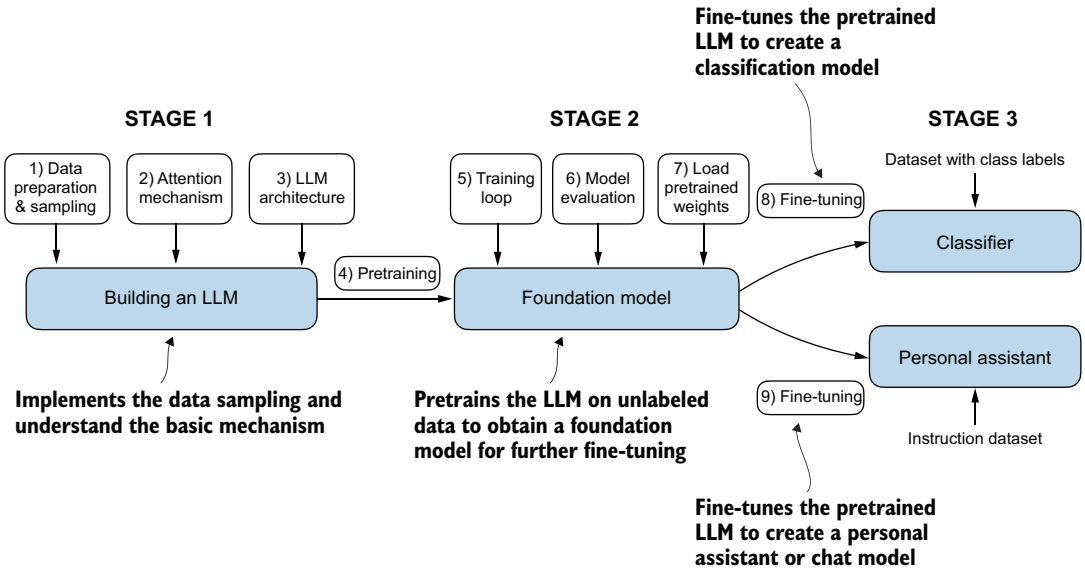
FROM SCRATCH

BUILD A Large Language Model

Sebastian Raschka

MANNING



The three main stages of coding a large language model (LLM) are implementing the LLM architecture and data preparation process (stage 1), pretraining an LLM to create a foundation model (stage 2), and fine-tuning the foundation model to become a personal assistant or text classifier (stage 3). Each of these stages is explored and implemented in this book.

Build a Large Language Model (From Scratch)

Build a Large Language Model (From Scratch)

SEBASTIAN RASCHKA

MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor:	Dustin Archibald
Technical editor:	David Caswell
Review editor:	Kishor Rit
Production editor:	Aleksandar Dragosavljević
Copy editors:	Kari Lucke and Alisa Larson
Proofreader:	Mike Beady
Technical proofreader:	Jerry Kuch
Typesetter:	Dennis Dalinnik
Cover designer:	Marija Tudor

ISBN: 9781633437166
Printed in the United States of America

brief contents

contents

preface

I've always been fascinated with language models. More than a decade ago, my journey into AI began with a statistical pattern classification class, which led to my first independent project: developing a model and web application to detect the mood of a song based on its lyrics.

Fast forward to 2022, with the release of ChatGPT, large language models (LLMs) have taken the world by storm and have revolutionized how many of us work. These models are incredibly versatile, aiding in tasks such as checking grammar, composing emails, summarizing lengthy documents, and much more. This is owed to their ability to parse and generate human-like text, which is important in various fields, from customer service to content creation, and even in more technical domains like coding and data analysis.

As their name implies, LLMs are "large"—very large—encompassing millions to billions of parameters. (For comparison, using more traditional machine learning or statistical methods, the Iris flower dataset can be classified with more than 90% accuracy using a small model with only two parameters.) However, despite the large size of LLMs compared to more traditional methods, LLMs don't have to be a black box.

In this book, you will learn how to build an LLM one step at a time. By the end, you will have a solid understanding of how an LLM, like the ones used in ChatGPT, works on a fundamental level. I believe that developing confidence with each part of the fundamental concepts and underlying code is crucial for success. This not only helps in fixing bugs and improving performance but also enables experimentation with new ideas.

Several years ago, when I started working with LLMs, I had to learn how to implement them the hard way, sifting through many research papers and incomplete code repositories to develop a general understanding. With this book, I hope to make LLMs more accessible by developing and sharing a step-by-step implementation tutorial detailing all the major components and development phases of an LLM.

I strongly believe that the best way to understand LLMs is to code one from scratch—and you'll see that this can be fun too!

Happy reading and coding!

acknowledgments

Writing a book is a significant undertaking, and I would like to express my sincere gratitude to my wife, Liza, for her patience and support throughout this process. Her unconditional love and constant encouragement have been absolutely essential.

I am incredibly grateful to Daniel Kleine, whose invaluable feedback on the in-progress chapters and code went above and beyond. With his keen eye for detail and insightful suggestions, Daniel's contributions have undoubtedly made this book a smoother and more enjoyable reading experience.

I would also like to thank the wonderful staff at Manning Publications, including Michael Stephens, for the many productive discussions that helped shape the direction of this book, and Dustin Archibald, whose constructive feedback and guidance in adhering to the Manning guidelines have been crucial. I also appreciate your flexibility in accommodating the unique requirements of this unconventional from-scratch approach. A special thanks to Aleksandar Dragosavljević, Kari Lucke, and Mike Beady for their work on the professional layouts and to Susan Honeywell and her team for refining and polishing the graphics.

I want to express my heartfelt gratitude to Robin Campbell and her outstanding marketing team for their invaluable support throughout the writing process.

Finally, I extend my thanks to the reviewers: Anandaganesh Balakrishnan, Anto Aravinth, Ayush Bihani, Bassam Ismail, Benjamin Muskalla, Bruno Sonnino, Christian Prokopp, Daniel Kleine, David Curran, Dibyendu Roy Chowdhury, Gary Pass, Georg Sommer, Giovanni Alzetta, Guillermo Alcántara, Jonathan Reeves, Kunal Ghosh, Nicolas Modrzyk, Paul Silisteanu, Raul Ciotescu, Scott Ling, Sriram Macharla, Sumit

Pal, Vahid Mirjalili, Vaijanath Rao, and Walter Reade for their thorough feedback on the drafts. Your keen eyes and insightful comments have been essential in improving the quality of this book.

To everyone who has contributed to this journey, I am sincerely grateful. Your support, expertise, and dedication have been instrumental in bringing this book to fruition. Thank you!

about this book

Build a Large Language Model (From Scratch) was written to help you understand and create your own GPT-like large language models (LLMs) from the ground up. It begins by focusing on the fundamentals of working with text data and coding attention mechanisms and then guides you through implementing a complete GPT model from scratch. The book then covers the pretraining mechanism as well as fine-tuning for specific tasks such as text classification and following instructions. By the end of this book, you'll have a deep understanding of how LLMs work and the skills to build your own models. While the models you'll create are smaller in scale compared to the large foundational models, they use the same concepts and serve as powerful educational tools to grasp the core mechanisms and techniques used in building state-of-the-art LLMs.

Who should read this book

Build a Large Language Model (From Scratch) is for machine learning enthusiasts, engineers, researchers, students, and practitioners who want to gain a deep understanding of how LLMs work and learn to build their own models from scratch. Both beginners and experienced developers will be able to use their existing skills and knowledge to grasp the concepts and techniques used in creating LLMs.

What sets this book apart is its comprehensive coverage of the entire process of building LLMs, from working with datasets to implementing the model architecture, pretraining on unlabeled data, and fine-tuning for specific tasks. As of this writing, no

other resource provides such a complete and hands-on approach to building LLMs from the ground up.

To understand the code examples in this book, you should have a solid grasp of Python programming. While some familiarity with machine learning, deep learning, and artificial intelligence can be beneficial, an extensive background in these areas is not required. LLMs are a unique subset of AI, so even if you're relatively new to the field, you'll be able to follow along.

If you have some experience with deep neural networks, you may find certain concepts more familiar, as LLMs are built upon these architectures. However, proficiency in PyTorch is not a prerequisite. Appendix A provides a concise introduction to PyTorch, equipping you with the necessary skills to comprehend the code examples throughout the book.

A high school–level understanding of mathematics, particularly working with vectors and matrices, can be helpful as we explore the inner workings of LLMs. Advanced mathematical knowledge is not necessary to grasp the key concepts and ideas presented in this book.

The most important prerequisite is a strong foundation in Python programming. With this knowledge, you'll be well prepared to explore the fascinating world of LLMs and understand the concepts and code examples presented in this book.

How this book is organized: A roadmap

This book is designed to be read sequentially, as each chapter builds upon the concepts and techniques introduced in the previous ones. The book is divided into seven chapters that cover the essential aspects of LLMs and their implementation.

Chapter 1 provides a high-level introduction to the fundamental concepts behind LLMs. It explores the transformer architecture, which forms the basis for LLMs such as those used on the ChatGPT platform.

Chapter 2 lays out a plan for building an LLM from scratch. It covers the process of preparing text for LLM training, including splitting text into word and subword tokens, using byte pair encoding for advanced tokenization, sampling training examples with a sliding window approach, and converting tokens into vectors that feed into the LLM.

Chapter 3 focuses on the attention mechanisms used in LLMs. It introduces a basic self-attention framework and progresses to an enhanced self-attention mechanism. The chapter also covers the implementation of a causal attention module that enables LLMs to generate one token at a time, masking randomly selected attention weights with dropout to reduce overfitting and stacking multiple causal attention modules into a multihead attention module.

Chapter 4 focuses on coding a GPT-like LLM that can be trained to generate human-like text. It covers techniques such as normalizing layer activations to stabilize neural network training, adding shortcut connections in deep neural networks to train models more effectively, implementing transformer blocks to create GPT models

of various sizes, and computing the number of parameters and storage requirements of GPT models.

Chapter 5 implements the pretraining process of LLMs. It covers computing the training and validation set losses to assess the quality of LLM-generated text, implementing a training function and pretraining the LLM, saving and loading model weights to continue training an LLM, and loading pretrained weights from OpenAI.

Chapter 6 introduces different LLM fine-tuning approaches. It covers preparing a dataset for text classification, modifying a pretrained LLM for fine-tuning, fine-tuning an LLM to identify spam messages, and evaluating the accuracy of a fine-tuned LLM classifier.

