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将深度学习应用于实践并从头开始实现大型语言模型（LLMs）所需的知识和技能。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空间。

然后，您将了解张量及其在 PyTorch 中的使用的基本概念。我们还将深入了解 PyTorch 的自动微分引擎，这是一个使我们能够方便且高效地使用反向传播的功能，这是神经网络训练的关键方面。

本附录旨在为那些刚开始接触 PyTorch 深度学习的新手提供入门指南。

虽然它从底层解释了 PyTorch，但并不打算全面覆盖 PyTorch 库。相反，我们将关注我们将用于实现LLMs的 PyTorch 基础知识。如果您已经熟悉深度学习，您可以跳过本附录，直接进入第 2 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 Papers With Code（https://paperswithcode.com/trends）平台，该平台跟踪和分析研究论文，PyTorch 自 2019 年以来一直是最广泛使用的深度学习库，差距很大。此外，根据 Kaggle 2022 年数据科学和机器学习调查（https://www.kaggle.com/c/kaggle-survey-2022），使用 PyTorch 的受访者数量约为 40%，并且每年都在增长。

PyTorch 之所以如此受欢迎，其中一个原因是其用户友好的界面和效率。尽管易于访问，它并没有在灵活性上妥协，允许高级用户调整模型的高级方面以进行定制和

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.

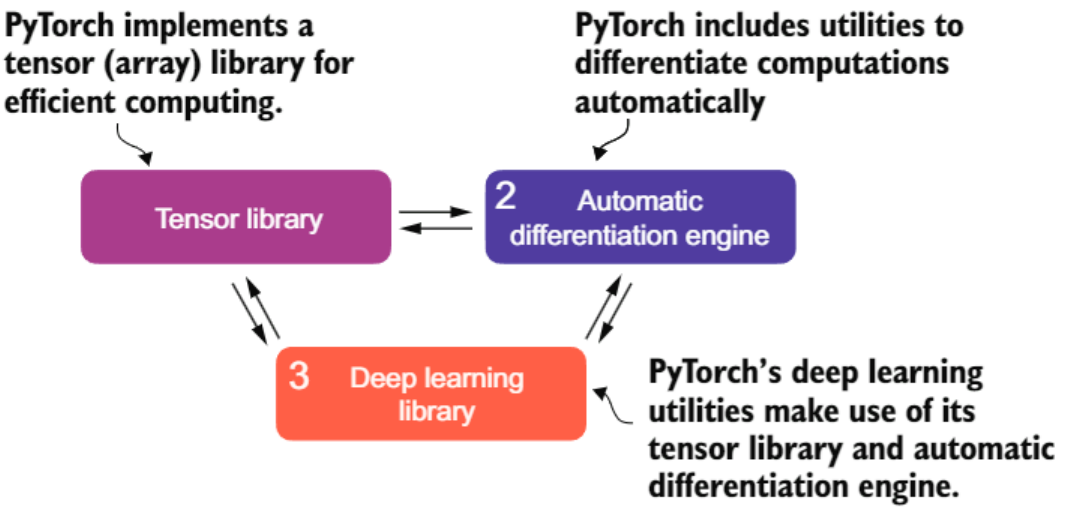


Figure A.1 PyTorch's three main components include a tensor library as a fundamental building block for computing, automatic differentiation for model optimization, and deep learning utility functions, making it easier to implement and train deep neural network models.

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，对于许多实践者和研究人员来说，PyTorch 在易用性和功能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。

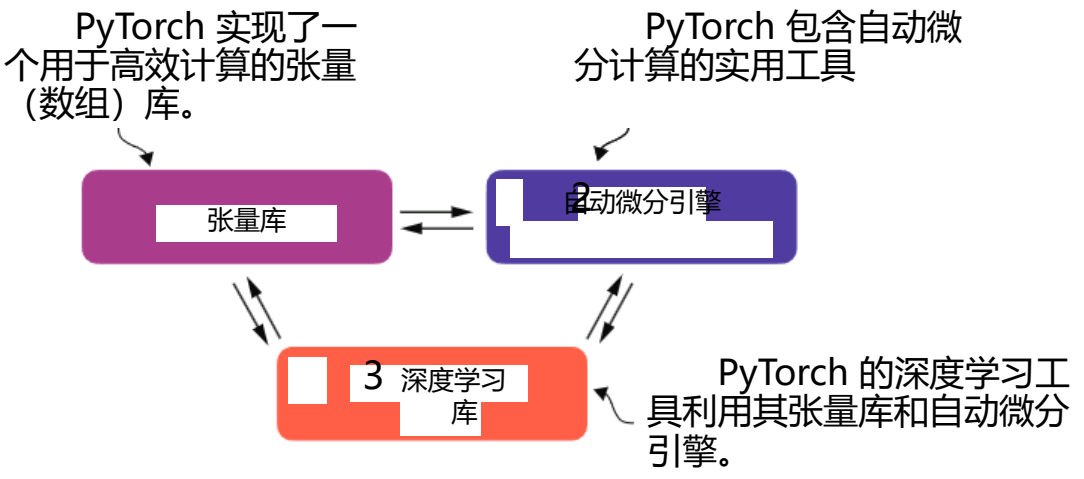


图 A.1 PyTorch 的三个主要组件包括作为计算基本构建块的张量库、用于模型优化的自动微分以及深度学习实用函数，使得实现和训练深度神经网络模型更加容易。

首先，PyTorch 是一个张量库，它扩展了数组导向编程库 NumPy 的概念，并增加了加速 GPU 计算的额外功能，从而实现了在 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 autograd，它能够自动计算张量操作的梯度，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

新闻中，LLMs 通常被称为 AI 模型。然而，LLMs 也是一种深度神经网络，PyTorch 是一个深度学习库。听起来很复杂？在我们继续之前，让我们简要总结一下这些术语之间的关系。

人工智能本质上是关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、识别模式和做出决策。（尽管取得了重大进展，但人工智能距离达到这种通用智能水平还远。）

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.

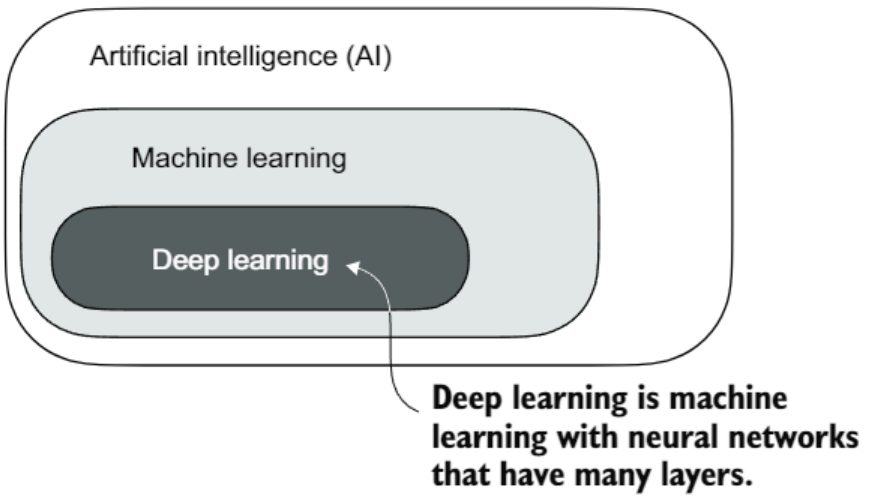


Figure A.2 Deep learning is a subcategory of machine learning focused on implementing deep neural networks. Machine learning is a subcategory of AI that is concerned with algorithms that learn from data. AI is the broader concept of machines being able to perform tasks that typically require human intelligence.

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。机器学习背后的关键思想是使计算机能够从数据中学习并做出预测或决策，而无需明确编程来执行该任务。这涉及到开发能够识别模式、从历史数据中学习，并在更多数据和反馈的帮助下随着时间的推移提高其性能的算法。

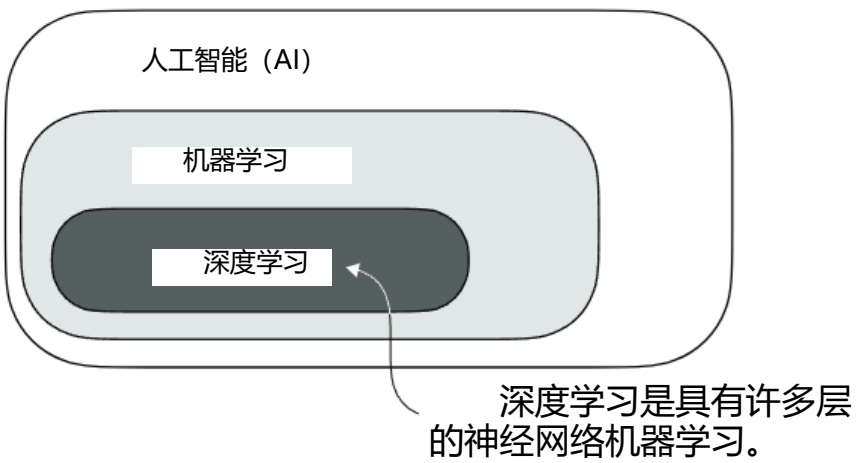


图 A.2 深度学习是机器学习的一个子类别，专注于实现深度神经网络。机器学习是人工智能的一个子类别，涉及从数据中学习的算法。人工智能是机器能够执行通常需要人类智能的任务的更广泛概念。

机器学习一直是人工智能发展的关键，推动了今天我们所看到的许多进步，包括 LLMs。机器学习还支持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、虚拟助手的语音识别，甚至自动驾驶汽车。机器学习的引入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，它专注于深度神经网络的训练和应用。这些深度神经网络最初是从人类大脑的工作原理中受到启发，特别是许多神经元之间的相互连接。在深度学习中，“深度”指的是人工神经元或节点之间的多层隐藏层，这使得它们能够模拟数据中的复杂非线性关系。与传统机器学习技术擅长简单模式识别不同，深度学习特别擅长处理图像、音频或文本等非结构化数据，因此特别适合于 LLMs。

机器学习和深度学习中典型的预测建模工作流程（也称为监督学习）在图 A.3 中进行了总结。

使用学习算法，在由示例及其对应标签组成的训练数据集上训练模型。例如，在电子邮件垃圾邮件分类器的情况下，训练数据集由电子邮件及其人类识别的“垃圾邮件”和“非垃圾邮件”标签组成。然后，训练好的模型可以用于新的观察（即新电子邮件）来预测它们的未知标签（“垃圾邮件”或“非垃圾邮件”）。当然，我们还想在训练和推理阶段之间添加模型评估。

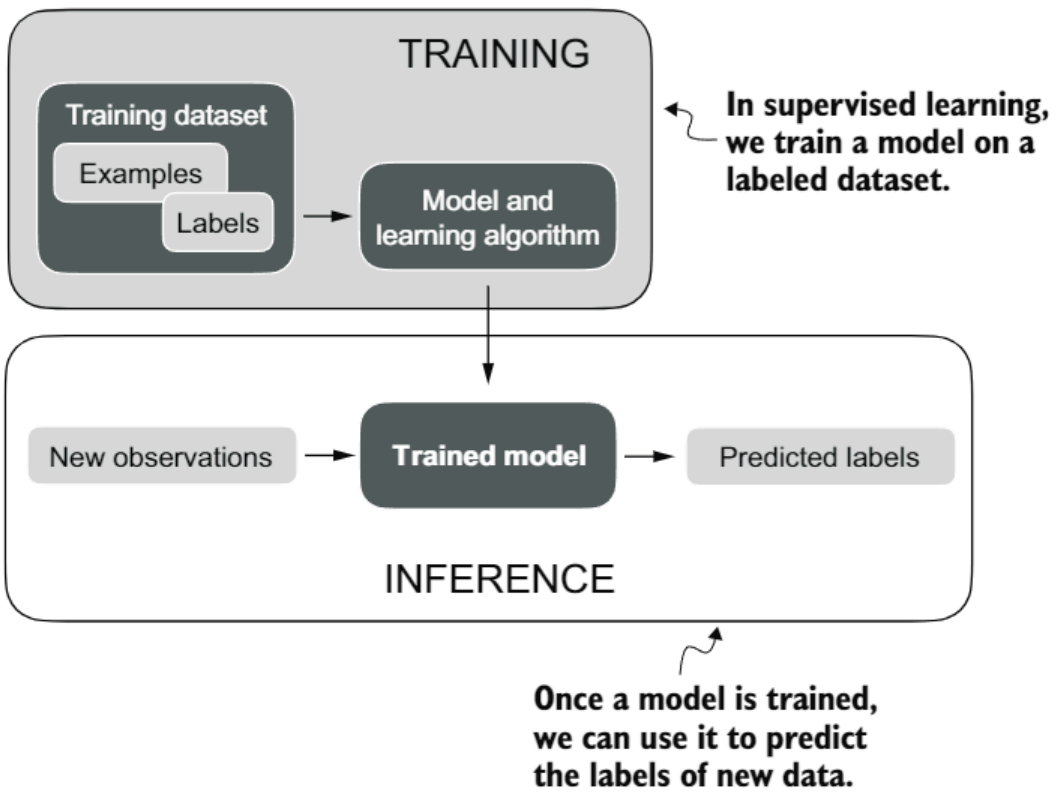


Figure A.3 The supervised learning workflow for predictive modeling consists of a training stage where a model is trained on labeled examples in a training dataset. The trained model can then be used to predict the labels of new observations.

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.

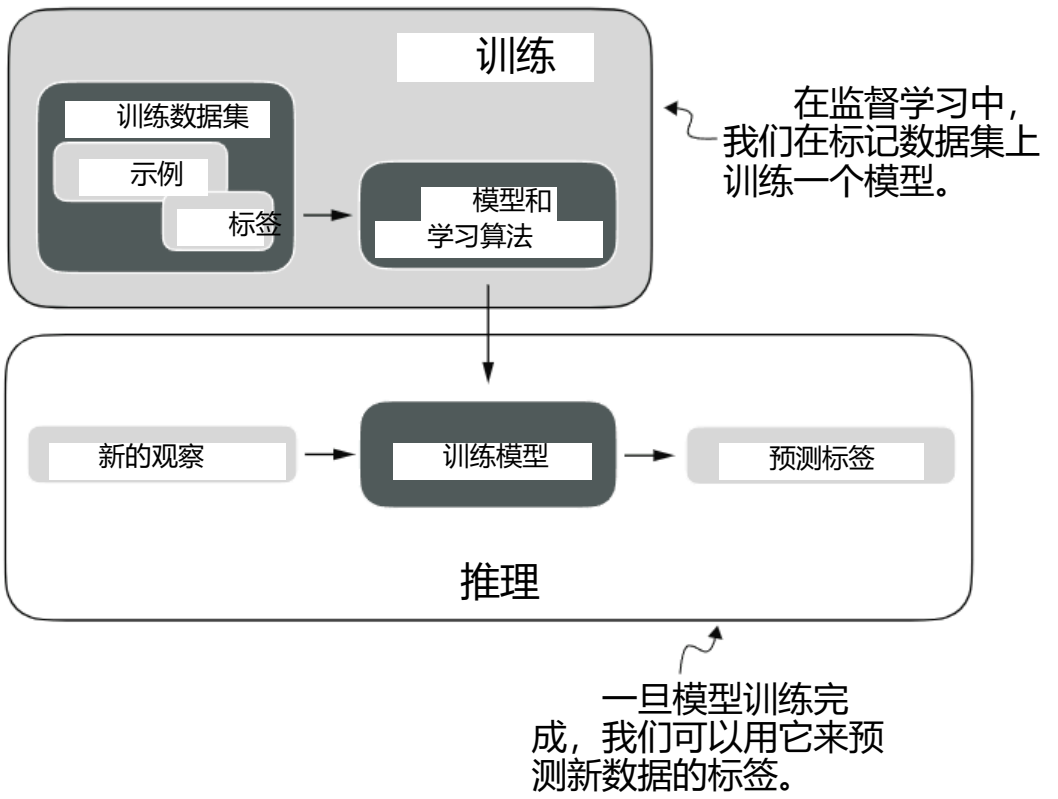


图 A.3 预测建模的监督学习工作流程包括一个训练阶段，在该阶段，模型在训练数据集中的标记示例上进行训练。训练好的模型可以用来预测新观察结果的标签。

我们还想在训练和推理阶段之间添加一个模型评估，以确保在将其用于实际应用之前，模型满足我们的性能标准。

如果我们训练LLMs来分类文本，训练和使用LLMs的工作流程与图 A.3 中描述的类似。如果我们对训练LLMs生成文本感兴趣，这是我们主要关注的焦点，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（第 1 章中引入的下一词预测任务）中推导出来。LLM将在推理过程中给定输入提示时生成全新的文本（而不是预测标签）。

A.1.3 安装 PyTorch

PyTorch 可以像安装其他 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，安装可能需要额外的说明。

Python 版本

许多科学计算库并不立即支持 Python 的最新版本。因此，在安装 PyTorch 时，建议使用比最新版本低一两个版本的 Python。例如，如果 Python 的最新版本是 3.13，则推荐使用 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.

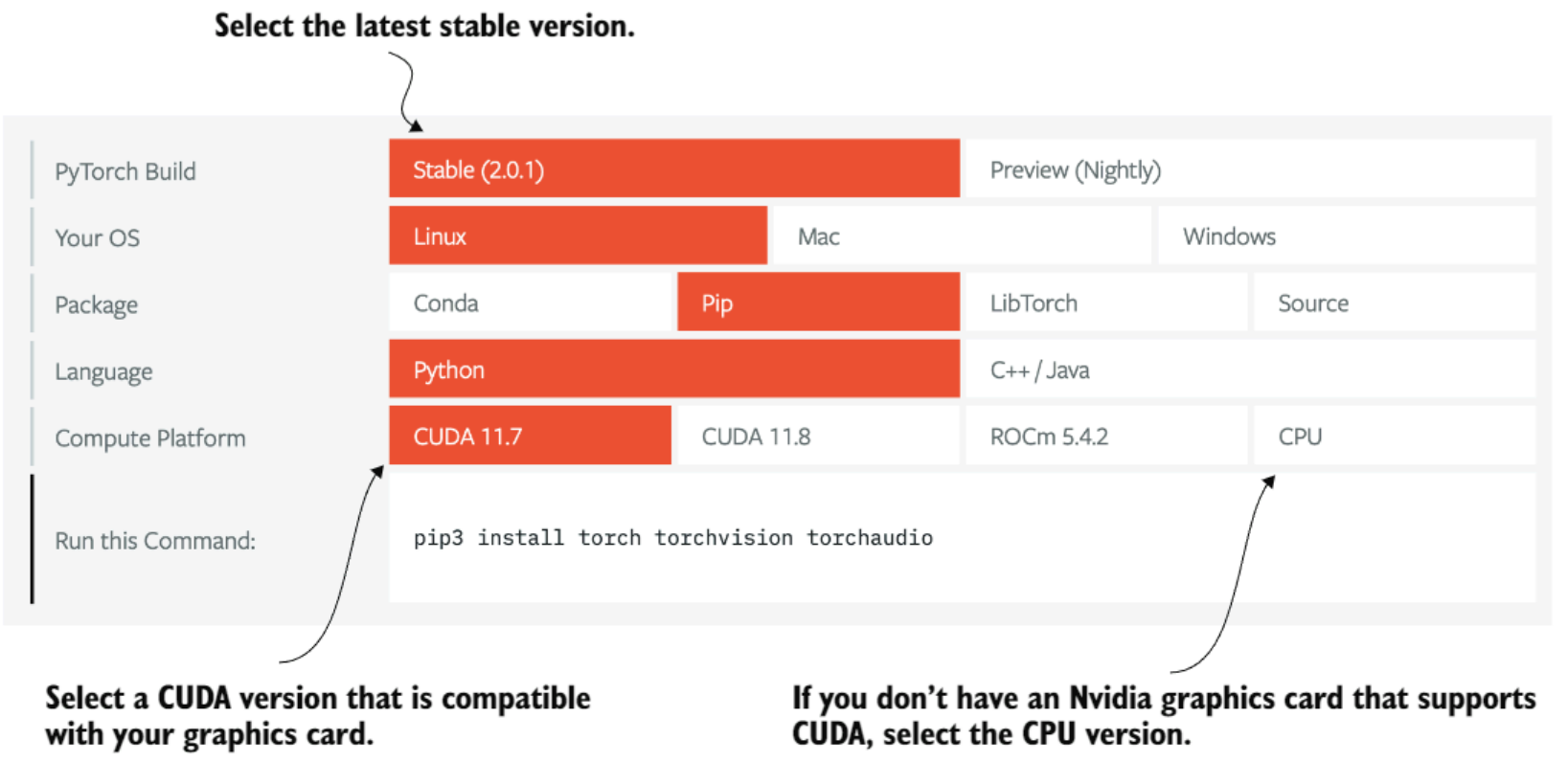


Figure A.4 Access the PyTorch installation recommendation on https://pytorch.org to customize and select the installation command for your system.

例如，PyTorch 有两个版本：一个仅支持 CPU 计算的轻量级版本和一个支持 CPU 和 GPU 计算的全功能版本。如果你的机器有一个可用于深度学习的 CUDA 兼容 GPU（理想情况下是 NVIDIA T4、RTX 2080 Ti 或更新的型号），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
```

（原文保持不变，因为“torch”是一个专有名词，通常指火炬或火把，在中文中也有对应的名称。）

假设您的计算机支持 CUDA 兼容的 GPU。在这种情况下，它将自动安装支持通过 CUDA 进行 GPU 加速的 PyTorch 版本，前提是您正在工作的 Python 环境已安装必要的依赖项（如 pip）。

注意：截至本文撰写时，PyTorch 已通过 ROCm 添加了对 AMD GPU 的实验性支持。有关更多信息，请参阅 https://pytorch.org。

明确安装与 CUDA 兼容的 PyTorch 版本时，通常最好指定 PyTorch 要兼容的 CUDA 版本。PyTorch 官方网站（https://pytorch.org）提供了在不同操作系统上安装具有 CUDA 支持的 PyTorch 的命令。图 A.4 显示了将安装 PyTorch 以及可选的 torchvision 和 torchaudio 库的命令。

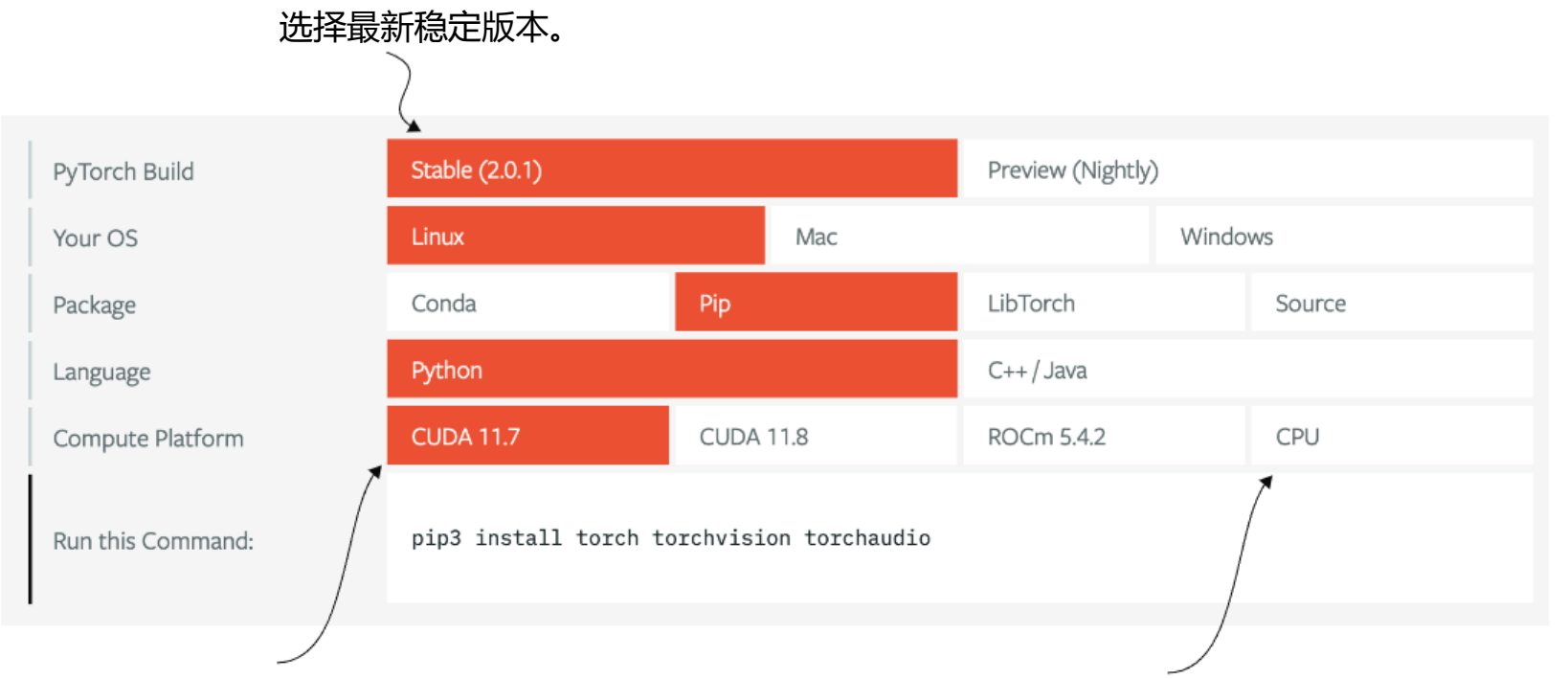


图 A.4 在 https://pytorch.org 上访问 PyTorch 安装推荐，以自定义并选择适合您系统的安装命令。

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.

我使用 PyTorch 2.4.0 进行示例，因此建议您使用以下命令安装确切版本以确保与本书兼容：

```
pip 安装 torch==2.4.0
```

然而，如前所述，鉴于您的操作系统，安装命令可能与这里显示的略有不同。因此，我建议您访问 https://pytorch.org，并使用安装菜单（见图 A.4）选择适合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。

检查 PyTorch 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 Torch

Python 库被命名为 PyTorch，主要是因为它是 Torch 库的延续，但为 Python 进行了适配（因此称为“PyTorch”）。“Torch”这个名字承认了该库在 Torch 中的根源，Torch 是一个广泛支持机器学习算法的科学计算框架，最初是用 Lua 编程语言创建的。

如果您正在寻找设置 Python 环境或安装本书中使用的其他库的额外推荐和说明，请访问本书的补充 GitHub 仓库：https://github.com/rasbt/ LLMs-从零开始。

安装 PyTorch 后，您可以通过在 Python 中运行以下代码来检查您的安装是否识别了内置的 NVIDIA GPU：

```
import torch
torch.cuda.is_available()
```

这返回

```
True
```

如果命令返回 True，您就设置好了。如果命令返回 False，您的计算机可能没有兼容的 GPU，或者 PyTorch 没有识别到它。虽然这本书的前几章不需要 GPU，它们专注于实现LLMs进行教育目的，但 GPU 可以显著加快与深度学习相关的计算。

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.

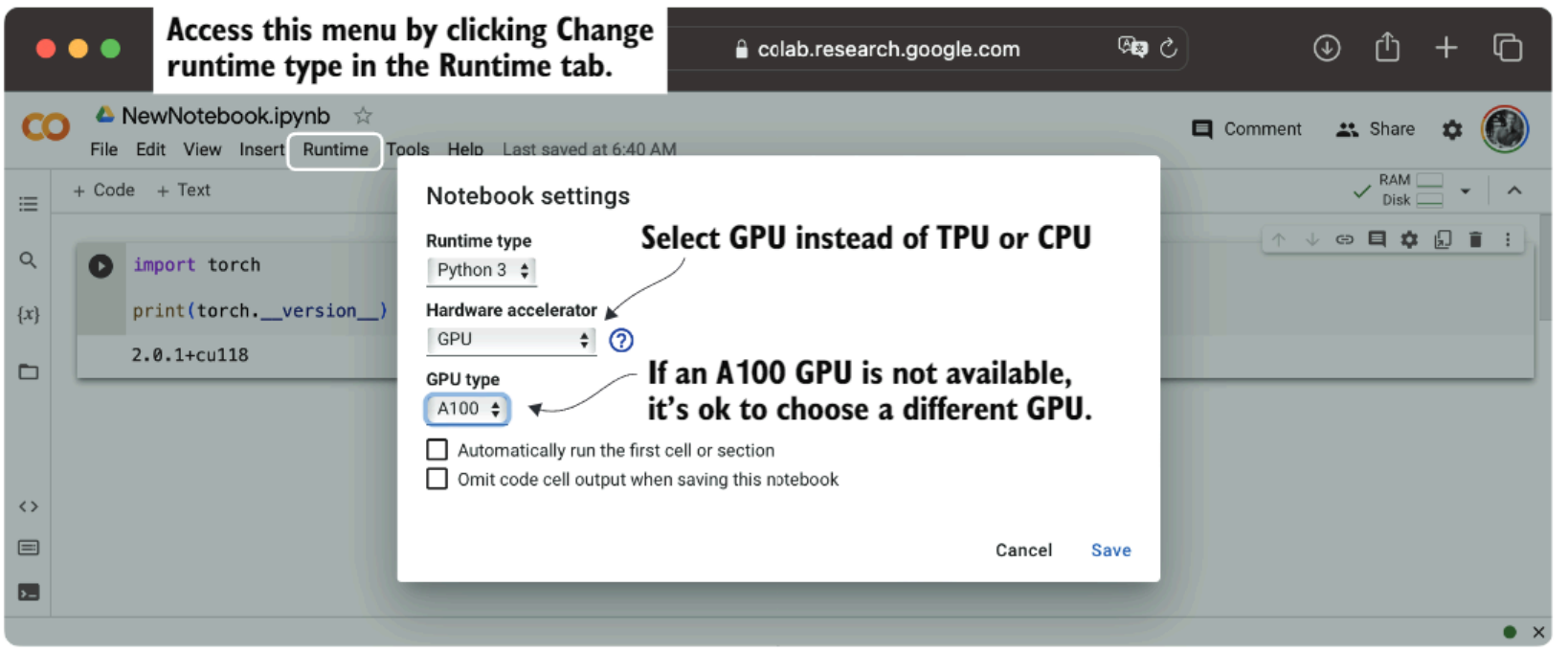


Figure A.5 Select a GPU device for Google Colab under the Runtime/Change Runtime Type menu.

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.

如果您没有访问 GPU 的权限，有多个云计算提供商允许用户以每小时计费的方式运行 GPU 计算。一个流行的类似 Jupyter 笔记本的环境是 Google Colab（https://colab.research.google.com），截至本文撰写时，它提供有限时间的 GPU 访问。使用运行时菜单，可以选择 GPU，如图 A.5 中的截图所示。

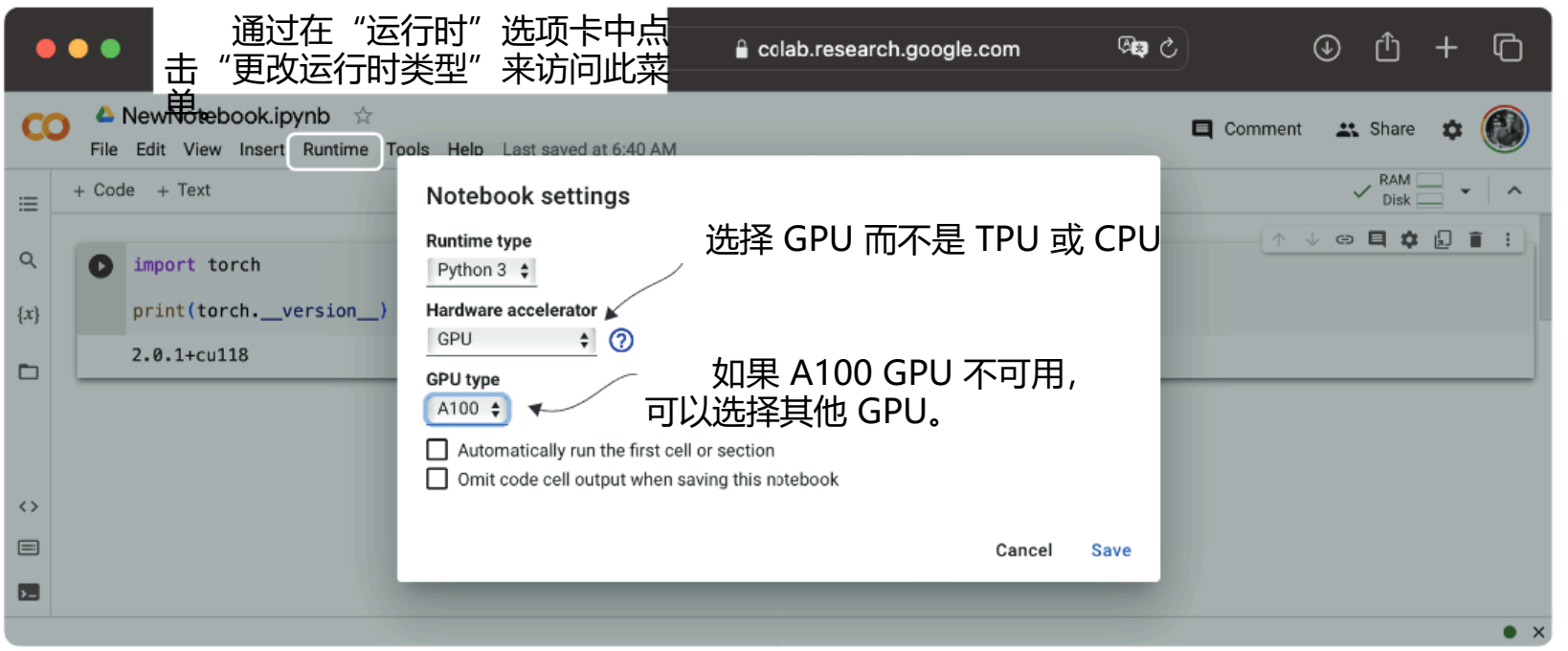


图 A.5 选择“运行/更改运行类型”菜单下的 GPU 设备用于 Google Colab。

PyTorch 在苹果硅上

如果您有一款搭载苹果硅芯片的苹果 Mac（如 M1、M2、M3 或更新的型号），您可以使用其功能来加速 PyTorch 代码执行。要使用您的苹果硅芯片进行 PyTorch，您首先需要像平时一样安装 PyTorch。然后，为了检查您的 Mac 是否支持使用其苹果硅芯片进行 PyTorch 加速，您可以在 Python 中运行一个简单的代码片段：

