": encoded_input_image,
  "mask_image":encoded_mask_image
}
```

```
def query_endpoint(payload):
    client = boto3.client('runtime.sagemaker')

    response = client.invoke_endpoint(
      EndpointName=endpoint_name,
      ContentType='application/json;jpeg',
      Accept = 'application/json;jpeg',
      Body=encoded_payload)

    return response

def parse_and_display_response(query_response):
    response_dict = json.loads(query_response['Body'].read())
    generated_images = response_dict['generated_images']

    for generated_image in generated_images:
      with BytesIO(
        base64.b64decode(
          generated_image.encode())) as generated_image_decoded:
            with Image.open(generated_image_decoded) as
              generated_image_np:
                generated_image_rgb =
                  generated_image_np.convert("RGB")
                generated_image_rgb.save("generated-image.png")

query_response = query_endpoint(payload)

parse_and_display_response(query_response)
```

The most common use cases for inpainting are image restoration use cases, such as repairing incomplete or damaged areas of building blueprints in architectural designs or removing cropping artifacts in medical imaging.

Outpainting

Outpainting expands images beyond their original borders to create larger-sized images. In Figure 10-7, we used the image of the green iguana as input, scaled the image by 0.5, provided an image mask that marks the outside frame to be changed, and instructed the model to outpaint. The right image in Figure 10-7 shows the generated image after outpainting.

The most common use cases for outpainting are artistic content generation, photography enhancement and editing, and video game design.

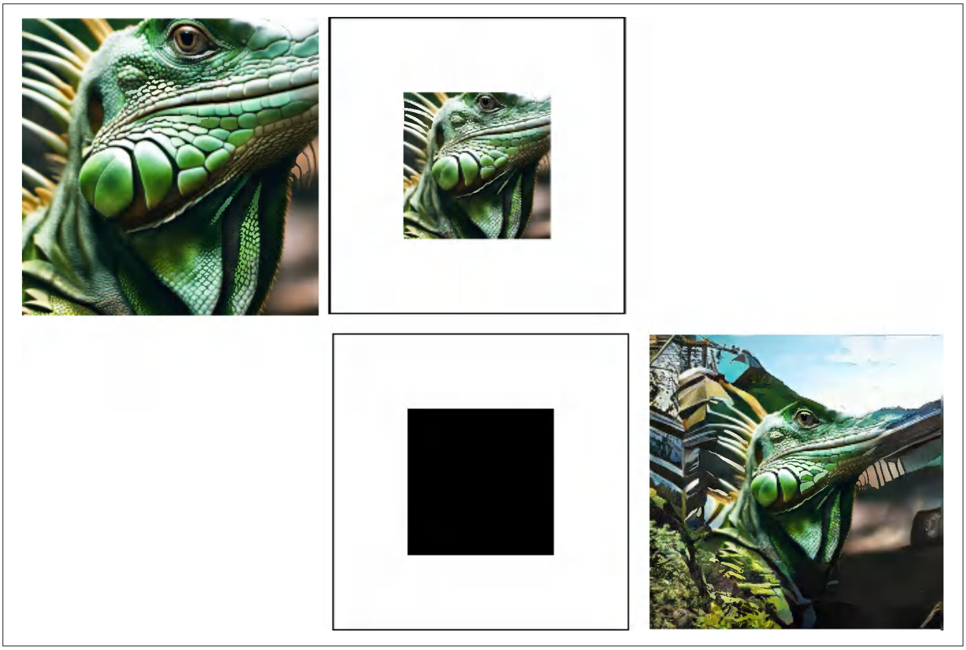


Figure 10-7. Outpainting expands images beyond their original borders

Depth-to-Image

Depth-to-image is a technique that generates new images from existing ones while preserving the shape and depth of the objects in the original image.

Depth-to-image is often used to explore different interior design styles while keeping the interior space and boundaries coherent with your input image, as shown in Figure 10-8.

Here, we generated an image with the prompt `Create an image of an ultra modern penthouse overlooking Lake Tahoe` (shown on the left). We then passed this image to a model with depth-to-image capabilities and prompted it with `city view, marble floor, minimalist lifestyle`. In the generated image (shown on the right), you can see how the overall image composition and depth of the objects is preserved, but the view changed from lake to city and the floors changed from hardwood to marble.

Figure 10-8. Changing interior designs using depth-to-image

Here is the code to generate the image in Figure 10-8 with Stable Diffusion 2's Depth FP16 model and Amazon SageMaker JumpStart:

```
input_img_file_name = "room.png"

endpoint_name = 'jumpstart-dft-sd-2-depth-fp16'

encoded_input_image = encode_img(input_img_file_name)

payload = {
  "prompt": "city view, marble floor, minimalist lifestyle",
  "image": encoded_input_image
}

query_response = query_endpoint(payload)

parse_and_display_response(query_response)
```

In marketing and branding, you can take photographs of your products and use depth-to-image to generate creative variations for digital advertisements or brochures, as shown in Figure 10-9.

Here, we generated an image with the prompt `Create an image of a fancy cocktail with beach in the background` (shown on the left). We then passed this image to a model with depth-to-image capabilities and prompted it with `nyc rooftop bar` (shown on the right).

Figure 10-9. Changing product marketing images using depth-to-image

Here is the code to generate the image in Figure 10-9 with Stable Diffusion 2 Depth FP16 model and Amazon SageMaker JumpStart:

```
input_img_file_name = "cocktail.png"

endpoint_name = 'jumpstart-dft-sd-2-depth-fp16'

encoded_input_image = encode_img(input_img_file_name)

payload = {
  "prompt": "nyc rooftop bar",
  "image": encoded_input_image
}

query_response = query_endpoint(payload)

parse_and_display_response(query_response)
```

In game development, you can use depth-to-image to generate different in-game landscapes from a base image that contains some elements you want to include.

Image Captioning and Visual Question Answering

When you align an LLM with a vision-based model, you get a multimodal large language model (MLLM), sometimes called visual language models (VLMs). These multimodal models accept inputs of different content modality.

The models know how to follow instructions and perform in-context learning for both text-based and multimodal tasks. These models are often used for image-to-text tasks that accept images as input and generate text as output.

Some popular image-to-text models include Flamingo from DeepMind[1] and Image-Aware Decoder Enhanced à la Flamingo with Interleaved Cross-attentionS (IDEFICS) from Hugging Face. These models are trained on datasets that more naturally interleave images and text—versus a dataset of image-text caption pairs. By interleaving images and text, the models tend to perform better on multimodal reasoning benchmarks.

While Flamingo is a proprietary model trained on a closed dataset, IDEFICS is based on the Flamingo architecture, is freely available, and is trained on a public dataset called OBELICS. OBELICS is an image-text dataset that consists of 140 million web pages extracted from the Common Crawl dataset and interleaved with 350 million images associated with these web pages. In addition, another 100 billion highly curated text tokens are added to the dataset to improve the model's language understanding.

> OBELICS and IDEFICS are acronyms that mimic names from a popular French comic book, *Asterix*, which stars a fictional character, Obelix, and his dog, Idefix.

IDEFICS is available in 9 billion- and 80 billion-parameter models with very powerful spatial and language understanding that aligns natural language with image perception. There are also instruction fine-tuned variants of both the 9 billion- and 80 billion parameter-models that are optimized for conversational applications.

During training, IDEFICS uses the pretrained LLaMA large language model in combination with a set of vision encoders and cross-attention layers that are trained on the interleaved text and image data from the OBELICS dataset. The cross-attention uses keys (k) and values (v) from the vision features (color, shape, etc.) along with queries (q) from the language features (tokens, input IDs, etc.).

Image-to-text powers many multimodal generative AI use cases, such as image captioning, content moderation, and VQA. Let's take a look at some of these use cases.

1 Jean-Baptiste Alayrac et al., "Flamingo: A Visual Language Model for Few-Shot Learning", *arXiv*, 2022.

Image Captioning

Image captioning automatically generates descriptive captions for images, combining computer vision and natural language processing. Image captioning is often used to enhance accessibility for visually impaired people, to assist in content indexing and retrieval, and to handle search-engine optimization (SEO) and social media sharing. It also has applications in education, automated content generation, assistive technology, and AI research, making visual content more meaningful and usable.

Content Moderation

Content moderation leverages the image-to-text capabilities of models to analyze content across visual and text elements. Those models can help detect deepfake content by analyzing visual and textual elements and uncover inconsistencies. They can enhance contextual content analysis by considering both text and images, leading to more nuanced moderation decisions. They can also help identify harmful content by leveraging both modalities and flagging offensive or inappropriate material more accurately.

Visual Question Answering

VQA tasks make use of image-to-text capabilities of a model to answer questions about images or visual content. These tasks require the model to understand both the visual information in the image and the textual content of the question to provide accurate and relevant answers. Figure 10-10 shows how VQA is able to correctly respond to the question, `Who makes this car?` from the image given in the multimodal prompt.

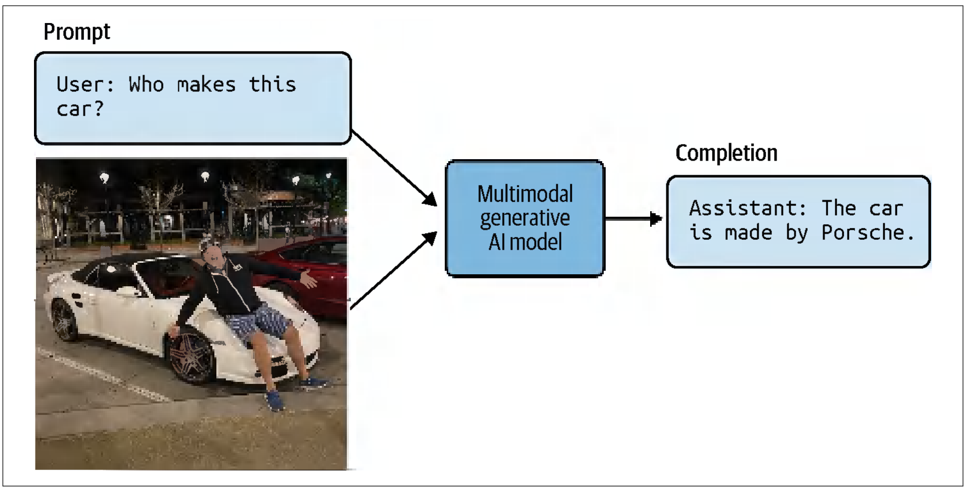


Figure 10-10. The model responds with the correct answer for a visual question

Similar to an LLM-based prompt, the text portion of the VQA prompt typically follows the format of `User: {question}\nAssistant:`. The code that follows implements the example in Figure 10-10 using the Hugging Face IDEFICS model for the VQA image-to-text task. Here, we are using the 9 billion-parameter IDEFICS instruct variant to ask questions of the image:

```
import torch
from transformers import IdeficsForVisionText2Text
from transformers import AutoProcessor

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

model_checkpoint = "HuggingFaceM4/idefics-9b-instruct"
model = IdeficsForVisionText2Text.from_pretrained(
  model_checkpoint)
processor = AutoProcessor.from_pretrained(model_checkpoint)
prompts = [
  "User: ",                           # input indicator
  "https://.../happy-car-chris.png" # image
  "Who makes this car?",              # question
  "Assistant: ",                      # output indicator
]

inputs = processor(prompts, return_tensors="pt").to(device)

generated_ids = model.generate(**inputs, max_length=100)

generated_text = processor.batch_decode(
  generated_ids, skip_special_tokens=True)[0]

print(generated_text)
```

Output:

```
Assistant: The car is made by Porsche.
```

VQA can also be combined with multimodal chain-of-thought prompting to simulate humanlike thought processes for more complex questions. In order to perform visual question answering, the model must iteratively reason over both the image and the question.

Figures 10-11 and 10-12 demonstrate the difference between multimodal standard prompting and chain-of-thought prompting, respectively. Adding `Think step-by-step.` to the prompt, as shown in Figure 10-12, directs the model to use chain-of-thought reasoning, and the model returns the correct answer.

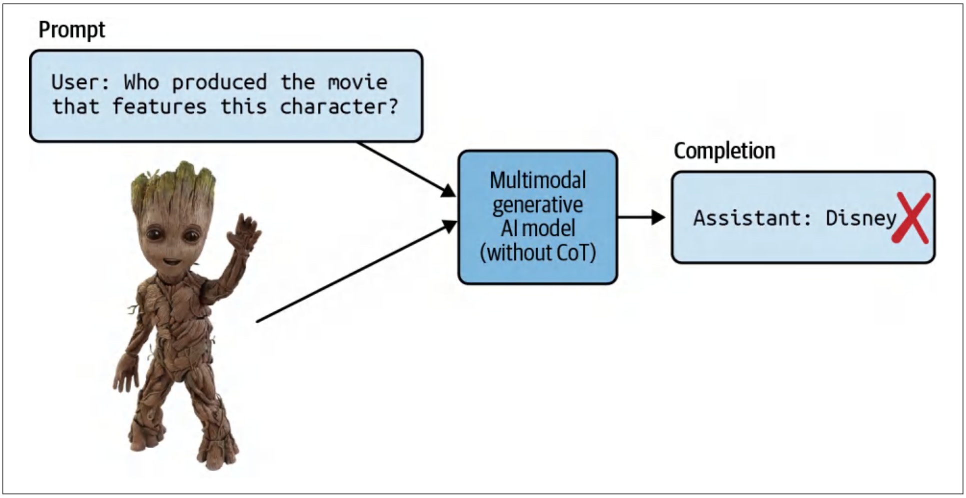


Figure 10-11. The model responds with an incorrect answer without multimodal chain-of-thought prompting

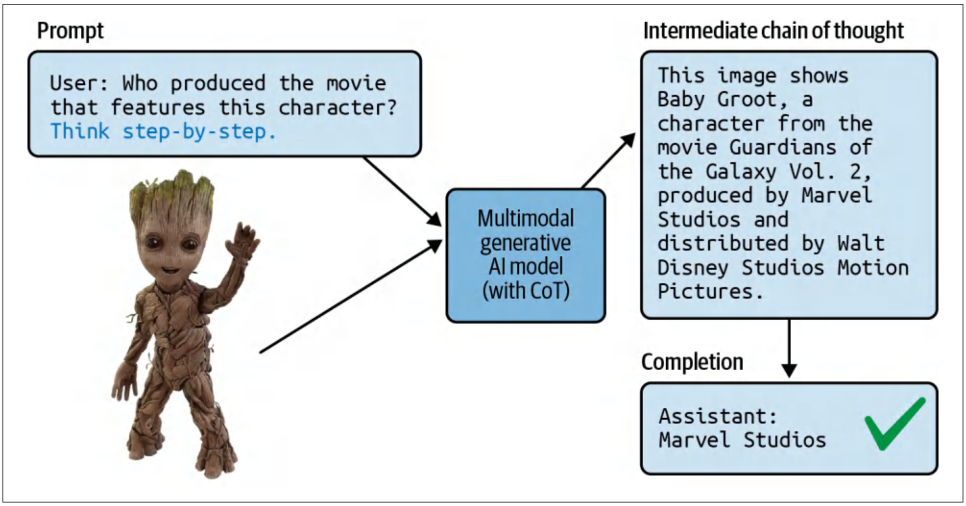


Figure 10-12. VQA with multimodal chain-of-thought prompting returns the correct answer

Here is the code to implement the chain-of-thought version of this prompt:

```
import torch
from transformers import IdeficsForVisionText2Text
from transformers import AutoProcessor

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

model_checkpoint = "HuggingFaceM4/idefics-9b-instruct"
model = IdeficsForVisionText2Text.from_pretrained(
  model_checkpoint)

processor = AutoProcessor.from_pretrained(model_checkpoint)

prompts = [
  "User: ",                                              # input indicator
  "https://.../baby-groot.jpg",                          # image
  "Who produced the movie that features this character?", # question
  "Assistant: "                                          # output indicator
]

inputs = processor(prompts, return_tensors="pt").to(device)

generated_ids = model.generate(**inputs, max_length=100)
generated_text = processor.batch_decode(generated_ids,
  skip_special_tokens=True)[0]

print(generated_text)
```

Output:

```
Assistant: Marvel Studios produced the movie that features this character.
```

VQA powers diverse applications, such as aiding visually impaired people, enhancing education, and assisting in content retrieval. It's also useful in medical diagnosis, tourism, gaming, and security, enabling users to interact with images through natural language questions.

Here is another example, this time using chain-of-thought reasoning to answer the question in the prompt, `How do I make this? Think step-by-step.`, referring to a pizza in Figure 10-13. Note the `Think step-by-step.` in Figure 10-13 and the code that follows.

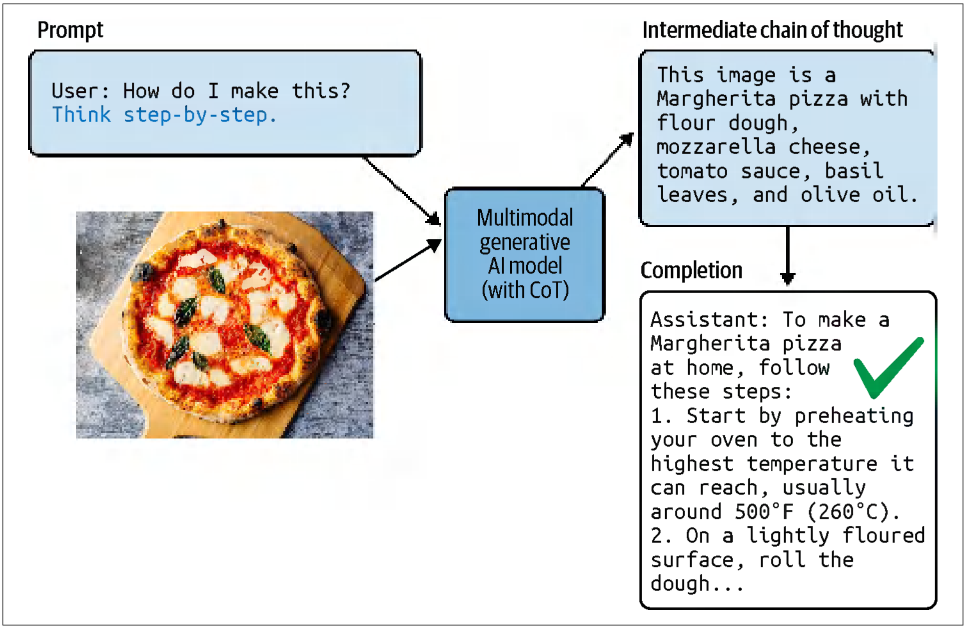


Figure 10-13. Visual question answering with chain-of-thought reasoning

```
model_checkpoint = "HuggingFaceM4/idefics-9b-instruct"
model = IdeficsForVisionText2Text.from_pretrained(
  model_checkpoint)

prompts = [
    "User: ",                                     # input indicator
    "https://.../margherita-pizza.jpg",           # image
    "How do I make this? Think step-by-step.",    # question
    "Assistant: "                                 # output indicator
]

inputs = processor(prompts, return_tensors="pt").to(device)

generated_ids = model.generate(**inputs, max_length=500)

generated_text = processor.batch_decode(generated_ids,
  skip_special_tokens=True)[0]

print(generated_text)
```

Output:

```
Assistant: To make a Margherita pizza at home, follow these steps:
1. Start by preheating your oven to the highest temperature it can reach,
usually around 500°F (260°C).
2. On a lightly floured surface, roll out your pizza dough into a circle.
3. Transfer the dough to a pizza stone or baking sheet.
4. Sprinkle a layer of tomato sauce on top of the dough, leaving a small border
around the edges.
5. Add a layer of fresh mozzarella cheese on top of the tomato sauce.
6. Add a layer of fresh basil leaves on top of the cheese.
7. Drizzle a small amount of olive oil over the pizza.
8. Place the pizza in the preheated oven and bake for 10-12 minutes, or until
the crust is golden and the cheese is melted and bubbly.
9. Remove the pizza from the oven and let it cool for a few minutes before
slicing and serving.
Optional: You can also add other toppings like sliced mushrooms, olives, or
pepperoni to your Margherita pizza."
```

Now that you've seen a few examples, let's explore ways to evaluate multimodal models for both text-to-image and image-to-text tasks. In the next section, you will learn about multimodal evaluation benchmarks, metrics, and datasets, including a nonverbal reasoning intelligence quotient (IQ) test used to evaluate human intelligence.

Model Evaluation

Evaluation benchmarks help quantify how well the multimodal model aligns visual perception and natural language. They also measure the model's ability to perform nonverbal reasoning. Evaluating multimodal models often requires a combination of qualitative human evaluation and quantitative statistical comparison.

As with any evaluation process, you will want to choose a dataset and benchmark to establish a baseline. This section shows you some common datasets, metrics, and benchmarks that you can use to evaluate your multimodal generative AI models across various tasks, including image generation, image modification, image classification, VQA, and nonverbal reasoning. Most evaluations are done with zero-shot inference, although few-shot is also an option in some cases.

Text-to-Image Generative Tasks

A great starting point for text-to-image generative tasks is the PartiPrompts dataset from the Parti project. This dataset consists of 1,600 English prompts across a number of categories, including world knowledge, animals, and indoor scenes, as shown in Figure 10-14.

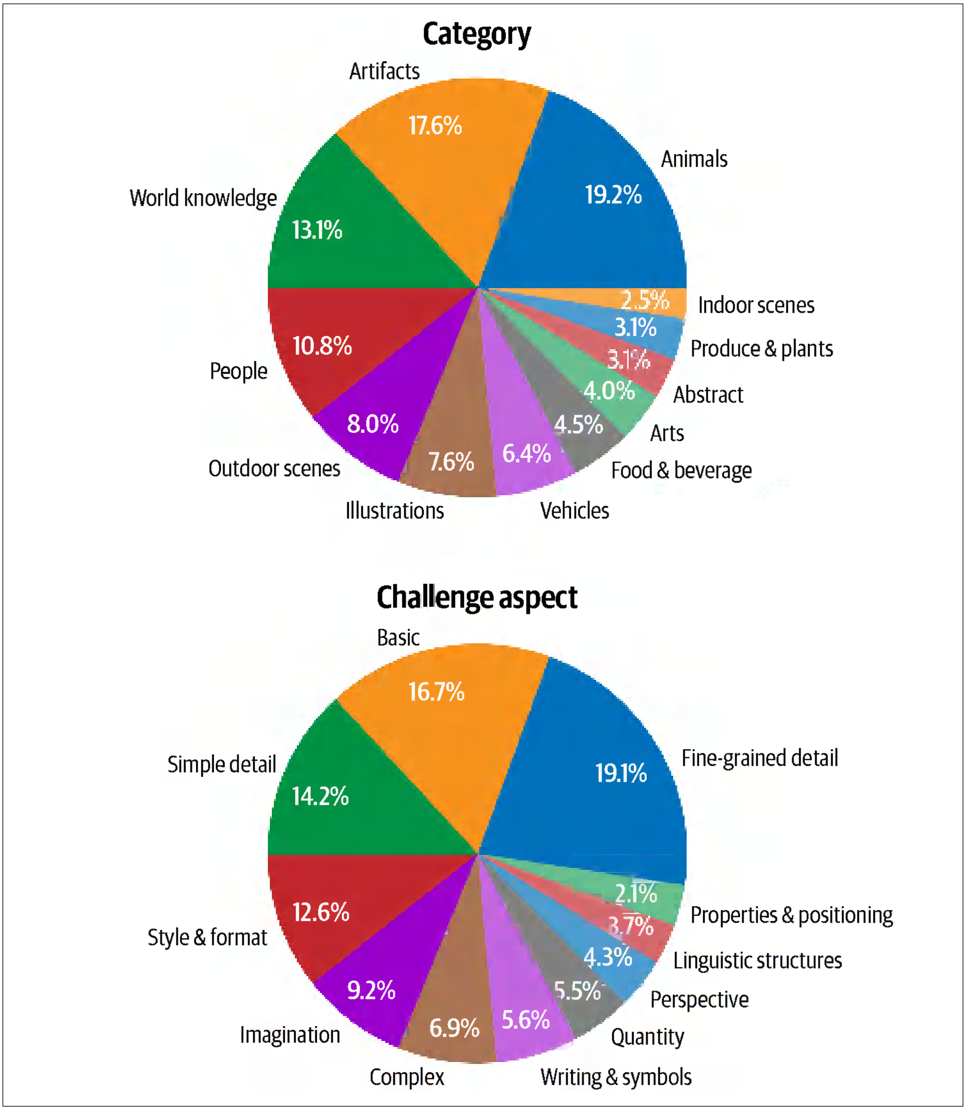


Figure 10-14. PartiPrompts dataset

The PartiPrompts dataset evaluates a number of text-to-image challenges, such as imagination, complexity, and fine-grained detail. You can select a subset of the Parti-Prompts dataset and manually evaluate the model's generated image for each of these prompts—or you can apply a more quantitative approach, which you will see next.

There are a few common ways to evaluate your model quantitatively, including CLIP score similarity, CLIP directional similarity, and Fréchet Inception Distance (FID). CLIP score similarity measures the semantic similarity, or compatibility, between each image and its caption. High CLIP score similarity implies a higher compatibility and is therefore desirable. CLIP directional similarity compares how similarly each image changes when making the same change to each caption. The higher the CLIP directional similarity score, the better, as the images appear to be more similar because they respond similarly to the same change in the prompt. FID measures the similarity between two image datasets.

Next, you will see an example of one of these evaluation metrics: CLIP score similarity. You will use this metric to compare Stable Diffusion 1.4 and 1.5:

```
from diffusers import StableDiffusionPipeline
import torch

model_checkpoint_1_4 = "runwayml/stable-diffusion-v1-4"
model_checkpoint_1_5 = "runwayml/stable-diffusion-v1-5"

sd_pipeline_1_4 = StableDiffusionPipeline.from_pretrained(
    model_checkpoint_1_4)

sd_pipeline_1_5 = StableDiffusionPipeline.from_pretrained(
    model_checkpoint_1_5)

prompts = [
    "a photo of an astronaut riding a horse on mars",
    "A high tech solarpunk utopia in the Amazon rainforest",
    "A pikachu fine dining with a view to the Eiffel Tower",
    "A mecha robot in a favela in expressionist style",
    "an insect robot preparing a delicious meal",
    "A small cabin on top of a snowy mountain in style of Disney, artstation",
]

images_1_4 = sd_pipeline_1_4(prompts,
  num_images_per_prompt=1, output_type="numpy").images

images_1_5 = sd_pipeline_1_5(prompts,
  num_images_per_prompt=1, output_type="numpy").images

from torchmetrics.functional.multimodal import clip_score
from functools import partial

clip_score_fn = partial(clip_score,
  model_name_or_path="openai/clip-vit-base-patch16")

def calculate_clip_score(images, prompts):
    images_int = (images * 255).astype("uint8")
    clip_score = clip_score_fn(
        torch.from_numpy(images_int).permute(0, 3, 1, 2),
          prompts).detach()
    return round(float(clip_score), 4)
```

```
sd_clip_score_1_4 = calculate_clip_score(images_1_4, prompts)
print(f"CLIP Score with v-1-4: {sd_clip_score_1_4}")
# CLIP Score with v-1-4: 34.9102