Chapter 7 explores the instruction fine-tuning process of LLMs. It covers preparing a dataset for supervised instruction fine-tuning, organizing instruction data in training batches, loading a pretrained LLM and fine-tuning it to follow human instructions, extracting LLM-generated instruction responses for evaluation, and evaluating an instruction-fine-tuned LLM.

About the code

To make it as easy as possible to follow along, all code examples in this book are conveniently available on the Manning website at https://www.manning.com/books/build-a-large-language-model-from-scratch, as well as in Jupyter notebook format on GitHub at https://github.com/rasbt/LLMs-from-scratch. And don't worry about getting stuck—solutions to all the code exercises can be found in appendix C.

This book contains many examples of source code both in numbered listings and in line with normal text. In both cases, source code is formatted in a `fixed-width font like this` to separate it from ordinary text.

In many cases, the original source code has been reformatted; we've added line breaks and reworked indentation to accommodate the available page space in the book. In rare cases, even this was not enough, and listings include line-continuation markers (➥). Additionally, comments in the source code have often been removed from the listings when the code is described in the text. Code annotations accompany many of the listings, highlighting important concepts.

One of the key goals of this book is accessibility, so the code examples have been carefully designed to run efficiently on a regular laptop, without the need for any special hardware. But if you do have access to a GPU, certain sections provide helpful tips on scaling up the datasets and models to take advantage of that extra power.

Throughout the book, we'll be using PyTorch as our go-to tensor and a deep learning library to implement LLMs from the ground up. If PyTorch is new to you, I recommend you start with appendix A, which provides an in-depth introduction, complete with setup recommendations.

liveBook discussion forum

Purchase of *Build a Large Language Model (From Scratch)* includes free access to liveBook, Manning's online reading platform. Using liveBook's exclusive discussion features, you can attach comments to the book globally or to specific sections or paragraphs. It's a snap to make notes for yourself, ask and answer technical questions, and receive help from the author and other users. To access the forum, go to https://livebook.manning.com/book/build-a-large-language-model-from-scratch/discussion. You can also learn more about Manning's forums and the rules of conduct at https://livebook.manning.com/discussion.

Manning's commitment to readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

Other online resources

Interested in the latest AI and LLM research trends?

- Check out my blog at https://magazine.sebastianraschka.com, where I regularly discusses the latest AI research with a focus on LLMs.

Need help getting up to speed with deep learning and PyTorch?

- I offer several free courses on my website at https://sebastianraschka.com/teaching. These resources can help you quickly get up to speed with the latest techniques.

Looking for bonus materials related to the book?

- Visit the book's GitHub repository at https://github.com/rasbt/LLMs-from-scratch to find additional resources and examples to supplement your learning.

about the author

SEBASTIAN RASCHKA, PhD, has been working in machine learning and AI for more than a decade. In addition to being a researcher, Sebastian has a strong passion for education. He is known for his bestselling books on machine learning with Python and his contributions to open source.

Sebastian is a staff research engineer at Lightning AI, focusing on implementing and training LLMs. Before his industry experience, Sebastian was an assistant professor in the Department of Statistics at the University of Wisconsin-Madison, where he focused on deep learning research. You can learn more about Sebastian at https://sebastianraschka.com.

about the cover illustration

The figure on the cover of *Build a Large Language Model (From Scratch)*, titled "Le duchesse," or "The duchess," is taken from a book by Louis Curmer published in 1841. Each illustration is finely drawn and colored by hand.

In those days, it was easy to identify where people lived and what their trade or station in life was just by their dress. Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional culture centuries ago, brought back to life by pictures from collections such as this one.

1 Understanding large language models

This chapter covers

- High-level explanations of the fundamental concepts behind large language models (LLMs)
- Insights into the transformer architecture from which LLMs are derived
- A plan for building an LLM from scratch

Large language models (LLMs), such as those offered in OpenAI's ChatGPT, are deep neural network models that have been developed over the past few years. They ushered in a new era for natural language processing (NLP). Before the advent of LLMs, traditional methods excelled at categorization tasks such as email spam classification and straightforward pattern recognition that could be captured with handcrafted rules or simpler models. However, they typically underperformed in language tasks that demanded complex understanding and generation abilities, such as parsing detailed instructions, conducting contextual analysis, and creating coherent and contextually appropriate original text. For example, previous generations of language models could not write an email from a list of keywords—a task that is trivial for contemporary LLMs.

LLMs have remarkable capabilities to understand, generate, and interpret human language. However, it's important to clarify that when we say language models "understand," we mean that they can process and generate text in ways that appear coherent and contextually relevant, not that they possess human-like consciousness or comprehension.

Enabled by advancements in deep learning, which is a subset of machine learning and artificial intelligence (AI) focused on neural networks, LLMs are trained on vast quantities of text data. This large-scale training allows LLMs to capture deeper contextual information and subtleties of human language compared to previous approaches. As a result, LLMs have significantly improved performance in a wide range of NLP tasks, including text translation, sentiment analysis, question answering, and many more.

Another important distinction between contemporary LLMs and earlier NLP models is that earlier NLP models were typically designed for specific tasks, such as text categorization, language translation, etc. While those earlier NLP models excelled in their narrow applications, LLMs demonstrate a broader proficiency across a wide range of NLP tasks.

The success of LLMs can be attributed to the transformer architecture that underpins many LLMs and the vast amounts of data on which LLMs are trained, allowing them to capture a wide variety of linguistic nuances, contexts, and patterns that would be challenging to encode manually.

This shift toward implementing models based on the transformer architecture and using large training datasets to train LLMs has fundamentally transformed NLP, providing more capable tools for understanding and interacting with human language.

The following discussion sets a foundation to accomplish the primary objective of this book: understanding LLMs by implementing a ChatGPT-like LLM based on the transformer architecture, step by step in code.

1.1 What is an LLM?

An LLM is a neural network designed to understand, generate, and respond to human-like text. These models are deep neural networks trained on massive amounts of text data, sometimes encompassing large portions of the entire publicly available text on the internet.

The "large" in "large language model" refers to both the model's size in terms of parameters and the immense dataset on which it's trained. Models like this often have tens or even hundreds of billions of parameters, which are the adjustable weights in the network that are optimized during training to predict the next word in a sequence. Next-word prediction is sensible because it harnesses the inherent sequential nature of language to train models on understanding context, structure, and relationships within text. It is a very simple task, and so it is surprising to many researchers that it can produce such capable models. In later chapters, we will discuss and implement the next-word training procedure step by step.

LLMs utilize an architecture called the *transformer*, which allows them to pay selective attention to different parts of the input when making predictions, making them especially adept at handling the nuances and complexities of human language.

Since LLMs are capable of *generating* text, they are also often referred to as a form of generative artificial intelligence, often abbreviated as *generative AI* or *GenAI*. As illustrated in figure 1.1, AI encompasses the broader field of creating machines that can perform tasks requiring human-like intelligence, including understanding language, recognizing patterns, and making decisions, and includes subfields like machine learning and deep learning.

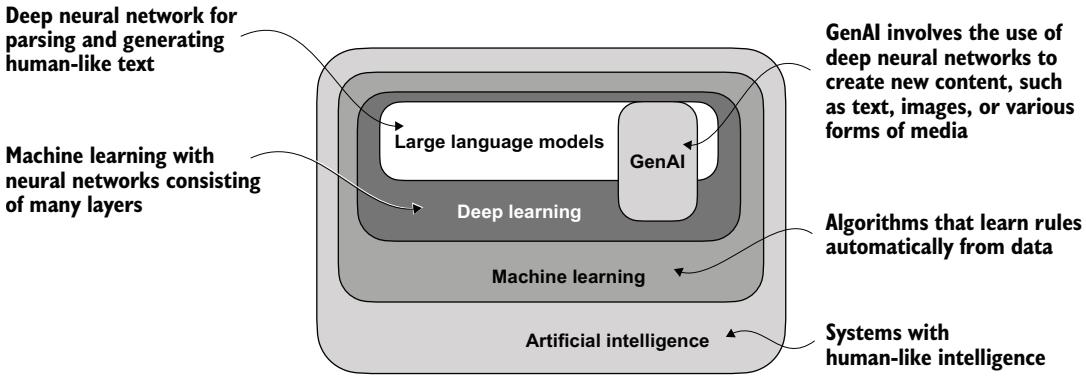


Figure 1.1 As this hierarchical depiction of the relationship between the different fields suggests, LLMs represent a specific application of deep learning techniques, using their ability to process and generate human-like text. Deep learning is a specialized branch of machine learning that focuses on using multilayer neural networks. Machine learning and deep learning are fields aimed at implementing algorithms that enable computers to learn from data and perform tasks that typically require human intelligence.

The algorithms used to implement AI are the focus of the field of machine learning. Specifically, machine learning involves the development of algorithms that can learn from and make predictions or decisions based on data without being explicitly programmed. To illustrate this, imagine a spam filter as a practical application of machine learning. Instead of manually writing rules to identify spam emails, a machine learning algorithm is fed examples of emails labeled as spam and legitimate emails. By minimizing the error in its predictions on a training dataset, the model then learns to recognize patterns and characteristics indicative of spam, enabling it to classify new emails as either spam or not spam.

As illustrated in figure 1.1, deep learning is a subset of machine learning that focuses on utilizing neural networks with three or more layers (also called deep neural networks) to model complex patterns and abstractions in data. In contrast to deep learning, traditional machine learning requires manual feature extraction. This means that human experts need to identify and select the most relevant features for the model.

While the field of AI is now dominated by machine learning and deep learning, it also includes other approaches—for example, using rule-based systems, genetic algorithms, expert systems, fuzzy logic, or symbolic reasoning.

Returning to the spam classification example, in traditional machine learning, human experts might manually extract features from email text such as the frequency of certain trigger words (for example, "prize," "win," "free"), the number of exclamation marks, use of all uppercase words, or the presence of suspicious links. This dataset, created based on these expert-defined features, would then be used to train the model. In contrast to traditional machine learning, deep learning does not require manual feature extraction. This means that human experts do not need to identify and select the most relevant features for a deep learning model. (However, both traditional machine learning and deep learning for spam classification still require the collection of labels, such as spam or non-spam, which need to be gathered either by an expert or users.)

Let's look at some of the problems LLMs can solve today, the challenges that LLMs address, and the general LLM architecture we will implement later.

1.2 Applications of LLMs

Owing to their advanced capabilities to parse and understand unstructured text data, LLMs have a broad range of applications across various domains. Today, LLMs are employed for machine translation, generation of novel texts (see figure 1.2), sentiment analysis, text summarization, and many other tasks. LLMs have recently been used for content creation, such as writing fiction, articles, and even computer code.