```
打印 torch.backends.mps 是否可用()
```

如果它返回 True，则表示您的 Mac 具有可用于加速 PyTorch 代码的 Apple Silicon 芯片。

练习 A.1

安装并设置 PyTorch 到您的计算机上

练习 A.2

运行位于 https://mng.bz/o05v 的补充代码，检查您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.

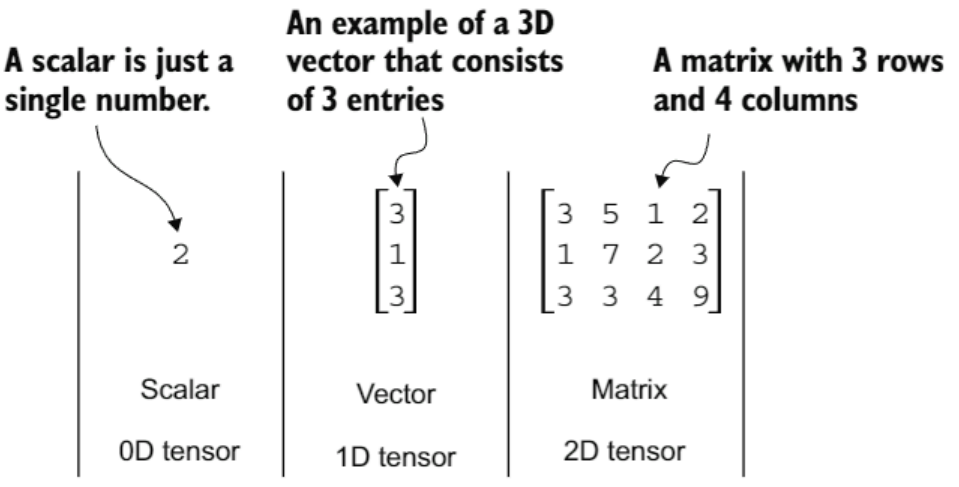


Figure A.6 Tensors with different ranks. Here 0D corresponds to rank 0, 1D to rank 1, and 2D to rank 2. A three-dimensional vector, which consists of three elements, is still a rank 1 tensor.

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

张量代表了一种数学概念，它将向量和矩阵推广到可能的高维。换句话说，张量是可以通过它们的阶数（或秩）来表征的数学对象，这提供了维数的数量。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。

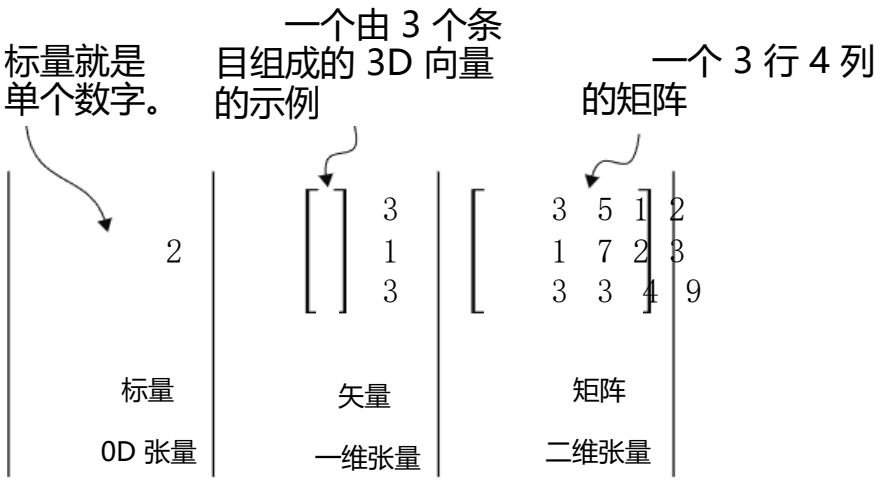


图 A.6 不同秩的张量。其中 0D 对应秩 0，1D 对应秩 1，2D 对应秩 2。由三个元素组成的三维向量仍然是一个秩 1 张量。

从计算角度来看，张量充当数据容器。例如，它们持有多维数据，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。在这种情况下，张量库充当数组库的功能。

PyTorch 张量类似于 NumPy 数组，但具有几个对深度学习很重要的附加功能。例如，PyTorch 添加了一个自动微分引擎，简化了梯度的计算（见 A.4 节）。PyTorch 张量还支持 GPU 计算，以加速深度神经网络训练（见 A.8 节）。

> **PyTorch 具有类似 NumPy 的 API**
>
> PyTorch 采用了 NumPy 数组 API 和语法的大部分内容来进行张量操作。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”快速了解最相关的概念，文章链接为 https://sebastianraschka.com/blog/2020/numpy-intro.html。

A.2.1 标量、向量、矩阵和张量

如前所述，PyTorch 张量是类似数组的结构的数据容器。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。对于更高维度的张量没有特定的术语，所以我们通常将三维张量称为 3D 张量，依此类推。我们可以使用 torch.tensor 函数创建 PyTorch 的 Tensor 类的对象，如下所示。

Listing A.1 Creating PyTorch tensors

```
import torch

tensor0d = torch.tensor(1)

tensor1d = torch.tensor([1, 2, 3])

tensor2d = torch.tensor([[1, 2],
                         [3, 4]])

tensor3d = torch.tensor([[[1, 2], [3, 4]],
                         [[5, 6], [7, 8]]])
```

Creates a zero-dimensional tensor (scalar) from a Python integer

Creates a one-dimensional tensor (vector) from a Python list

Creates a two-dimensional tensor from a nested Python list

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
```

列表 A.1 创建 PyTorch 张量

```
导入 torch
    （原文保持不变，因为“torch”是一个专有名词，通常指定深度学习库，在中文中也有对应的名称。）
tensor0d = torch.tensor(1)

tensor1d = torch.tensor([1,]      2,  3])

tensor2d = torch.tensor([[1, 2],
                         [3, 4]]]

tensor3d = torch.tensor([[[1, 2], [3, 4], [5, 6]],], dtype=torch.float32)
                         [5, 6], [7,]   8]]])
```

从 Python 整数创建零维张量（标量）

创建一个从 Python 列表生成的一维张量（向量）

创建一个由嵌套 Python 列表组成的二维张量

创建一个由嵌套 Python 列表构成的三维张量

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：

```
tensor1d = torch.tensor([1, 2, 3]) 打
印(tensor1d 的数据类型)
```

这会打印

```
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 的数据类型)
```

输出结果

```
torch.float32
```

这一选择主要是由于精度和计算效率之间的平衡。32 位浮点数对于大多数深度学习任务来说提供了足够的精度，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快模型训练和推理速度。

此外，可以使用张量的.to 方法更改精度。以下代码通过将 64 位整数张量转换为 32 位浮点张量来演示这一点：

```
floatvec = tensor1d.to(torch.float32) 打
印(floatvec.dtype)
```

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

关于 PyTorch 中可用的不同张量数据类型的信息，请查看官方文档：https://pytorch.org/docs/stable/tensors.html。

A.2.3 常见 PyTorch 张量操作

全面覆盖所有不同的 PyTorch 张量操作和命令超出了本书的范围。然而，在本书中介绍相关操作时，我会简要描述它们。

我们已经介绍了 torch.tensor()函数来创建新的张量：

```
tensor2d = 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 的形状
```

输出结果

```
torch.Size([2, 3])
```

如您所见，.shape 返回 [2, 3]，表示张量有两行三列。要将张量重塑为 3 × 2 张量，我们可以使用 .reshape 方法：

```
print(tensor2d.reshape(3, 2))
```

这会打印

```
张量([[1, 2],
         [3, 4]
         [5, 6]]]
```

然而，请注意，在 PyTorch 中重塑张量的更常见命令是

```
.view(): 显示视图()

打印(tensor2d.view(3,))     2))
```

输出结果

```
张量([[1, 2],
         [3, 4]
         [5, 6]]]
```

类似于 .reshape 和 .view，在许多情况下，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 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 最初遵循了原始 Lua Torch 语法约定，但随后根据普遍要求，增加了使其类似于 NumPy 的语法。 (PyTorch 中.view()和.reshape()之间的细微差别在于它们对内存布局的处理：.view()要求原始数据是连续的，如果不是，则会失败，而.reshape()将始终工作，如果需要，会复制数据以确保所需的形状。)

接下来，我们可以使用 .T 来转置一个张量，这意味着将其沿对角线翻转。注意，这与重塑张量类似，如以下结果所示：

```
打印 tensor2d 的转置
```

输出结果

```
张量([[1,      4],
         [2, 5]
         [3,  6]])
```

最后，在 PyTorch 中乘以两个矩阵的常用方法是使用.matmul 方法：

```
print(tensor2d 矩阵乘以 tensor2d 的转置)
```

输出结果

```
张量([[14, 32],
         [32, 77]]]
```

然而，我们也可以采用@运算符，它以更紧凑的方式完成相同的事情：

```
打印(tensor2d @)     tensor2d.T
```

这会打印

```
张量([[14, 32],
         [32, 77]]]
```

如前所述，在需要时，我会介绍额外的操作。对于想要浏览 PyTorch 中所有不同张量操作的读者（我们不需要这些中的大多数），我建议查看官方文档：https://pytorch.org/docs/stable/tensors.html。

A.3 将模型视为计算图

现在让我们看看 PyTorch 的自动微分引擎，也称为 autograd。PyTorch 的 autograd 系统提供了自动计算动态计算图中梯度的函数。

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。在深度学习的背景下，计算图定义了

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.

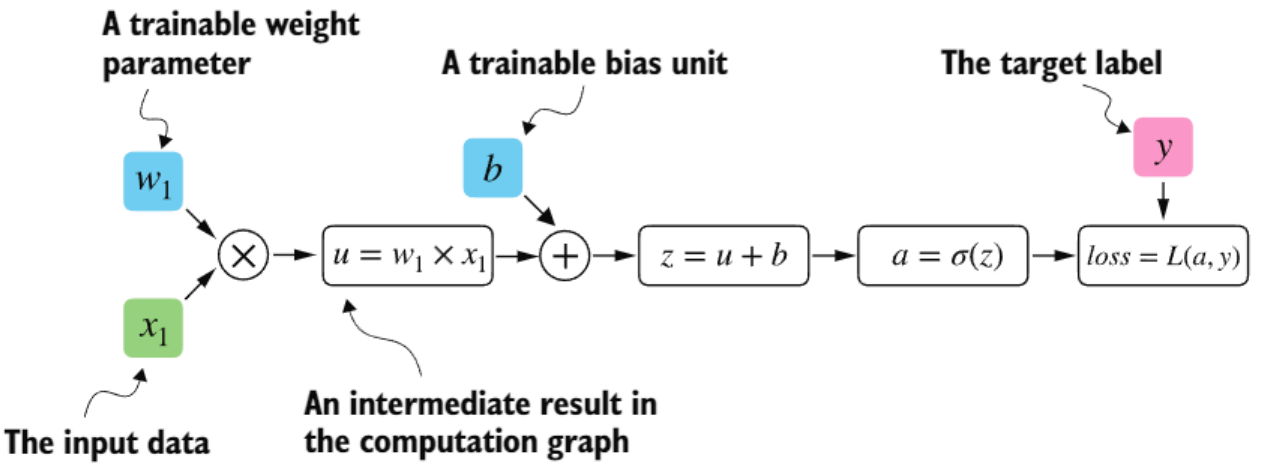


Figure A.7 A logistic regression forward pass as a computation graph. The input feature x_1 is multiplied by a model weight w_1 and passed through an activation function σ after adding the bias. The loss is computed by comparing the model output a with a given label y.

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.

计算图展示了计算神经网络输出所需的计算顺序，我们将需要它来计算反向传播所需的梯度，这是神经网络的主要训练算法。

让我们通过一个具体例子来说明计算图的概念。

以下代码实现了简单逻辑回归分类器的正向传播（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 到 1 之间的分数，在计算损失时将该分数与真实类别标签（0 或 1）进行比较。

列表 A.2 逻辑回归前向传播

```
导入 torch.nn.functional 作为 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) 损失 =

F.二元交叉熵(a, y)
```

这个导入语句是 PyTorch 中的一种常见约定，用于防止代码行过长。

真实标签

输入特征

权重参数

偏差单元

网络输入

激活和输出

如果前面的代码中不是所有组件都对你有意义，不要担心。这个示例的目的不是实现逻辑回归分类器，而是说明我们可以将一系列计算视为一个计算图，如图 A.7 所示。

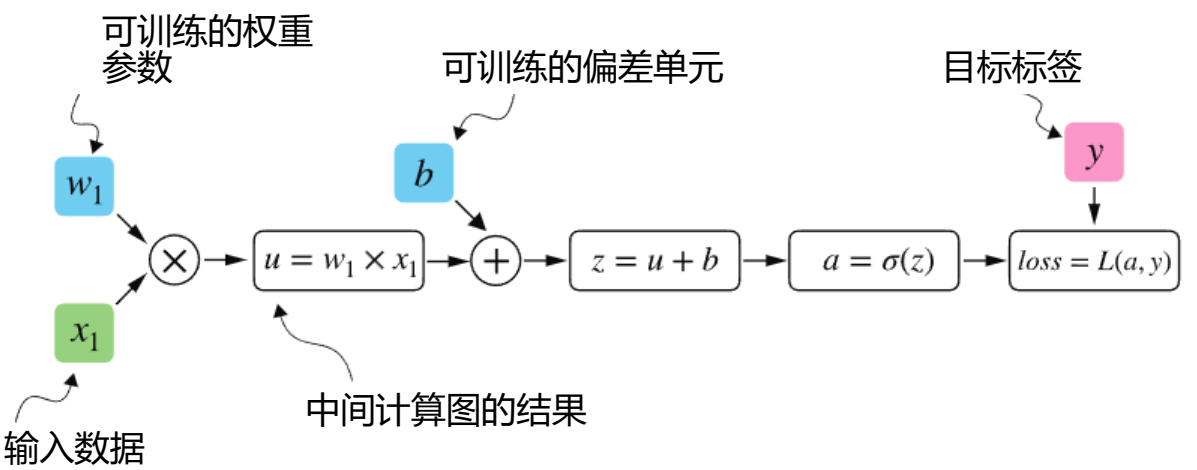


图 A.7 逻辑回归前向传播作为一个计算图。输入特征 x 乘以模型权重 w，并在添加偏差后通过激活函数σ，损失通过比较模型输出 a 与给定标签 y 来计算。

实际上，PyTorch 在后台构建这样的计算图，我们可以利用这个计算图来计算损失函数相对于模型参数（此处为 w 和 b）的梯度，以训练模型。

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.

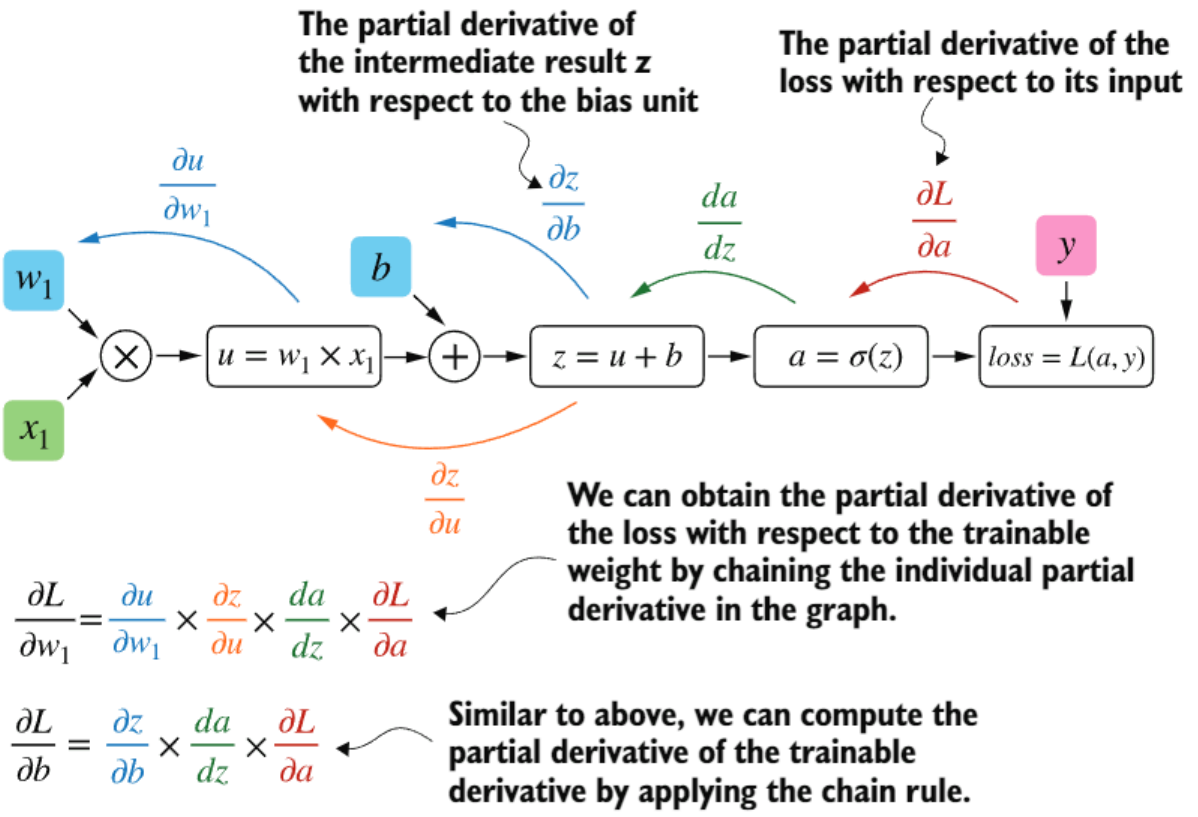


Figure A.8 The most common way of computing the loss gradients in a computation graph involves applying the chain rule from right to left, also called reverse-model automatic differentiation or backpropagation. We start from the output layer (or the loss itself) and work backward through the network 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

如果我们使用 PyTorch 进行计算，它将默认内部构建一个计算图，如果其终端节点之一设置了 requires_grad 属性为 True。如果我们想计算梯度，这很有用。在通过流行的反向传播算法训练神经网络时需要梯度，这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。

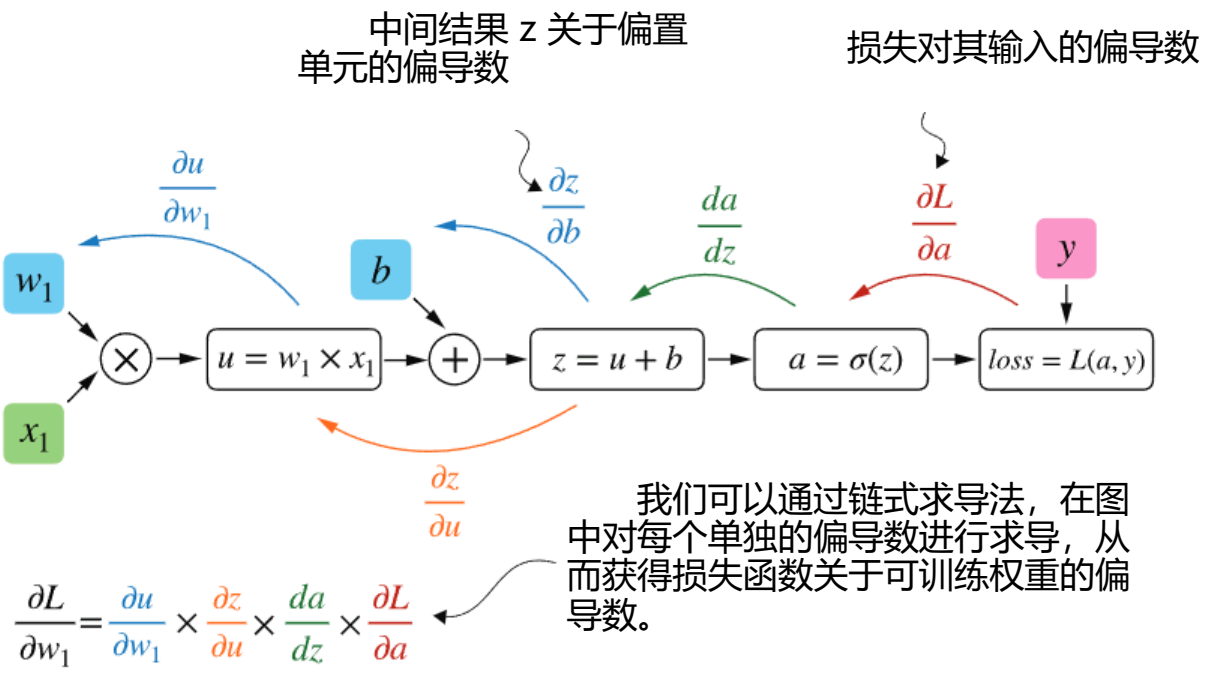


图 A.8 在计算图中计算损失梯度的最常见方法是从右向左应用链式法则，也称为反向模型自动微分或反向传播。我们从输出层（或损失本身）开始，通过网络反向工作到输入层。我们这样做是为了计算损失相对于网络中每个参数（权重和偏置）的梯度，这告诉我们如何在训练期间更新这些参数。

P

图 A.8 显示了偏导数，它衡量函数相对于其变量的变化率。梯度是一个包含多变量函数所有偏导数的向量，即输入变量超过一个的函数。

如果您不熟悉或记不起微积分中的偏导数、梯度或链式法则，不要担心。从高层次来看，您需要了解这本书的是，链式法则是通过计算图中的模型参数来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，该损失函数作为衡量代理的

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]),)
```

该代理用于通过梯度下降等方法衡量模型性能。我们将在第 A.7 节回顾 PyTorch 中此训练循环的计算实现。

这与前面提到的 PyTorch 库的第二个组件自动微分（autograd）引擎有何关联？PyTorch 的 autograd 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，我们可以计算关于模型参数 w1 的损失梯度，如下所示列表。

列表 A.3 通过自动微分计算梯度

```
    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) -> torch.sigmoid(z)

    损失 =  F.二元交叉熵(a, y)


    grad_L_w1 = 梯度（损失，w1，保留图=True）grad_L_b =
梯度（损失，b，保留图=True）
```

默认情况下，PyTorch 在计算梯度后会销毁计算图以释放内存。然而，由于我们将重用这个

计算图简而言之，我们设置 retain_graph=True 以使

模型参数给出的损失结果值

```
    打印(grad_L_w1)
    打印(grad_L_b)
```

这会打印

```
    （张量
([-0.0898])，） （张量
([-0.0817])，）
```

这里，我们一直在手动使用 grad 函数，这对于实验、调试和展示概念很有用。但是，在实践中，PyTorch 提供了更多高级工具来自动化这个过程。例如，我们可以对损失调用.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：

```
    loss.backward() 逆向传播损失
    打印 w1 的梯度
    打印(b.grad)
```

输出结果

```
    （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.

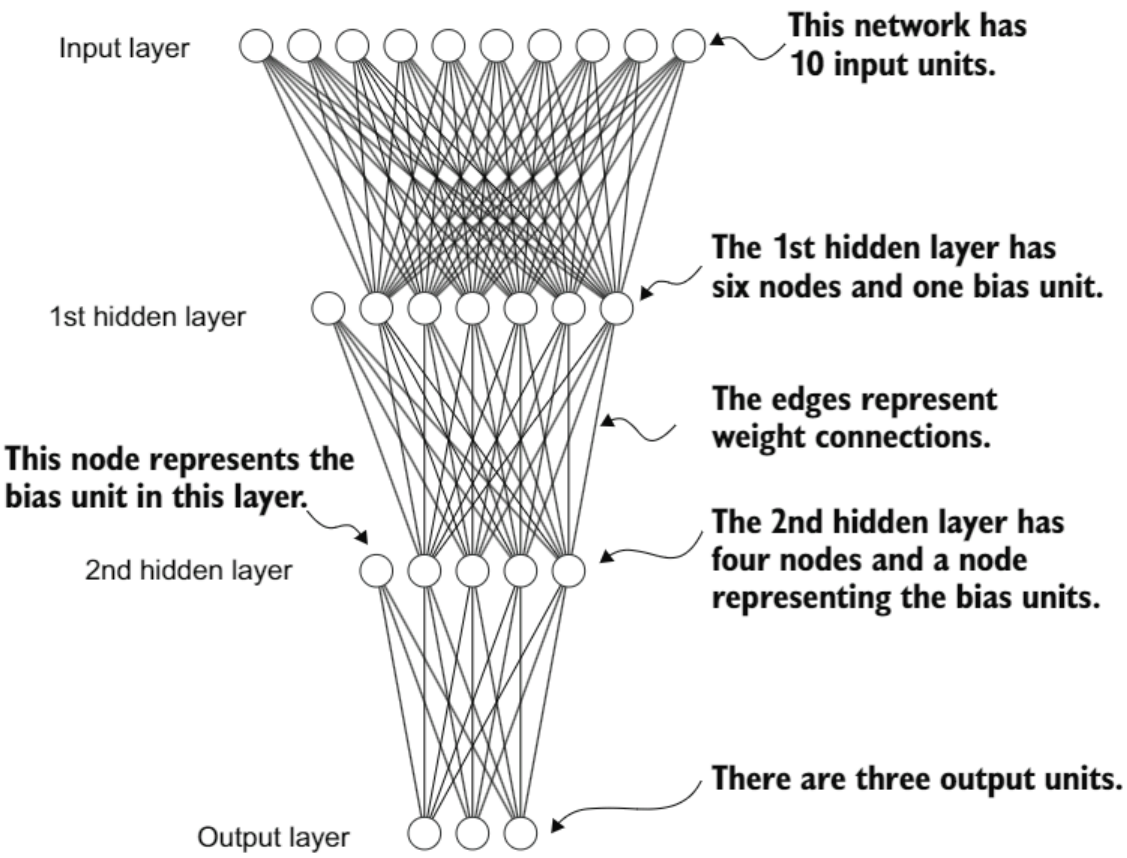


Figure A.9 A multilayer perceptron with two hidden layers. Each node represents a unit in the respective layer. For illustration purposes, each layer has a very small number of nodes.

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

我已经为你提供了大量信息，你可能对微积分概念感到不知所措，但别担心。虽然这些微积分术语是解释 PyTorch 的 autograd 组件的手段，但你只需要记住的是，PyTorch 通过.backward 方法为我们处理微积分——我们不需要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 PyTorch 作为一个实现深度神经网络的库进行关注。为了提供一个具体的例子，让我们看看图 A.9 中所示的多层感知器，一个全连接神经网络。

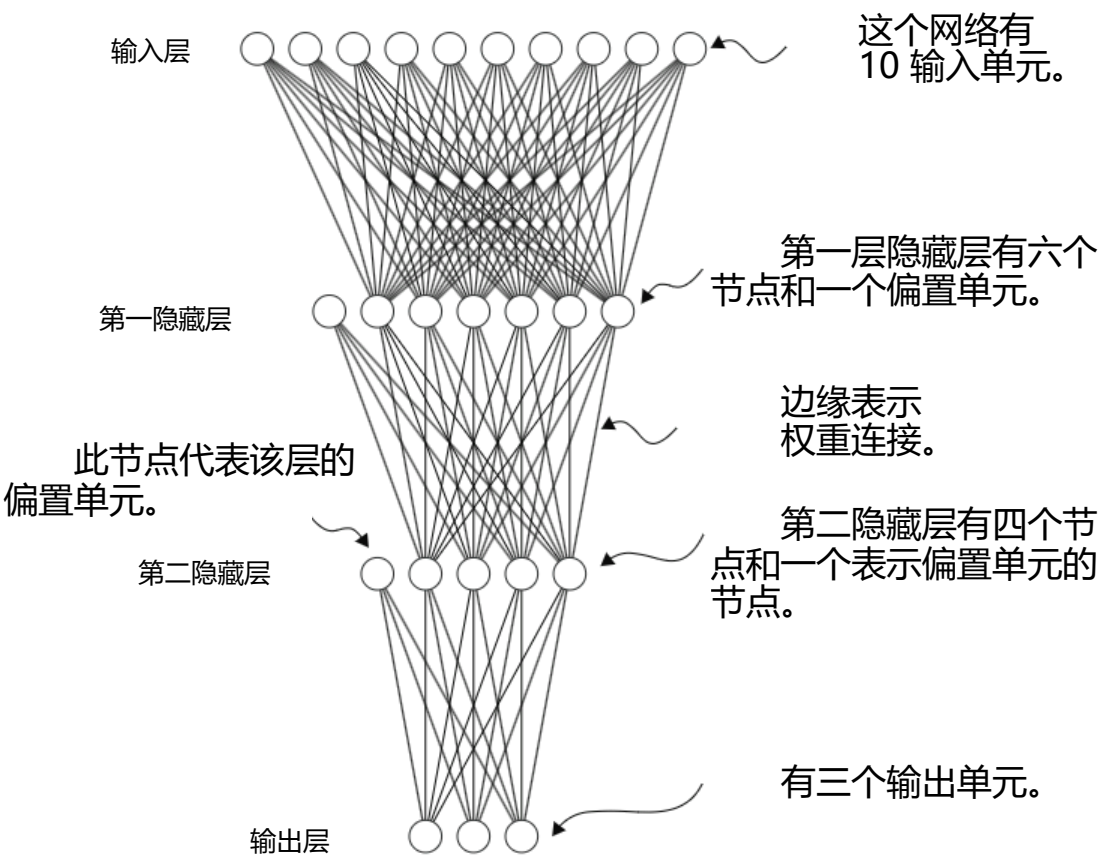


图 A.9 具有两个隐藏层的多层感知器。每个节点代表相应层中的一个单元。为了说明目的，每一层都有非常少的节点。

在 PyTorch 中实现神经网络时，我们可以通过继承 torch.nn.Module 类来定义自己的自定义网络架构。这个 Module 基类提供了很多功能，使得构建和训练模型更加容易。例如，它允许我们封装层和操作，并跟踪模型的参数。

在这个子类中，我们在__init__构造函数中定义网络层，并在 forward 方法中指定层之间的交互方式。forward 方法描述了输入数据如何通过网络并作为一个计算图汇集在一起。相比之下，我们通常不需要自己实现的 backward 方法，在训练期间用于根据模型参数计算损失函数的梯度（见 A.7 节）。以下列表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
```

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

The Linear layer takes the number of input and output nodes as arguments.

Nonlinear activation functions are placed between the hidden layers.

The number of output nodes of one hidden layer has to match the number of inputs of the next layer.

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

以下代码实现了一个具有两个隐藏层的经典多层感知器，以展示典型用法模块类。

列表 A.4 多层感知器，具有两个隐藏层

```
class 神经网络(torch.nn.Module):
    def __init__(self, num_inputs, num_outputs):
        super().__init__()

        self.layers = torch.nn.Sequential()

            # 1st 隐藏层
         torch.nn.Linear(num_inputs, 30),
         torch.nn.ReLU()


            # 第二隐藏层
         torch.nn.Linear(30, 20),
         torch.nn.ReLU()


        输出层 torch.nn.Linear(20,
     num_outputs)



    def forward(self, x): # 定义前向传播函数
        logits = self.layers(x)
     返回 logits
```

编码输入和输出的数量作为变量，使我们能够为具有不同特征和类别的数据集重用相同的代码

线性层将输入和输出节点的数量作为参数。

非线性激活函数置于隐藏层之间。

一个隐藏层的输出节点数必须与下一层的输入数相匹配。

最后一层的输出称为 logits。

我们可以如下实例化一个新的神经网络对象：

```
模型    神经网络(50, 3)
```

在使用此新模型对象之前，我们可以调用 print 在模型上查看其结构的摘要：

```
打印(model)
```

这会打印

```
    神经网络( (层):
Sequential(
    (0): 线性层(in_features=50, out_features=30, bias=True) (1):
ReLU() (2): 线性层(in_features=30, out_features=20, bias=True) (3):
ReLU() (4): 线性层(in_features=20, out_features=3, bias=True) ) )
```

请注意，在实现 NeuralNetwork 类时我们使用 Sequential 类。Sequential 不是必需的，但如果我们要按特定顺序执行一系列层，它可以使我们的工作更简单，就像这里的情况一样。这样，在__init__构造函数中将 self.layers = Sequential(...)实例化之后，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 NeuralNetwork 中逐个调用每个层
前进方法。

接下来，让我们检查这个模型的可训练参数总数：

```
num_params = 模型参数总数：sum(p.numel() for p in model.parameters() if
p.requires_grad) 打印("Total number of trainable model parameters:", num_params)
```

这会打印