sd_clip_score_1_5 = calculate_clip_score(images_1_5, prompts)
print(f"CLIP Score with v-1-5: {sd_clip_score_1_5}")
# CLIP Score with v-1-5: 36.2137
```

Here, you see that Stable Diffusion 1.5 has an improved CLIP score similarity over its predecessor, Stable Diffusion 1.4. This implies that Stable Diffusion 1.5 maintains a higher semantic similarity, or compatibility, between the given prompts and generated images.

Forward Diffusion

Next, you will see how to evaluate common image-to-text generative AI tasks, including image captioning and VQA. An image caption is a text-based description of an image. Remember that VQA tasks ask questions about an image using natural language text.

You can use a number of multimodal datasets, including ImageNet and Rendered SST2 for image classification tasks and VQAv2 and VizWiz-VQA for visual question answering tasks. Primarily, zero-shot inference is used, but few-shot is also an option for some evaluation tasks.

For image-classification task evaluation, you can pass the image and a prompt like "This is an image of the following category:" to your multimodal generative AI model to generate a text-based completion with the predicted category. You can use a dataset like ImageNet, which includes approximately 1 million training images across 1,000 categories. You would evaluate the model's accuracy of predicting the correct category from the ImageNet ground truth category.

For example, you can ask the model to predict if text in the image contains any hate speech. In this case, you are evaluating the model's ability to understand the meaning of text embedded in the image—and its ability to detect hate speech.

Nonverbal Reasoning

To determine how well a multimodal model performs nonverbal reasoning, you can use Raven's Progressive Matrices (RPM). RPM is often used to measure general human intelligence and often used to determine a human's IQ.

Raven's IQ test is similar to in-context, few-shot learning with large language models where full examples are provided—and the model is asked to complete the missing example. The main difference is that the prompt includes shapes and symbols instead of language. As such, the model learns to recognize abstract concepts and patterns in the given image, as shown in Figure 10-15, adapted from the paper "Language Is Not All You Need: Aligning Perception with Language Models."[2]

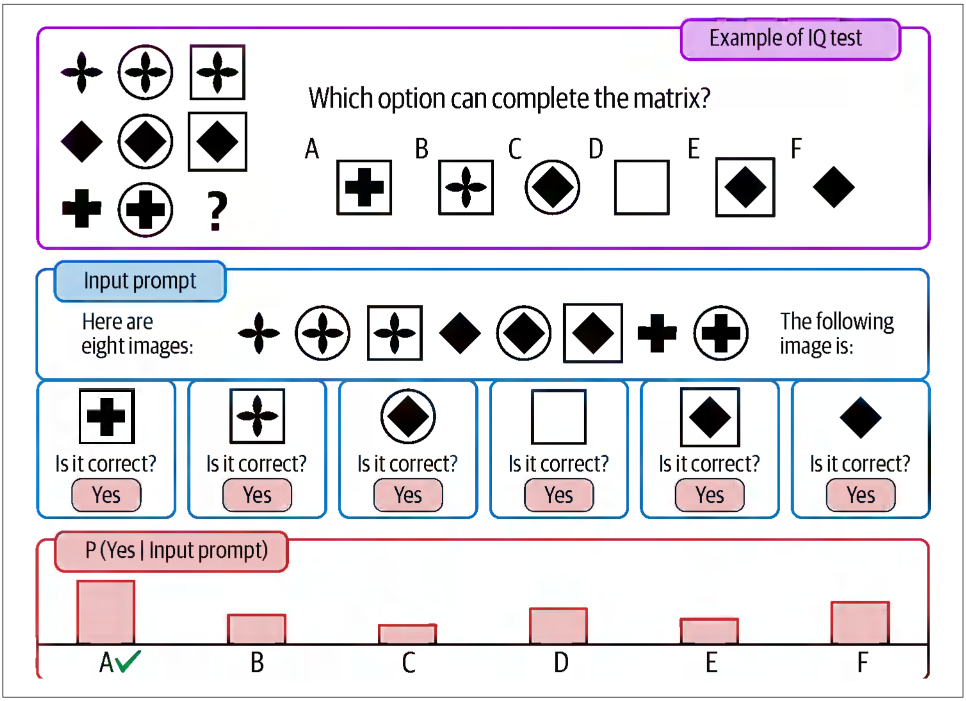


Figure 10-15. Evaluate nonverbal reasoning with Raven's Progressive Matrices IQ test (source: adapted from an image in Shaohan Huang et al.)

The prompt's context includes a text-based instruction, "Here are eight images:" followed by each possible image completion wrapped in "The following image is:" and "Is it correct?" The model returns a probability distribution across all the possible images that can complete the matrix. The image with the highest probability is the predicted answer. By comparing to RPM's ground truth answer, you can determine the model's accuracy for this nonverbal reasoning task.

Now that you've seen examples of various multimodal generative AI tasks, let's dive deep into the powerful diffusion architecture that powers many of these multimodal models, including Stable Diffusion.

2 Shaohan Huang et al., "Language Is Not All You Need: Aligning Perception with Language Models", *arXiv*, 2023.

The rest of this chapter is very technical and dives deep into how diffuser-based models were built and trained. You may wish to use it as a reference for debugging and tuning diffuser-based generative models in the future; however, it is not required to understand how to use these models. Feel free to skip to Chapter 11 to explore ways to control image generation and fine-tune multimodal generative models for your use cases and datasets.

Diffusion Architecture Fundamentals

Diffusion models support a variety of key tasks for multimodal models, including image generation, upscaling, and inpainting. Early multimodal models often utilized variational autoencoders (VAEs) followed by the next generation of multimodal models created using generative adversarial network (GAN) architectures. However, most of the recent multimodal models use diffusion-based architectures, including Stable Diffusion.

Diffusion-based architectures are a common choice for recent multimodal foundation models because they offer a high degree of control in quality and diversity of images generated. This architecture has three primary components to cover, including the processes of forward diffusion and reverse diffusion, combined with the underlying U-Net architecture (which will be described in "U-Net" on page 223).

Forward Diffusion

The first step in training a diffusion model includes providing data as input that goes through a process called *forward diffusion*, as shown in Figure 10-16.

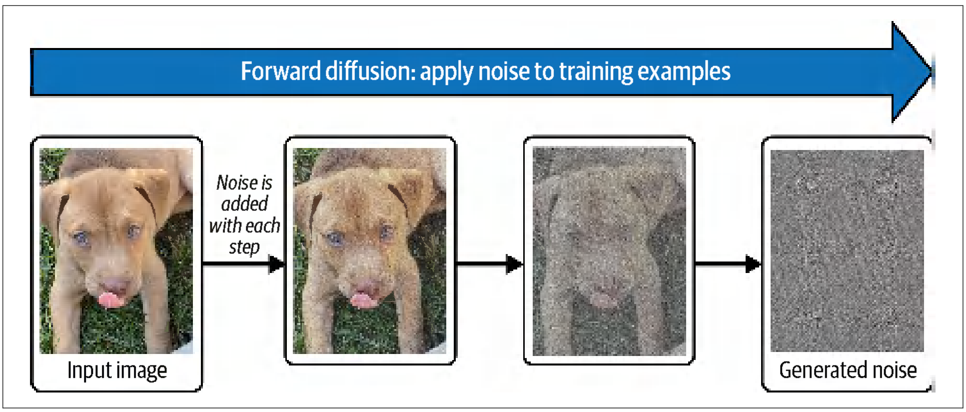


Figure 10-16. Training examples are created by applying noise to input images

Forward diffusion applies Gaussian noise to the input over a series of steps, depending on the amount of noise to be applied. At a high level, the noise is really random pixels or distortions applied to the image.

The forward-diffusion process is how training examples are created, so this same process is applied to multiple input images to create a number of training examples that will then be used for the image generation model. During this process, you are able to control the amount of noise that gets added to the image over a series of steps, which also means you're able to create multiple training examples per image, with varying degrees of noise applied, for each of the images in the training dataset.

Reverse Diffusion

Once you have your training examples, a second model is trained to predict noise in an image then removes the noise to generate an image. This process is known as *reverse diffusion*. Reverse diffusion takes the noisy image on input, along with a number of denoising steps, to create a clearer image. During the reverse-diffusion process, the noise in the image is predicted using the trained noise predictor, then removed and replaced with an image that is closer to the distribution the model was trained on, as shown in Figure 10-17.

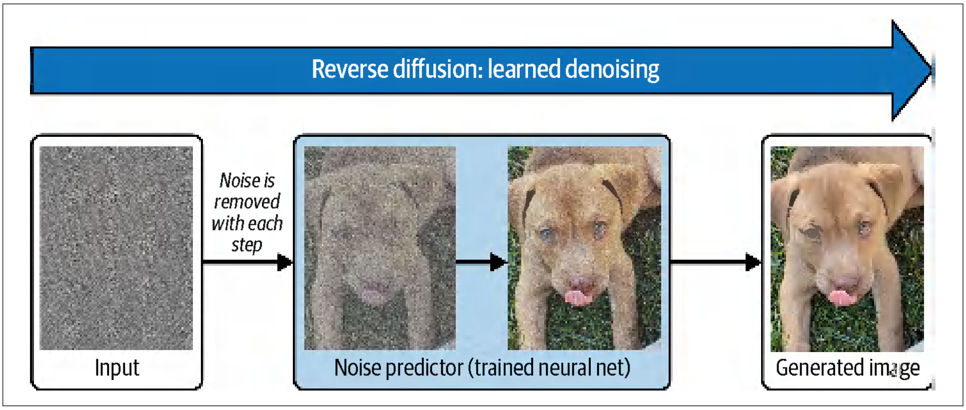


Figure 10-17. Reverse diffusion removes noise from an input to generate a new image

Although there are different types of diffusion-based model architectures, they all follow the same principle of adding noise during training and then training a neural network to reverse the noise. The most common underlying neural network is U-Net, which was originally introduced in a 2015 research paper.[3]

3 Olaf Ronneberger et al., "U-Net: Convolutional Networks for Biomedical Image Segmentation", *arXiv*, 2015.

U-Net

At a high level, the trained U-Net model is made up of an encoder followed by a decoder. The encoder is responsible for extracting features from the input image. The encoder has repeated convolutional layers to extract intermediate features and then max pooling layers to perform the downsampling, as shown in Figure 10-18.

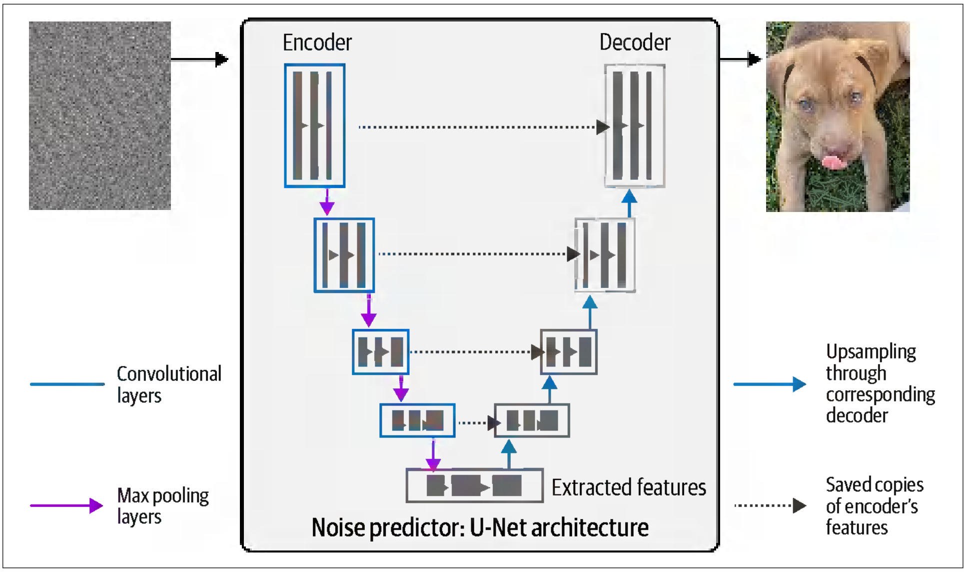


Figure 10-18. U-Net architecture common in diffusion-based foundation models

The corresponding decoder then upsamples the extracted features with saved copies of the encoder's features concatenated on the decoder's features by connected paths. The final layer then produces the output—in this case, the final generated image. Because the encoder and the decoder are symmetrical and connected by paths, it forms the U shape, resulting in its name, U-Net.

In summary, diffusion-based architectures have three primary components. The first is a process known as forward diffusion used to create training examples by adding a determined amount of noise to an image over a series of noising steps. The images created through forward diffusion are then used to create a noise predictor, typically utilizing a U-Net architecture, which can then be used to predict noise and reverse the added noise, through reverse diffusion, in order to generate new images.

This architecture serves as one of the foundational components for many multimodal models, including Stable Diffusion. In the next sections, we'll dive deeper into the Stable Diffusion 2 and Stable Diffusion XL architectures.

Stable Diffusion 2 Architecture

Stable Diffusion is a latent diffusion model (LDM) supporting image generation and image modification tasks. You can use Stable Diffusion as is or to fine-tune for your specific task. The power of these multimodal models is the ability to provide instructional text within the prompt to control the image that gets generated.

In addition to the prompt itself, there are also built-in configurations that allow you to control the image generated, such as the ability to supply a negative prompt that excludes specific elements from the generated image. These controls are covered in more detail in a bit, but first let's understand more about the Stable Diffusion architecture and how it works.

Similar to other foundation models, there are different versions of Stable Diffusion that vary in the training data and underlying components used within each model's architecture.

Stable Diffusion 2 was created by researchers from CompVis, Stability AI, and LAION—and was trained on a subset of the LAION-5B dataset, which contains 5 billion image-text pairs where approximately 50% are English and 50% are from 100+ other languages. The text associated with each image, called the caption, represents the contents of the image, as shown in Figure 10-19.

Figure 10-19. Image-caption pairs

Stable Diffusion is not a single model but a collection of components and models that form the foundation of the underlying architecture that is able to understand multiple modalities, including both text and image data. The key elements of the Stable Diffusion architecture include a text encoder, a diffusion process, and an image decoder, as shown in Figure 10-20. Each of these elements has its own corresponding neural network.

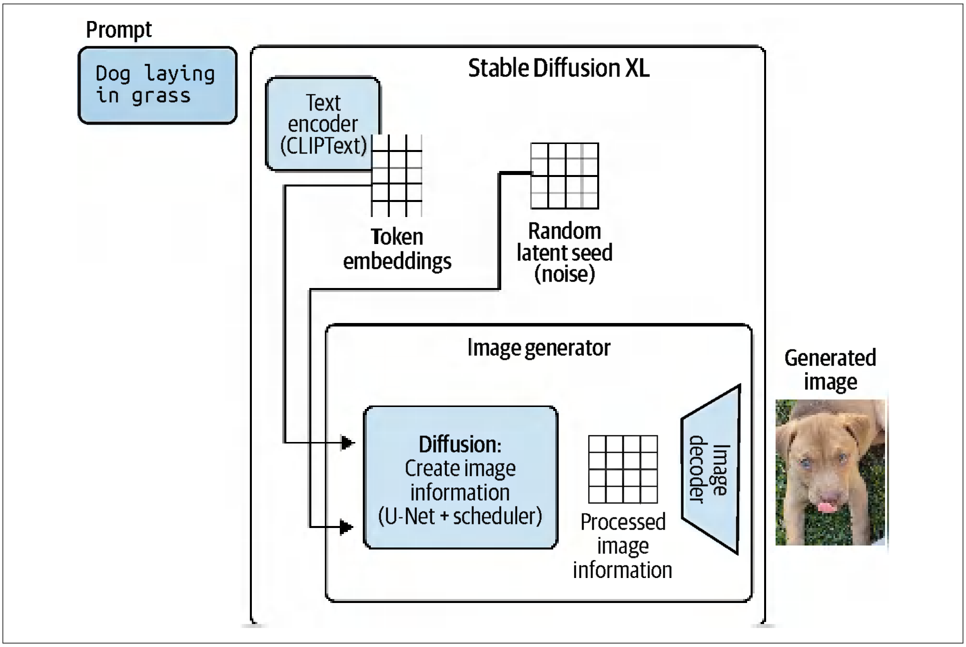


Figure 10-20. Stable Diffusion architecture for text-to-image generation tasks

Let's discuss each of these components in more detail, starting with the text encoder.

Text Encoder

In the case of Stable Diffusion v2.1, the text encoder is a pretrained, Transformer-based model called OpenCLIP. This model is pretrained on 32 billion text-image pairs and allows you to compute representations of images and text and then measure how similar they are, making it ideal for image classification, image retrieval, and image generation. In the specific example of text-to-image generation, the text encoder takes the input text and converts it into token embeddings that represent the input text.

The underlying language model contributes significantly to the performance of multimodal models like Stable Diffusion. Each new version of Stable Diffusion models has shown the trend to continue modifying the underlying architecture to use the most current and largest large language models to continue to improve performance with each new version.

OpenCLIP is pretrained for both image encoding and text encoding using the multimodal pretraining dataset with image and text pairings. An example illustrating the way OpenCLIP is trained is shown in Figure 10-21.

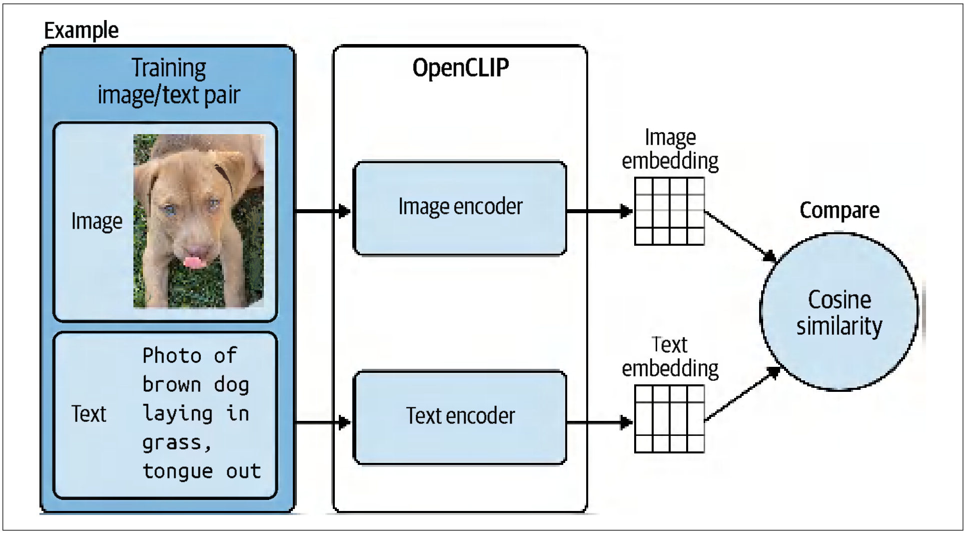


Figure 10-21. How OpenCLIP is trained using image and text pairs

During initial training, the similarity between text and image is expected to be low; however, as the model is updated by repeating the process through the training dataset, the resulting encoders are able to produce embeddings where an image and the matching text are increasingly similar.

For this process to be effective, the training data also needs to include negative examples where the text and the image do not match, in which case the model should assign low similarity scores. For text-to-image tasks, Stable Diffusion takes advantage of OpenCLIP's text encoder to convert the input prompt into token embeddings.

Next, the token embeddings are fed into the second component of the architecture, which utilizes the diffusion architecture discussed in the previous section.

U-Net and Diffusion Process

As previously mentioned, Stable Diffusion is an LDM, meaning it operates in a latent space, which has proven faster than previous models that operated in the pixel space. Stable Diffusion has two latent spaces: a prompt/text latent space and an image representation space.

The generated output is still represented in the pixel space in the form of a generated image; however, the computations within the diffusion process all happen in the latent space, which is less computationally intensive. The first input includes the token embeddings from the input text supplied in the prompt.