LLMs can also power sophisticated chatbots and virtual assistants, such as OpenAI's ChatGPT or Google's Gemini (formerly called Bard), which can answer user queries and augment traditional search engines such as Google Search or Microsoft Bing.

Moreover, LLMs may be used for effective knowledge retrieval from vast volumes of text in specialized areas such as medicine or law. This includes sifting through documents, summarizing lengthy passages, and answering technical questions.

In short, LLMs are invaluable for automating almost any task that involves parsing and generating text. Their applications are virtually endless, and as we continue to innovate and explore new ways to use these models, it's clear that LLMs have the potential to redefine our relationship with technology, making it more conversational, intuitive, and accessible.

We will focus on understanding how LLMs work from the ground up, coding an LLM that can generate text. You will also learn about techniques that allow LLMs to carry out queries, ranging from answering questions to summarizing text, translating text into different languages, and more. In other words, you will learn how complex LLM assistants such as ChatGPT work by building one step by step.

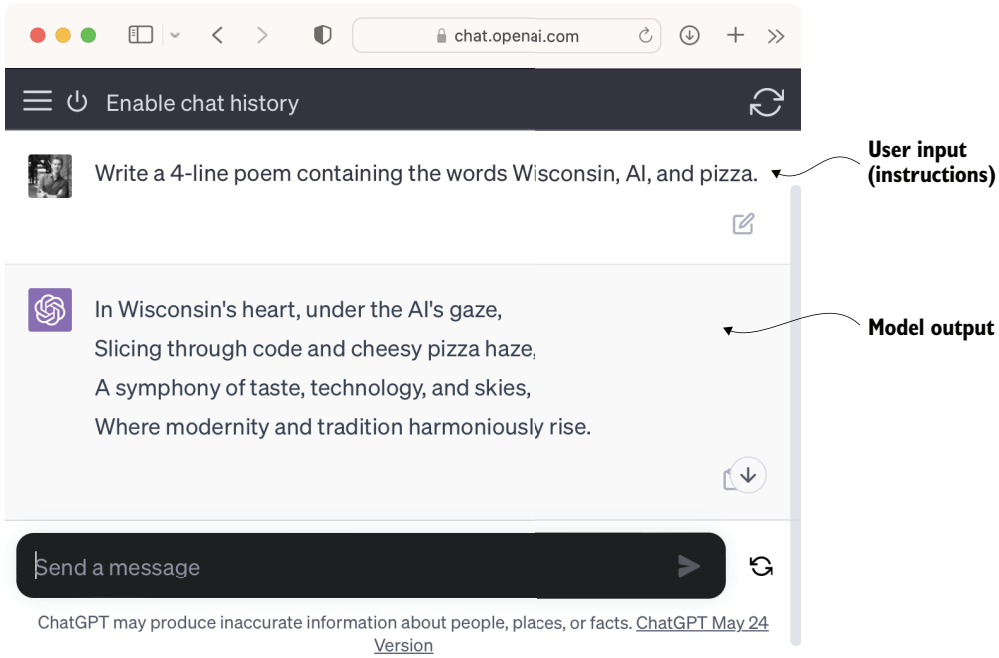


Figure 1.2 LLM interfaces enable natural language communication between users and AI systems. This screenshot shows ChatGPT writing a poem according to a user's specifications.

1.3 Stages of building and using LLMs

Why should you build your own LLM? Coding an LLM from the ground up is an excellent exercise to understand its mechanics and limitations. Also, it equips us with the required knowledge for pretraining or fine-tuning existing open source LLM architectures to our own domain-specific datasets or tasks.

> **NOTE** Most LLMs today are implemented using the PyTorch deep learning library, which is what we will use. Readers can find a comprehensive introduction to PyTorch in appendix A.

Research has shown that when it comes to modeling performance, custom-built LLMs—those tailored for specific tasks or domains—can outperform general-purpose LLMs, such as ChatGPT, which are designed for a wide array of applications. Examples of these include BloombergGPT (specialized for finance) and LLMs tailored for medical question answering (see appendix B for details).

Using custom-built LLMs offers several advantages, particularly regarding data privacy. For instance, companies may prefer not to share sensitive data with third-party LLM providers like OpenAI due to confidentiality concerns. Additionally, developing smaller, custom LLMs enables deployment directly on customer devices, such as laptops and smartphones, which is something companies like Apple are currently exploring.

This local implementation can significantly decrease latency and reduce server-related costs. Furthermore, custom LLMs grant developers complete autonomy, allowing them to control updates and modifications to the model as needed.

The general process of creating an LLM includes pretraining and fine-tuning. The "pre" in "pretraining" refers to the initial phase where a model like an LLM is trained on a large, diverse dataset to develop a broad understanding of language. This pretrained model then serves as a foundational resource that can be further refined through fine-tuning, a process where the model is specifically trained on a narrower dataset that is more specific to particular tasks or domains. This two-stage training approach consisting of pretraining and fine-tuning is depicted in figure 1.3.

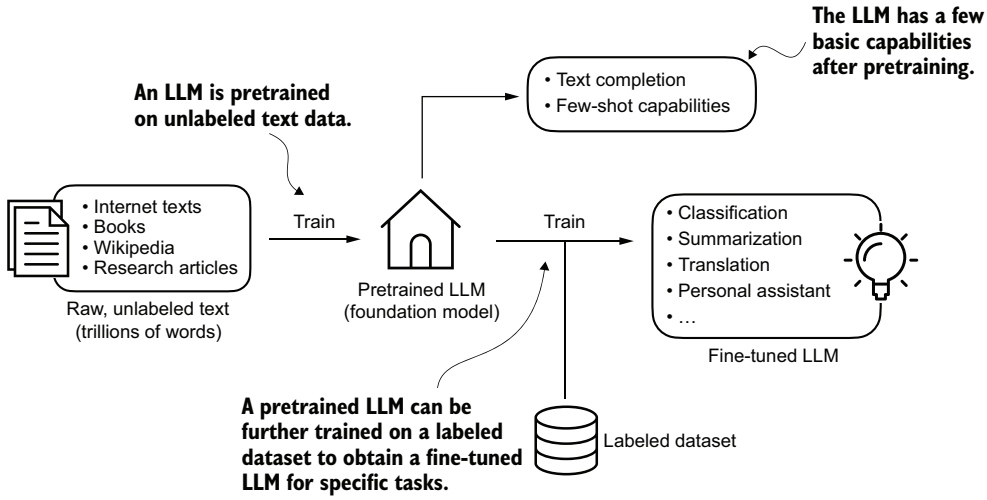


Figure 1.3 Pretraining an LLM involves next-word prediction on large text datasets. A pretrained LLM can then be fine-tuned using a smaller labeled dataset.

The first step in creating an LLM is to train it on a large corpus of text data, sometimes referred to as *raw* text. Here, "raw" refers to the fact that this data is just regular text without any labeling information. (Filtering may be applied, such as removing formatting characters or documents in unknown languages.)

> **NOTE** Readers with a background in machine learning may note that labeling information is typically required for traditional machine learning models and deep neural networks trained via the conventional supervised learning paradigm. This is not the case for the pretraining stage of LLMs. In this phase, LLMs use self-supervised learning, where the model generates its own labels from the input data.

This first training stage of an LLM is also known as *pretraining*, creating an initial pretrained LLM, often called a *base* or *foundation model*. A typical example of such a model is the GPT-3 model (the precursor of the original model offered in ChatGPT). This model is capable of text completion—that is, finishing a half-written sentence provided by a user. It also has limited few-shot capabilities, which means it can learn to perform new tasks based on only a few examples instead of needing extensive training data.

After obtaining a pretrained LLM by training on large text datasets, where the LLM is trained to predict the next word in the text, we can further train the LLM on labeled data, also known as *fine-tuning*.

The two most popular categories of fine-tuning LLMs are *instruction fine-tuning* and *classification fine-tuning*. In instruction fine-tuning, the labeled dataset consists of instruction and answer pairs, such as a query to translate a text accompanied by the correctly translated text. In classification fine-tuning, the labeled dataset consists of texts and associated class labels—for example, emails associated with "spam" and "not spam" labels.

We will cover code implementations for pretraining and fine-tuning an LLM, and we will delve deeper into the specifics of both instruction and classification fine-tuning after pretraining a base LLM.

1.4 Introducing the transformer architecture

Most modern LLMs rely on the *transformer* architecture, which is a deep neural network architecture introduced in the 2017 paper "Attention Is All You Need" (https://arxiv.org/abs/1706.03762). To understand LLMs, we must understand the original transformer, which was developed for machine translation, translating English texts to German and French. A simplified version of the transformer architecture is depicted in figure 1.4.

The transformer architecture consists of two submodules: an encoder and a decoder. The encoder module processes the input text and encodes it into a series of numerical representations or vectors that capture the contextual information of the input. Then, the decoder module takes these encoded vectors and generates the output text. In a translation task, for example, the encoder would encode the text from the source language into vectors, and the decoder would decode these vectors to generate text in the target language. Both the encoder and decoder consist of many layers connected by a so-called self-attention mechanism. You may have many questions regarding how the inputs are preprocessed and encoded. These will be addressed in a step-by-step implementation in subsequent chapters.