```
总训练数量                    模型参数：2213
```

每个需要_grad=True 的参数都算作可训练参数，将在训练过程中更新（见 A.7 节）。

在前面两个隐藏层的情况下，这些可训练参数包含在 torch.nn.Linear 层中。线性层将输入与权重矩阵相乘并加上偏置向量。这有时被称为前馈或全连接层。

基于这里执行的 print(model) 调用，我们可以看到第一个线性层在 layers 属性中的索引位置为 0。我们可以按以下方式访问相应的权重参数矩阵：

```
打印(model.layers[0].weight)
```

这会打印

```
参数包含：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)
```

由于这个大矩阵没有全部显示，让我们使用.shape 属性来显示其维度：

```
打印(model.layers[0].weight.shape)
```

结果是

```
torch.Size([30, 50])
```

同样，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 30x50 的矩阵，我们可以看到 requires_grad 被设置为

这意味着其条目是可训练的——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前面的代码，权重矩阵中的数字可能与显示的不同。模型权重以小的随机数初始化，每次实例化网络时都不同。在深度学习中，用小的随机数初始化模型权重是为了在训练期间打破对称性。否则，节点在反向传播期间将执行相同的操作和更新，这不会允许网络从输入到输出学习复杂的映射。

然而，虽然我们希望继续使用小的随机数作为我们层权重的初始值，但我们可以通过手动设置种子 manual_seed 来使 PyTorch 的随机数生成器初始化可重现：

```
    torch 手动设置随机种子(123) 模型
= 神经网络(50, 3) 打印(模型.层[0].权
重)
```

结果是

```
    包含参数：
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)
```

现在我们已经花了一些时间检查了 NeuralNetwork 实例，让我们简要看看它是如何通过前向传递来使用的：

```
    torch.manual_seed(123) X
= torch.rand((1, 50)) out =
model(X) 打印(out)
```

结果是

```
    张量([[-0.1262,         [0.1080, -0.1792]], grad_fn=)
```

在上一段代码中，我们生成了一个随机的单个训练示例 X 作为玩具输入（请注意，我们的网络期望 50 维的特征向量），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正向传播。

正向传播是指从输入张量计算输出张量的过程。这涉及到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，最终到达输出层。

这三个返回的数字对应于分配给三个输出节点之一的分数。注意，输出张量还包括一个 grad_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= 表示在计算图中计算变量所使用的最后一个函数。特别是，grad_fn= 表示我们正在检查的张量是通过矩阵乘法和加法操作创建的。PyTorch 在计算梯度时会使用这个信息。

传播。grad_fn=中的部分指定了

操作已执行。在这种情况下，它是一个 Addmm 操作。Addmm 代表矩阵乘法（mm）后跟加法（Add）。

如果我们只想使用网络而不进行训练或反向传播——例如，如果我们用它进行训练后的预测——构建这个反向传播的计算图可能会造成浪费，因为它执行了不必要的计算并消耗了额外的内存。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。这告诉 PyTorch 它不需要跟踪梯度，这可以显著节省内存和计算：

```
使用 torch.no_grad():
    out = 模型(X)
打印(out)
```

结果是

```
张量([[-0.1262,         0.1080, -0.1792]])
```

在 PyTorch 中，通常的做法是编写模型，使其返回最后一层的输出（logits），而不将它们传递给非线性激活函数。这是因为 PyTorch 常用的损失函数将 softmax（或二分类中的 sigmoid）操作与负对数似然损失结合成一个类别。这样做的原因是数值效率和稳定性。因此，如果我们想计算预测的类别成员概率，我们必须显式调用 softmax 函数：

```
使用 torch.no_grad():
输出：out = torch.softmax(model(X),
dim=1) 打印(out)
```

这会打印

```
张量([[0.3113,         0.3934, 0.2952]]))
```

这些值现在可以解释为类成员概率，其总和为 1。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.

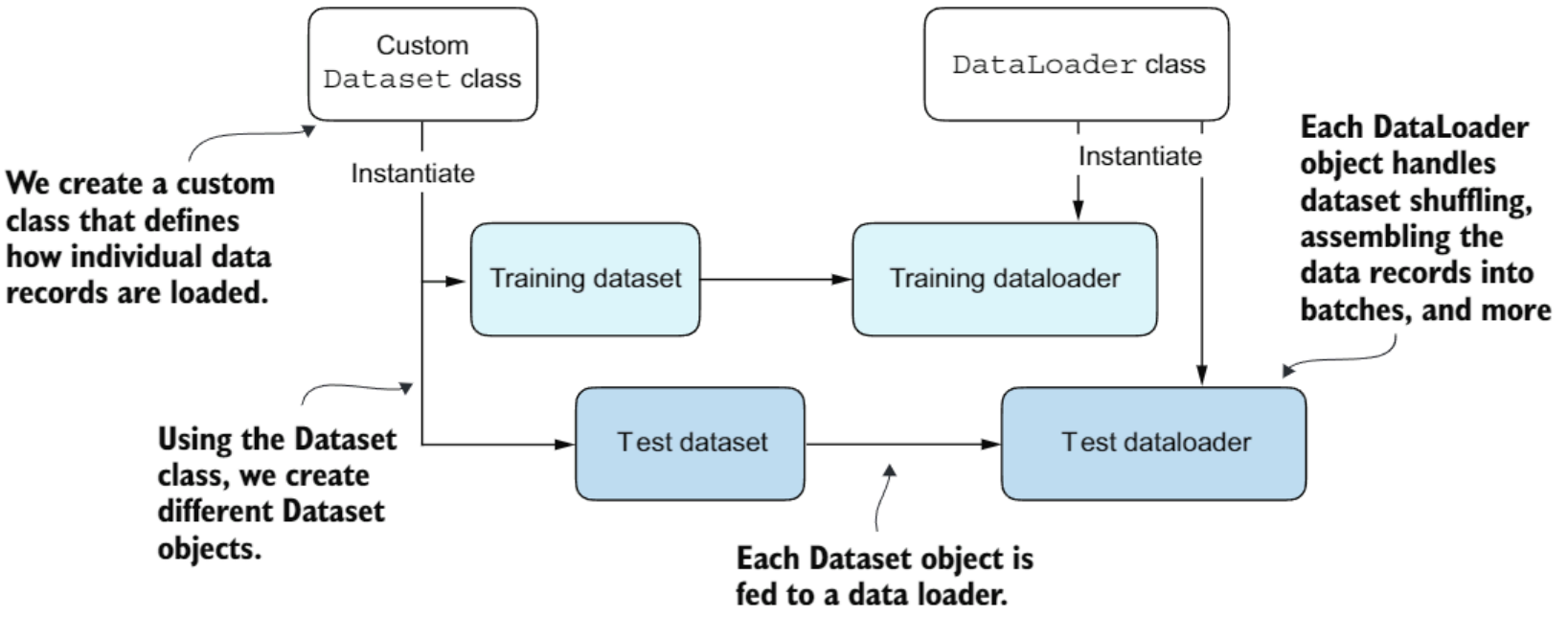


Figure A.10 PyTorch implements a `Dataset` and a `DataLoader` class. The `Dataset` class is used to instantiate objects that define how each data record is loaded. The `DataLoader` handles how the data is shuffled and assembled into batches.

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 设置高效的数据加载器

在我们可以训练我们的模型之前，我们必须简要讨论在 PyTorch 中创建高效的数据加载器，我们将在训练过程中迭代它。PyTorch 中数据加载的整体思想如图 A.10 所示。

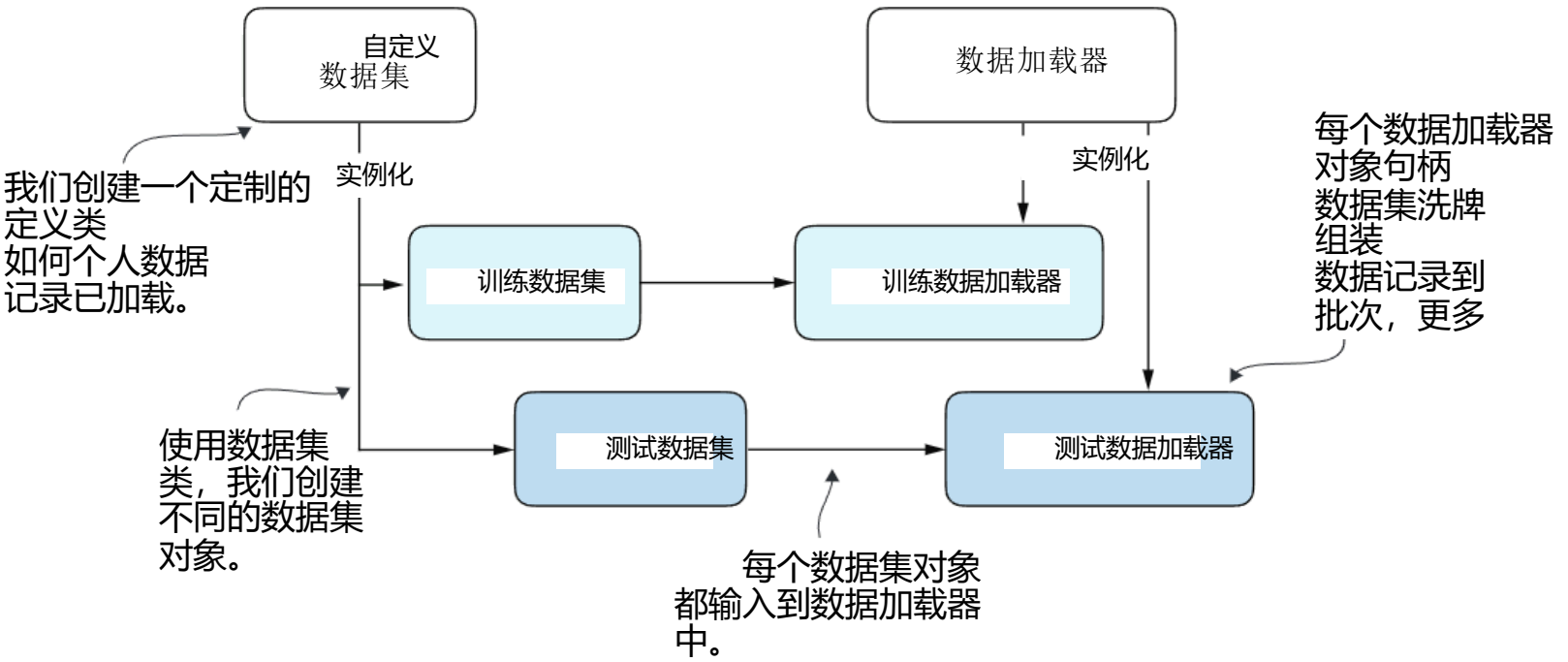


图 A.10 PyTorch 实现了 Dataset 和 DataLoader 类。Dataset 类用于实例化对象，定义如何加载每个数据记录。DataLoader 处理数据的打乱和组装成批处理。

根据图 A.10，我们将实现一个自定义的 Dataset 类，我们将使用它来创建一个训练集和一个测试集，然后我们将使用这些数据集来创建数据加载器。让我们先创建一个包含五个训练示例的简单玩具数据集，每个示例有两个特征。与训练示例一起，我们还创建了一个包含相应类别标签的张量：三个示例属于类别 0，两个示例属于类别 1。此外，我们还制作了一个包含两个条目的测试集。创建此数据集的代码如下所示。

列表 A.5 创建一个小型玩具数据集

```
    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])                                          0, 1, 1]

    X_test = torch.tensor([
         [-0.8, 2.8]
         [2.6, -1.6]
    ])
    y_test   = torch.tensor([0, 1])
预测结果
```

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 要求类标签从 0 开始，最大的类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，如果我们有类标签 0、1、2、3 和 4，神经网络输出层应包含五个节点。

接下来，我们通过从 PyTorch 的 Dataset 父类派生来创建一个自定义数据集类，名为 ToyDataset，如下所示。

列表 A.6 定义一个自定义数据集类

```
from  torch.utils.data 导入 Dataset

class 玩具数据集(Dataset):
     def __init__(self, X, y):
          self.features = X
          self.labels = y

     def __getitem__(self, 索引):
          one_x = self.features[index]
      one_y = self.labels[index] 返回
      one_x, one_y

     def __len__(self): # 获取长度
          返回 self.labels 形状的 0 维长度

train_ds = 玩具数据集(X_train, y_train)
test_ds = 玩具数据集(X_test, y_test)
```

检索一条数据记录及其对应标签的说明

使用说明返回数据集的总长度

本自定义 ToyDataset 类的目的是实例化一个 PyTorch DataLoader。但在我们到达这一步之前，让我们简要地回顾一下其一般结构。

玩具数据集代码。

PyTorch 中，自定义 Dataset 类的三个主要组件是

构造函数 __init__，__getitem__ 方法，以及 __len__ 方法（参见列表 A.6）。在 __init__ 方法中，我们设置可以在后续的 __getitem__ 和 __len__ 方法中访问的属性。这些可以是文件路径、文件对象、数据库连接器等等。由于我们创建了一个驻留在内存中的张量数据集，我们只需将这些属性分配给 X 和 y，它们是我们张量对象的占位符。

在 __getitem__ 方法中，我们定义了通过索引从数据集中返回恰好一个项目的指令。这指的是与单个训练示例或测试实例对应的特征和类标签。（数据加载器将提供这个索引，我们将在稍后介绍。）

最后，__len__方法包含获取数据集长度的指令。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，我们可以进行双重检查：

打印(train_ds).长度

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
)
```

The ToyDataset instance created earlier serves as input to the data loader.

Whether or not to shuffle the data

The number of background processes

It is not necessary to shuffle a test dataset.

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，如下所示。

列表 A.7 实例化数据加载器

```
from torch.utils.data 导入 DataLoader


torch 手动设置随机种子(123)

train_loader = DataLoader(
    数据集=train_ds,
    batch_size=2,
    shuffle=True,
    num_workers=0
)

test_loader = DataLoader(
    dataset=test_ds,
    batch_size=2,
    shuffle=False,
    num_workers=0
)
```

早期创建的 ToyDataset 实例作为数据加载器的输入。

是否打乱数据

背景进程数量

无需洗牌测试数据集。

在实例化训练数据加载器后，我们可以遍历它。遍历 test_loader 的方式类似，但为了简洁起见省略了。

```
for idx, (x, y) 在 enumerate(train_loader) 中
    打印(f"批量")    {idx+1}:", x,    y)
```

结果是

```
批量    张量([[-1.2000,            3.1000]
                    [-0.5000]    2.6000]]) 张量([0,])      0])
批量    2: tensor([[ 2.3000, -1.1000],
                    [-0.9000]    2.9000]]) 张量([1,])      0])
批量    3: 张量([[ 2.7000, -1.5000]]) 张量([1])
```

根据前面的输出，我们可以看到 train_loader 遍历训练数据集，每个训练示例恰好访问一次。这被称为一个训练周期。由于我们在这里使用 torch.manual_seed(123)初始化了随机数生成器，你应该得到相同的训练示例打乱顺序。然而，如果你第二次遍历数据集，你会看到打乱顺序会改变。这是为了防止深度神经网络在训练过程中陷入重复的更新循环。

我们在这里指定了批量大小为 2，但第三个批量只包含一个示例。这是因为我们有五个训练示例，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

实际上，在训练周期中，将一个显著较小的批次作为最后一个批次可能会干扰训练过程中的收敛。为了防止这种情况，设置 drop_last=True，这将丢弃每个周期中的最后一个批次，如下所示列表。

列表 A.8 训练加载器丢弃最后一个批次

```
train_loader = DataLoader(
    数据集=train_ds,
    batch_size=2,
    shuffle=True,
    num_workers=0,
    drop_last=True
)删除最后
```

现在，遍历训练加载器时，我们可以看到最后一个批次被省略了：

```
for idx, (x, y) in enumerate(训练加载器):
    打印(f"批次 {idx+1}:", x, y)
```

结果是

```
批次 1：tensor([[-0.9000,          2.9000]
         [ 2.3000, -1.1000]]) 张量([0,          1])
批次 2：张量([[         [2.7000, -1.5000]
         [-0.5000]     2.6000]]) 张量([1,])      0])
```

最后，让我们讨论一下在 DataLoader 中设置 num_workers=0 的情况。在 PyTorch 的 DataLoader 函数中，此参数对于并行化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。这看起来可能没有问题，但当我们在 GPU 上训练更大的网络时，这可能会导致模型训练期间出现显著的减速。在这种情况下，CPU 必须花费时间来加载和预处理数据，而不仅仅是关注深度学习模型的处理。因此，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以并行加载数据，从而释放主进程专注于训练您的模型，并更好地利用系统资源（图 A.11）。

然而，如果我们正在处理非常小的数据集，设置 num_workers 为 1 或更大可能并不必要，因为总训练时间只需几秒钟。因此，如果您正在处理小型数据集或交互式环境，如 Jupyter 笔记本，增加 num_workers 可能不会提供任何明显的加速。实际上，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，当您的数据集较小时，这可能会比实际的数据加载时间更长。

此外，对于 Jupyter 笔记本，将 num_workers 设置大于 0 有时会导致不同进程之间资源共享相关的问题，从而引发错误或笔记本崩溃。因此，了解

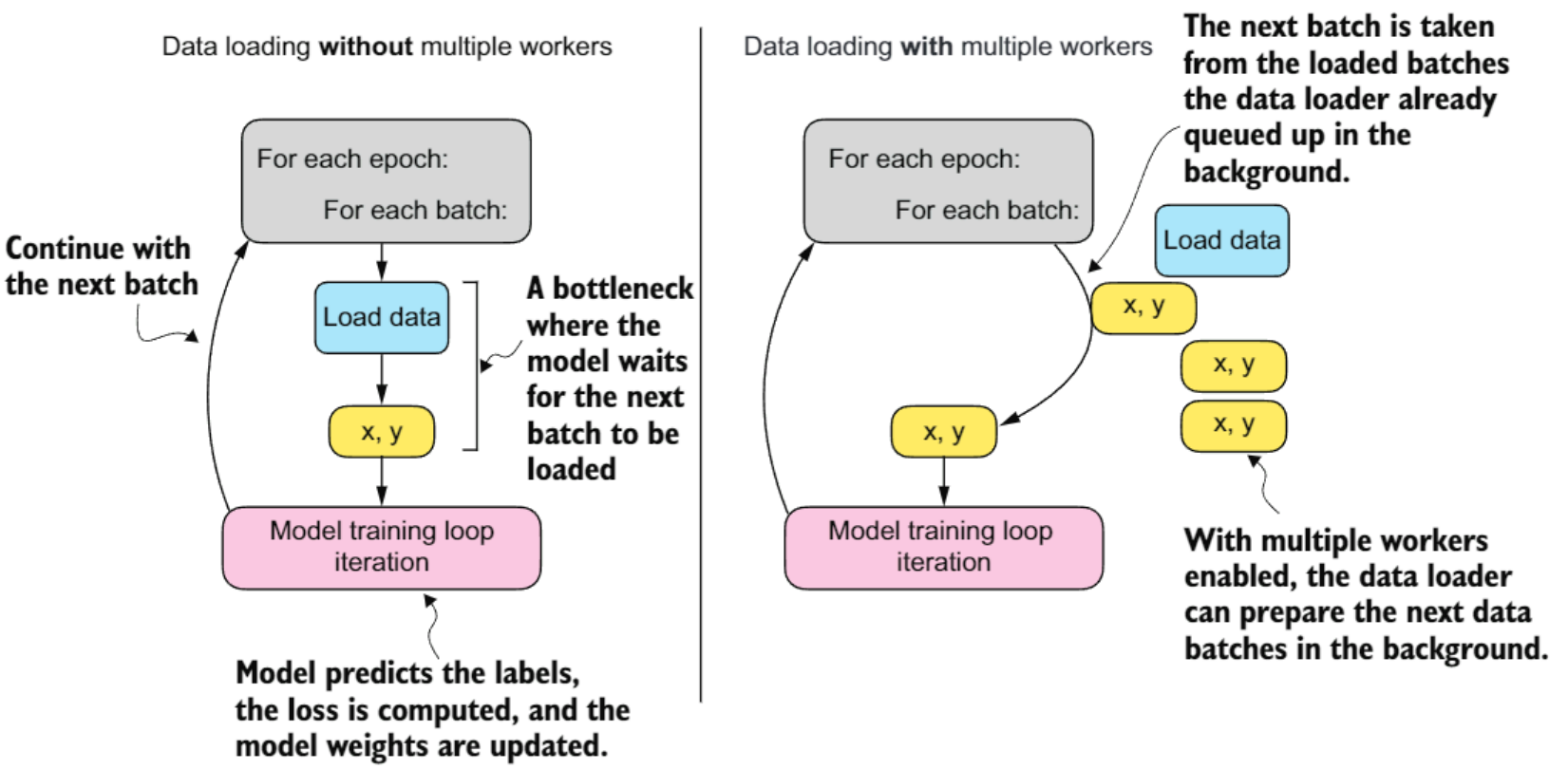


Figure A.11 Loading data without multiple workers (setting `num_workers=0`) will create a data loading bottleneck where the model sits idle until the next batch is loaded (left). If multiple workers are enabled, the data loader can queue up the next batch in the background (right).

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 dataset has two features and two classes.

The optimizer needs to know which parameters to optimize.

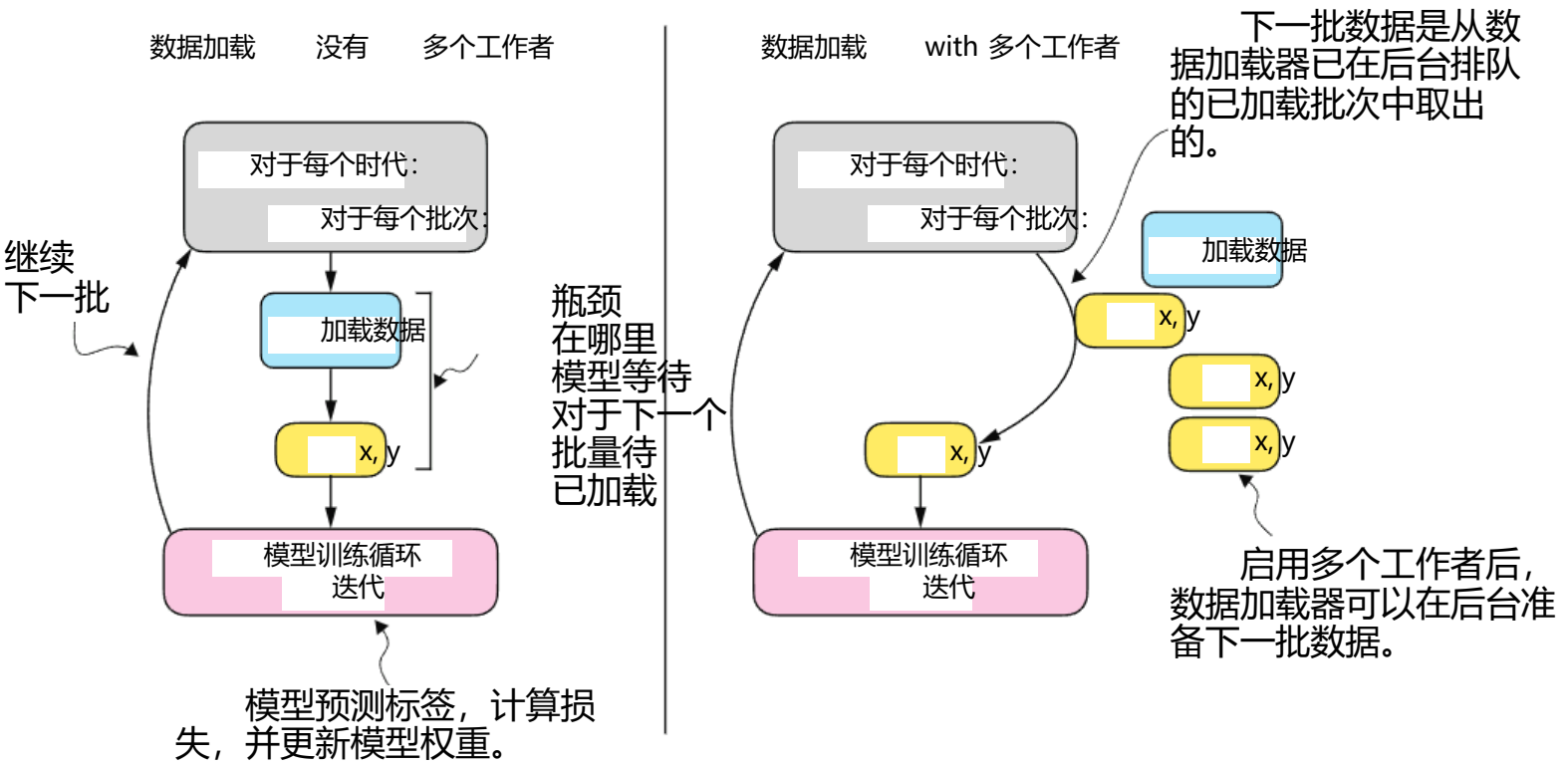


图 A.11 不使用多个工作者（设置 num_workers=0）加载数据将创建一个数据加载瓶颈，模型在此空闲，直到加载下一个批次（左侧）。如果启用多个工作者，数据加载器可以在后台排队下一个批次（右侧）。

权衡利弊，并计算决定设置 num_workers 参数。当正确使用时，它可能是一个有益的工具，但应适应您特定的数据集大小和计算环境以获得最佳结果。

在我的经验中，将 num_workers 设置为 4 通常在许多实际数据集上能带来最佳性能，但最佳设置取决于您的硬件以及用于加载 Dataset 类中定义的训练示例的代码。

A.7 一个典型的训练循环

现在让我们在玩具数据集上训练一个神经网络。以下列表显示了训练代码。

列表 A.9 神经网络在 PyTorch 中的训练

```
导入 torch.nn.functional 作为 F

torch.manual_seed(123) 模型 = 神经网络(num_inputs=2,
num_outputs=2) 优化器 = torch.optim.SGD(

model.parameters(), lr=0.5 )


num_epochs = 3
for epoch 在 range(num_epochs) 范围内:

    model.train()
```

数据集有两个功能和两个类

优化器需要知道要优化哪些参数。

```
for batch_idx, (features, labels) in enumerate(train_loader):
    logits = model(features)

    loss = F.cross_entropy(logits, labels)

    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 Loss: {loss:.2f}")

model.eval()
# Insert optional model evaluation code
```

Sets the gradients from the previous round to 0 to prevent unintended gradient accumulation

Computes the gradients of the loss given the model parameters

The optimizer uses the gradients to update the model parameters.

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 | Trainl 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 = 逻辑模型(特征)

    损失 = F.cross_entropy(logits, labels)

    optimizer.zero_grad()
梯度清零() loss.backward()
反向传播()
optimizer.step() 更新参数

    ### 记录打印(f"Epoch:
{epoch+1:03d}/{num_epochs:03d}")
          " | 批次 {batch_idx:03d}/{len(train_loader):03d} | 训练损
      失: {loss:.2f}"

model.eval() # 插入可选模
型
```

将上一轮的梯度设置为 0 以防止意外梯度累积

优化器使用梯度来更新模型参数。

计算梯度关于损失鉴于模型参数

评估代码

运行此代码将产生以下输出：

```
第 1 个周期：001/003 | 批次 000/002 | 训练损失：0.75 第 1
个周期：001/003 | 批次 001/002 | 训练损失：0.65 第 2 个周期：
002/003 | 批次 000/002 | 训练损失：0.44 第 2 个周期：002/003
| 批次 001/002 | 训练损失：0.13 第 3 个周期：003/003 | 批次
000/002 | 训练损失：0.03 第 3 个周期：003/003 | 批次 001/002
| 训练损失：0.00
```

如我们所见，损失在三个 epoch 后达到 0，这是模型在训练集上收敛的标志。在这里，我们用一个有两个输入和两个输出的模型进行初始化，因为我们的玩具数据集有两个输入特征和两个类别标签需要预测。我们使用了一个学习率（lr）为 0.5 的随机梯度下降（SGD）优化器。学习率是一个超参数，意味着它是一个可调整的设置，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，使得损失在经过一定数量的 epoch 后收敛——epoch 的数量是另一个需要选择的超参数。

练习 A.3

神经网络在列表 A.9 中引入了多少个参数?

在实践中，我们经常使用第三个数据集，即所谓的验证数据集，以找到最佳的超参数设置。验证数据集类似于测试集。然而，我们只想精确使用一次测试集以避免对评估造成偏差，而我们通常多次使用验证集来调整模型设置。

我们也引入了名为 model.train()和 model.eval()的新设置。正如这些名称所暗示的，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表现不同的组件（如 dropout 或批归一化层）是必要的。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

或我们的 NeuralNetwork 类中受这些设置影响的其他组件，使用 model.train()和 model.eval()在我们的前代码中是多余的。然而，为了防止在更改模型架构或重用代码来训练不同模型时出现意外行为，最好还是包括它们。

如前所述，我们直接将 logits 传递给交叉熵损失函数，该函数将内部应用 softmax 函数以提高效率和数值稳定性。然后，调用 loss.backward()将计算 PyTorch 在后台构建的计算图中的梯度。optimizer.step()方法将使用梯度来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后的负梯度加到参数上。

注意：为了防止梯度累积，在每次更新轮次中包含 optimizer.zero_grad()调用以将梯度重置为 0 是很重要的。否则，梯度将累积，这可能是不可取的。

在训练好模型后，我们可以用它来进行预测：

```
    model.eval() 使用
torch.no_grad():
    outputs = 模型(X_train) 打
印(outputs)
```

结果为

```
    张量([[ 2.8569,       -4.1618]
           [2.5382, -3.7548],
      [2.0944, -3.1820],
      [-1.4814, 1.4816],
      [-1.7176,]
                          1.7342]])
```

获取类别成员概率后，我们可以使用 PyTorch 的 softmax 函数：

```
    torch.set_printoptions(sci_mode=False)
probas = torch.softmax(outputs, dim=1) 打印
(probas)
```

这输出

```
    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（列）表示训练示例有 99.91%的概率属于该类别

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.