The text embeddings are used multiple times by the noise predictor in the U-Net, and the U-Net consumes these tokens through a cross-attention mechanism that will be discussed in more detail in a bit. The second input is a random array of noise, known as the latent seed. You can optionally control this array by setting the seed value when prompting the model. If you set the seed to a specific value, you will always get the same tensor array as the input noise array; otherwise, it is randomly generated.

One of the controls you can provide to Stable Diffusion is specifying the number of sampling steps in the U-Net architecture. Each step consumes the latent space array on input and produces another array that more closely aligns with the input text in combination with all of the other visual information the model identified from all of the images the model was trained on. The diffusion process is shown in Figure 10-22.

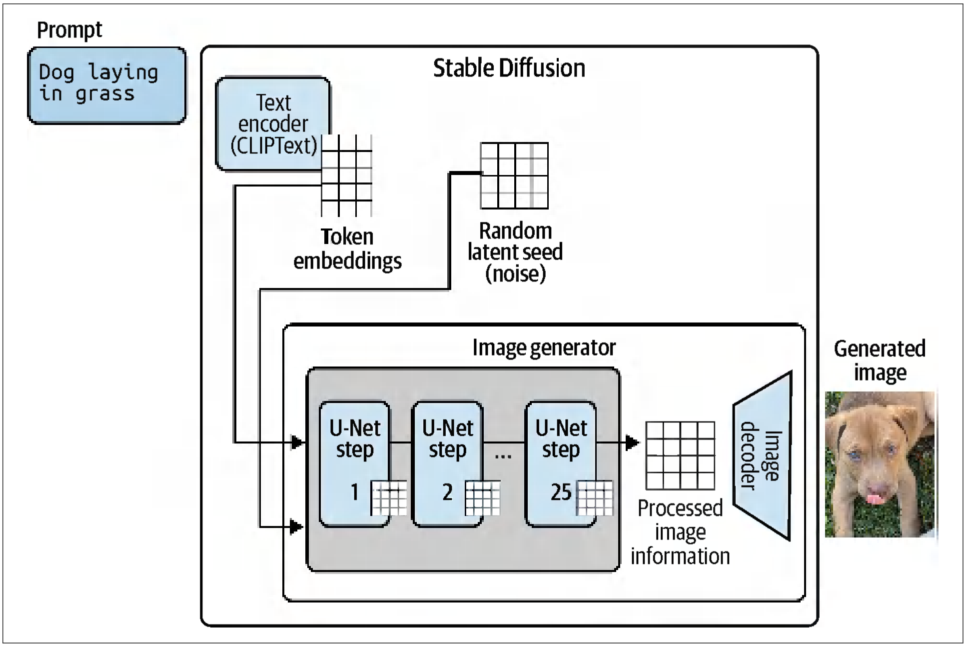


Figure 10-22. Using Stable Diffusion to generate a new image based on token embeddings and a noise array

If you were to visualize each of the latent arrays produced in each step through an image decoder, the resulting images would show reverse diffusion in action. Keep in mind that this U-Net architecture is a modification of the one previously discussed, which focused only on generating a random image. To be able to support text as well, the architecture is modified to add support for text inputs or instructions, which is called text conditioning.

Text Conditioning

Text conditioning involves adding attention layers between the network layers to process the text that is fed into the diffusion model. Other conditioning inputs (like semantic maps or images) are also valid, but in this case we'll focus on text-to-image, which specifically uses text conditioning.

Cross-Attention

The U-Net consumes these layers through a cross-attention mechanism that merges the text prompt and the image representations. If we zoom in on the U-Net architecture supporting each of the steps for Stable Diffusion, you'll see the addition of an attention layer for handling text embeddings, as illustrated in Figure 10-23.

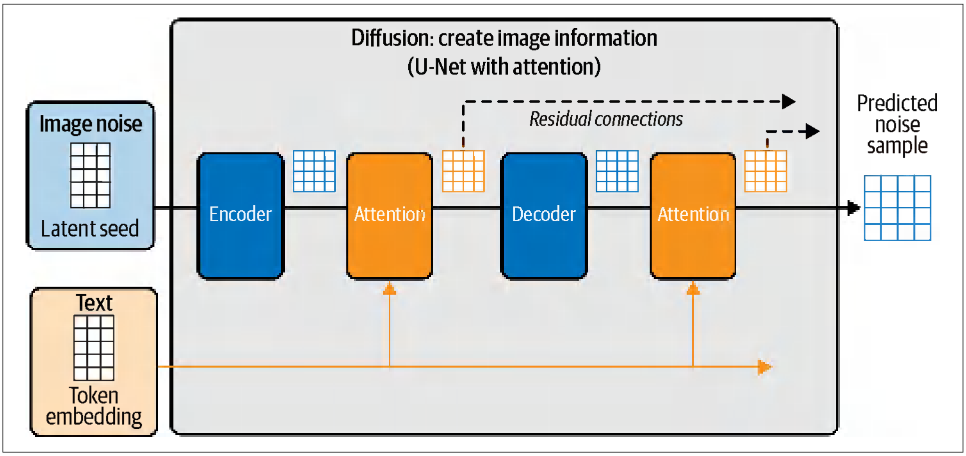


Figure 10-23. U-Net architecture with attention added for text conditioning

Some of the outputs are fed into additional processing later in the architecture through residual connections between the encoder and decoder. The cross-attention layers can be fine-tuned by using parameter-efficient fine-tuning (PEFT) techniques such as LoRA, as discussed in Chapter 6. The ability to fine-tune the cross-attention layers will be explored in Chapter 11.

Scheduler

Within the U-Net architecture, there is an additional key element called the scheduler, which is an algorithm. The U-Net architecture iteratively denoises the random latent seed image supplied on input while being conditioned to the text embeddings. Schedulers are used to control the denoising process in terms of the number of denoising steps and what algorithm to use to find the denoised sample.

The scheduler behaves differently depending on whether you're using the model for training or inference. During training, the scheduler takes a model output, referred to as a sample, from a specific point in the diffusion process and applies noise to the image according to a noise schedule and an update rule.

The noise schedule controls the noise level applied at each step. The noise is highest in the first step and gradually reduces through the iterative steps in the diffusion process. At each step in the process, the goal is to produce an image with a noise level that matches the noise schedule.

During inference, the scheduler is used to generate images from the noise; you can also specify controls, like how many images to generate. There are different scheduling algorithms that can be used to perform the computation, and Stable Diffusion supports a variety of available schedulers, many of which are conveniently packaged in Hugging Face's Diffusers library.

Image Decoder

The final output of the diffusion component includes the denoised latent image representation (seen as the process image information in Figure 10-22). This representation is then passed into the final component of the Stable Diffusion architecture, which is the image decoder.

The image decoder is actually an autoencoder that creates the final image using the processed image representation. This is when you are finally able to convert the latent space representation of an image into a visual pixel representation.

Stable Diffusion XL Architecture

Stable Diffusion XL is the latest foundation model from Stability AI and has enhancements, allowing for even more realistic images. XL has several image modification capabilities built-in, including inpainting, outpainting, and image-to-image.

So instead of utilizing a separate fine-tuned model from the base Stable Diffusion 2 model for inpainting, the XL model includes this in the base model. Several of the architecture components previously discussed apply to the XL architecture but there are several differences highlighted in this section that contribute to the advanced performance of this version.

U-Net and Cross-Attention

The XL architecture is using a U-Net backbone architecture that is three times larger than previous versions of Stable Diffusion. For comparison, XL has 2.6 billion U-Net parameters compared to 865 million in version 2. The modified architecture also includes more attention blocks at the lower layers of the U-Net and a larger cross-attention context used by a second text encoder. As a result of that second text encoder, XL supports two prompts, one for each encoder, that can be used to combine concepts, which can potentially help boost quality.

Refiner

The other significant enhancement to the XL model architecture includes the addition of a refinement model used to further enhance the fidelity of the generated image. As shown in Figure 10-24, this refinement model takes the output of the latent image produced by the base model and performs image-to-image enhancements.

Stable Diffusion 2 was trained on 768 × 768 pixel images that are then further compressed into latent space, but the optimal inference resolution remains consistent with the image size used in training. However, Stable Diffusion XL was trained on several aspect ratios and supports images between 768 and 1,024 pixels.

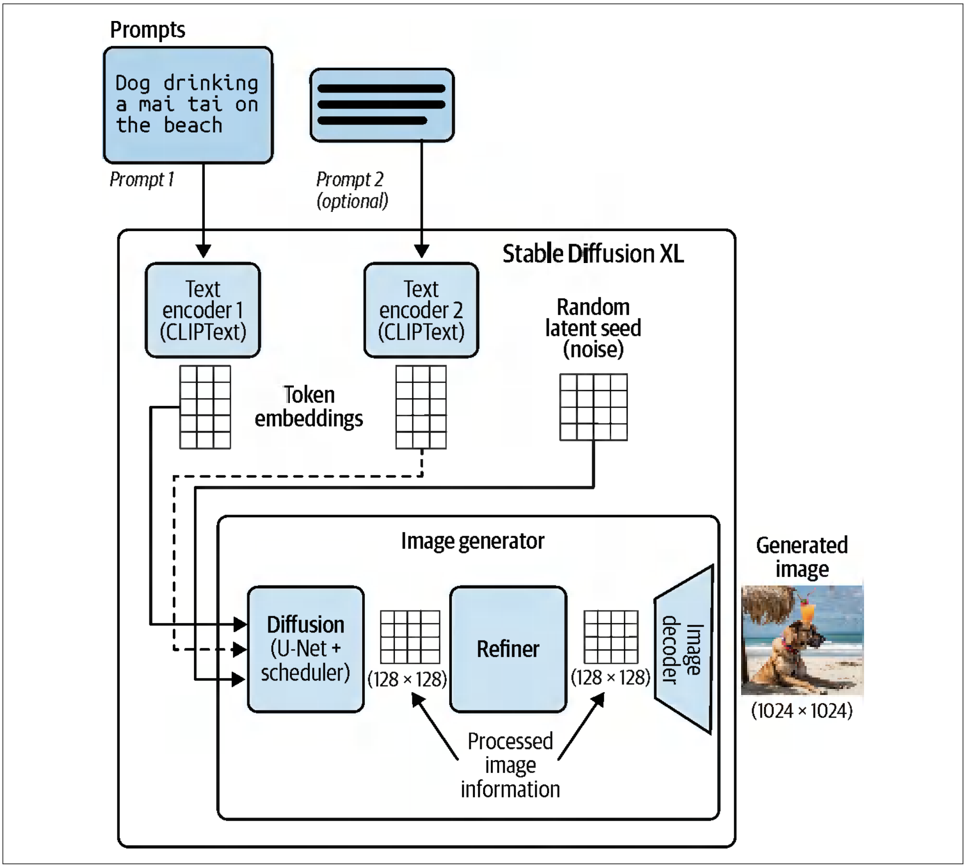


Figure 10-24. Stable Diffusion XL architecture

Conditioning

The XL model also includes two unique conditioning schemes to improve image generation. The first, is conditioning the model on the image size. In previous architectures, the limitations imposed on image size caused training samples to be dropped, impacting performance and the ability to generalize, or to be upscaled before training, often resulting in low image quality.

XL is conditioned using an additional input of image size and height. At inference, you can then set the desired apparent resolution. The second conditioning scheme is implemented to mitigate random cropping. Random cropping happens during training, but it can result in quality issues during image generation, such as a dog losing an ear. During data loading, the crop coordinates are noted and fed into the model as conditioning parameters.

The two-stage architecture of diffusion and refinement, as well as the additional conditioning for image size and cropping, helps improve the quality of generated images. Stable Diffusion XL also exposes more parameters during inference to control the output generated. For example, you can use `style_preset` (described in the section "Image Editing and Enhancement" on page 199) to give the model additional guidance on how the image should be generated, as shown in Figure 10-25.

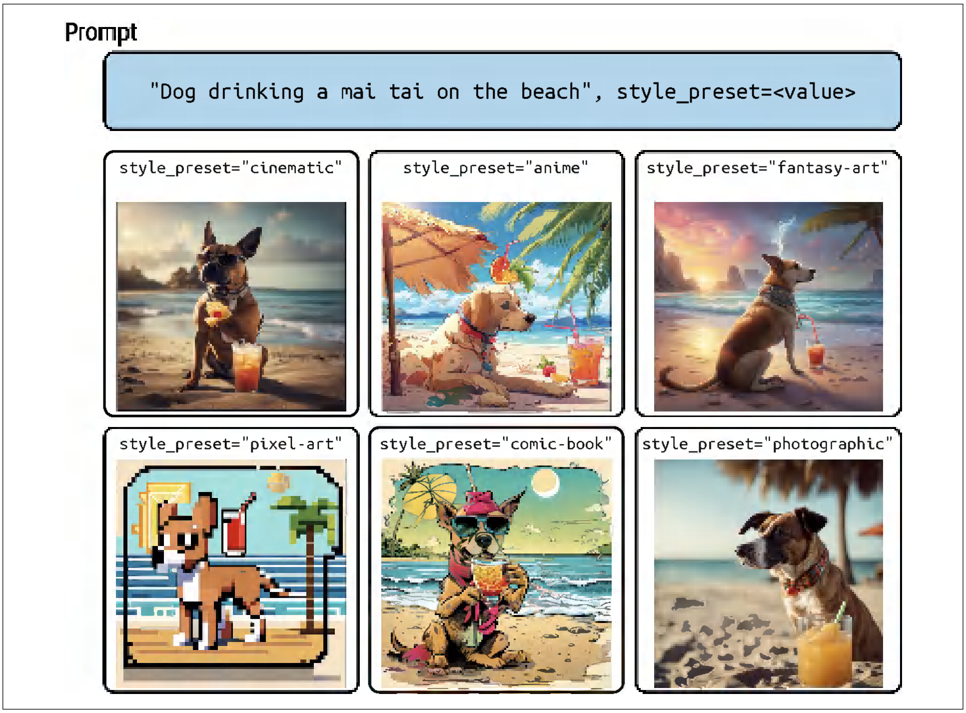


Figure 10-25. Stable Diffusion XL adds parameters to customize the generated image

Summary

Aligning perception with language using multimodal generative AI models is a very active area of research. This chapter highlighted some of the common multimodal generative AI tasks, including image generation, modification, captioning, classification, visual question answering, and nonverbal reasoning.

Next, you learned about diffusers and the evolution of the Stable Diffusion architecture. You also learned how to evaluate your multimodal generative AI models using datasets like PartiPrompts, ImageNet and VizWiz. In addition, you learned about Raven's Progressive Matrices and IQ test to evaluate the generative model's humanlike ability to perform nonverbal reasoning from symbols and images.

In Chapter 11, you will learn how to control image generation using Stable Diffusion and ControlNet. You will also see how to apply fine-tuning and reinforcement learning and enhancement (RLHF) to improve multimodal generation customized for your datasets and aligned to human preferences, such as helpfulness, honesty, and harmlessness.

CHAPTER 11

Controlled Generation and Fine-Tuning with Stable Diffusion

Controlling generation is an active area of research with many cutting-edge techniques introduced only recently. The goal of these techniques is to augment diffusion models to better handle common image tasks such as edge detection and segmentation maps. These techniques provide fine-grained control over image generation.

In this chapter, you will learn about a powerful technique called ControlNet to augment and improve text-to-image generation for models like Stable Diffusion. Additionally, you will explore multimodal fine-tuning with tools like DreamBooth, algorithms such as textual inversion, and optimizations including parameter-efficient fine-tuning (PEFT). Lastly, you will revisit reinforcement learning from human feedback (RLHF) in the context of aligning multimodal models with human preferences, including helpfulness, honesty, and harmlessness (HHH).

ControlNet

Described in a 2023 paper,[1] ControlNet is a popular way to train various controls that improve your image-based generative tasks. ControlNet is a deep neural network that works with diffusion models like Stable Diffusion.

During training, a control learns a specific task, such as edge-detection or depth-mapping, from a set of given inputs. A relatively small amount of data is required to train a very powerful control. You can train your own controls using ControlNet or choose from a large number of pretrained controls.

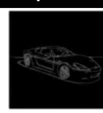
1 Lvmin Zhang et al., "Adding Conditional Control to Text-to-Image Diffusion Models", *arXiv*, 2023.

Let's use Figure 11-1 as the base image to apply some of the more common pretrained ControlNet controls. After applying a control to this base image, you can generate new images with Stable Diffusion that follow the guidance created by the output of the control.

Figure 11-1. Original base image to apply ControlNet controls to generate new images

Table 11-1 shows examples of some common pretrained ControlNet controls. These control examples are described in more detail in an AWS blog post in the context of generating new and creative marketing images using the base image.

Table 11-1. Example descriptions and image maps of conditional control

Control name	Control description	Control output
Canny edge map	A monochrome image with white edges on a black background	
Depth	A grayscale image with black (representing deep areas) and white (representing shallow areas)	
Hed boundary detector	A monochrome image with white soft edges on a black background	

Control name	Control description	Control output
Scribble	A hand-drawn monochrome image with white outlines on a black background	

You take the output from the control and pass it to Stable Diffusion to generate a new image with a new prompt—with the control output as the guide.

Table 11-2 shows examples of newly generated images that use the output of each control in Table 11-1—along with a new prompt—to guide the generation and create fun new images that look similar to the original image.

Table 11-2. Images generated by Stable Diffusion using each control

Control name	New prompt	Stable Diffusion with ControlNet
Canny edge map	`metal orange colored car, complete car, color photo, out doors in a pleasant landscape, realistic, high quality`	
Depth	`metal red colored car, complete car, color photo, out doors in a pleasant landscape on beach, realistic, high quality`	
Hed boundary detector	`metal white colored car, complete car, color photo, in a city, at night, realistic, high quality`	
Scribble	`metal blue colored car, similar to original car, com plete car, color photo, outdoors, breathtaking view, real istic, high quality, different viewpoint`	

Let's walk through how to use the Canny edge map control and detect edges using the lefthand image in Figure 11-2 as the base image.

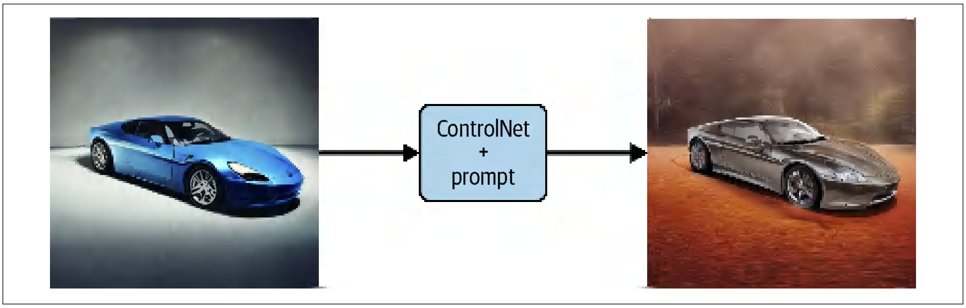


Figure 11-2. Converting base image into new image with ControlNet and a prompt

The code uses the OpenCV library to extract the edges using the Canny edge map ControlNet control:

```
from diffusers import StableDiffusionControlNetPipeline
from diffusers.utils import load_image

# Load the image
image = load_image("https://.../car.png"
)
# Render the canny edge map for this particular image
import cv2
from PIL import Image
import numpy as np

image = np.array(image)

low_threshold = 100
high_threshold = 200

image = cv2.Canny(image, low_threshold, high_threshold)
image = image[:, :, None]
image = np.concatenate([image, image, image], axis=2)
canny_image = Image.fromarray(image)
canny_image
```

Figure 11-3 shows the output of the Canny edge map control applied to the base image. This image represents the edges of each object in the base image.

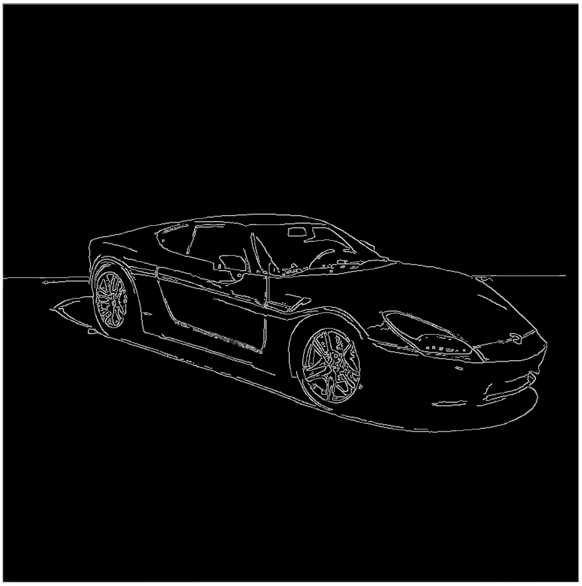


Figure 11-3. Canny edge map for the given image

This edge map is then passed to Stable Diffusion to control the generation of a new image using a new prompt, as shown in the following code example:

```
from diffusers import StableDiffusionControlNetPipeline
from diffusers import ControlNetModel
import torch
canny = ControlNetModel.from_pretrained(
    "lllyasviel/sd-controlnet-canny",
    torch_dtype=torch.float16)

sd_pipe = StableDiffusionControlNetPipeline.from_pretrained(
    "runwayml/stable-diffusion-v1-5",
    controlnet=canny,
    torch_dtype=torch.float16)

generator = torch.manual_seed(0)

out_image = sd_pipe(
    """
    metal orange colored car, complete car, color photo,
    outdoors in a pleasant landscape, realistic, high quality
    """,
    num_inference_steps=20,
    generator=generator,
    image=canny_image
```

```
).images[0]

out_image
```

Figure 11-4 shows the newly generated image from Stable Diffusion using the Canny edge map, which guides the generation of the new prompt, `metal orange colored car, complete car, color photo, outdoors in a pleasant landscape, realistic, high quality`. This process is useful for generating new and creative images that contain roughly the same objects as the original, guided by the ControlNet controls applied to the base image.

Figure 11-4. Newly generated image from Stable Diffusion using Canny edge map and prompt

While ControlNet and pretrained controls are very powerful, you may need to directly fine-tune a diffusion model with your specific image dataset to improve your generated images; for example, you may want to use a set of brand-specific logos or your product catalog. In the next section, you will learn some techniques to fine-tune Stable Diffusion using tools like DreamBooth and algorithms like textual inversion.

Fine-Tuning

Similarly to transformer-based large language models (LLMs), you can fine-tune diffusion models such as Stable Diffusion through various techniques. Fine-tuning allows you to customize image generation to include image data not captured in the original corpus of training data. This can include any image data, such as images of people, pets, or logos.

Fine-tuning allows you to generate realistic images that include subjects unknown to the pretrained model. A few common options for fine-tuning are included in this section, including DreamBooth, DreamBooth with LoRA, and textual inversion.

DreamBooth

DreamBooth originated from a research paper[2] in 2023, which introduced the method as able to personalize text-to-image models using just a few (three to five) sample images. While many use it for fun to generate their own personal images or images of their pets, it does have broader uses in generating creative content.

DreamBooth includes a number of applications for image generation, which will be outlined in detail later in this section. Fine-tuning using DreamBooth is done using a small sample of input images containing the subject you want to use for fine-tuning. You can also supply a unique identifier for the subject in your prompt. For example, Figure 11-5 uses "Molly" as the unique identifier for the "dog" subject.

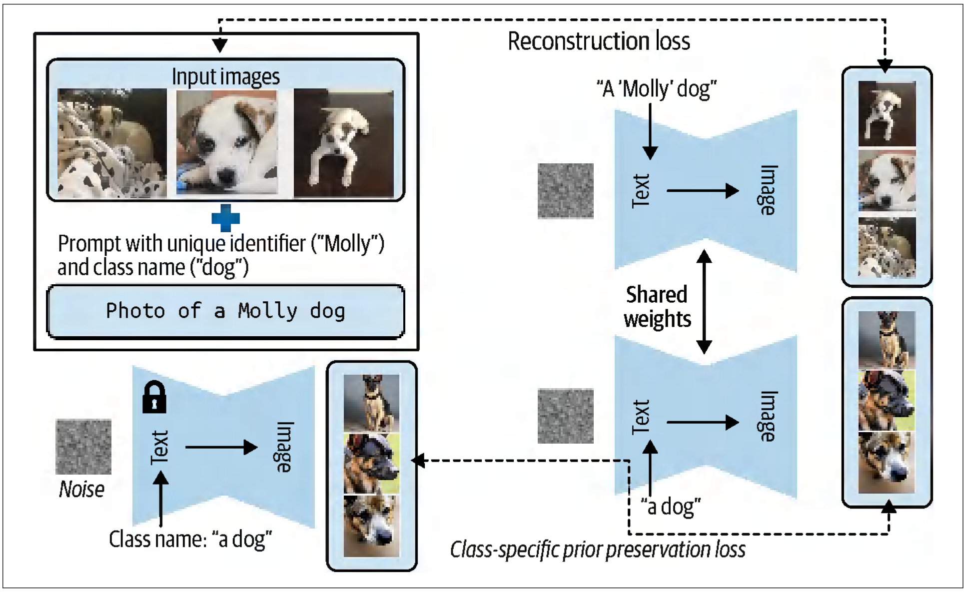


Figure 11-5. Fine-tuning a text-to-image diffusion model using DreamBooth

2 Nataniel Ruiz et al., "DreamBooth: Fine Tuning Text-to-Image Diffusion Models for Subject-Driven Generation", *arXiv*, 2023.

DreamBooth then uses those inputs to fine-tune a text-to-image diffusion model in parallel with a class-specific prior preservation loss that uses the semantic prior that the model has on the provided class to create diverse instances belonging to that subject's class as provided on input.

By fine-tuning using DreamBooth and a few input images, we're now able to create images of Molly in scenes she's never been in. This is an example of *recontextualization*. There are other uses of this application that can be applied to broader use cases such as marketing. As an example, the input image to fine-tune using DreamBooth could contain a new product. After fine-tuning, using that input image, that product (or subject) can then be used to generate images of the product with unique backgrounds or in different environments.

Art rendition is an application that allows you to create artistic depictions of your fine-tuned subject in the style of famous painters. As an example, you can generate creative content with images of your dog in a Vincent Van Gogh–style portrait, as shown in Figure 11-6.

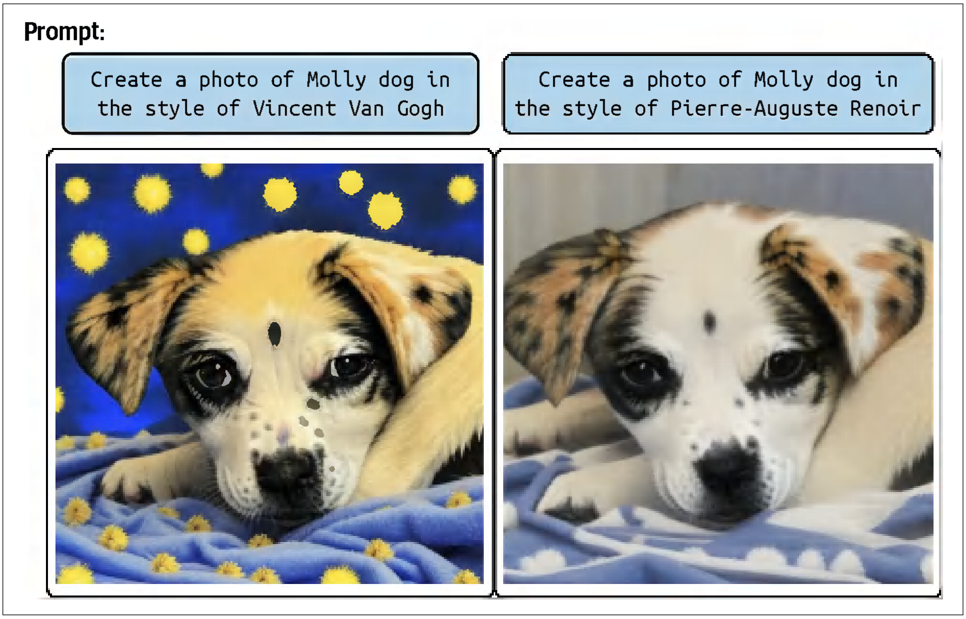


Figure 11-6. Art rendition using the DreamBooth fine-tuned model

Text-guided view synthesis is an application that allows you to synthesize images with specific viewpoints for a subject. Here, you can supply input images for fine-tuning, then generate different viewpoints on those pictures, such as viewing your dog from the side or the back, based on the instructions provided in the prompt. DreamBooth also supports property modification, which allows you to modify a specific aspect of the input image, such as color.

Finally, DreamBooth also supports accessorization, which allows you to preserve the subject in the input training images but modify the image with specific accessories like costumes or hats, as shown in Figure 11-7.

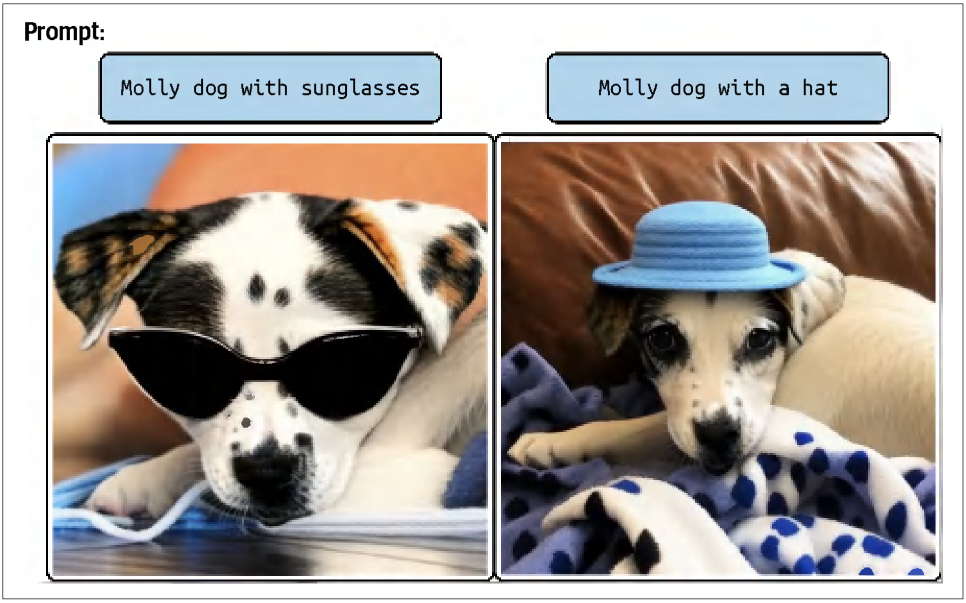


Figure 11-7. Accessorization using the DreamBooth fine-tune model

In this section, you learned how to fine-tune a Stable Diffusion model using DreamBooth and only a few images as input. DreamBooth uses the subject in those images combined with the provided dataset information containing a subject identifier and class to train a new model.

This is an entirely independent new model, but creating a new model for each subject or concept introduced may not be memory or storage efficient. In the next section, you will learn how to perform PEFT on Stable Diffusion with DreamBooth and LoRA.

DreamBooth and PEFT-LoRA

Chapter 6 introduced the concept of PEFT and Low-Rank Adaptation (LoRA) in the context of LLMs. LoRA can also be applied to multimodal models like Stable Diffusion. As previously discussed, the diffuser component of the Stable Diffusion architecture includes cross-attention layers that align images and text.

LoRA can be used to fine-tune those cross-attention layers using the same low-rank matrix approach discussed in Chapter 6, which results in a much smaller model adapter—typically 2 to 500 MBs versus roughly 5 GB for a Stable Diffusion model fully fine-tuned with DreamBooth. As described in Chapter 6, you will need to combine the artifact with the original Stable Diffusion model to perform inference.

Similar to language-based LoRA fine-tuning in Chapter 6, you can specify the LoRA rank and target modules for the Stable Diffusion model, as shown in the code sample, which targets the cross-attention layers:

```
target_modules = ["to_q", "to_v", "query", "value"]

config = LoraConfig(
  r=16,
  target_modules=target_modules,
  ...
)
model = get_peft_model(model, config)
```

Let's continue to use Molly as an example and take a look at the images generated with the fine-tuned model using LoRA. Keep in mind that the new fine-tuned model is only 10 MB in size, including the text encoder and the U-Net! The new model is prompted with this code:

```
img_list = pipe(["Molly dog on a beach"]*3, num_inference_steps=50).images
image_grid([x.resize((128,128)) for x in img_list], 1,3)
```

Three new images are generated with Molly dog on the beach. The new generated images shown in Figure 11-8 are similar in content and quality from the previously fine-tuned model.

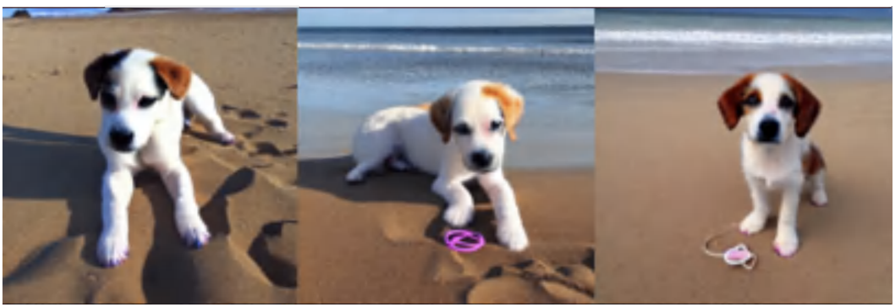


Figure 11-8. LoRA fine-tuned Stable Diffusion model with similar results

DreamBooth fine-tunes all of the parameters in the diffusion model, while keeping only the text transformer frozen, resulting in a new diffusion model. Next, you will learn a relatively lightweight fine-tuning technique called textual inversion, which is used to personalize image-based generative models with just a few images. This technique works by learning a token embedding for a new text-based token representing a concept while keeping the remaining components of the Stable Diffusion model frozen.

Textual Inversion

Textual inversion originated from a research paper[3] in 2022 that introduced a technique for personalizing text-to-image models by learning to represent new concepts in the embedding space while keeping the pretrained text-to-image model frozen. This method allows you to personalize text-to-image models using just a few sample images and without needing to alter the base foundation model.

Fine-tuning with textual inversion relies on a few sample images that represent a concept, such as an object or a style, in combination with a learnable token. The learnable token can be a pseudoword, such as "M*," or represent natural language phrases or sentences, such as "molly-dog." Then, during fine-tuning, the pseudoword is converted into tokens and the model learns to represent the concept through new word(s) in the embedding space. These learned embeddings are contained in adapters that are much smaller in size than the original or fine-tuned Stable Diffusion model.

Once the model has been tuned, the base foundation model is deployed along with the tuned textual inversion model, which is really a learned embedding. During inference, the prompt can take advantage of the learned token or pseudoword, as shown in Figure 11-9.

The prompt text containing the pseudoword is converted into tokens, which are then converted into embeddings. During training, the pseudoword was learned as a new token embedding, shown here as "V*." The model output is used to condition the diffusion model to be able to understand the prompt and new concept.

3 Rinon Gal et al., "An Image Is Worth One Word: Personalizing Text-to-Image Generation Using Textual Inversion", *arXiv*, 2022.

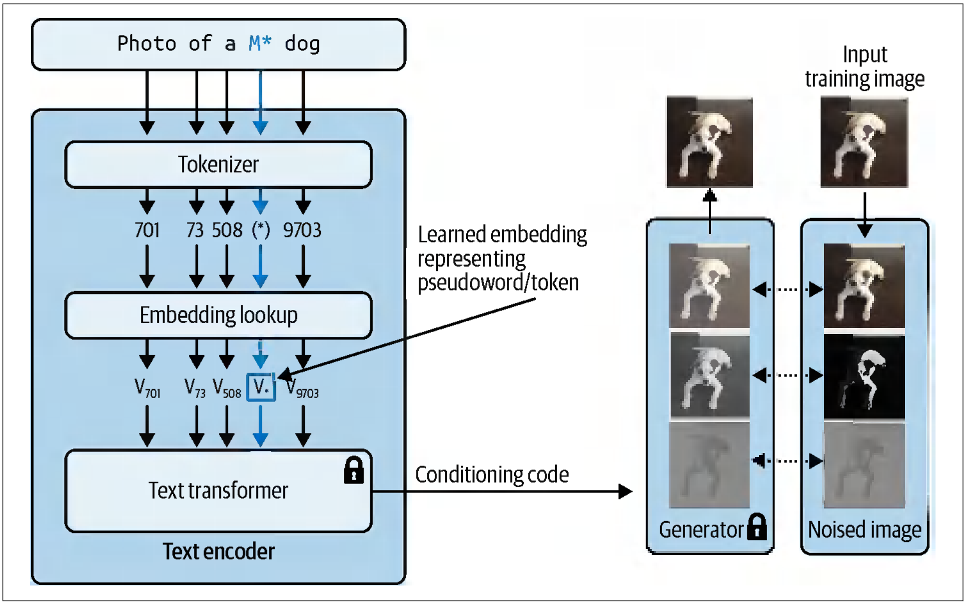


Figure 11-9. Textual inversion trains the text encoder on a pseudoword identifier of the custom concept

To tune a Stable Diffusion model using textual inversion, Hugging Face provides convenient libraries and training code in their Diffusers library. In this example, we supply a few images of Molly the dog in the training input along with key parameters (`object` or `style`) that guide the training including the concept to be learned, which is noted as the `learnable_property`. In this example, the `learnable_property` is an `object`, or more specifically, a dog. This guides the prompt templates that will be used as part of the training data, as shown in the code:

```
imagenet_templates_small = [
    "a photo of a {}",
    "a rendering of a {}",
    "a cropped photo of the {}",
    "the photo of a {}",
    "a photo of a clean {}",
    "a photo of a dirty {}",
    "a dark photo of the {}",
    "a photo of my {}",
    "a photo of the cool {}",
    "a close-up photo of a {}",
    "a bright photo of the {}",
    "a cropped photo of a {}",
    "a photo of the {}",
...
]
```

```
class TextualInversionDataset(Dataset):
    def __init__(
        self,
        data_root,
        tokenizer,
        learnable_property="object", # [object, style]
        placeholder_token="M*",
    ):
        self.templates = imagenet_templates_small
        ...
```

`placeholder_token` is the value you are going to use to represent your new concept. In the given example, we've identified `M*` to represent the concept, more specifically the `object`, to be learned. Again, the `object` is Molly in this case. This will also be the pseudoword, or token, that will be used in prompting to generate images containing the object identified in the images supplied during fine-tuning.

`initializer_token` is another important parameter shown in the following example. This parameter is used during fine-tuning to initialize word embeddings with single-word descriptions of the object. In this case, `initializer_token` is set to `dog` because Molly is a dog. Both `placeholder_token` and `initializer_token` are used together in the following code. The full code is in the GitHub repository associated with this book:

```
import torch
from transformers import CLIPTokenizer

model_checkpoint = "..." # CLIP model checkpoint

# Load tokenizer
tokenizer = CLIPTokenizer.from_pretrained(model_checkpoint)

initializer_tokens = ["dog"]
initializer_token_id =
  tokenizer.convert_tokens_to_ids(initializer_tokens)[0]

placeholder_tokens = ["M*"]
placeholder_token_ids =
  tokenizer.convert_tokens_to_ids(placeholder_tokens)

# Resize the token embeddings for pseudo-word tokens
text_encoder.resize_token_embeddings(len(tokenizer))

# Initialize the newly added placeholder token with
# the embeddings of the initializer token
token_embeddings = text_encoder.get_input_embeddings().weight.data
with torch.no_grad():
  for token_id in placeholder_token_ids:
    token_embeddings[token_id] =
      token_embeddings[initializer_token_id].clone()
```

Once the model is fine-tuned, you can deploy the model by loading it into a pipeline that includes the Stable Diffusion foundation model in addition to the trained model that has learned the pseudoword embedding. To do this, you again use the `StableDiffusionPipeline` class to load the original pretrained Stable Diffusion foundation model along with the adapted textual inversion model:

```
from diffusers import StableDiffusionPipeline
import torch

pipe = StableDiffusionPipeline.from_pretrained(
  "runwayml/stable-diffusion-v1-5")

pipe.load_textual_inversion(
  "./textual-inversion-molly/molly.pt", token="M*")
```

Once the model is deployed and ready for inference, you can send new prompts into the model that include the pseudo-word `M*`, for the `object` the model has been fine-tuned on, in this case Molly.

Prompt:

```
User: An oil painting of M*
```

The prompt is used by the pipeline to generate an image containing the `object` represented by the pseudoword provided—`M*`, in the prompt:

```
image = pipe(prompt, num_inference_steps=50).images[0]
image.save("molly-dog.png")
```

Figure 11-10 is the generated oil painting image of Molly. As you can see, we didn't use the text `Molly` in the prompt, but instead used the pseudoword, `M*`, that represents the object, Molly, identified during fine-tuning.

To summarize, textual inversion is a way to adapt a pretrained text-to-image model such as Stable Diffusion without performing full fine-tuning. This method allows for image generation using a concept, defined as either an `object` or a `style`, that is not included as part of the foundation models' original training data.

Next, you will learn how to fine-tune and align your Stable Diffusion model for human preference using RLHF.

Figure 11-10. Prompt completion for generated image using textual inversion adapter

Human Alignment with Reinforcement Learning from Human Feedback

It's possible to fine-tune diffusion models with reinforcement learning to improve things like image compressibility, aesthetic quality, and prompt-image alignment. This approach is similar to the RLHF process, which you explored in Chapter 7 to align large language models to generate more helpful, honest, and harmless text. The difference here is that RLHF is used to align multimodal models to generate content that is more helpful, honest, and harmless (HHH).

A proposed modification of the Proximal Policy Optimization (PPO) algorithm, which you learned about in Chapter 7, to apply RLHF to diffusion models is called Denoising Diffusion Policy Optimization (DDPO). In reinforcement learning (RL) terminology, each denoising step is an action. DDPO pays attention to the entire sequence of denoising steps in order to better maximize the reward of the final generated image. A sample implementation of DDPO for fine-tuning diffusion models, implemented in PyTorch with LoRA support, has been made available by the research authors on GitHub.

Let's look at an example. Say you want to apply RL to fine-tune Stable Diffusion to generate more aesthetically appealing images. In the scenario shown in Figure 11-11, you can use a model that's been trained on human preferences for aesthetically appealing images, such as LAION Aesthetics, as your RL reward model.

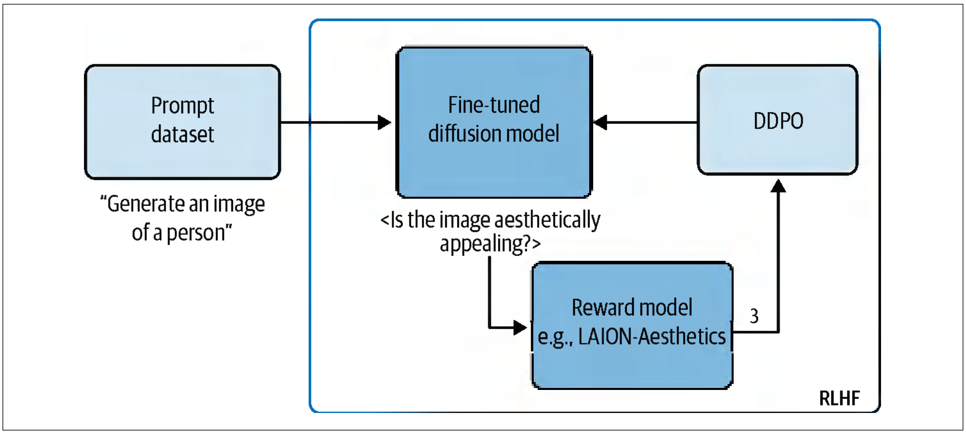


Figure 11-11. Fine-tuning a diffusion model with reinforcement learning and DDPO

The LAION-Aesthetics predictor has been trained on 176,000 human image ratings and predicts the rating people would give when they were asked, "How much do you like this image on a scale from 1 to 10?"

You could also fine-tune diffusion models in support of content moderation where the reward model returns a negative reward if the model generates inappropriate images. In such a scenario, you could use a managed service like Amazon Rekognition as the reward model, as it supports content moderation, as shown in Figure 11-12.

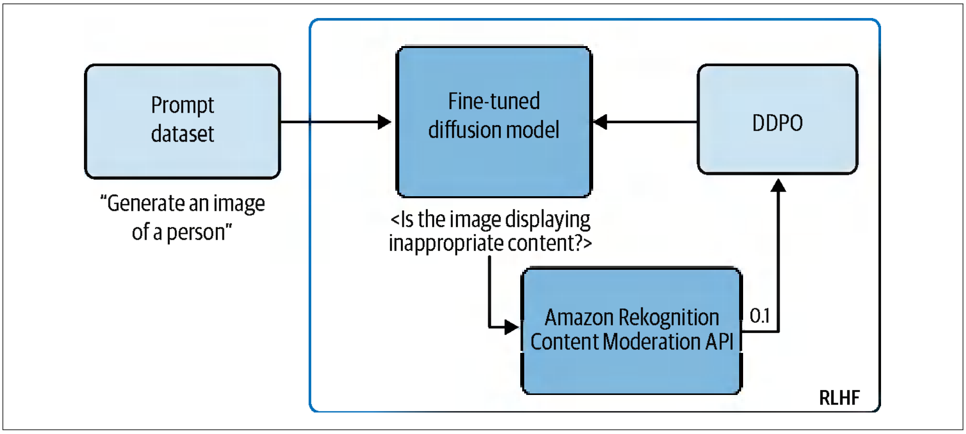


Figure 11-12. Using Amazon Rekognition as the reward model to detect unwanted content

Amazon Rekognition Content Moderation API uses deep learning to detect different types of inappropriate content. Beyond just flagging an image or video based on the presence of inappropriate or offensive content, it also returns a hierarchical list of labels with confidence scores. Here is a sample JSON response from Amazon Rekognition Content Moderation:

```
{
"ModerationLabels": [
    {
        "Confidence": 99.24723052978516,
        "ParentName": "",
        "Name": "Visually Disturbing"
    },
    {
        "Confidence": 99.24723052978516,
        "ParentName": "Visually Disturbing",
        "Name": "Air Crash"
    },
    {
        "Confidence": 88.25341796875,
        "ParentName": "Visually Disturbing",
        "Name": "Explosions And Blasts"
    }
]
}
```

Summary

In this chapter, you learned how to apply conditional controls to Stable Diffusion to influence how your model generates images. You also explored how to fine-tune multimodal generative AI models with your own custom datasets and human preferences using ControlNet, textual inversion, DreamBooth, PEFT, and RLHF.

In Chapter 12, you will learn how to use the Amazon Bedrock managed service for your generative AI use cases and tasks.

CHAPTER 12
Amazon Bedrock: Managed Service for Generative AI

Throughout the book, you have seen examples of Amazon SageMaker JumpStart for fine-tuning and deploying foundation models using SageMaker infrastructure. Amazon Bedrock, on the other hand, is a managed service that offers a completely serverless experience through a simple API.

In this chapter, you will explore Amazon Bedrock, including how to access the Bedrock API, the available foundation models (FMs), and Bedrock data privacy and network security. You will learn how to use Bedrock to implement retrieval-augmented generation, semantic-search, and agent-based use cases. You will also see how you can privately fine-tune the Bedrock foundation models using your own custom datasets.

First, let's discuss the available foundation models within Amazon Bedrock—and how to build upon those foundation models.

Bedrock Foundation Models

Amazon Bedrock supports foundation models from Amazon and various third-party companies, including AI21 Labs, Anthropic, Cohere, Meta, Stability AI, and others.

You access these foundation models through the AWS Management Console, AWS CLI, or AWS SDK. The code examples in this chapter will use the AWS SDK for Python called boto3. You can use the Bedrock Python function `list_foundational_models()` to see the most up-to-date list of available models.

Working with Amazon Bedrock is as simple as selecting a foundation model for your use case and then making a few API calls. You can use the Bedrock model playground to experiment with the available foundation models and select the one that fits your use case and dataset.

Remember that when evaluating different models, you should first try various prompt engineering techniques discussed in Chapters 2 and 10, including in-context learning with few-shot inference. You can also adjust the inference configuration parameters, including `temperature`, `top_p`, and `top_k`, as you learned in Chapter 2.

Amazon Titan Foundation Models

Amazon Titan foundation models are general-purpose models, pretrained on large datasets, that you can use as is or customize by fine-tuning the models with your own data for a particular task.

Titan Text are large language models for tasks such as text summarization, text generation, classification, question-answer, and information extraction. They are also trained on different programming languages, as well as rich text format (RTF), including tables, JSON, and CSV.

The Titan Text Embeddings model translates text inputs, such as words, phrases, or possibly large units of text, into numerical representations known as embedding vectors. As you learned in Chapter 1, embedding vectors capture the semantic meaning of the text in a high-dimension vector space.

After converting your documents into embeddings, you can store the embeddings in a vector store capable of performing embedding-level tasks such as similarity search. With similarity search, you can write a query, convert it into an embedding, then search the vector store for documents that match your query text. Comparing embeddings often produces more relevant and useful contextual search results than traditional word or *n*-gram matching search algorithms.

Stable Diffusion Foundation Models from Stability AI

With Amazon Bedrock, you can access Stability AI's text-to-image and image-to-image foundation model, Stable Diffusion, as described in Chapter 11. Stable Diffusion can generate unique, realistic, high-quality images, art, logos, and designs with just a few words in a text-based prompt.

Next, you will explore the model inference APIs and start generating content with the foundation models available in Amazon Bedrock.

Bedrock Inference APIs

The following example performs a Bedrock Inference API request using the Python SDK (boto3) for Amazon Bedrock—specifically, the `invoke_model()` API—

to generate content using text-to-text models, text-to-image models, and embedding models. The `modelId` parameter identifies the foundation model you want to use:

```
import boto3
import json

bedrock_runtime = boto3.client(
    service_name='bedrock-runtime'
)

modelId = "..." # Amazon Bedrock foundation model

body = json.dumps(
    {
        "inputText": "This is where you place your input text"
    }
)

response = bedrock_runtime.invoke_model(
    modelId=modelId,
    body=body
)

response_body = json.loads(response.get("body").read())

print(response_body.get("results")[0].get("outputText"))
```

Here, and in most examples in this chapter, we are assuming the `body` JSON object uses `inputText` for the prompt. This may be different depending on the model. For more information on the latest models and prompt formats, see the Bedrock documentation.

Bedrock also offers an `InvokeModelWithResponseStream` API that lets you invoke the specified model to run inference using the provided input but streams the response as the model generates the output, as shown in Figure 12-1.

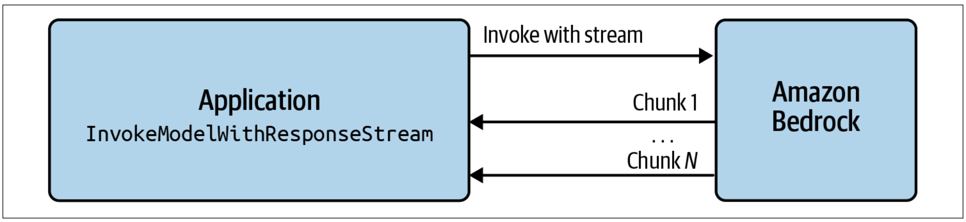


Figure 12-1. Bedrock's `InvokeModelWithResponseStream` API lets you start reading the response as soon as the first chunk is available

Streaming responses are particularly useful for responsive chat interfaces to maintain the liveness of an interactive application. Here is a Python code example using Bedrock's `InvokeModelWithResponseStream` API:

```
response = bedrock_runtime.invoke_model_with_response_stream(
    modelId=modelId,
    body=body)

stream = response.get('body')
if stream:
    for event in stream:
        chunk=event.get('chunk')
        if chunk:
            print(json.loads(chunk.get('bytes').decode))
```

Next is a deeper dive into Amazon Bedrock's Inference API for large language models.

Large Language Models

As discussed in Chapter 2, foundation models expose a set of generative configuration parameters that influence the model's output during inference. These configuration parameters give you control of the model's response, including the diversity and number of tokens. The available parameters may be different across model providers and model families, but most models support `temperature`, `top_k`, and `top_p`.

Here's an example Bedrock Inference API request using the `invoke_model()` API that includes configuration parameters for a prompt using a Bedrock model:

```
import boto3
import json

bedrock_runtime = boto3.client(
    service_name='bedrock-runtime'
)

prompt = "<your prompt here>"

body = json.dumps({
   "inputText": "This is where you place your input text",
   "textGenerationConfig": {
      "temperature":0,
      "topP":1
    }
})

modelId = '...' # Amazon Bedrock foundation models:
                # Amazon Titan Text
                # Anthropic Claude
                # AI21 Jurassic
```

```
                                # Cohere Command
                                # Meta Llama2
                                # etc.

response = bedrock_runtime.invoke_model(
    body=body,
    modelId=modelId)

response_body = json.loads(response.get('body').read())

print(response_body.get('results')[0].get('outputText'))
```

Generate SQL Code

Many text generation models, including those available in Amazon Bedrock, have been pretrained on vast amounts of text data, including code samples. In fact, code generation was one of the earliest use cases for generative models and services like Amazon CodeWhisperer and GitHub Copilot.

The example here uses Amazon Bedrock to generate a SQL query equivalent to `SELECT id FROM students ORDER BY age DESC LIMIT 1` using a natural language prompt that first defines the table, then describes the query:

```
prompt = """
I have a table called 'students' with fields 'id', 'age', 'year_enrollment',
'subject', 'grade'. Write me a SQL Query that returns the 'id' with the highest
'age'.
"""
body = json.dumps({"inputText": prompt})

modelId = '...'

response = bedrock_runtime.invoke_model(
    body=body,
    modelId=modelId)

response_body = json.loads(response.get('body').read())

print(response_body.get('results')[0].get('outputText'))
```

Summarize Text

Another popular generative AI use case is summarizing text. Let's build a prompt that asks the model to summarize the given passage wrapped in `<text></text>`, as shown here:

```
prompt = """
Please provide a summary of the following text. Do not add any information that
is not mentioned in the text below.

<text>
```

```
AWS took all of that feedback from customers, and today we are excited to
announce Amazon Bedrock, a new service that makes generative foundation models
accessible via an API. Bedrock is the easiest way for customers to build and
scale generative AI-based applications using FMs, democratizing access for all
builders.
</text>
"""
```

Next, define the API request body that includes the prompt, in this case called `inputText`, and the text generation configuration settings:

```
body = json.dumps(
    {
        "inputText": prompt,
        "textGenerationConfig":{
            "maxTokenCount":128,
            "temperature":0,
            "topP":1
        }
    }
)
```

Now you can send the API request to Bedrock. You can do this using the `invoke_model_with_response_stream()` API:

```
import json

response = bedrock_runtime.invoke_model_with_response_stream(
    body=body,
    modelId=modelId)
stream = response.get('body')
output = []

if stream:
    for event in stream:
        chunk = event.get('chunk')
        if chunk:
            chunk_obj = json.loads(chunk.get('bytes').decode())
            text = chunk_obj['outputText']
            output.append(text)

print(''.join(output))
```

Next, you will see how to generate embeddings with Amazon Bedrock.

Embeddings

As discussed in Chapter 3, embeddings are a key concept in generative AI and machine learning in general. An embedding is a representation of an object, such as a word, an image, or a video, in a vector space. Semantically similar objects will have embeddings that are closer together in the vector space, as you saw in Chapter 9, in the context of retrieval-augmented generation (RAG) to augment your prompts.

You can use Amazon Bedrock models to retrieve the embedding vector for any input string. You can then compare the distances between vectors to find the most related text strings. Common use cases for embeddings include semantic search, recommendations, and classifications.

A popular use case of embeddings is to cluster together text with similar semantic meaning. In the next code example, you will generate a heatmap that shows that documents originating from the same category—either animals, US cities, or colors, in this case—have much closer embedding vectors than documents from different categories.

First, define a `get_embedding` function that calls the Bedrock API and uses a Titan Text Embeddings model to generate an embedding. It will return the actual embedding from the API response body, as shown in the code:

```
def get_embedding(body, modelId, accept, contentType):
    response = bedrock_runtime.invoke_model(
            body=body,
            modelId=modelId)

    response_body = json.loads(response.get('body').read())

    embedding = response_body.get('embedding')

    return embedding
```

To test the code, you can use the following sample input text:

```
body = json.dumps(
    {
        "inputText": "<your prompt here>"
    }
)

modelId = '...'

embedding = get_embedding(body, modelId)
print(embedding)
```

The function will return the embedding vector retrieved from the Bedrock API response, similar to this:

```
[0.53515625, -0.0546875, -0.049804688, -0.16992188,
0.42382812, 0.15234375, -0.10839844, ...]
```

Next, you will generate the heatmap that visualizes the distance between any pair of sentences in the embedding space. The distance between any pair of sentences is computed by the cosine similarity of corresponded embedding vectors. Note that the cosine similarity of two vectors is the inner product of the normalized vectors scaled to unit length 1.

```
import sklearn
from sklearn.preprocessing import normalize
import numpy as np
import seaborn as sns

def plot_similarity_heatmap(text_labels, embeddings, rotation):
    inner_product = np.inner(embeddings, embeddings)
    sns.set(font_scale=1.1)
    graph = sns.heatmap(
        inner_product,
        xticklabels=text_labels,
        yticklabels=text_labels,
        vmin=np.min(inner_product),
        vmax=1,
        cmap="BuPu",
    )
    graph.set_xticklabels(text_labels, rotation=rotation)
    graph.set_title("Semantic Textual Similarity Between Sentences")
```

Next, define a few sentences and create the embeddings using Amazon Bedrock:

```
phrases = [
    # Animals
    "Shelbee's dog, Molly, is so cute.",
    "Antje hates cats.",
    "Chris's first dog was very cute.",
    # U.S. Cities
    "Chicago is the place where I'm from.",
    "I work in San Francisco.",
    "Washington D.C. is a great place to visit.",
    # Color
    "What is your favorite color?",
    "Is Molly brown?",
    "Are your eyes blue?"
]

embeddings = []

for phrase in phrases:
    query_response = get_embedding(
      body=json.dumps({"inputText": phrase}),
      modelId="...")
    embeddings.append(query_response)

# Normalization before inner product
embeddings = normalize(np.array(embeddings), axis=1)

%matplotlib inline
%config InlineBackend.figure_format = 'retina'

plot_similarity_heatmap(phrases, embeddings, 90)
```

The output in Figure 12-2 shows animal phrases clustered together, while phrases about US cities, phrases, and colors are each clustered independently. The darker the color, the larger the cosine similarity (smaller the distance).

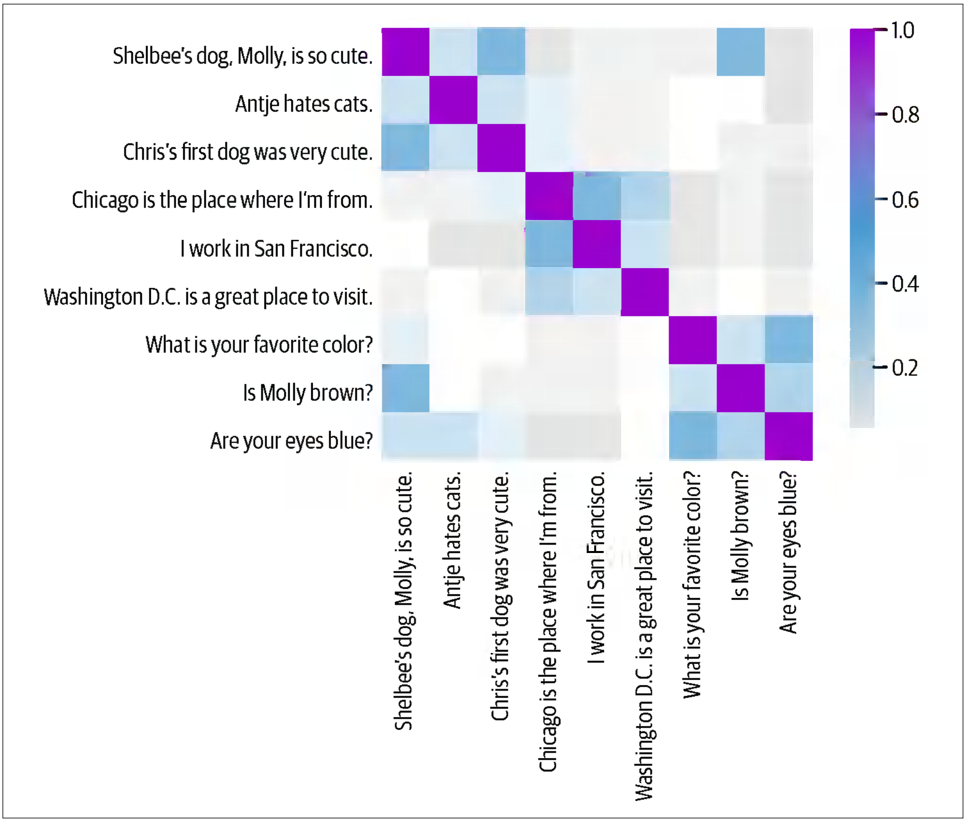


Figure 12-2. Heatmap showing semantic textual similarity between sentences

Next, you will learn how to fine-tune Bedrock foundation models with your own custom datasets.

Fine-Tuning

Now, what happens when you decide to customize the model? As soon as you fine-tune a model with your data, as shown in Figure 12-3, Amazon Bedrock deploys a custom model endpoint to host your fine-tuned model.

This becomes your own running instance of the model customized with your own dataset. Fine-tuned models are invoked the same way as base models: via the Amazon Bedrock Console playground or through an API. And remember that your model inputs and outputs will remain completely private to your environment and will not be accessible by anyone except you.

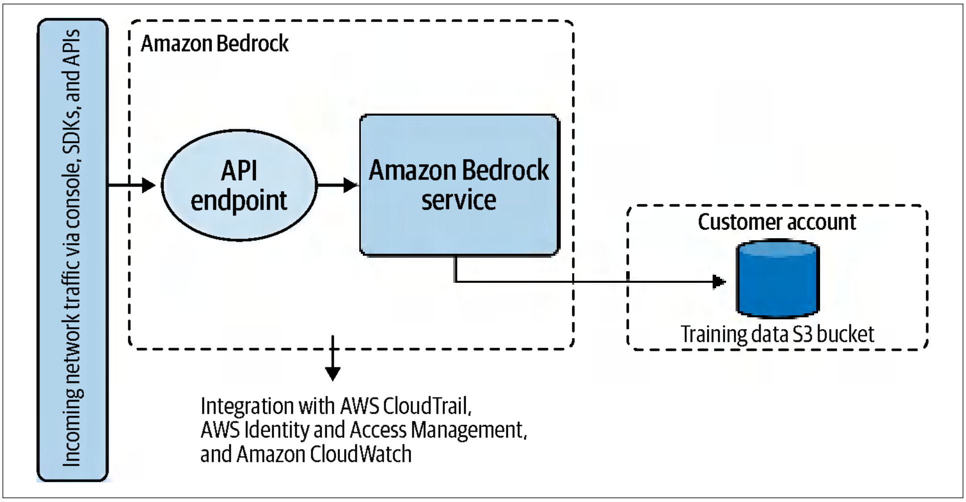


Figure 12-3. Amazon Bedrock model fine-tuning

Access between Amazon Bedrock and your S3 bucket can be configured securely, privately, and entirely on the AWS network backbone using VPC Endpoints (which are described in "Data Privacy and Network Security" on page 270).

After the fine-tuned model is trained, the weights are encrypted and delivered to the fine-tuned output bucket. Next, Amazon Bedrock creates and activates a hosting endpoint. Once the custom Bedrock hosting endpoint is activated, you can send inference requests with your prompts and receive completions from your fine-tuned generative AI model. Remember that model providers have no visibility or access to your fine-tuned weights.

With Amazon Bedrock, you can privately fine-tune foundation models using labeled data with just a few clicks or API calls. All you need is your dataset stored in S3 using the JSON Lines format, as shown here where `completion` is the label for the provided `prompt`:

```
{'prompt': 'I love going to the movies', 'completion': 'Positive'}
{'prompt': 'This new shirt is gorgeous', 'completion': 'Positive'}
{'prompt': 'The weather is awful', 'completion': 'Negative'}
{'prompt': 'This movie is terrible', 'completion': 'Negative'}
...
```

You then call `create_model_customization_job()` to start fine-tuning processing using your dataset and a given base foundation model available in Amazon Bedrock that supports fine-tuning. For the latest list of models that support fine-tuning in Amazon Bedrock, please see the Bedrock documentation.

Besides the training data, you also need to provide a job name, a name for the custom model, the base model identifier, where to store the fine-tuning outputs (e.g., training loss), and hyperparameters. An example fine-tuning job creation API call is shown in the following code:

```
import boto3

bedrock = boto3.client(service_name='bedrock')

input_training_data = "s3://<BUCKET>/train.jsonl"
output_data = "s3://<BUCKET>/output/"
bedrock.create_model_customization_job(
    jobName="my-job",
    customModelName="my-fine-tuned-model",
    baseModelIdentifier="...", # Bedrock foundation model
    trainingDataConfig={"s3Uri": input_training_data},
    outputDataConfig={"s3Uri": output_data},
    hyperParameters={
        ...
    }
)

fine_tuning_status = None
while fine_tuning_status != "Completed":
  fine_tuning_status = bedrock.get_model_customization_job(
    jobIdentifier="my-job")["status"]
print("Model was successfully fine-tuned!")
```

Once the fine-tuning job status changes to `Completed`, Amazon Bedrock can deploy your custom model accessible with the `invoke_model()` API. Once the model is deployed, you can invoke the model with your prompts, as shown in this code:

```
body = json.dumps(
    {
        "inputText": "I love this beach.",
        "textGenerationConfig":{
            "maxTokenCount":128,
            "temperature":0,
            "topP":1
        }
    }
)
response = bedrock_runtime.invoke_model(
    modelId=<deployed model identifier>,
    body=body)
```

In this example, you fine-tuned the model to classify input text as `positive` or `negative` sentiment. Therefore, the model completion for the inference request `I love this beach` will be `positive`.

Next, you will see how to use Amazon Bedrock to create fully managed agents capable of performing actions using AWS Lambda functions. AWS Lambda lets you run code without provisioning or managing servers.

Agents

With agents for Amazon Bedrock, you can build generative AI applications that manage and perform tasks by making API calls to your company systems. As you learned in Chapter 9, agents orchestrate prompt-completion workflows between user requests, foundation models, and external systems.

Similarly, agents for Amazon Bedrock make use of Bedrock's foundation models and advanced prompting strategies to understand user requests, break down complex tasks into multiple steps, carry on a conversation to collect additional information, and take actions to fulfill the request.

Using agents for Amazon Bedrock, you can automate tasks for your internal or external customers, such as managing retail orders or processing insurance claims. For example, an agent-powered generative AI ecommerce application can not only respond to the question, "Do you have this jacket in blue?" with a simple answer but can also help you with the task of updating your order or managing an exchange.

For this to work, you first need to select a Bedrock foundation model, then give the agent access to application APIs and knowledge bases, as shown in Figure 12-4.

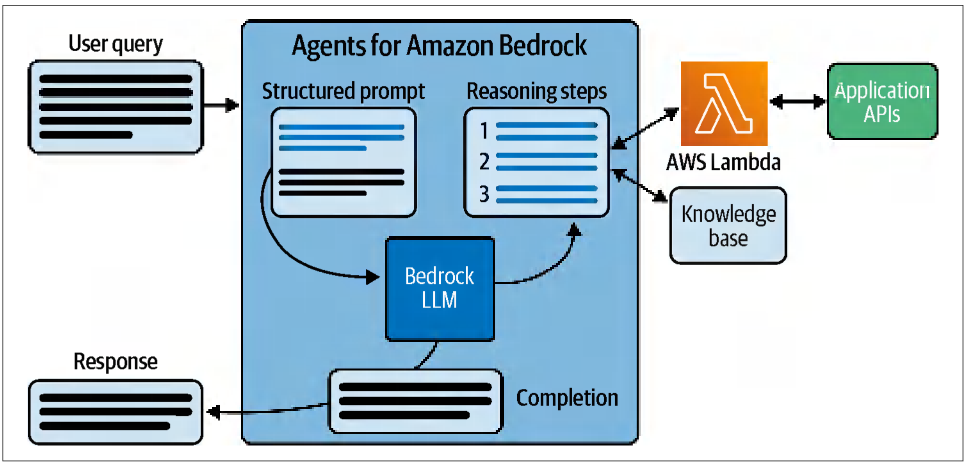


Figure 12-4. Agents for Amazon Bedrock can manage and perform tasks by making API calls or accessing a company knowledge base

Let's assume you are a developer at an insurance company and want to provide a generative AI application that helps the insurance agency owners automate repetitive tasks using an API. You first define the actions (API calls) that the agent is allowed to perform in an ActionGroup that is mapped to an AWS Lambda function.

The following code shows an example AWS Lambda function that implements the business logic to manage insurance claims by pulling a list of open claims and sending reminders to policyholders:

```
import json
import time

def open_claims():
    return {
        "response":
              ...
        }

def send_reminders():
    return {
        "response":
              ...
        }

def lambda_handler(event, context):
    api_path = event['apiPath']

    if api_path == '/claims':
        body = open_claims()
    elif api_path == '/send-reminders':
        body =  send_reminders()

    ...

    response_body = {
        'application/json': {
            'body': str(body)
        }
    }

    action_response = {
        'actionGroup': event['actionGroup'],
        'apiPath': event['apiPath'],
        'httpMethod': event['httpMethod'],
        'httpStatusCode': 200,
        'responseBody': response_body,
        ...
    }

    api_response = {
        'messageVersion': '1.0',
        'response': action_response,
```

```
        ...
    }

    return api_response
```

Together with the AWS Lambda function, you also need to provide an OpenAPI schema file with the API descriptions, structure, and parameters. Here is an example OpenAPI schema for the `/claim` API call:

```
{
  "openapi": "3.0.0",
  "info": {
    "title": "Insurance Claims Automation API",
    "version": "1.0.0",
    "description": "APIs for managing insurance claims for policyholder."
  },
  "paths": {
    "/claims": {
      "get": {
        "summary": "Gets the list of all open insurance claims",
        "description": "Gets list of open claims for policyholder.",
        "operationId": "getAllOpenClaims",
        "responses": {
          "200": {
            "description": "Gets list of open claims for policyholder.",
            "content": {
              "application/json": {
                "schema": {
                  "type": "array",
                  "items": {
                    "type": "object",
                    "properties": {
                      "claimId": {
                        "type": "string",
                        "description": "Unique ID of the claim."
                      },
                      "policyHolderId": {
                        "type": "string",
                        "description": "Unique ID of the policyholder."
                      },
                      "claimStatus": {
                        "type": "string",
                        "description": "The status of the claim, Open or Closed."
                      }
  ...
  }
}
```

When a user asks the agent to complete a task, Amazon Bedrock will use the FM you configured for the agent to identify the sequence of actions, invoke the corresponding Lambda functions in the right order to solve the user-requested task, and provide responses back to the user in natural language. For example, the virtual insurance-agent assistant can now perform tasks such as "send a reminder to all policyholders with policies needing renewal in the next 60 days."

With fully managed agents, you don't have to worry about provisioning or managing infrastructure. In addition, agents are integrated into the AWS services for monitoring, encryption, user permissions, and API invocation management. You can use agents for Amazon Bedrock to increase productivity, improve your customer service experience, or automate DevOps tasks.

The previous examples focused on the generating text and embeddings with the text-based models in Amazon Bedrock. Next, you'll see how to generate and modify images using Amazon Bedrock and an image-based model, Stable Diffusion.

Multimodal Models

For image generation use cases, Bedrock offers text-to-image and image-to-image models, including Stability AI's Stable Diffusion XL model. In Chapter 10, you explored the architecture that powers diffusion models, learned how to efficiently prompt image generation models, and how to apply advanced techniques such as inpainting and textual inversion to guide the image generation toward your desired output.

Next you will see how to generate images using prompts and negative prompts with Amazon Bedrock and Stable Diffusion from model provider, Stability AI.

Create Images from Text

To create images from text, start with a description of the image you want the model to create. As discussed in Chapter 10, you can provide some negative prompts to guide the model to avoid certain types of outputs. Note that you need to assign a negative weight to each negative prompt. After setting up the prompt, you call `bedrock.invoke_model()` to generate the image:

```
prompt = """
Golden retriever playing catch at a tropical, sunny beach
with palm trees in the background.
"""
```

```
negative_prompts = [
    "poorly rendered",
    "poor background details",
    "poorly drawn dog",
    "disfigured dog features",
]
request = json.dumps({
    "text_prompts": (
        [{"text": prompt, "weight": 1.0}]
        + [{"text": negprompt, "weight": -1.0} for negprompt in negative_prompts]
    ),
    "style_preset": style_preset,
    ...
})

modelId = "stability.stable-diffusion-xl"

response = bedrock_runtime.invoke_model(
  body=request, modelId=modelId)

response_body = json.loads(response.get("body").read())
```

Bedrock's `InvokeModel` provides access to the Stable Diffusion XL model by setting the right model ID and returns a JSON response that includes a Base64 encoded string representing the image. You can decode the Base64 string to binary and load it with an image processing library, such as Pillow, that can read PNG files. The generated output is shown in Figure 12-5:

```
import base64,
import io
import os
from PIL import Image

base_64_img_str = response_body["artifacts"][0].get("base64")

image_1 = Image.open(
  io.BytesIO(
    base64.decodebytes(bytes(base_64_img_str, "utf-8"))
  )
)

image_1
```

Figure 12-5. Image generated from text input

Create Images from Images

You can also start from an image—such as the Figure 12-5 image—and ask our Stable Diffusion model to change a detail. For example, you can change the breed of the dog to a poodle. To do this, you can make another request with the change request as well as the image you previously generated in a Base64 encoding. You can write a short helper function like this to convert images to Base64 encoding:

```
def image_to_base64(img):
    buffer = io.BytesIO()
    img.save(buffer, format="PNG")
    return base64.b64encode(buffer.getvalue()).decode("utf-8")
```

Now, you can make another Bedrock API request with the `change_prompt` and the previous image `init_image`, as shown here. Figure 12-6 shows the output of this image-modification request:

```
change_prompt = "Change the dog to be a poodle"
request = json.dumps({
    "text_prompts": (
        [{"text": change_prompt, "weight": 1.0}]
        + [{"text": negprompt, "weight": -1.0} \
          for negprompt in negative_prompts]
    ),
    "init_image": image_to_base64(image_1),
    "style_preset": style_preset,
```

```
    ...
})

modelId = "stability.stable-diffusion-xl"

response = bedrock_runtime.invoke_model(body=request,
  modelId=modelId)

response_body = json.loads(response.get("body").read())
image_2_b64_str = response_body["artifacts"][0].get("base64")
image_2 = Image.open(io.BytesIO(
  base64.decodebytes(bytes(image_2_b64_str, "utf-8")))
)

image_2
```

Figure 12-6. Changing a portion of the image using text prompt and original image

Next you will learn how Amazon Bedrock keeps your data private, including data encryption in flight and on disk.

Data Privacy and Network Security

With Amazon Bedrock, all of your prompts, completions, and fine-tuned models remain private to your AWS account. Your data is not used to improve the Bedrock service. Additionally, your data is not shared with third-party model providers.

All data is isolated per AWS customer and remains in the AWS region where Bedrock processes the data. This helps with General Data Protection Regulation (GDPR) and other regulations that require data sovereignty. All data is encrypted in transit over the network with a minimum of TLS 1.2 encryption. At rest on disk, your data—including your fine-tuned models—is encrypted with AES-256 using either AWS KMS–managed data encryption keys or your own customer-managed encryption keys.

For secure and private connectivity between your applications and the Amazon Bedrock generative AI–managed service, you can configure your AWS account and virtual private cloud (VPC) to use AWS VPC Endpoints. VPC Endpoints, built on AWS PrivateLink, use the private AWS network backbone to securely connect to the Amazon Bedrock service, as shown in Figure 12-7.

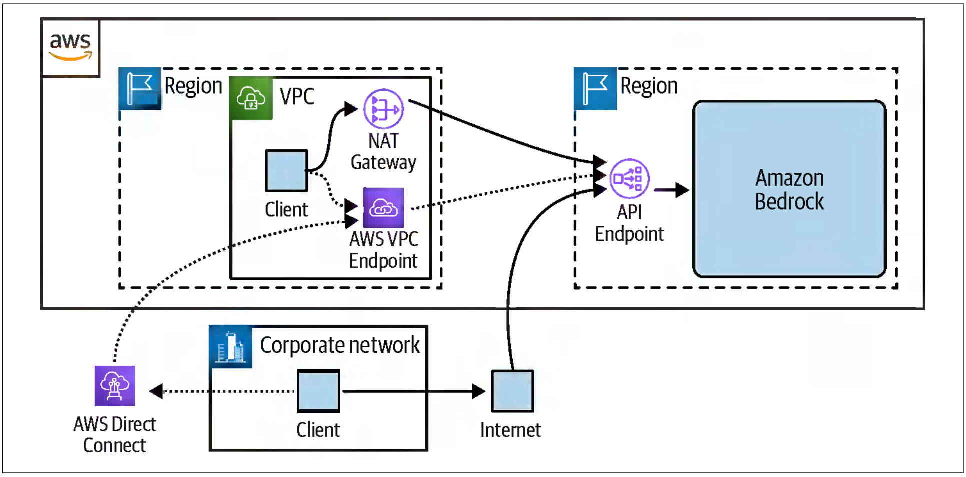


Figure 12-7. Network architecture diagram to connect applications to Amazon Bedrock

Figure 12-7 includes public connectivity through the internet and private connectivity using a VPC Endpoint. By using a VPC Endpoint, your data never needs to traverse the public internet. Instead, it remains on the low-latency, highly redundant private AWS backbone.

The diagram also shows how to connect to your AWS VPC using Direct Connect from your on-premises network. This gives you private connectivity from your network provider to the AWS VPC. From there, you can use a VPC Endpoint to keep all traffic between your private on-premises network to Amazon Bedrock entirely over the AWS network backbone. This avoids having to send any on-premises data over the internet.

Governance and Monitoring

Amazon Bedrock is integrated with AWS Identity and Access Management (IAM) to help you manage permissions, including access to specific foundation models and features such as fine-tuning. All AWS-managed service API activity—including Amazon Bedrock activity—is logged to the AWS CloudTrail service within your account. This activity monitoring helps you keep a record of who accessed which models—and when they accessed those models.

Amazon Bedrock also emits data points to Amazon CloudWatch to track common metrics such as `InputTokenCount`, `OutputTokenCount`, `InvocationLatency`, and (number of) `Invocations`. This near-real time telemetry helps you monitor usage and troubleshoot performance issues for your generative AI applications, integrating with the Amazon Bedrock service.

Summary

In this chapter, you learned how to use the managed Amazon Bedrock service for generative AI. You explored how to use foundation models for both text and image use cases. You also learned how to fine-tune and deploy a generative model using Amazon Bedrock and your custom text and image datasets. You also saw how to implement a context-aware reasoning application with agents for Amazon Bedrock. These agents augment a foundation model's behavior by using chain-of-thought reasoning with external data sources and API calls.

Next, you learned how Amazon Bedrock participates in your existing data privacy and network security profiles by supporting in-transit encryption with TLS, at-rest encryption with KMS, and private AWS networking with VPC Endpoints. Lastly, you learned that Amazon Bedrock privately tracks API activity and metrics using the AWS CloudTrail and Amazon CloudWatch within your AWS account.

Index

D

E

F

G

H

I

M

N

O

P

Q

R

S

T

About the Authors

Chris Fregly is a Principal Solutions Architect for generative AI at Amazon Web Services based in San Francisco, California. Chris holds every AWS certification. He is also cofounder of the global Generative AI on AWS Meetup. Chris regularly speaks at AI and machine learning meetups and conferences across the world. Previously, Chris was an engineer at Databricks and Netflix, where he worked on scalable big data and machine learning products and solutions. He is also coauthor of the O'Reilly book *Data Science on AWS*.

Antje Barth is a Principal Developer Advocate for generative AI at Amazon Web Services based in San Francisco, California. She is also cofounder of the global Generative AI on AWS Meetup and the Düsseldorf chapter of Women in Big Data. Antje frequently speaks at AI and machine learning conferences and meetups around the world. Prior to joining AWS, Antje worked in solutions engineering roles at MapR and Cisco, helping developers leverage big data, containers, and Kubernetes platforms in the context of AI and machine learning. She is also coauthor of the O'Reilly book *Data Science on AWS*.

Shelbee Eigenbrode is a Principal Solutions Architect for generative AI at Amazon Web Services based in Denver, Colorado. She is cofounder of the Denver chapter of Women in Big Data. Shelbee holds six AWS certifications and has been in technology for 23 years, spanning multiple industries, technologies, and roles. She focuses on combining her DevOps and ML backgrounds to deliver ML workloads at scale. With over 35 patents granted across various technology domains, Shelbee has a passion for continuous innovation and using data to drive business outcomes.

Colophon

The animal on the cover of *Generative AI on AWS* is the brown-backed mockingbird (*Mimus dorsalis*).

This nonmigratory, terrestrial mockingbird inhabits arid landscapes of brush and desert shrub in Argentina and Bolivia, where it forages for food and constructs its nest of twigs among cacti or bushes. Because it has not been much studied, little else is known about the diet, feeding, and breeding behaviors of the brown-backed mockingbird.

Recordings of its call, luckily, are fairly easy to come by, and witnesses to it have described the song of the brown-backed mockingbird as a series of repeated harsh notes and chuckles. Mockingbirds in general are well known for their habit of mimicking the songs of other birds and the sounds of insects and amphibians, often loudly and in rapid succession. One group of researchers has even compared them—

with their ability to create novel song patterns—to the great classical composers and modern artists like Kendrick Lamar.

Though its population size has not been precisely quantified, the brown-backed mockingbird has been categorized by the IUCN as a species of least concern. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on an antique line engraving from Cassell's *Natural History*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.