A key component of transformers and LLMs is the self-attention mechanism (not shown), which allows the model to weigh the importance of different words or tokens in a sequence relative to each other. This mechanism enables the model to capture long-range dependencies and contextual relationships within the input data, enhancing its ability to generate coherent and contextually relevant output. However, due to

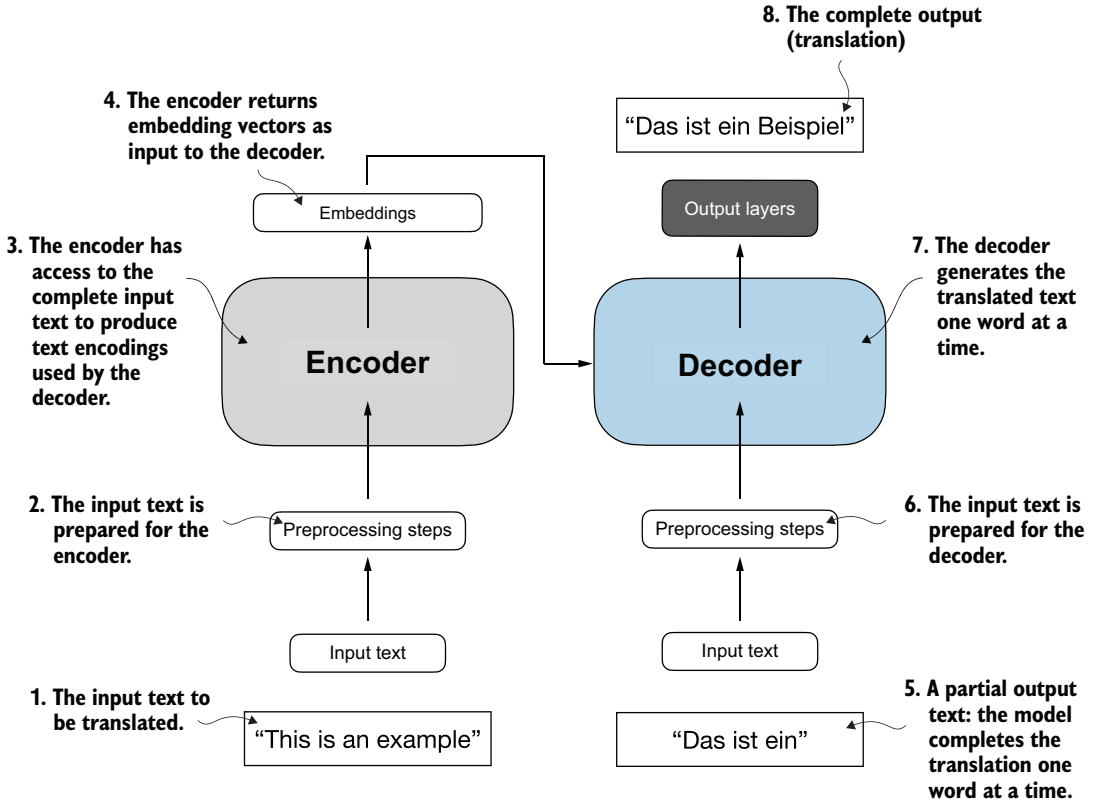


Figure 1.4 A simplified depiction of the original transformer architecture, which is a deep learning model for language translation. The transformer consists of two parts: (a) an encoder that processes the input text and produces an embedding representation (a numerical representation that captures many different factors in different dimensions) of the text that the (b) decoder can use to generate the translated text one word at a time. This figure shows the final stage of the translation process where the decoder has to generate only the final word ("Beispiel"), given the original input text ("This is an example") and a partially translated sentence ("Das ist ein"), to complete the translation.

its complexity, we will defer further explanation to chapter 3, where we will discuss and implement it step by step.

Later variants of the transformer architecture, such as BERT (short for *bidirectional encoder representations from transformers*) and the various GPT models (short for *generative pretrained transformers*), built on this concept to adapt this architecture for different tasks. If interested, refer to appendix B for further reading suggestions.

BERT, which is built upon the original transformer's encoder submodule, differs in its training approach from GPT. While GPT is designed for generative tasks, BERT and its variants specialize in masked word prediction, where the model predicts masked

or hidden words in a given sentence, as shown in figure 1.5. This unique training strategy equips BERT with strengths in text classification tasks, including sentiment prediction and document categorization. As an application of its capabilities, as of this writing, X (formerly Twitter) uses BERT to detect toxic content.

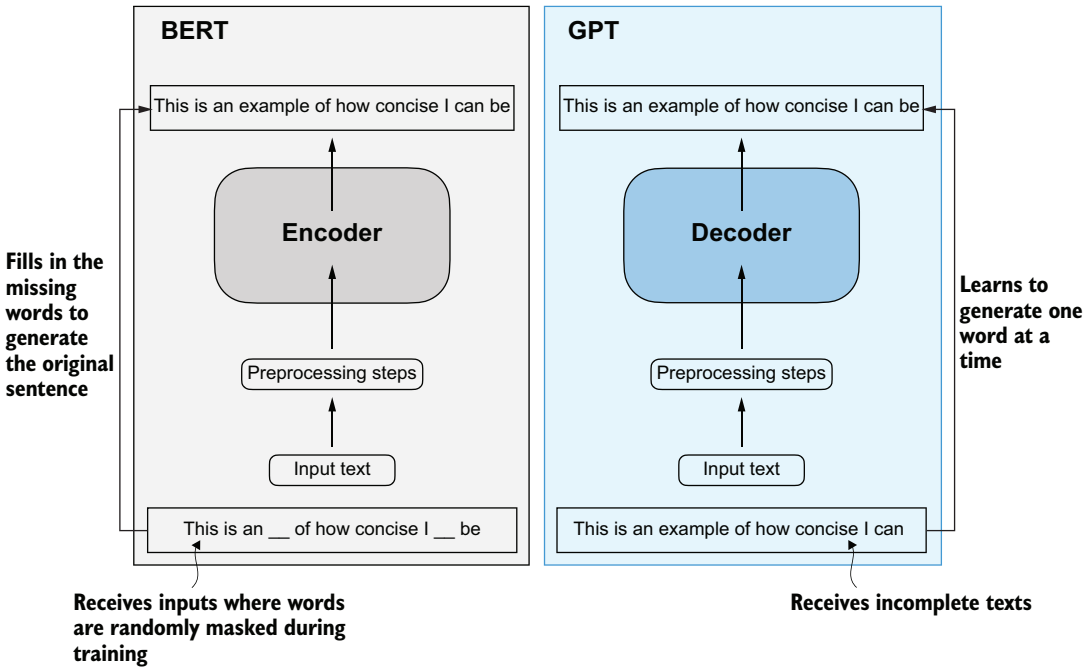


Figure 1.5 A visual representation of the transformer's encoder and decoder submodules. On the left, the encoder segment exemplifies BERT-like LLMs, which focus on masked word prediction and are primarily used for tasks like text classification. On the right, the decoder segment showcases GPT-like LLMs, designed for generative tasks and producing coherent text sequences.

GPT, on the other hand, focuses on the decoder portion of the original transformer architecture and is designed for tasks that require generating texts. This includes machine translation, text summarization, fiction writing, writing computer code, and more.

GPT models, primarily designed and trained to perform text completion tasks, also show remarkable versatility in their capabilities. These models are adept at executing both zero-shot and few-shot learning tasks. Zero-shot learning refers to the ability to generalize to completely unseen tasks without any prior specific examples. On the other hand, few-shot learning involves learning from a minimal number of examples the user provides as input, as shown in figure 1.6.

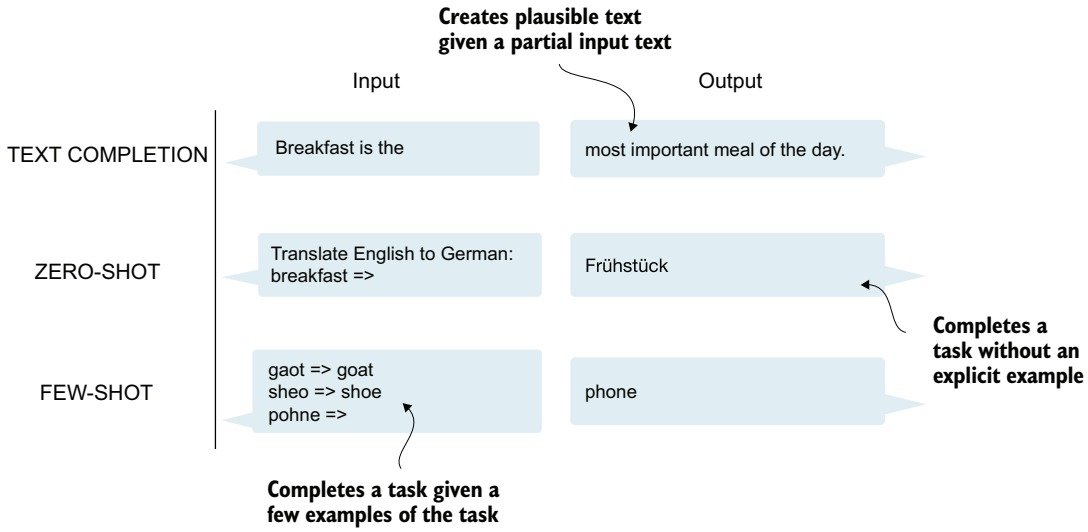


Figure 1.6 In addition to text completion, GPT-like LLMs can solve various tasks based on their inputs without needing retraining, fine-tuning, or task-specific model architecture changes. Sometimes it is helpful to provide examples of the target within the input, which is known as a few-shot setting. However, GPT-like LLMs are also capable of carrying out tasks without a specific example, which is called zero-shot setting.

Transformers vs. LLMs

Today's LLMs are based on the transformer architecture. Hence, transformers and LLMs are terms that are often used synonymously in the literature. However, note that not all transformers are LLMs since transformers can also be used for computer vision. Also, not all LLMs are transformers, as there are LLMs based on recurrent and convolutional architectures. The main motivation behind these alternative approaches is to improve the computational efficiency of LLMs. Whether these alternative LLM architectures can compete with the capabilities of transformer-based LLMs and whether they are going to be adopted in practice remains to be seen. For simplicity, I use the term "LLM" to refer to transformer-based LLMs similar to GPT. (Interested readers can find literature references describing these architectures in appendix B.)

1.5 Utilizing large datasets

The large training datasets for popular GPT- and BERT-like models represent diverse and comprehensive text corpora encompassing billions of words, which include a vast array of topics and natural and computer languages. To provide a concrete example, table 1.1 summarizes the dataset used for pretraining GPT-3, which served as the base model for the first version of ChatGPT.

Table 1.1 The pretraining dataset of the popular GPT-3 LLM

Dataset name	Dataset description	Number of tokens	Proportion in training data
CommonCrawl (filtered)	Web crawl data	410 billion	60%
WebText2	Web crawl data	19 billion	22%
Books1	Internet-based book corpus	12 billion	8%
Books2	Internet-based book corpus	55 billion	8%
Wikipedia	High-quality text	3 billion	3%

Table 1.1 reports the number of tokens, where a token is a unit of text that a model reads and the number of tokens in a dataset is roughly equivalent to the number of words and punctuation characters in the text. Chapter 2 addresses tokenization, the process of converting text into tokens.

The main takeaway is that the scale and diversity of this training dataset allow these models to perform well on diverse tasks, including language syntax, semantics, and context—even some requiring general knowledge.