91%属于类别 0，0.09%属于类别 1。（在这里使用 set_printoptions 调用是为了使输出更易读。）我们可以使用 PyTorch 的 argmax 函数将这些值转换为类别标签预测，该函数返回每行中最高值的索引位置，如果我们设置 dim=1（设置 dim=0 将返回每列中的最高值）：

```
   predictions = torch.argmax(probas, dim=1) 打印
(predictions)
```

这会打印

```
张量([0, 0, 0]) 0, 1, 1]]
```

注意，为了获得类别标签，无需计算 softmax 概率。我们也可以直接对 logits（输出）应用 argmax 函数：

```
   predictions = torch.argmax(outputs, dim=1) 打印
(predictions)
```

输出结果

```
张量([0, 0, 0]) 0, 1, 1]]
```

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，我们可以通过肉眼与真实训练标签进行比较，看到模型 100%正确。我们可以使用比较运算符==来再次检查：

```
预测 y_train
```

结果为

```
张量([True, 确实，确实， 确实，确实]]
```

使用 torch.sum，我们可以计算正确预测的数量：

```
torch.sum(predictions == y_train)
```

输出结果

```
5
```

由于数据集包含五个训练示例，我们有五个预测是正确的，这对应着 5/5 × 100% = 100%的预测准确率。

为了泛化预测精度的计算，让我们实现一个 compute_accuracy 函数，如下所示列表。

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")
```

列表 A.10 一个计算预测准确率的函数

```
def 计算模型准确度                    数据加载器):

    model = model.eval()
正确率 = 0.0 总样本数 = 0


    for idx, (特征, 标签)              in enumerate(dataloader): 遍历数据加载器

        使用 torch.no_grad():
            logits = 逻辑模型(特征)


        predictions = torch.argmax(logits, dim=1)
    compare = labels == predictions 正确 +=
    torch.sum(compare) 总示例数 += len(compare)


    返回     (正确 / 总示例).item()
```

返回一个根据标签是否匹配返回 True/False 值的张量

求和操作计算 True 值的数量。

正确预测的分数，介于 0 和 1 之间的值。.item() 返回张量的值作为 Python 浮点数。

代码遍历数据加载器以计算正确预测的数量和比例。当我们处理大型数据集时，通常由于内存限制，我们只能调用模型的数据集的一小部分。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量大小相同。compute_accuracy 函数的内部结构与我们在将 logits 转换为类别标签时使用的方法类似。

我们可以将函数应用于训练：

```
打印（计算准确率（模型，              train_loader))
```

结果是

```
1.0
```

同样，我们可以将函数应用于测试集：

```
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

现在我们已经训练了我们的模型，让我们看看如何保存它，以便我们以后可以重用。以下是我们在 PyTorch 中保存和加载模型的推荐方法：

```
torch.save(model.state_dict(), "model.pth") 保存模型状态字典到 "model.pth" 文件中
```

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，它将模型中的每一层映射到其可训练参数（权重和偏置）。"model.pth"是保存到磁盘上的模型文件的任意文件名。我们可以给它任何名字和文件扩展名；然而，.pth 和.pt 是最常见的约定。

一旦我们保存了模型，就可以从磁盘恢复它：

```
    model = 神经网络(2, 2)
model.load_state_dict(torch.load("model.pth"))
```

torch.load("model.pth")函数读取文件 "model.pth" 和 recon-
结构体 the Python 词典 对象 包含 the 模型 参数 while

model.load_state_dict() 将这些参数应用于模型，实际上是从我们保存它的那一刻恢复了其学习状态。

该行模型 = NeuralNetwork(2, 2)在您在同一会话中执行保存模型的代码时不是严格必要的。然而，我将其包含在这里是为了说明我们需要在内存中有一个模型实例来应用保存的参数。在这里，NeuralNetwork(2, 2)架构需要与原始保存的模型完全匹配。

A.9 优化使用 GPU 的训练性能

接下来，我们将探讨如何利用 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 计算背后的主要概念。然后，我们将在单个 GPU 上训练一个模型。最后，我们将探讨使用多个 GPU 进行分布式训练。

A.9.1 PyTorch 在 GPU 设备上的计算

修改训练循环以可选地在 GPU 上运行相对简单，只需更改三行代码（见 A.7 节）。在我们进行修改之前，理解 PyTorch 中 GPU 计算背后的主要概念至关重要。在 PyTorch 中，设备是计算和数据驻留的地方。CPU 和 GPU 是设备的例子。PyTorch 张量驻留在设备中，其操作在相同的设备上执行。

让我们看看这个在实际操作中是如何工作的。假设您已安装了与 GPU 兼容的 PyTorch 版本（见 A.1.3 节），我们可以通过以下代码来验证我们的运行时确实支持 GPU 计算：

```
打印 torch.cuda.is_available()
```

结果是

```
True
```

现在，假设我们有两个可以相加的张量；这个计算默认将在 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.

```
    tensor_1 = torch.tensor([1., 2., 3.])
tensor_2 = torch.tensor([4., 5., 6.]) 打印
(tensor_1 + tensor_2)
```

这输出

```
张量([5., 7.,])    9.])
```

我们现在可以使用.to()方法。这个方法和我们用来改变张量数据类型（见2.2.2）的方法相同，用于将这些张量传输到 GPU 上并在那里执行加法操作：

```
    tensor_1 = tensor_1.to("cuda")
tensor_2 = tensor_2.to("cuda") 打印
(tensor_1 + tensor_2)
```

输出结果

```
张量([5., 7.,])    9.],  device='cuda:0'
```

结果张量现在包括设备信息，device='cuda:0'，这意味着张量位于第一个 GPU 上。如果你的机器有多个 GPU，你可以指定要将张量传输到哪个 GPU。你通过在传输命令中指定设备 ID 来实现这一点。例如，你可以使用.to("cuda:0")

```
.to("cuda:1"), 等等。
```

然而，所有张量必须在同一设备上。否则，计算将失败，其中一个张量位于 CPU 上，另一个位于 GPU 上：

```
    tensor_1 = tensor_1.to("cpu") 打
印(tensor_1 + tensor_2)
```

结果为

```
运行时错误 追踪回溯（最后最近调用） 在 ()

      1 tensor_1 = tensor_1 转移到("cpu")
请提供需要翻译的文本
运行时错误：期望所有张量都在同一设备上，但至少有两个设备，cuda:0 和 cpu!
```

总的来说，我们只需将张量转移到同一 GPU 设备上，PyTorch 会处理其余部分。

A.9.2 单 GPU 训练

现在我们已经熟悉了将张量传输到 GPU 的过程，我们可以修改训练循环以在 GPU 上运行。这一步只需要更改三行代码，如下所示列表所示。

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)

optimizer = torch.optim.SGD(model.parameters(), lr=0.5)

num_epochs = 3

for epoch in range(num_epochs):

    model.train()
    for batch_idx, (features, labels) in enumerate(train_loader):
        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()
    # Insert optional model evaluation code
```

Defines a device variable that defaults to a GPU

Transfers the model onto the GPU

Transfers the data 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.

列表 A.11 GPU 上的训练循环

```
    torch 手动设置随机种子(123) model = 神
经网络(num_inputs=2,                                    num_outputs=2)

    device = torch.device("cuda") 模
型 = 模型.to(device)

    优化器       =  torch.optim.SGD（模型参数，学习率=0.5）

    num_epochs = 3

num_epochs = 3
    for epoch 在 range(num_epochs):

        model.train()
        for batch_idx, (features, labels) in enumerate(train_loader):
            features, labels = features.to(设备), labels.to(设备) logits = 模
         型(features) 损失 = F.cross_entropy(logits, labels) # 损失函数


            optimizer.zero_grad()
         梯度清零() loss.backward()
         反向传播()
         optimizer.step() 更新参数

            ### 记录打印(f"Epoch:
         {epoch+1:03d}/{num_epochs:03d}")
                    " | 批次 {batch_idx:03d}/{len(train_loader):03d} | 训练/验
               证损失: {loss:.2f}"
        model.eval() # 插入可选模
    型
                                                评估代码
```

定义一个默认为 GPU 的设备变量

转移模型进入 GPU

将数据传输到 GPU

运行前面的代码将输出以下内容，类似于在 CPU 上获得的结果（第 A.7 节）：

```
    第 1/3 个周期 | 第 000/002 个批次 | 训练/验证损失：0.75 第
1/3 个周期 | 第 001/002 个批次 | 训练/验证损失：0.65 第 2/3 个周
期 | 第 000/002 个批次 | 训练/验证损失：0.44 第 2/3 个周期 | 第
001/002 个批次 | 训练/验证损失：0.13 第 3/3 个周期 | 第 000/002
个批次 | 训练/验证损失：0.03 第 3/3 个周期 | 第 001/002 个批次 |
训练/验证损失：0.00
```

我们可以使用.to("cuda")代替 device = torch.device("cuda")。转移

张量转换为"cuda"代替 torch.device("cuda")同样有效且更简洁（见 A.9.1 节）。我们还可以修改语句，使代码在没有 GPU 的情况下也能在 CPU 上执行。这是共享 PyTorch 代码的最佳实践：

```
    设备     = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        = 火炬设备("cuda" 如果 torch.cuda.is_available() 否则 "cpu")
```

在这种情况下，由于 CPU 到 GPU 的内存传输成本，我们可能不会看到速度提升。然而，在训练深度神经网络时，我们可以期待一个显著的速度提升，尤其是在LLMs时。

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

PyTorch 在 macOS 上

在搭载苹果硅芯片的苹果 Mac（如 M1、M2、M3 或更新的型号）上，而不是搭载 Nvidia GPU 的电脑上，您可以更改

```
device = torch.device("cuda" if torch.cuda.is_available() 否则 CPU
```

to

```
device = torch.device()
"mps" if torch.backends.mps.可用() else "cpu" )
```

利用这款芯片。

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，你开始看到 GPU 上的矩阵乘法比 CPU 快？提示：使用 Jupyter 中的%timeit 命令来比较运行时间。例如，给定矩阵 a 和 b，在新笔记本单元中运行命令%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过程分配到多个 GPU 和机器上的概念。为什么需要这样做呢？即使可以在单个 GPU 或机器上训练模型，这个过程也可能非常耗时。通过将训练过程分配到多个机器上，每个机器上可能有多个 GPU，可以显著减少训练时间。这在模型开发的实验阶段尤为重要，在这个阶段可能需要进行多次训练迭代来微调模型参数和架构。

注意：对于本书，不需要访问或使用多个 GPU。本节包含给那些对 PyTorch 中多 GPU 计算工作原理感兴趣的人。

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的设备上，并同时处理这些数据子集来实现并行化。

如何工作？PyTorch 在每个 GPU 上启动一个单独的进程，每个进程接收并保留模型的一个副本，这些副本将在训练过程中同步。为了说明这一点，假设我们有两个 GPU，我们想使用它们来训练一个神经网络，如图 A.12 所示。

每个 GPU 都将收到模型的一个副本。然后，在每次训练迭代中，每个模型将从数据加载器接收一个 minibatch（或简称“batch”）。我们

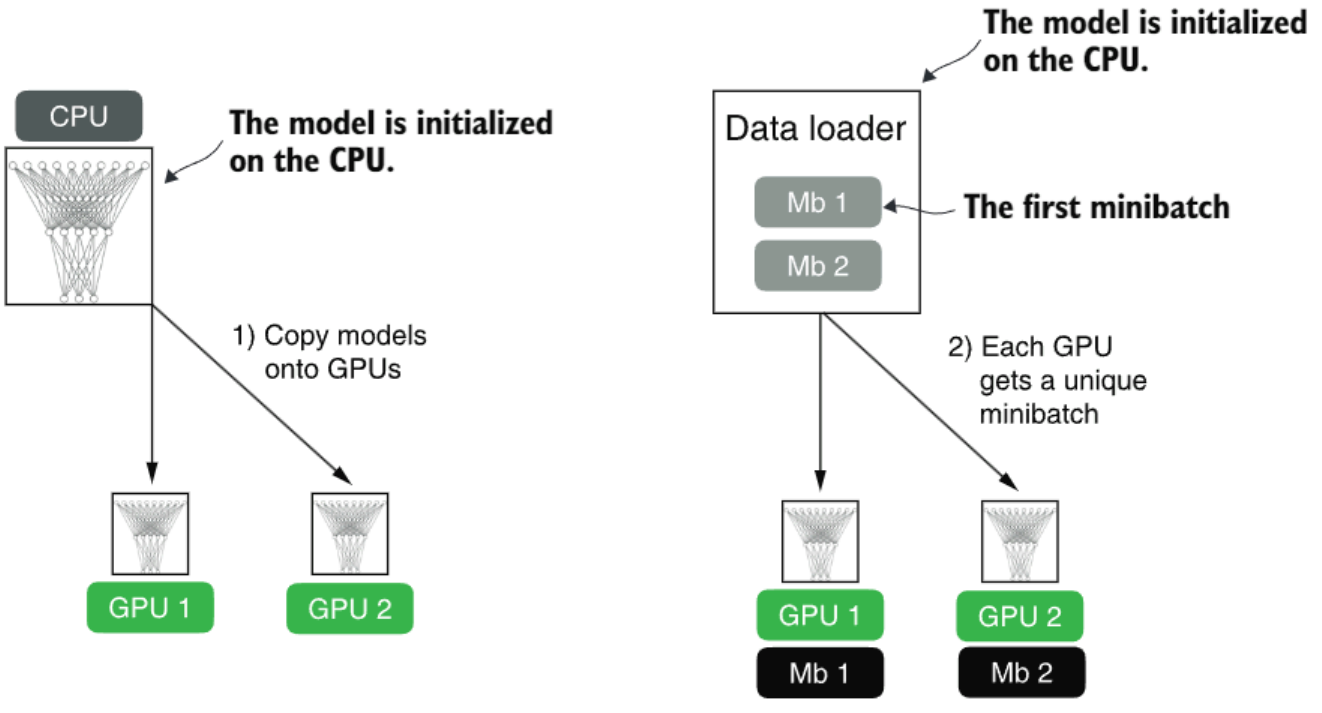


Figure A.12 The model and data transfer in DDP involves two key steps. First, we create a copy of the model on each of the GPUs. Then we divide the input data into unique minibatches that we pass on to each model copy.

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.

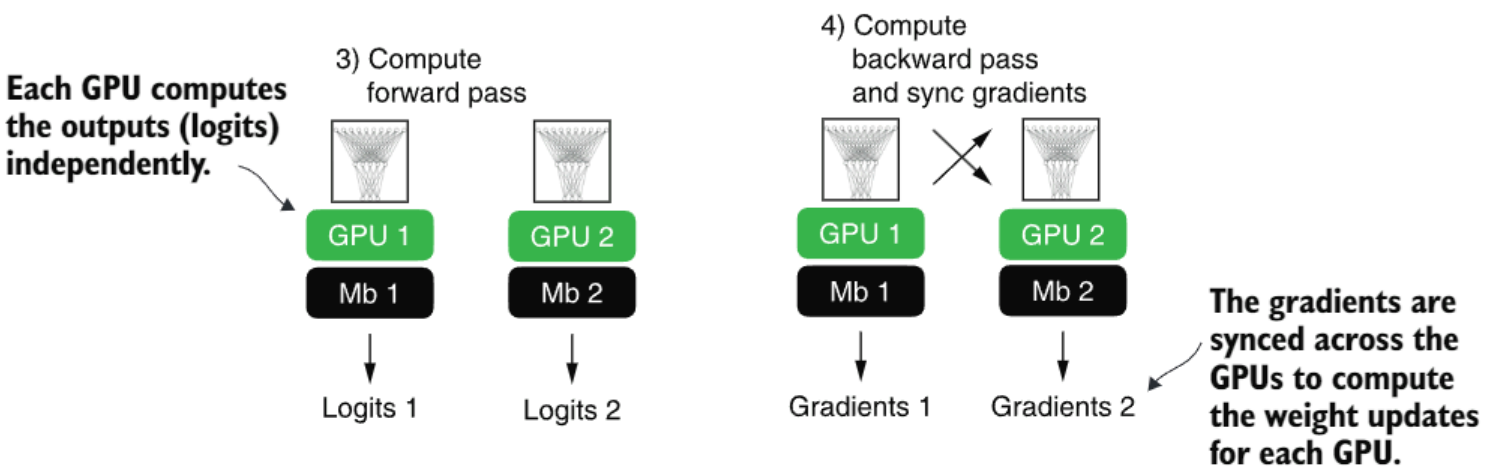


Figure A.13 The forward and backward passes in DDP are executed independently on each GPU with its corresponding data subset. Once the forward and backward passes are completed, gradients from each model replica (on each GPU) are synchronized across all GPUs. This ensures that every model replica has 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

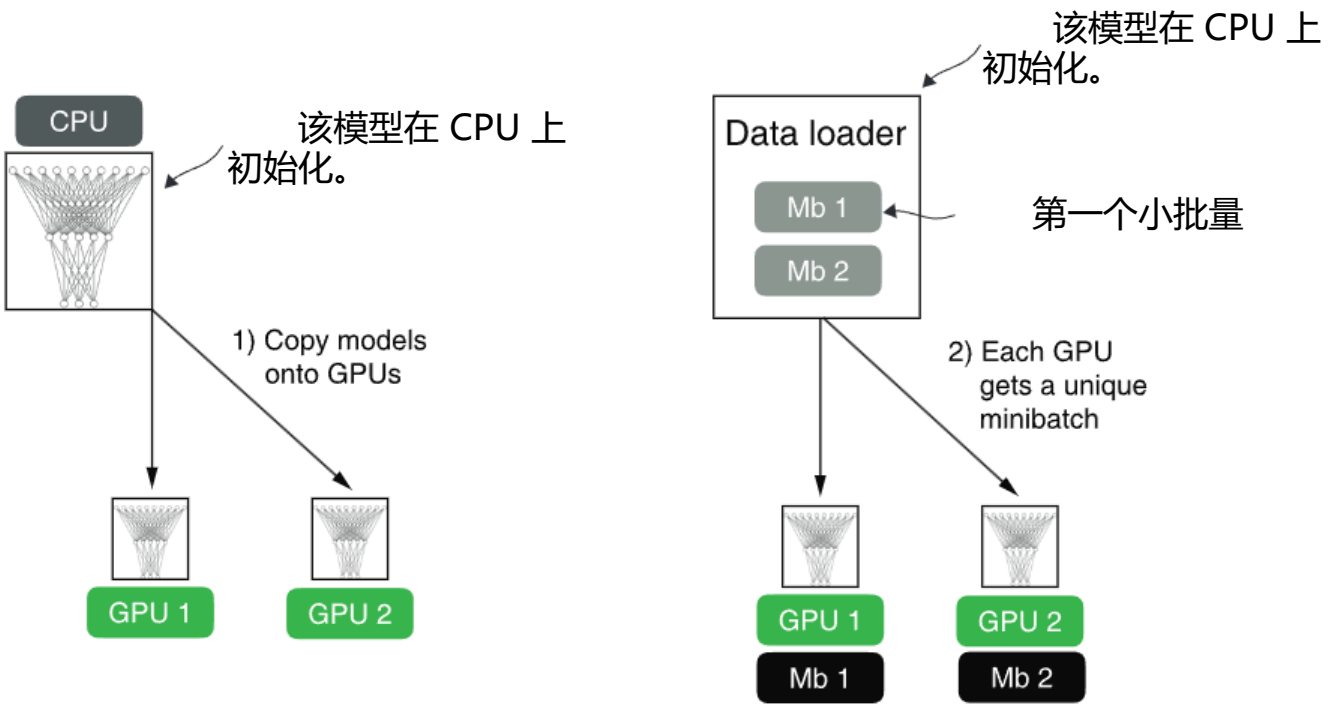


图 A.12 DDP 中的模型和数据传输涉及两个关键步骤。首先，我们在每个 GPU 上创建模型的一个副本。然后，我们将输入数据划分为唯一的 minibatch，并将它们传递给每个模型副本。

我们可以使用 DistributedSampler 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，模型副本将返回不同的 logits 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过程中平均并同步，以更新模型。这样，我们确保模型不会发散，如图 A.13 所示。

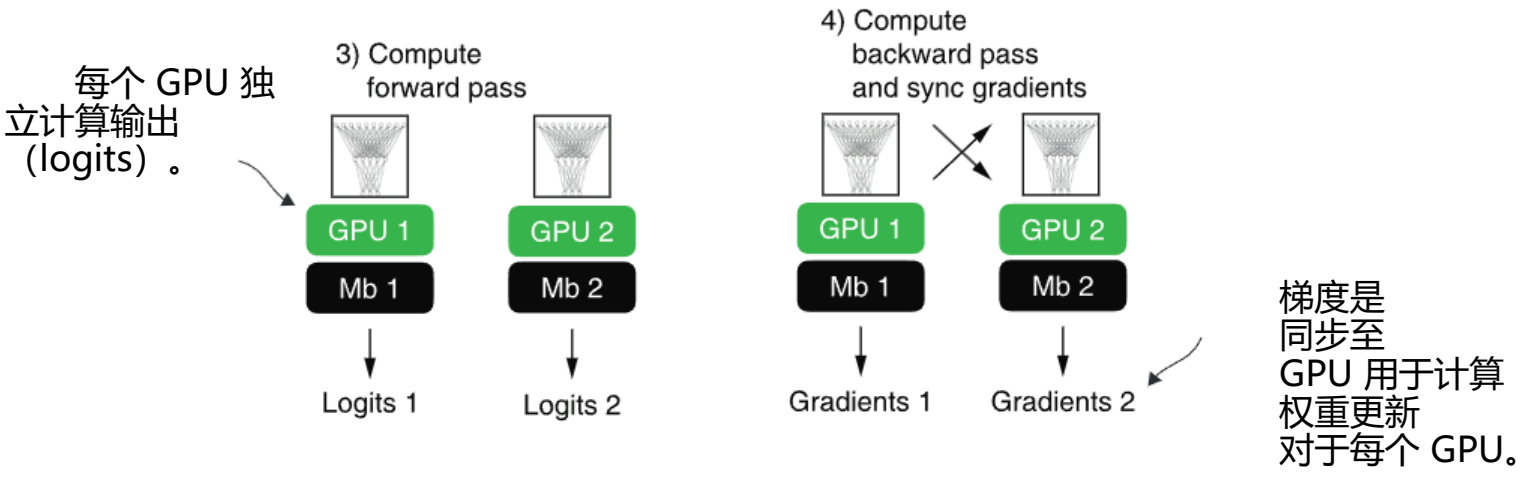


图 A.13 DDP 中，前向和反向传播在每个 GPU 上独立执行，对应其对应的数据子集。一旦前向和反向传播完成，每个模型副本（每个 GPU）的梯度将在所有 GPU 之间同步。这确保了每个模型副本都有相同的更新权重。

使用 DDP 的好处是，与单个 GPU 相比，它为处理数据集提供了更高的速度。除了设备之间微小的通信开销之外，

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 mods. 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

除了使用 DDP 带来的轻微通信开销外，理论上使用两个 GPU 可以在一半的时间内处理一个训练周期，与仅使用一个 GPU 相比。时间效率随着 GPU 数量的增加而提高，如果我们有八个 GPU，那么我们可以将一个周期处理速度提高八倍，依此类推。

注意：DDP 在像 Jupyter 笔记本这样的交互式 Python 环境中无法正常工作，因为这些环境没有像独立 Python 脚本那样处理多进程。因此，以下代码应作为脚本执行，而不是在 Jupyter 这样的笔记本界面中。DDP 需要生成多个进程，并且每个进程都应该有自己的 Python 解释器实例。

现在让我们看看这在实践中是如何工作的。为了简洁，我专注于需要调整以进行 DDP 训练的代码的核心部分。然而，想要在自己的多 GPU 机器或选择的云实例上运行代码的读者应使用本书 GitHub 仓库中提供的独立脚本，网址为 https://github.com/rasbt/LLMs-from-scratch。

首先，我们导入一些用于 PyTorch 分布式训练的额外子模块、类和函数，如下所示列表所示。

列表 A.12 PyTorch 分布式训练工具

```
    import torch.multiprocessing as mp
从 torch.utils.data.distributed 导入 DistributedSampler
从 torch.nn.parallel 导入 DistributedDataParallel as DDP
从 torch.distributed 导入
                                                初始化进程组，销毁进程组
```

在深入探讨使训练与 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工具的原理和用法，这些工具是我们与 DistributedDataParallel 类一起需要的。

PyTorch 的 multiprocessing 子模块包含诸如 multiprocessing 之类的功能.spawn，我们将用它来创建多个进程并将函数并行应用于多个输入。我们将用它为每个 GPU 创建一个训练进程。如果我们为训练创建了多个进程，我们需要一种方法将这些不同的进程分配到数据集。为此，我们将使用 DistributedSampler。

and初始化进程组process_group 用于初始化和退出分布式训练模式。init_process_group 函数应在训练脚本开始时调用，以初始化分布式设置中每个进程的进程组，destroy_process_group 应在训练脚本结束时调用，以销毁指定的进程组并释放其资源。以下列表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 NeuralNetwork 模型的 DDP 训练。

列表 A.13 模型训练采用 DistributedDataParallel 策略

```
def ddp_setup(排名, 世界大小):
    os.environ["MASTER_ADDR"]      = localhost
```

主节点地址

```
    os.environ["MASTER_PORT"] = "12345"
    init_process_group(
        backend="nccl",
        rank=rank,
        world_size=world_size
    )
    torch.cuda.set_device(rank)

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

def main(rank, world_size, num_epochs):
    ddp_setup(rank, world_size)
    train_loader, test_loader = prepare_dataset()
    model = NeuralNetwork(num_inputs=2, num_outputs=2)
    model.to(rank)
    optimizer = torch.optim.SGD(model.parameters(), lr=0.5)
    model = DDP(model, device_ids=[rank])
    for epoch in range(num_epochs):
    for features, labels in train_loader:
            features, labels = features.to(rank), labels.to(rank)
            # insert model prediction and backpropagation code
            print(f"[GPU{rank}] Epoch: {epoch+1:03d}/{num_epochs:03d}"
                  f" | Batchsize {labels.shape[0]:03d}"
                  f" | Train/Val Loss: {loss:.2f}")

    model.eval()
    train_acc = compute_accuracy(model, train_loader, device=rank)
    print(f"[GPU{rank}] Training accuracy", train_acc)
    test_acc = compute_accuracy(model, test_loader, device=rank)
    print(f"[GPU{rank}] Test accuracy", test_acc)
    destroy_process_group()

if __name__ == "__main__":
    print("Number of GPUs available:", torch.cuda.device_count())
    torch.manual_seed(123)
    num_epochs = 3
    world_size = torch.cuda.device_count()
    mp.spawn(main, args=(world_size, num_epochs), nprocs=world_size)
```

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

Distibuted-Sampler takes care of the shuffling now.

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

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" 初
始化进程组(
        backend="nccl"
    rank=rank,
world_size=world_size)
torch.cuda.set_device(rank)
```

世界大小
这是
数量
GPU
use.

任何空闲端口
在机器上

nccl 代表 NVIDIA 集体通信库。

GPU 索引指的是我们想要使用的 GPU 的索引。

设置当前 GPU 设备，张量将在其上分配并执行操作

```
准备数据集()
    # 插入数据集准备代码 train_loader =
DataLoader(
        数据集=train_ds,
        batch_size=2,
        shuffle=False,
        pin_memory=True
    drop_last=True,
sampler=DistributedSampler(train_ds) 返回
train_loader, test_loader
```

分布式
采样器
照顾
洗牌
now.

启用在 GPU 上训练时的更快内存传输

将数据集分割成每个进程（GPU）的独立、不重叠的子集

```
def main(排名, 世界大小, 迭代次数):
    ddp_setup(rank, world_size) train_loader, test_loader =
prepare_dataset() 模型 = NeuralNetwork(num_inputs=2, num_outputs=2)
模型.to(rank) 优化器 = torch.optim.SGD(模型.parameters(), lr=0.5)
模型 = DDP(模型, device_ids=[rank]) for epoch in range(num_epochs):
for features, labels in train_loader:
```

主要功能
运行模型
训练

排名是
GPU ID

```
        features, labels = features.to(rank), labels.to(rank) # 插入模型预
测和反向传播代码 print(f"[GPU{rank}] Epoch:
{epoch+1:03d}/{num_epochs:03d}")
                f" | 批处理大小 {labels.shape[0]:03d}" " |
            训练/验证损失: {loss:.2f}"

    model.eval()
    训练准确率 = 计算准确度（模型，训练加载器，设备=rank）打印(f"[GPU{rank}]
训练准确率", train_acc) 测试准确率 = 计算准确度（模型，测试加载器，设备
=rank）打印(f"[GPU{rank}] 测试准确率", test_acc) 销毁进程组()
```

清理资源
分配

```
if __name__ == "__main__": # 如果当前模块是直接运行的，则执行以下代码
    打印“可用的 GPU 数量：”，torch.cuda.device_count()
torch.manual_seed(123) num_epochs = 3 world_size =
torch.cuda.device_count() mp.spawn(main, args=(world_size, num_epochs),
                                  nprocs=world_size)
```

启动主功能，使用多个进程，其中
nprocs=world_size 表示每个 GPU 一个进程。

在运行此代码之前，让我们总结一下它的工作原理以及前面的注释。我们在底部有一个 __name__ == "__main__" 子句，其中包含当我们将代码作为 Python 脚本运行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 `multiprocessesing.spawn` function. Here, the `spawn` function launches one process per GPU setting `nproces=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
```

这段代码首先使用 torch.cuda.device_count()打印可用的 GPU 数量，设置随机种子以确保可重复性，然后使用 PyTorch 创建新的进程。

多进程创建函数。在这里，spawn 函数为每个 GPU 启动一个进程，设置 nproces=world_size，其中 world size 是可用的 GPU 数量。此 spawn 函数在同一个脚本中定义的 main 函数中启动代码，并通过 args 提供一些额外的参数。请注意，main 函数有一个 rank 参数，我们不包括在 mp.spawn()调用中。这是因为 rank，即我们用作 GPU ID 的进程 ID，已经自动传递。

主函数通过 ddp_setup 设置分布式环境，另一个我们定义的函数加载训练集和测试集，设置模型并执行训练。与单 GPU 训练（第 A.9.2 节）相比，我们现在通过 .to(rank) 将模型和数据传输到目标设备，我们用 rank 来指代 GPU 设备 ID。我们还通过 DDP 包装模型，这使不同 GPU 在训练期间能够同步梯度。训练完成后，我们评估模型后，使用 destroy_process_group() 清理退出分布式训练并释放分配的资源。

之前我提到，每个 GPU 将接收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)。

最后要讨论的函数是 ddp_setup。它设置主节点的地址和端口，以便不同进程之间进行通信，使用 NCCL 后端（专为 GPU 到 GPU 通信设计）初始化进程组，并设置排名（进程标识符）和全局大小（进程总数）。最后，它指定与当前模型训练进程排名对应的 GPU 设备。

S GPU -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，而您只想使用其中一个 GPU，例如索引为 0 的 GPU。而不是运行 python some_script.py，您可以从终端运行以下代码：

```
CUDA_VISIBLE_DEVICES=0      python   some_script.py
```

或者，如果你的机器有四个 GPU 而你只想使用第一个和第三个 GPU，你可以使用

```
CUDA_VISIBLE_DEVICES=0,2      python   some_script.py
```

请注意，您提供的文本看起来像是一个命令行参数，通常用于指定在 CUDA 环境中使用的 GPU 设备。在这种情况下，文本本身可能不需要翻译，因为它是一个编程相关的专有名词。如果需要翻译，以下是一个可能的翻译：

CUDA 可见设备

这里 CUDA_VISIBLE_DEVICES 的这种做法是一种简单有效的方式来管理 GPU 分配，而无需修改您的 PyTorch 脚本。

现在让我们运行这段代码，通过在终端中以脚本形式启动代码来实际看看它是如何工作的：

```
python   ch02-DDP 脚本.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:                                  <-- Only print in the first process
    print("Test accuracy: ", accuracy)
```

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，它应在单 GPU 和多 GPU 机器上都能工作。如果我们在这台单 GPU 上运行此代码，我们应该看到以下输出：

```
PyTorch 版本：2.2.1+cu117 CUDA 可用：是 可用 GPU 数量：1 [GPU0]
Epoch：001/003 | 批大小 002 | 训练/验证损失：0.62 [GPU0] Epoch：001/003 |
批大小 002 | 训练/验证损失：0.32 [GPU0] Epoch：002/003 | 批大小 002 | 训
练/验证损失：0.11 [GPU0] Epoch：002/003 | 批大小 002 | 训练/验证损失：
0.07 [GPU0] Epoch：003/003 | 批大小 002 | 训练/验证损失：0.02 [GPU0]
Epoch：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（第 A.9.2 节）的输出相似，这是一个很好的合理性检查。

现在，如果我们在一台拥有两个 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：是 可用 GPU 数量：2 [GPU1]
Epoch：001/003 | 批大小 002 | 训练/验证损失：0.60 [GPU0] Epoch：001/003 |
批大小 002 | 训练/验证损失：0.59 [GPU0] Epoch：002/003 | 批大小 002 | 训
练/验证损失：0.16 [GPU1] Epoch：002/003 | 批大小 002 | 训练/验证损失：
0.17 [GPU0] Epoch：003/003 | 批大小 002 | 训练/验证损失：0.05 [GPU1]
Epoch：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
```

如预期，我们可以看到一些批次在第一个 GPU（GPU0）上处理，而其他批次在第二个（GPU1）上处理。然而，当打印训练和测试准确率时，我们看到了重复的输出行。每个进程（换句话说，每个 GPU）独立地打印测试准确率。由于 DDP 将模型复制到每个 GPU，并且每个进程独立运行，因此如果你在测试循环中包含打印语句，每个进程都会执行它，导致重复的输出行。如果你觉得这很麻烦，你可以使用每个进程的 rank 来控制你的打印语句：