GPT-3 dataset details

Table 1.1 displays the dataset used for GPT-3. The proportions column in the table sums up to 100% of the sampled data, adjusted for rounding errors. Although the subsets in the Number of Tokens column total 499 billion, the model was trained on only 300 billion tokens. The authors of the GPT-3 paper did not specify why the model was not trained on all 499 billion tokens.

For context, consider the size of the CommonCrawl dataset, which alone consists of 410 billion tokens and requires about 570 GB of storage. In comparison, later iterations of models like GPT-3, such as Meta's LLaMA, have expanded their training scope to include additional data sources like Arxiv research papers (92 GB) and StackExchange's code-related Q&As (78 GB).

The authors of the GPT-3 paper did not share the training dataset, but a comparable dataset that is publicly available is *Dolma: An Open Corpus of Three Trillion Tokens for LLM Pretraining Research* by Soldaini et al. 2024 (https://arxiv.org/abs/2402.00159). However, the collection may contain copyrighted works, and the exact usage terms may depend on the intended use case and country.

The pretrained nature of these models makes them incredibly versatile for further fine-tuning on downstream tasks, which is why they are also known as base or foundation models. Pretraining LLMs requires access to significant resources and is very expensive. For example, the GPT-3 pretraining cost is estimated to be $4.6 million in terms of cloud computing credits (https://mng.bz/VxEW).

The good news is that many pretrained LLMs, available as open source models, can be used as general-purpose tools to write, extract, and edit texts that were not part of the training data. Also, LLMs can be fine-tuned on specific tasks with relatively smaller datasets, reducing the computational resources needed and improving performance.

We will implement the code for pretraining and use it to pretrain an LLM for educational purposes. All computations are executable on consumer hardware. After implementing the pretraining code, we will learn how to reuse openly available model weights and load them into the architecture we will implement, allowing us to skip the expensive pretraining stage when we fine-tune our LLM.

1.6 A closer look at the GPT architecture

GPT was originally introduced in the paper "Improving Language Understanding by Generative Pre-Training" (https://mng.bz/x2qg) by Radford et al. from OpenAI. GPT-3 is a scaled-up version of this model that has more parameters and was trained on a larger dataset. In addition, the original model offered in ChatGPT was created by fine-tuning GPT-3 on a large instruction dataset using a method from OpenAI's InstructGPT paper (https://arxiv.org/abs/2203.02155). As figure 1.6 shows, these models are competent text completion models and can carry out other tasks such as spelling correction, classification, or language translation. This is actually very remarkable given that GPT models are pretrained on a relatively simple next-word prediction task, as depicted in figure 1.7.

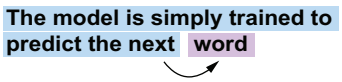


Figure 1.7 In the next-word prediction pretraining task for GPT models, the system learns to predict the upcoming word in a sentence by looking at the words that have come before it. This approach helps the model understand how words and phrases typically fit together in language, forming a foundation that can be applied to various other tasks.

The next-word prediction task is a form of self-supervised learning, which is a form of self-labeling. This means that we don't need to collect labels for the training data explicitly but can use the structure of the data itself: we can use the next word in a sentence or document as the label that the model is supposed to predict. Since this next-word prediction task allows us to create labels "on the fly," it is possible to use massive unlabeled text datasets to train LLMs.

Compared to the original transformer architecture we covered in section 1.4, the general GPT architecture is relatively simple. Essentially, it's just the decoder part without the encoder (figure 1.8). Since decoder-style models like GPT generate text by predicting text one word at a time, they are considered a type of *autoregressive* model. Autoregressive models incorporate their previous outputs as inputs for future

predictions. Consequently, in GPT, each new word is chosen based on the sequence that precedes it, which improves the coherence of the resulting text.

Architectures such as GPT-3 are also significantly larger than the original transformer model. For instance, the original transformer repeated the encoder and decoder blocks six times. GPT-3 has 96 transformer layers and 175 billion parameters in total.

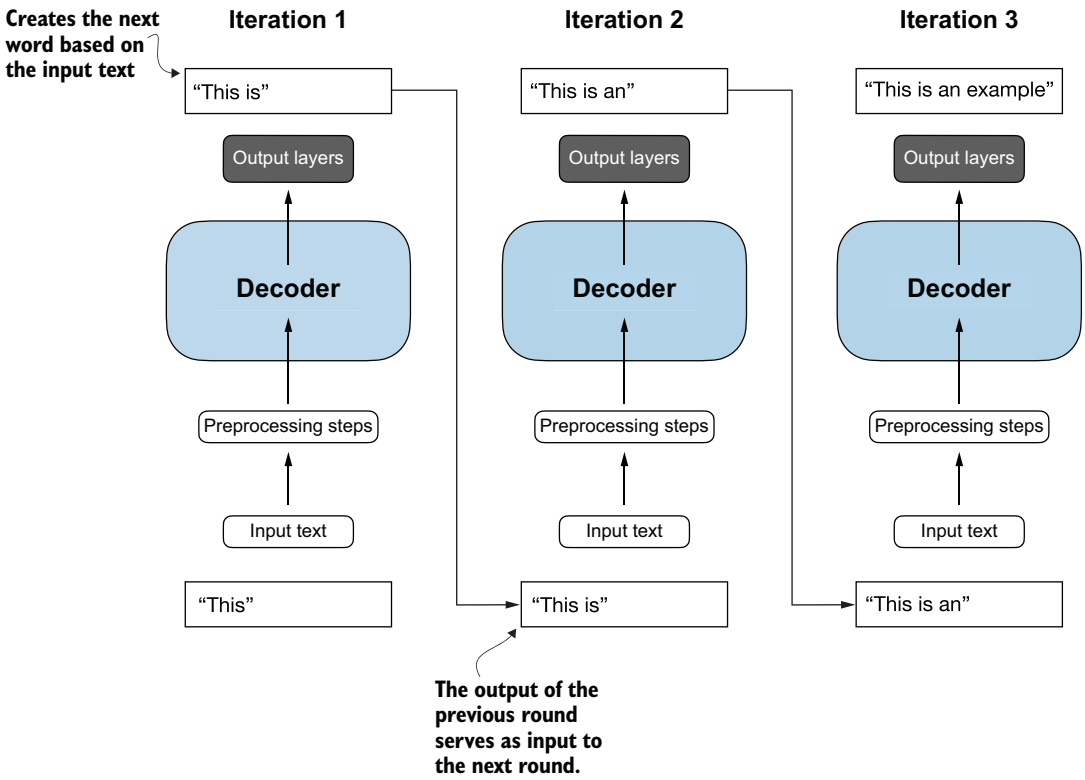


Figure 1.8 The GPT architecture employs only the decoder portion of the original transformer. It is designed for unidirectional, left-to-right processing, making it well suited for text generation and next-word prediction tasks to generate text in an iterative fashion, one word at a time.

GPT-3 was introduced in 2020, which, by the standards of deep learning and large language model development, is considered a long time ago. However, more recent architectures, such as Meta's Llama models, are still based on the same underlying concepts, introducing only minor modifications. Understanding GPT remains highly relevant. I focus on implementing the prominent architecture behind GPT and provide pointers to specific tweaks used by alternative LLMs.

Although the original transformer model, consisting of encoder and decoder blocks, was explicitly designed for language translation, GPT models—despite their larger yet

simpler decoder-only architecture aimed at next-word prediction—are also capable of performing translation tasks. This capability was initially unexpected to researchers, as it emerged from a model primarily trained on a next-word prediction task, which is a task that did not specifically target translation.

The ability to perform tasks that the model wasn't explicitly trained to perform is called an *emergent behavior*. This capability isn't explicitly taught during training but emerges as a natural consequence of the model's exposure to vast quantities of multilingual data in diverse contexts. The fact that GPT models can "learn" the translation patterns between languages and perform translation tasks even though they weren't specifically trained for it demonstrates the benefits and capabilities of these large-scale, generative language models. We can perform diverse tasks without using diverse models for each.

1.7 Building a large language model

Now that we've laid the groundwork for understanding LLMs, let's code one from scratch. We will take the fundamental idea behind GPT as a blueprint and tackle this in three stages, as outlined in figure 1.9.

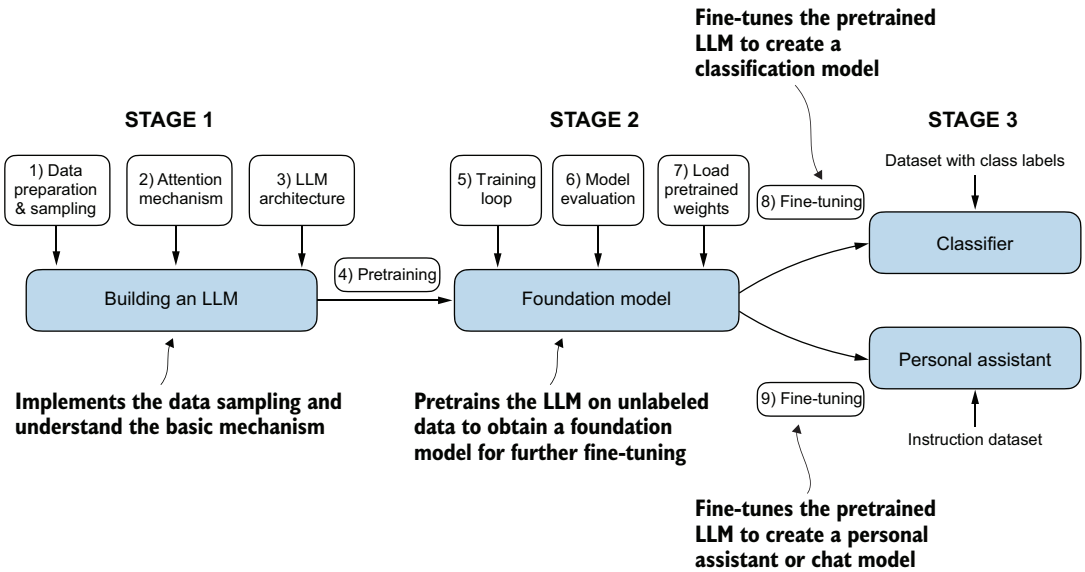


Figure 1.9 The three main stages of coding an LLM are implementing the LLM architecture and data preparation process (stage 1), pretraining an LLM to create a foundation model (stage 2), and fine-tuning the foundation model to become a personal assistant or text classifier (stage 3).

In stage 1, we will learn about the fundamental data preprocessing steps and code the attention mechanism at the heart of every LLM. Next, in stage 2, we will learn how to code and pretrain a GPT-like LLM capable of generating new texts. We will also go over the fundamentals of evaluating LLMs, which is essential for developing capable NLP systems.

Pretraining an LLM from scratch is a significant endeavor, demanding thousands to millions of dollars in computing costs for GPT-like models. Therefore, the focus of stage 2 is on implementing training for educational purposes using a small dataset. In addition, I also provide code examples for loading openly available model weights.

Finally, in stage 3, we will take a pretrained LLM and fine-tune it to follow instructions such as answering queries or classifying texts—the most common tasks in many real-world applications and research.

I hope you are looking forward to embarking on this exciting journey!

Summary

- LLMs have transformed the field of natural language processing, which previously mostly relied on explicit rule-based systems and simpler statistical methods. The advent of LLMs introduced new deep learning-driven approaches that led to advancements in understanding, generating, and translating human language.
- Modern LLMs are trained in two main steps:
  - First, they are pretrained on a large corpus of unlabeled text by using the prediction of the next word in a sentence as a label.
  - Then, they are fine-tuned on a smaller, labeled target dataset to follow instructions or perform classification tasks.
- LLMs are based on the transformer architecture. The key idea of the transformer architecture is an attention mechanism that gives the LLM selective access to the whole input sequence when generating the output one word at a time.
- The original transformer architecture consists of an encoder for parsing text and a decoder for generating text.
- LLMs for generating text and following instructions, such as GPT-3 and ChatGPT, only implement decoder modules, simplifying the architecture.
- Large datasets consisting of billions of words are essential for pretraining LLMs.
- While the general pretraining task for GPT-like models is to predict the next word in a sentence, these LLMs exhibit emergent properties, such as capabilities to classify, translate, or summarize texts.

- Once an LLM is pretrained, the resulting foundation model can be fine-tuned more efficiently for various downstream tasks.
- LLMs fine-tuned on custom datasets can outperform general LLMs on specific tasks.

2 Working with text data

This chapter covers

- Preparing text for large language model training
- Splitting text into word and subword tokens
- Byte pair encoding as a more advanced way of tokenizing text
- Sampling training examples with a sliding window approach
- Converting tokens into vectors that feed into a large language model

So far, we've covered the general structure of large language models (LLMs) and learned that they are pretrained on vast amounts of text. Specifically, our focus was on decoder-only LLMs based on the transformer architecture, which underlies the models used in ChatGPT and other popular GPT-like LLMs.

During the pretraining stage, LLMs process text, one word at a time. Training LLMs with millions to billions of parameters using a next-word prediction task yields models with impressive capabilities. These models can then be further fine-tuned to follow general instructions or perform specific target tasks. But before we can implement and train LLMs, we need to prepare the training dataset, as illustrated in figure 2.1.

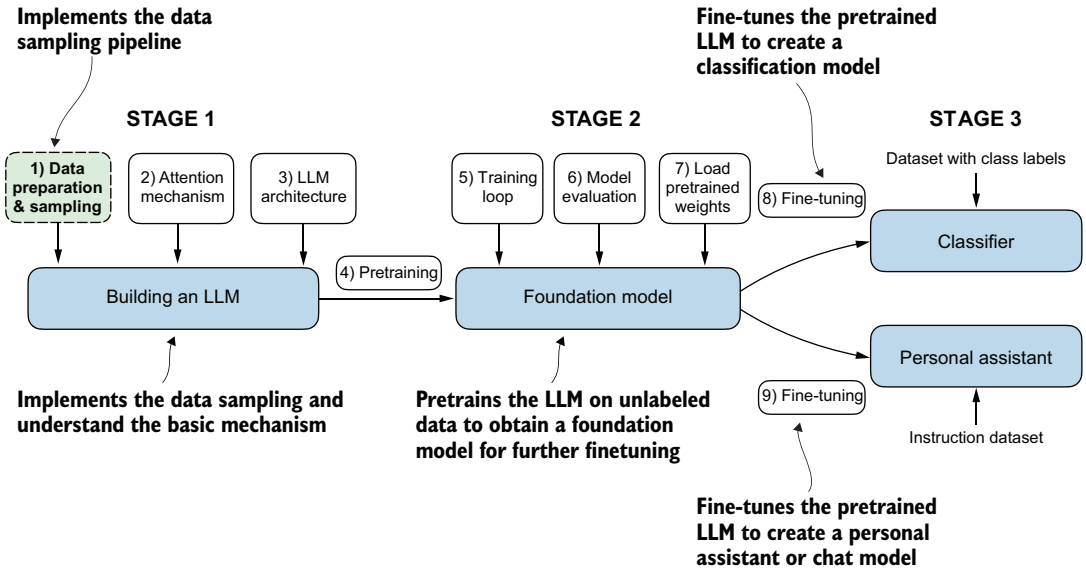


Figure 2.1 The three main stages of coding an LLM. This chapter focuses on step 1 of stage 1: implementing the data sample pipeline.

You'll learn how to prepare input text for training LLMs. This involves splitting text into individual word and subword tokens, which can then be encoded into vector representations for the LLM. You'll also learn about advanced tokenization schemes like byte pair encoding, which is utilized in popular LLMs like GPT. Lastly, we'll implement a sampling and data-loading strategy to produce the input-output pairs necessary for training LLMs.

2.1 Understanding word embeddings

Deep neural network models, including LLMs, cannot process raw text directly. Since text is categorical, it isn't compatible with the mathematical operations used to implement and train neural networks. Therefore, we need a way to represent words as continuous-valued vectors.

> **NOTE** Readers unfamiliar with vectors and tensors in a computational context can learn more in appendix A, section A.2.2.

The concept of converting data into a vector format is often referred to as *embedding*. Using a specific neural network layer or another pretrained neural network model, we can embed different data types—for example, video, audio, and text, as illustrated in figure 2.2. However, it's important to note that different data formats require distinct embedding models. For example, an embedding model designed for text would not be suitable for embedding audio or video data.

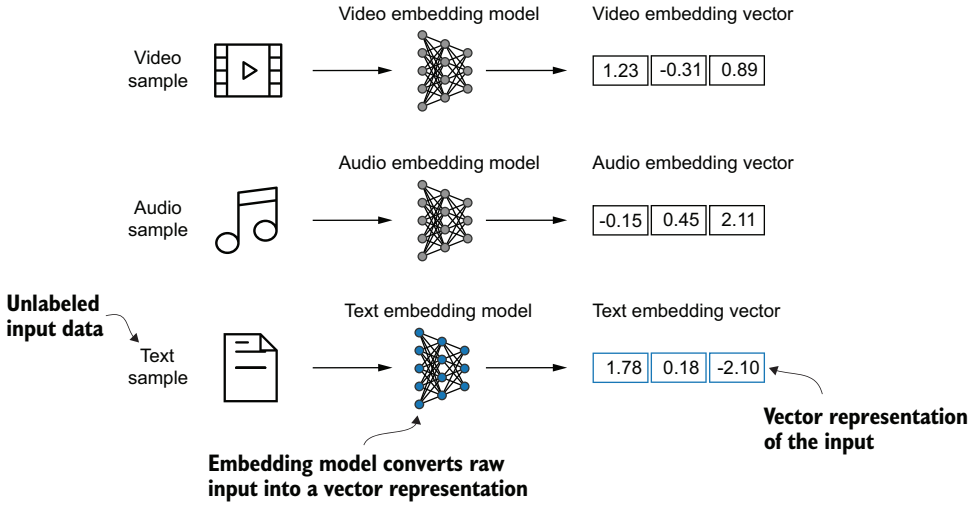


Figure 2.2 Deep learning models cannot process data formats like video, audio, and text in their raw form. Thus, we use an embedding model to transform this raw data into a dense vector representation that deep learning architectures can easily understand and process. Specifically, this figure illustrates the process of converting raw data into a three-dimensional numerical vector.

At its core, an embedding is a mapping from discrete objects, such as words, images, or even entire documents, to points in a continuous vector space—the primary purpose of embeddings is to convert nonnumeric data into a format that neural networks can process.

While word embeddings are the most common form of text embedding, there are also embeddings for sentences, paragraphs, or whole documents. Sentence or paragraph embeddings are popular choices for *retrieval-augmented generation*. Retrieval-augmented generation combines generation (like producing text) with retrieval (like searching an external knowledge base) to pull relevant information when generating text, which is a technique that is beyond the scope of this book. Since our goal is to train GPT-like LLMs, which learn to generate text one word at a time, we will focus on word embeddings.

Several algorithms and frameworks have been developed to generate word embeddings. One of the earlier and most popular examples is the *Word2Vec* approach. Word2Vec trained neural network architecture to generate word embeddings by predicting the context of a word given the target word or vice versa. The main idea behind Word2Vec is that words that appear in similar contexts tend to have similar meanings. Consequently, when projected into two-dimensional word embeddings for visualization purposes, similar terms are clustered together, as shown in figure 2.3.

Word embeddings can have varying dimensions, from one to thousands. A higher dimensionality might capture more nuanced relationships but at the cost of computational efficiency.

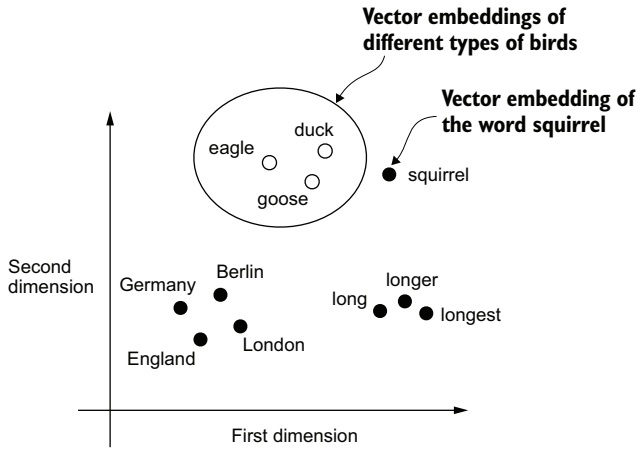


Figure 2.3 If word embeddings are two-dimensional, we can plot them in a two-dimensional scatterplot for visualization purposes as shown here. When using word embedding techniques, such as Word2Vec, words corresponding to similar concepts often appear close to each other in the embedding space. For instance, different types of birds appear closer to each other in the embedding space than in countries and cities.