```
if rank == 0: 如果 rank 等于 0:
    打印("测试准确度：")       ", 准确度)
```

仅打印在第一个进程中

这是 DDP 通过分布式训练工作的概述。如果您想了解更多详细信息，我建议您查看官方 API 文档，网址为 https:// mng.bz/9dPr。

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.

替代 PyTorch 多 GPU 训练的 API

如果您更喜欢在 PyTorch 中使用更直接的方式处理多个 GPU，可以考虑使用开源的 Fabric 库等附加 API。我在“加速 PyTorch 模型训练：使用混合精度和完全分片数据并行”一文中对此进行了介绍（https://mng.bz/jXle）。

摘要

- PyTorch 是一个开源库，包含三个核心组件：张量库、自动微分函数和深度学习工具。
- PyTorch 的 tensor 库类似于 NumPy 这样的数组库。
- 在 PyTorch 的上下文中，张量是表示标量、向量、矩阵和更高维数组的数组样式的数据结构。
- PyTorch 张量可以在 CPU 上执行，但 PyTorch 张量格式的一个主要优势是其 GPU 支持，可以加速计算。
- PyTorch 中的自动微分（autograd）功能使我们能够方便地使用反向传播训练神经网络，无需手动求导梯度。
- PyTorch 中的深度学习工具提供了创建自定义深度神经网络的构建模块。
- PyTorch 包括 Dataset 和 DataLoader 类来设置高效的数据加载管道。
- 在 CPU 或单个 GPU 上训练模型最容易。
- 使用 DistributedDataParallel 是 PyTorch 中在可用多个 GPU 时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
参考文献和进一步阅读

第一章

定制构建的LLMs能够在团队中超越通用型LLMs，正如彭博社通过一种在金融数据上从头开始预训练的 GPT 版本所展示的那样。定制的LLM在金融任务上优于 ChatGPT，同时在通用LLM基准测试中保持良好性能。

- 布隆伯格 GPT：一款用于金融的大语言模型（2023）吴等人，https://arxiv.org/abs/2303.17564

现有LLMs可以适应和微调，以超越一般的LLMs，谷歌研究和谷歌 DeepMind 在医疗环境中展示了这一点：

- 《基于大型语言模型的专家级医学问答》 (2023) by Singhal 等人，https://arxiv.org/abs/2305.09617

以下论文提出了原始的 Transformer 架构：

- “注意力即是所需” (2017) by Vaswani 等人，https://arxiv.org/abs/1706.03762

在原始编码器风格的 Transformer，称为 BERT，参见

- BERT：深度双向变换器在语言理解方面的预训练（2018）由 Devlin 等人撰写，https://arxiv.org/abs/1810.04805

描述解码器风格的 GPT-3 模型的论文，该模型启发了现代LLMs，并将作为在此书中从头实现LLM的模板

- "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者》（2020）布朗等人，https://arxiv.org/abs/2005.14165

以下涵盖了原始图像分类视觉 Transformer，这表明 Transformer 架构不仅限于文本输入：

- 一张图片胜过 16x16 个字：大规模图像识别中的 Transformer（2020）由 Dosovitskiy 等人著，https://arxiv.org/abs/2010.11929

以下实验性（但不太流行）的LLM架构作为例子，表明并非所有LLMs都需要基于 Transformer 架构：

- RWKV：为 Transformer 时代重新发明 RNN（2023）彭等人，https://arxiv.org/abs/2305.13048
- 《鬣狗等级：迈向更大的卷积语言模型》（2023）由 Poli 等人著，https://arxiv.org/abs/2302.10866
- “Mamba：使用选择性状态空间的线性时间序列建模”（2023）由 Gu 和 Dao 著，https://arxiv.org/abs/2312.00752

Meta AI 的模型是 GPT-like 模型的一个流行实现，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）由 Touvron 等人著，https://arxiv.org/abs/2307.092881

关于 1.5 节中数据集引用的详细信息感兴趣的读者，本文描述了由 Eleuther AI 整理的公开可用 The Pile 数据集：

- 《堆：用于语言建模的 800GB 多样化文本数据集》（2020）高等人，https://arxiv.org/abs/2101.00027

以下论文提供了对 InstructGPT 进行 GPT-3 微调的参考，该内容在第 1.6 节中提及，将在第 7 章中详细讨论：

- 训练语言模型以遵循人类反馈（2022）由欧阳等人著，https://arxiv.org/abs/2203.02155

第二章

读者如对嵌入空间与潜在空间以及向量表示的普遍概念感兴趣，可在我的书的第一章中找到更多信息

- 机器学习 Q&A 与 AI（2023）由 Sebastian Raschka 著，https://leanpub.com/machine-learning-q-and-ai

以下论文更深入地讨论了如何将字节对编码用作标记化方法：

- 神经网络罕见词子词单元翻译（2015）由 Sennrich 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

07909 GPT-2 训练所使用的字节对编码分词器的代码已被 OpenAI 开源：

- https://github.com/openai/gpt-2/blob/master/src/encoder.py

OpenAI 提供了一个交互式 Web UI 来展示 GPT 模型中的字节对分词器是如何工作的：

- https://platform.openai.com/tokenizer

对于对编码和从头开始训练 BPE 分词器感兴趣的读者，Andrej Karpathy 的 GitHub 仓库 minbpe 提供了一个最小化和可读的实现：

- 《一个 BPE 分词器的最小实现》，“https://github.com/karpathy/minbpe”

读者如想了解一些其他流行LLMs所使用的替代分词方案，可以在 SentencePiece 和 WordPiece 论文中找到更多信息：

- “SentencePiece：一种简单且语言无关的神经网络文本处理子词标记器和去标记器”（2018）由 Kudo 和 Richardson 著，https://aclanthology.org/D18-2012/
- 快速 WordPiece 分词（2020）由 Song 等人著，https://arxiv.org/abs/2012.15524

第三章

读者如想了解更多关于 Bahdanau 注意力机制在 RNN 和语言翻译方面的内容，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进行神经机器翻译》（2014）由 Bahdanau、Cho 和 Bengio 著，https://arxiv.org/abs/1409.0473

自注意力（Self-Attention）的概念，即缩放点积注意力，最初在原始的 Transformer 论文中被提出：

- “注意力即是所需” (2017) by Vaswani 等人，https://arxiv.org/abs/1706.03762

FlashAttention 是一种高度高效的自我注意力机制实现，通过优化内存访问模式加速计算过程。在数学上，FlashAttention 与标准自我注意力机制相同，但优化了计算过程以提高效率：

- FlashAttention：具有 IO 感知的快速且内存高效的精确注意力机制（2022）由 Dao 等人提出，https://arxiv.org/abs/2205.14135
- FlashAttention-2：更快的注意力机制，更好的并行性和工作分区（2023）作者：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 实现了一个支持 FlashAttention 以提高效率的自注意力机制和因果注意力机制的功能。此功能为测试版，可能发生变化：

- 缩放点积注意力文档：https://mng.bz/NRJd

PyTorch 还实现了一个基于缩放_的高效的 MultiHeadAttention 类点积函数：

- 多头注意力文档：https://mng.bz/DdJV

Dropout 是一种用于神经网络的正则化技术，通过在训练过程中随机丢弃网络中的单元（及其连接）来防止过拟合：

- Dropout：防止神经网络过拟合的简单方法（2014）由 Srivastava 等人提出，https://jmlr.org/papers/v15/srivastava14a.html

在实践中使用基于缩放点积注意力的多头注意力仍然是自注意力中最常见的变体，但作者发现，即使没有值权重矩阵和投影层，也能实现良好的性能：

- 简化 Transformer 块（2023）由 He 和 Hofmann 著，https://arxiv.org/abs/2311.01906

第四章

该论文介绍了一种通过归一化隐藏层中神经元的总输入来稳定隐藏状态动态的神经网络的技术，与之前发表的方法相比，显著减少了训练时间：

- 层归一化（2016）由 Ba、Kiros 和 Hinton 提出，https://arxiv.org/abs/1607.06450

后层归一化，用于原始的 Transformer 模型，在自注意力机制和前馈网络之后应用层归一化。相比之下，预层归一化，如 GPT-2 和更新的LLMs模型所采用的，在这些组件之前应用层归一化，这可以导致更稳定的训练动态，并在某些情况下已被证明可以提高性能，如以下论文中所述：

- 《Transformer 架构中的层归一化》（2020）由熊等著，https://arxiv.org/abs/2002.04745
- ResiDual：具有双重残差连接的 Transformer（2023）由 Tie 等人撰写，https://arxiv.org/abs/2304.14802

现代LLMs中流行的 LayerNorm 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，而不在平方之前减去均值，从而简化了归一化过程。这意味着在计算尺度之前不会对数据进行中心化。RMSNorm 在更多细节中描述。

- "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

- “张和森尼里克（2019）的《均方根层归一化》”（2019），https://arxiv.org/abs/1910.07467

RMSNorm 在《高斯误差线性单元（GELU）激活函数》中描述得更详细。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，以建模层输出，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) by Hendricks and Gimpel，https://arxiv.org/abs/1606.08415

GPT-2 论文介绍了一系列具有不同大小的基于 transformer 的模型——1.24 亿、3.55 亿、7.74 亿和 15 亿参数： 《语言模型是无监督的多任务学习者》（2019）由 Radford et

al., https://mng.bz/lMgo

OpenAI 的 GPT-3 基本上采用了与 GPT-2 相同的架构，除了最大的版本（1750 亿）比最大的 GPT-2 模型大 100 倍，并且使用了更多的数据进行训练。感兴趣的读者可以参考 OpenAI 的官方 GPT-3 论文和 Lambda Labs 的技术概述，该概述计算在单个 RTX 8000 消费级 GPU 上训练 GPT-3 需要 665 年。

- 《语言模型是少样本学习者》（2023）布朗等人，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 是一个具有简约但高效实现的 GPT-2 模型的代码仓库，类似于本书中实现的模型。虽然本书中的代码与 nanoGPT 不同，但这个仓库启发了将一个大型 GPT Python 父类实现重组为更小的子模块：

- NanoGPT，一个用于训练中等规模 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 个标记时，LLMs中的大部分计算都花费在前馈层而不是注意力层

- 在长期（语境）运行中，Harm de Vries，https://www.harmdevries.com/post/语境长度/

第五章

关于详细说明损失函数以及应用对数变换以使其更容易进行数学优化的信息，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/jiaweizzhao/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

; 视频链接：com/watch?v= GxJe0DZvydM 作者接下来的讲座和代码示例解释了 PyTorch 的交叉熵函数在底层是如何工作的：

- L8.7.1 单热编码和多类别交叉熵，https://www.youtube.com/watch?v=4n71-tZ94yk
- 理解 PyTorch 中的 Onehot 编码和交叉熵

以下两篇论文详细介绍了用于预训练LLMs的数据集、超参数和架构细节

- Pythia：用于分析大型语言模型训练和扩展的套件（2023）由 Biderman 等人著，https://arxiv.org/abs/2304.01373
- OLMo：加速语言模型科学（2024）由 Groeneveld 等人著，https://arxiv.org/abs/2402.00838

以下补充代码可用于本书，包含从 Project Gutenberg 准备 60,000 本公共领域书籍用于LLM训练的说明：

- 预训练 GPT 于 Project Gutenberg 数据集，https://mng.bz/Bdw2

第五章讨论了LLMs的预训练，附录 D 涵盖了更高级的训练功能，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经预训练的LLMs，以及额外的技巧和见解。

- 简单且可扩展的大语言模型预训练策略（2024）伊布拉希姆等人，https://arxiv.org/abs/2403.08763

彭博 GPT 是训练于通用和特定领域文本语料库的特定领域LLM的例子，特别是在金融领域：

- 布隆伯格 GPT：一款用于金融的大语言模型（2023）吴等人，https://arxiv.org/abs/2303.17564

GaLore 是一个旨在使LLM预训练更高效的新近研究项目。所需的代码更改仅归结为在训练函数中将 PyTorch 的 AdamW 优化器替换为 galore-torch Python 包提供的 GaLoreAdamW 优化器：

- GaLore：基于梯度低秩投影的内存高效LLM训练（2024）赵等人，https://arxiv.org/abs/2403.03507
- GaLore 代码仓库，https://github.com/jiaweizzhao/GaLore

以下论文和资源公开分享了用于LLMs的大规模预训练数据集，这些数据集包含数百 GB 到 TB 级的文本数据：

- Dolma：用于LLM预训练研究的三万亿标记开放语料库（2024）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

- 《堆：用于语言建模的 800GB 多样化文本数据集》（2020）高等人，https://arxiv.org/abs/2101.00027
- 《Falcon LLM的 RefinedWeb 数据集：仅使用网络数据超越精选语料库》，（2023）由 Penedo 等人撰写，https://arxiv.org/abs/2306.01116
- “红睡衣”，由 Together AI 创作，https://mng.bz/d6nw
- FineWeb 数据集，包含来自 CommonCrawl 的超过 1500 万亿个清洗和去重后的英文网络数据，https://mng.bz/rVzy

00159 原先介绍 top-k 采样的论文是 Fan 等人于 2018 年发表的“分层神经故事生成”，https://arxiv.org/

abs/1805.04833

一种替代 top-k 采样的方法是 top-p 采样（第 5 章未涉及），它从累积概率超过阈值 p 的最小 top tokens 集合中进行选择，而 top-k 采样则按概率从 top k tokens 中进行选择：

- Top-p 采样，https://zh.wikipedia.org/wiki/Top-p 采样

束搜索（第 5 章未涉及）是一种替代解码算法，通过在每个步骤中仅保留得分最高的部分序列来平衡效率和质量：

- “多样化束搜索：从神经序列模型解码多样化解决方案”（2016）由 Vijayakumar 等人著，https://arxiv.org/abs/1610.02424

第六章

额外资源，讨论不同类型的微调

- 使用和微调预训练的 Transformer，https://mng.bz/VxJG
- 微调大型语言模型，https://mng.bz/x28X

附加实验，包括比较调整第一个输出标记与最后一个输出标记的效果，可以在 GitHub 上的补充代码材料中找到：

- 额外垃圾邮件分类实验，https://mng.bz/AdJx

对于二元分类任务，例如垃圾邮件分类，在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那样：

- 损失学习——在 PyTorch 中优化负对数似然和交叉熵

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在LLM中微调不同层的额外实验，该文章显示，除了输出层外，微调最后一个 Transformer 块可以显著提高预测性能：

- 微调大型语言模型，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：

- “不平衡学习用户指南,” https://mng.bz/2KNa

对于想要对垃圾邮件进行分类而不是垃圾短信的读者，以下资源提供了一个方便的 CSV 格式的大规模电子邮件垃圾邮件分类数据集，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，基于编码器的模型如 BERT 和 RoBERTa 可以在分类任务中有效。

- BERT：深度双向变换器在语言理解方面的预训练（2018）由 Devlin 等人撰写，https://arxiv.org/abs/1810.04805
- RoBERTa：一种鲁棒优化的 BERT 预训练方法（2019）刘等人，https://arxiv.org/abs/1907.11692
- “对 50k IMDB 电影评论进行情感分类的附加实验,” https://mng.bz/PZJR

近期论文显示，通过在分类微调期间移除因果掩码以及进行其他修改，可以进一步提高分类性能：

- 标签监督的 LLaMA 微调（2023）由李等人著，https://arxiv.org/abs/2310.01208
- LLM2Vec：大型语言模型是秘密强大的文本编码器（2024）由 BehnamGhader 等人著，https://arxiv.org/abs/2404.05961

第七章

阿尔帕卡数据集用于指令微调，包含 52,000 个指令-响应对，是首批且最受欢迎的公开指令微调数据集之一：

- 斯坦福 Alpaca：一个遵循指令的骆驼模型，https://github.com/tatsu-lab/stanford_alpaca

额外可供用于指令微调的公开可访问数据集包括

- 利马，https://huggingface.co/datasets/GAIR/lima - 更多信息请参阅“LIMA：对齐的‘少即是多’”，周等，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 - 包含 805,000 条指令-回复对的巨型数据集；更多信息，请参阅 Ding 等人撰写的“通过扩展高质量指令对话来增强聊天语言模型”，https://arxiv.org/abs/2305.14233

- 阿尔帕卡 GPT4，https://mng.bz/AaOp - 使用 GPT-4 而不是 GPT-3.5 生成的类似阿尔帕卡的包含 52,000 条指令-响应对的数据库

11206 Phi-3 是一个拥有 38 亿参数的模型，其指令微调版本据称与更大的专有模型（如 GPT-3.5）相当。

- Phi-3 技术报告：手机上高度强大的本地语言模型（2024）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 Llama3 模型中生成 30 万个高质量的指令-响应对。在这些指令示例上微调的预训练 Llama 3 基础模型与原始指令微调的 Llama-3 模型表现相当。

- 《从零开始通过提示对齐LLMs进行对齐数据合成的 Magpie：Xu 等人，2024》https://arxiv.org/abs/2406.08464

研究表明，在指令微调中不遮蔽指令和输入可以有效提高各种 NLP 任务和开放式生成基准的性能，尤其是在训练包含长指令和简短输出的数据集或使用少量训练示例时

- 指令损失超指令微调（2024）由 Shi 著，https://arxiv.org/abs/2405.14394

普罗米修斯和 PHUDGE 在 1001#上公开可用，它们在评估具有自定义标准的长篇回复方面与 GPT-4 相当。我们不使用这些工具，因为在撰写本文时，它们不受 Ollama 支持，因此无法在笔记本电脑上高效执行。

- “普罗米修斯：在语言模型中诱导细粒度评估能力”（2023）by Kim et al., https://arxiv.org/abs/2310.08491
- PHUDGE：Phi-3 作为可扩展的评判者（2024）由 Deshwal 和 Chawla 著，“https://arxiv.org/abs/2405.08029”
- “普罗米修斯 2：一种专注于评估其他语言模型的开源语言模型”（2024），由金等人著，https://arxiv.org/abs/2405.01535

以下报告的结果支持这样的观点：大型语言模型主要在预训练期间获取事实性知识，而微调主要增强它们使用这些知识的高效性。此外，这项研究还探讨了

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

这项研究探讨了使用新事实信息微调大型语言模型如何影响它们使用现有知识的能力，揭示出模型学习新事实的速度较慢，并且在微调过程中引入新事实增加了模型生成错误信息的倾向

- “在新的知识上对LLMs进行微调是否会鼓励幻觉？”（2024）作者：Gekhman，https://arxiv.org/abs/2405.05904

偏好微调是在指令微调之后的一个可选步骤，用于使LLM更符合人类偏好。作者以下文章提供了更多关于此过程的信息：

- “LLM 训练：强化学习与人类反馈及其替代方案，” https://mng.bz/ZVPm
- 奖励模型预训练和评估技巧，https://mng.bz/RNXj

附录 A

虽然附录 A 应该足以让您跟上进度，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：

- 机器学习：PyTorch 与 Scikit-Learn（2022）作者：Sebastian Raschka 刘海登，瓦希德·米尔贾利利。ISBN 978-1801819312
- 深度学习与 PyTorch（2021）由 Eli Stevens、Luca Antiga 和 Thomas Viehmann 著。ISBN 978-1617295263

对于更深入的了解张量概念，读者可以找到我录制的一个 15 分钟的视频教程：

- 第 4.1 讲：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模型评估的内容，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）由 Sebastian Raschka 著，https://arxiv.org/abs/1811.12808

为对微积分感兴趣的读者提供复习或轻松入门的机会，我在我的网站上撰写了一章关于微积分的内容，该内容免费提供：

- 《微积分导论》，作者：Sebastian Raschka，https://mng.bz/WEyW

为什么 PyTorch 不在后台自动调用 optimizer.zero_grad()？在某些情况下，可能希望累积梯度，PyTorch 将保留此选项供我们选择。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》——Sebastian Raschka，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

本附录涵盖了 DDP，这是一种在多个 GPU 上训练深度学习模型的流行方法。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在不同的 GPU 之间分配大型层。有关更多信息，请参阅此概述，其中包含进一步链接到 API 文档：

- 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
练习解答

完整的练习答案的代码示例可以在补充的 GitHub 仓库中找到，网址为 https://github.com/rasbt/LLMs-from-scratch。

第二章

练习 2.1

您可以通过每次提示编码器一个字符串来获取单个标记 ID：

```
    打印(tokenizer.encode("Ak")) 打
印(tokenizer.encode("w")) # ...
```

这会打印

```
[33901]
[86]
# ...
```

您可以使用以下代码来组装原始字符串：

```
打印（tokenizer.decode([33901,])     86, 343,   86, 220, 959]))
```

这返回

```
'Akwirw ier' 的翻译为：阿基尔维耶尔
```

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

数据加载器的代码，其中 max_length=2，stride=2：

```
dataloader = 创建数据加载器(
raw_text, 批处理大小=4, 最大长度=2, 步长=2 )
```

它产生以下格式的批次：

```
张量([[      40, 367]
        [2885, 1464]
        [1807, 3619]
        [ 402,   271]])
```

第二数据加载器的代码，最大长度为 8，步长为 2：

```
dataloader = 创建数据加载器(
raw_text, 批处理大小=4, 最大长度=8, 步长=2 )
```

一个示例批次看起来像

```
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]])
```

第三章

练习 3.1

正确的权重分配是

```
    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)
```

练习 3.2

为了实现 2 维输出，类似于我们在单头注意力中使用的，我们需要将投影维度 d_out 改为 1。

```
    d_out = 1 mha = 多头注意力包装器(d_in,
                                                d_out,  块大小，0.0,        num_heads=2)
```

练习 3.3

GPT-2 最小模型的初始化为

```
    block_size = 1024
    d_in, d_out = 768, 768 num_heads
= 12 mha = MultiHeadAttention(d_in,
                                          d_out,  块大小         0.0, 头数)
```

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,
```

第四章

练习 4.1

我们可以按照以下方式计算前馈和注意力模块中的参数数量：

```
区块    = TransformerBlock(GPT_CONFIG_124M)

总参数数量 = sum(p.numel() for p in block.ff.parameters()) 打印(f"前
馈模块中的总参数数量：")                                          {总参数:,}")

总参数数量 = sum(p.numel() for p in block.att.parameters()) 打印(f"注
意模块中的总参数数量：")                                          {总参数:,}")
```

我们可以看到，前馈模块包含的参数数量大约是注意力模块的两倍：

```
总参数数量在前馈模块中：4,722,432 总参数数量在注意力模块中：2,360,064
```

练习 4.2

要实例化其他 GPT 模型大小，我们可以修改配置字典，如下所示（此处以 GPT-2 XL 为例）：

```
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)
```

然后，重新使用第 4.6 节中的代码来计算参数数量和 RAM 需求，我们发现

```
gpt2-xl：
总参数数量：1,637,792,000，考虑权重捆绑的可训练参数数量：1,557,380,800，模型总大
小：6247.68 MB
```

练习 4.3

第四章中有三个不同的地方使用了 dropout 层：嵌入层、快捷层和多头注意力模块。我们可以通过在配置文件中分别编码每个层的 dropout 率，然后相应地修改代码实现来控制每个层的 dropout 率。

修改后的配置如下：

```
GPT_CONFIG_124M = {
     "vocab_size": 50257,
 "context_length": 1024,
 "emb_dim": 768
```

```
    "n_heads": 12,
    "n_layers": 12,
    "drop_rate_attn": 0.1,
    "drop_rate_shortcut": 0.1,
    "drop_rate_emb": 0.1,
    "qkv_bias": False
}
```

Dropout for multi-head attention

Dropout for shortcut 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"],
            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_shortcut"]
        )

    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"])
```

Dropout for multi-head attention

Dropout for shortcut connections

Dropout for embedding layer

```
    "n_heads": 12,
    "n_layers": 12,
"drop_rate_attn": 0.1,
"drop_rate_shortcut": 0.1,
"drop_rate_emb": 0.1,
"qkv_bias": False }
```

多 dropout 头部注意力

dropout 用于快捷连接

Dropout 用于嵌入层

修改后的 TransformerBlock 和 GPTModel 看起来

```
class TransformerBlock(神经网络模块):
    def __init__(self, cfg):
        super().__init__()
        self.att = 多头注意力
        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"]) self.ff =
     FeedForward(cfg) self.norm1 =
     LayerNorm(cfg["emb_dim"]) self.norm2 =
     LayerNorm(cfg["emb_dim"]) self.drop_shortcut =
     nn.Dropout()
```

多 dropout 头部注意力

```
        cfg["drop_rate_shortcut"] )
```

dropout 用于快捷连接

```
    def forward(self, x): # 定义前向传播函数
        快捷键 = x
        x = self.norm1(x)
        x = self.att(x)
        x = self.drop_shortcut(x) x
     = x + shortcut

        快捷键 = x
        x = self.norm2(x)
        x = self.ff(x)
        x = self.drop_shortcut(x) x
     = x + shortcut return x



class GPT 模型(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"])
```

Dropout 用于嵌入层

```
        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 logitss
```

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 32× 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(
            *[TransformerBlock（cfg）for]     _ 在 range(cfg["n_layers"])范围内]]

        self.final_norm = 层归一化(cfg["emb_dim"])
    self.out_head = 线性层(
        cfg["emb_dim"], cfg["vocab_size"], 偏置=False)


def forward(self, 输入索引):
        批大小，序列长度 = in_idx.shape，词嵌
    入 = self.tok_emb(in_idx)，位置嵌入 =
    self.pos_emb(
        torch.arange(seq_len, 设备=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)
    返回 logitss
```

第五章

练习 5.1

我们可以使用本节中定义的 print_sampled_tokens 函数来打印出“pizza”这个标记（或单词）被采样的次数。让我们从 5.3.1 节中定义的代码开始。

“pizza”标记在温度为 0 或 0.1 时以 0x 采样，当温度提升至 5 时以 32×采样。估计概率为 32/1000 × 100% = 3.2%。

实际概率为 4.3%，包含在重新缩放的 softmax 概率中

张量（scaled_probas[2][6]）

练习 5.2

Top-k 采样和温度缩放是必须根据LLM和所需的输出多样性和随机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）并且将温度设置在 1 以下时，模型的输出变得更加不随机和确定。此设置在需要生成的文本更加可预测、连贯，并且更接近基于训练数据的最大可能性结果时很有用。

适用于如此低的 k 和温度设置的用途包括生成正式文件或报告，其中清晰度和准确性最为重要。其他应用示例包括技术分析或代码生成任务，其中精度至关重要。此外，问答和教育内容需要准确答案，当温度低于 1 时有助于提高准确性。

另一方面，当使用LLMs进行头脑风暴或生成创意内容，如小说时，较大的 top-k 值（例如，20 至 40 范围内的值）和高于 1 的温度值是有用的。

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

有多种方法可以强制生成函数具有确定性行为：

1. 设置 top_k 为 None 并应用无温度缩放
2. 设置 top_k=1

练习 5.4

本质上，我们必须加载我们在第一章中保存的模型和优化器：

```
    checkpoint = 加载 torch 模型("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"])
```

然后，使用 num_epochs=1 调用 train_simple_function 来为模型训练另一个 epoch。

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_loss_loader(train_loader, gpt, device) 验证损失
= calc_loss_loader(val_loader, gpt, device)
```

124 百万参数的结果损失如下：

```
    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的表现处于同一水平。这可能有多种解释：

1. 《判决》不是 OpenAI 训练 GPT-2 时的预训练数据集的一部分。因此，该模型并未显式地过度拟合训练集，在《判决》的训练集和验证集部分表现相似。（验证集损失略低于训练集损失，这在深度学习中是不常见的。然而，这很可能是由于随机噪声，因为数据集相对较小。在实践中，如果没有过度拟合，训练集和验证集的性能预计将大致相同）。

2. 《判决》是 GPT-2 的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度拟合了训练数据，因为验证集也已经被用于训练。为了评估过度拟合的程度，我们需要一个在 OpenAI 完成 GPT-2 训练后生成的新的数据集，以确保它不可能成为预训练的一部分。

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).

练习 5.6

在主章节中，我们尝试了最小的 GPT-2 模型，它只有 1240 万个参数。这样做的原因是为了尽可能降低资源需求。然而，您可以通过最小的代码更改轻松地尝试更大的模型。例如，在第五章中，我们只需更改以下两行代码，就可以加载 1.558 亿个参数的模型，而不是 1240 万个参数的模型：

```
    hparams, params = 下载并加载 gpt2（模型大小为"124M"，模型目录为"gpt2"）model_name =
"gpt2-small (124M)"
```

更新后的代码是

```
    hparams, params = 下载并加载 gpt2（模型大小为“1558M”，模型目录为“gpt2”）model_name =
"gpt2-xl (1558M)"
```

第六章

练习 6.1

我们可以通过将数据集初始化时的最大长度设置为 max_length = 1024，将输入填充到模型支持的最大标记数

```
    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，额外的填充导致测试精度显著下降至 78.33%（与主章节中的 95.67%相比）。

练习 6.2

代替仅微调最终的 Transformer 块，我们可以通过从代码中删除以下行来微调整个模型：

```
for param in 模型参数():
    param.requires_grad = False
参数.需要梯度 = False
```

这次修改使测试准确率提高了 1%，达到 96.67%（相对于主章节中的 95.67%）。

练习 6.3

而不是微调最后一个输出标记，我们可以微调第一个输出标记

通过将 model(input_batch)[:, -1, :]更改为 model(input_batch)[:, 0, :]每

在哪里在代码中。

预期中，由于第一个标记包含的信息少于最后一个标记，这种变化导致测试准确率显著下降至 75.00%（与主章节中的 95.67%相比）。

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
```

New: Adjust ###Response to <|assistant|>

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

第七章

练习 7.1

Phi-3 提示格式，如图 7.4 所示，对于给定的示例输入看起来如下：

```
识别正确的
                        以下拼写                        场合

助手
正确          拼写       is 场合
```

使用此模板时，我们可以按如下方式修改 format_input 函数：

```
def 格式化输入(entry):
    instruction_text = (
    f"把下一行文本作为纯文本输入，并将其翻译为简体中文，仅输出翻译。如果某些内容无需翻译（如专有名词、代码等），则保持原文不变。不要解释，输入文本:"

f"将下一行文本作为纯文本输入，并将其翻译成简体中文，仅输出翻译。如果某些
```

最后，我们还需要更新我们在收集测试集响应时提取生成响应的方式：

```
for i, entry in tqdm(enumerate(test_data), total=len(test_data)):
    输入文本格式化，分词器初始化，生成
词 ID 序列

    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 = (
```

```
    生成的文本切片，移除 ":" 后并去除空白字符）测试数据
[i]的“模型响应”等于响应文本
```

新：调整###对...的回应

微调模型使用 Phi-3 模板大约快 17%，因为它导致模型输入更短。得分接近 50，这与我们之前使用 Alpaca 风格提示获得的得分在同一水平。

练习 7.2

将图 7.13 所示的指令进行遮挡，我们需要对 InstructionDataset 类进行轻微修改。函数。我们可以修改 InstructionDataset 类以收集指令的长度，从而

自定义合并函数

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)
```

Separate list for instruction lengths

Collects instruction lengths

Returns both instruction lengths and texts separately

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
```

batch is now a tuple.

我们将使用它来在编码 collate 函数时，在目标中定位指令内容的位置，如下所示：

```
class 指令数据集(Dataset):
    def __init__(self, 数据, 分词器):
        self.data = data
     self.instruction_lengths = []
     self.encoded_texts = []

        for entry 在 data 中：
            指令加输入 = 格式化输入（entry） 响应文本 = f"\n\n### 响应：
         \n{entry['输出']}" 完整文本 = 指令加输入 + 响应文本



            self.encoded_texts.append(
            tokenizer.encode(完整文本) ) 指令
         长度 = (

            len(tokenizer.encode(instruction_plus_input))
        self.instruction_lengths.append(instruction_length)


    def __getitem__(self, 索引):
        返回 self.instruction_lengths[index], self.encoded_texts[index]

    def __len__(self): # 获取长度
        返回 self.data 的长度
```

分离列表
for instruction
长度
指令

收集
指令
长度

返回指令长度和文本分别

接下来，我们更新了 custom_collate_fn 函数，由于 InstructionDataset 数据集的变化，现在每个批次都是一个包含(instruction_length, item)的元组，而不是仅仅包含 item。此外，我们现在在目标 ID 列表中屏蔽相应的指令标记：

```
def custom_collate_fn(
    批量
    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 = [], []
 批次最大长度 = max(len(item)+1 for instruction_length, item 在 batch) 输入列表，目标列表 = [], []
    for 指令长度, 项目 in 批次:
        new_item = item.copy()
     new_item += [填充令牌 ID] 填充后的序列 = (

        new_item + [填充令牌 ID] * (batch_max_length - len(new_item)) 输入 = torch.tensor(padded[:-1]) 目标 = torch.tensor(padded[1:])


        mask = 目标 == 填充令牌 ID
```

批处理现在是一个元组。

```
        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() 如
    果 indices.numel() > 1:
            targets[indices[1:]] = 忽略索引

        targets[:instruction_length-1] = -100

        如果 allowed_max_length 不是 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)

输入张量 = torch.stack(输入列表).移动到设备 targets_tensor =
torch.stack(目标列表).移动到设备
    返回 inputs_tensor, targets_tensor
```

屏蔽所有输入目标中的指令标记

在评估使用此指令掩码方法微调的模型时，其表现略差（使用第 7 章的 Ollama Llama 3 方法大约差 4 分）。这与“带有指令损失的指令调整”论文中的观察结果一致（https://arxiv.org/abs/2405.14394）。

练习 7.3

为了在原始斯坦福 Alpaca 数据集（https://github.com/tatsu-lab/stanford_alpaca）上微调模型，我们只需更改文件 URL 即可

```
    url = "https://raw.githubusercontent.com/rasbt/LLMs-从零开始/main/ch07/ 01_章节代
码/instruction-data.json"
```

to

```
    url = "https://raw.githubusercontent.com/tatsu-
lab/stanford_alpaca/main/alpaca_data.json"
```

请注意，该数据集包含 52,000 条条目（比第 7 章多 50 倍），且条目长度比我们在第 7 章中使用的条目长。

因此，强烈建议在 GPU 上运行训练。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。除了降低批大小外，您还可以考虑将允许的最大长度从 1024 降低到 512 或 256。

以下是从 Alpaca 数据集中选取的一些示例，包括生成的模型响应：

练习 7.4

使用 LoRA 微调模型时，请使用附录 E 中的相关类和函数：

```
from  附录 E 导入 LoRALayer, LinearWithLoRA,                    替换为 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 7.5 节中的模型加载代码下方添加以下代码行：

```
    总参数数 = sum(p.numel() for p in model.parameters() if p.requires_grad) 打印(f"训练前总
可训练参数数: {total_params:,}")


    for param in 模型参数():
        param.requires_grad = False
     参数.需要梯度 = False
    总可训练参数数：{total_params:,}，替换线性层为 LoRA（rank=16, alpha=16）
    总参数数 = sum(p.numel() for p in model.parameters() if p.requires_grad) 打印(f"总可训
练 LoRA 参数数: {total_params:,}") 模型.to(device)
```

请注意，在 Nvidia L4 GPU 上，使用 LoRA 进行微调在 L4 上运行需要 1.30 分钟。在相同的 GPU 上，原始代码运行需要 1.80 分钟。因此，在这种情况下，LoRA 大约快 28%。使用第 7 章中 Ollama Llama 3 方法评估的分数约为 50，与原始模型在同一水平。

附录 A

练习 A.1

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。从编程的角度来看，我们可以这样计算参数数量：

```
    模型 = 神经网络(2, 2) num_params = sum(p.numel() for p in model.parameters() if
p.requires_grad) 打印("模型可训练参数总数：", num_params)
```

这返回

```
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 30 个偏置单元
- 第二隐藏层—30 个输入单元乘以 20 个节点加 20 个偏置单元
- 输出层—20 个输入节点乘以 2 个输出节点加 2 个偏置单元

然后，将每一层的所有参数相加得到 $2 \times 30 + 30 + 30 \times 20 + 20 + 20 \times 2 + 2 = 752$。

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 µs ± 8.7 µs per loop
```

When executed on a GPU,

```
a, b = a.to("cuda"), b.to("cuda")
%timeit a @ b
```

the result was

```
13.8 µs ± 425 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 $2\times30+30 + 30\times20+20 + 20\times2+2 = 752$.