While we can use pretrained models such as Word2Vec to generate embeddings for machine learning models, LLMs commonly produce their own embeddings that are part of the input layer and are updated during training. The advantage of optimizing the embeddings as part of the LLM training instead of using Word2Vec is that the embeddings are optimized to the specific task and data at hand. We will implement such embedding layers later in this chapter. (LLMs can also create contextualized output embeddings, as we discuss in chapter 3.)

Unfortunately, high-dimensional embeddings present a challenge for visualization because our sensory perception and common graphical representations are inherently limited to three dimensions or fewer, which is why figure 2.3 shows two-dimensional embeddings in a two-dimensional scatterplot. However, when working with LLMs, we typically use embeddings with a much higher dimensionality. For both GPT-2 and GPT-3, the embedding size (often referred to as the dimensionality of the model's hidden states) varies based on the specific model variant and size. It is a tradeoff between performance and efficiency. The smallest GPT-2 models (117M and 125M parameters) use an embedding size of 768 dimensions to provide concrete examples. The largest GPT-3 model (175B parameters) uses an embedding size of 12,288 dimensions.

Next, we will walk through the required steps for preparing the embeddings used by an LLM, which include splitting text into words, converting words into tokens, and turning tokens into embedding vectors.

2.2 Tokenizing text

Let's discuss how we split input text into individual tokens, a required preprocessing step for creating embeddings for an LLM. These tokens are either individual words or special characters, including punctuation characters, as shown in figure 2.4.

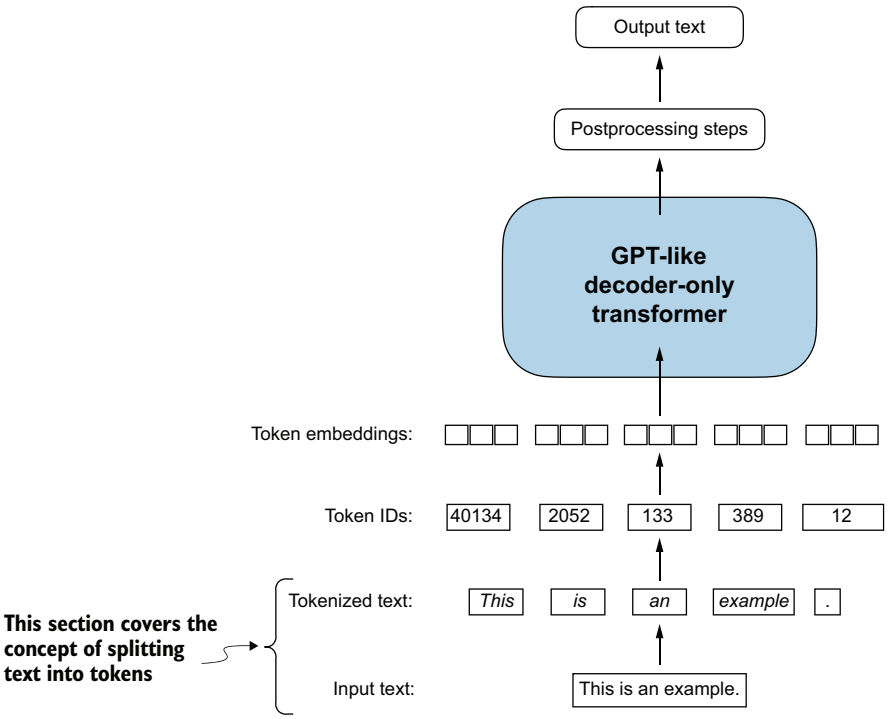


Figure 2.4 A view of the text processing steps in the context of an LLM. Here, we split an input text into individual tokens, which are either words or special characters, such as punctuation characters.

The text we will tokenize for LLM training is "The Verdict," a short story by Edith Wharton, which has been released into the public domain and is thus permitted to be used for LLM training tasks. The text is available on Wikisource at https://en.wikisource.org/wiki/The_Verdict, and you can copy and paste it into a text file, which I copied into a text file `"the-verdict.txt"`.

Alternatively, you can find this `"the-verdict.txt"` file in this book's GitHub repository at https://mng.bz/Adng. You can download the file with the following Python code:

```
import urllib.request
url = ("https://raw.githubusercontent.com/rasbt/"
       "LLMs-from-scratch/main/ch02/01_main-chapter-code/"
       "the-verdict.txt")
file_path = "the-verdict.txt"
urllib.request.urlretrieve(url, file_path)
```

Next, we can load the `the-verdict.txt` file using Python's standard file reading utilities.

Listing 2.1 Reading in a short story as text sample into Python

```
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()
print("Total number of character:", len(raw_text))
print(raw_text[:99])
```

The print command prints the total number of characters followed by the first 99 characters of this file for illustration purposes:

```
Total number of character: 20479
I HAD always thought Jack Gisburn rather a cheap genius--though a good fellow
     enough--so it was no
```

Our goal is to tokenize this 20,479-character short story into individual words and special characters that we can then turn into embeddings for LLM training.

> **NOTE** It's common to process millions of articles and hundreds of thousands of books—many gigabytes of text—when working with LLMs. However, for educational purposes, it's sufficient to work with smaller text samples like a single book to illustrate the main ideas behind the text processing steps and to make it possible to run it in a reasonable time on consumer hardware.

How can we best split this text to obtain a list of tokens? For this, we go on a small excursion and use Python's regular expression library `re` for illustration purposes. (You don't have to learn or memorize any regular expression syntax since we will later transition to a prebuilt tokenizer.)

Using some simple example text, we can use the `re.split` command with the following syntax to split a text on whitespace characters:

```
import re
text = "Hello, world. This, is a test."
result = re.split(r'(\s)', text)
print(result)
```

The result is a list of individual words, whitespaces, and punctuation characters:

```
['Hello,', ' ', 'world.', ' ', 'This,', ' ', 'is', ' ', 'a', ' ', 'test.']
```

This simple tokenization scheme mostly works for separating the example text into individual words; however, some words are still connected to punctuation characters that we want to have as separate list entries. We also refrain from making all text lowercase because capitalization helps LLMs distinguish between proper nouns and common nouns, understand sentence structure, and learn to generate text with proper capitalization.

Let's modify the regular expression splits on whitespaces (`\s`), commas, and periods (`[,.]`):

```
result = re.split(r'([,.]|\s)', text)
print(result)
```

We can see that the words and punctuation characters are now separate list entries just as we wanted:

```
['Hello', ',', '', ' ', 'world', '.', '', ' ', 'This', ',', '', ' ', 'is',
' ', 'a', ' ', 'test', '.', '']
```

A small remaining problem is that the list still includes whitespace characters. Optionally, we can remove these redundant characters safely as follows:

```
result = [item for item in result if item.strip()]
print(result)
```

The resulting whitespace-free output looks like as follows:

```
['Hello', ',', 'world', '.', 'This', ',', 'is', 'a', 'test', '.']
```

> **NOTE** When developing a simple tokenizer, whether we should encode whitespaces as separate characters or just remove them depends on our application and its requirements. Removing whitespaces reduces the memory and computing requirements. However, keeping whitespaces can be useful if we train models that are sensitive to the exact structure of the text (for example, Python code, which is sensitive to indentation and spacing). Here, we remove whitespaces for simplicity and brevity of the tokenized outputs. Later, we will switch to a tokenization scheme that includes whitespaces.

The tokenization scheme we devised here works well on the simple sample text. Let's modify it a bit further so that it can also handle other types of punctuation, such as question marks, quotation marks, and the double-dashes we have seen earlier in the first 100 characters of Edith Wharton's short story, along with additional special characters:

```
text = "Hello, world. Is this-- a test?"
result = re.split(r'([,.:;?_!"()\']|--|\s)', text)
result = [item.strip() for item in result if item.strip()]
print(result)
```

The resulting output is:

```
['Hello', ',', 'world', '.', 'Is', 'this', '--', 'a', 'test', '?']
```

As we can see based on the results summarized in figure 2.5, our tokenization scheme can now handle the various special characters in the text successfully.

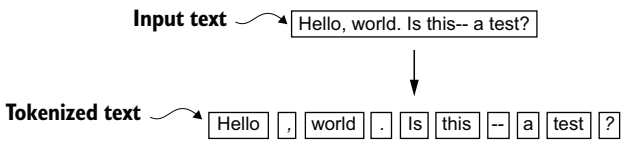


Figure 2.5 The tokenization scheme we implemented so far splits text into individual words and punctuation characters. In this specific example, the sample text gets split into 10 individual tokens.

Now that we have a basic tokenizer working, let's apply it to Edith Wharton's entire short story:

```
preprocessed = re.split(r'([,.:;?_!"()\']|--|\s)', raw_text)
preprocessed = [item.strip() for item in preprocessed if item.strip()]
print(len(preprocessed))
```

This print statement outputs `4690`, which is the number of tokens in this text (without whitespaces). Let's print the first 30 tokens for a quick visual check:

```
print(preprocessed[:30])
```

The resulting output shows that our tokenizer appears to be handling the text well since all words and special characters are neatly separated:

```
['I', 'HAD', 'always', 'thought', 'Jack', 'Gisburn', 'rather', 'a',
'cheap', 'genius', '--', 'though', 'a', 'good', 'fellow', 'enough',
'--', 'so', 'it', 'was', 'no', 'great', 'surprise', 'to', 'me', 'to',
'hear', 'that', ',', 'in']
```

2.3 Converting tokens into token IDs

Next, let's convert these tokens from a Python string to an integer representation to produce the token IDs. This conversion is an intermediate step before converting the token IDs into embedding vectors.

To map the previously generated tokens into token IDs, we have to build a vocabulary first. This vocabulary defines how we map each unique word and special character to a unique integer, as shown in figure 2.6.