练习 A.2

精确的运行时结果将特定于用于此实验的硬件。在我的实验中，即使对于如下小矩阵乘法，使用连接到 V100 GPU 的 Google Colab 实例也能观察到显著的加速：

```
    a = torch.rand(100, 200) b
= torch.rand(200, 300) %timeit
a@b
a = torch.rand(100, 200) b =
```

在 CPU 上，这导致了

```
63.8  µs ±8.7  µs per  loop
```

当在 GPU 上执行时，

```
    a, b = a.to("cuda"), b.to("cuda")
%timeit a @ b
```

结果为

```
13.8  µs ±425  ns  per  loop
```

在这种情况下，在 V100 上，计算速度大约快了四倍。

练习 A.3

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 和 20 个节点。从编程的角度来看，我们可以这样计算参数数量：

```
    模型 = 神经网络(2, 2) num_params = sum(p.numel() for p in 模型.parameters() if
p.requires_grad) 打印("模型可训练参数总数：", num_params)
```

这返回

```
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 30 个偏置单元
- 第二隐藏层：30 个输入单元乘以 20 个节点加 20 个偏置单元
- 输出层：20 个输入节点乘以 2 个输出节点加 2 个偏置单元

然后，将每一层的所有参数相加得到 $2\times30+30 + 30\times20+20 + 20\times2+2 = 752$。

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 µs ± 8.7 µs per loop
```

When executed on a GPU

```
a, b = a.to("cuda"), b.to("cuda")
%timeit a @ b
```

The result was

```
13.8 µs ± 425 ns per loop
```

In this case, on a V100, the computation was approximately four times faster.

练习 A.4

精确的运行时结果将特定于用于此实验的硬件。在我的实验中，即使对于小型矩阵乘法，使用连接到 V100 GPU 的 Google Colab 实例也能观察到显著的加速：

```
    a = torch.rand(100, 200) b
= torch.rand(200, 300) %timeit
a@b
a = torch.rand(100, 200) b =
```

在 CPU 上这导致了

```
    63.8  微秒 ± 8.7微秒     loop
```

当在 GPU 上执行时

```
    a, b = a.to("cuda"), b.to("cuda")
%timeit a @ b
```

结果为

```
    13.8  微秒 ± 425ns per  loop
```


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 fine-tuning 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()
```

Vocabulary size
Shortened context length (orig: 1024)
Embedding dimension
Number of attention heads
Number of layers
Dropout rate
Query-key-value bias

After initializing the model, we need to initialize the data loaders. First, we load the "The Verdict" short story:

附录 D 添加铃声和哨声至于训练循环

在本附录中，我们增强了第 5 至 7 章中涵盖的预训练和微调过程的训练函数。特别是，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纳入训练函数，并在LLM上进行预训练。

为了使代码自包含，我们重新初始化了第 5 章中训练的模型：

```
import torch
from  第四章 导入 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 } 设备 = torch.device("cuda" if
torch.cuda.is_available() else "cpu") torch.manual_seed(123) 模型 =
GPTModel(GPT_CONFIG_124M) 模型.to(设备) 模型.eval()
```

词汇量

缩短的上下文长度（原：1024）

嵌入维度

注意头数量

层数数量

辍学率

查询键值偏差

初始化模型后，我们需要初始化数据加载器。首先，我们加载《裁决》短篇小说：

```
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
    导入    urllib.request

    file_path = "the-verdict.txt"

    url = (
    "https://raw.githubusercontent.com/rasbt/LLMs-从零开始/"
"main/ch02/01_主要章节代码/判决.txt" )



    如果 os.path.exists(file_path) 返回 False:
        使用 urllib.request.urlopen(url) 作为 response:
        text_data = response.read().decode('utf-8') 使用
     open(file_path, "w", encoding="utf-8") as file:
            file.write(text_data) # 写入文本数据到文件
    否则：
        with open(file_path, "r", encoding="utf-8") as file:
            请提供需要翻译的文件或数据
```

接下来，我们将 text_data 加载到数据加载器中：

```
    来自前几章                    导入    创建数据加载器_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], 批处理大小=2, 最大长度
        =GPT_CONFIG_124M["上下文长度"], 步长
        =GPT_CONFIG_124M["上下文长度"], 最后一个丢弃=True, 混洗
        =True, 工作进程数=0



    )
    val_loader = 创建_dataloader_v1()
    text_data[split_idx:], 批量大小=2, 最大长度
=GPT_CONFIG_124M["context_length"], 步长
=GPT_CONFIG_124M["context_length"], drop_last=False,
shuffle=False, 工作进程数=0
```

D.1 学习率预热

实现学习率预热可以稳定复杂模型如LLMs的训练。这个过程涉及将学习率从非常低的初始值（initial_lr）逐渐增加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训练阶段遇到大而破坏稳定的更新的风险。

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)
print(warmup_steps)
```

20% warmup

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

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:
            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"])
```

This increment is determined by how much we increase the inital_lr in each of the 20 warmup steps.

Executes a typical training loop iterating over the batches in the training loader in each epoch

Updates the learning rate if we are still in the warmup phase

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，增加到最大学习率 0.01：

```
n_epochs = 15
初始学习率 = 0.0001
peak_lr = 0.01
warmup_steps = 20
```

预热步数 = 20

预热步骤的数量通常设置在总步骤数的 0.1%到 20%之间，我们可以按以下方式计算：

```
total_steps = len(train_loader) * n_epochs
warmup_steps = int(0.2 * total_steps) 打印
(warmup_steps)
```

20% 预热

这会打印 27，意味着我们有 20 个预热步骤，在最初的 27 个训练步骤中将初始学习率从 0.0001 增加到 0.01。

接下来，我们实现一个简单的训练循环模板来展示这个过程：

```
优化器 = torch.optim.AdamW(model.parameters(), weight_decay=0.1) lr_increment
= (peak_lr - initial_lr) / warmup_steps

全局步数 = -1
track_lrs   = []

for epoch 在 range(n_epochs):
    for 输入批次，目标批次 in 训练加载器：
        optimizer.zero_grad()
    global_step += 1
    优化器清零梯度() 全局步数
    加 1
        如果 global_step < warmup_steps:
        lr = 初始 lr + 全局步长 * lr 增量 else:

            lr =  峰值学习率

        for param_group in optimizer.param_groups:
        param_group["lr"] = lr track_lrs.append(优化
    器.param_groups[0]["lr"])
```

这次增加
根据每次预热步骤中增加的初始学习率（initial_lr）的量来决定。

执行每个 epoch 中训练加载器批次的典型训练循环

应用
计算
学习
汇率
优化器

更新学习率，
如果仍在预热阶段

在一个完整的训练循环中，会计算损失和模型更新，这里为了简化省略了这些内容。

在运行前面的代码后，我们可视化训练循环如何改变学习率，以验证学习率预热是否按预期工作：

```
导入    matplotlib.pyplot 作为   plt

plt.ylabel("学习率") plt.xlabel("步数")
total_training_steps = 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).

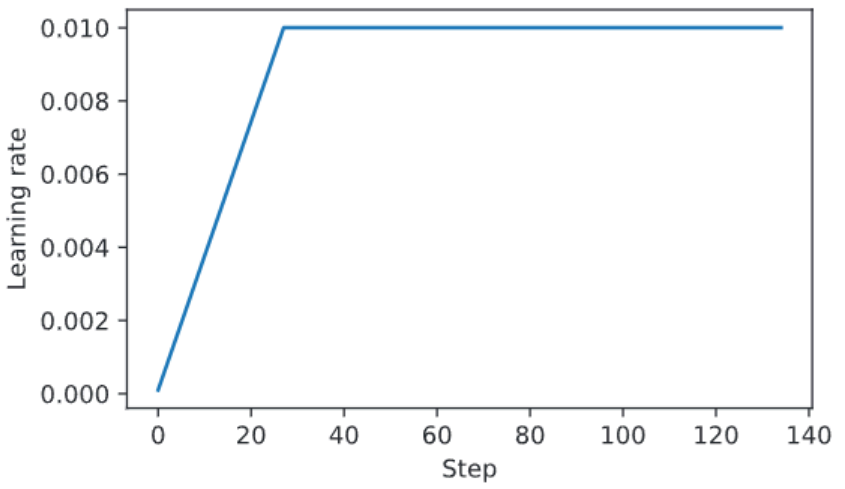


Figure D.1 The learning rate warmup increases the learning rate for the first 20 training steps. After 20 steps, the learning rate reaches the peak of 0.01 and remains constant for the rest of the training.

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_increment
        else:
            progress = ((global_step - warmup_steps) /
                        (total_training_steps - warmup_steps))
```

Applies linear warmup

Uses cosine annealing after warmup

结果图显示，学习率从低值开始，经过 20 步增加到最大值（图 D.1）。

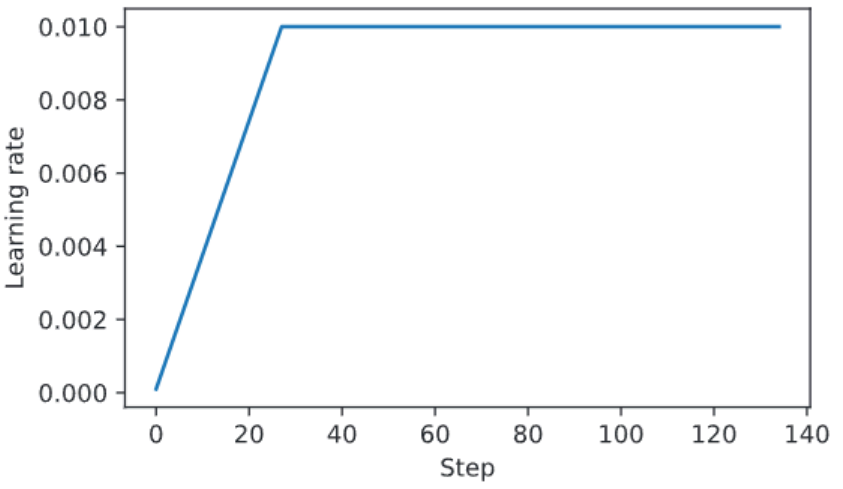


图 D.1 学习率预热增加了前 20 个训练步骤的学习率。20 步之后，学习率达到峰值 0.01，并在剩余的训练中保持不变。

接下来，我们将进一步调整学习率，使其在达到最大学习率后降低，这有助于进一步提高模型训练效果。

D.2 余弦衰减

另一种广泛采用的训练复杂深度神经网络的技术是余弦衰减。这种方法在整个训练周期中调节学习率，使它在预热阶段之后遵循余弦曲线。

在它的流行变体中，余弦衰减将学习率降低（或衰减）到几乎为零，模仿半余弦周期的轨迹。余弦衰减中逐渐降低的学习率旨在减缓模型更新其权重的速度。这尤其重要，因为它有助于在训练过程中最小化超过损失最小值的风险，这对于确保训练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：

```
    导入     math

    min_lr = 0.1 * 初始学习率 track_lrs = [] 学习率增量 = (峰值
学习率 - 初始学习率) / 预热步数 global_step = -1



    for epoch 在 range(n_epochs):
         for 输入批次，目标批次 in 训练加载器：
              optimizer.zero_grad()
          global_step += 1

              如果 global_step < warmup_steps:
              lr = 初始学习率 + 全局步数 * 学习率增量 else:

                   进展       = ((global_step - warmup_steps) /
                          (total_training_steps - warmup_steps))
```

适用于线性预热

使用余弦退火（预热后）

```
        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).

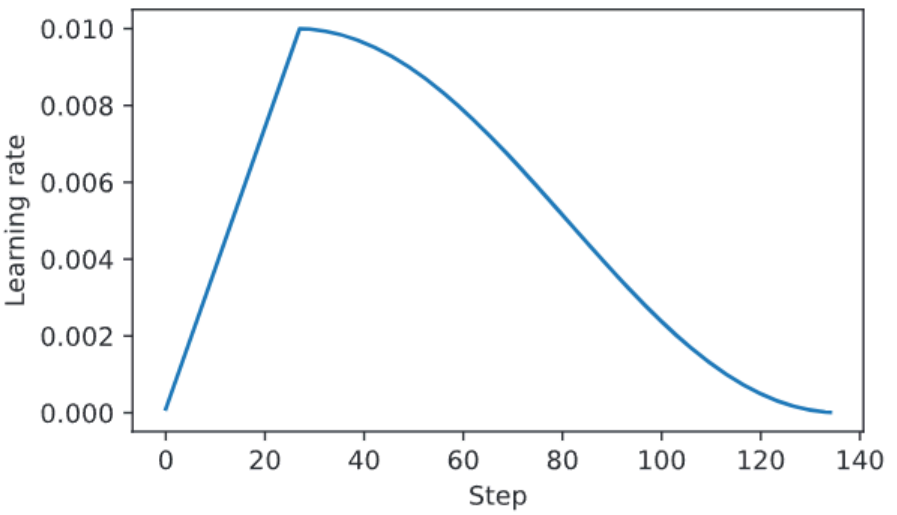


Figure D.2 The first 20 steps of linear learning rate warmup are followed by a cosine decay, which reduces the learning rate in a half-cosine cycle until it reaches its minimum point at the end of training.

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 downscaled 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 $\boldsymbol{v}$ composed of components $\boldsymbol{v} = [v_1, v_2, \ldots, v_n]$, the L2 norm is

$$|\boldsymbol{v}|_2 = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}$$

```
        lr = min_lr + (peak_lr - min_lr) * 0.5 * (
        + math.cos(math.pi * 进度) )


    for param_group in optimizer.param_groups:
    param_group["lr"] = lr track_lrs.append(优化
器.param_groups[0]["lr"])
```

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("步数")
plt.plot(range(总训练步数), track_lrs) plt.show()
```

结果的学习率图显示，学习率开始于线性预热阶段，增加 20 步后达到最大值。在 20 步线性预热之后，余弦衰减开始，学习率逐渐降低，直到达到最小值（图 D.2）。

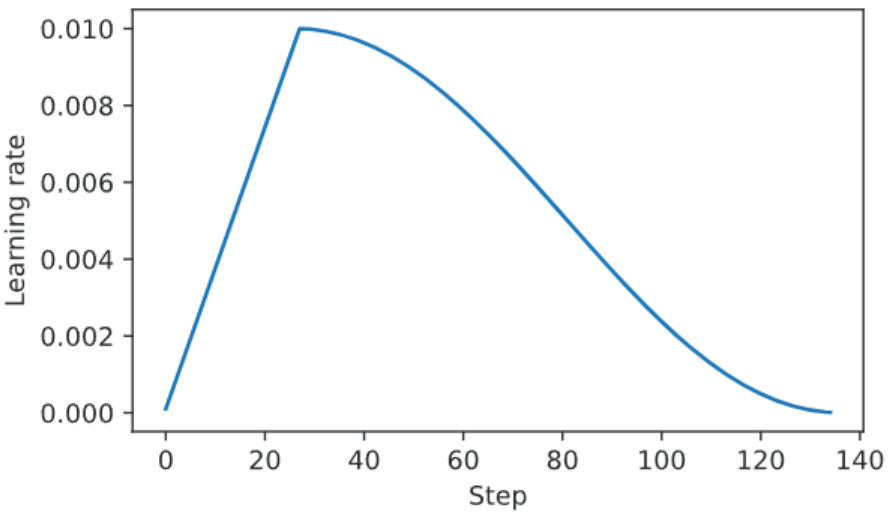


图 D.2 线性学习率预热的前 20 步之后，接着是余弦衰减，学习率在半余弦周期内减少，直到训练结束时达到最小值。

D.3 梯度裁剪

梯度裁剪是增强LLM训练稳定性的另一种重要技术。这种方法涉及设置一个阈值，当梯度超过此阈值时，将其下缩到预定的最大幅度。这个过程确保在反向传播过程中模型参数的更新保持在可管理的范围内。

例如，在 PyTorch 的 clip_grad_norm_函数中应用 max_norm=1.0 设置确保梯度的范数不超过 1.0。在这里，“范数”一词表示梯度向量在模型参数空间中的长度或大小，具体指的是 L2 范数，也称为欧几里得范数。

在数学术语中，对于由分量 v = [v, v, ..., v] 组成的向量 v，L2 范数为

$$|\boldsymbol{v}|_2 = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}$$

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 `max_norm`$/|\mathbf{G}|_2 = 1/5$. Consequently, the adjusted gradient matrix **G'** becomes

$$\mathbf{G}' = \frac{1}{5} \times G = \begin{bmatrix} \frac{1}{5} & \frac{2}{5} \\ \frac{3}{5} & \frac{4}{5} \end{bmatrix}$$

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))
```

L2 范数是这种计算方法也应用于矩阵。例如，考虑一个由梯度矩阵给出

$$\mathbf{G} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

如果我们想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

$$|\mathbf{G}|_2 = \sqrt{1^2 + 2^2 + 2^2 + 4^2} = \sqrt{25} = 5$$

由于|G|=5 超出了我们的 max_norm 1 的限制，我们将梯度缩放以确保它们的范数正好等于 1。这是通过一个缩放因子实现的，计算为 max_norm/|G|=1/5。因此，调整后的梯度矩阵 G'变为

$$\mathbf{G}' = \frac{1}{5} \times G = \begin{bmatrix} \frac{1}{5} & \frac{2}{5} \\ \frac{3}{5} & \frac{4}{5} \end{bmatrix}$$

为了说明梯度裁剪的过程，我们首先初始化一个新的模型，并计算一个训练批次的损失，类似于标准训练循环中的步骤：

```
从第五章导入 calc_loss_batch

torch 手动设置随机种子(123) 模型 = GPTModel(GPT_CONFIG_124M) 模型转移到设
备 device 上 损失 = calc_loss_batch(input_batch, target_batch, 模型, 设备) 损
失反向传播()
```

在调用.backward()方法后，PyTorch 计算损失梯度并将它们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，通过扫描模型权重张量的所有.grad 属性来识别最高的梯度值，在调用.backward()之后：

```
def find_highest_gradient(model):
    最大梯度 = None
    for param 在 model.parameters() 中：
        如果 param.grad 不是 None：
            grad_values = param.grad.data.flatten() 最大梯度参数 =
        grad_values.max() 如果最大梯度为空或最大梯度参数 > 最大梯
        度：
                最大梯度    最大梯度参数
返回最大梯度 打印
（find_highest_gradient(model)）
```

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-6):

    train_losses, val_losses, track_tokens_seen, track_lrs = [], [], [], []
    tokens_seen, global_step = 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

            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))
```

Retrieves the initial learning rate from the optimizer, assuming we use it as the peak learning rate

Calculates the total number of iterations in the training process

Calculates the learning rate increment during the warmup phase

Adjusts the learning rate based on the current phase (warmup or cosine annealing)

前述代码识别出的最大梯度值是

```
张量(0.0411)
```

现在应用梯度裁剪，看看这对最大梯度值有何影响：

```
    torch.nn.utils.clip_grad_norm_(模型参数(), 最大范数=1.0) 打印
(find_highest_gradient(模型))
```

应用梯度裁剪（最大范数为 1）后的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改进了 train_model_simple 训练函数（见第 5 章）：线性预热、余弦衰减和梯度裁剪。这些方法共同帮助稳定LLM训练。

代码，与 train_model_simple 注释的更改相比，如下所示：

```
from  第五章导入 evaluate_model,                    生成并打印样本


def 训练模型(模型, 训练加载器, 验证加载器, 优化器, 设备,)
                    n_epochs, 评估频率, 评估迭代次数, 起始上下文, 分词器, 预热步数, 初
                始学习率=3e-05, 最小学习率=1e-6):

    训练损失，验证损失，跟踪看到的标记，跟踪学习率                                = [], [],    [],  []
    tokens_seen，已看到标记 global_step =    0, -1

    peak_lr = 优化器参数组[0]中的"lr" total_training_steps =
 train_loader 长度 * n_epochs lr_increment = (peak_lr
                                        - 初始学习率) / 预热步数

    for epoch 在 range(n_epochs):
         model.train()
         for 输入批次，目标批次 in 训练加载器:
              optimizer.zero_grad()
          全局步数 += 1

              如果 global_step < warmup_steps:
              lr = 初始 lr + 全局步数 * lr 增量

                  progress = ((全局步数 - warmup_steps) / (总训练步数 -
                                                                warmup_steps))
                  lr = min_lr + (peak_lr - min_lr) * 0.5 * (
                       1 + math.cos(math.pi *     progress))
```

检索优化器中的初始学习率
假设我们将其用作峰值学习率

计算
总数量
迭代次数在
训练过程

计算预热阶段的
学习率增量

调整
学习率
基于
当前阶段
预热或
余弦
退火

```
            for param_group in optimizer.param_groups:
                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:
                torch.nn.utils.clip_grad_norm_(
                    model.parameters(), max_norm=1.0
                )

            optimizer.step()
            tokens_seen += input_batch.numel()

            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}"
                )
```

Applies the calculated learning rate to the optimizer

Applies gradient clipping after the warmup phase to avoid exploding gradients

Everything below here remains unchanged compared to the train_model_simple function used in chapter 5.

```
            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:
    param_group["lr"] = lr track_lrs.append(lr) 损失 =
calc_loss_batch(input_batch, target_batch, model, device) 损失.backward()
```

应用计算出的学习率到优化器

```
    如果 global_step > warmup_steps:
         torch.nn.utils.clip_grad_norm_
         model.parameters(), 最大范数=1.0 )
```

在预热阶段之后应用梯度裁剪以避免梯度爆炸

```
    optimizer.step() tokens_seen +=
input_batch.numel()
```

以下内容与第 5 章中使用的 train_model_simple 函数相比保持不变。

```
    if global_step % eval_freq == 0: # 如果 global_step 对 eval_freq 取模等于 0：
         训练损失，验证损失 = 评估模型(
         模型，训练加载器，验证加载器，设备，评估
     迭代器）将训练损失添加到 train_losses 中，将
     验证损失添加到 val_losses 中，将已见令牌添加
     到 track_tokens_seen 中，打印(f"Ep
     {epoch+1}")
                                        (迭代 {global_step:06d})        "
         f"训练损失 {train_loss:.3f}, " f"验
     证损失 {val_loss:.3f}"
         生成并打印示例()
         模型、分词器、设备、起始上下文）
```

返回 训练损失 val_losses 跟踪已看到的令牌 track_lrs

在定义了 train_model 函数之后，我们可以以类似的方式使用它来训练模型，与我们在预训练中使用的 train_model_simple 方法相比：

```
    导入 tiktoken

    torch.manual_seed(123) 模型 = GPTModel(GPT_CONFIG_124M) 模型.to(device)
peak_lr = 5e-4 优化器 = torch.optim.AdamW(模型.parameters(), weight_decay=0.1) 分
词器 = tiktoken.get_encoding("gpt2")

    n_epochs = 15
    训练损失，验证损失，已见令牌，学习率 = 训练模型(
    模型，训练加载器，验证加载器，优化器，设备，n_epochs=n_epochs，评估频率=5，评估迭
代=1，起始上下文="每一点努力都让你前进"，分词器=分词器，预热步数=warmup_steps，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:

```
Ep 1 (Iter 000000): Train loss 10.934, Val loss 10.939
Ep 1 (Iter 000005): Train loss 9.151, Val loss 9.461
Every effort moves you,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
Ep 2 (Iter 000010): Train loss 7.949, Val loss 8.184
Ep 2 (Iter 000015): Train loss 6.362, Val loss 6.876
Every effort moves you,,,,,,,,,,,,,,,,,,, the,,,,,,,, the,,,,,,,,,,,
the,,,,,,,,
...
Ep 15 (Iter 000130): Train loss 0.041, Val loss 6.915
Every effort moves you?"  "Yes--quite insensible to the irony. She wanted him
vindicated--and by me!"  He laughed again, and threw back his head to look up
at the sketch of the donkey. "There were days when I
```

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 或类似笔记本电脑上大约需要 5 分钟完成，并打印以下输出：

```
第 1 集（迭代 000000）：训练损失 10.934，验证损失 10.939 第 1 集（迭代 000005）：训练损失 9.151，验证损失 9.461 每一份努力都在推动你,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
第 2 集（迭代 000010）：训练损失 7.949，验证损失 8.184 第 2 集（迭代 000015）：训练损失 6.362，验证损失 6.876 每一份努力都在推动你,,,,,,,,,,,,,,,,,, the,,,,,,,,, the,,,,,,,,,,, the,,,,,,,, ... 第 15 集（迭代 000130）：训练损失 0.041，验证损失 6.915 每一份努力都在推动你？" "是的——对讽刺毫无感觉。她希望他被证明无罪——而且是通过我！" 他又笑了，仰起头看着驴子的素描。"有过那么几天，我
```

与预训练一样，由于数据集非常小，模型在几个 epoch 后开始过拟合，并且我们对它进行了多次迭代。尽管如此，我们可以看到函数正在起作用，因为它最小化了训练集损失。

读者被鼓励在一个更大的文本数据集上训练模型，并将使用此更复杂的训练函数获得的结果与使用 train_model_simple 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

参数高效的 LoRA 微调

低秩自适应（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示例。然而，LoRA 微调也适用于第 7 章中讨论的监督指令微调。

E.1 洛拉简介

LoRA 是一种技术，通过仅调整模型权重参数的小子集，将预训练模型适应特定（通常较小）的数据集，以更好地适应。其中“低秩”方面指的是将模型调整限制在总权重参数空间的小维子空间中的数学概念。这有效地捕捉了训练过程中权重参数变化的最有影响力的方向。LoRA 方法很有用且受欢迎，因为它能够使大型模型在特定任务数据上高效地进行微调，显著降低了通常用于微调的计算成本和资源。

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，我们专注于单个层。

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，该矩阵包含有关我们希望更新原始权重参数以最小化训练过程中的损失函数的信息。此后，我将“权重”一词作为模型权重参数的简称。

在常规训练和微调中，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.

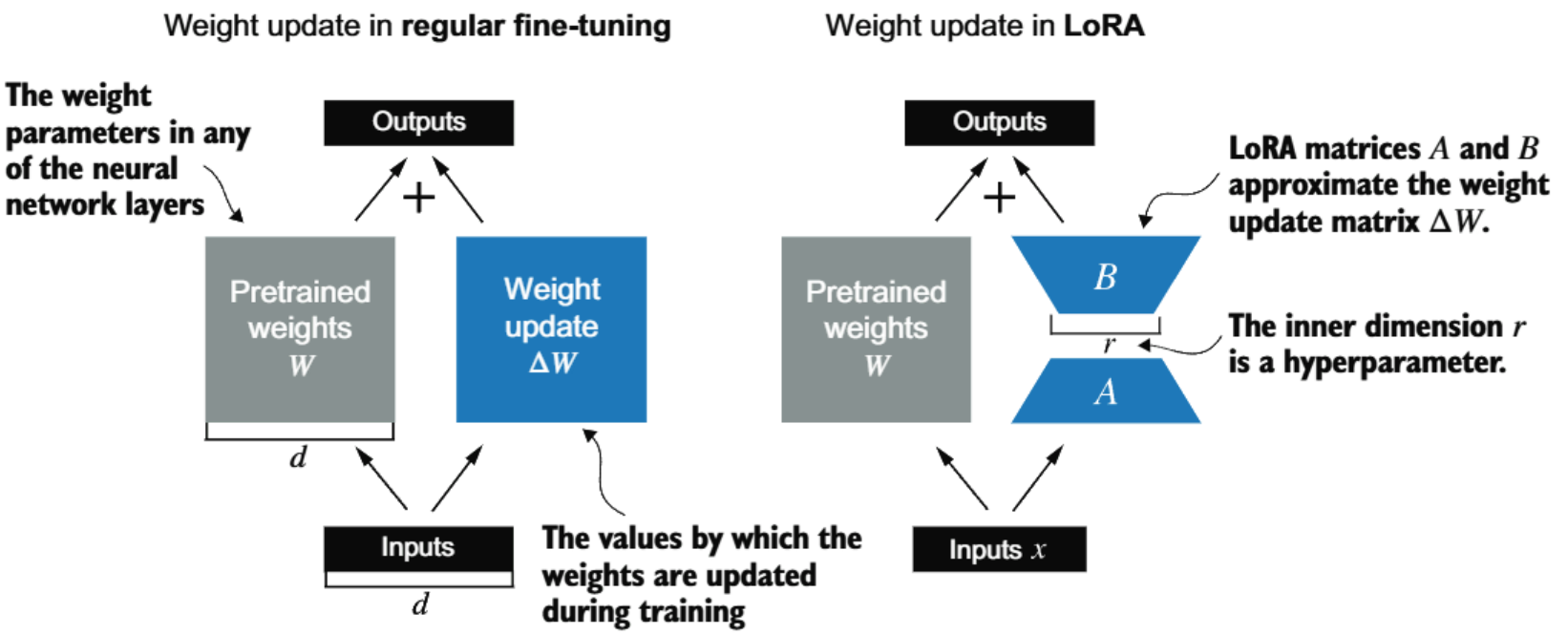


Figure E.1 A comparison between weight update methods: regular fine-tuning and LoRA. Regular fine-tuning involves updating the pretrained weight matrix *W* directly with ΔW (left). LoRA uses two smaller matrices, *A* and *B*, to approximate ΔW, where the product *AB* is added to *W*, and *r* denotes 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

$$x(W + \Delta W) = xW + x\Delta W$$

LoRA 方法，由胡等人提出（https://arxiv.org/abs/2106.09685），通过学习其近似值，为计算权重更新 ΔW 提供了一种更有效的方法：

$$\Delta W \approx AB$$

A 和 B 是两个远小于 W 的矩阵，且 AB 表示 A 和 B 之间的矩阵乘积。

使用 LoRA，我们然后可以重新表述我们之前定义的权重更新：

$$W_{updated} = W + AB$$

图 E.1 展示了全微调和 LoRA 并排的权重更新公式。

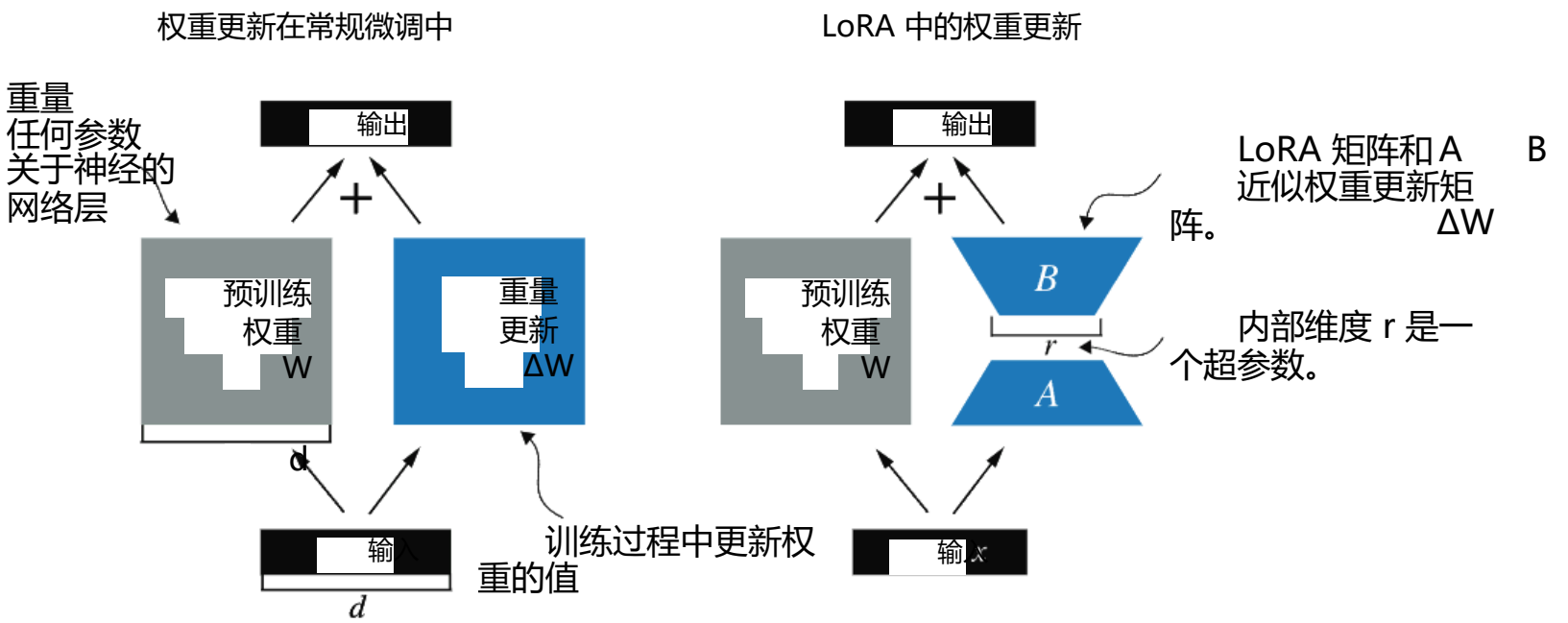


图 E.1 权重更新方法比较：常规微调和 LoRA。常规微调直接用ΔW 更新预训练权重矩阵 W（左）。LoRA 使用两个较小的矩阵 A 和 B 来近似ΔW，其中 AB 的乘积加到 W 上，r 表示内维，是一个可调的超参数（右）。

如果您仔细观察，可能会注意到图 E.1 中全微调和 LoRA 的视觉表示与之前展示的公式略有不同。这种差异归因于矩阵乘法的分配律，它允许我们将原始权重和更新后的权重分开，而不是将它们合并。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)
```

我们可以将计算表示为同样地，我们可以为 LoRA 写出以下内容：

$$x(W + AB) = xW + xAB$$

除了在训练过程中减少需要更新的权重数量外，将 LoRA 权重矩阵与原始模型权重分开的能力使 LoRA 在实际应用中更加有用。实际上，这允许在训练后使用模型时，预训练的模型权重保持不变，而 LoRA 矩阵在训练后动态应用。

将 LoRA 权重分开在实践中有很大用处，因为它允许在不存储多个完整版本的情况下进行模型定制。这减少了存储需求并提高了可扩展性，因为当我们为每个特定客户或应用定制LLMs时，只需调整和保存较小的 LoRA 矩阵即可。

接下来，让我们看看如何使用 LoRA 来微调一个用于垃圾邮件分类的LLM，类似于第 6 章中的微调示例。

E.2 准备数据集

在将 LoRA 应用于垃圾邮件分类示例之前，我们必须加载我们将要使用的 dataset 和预训练模型。这里的代码重复了第 6 章中的数据准备。（我们也可以打开并运行第 6 章的 notebook，并在 E.4 节中插入 LoRA 代码。）

首先，我们下载数据集并将其保存为 CSV 文件。

列表 E.1 下载并准备数据集

```
    from pathlib import Path
导入 pandas 作为 pd 从 ch06 导
入 (
    下载并解压垃圾数据，创建平衡数据
集，随机分割



    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"



    下载并解压垃圾数据（url，zip_path，）                extracted_path，数据文件路径）

    df = pd.read_csv()
    data_file_path，分隔符="\t"，标题=None，名称=["标签"，"文本"] ） balanced_df
= create_balanced_dataset(df) balanced_df["标签"] = balanced_df["标
签"].map({"ham": 0,
                                                                "垃圾邮件": 1})

    train_df, validation_df, test_df =        random_split(balanced_df,)      0.7,   0.1)
    train_df 保存为"train.csv"，不包含索引
```

```
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,
)
```

```
    index=None) 验证数据集.to_csv("validation.csv",
index=None) 测试数据集.to_csv("test.csv", index=None)
```

接下来，我们创建 SpamDataset 实例。

列表 E.2 实例化 PyTorch 数据集

```
    import torch
    从 torch.utils.data 导入 Dataset，导入
tiktoken，从 chapter06 导入 SpamDataset



    tokenizer = tiktoken.get_encoding("gpt2")
train_dataset = SpamDataset("train.csv", max_length=None,
         tokenizer=分词器
    )
    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)
```

在创建 PyTorch 数据集对象后，我们实例化数据加载器。

列表 E.3 创建 PyTorch 数据加载器

```
    from  torch.utils.data 导入 DataLoader

    num_workers = 0
    batch_size = 8

    torch 手动设置随机种子(123)

    train_loader = DataLoader(
         dataset=train_dataset,
     批大小=batch_size, 打乱顺序
     =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, )
数据集=val_dataset, 批大小
=batch_size, 工作进程数

    test_loader = DataLoader(
    dataset=test_dataset, 批大
小=batch_size, 工作进程数
=num_workers, 不丢弃最后
=False, )
```

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

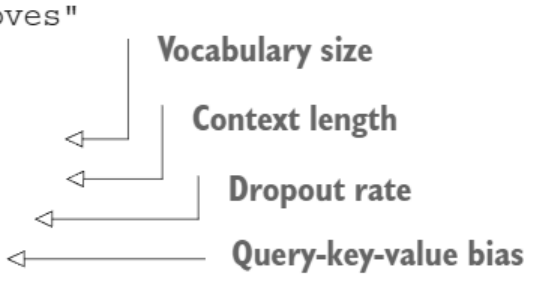
We repeat the code from chapter 6 to load and prepare the pretrained GPT model. We begin by downloading the model weights and loading them into the `GPTModel` class.

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
}
```



作为验证步骤，我们遍历数据加载器，并检查每个批次包含八个训练示例，每个训练示例由 120 个标记组成：

```
    打印("训练加载器：") for 输入批次，目标批次 in 训练
加载器：
        pass

    打印("输入批次维度：", input_batch.shape) 打印("标签批次维
度", target_batch.shape)
```

输出结果

```
    列车加载器：
    输入批次维度：torch.Size([8, 120]) 标签批次维度
torch.Size([8])
```

最后，我们打印出每个数据集中批次的总数：

```
    打印(f"{train_loader 长度} 训练批次") 打印(f"
{val_loader 长度} 验证批次") 打印(f"{test_loader 长度}
测试批次")
```

在这种情况下，我们每个数据集有以下批次数：

```
    130  训练批次
    19  验证批次
    38  测试批次
```

E.3 初始化模型

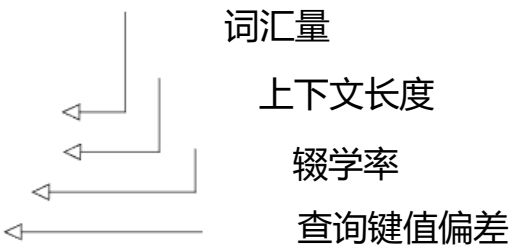
我们重复第 6 章中的代码来加载和准备预训练的 GPT 模型。我们首先下载模型权重，并将它们加载到 GPTModel 类中。

列表 E.4 加载预训练的 GPT 模型

```
    从 gpt_download 导入 download_and_load_gpt2 从
chapter04 导入 GPTModel 从 chapter05 导入
load_weights_into_gpt
```

```
    选择模型 = "gpt2-small (124M)" 输入提示
= "每一步努力"
```

```
    BASE_CONFIG = {
    "vocab_size": 50257,
"context_length": 1024,
"drop_rate": 0.0,
"qkv_bias": 真确 }
```



```
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 corrected, 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 更新(model_configs[选择模型])

    模型大小 = CHOOSE_MODEL.split(" ")[-1].lstrip("(").rstrip(")") 设置，参数
= 下载并加载 gpt2(
    model_size=model_size, 模型目录="gpt2" )


    model = GPT 模型(BASE_CONFIG) 将权重加载
到 gpt(model, params) model.eval()
```

为确保模型正确加载，让我们再次检查它是否生成连贯的文本：

```
    从第四章导入 generate_text_simple
          第五章导入 text_to_token_ids, token_ids_to_text

    输入文本:每一份努力都让你更近一步
text_1
    token_ids = 生成简单文本()
翻译:    model=model, idx=将 text_1 转换为 token_id(text_1,
zh-cn    分词器), max_new_tokens=15,
context_size=BASE_CONFIG["上下文长度"] )
    打印(token_ids_to_text(token_ids, tokenizer))
```

以下输出显示模型生成了连贯的文本，这是模型权重加载正确的指标：

```
    每一份努力都推动你前进。第一步是
                                理解                重要性          of  你的工作
```

接下来，我们为分类微调准备模型，类似于第 6 章，我们替换输出层：

```
    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)
```

最后，我们计算未微调模型的初始分类准确率（我们预计这大约是 50%，这意味着该模型还不能可靠地区分垃圾邮件和非垃圾邮件）：

```
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.

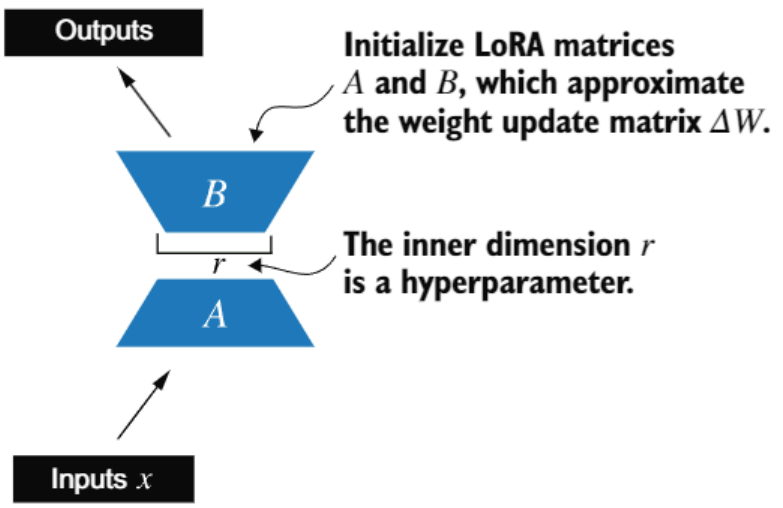


Figure E.2 The LoRA matrices *A* and *B* are applied to the layer inputs and are involved in computing the model outputs. The inner dimension *r* of these matrices serves as a setting that adjusts the number of trainable parameters by varying the sizes of *A* and *B*.

In code, this LoRA layer can be implemented as follows.

```
    从第 06 章导入 calc_accuracy_loader

    torch 手动设置随机种子(123) 训练准确率 =
calc_accuracy_loader()
    train_loader, model, device, num_batches=10)
val_accuracy = calc_accuracy_loader()

    val_loader, model, device, num_batches=10) 测试准确
率 = calc_accuracy_loader()

    test_loader, model, device, num_batches=10 )



    打印训练准确率：{train_accuracy*100:.2f}% 打印验证准确率：
{val_accuracy*100:.2f}% 打印测试准确率：{test_accuracy*100:.2f}%
```

初始预测准确率

```
    训练准确率：46.25% 验证准确
率：45.00% 测试准确率：48.75%
```

E.4 参数高效的 LoRA 微调

接下来，我们使用 LoRA 修改和微调LLM。我们首先初始化一个 LoRALayer，该层创建矩阵 A 和 B，以及 alpha 缩放因子和秩（r）设置。此层可以接受输入并计算相应的输出，如图 E.2 所示。

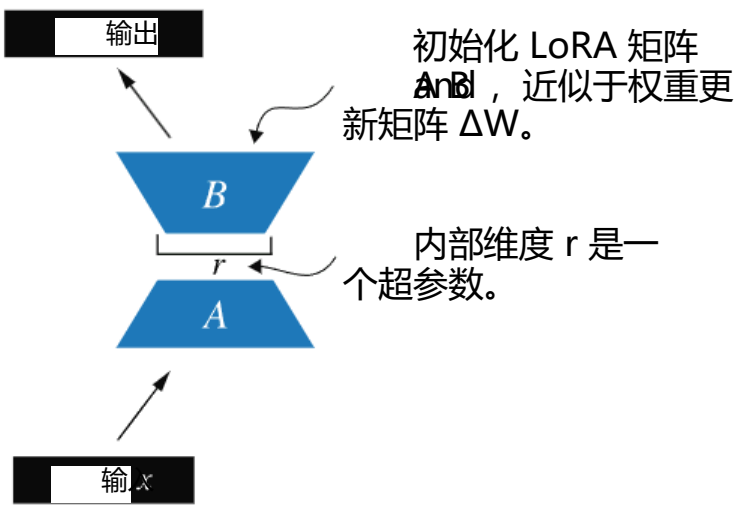


图 E.2 LoRA 矩阵 A 和 B 应用于层输入，并参与计算模型输出。这些矩阵的内维 r 作为设置，通过改变 A 和 B 的大小来调整可训练参数的数量。

在代码中，这个 LoRA 层可以按以下方式实现。

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, `alpha`, 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.

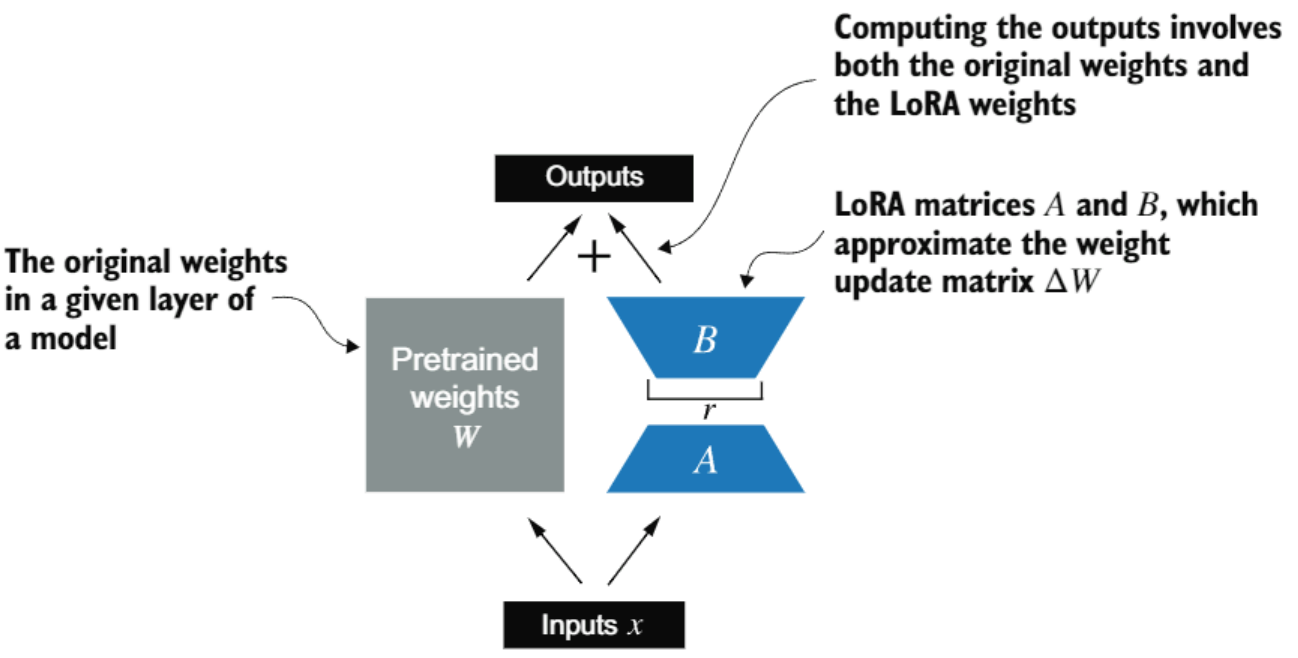


Figure E.3 The integration of LoRA into a model layer. The original pretrained weights (W) of a layer are combined with the outputs from LoRA matrices (A and B), which approximate the weight update matrix (ΔW). The final output is calculated by adding the output of the adapted layer (using LoRA weights) to the original output.

列表 E.5 实现 LoRA 层

```
导入     math

class LoRALayer(洛拉层(torch.nn.Module)):
     def __init__(self, 输入维度, 输出维度, 级别, 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): # 定义前向传播函数
          x = self.alpha * (x @ self.A @ self.B)
      return x
```

用于线性层的相同初始化

在 PyTorch

矩阵 A 和 B 的内维由排名控制。本质上，此设置决定了 LoRA 引入的额外参数数量，通过参数数量在模型的适应性和效率之间创建平衡。

其他重要设置，alpha，作为低秩自适应输出的缩放因子。它主要决定了自适应层输出对原始层输出的影响程度。这可以被视为一种调节低秩自适应对层输出的影响的方法。我们迄今为止实现的 LoRALayer 类使我们能够转换层的输入。

在 LoRA 中，典型目标是用现有线性层进行替换，允许直接将权重更新应用于预训练的现有权重，如图 E.3 所示。

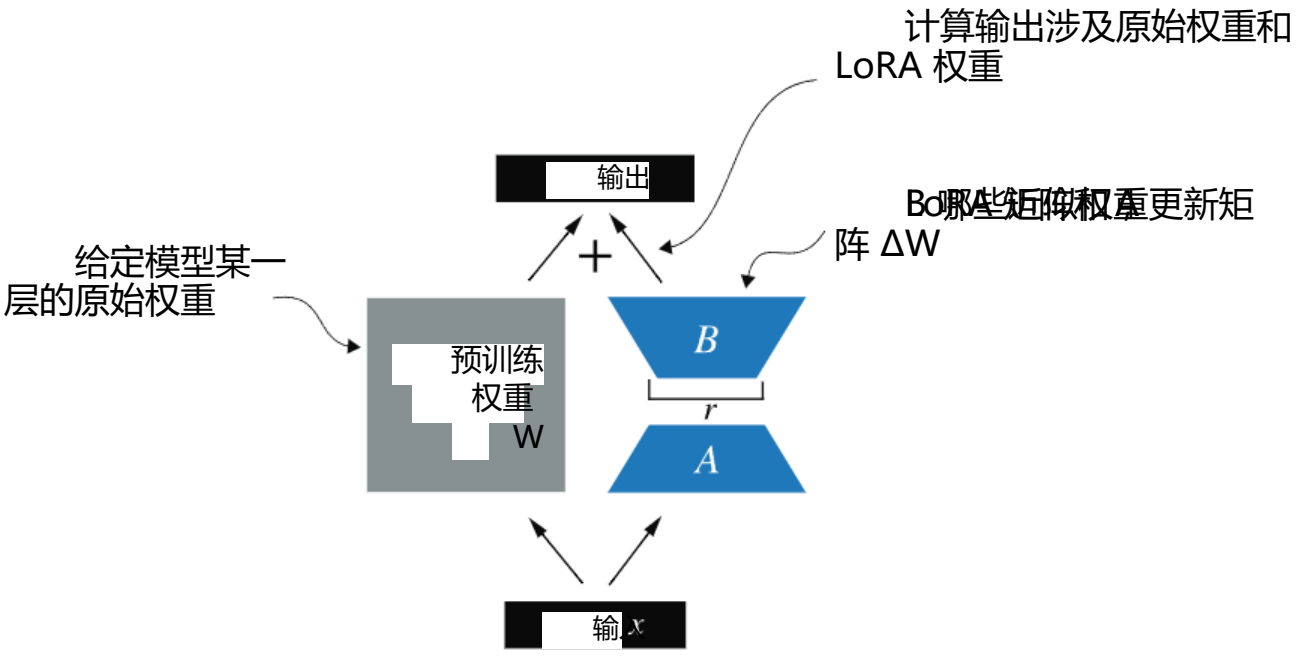


图 E.3 模型层中 LoRA 的集成。层的原始预训练权重（W）与 LoRA 矩阵（A 和 B）的输出相结合，这些输出近似于权重更新矩阵（ΔW）。最终输出是通过将调整后的层（使用 LoRA 权重）的输出添加到原始输出中计算得出的。

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)
```

Replaces the Linear layer with LinearWithLoRA

Recursively applies the same function to child modules

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.

为了整合原始线性层权重，我们现在创建一个 LinearWithLoRA 层。该层利用之前实现的 LoRALayer，旨在替换神经网络中的现有线性层，如 GPTModel 中的自注意力模块或前馈模块。

列表 E.6 替换 LinearWithLora 层为 Linear 层

```
class 线性 LoRA(torch.nn.Module):
    def __init__(self, 线性, 级别, alpha):
        super().__init__()
     self.linear = linear
     self.lora = LoRALayer(
         linear.in_features, linear.out_features, rank, alpha


    def forward(self, x): # 定义前向传播函数
        返回 self.linear(x)       +  self.lora(x)
```

这段代码结合了标准的线性层和 LoRALayer。前向方法通过将原始线性层和 LoRA 层的输出结果相加来计算输出。

由于权重矩阵 B（LoRALayer 中的 self.B）初始化为零值，矩阵 A 和 B 的乘积结果为零矩阵。这确保了乘法不会改变原始权重，因为加上零不会改变它们。

将 LoRA 应用于先前定义的 GPTModel，我们引入了 replace_linear_with_lora 函数。此函数将模型中所有现有的线性层替换为新创建的 LinearWithLoRA 层：

```
def 替换线性层为 LoRA(model, rank, alpha)
    for name, 模块 in 模型.named_children():
        如果 isinstance(module, torch.nn.Linear):
        setattr(model, name, 线性带 LoRA(module, rank, alpha)) else:

            将线性替换为 LoRA(module, rank,              alpha)
```

替换为带有 LoRA 的线性层

递归地对子模块应用相同的函数

我们已经实现了所有必要的代码，以将 GPTModel 中的线性层替换为新开发的 LinearWithLoRA 层，以实现参数高效的微调。接下来，我们将将 LinearWithLoRA 升级应用于 GPTModel 中找到的所有线性层，包括多头注意力、前馈模块和输出层，如图 E.4 所示。

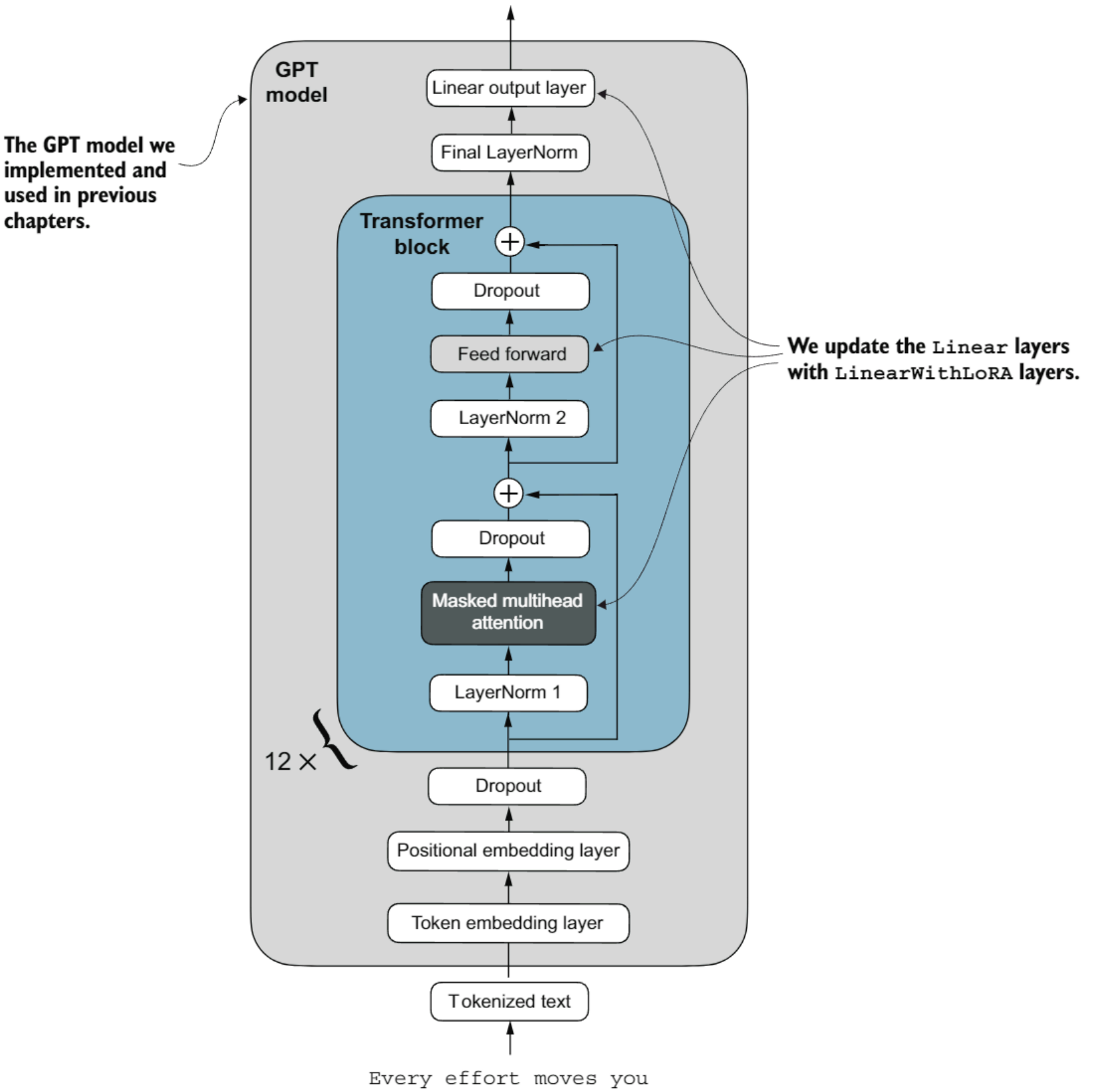


Figure E.4 The architecture of the GPT model. It highlights the parts of the model where `Linear` layers are upgraded to `LinearWithLoRA` layers for parameter-efficient fine-tuning.

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
```

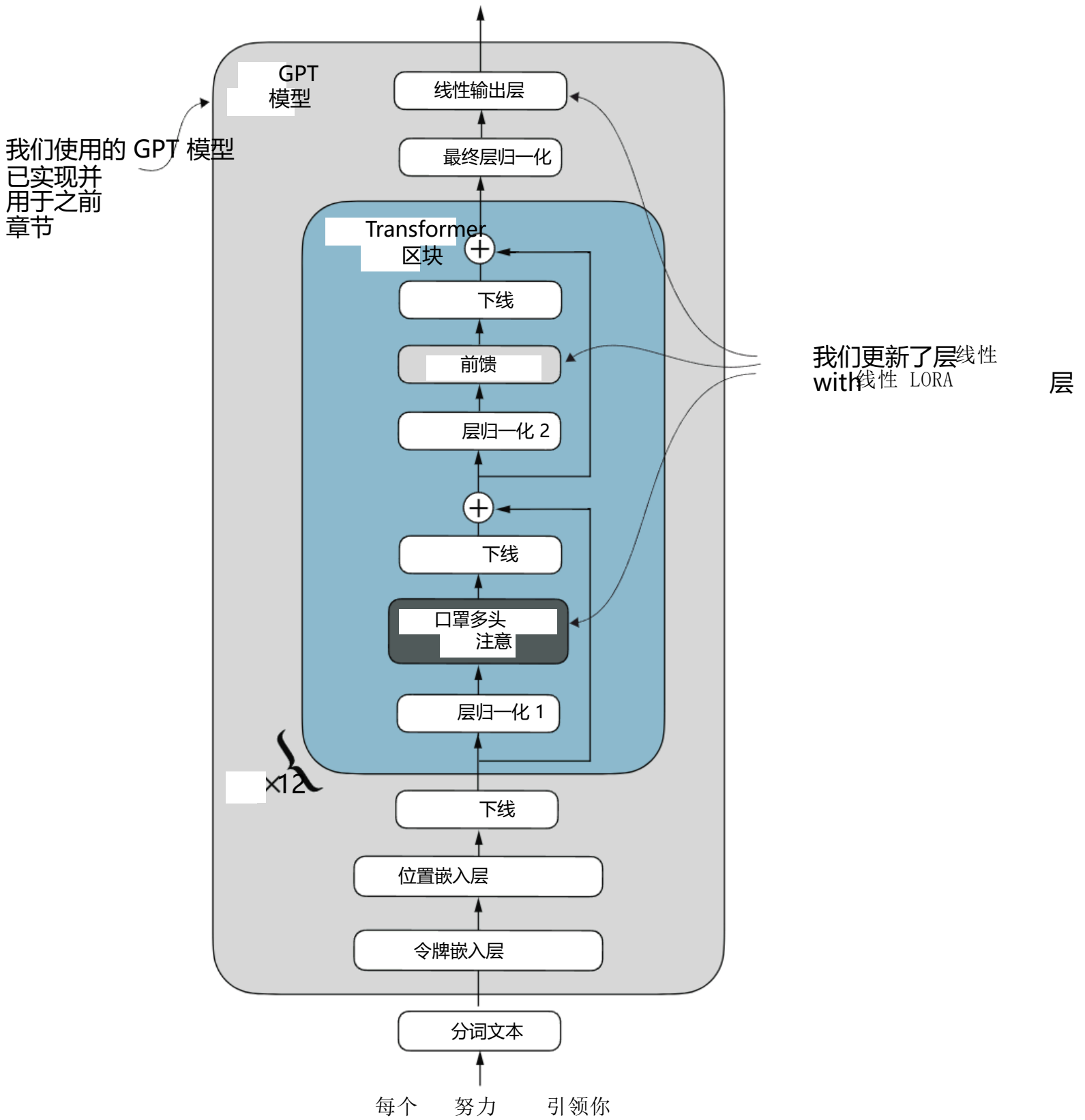


图 E.4 GPT 模型的架构。它突出了模型中线性层升级到 LinearWithLoRA 层以实现参数高效微调的部分。

在我们应用 LinearWithLoRA 层升级之前，我们首先冻结原始模型参数：

```
总参数数（训练前）：{total_params:,}

for param 在 model.parameters() 中：
    参数.需要梯度          = 错误
```

```
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 50× 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()
        )
```

```
总可训练参数数：{total_params:,}
```

现在，我们可以看到这 1.24 亿个模型参数中没有任何一个是可训练的：

```
总训练参数前：124,441,346 总训练参数后：0
```

接下来，我们使用 replace_linear_with_lora 来替换线性层：

```
将线性层替换为 LoRA（模型，rank=16，alpha=16）总参数数 = sum(p.numel() for p in
model.parameters() if p.requires_grad) 打印(f"总可训练 LoRA 参数数: {total_params:,}")
```

在添加 LoRA 层后，可训练参数数量如下：

```
总计    可训练的    LoRA  参数：2,666,528
```

如您所见，使用 LoRA 时，我们几乎将可训练参数的数量减少了 50 倍。16 的秩和 α 是良好的默认选择，但也很常见增加秩参数，这反过来又增加了可训练参数的数量。α 通常选择为秩的一半、两倍或与秩相等。

让我们验证层是否已按预期修改，通过打印模型架构：

```
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu") 模
型.to(device) 打印(model)
```

输出结果

```
   GPT 模型（（tok_emb）：嵌入层（50257，
768）（pos_emb）：嵌入层（1024，768））


      (drop_emb): Dropout(p=0.0, inplace=False)
  (trf_blocks): Sequential( ...)
  (丢弃嵌入): Dropout(p=0.0, inplace=False)
  (trf_blocks): Sequential(    )
       (11): Transformer 块( (att):
    多头注意力(

            (W_query): 线性 LoRA( (linear): 线性(in_features=768, out_features=768,
        bias=True)
            (lora): LoRALayer() ) (W_key): LinearWithLoRA( (linear):
        Linear(in_features=768, out_features=768, bias=True)

        (LoRA 层) ) (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)

        (洛拉): LoRALayer() ) (W_value): LinearWithLoRA( (线性):
        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): 线性带 LoRA( (linear): 线性(in_features=768, out_features=768,
bias=True) (lora): LoRALayer() ) (dropout): Dropout(p=0.0, inplace=False) )
(ff): 前馈( (layers): Sequential(

    (0): LinearWithLoRA( (linear): 线性(in_features=768, out_features=3072,
bias=True) (lora): LoRALayer() ) (1): GELU() (2): LinearWithLoRA( (linear): 线性
(in_features=3072, out_features=768, bias=True) (lora): LoRALayer() ) ) ) (norm1):
层归一化() (norm2): 层归一化()

    (drop_resid): Dropout(p=0.0, inplace=False) ) )
(final_norm): 层归一化() (out_head): 线性带 LoRA()

    (linear): 线性(in_features=768, out_features=2, bias=True) (lora):
LoRALayer() ) )
```

该模型现在包括新的 LinearWithLoRA 层，这些层本身由原始的 Linear 层组成，设置为不可训练，以及新的 LoRA 层，我们将对其进行微调。

在开始微调模型之前，让我们先计算初始分类准确率：

```
    torch 手动设置随机种子(123)

    训练准确率 = calc_accuracy_loader(
    train_loader, model, device, num_batches=10)
val_accuracy = calc_accuracy_loader()

    val_loader, model, device, num_batches=10) 测试准确
率 = calc_accuracy_loader()

    test_loader, 模型, 设备, num_batches=10 )

    打印训练准确率：{train_accuracy*100:.2f}% 打印验证准确率：
{val_accuracy*100:.2f}% 打印测试准确率：{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_minutes:.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%
```

结果准确度值是

```
    训练准确率：46.25% 验证准确
率：45.00% 测试准确率：48.75%
```

这些精度值与第 6 章中的值相同。这个结果发生是因为我们用零初始化了 LoRA 矩阵 B。因此，矩阵 AB 的乘积得到一个零矩阵。这确保了乘法不会改变原始权重，因为加上零不会改变它们。

现在让我们进入激动人心的部分——使用第 6 章中的训练函数微调模型。训练在 M3 MacBook Air 笔记本电脑上大约需要 15 分钟，在 V100 或 A100 GPU 上不到半分钟。

列表 E.7 微调带有 LoRA 层的模型

```
    import time
从 chapter06 导入 train_classifier_simple


    start_time = time.time() torch.manual_seed(123) optimizer =
torch.optim.AdamW(model.parameters(), lr=5e-5,

                                                    weight_decay=0.1) 重量衰减=0.1


    num_epochs = 5
    训练损失，验证损失，训练准确率，验证准确率，已见示例 =
        训练简单分类器
        model, train_loader, val_loader, optimizer, device,
    num_epochs=num_epochs, eval_freq=50, eval_iter=5,
    tokenizer=tokenizer ()


训练完成耗时：{execution_time_minutes}分钟

                                        {执行时间（分钟）:.2f}          分钟。")
```

训练过程中我们看到的输出

```
    第 1 集（步骤 000000）：训练损失 3.820，验证损失 3.462 第 1 集
（步骤 000050）：训练损失 0.396，验证损失 0.364 第 1 集（步骤
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% 第 2 集（步骤 000150）：训练损失 0.135，验证损失
0.073 第 2 集（步骤 000200）：训练损失 0.008，验证损失 0.052 第 2
集（步骤 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% 第 3 集（步骤 000300）：训练损失
0.096，验证损失 0.080 第 3 集（步骤 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% 第 4 集（步骤
000400）：训练损失 0.003，验证损失 0.151 第 4 集（步骤 000450）：训
练损失 0.008，验证损失 0.077 第 4 集（步骤 000500）：训练损失
0.001，验证损失 0.147 训练准确率：100.00% | 验证


                                        准确度：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.