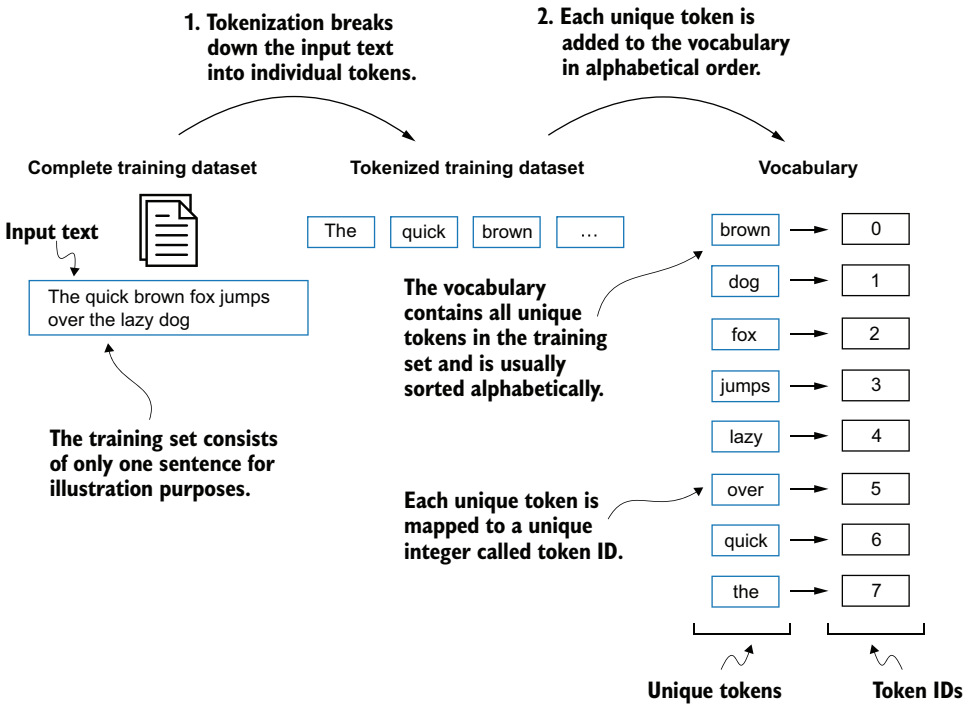


Figure 2.6 We build a vocabulary by tokenizing the entire text in a training dataset into individual tokens. These individual tokens are then sorted alphabetically, and duplicate tokens are removed. The unique tokens are then aggregated into a vocabulary that defines a mapping from each unique token to a unique integer value. The depicted vocabulary is purposely small and contains no punctuation or special characters for simplicity.

Now that we have tokenized Edith Wharton's short story and assigned it to a Python variable called `preprocessed`, let's create a list of all unique tokens and sort them alphabetically to determine the vocabulary size:

```
all_words = sorted(set(preprocessed))
vocab_size = len(all_words)
print(vocab_size)
```

After determining that the vocabulary size is 1,130 via this code, we create the vocabulary and print its first 51 entries for illustration purposes.

Listing 2.2 Creating a vocabulary

```
vocab = {token:integer for integer,token in enumerate(all_words)}
for i, item in enumerate(vocab.items()):
    print(item)
    if i >= 50:
        break
```

The output is

```
('!', 0)
('"', 1)
("'", 2)
...
('Her', 49)
('Hermia', 50)
```

As we can see, the dictionary contains individual tokens associated with unique integer labels. Our next goal is to apply this vocabulary to convert new text into token IDs (figure 2.7).

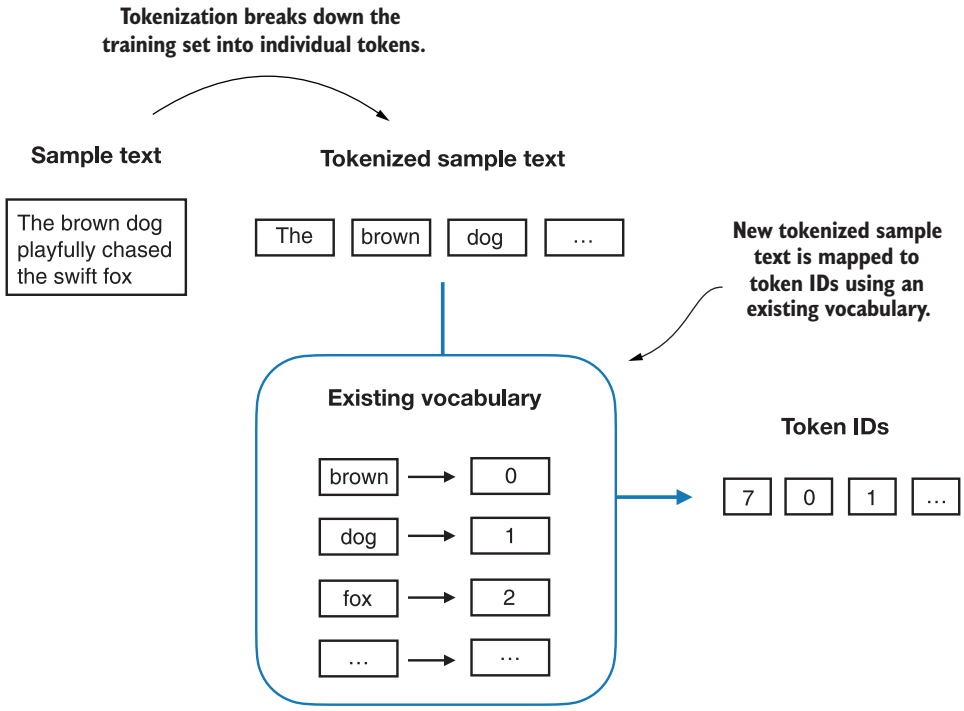


Figure 2.7 Starting with a new text sample, we tokenize the text and use the vocabulary to convert the text tokens into token IDs. The vocabulary is built from the entire training set and can be applied to the training set itself and any new text samples. The depicted vocabulary contains no punctuation or special characters for simplicity.

When we want to convert the outputs of an LLM from numbers back into text, we need a way to turn token IDs into text. For this, we can create an inverse version of the vocabulary that maps token IDs back to the corresponding text tokens.

Let's implement a complete tokenizer class in Python with an `encode` method that splits text into tokens and carries out the string-to-integer mapping to produce token IDs via the vocabulary. In addition, we'll implement a `decode` method that carries out the reverse integer-to-string mapping to convert the token IDs back into text. The following listing shows the code for this tokenizer implementation.

Listing 2.3 Implementing a simple text tokenizer

```
class SimpleTokenizerV1:
    def __init__(self, vocab):
        self.str_to_int = vocab
        self.int_to_str = {i:s for s,i in vocab.items()}

    def encode(self, text):
        preprocessed = re.split(r'([,.?_!"()\']|--|\s)', text)
        preprocessed = [
            item.strip() for item in preprocessed if item.strip()
        ]
        ids = [self.str_to_int[s] for s in preprocessed]
        return ids

    def decode(self, ids):
        text = " ".join([self.int_to_str[i] for i in ids])

        text = re.sub(r'\s+([,.?!"()\'])', r'\1', text)
        return text
```

Stores the vocabulary as a class attribute for access in the encode and decode methods

Creates an inverse vocabulary that maps token IDs back to the original text tokens

Processes input text into token IDs

Converts token IDs back into text

Removes spaces before the specified punctuation

Using the `SimpleTokenizerV1` Python class, we can now instantiate new tokenizer objects via an existing vocabulary, which we can then use to encode and decode text, as illustrated in figure 2.8.

Let's instantiate a new tokenizer object from the `SimpleTokenizerV1` class and tokenize a passage from Edith Wharton's short story to try it out in practice:

```
tokenizer = SimpleTokenizerV1(vocab)
text = """It's the last he painted, you know,"
       Mrs. Gisburn said with pardonable pride."""
ids = tokenizer.encode(text)
print(ids)
```

The preceding code prints the following token IDs:

```
[1, 56, 2, 850, 988, 602, 533, 746, 5, 1126, 596, 5, 1, 67, 7, 38, 851, 1108,
754, 793, 7]
```

Next, let's see whether we can turn these token IDs back into text using the decode method:

```
print(tokenizer.decode(ids))
```

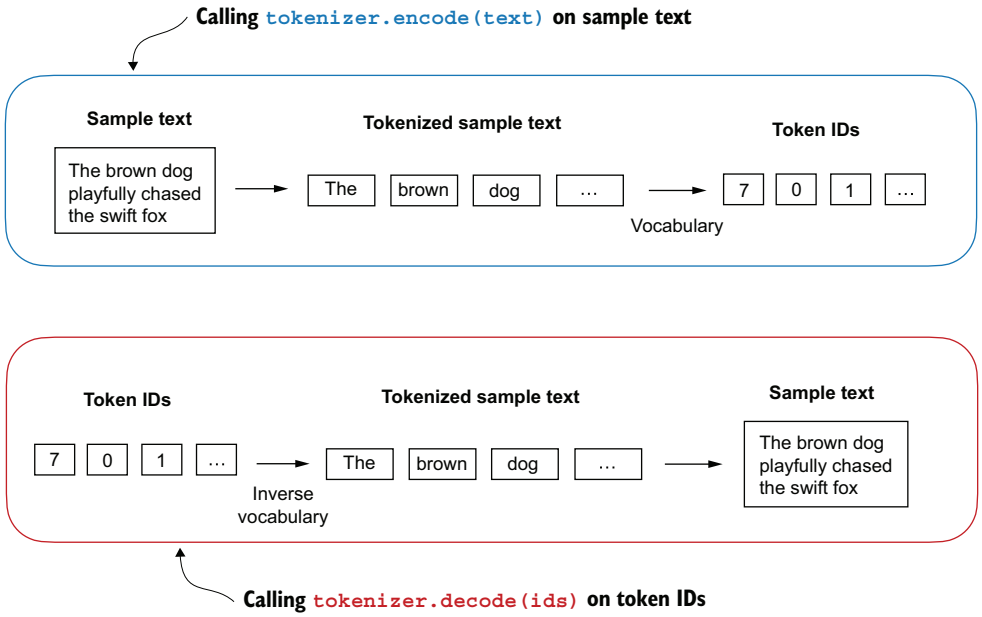


Figure 2.8 Tokenizer implementations share two common methods: an encode method and a decode method. The encode method takes in the sample text, splits it into individual tokens, and converts the tokens into token IDs via the vocabulary. The decode method takes in token IDs, converts them back into text tokens, and concatenates the text tokens into natural text.

This outputs:

```
'" It\' s the last he painted, you know," Mrs. Gisburn said with
pardonable pride.'
```

Based on this output, we can see that the decode method successfully converted the token IDs back into the original text.

So far, so good. We implemented a tokenizer capable of tokenizing and detokenizing text based on a snippet from the training set. Let's now apply it to a new text sample not contained in the training set:

```
text = "Hello, do you like tea?"
print(tokenizer.encode(text))
```

Executing this code will result in the following error:

```
KeyError: 'Hello'
```

The problem is that the word "Hello" was not used in the "The Verdict" short story. Hence, it is not contained in the vocabulary. This highlights the need to consider large and diverse training sets to extend the vocabulary when working on LLMs.