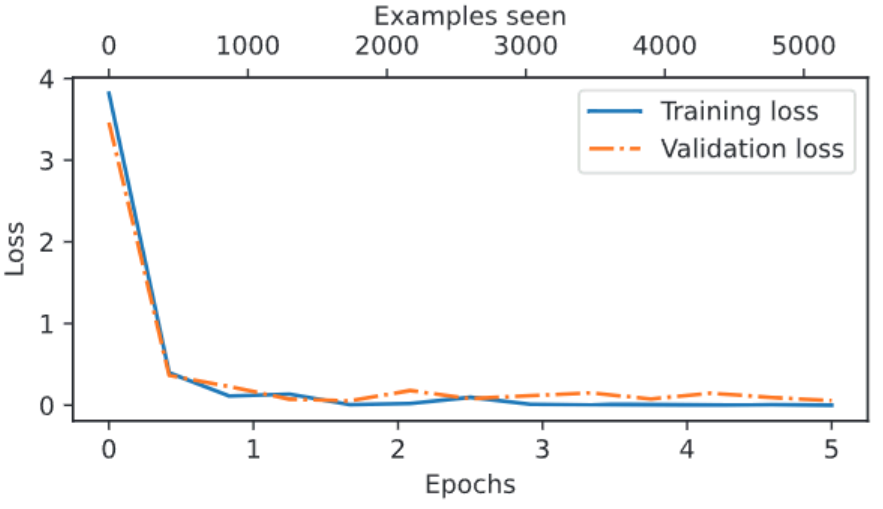


Figure E.5 The training and validation loss curves over six epochs for a machine learning model. Initially, both training and validation loss decrease sharply and then they level off, indicating the model is converging, which means that it is not expected to improve noticeably with further training.

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}%")
```

```
    第 5 集（步骤 000550）：训练损失 0.007，验证损失 0.094 第 5
集（步骤 000600）：训练损失 0.000，验证损失 0.056 50% 训练准确
率：100.00% | 验证准确率：
                                                        97.50%

    训练完成                    在 12.10    分钟。
```

训练模型使用 LoRA 比不使用 LoRA 所需时间更长（见第 6 章），因为 LoRA 层在正向传播过程中引入了额外的计算。然而，对于更大的模型，由于反向传播变得成本更高，使用 LoRA 的模型通常比不使用 LoRA 的模型训练得更快。

如您所见，该模型接受了完美的训练，并且验证准确度非常高。让我们也可视化损失曲线，以便更好地观察训练是否收敛：

```
    from  第六章 导入绘图值

    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, 标签="损失" )
```

图 E.5 展示了结果。

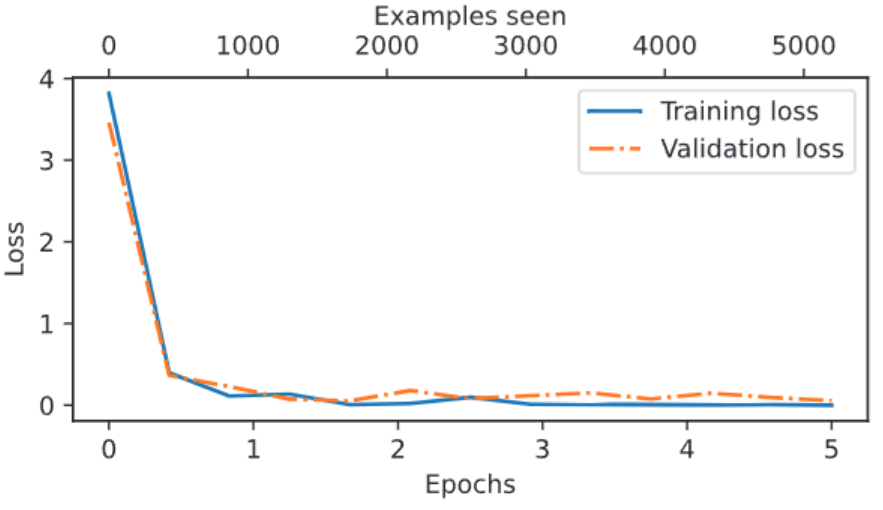


图 E.5 机器学习模型在六个时期内的训练和验证损失曲线。最初，训练和验证损失急剧下降，然后趋于平稳，表明模型正在收敛，这意味着进一步训练不太可能显著提高。

除了根据损失曲线评估模型外，还要计算整个训练集、验证集和测试集的准确率（在训练过程中，我们通过五个批次近似计算了训练集和验证集的准确率）

```
    eval_iter=5 设置):

    训练准确率 = calc_accuracy_loader(train_loader, model, device) 验证准确率 =
calc_accuracy_loader(val_loader, model, device) 测试准确率 =
calc_accuracy_loader(test_loader, model, device)

    打印训练准确率：{train_accuracy*100:.2f}% 打印验证准确率：
{val_accuracy*100:.2f}% 打印测试准确率：{test_accuracy*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).

结果准确度值是

```
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，该模型在训练、验证和测试数据集上表现良好。训练准确率达到100%，模型已经完美地学习了训练数据。然而，验证和测试准确率（分别为 96.64%和97.33%）略低，表明存在一定程度过拟合，因为模型在未见过的数据上的泛化能力不如训练集。总体而言，考虑到我们仅微调了相对较少的模型权重（270 万 LoRA 权重而不是原始的 1.24 亿模型权重），结果非常令人印象深刻。

index

Symbols

Numerics

A

B

C

索引

符号

数值

A

B

C

D

E

D

E

F

G

I

F

G

I

K

L

M

N

K

L

M

N

O

P

Q

R

O

P

Q

R

S

T

S

T

U

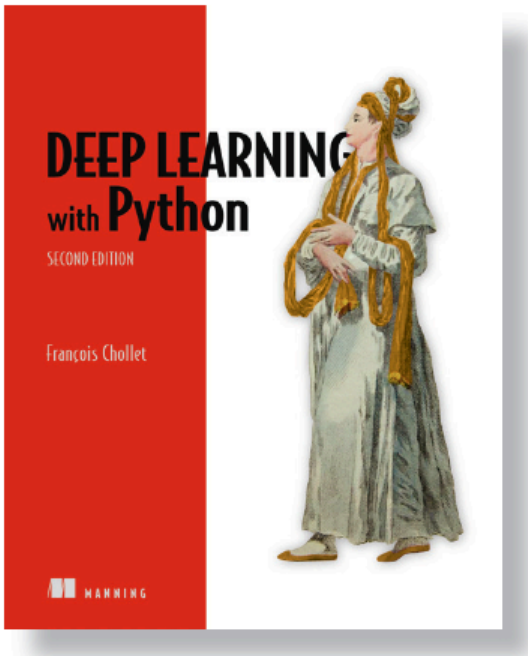
V

W

X

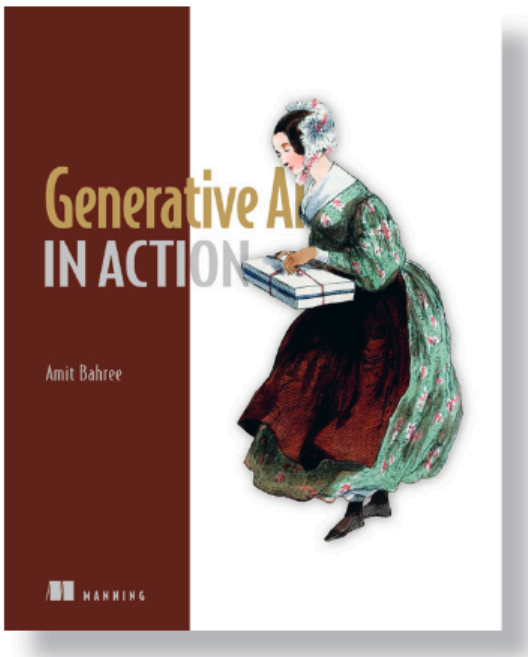
Z

RELATED MANNING TITLES



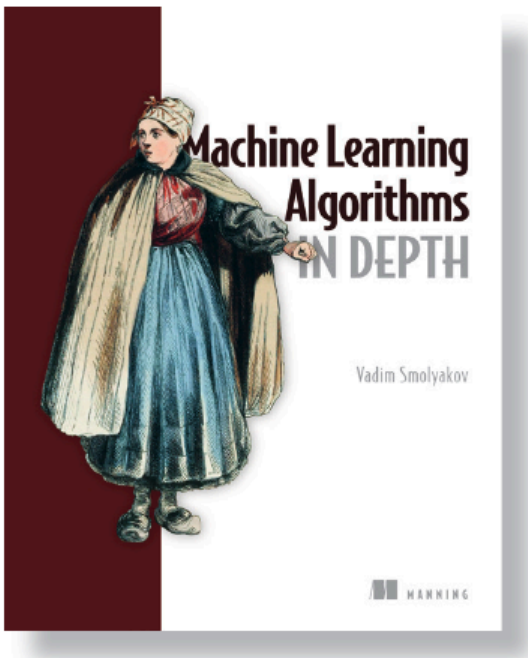
Deep Learning with Python, Second Edition
by Francois Chollet

ISBN 9781617296864
504 pages, $59.99
October 2021



Generative AI in Action
by Amit Bahree

ISBN 9781633436947
469 pages (estimated), $59.99
October 2024 (estimated)

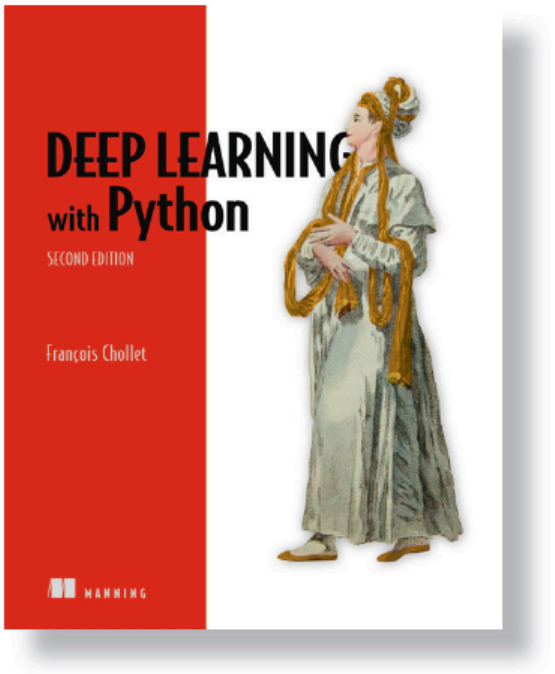


Machine Learning Algorithms in Depth
by Vadim Smolyakov

ISBN 9781633439214
328 pages, $79.99
July 2024

For ordering information, go to www.manning.com

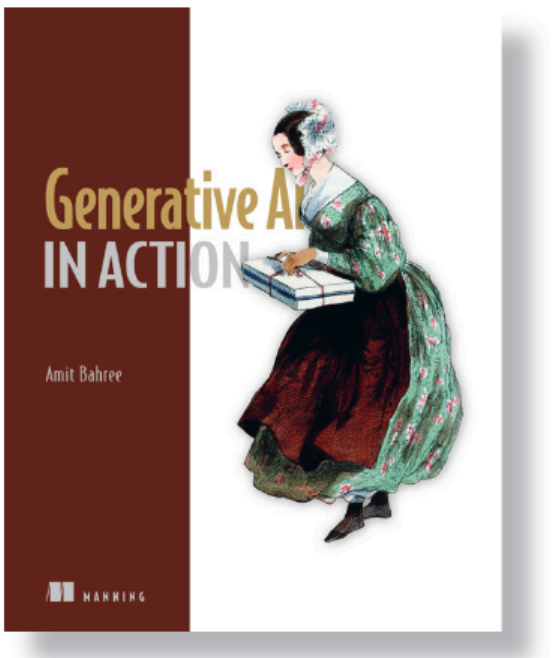
相关曼宁标题



深度学习与 Python，第 2 版

弗朗索瓦·肖莱

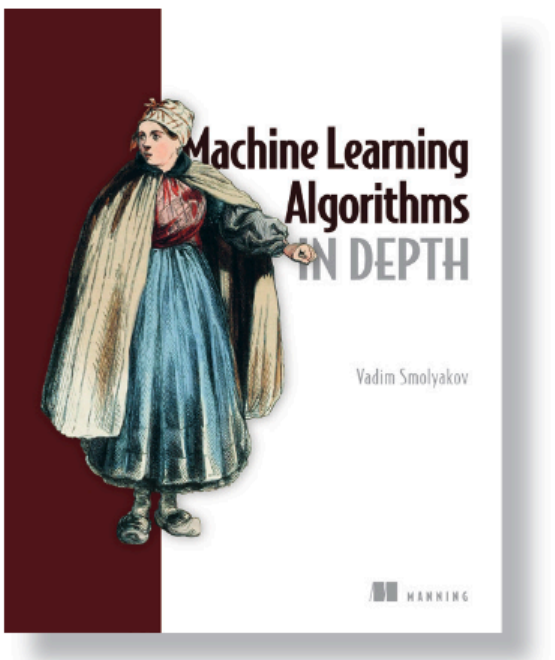
ISBN 9781617296864
504 页，59.99 美元
2021 年 10 月



生成式 AI 实践

阿米特·巴雷

ISBN 9781633436947
469 页（预估），59.99 美元，2024 年 10 月（预估）



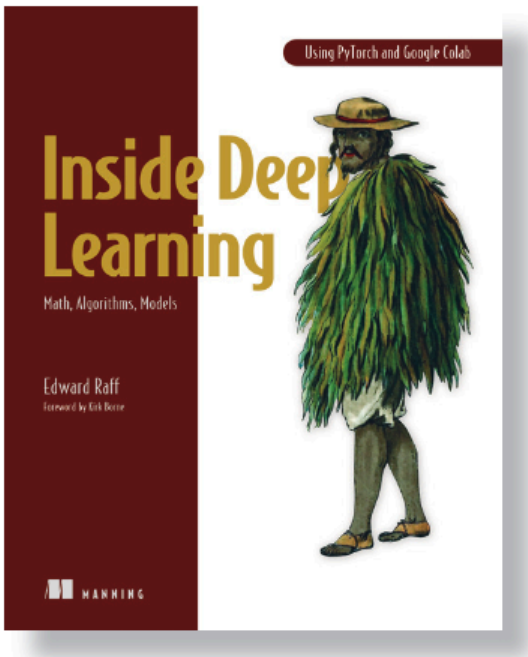
机器学习算法深度

由瓦迪姆·斯莫利亚科夫

ISBN 9781633439214
328 页，79.99 美元，2024 年 7 月

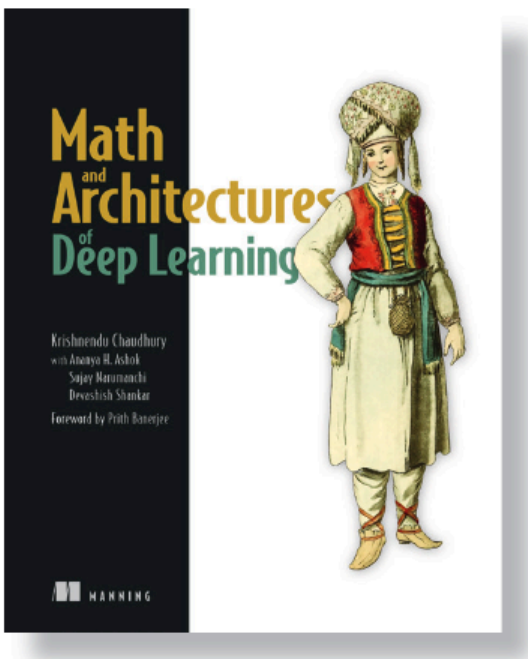
有关订购信息，请访问 www.manning.com

RELATED MANNING TITLES



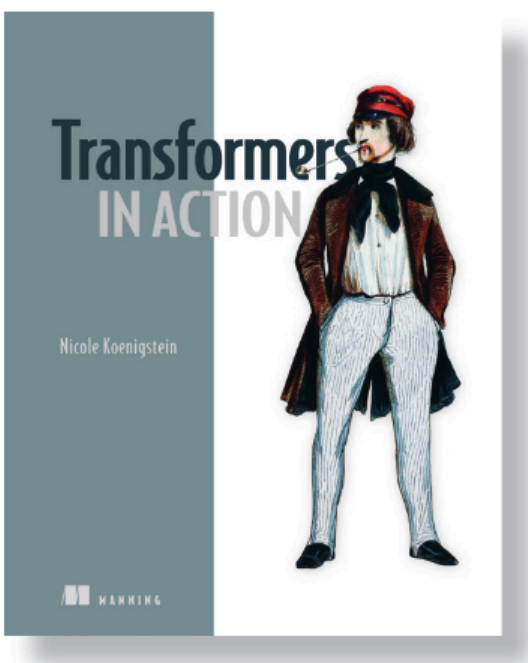
Inside Deep Learning
by Edward Raff
Foreword by Kirk Borne

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

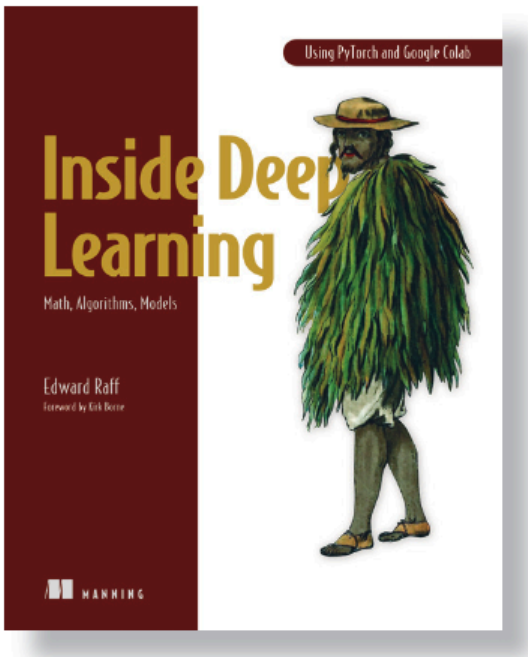


Transformers in Action
by Nicole Koenigstein

ISBN 9781633437883
393 pages (estimated), $59.99
February 2025 (estimated)

For ordering information, go to www.manning.com

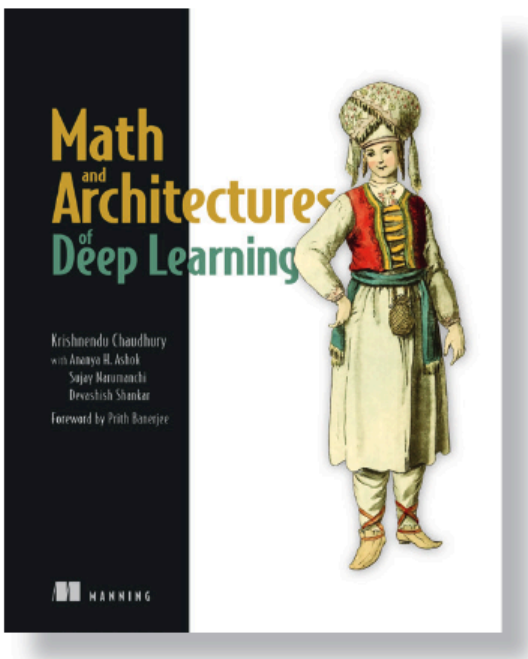
相关曼宁标题



深度学习内部

爱德华·拉夫 前言
由柯克·伯恩撰写

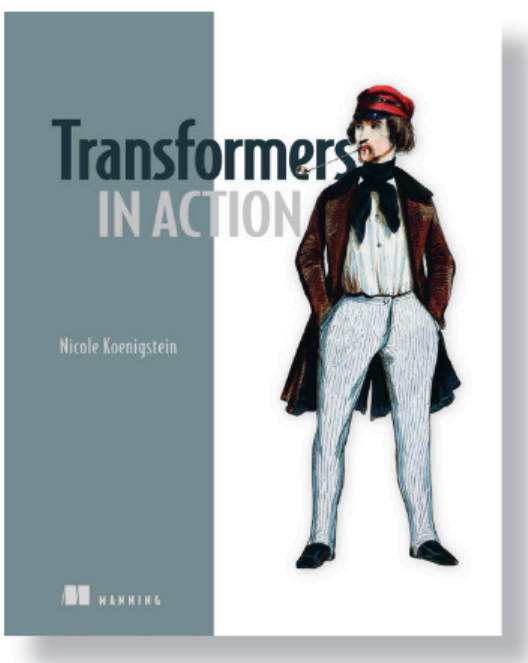
ISBN 9781617298639
600 页，59.99 美元
2022 年 4 月



数学与深度学习架构

由克里希嫩杜·乔杜里、安亚娜·阿什克、苏贾伊·纳鲁马奇、德瓦希什·尚卡尔作序，普里特·班纳杰前言

ISBN 9781617296482
552 页，$69.99
四月 2024



Transformer 实战

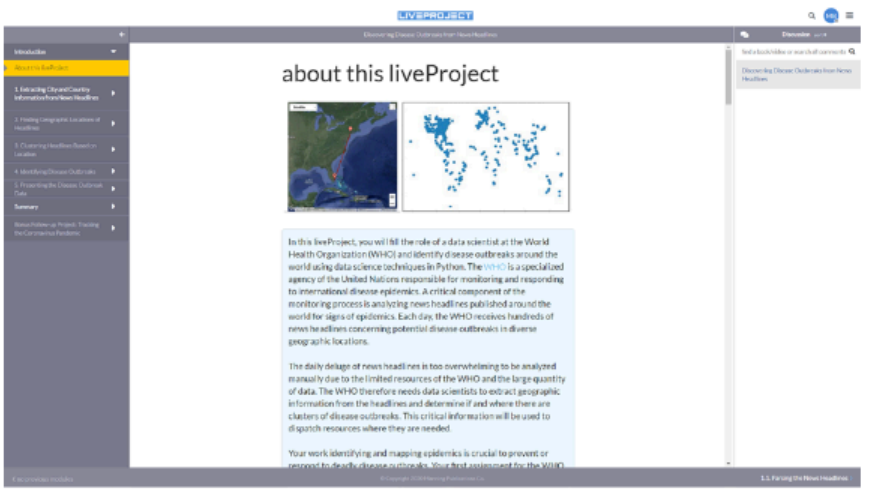
Nicole Koenigstein

ISBN 9781633437883
393 页（预估），59.99 美元，2025 年 2 月（预估）

有关订购信息，请访问 www.manning.com

MANNING

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.

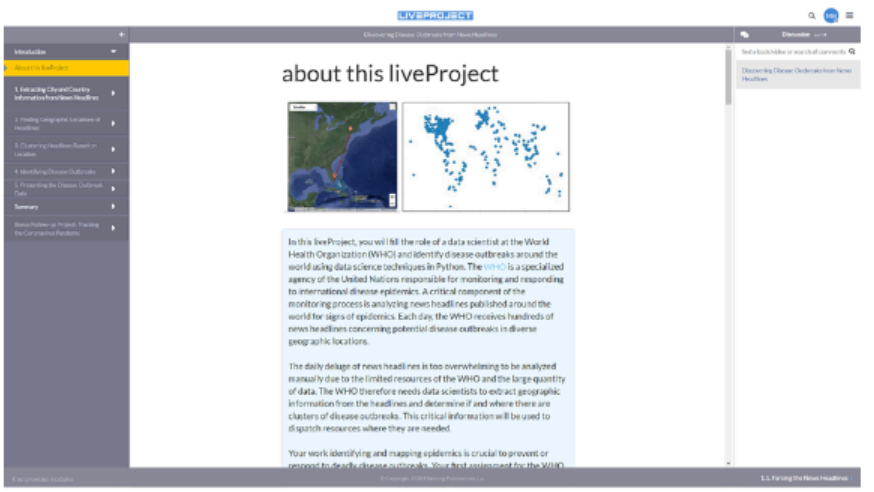
Here's how liveProject works:

- **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!

MANNING

LIVEPROJECT



动手项目，学习你的方式

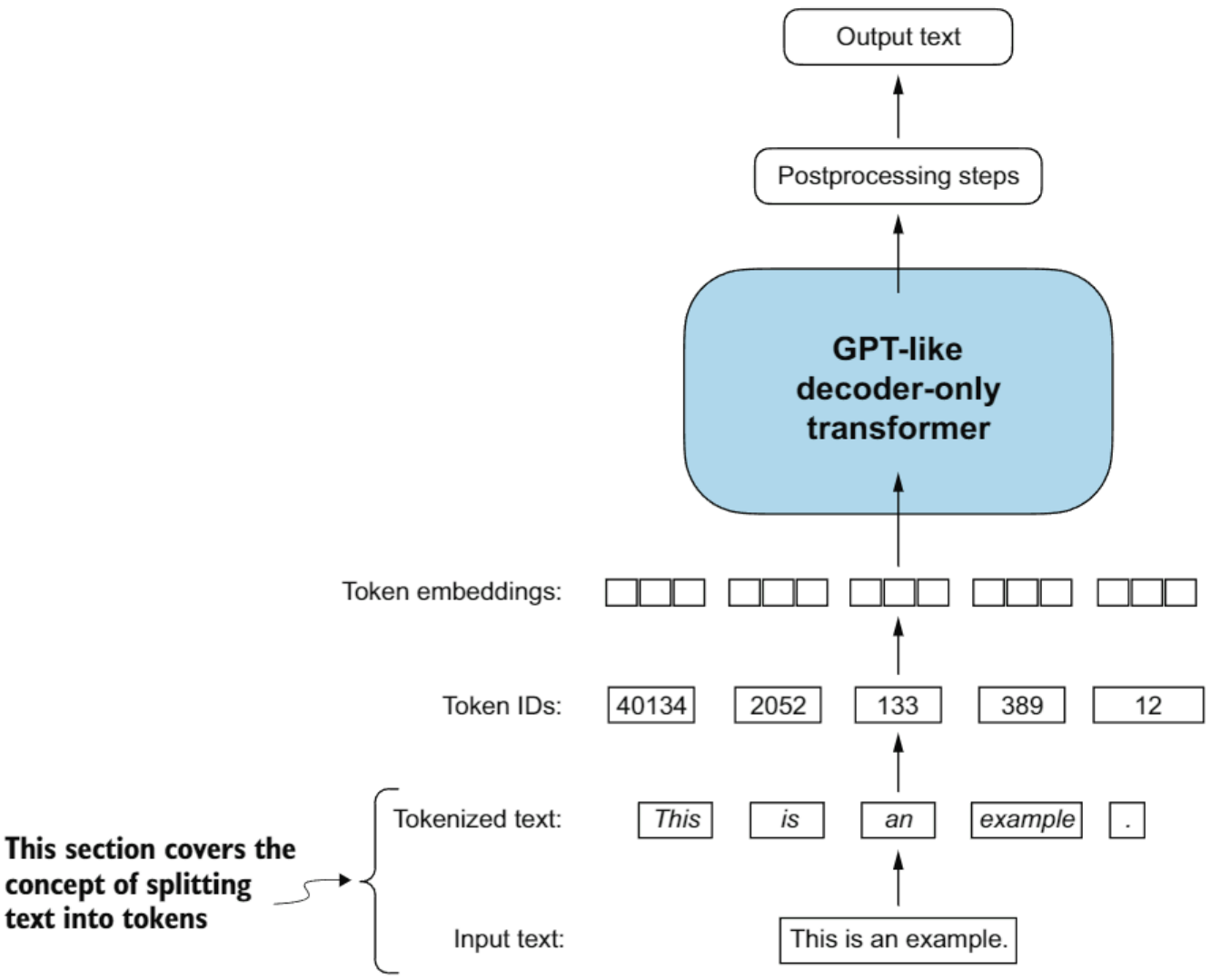
实时项目是提升技能的激动人心的方式，就像在工作中学习一样。

在一个 Manning liveProject 中，你将面对真实的 IT 挑战并制定自己的解决方案。为确保你成功，你将获得 90 天对精选的 Manning 书籍和视频资源的全面且无限访问权限。

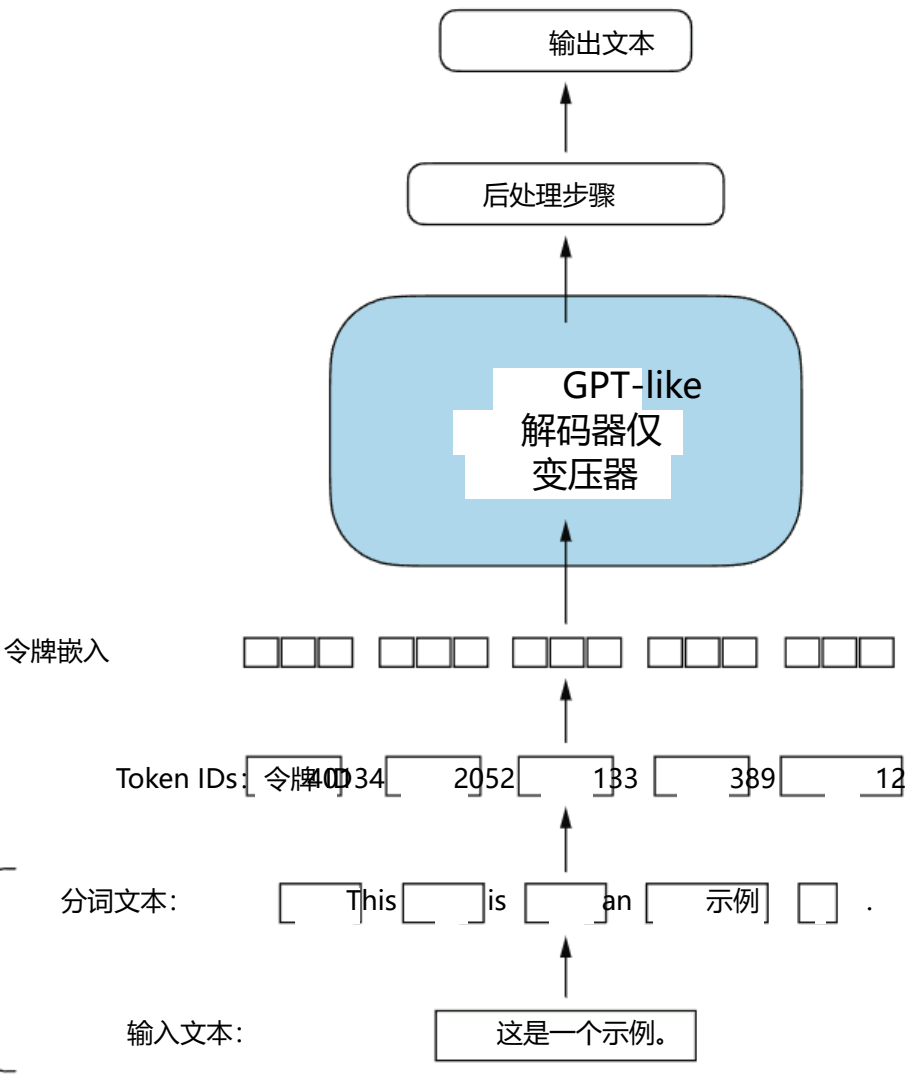
这里是如何使用 liveProject 的：

- 可达成里程碑。每个项目都被分解为步骤和部分，以便您可以跟踪您的进度。
- 协作和建议。通过聊天、工作组以及同行项目评审与其他 liveProject 参与者合作。
- 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。
- 一切成功所需。每个 liveProject 都附带数据集和精心挑选的学习资源。
- 构建你的投资组合。所有 liveProjects 教授的技能都是当前需求之列的行业。当你完成时，你将获得成功带来的满足感，以及一个真正可以添加到你的作品集的项目。

探索 www.manning.com 上的数十个数据、开发和云工程现场项目！



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.



本节涵盖将文本拆分为标记的概念

文本处理步骤在LLM的上下文中的视图。过程从输入文本开始，将其分解为标记，然后将其转换为数值标记 ID。这些 ID 与作为 GPT 模型输入的标记嵌入相关联。该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。此流程说明了标记化的基本操作。

嵌入、转换和后处理，这些在本书从头至尾实现的 GPT 模型中都有涉及。

AI

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

MANNING

"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

AI

构建一个 大型语言模型

塞巴斯蒂安·拉斯卡

物理学家理查德·费曼据说曾说过：“我不理解我无法构建的东西。”基于这个同样强大的原则，畅销书作者塞巴斯蒂安·拉斯克卡一步步引导你构建一个可以在你的笔记本电脑上运行的 GPT-style LLM。这是一本引人入胜的书，涵盖了整个过程的所有阶段，从规划、编码到训练和微调。

构建大型语言模型（从零开始）是一本实用的

这是一次令人满意的亲身体验，深入探索生成式 AI 的基础。无需依赖任何现有的LLM库，您将编写一个基础模型，将其进化为文本分类器，最终创建一个能够遵循您对话指令的聊天机器人。而且，您将真正理解它，因为它是您自己构建的！

里面的内容

- 规划和编写一个类似于 GPT-2 的LLM
- 加载预训练权重
- 构建完整的训练流程
- 微调您的LLM进行文本分类
- 开发 LLM 以下文本遵循人类指令

读者需要具备中级 Python 技能和一些机器学习知识。您创建的LLM可以在任何现代笔记本电脑上运行，并可选择性使用 GPU。

塞巴斯蒂安·拉斯赫卡是 Lightning AI 的资深研究工程师，在那里他从事LLM研究并开发开源软件。

这本书的技术编辑是 David Caswell。

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

曼宁

真正鼓舞人心！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

这是迄今为止对语言模型最易懂和最全面的解释！它独特的实用教学风格达到了其他方式无法达到的理解水平。”

—卡梅隆·沃尔夫 高级科学家，Netflix

塞巴斯蒂安结合了深厚的知识、实用的工程技能以及将复杂想法简化的天赋。这就是你需要的手册！”

—奇普·黄，作者
设计机器学习系统
人工智能工程

权威、最新内容覆盖。强烈推荐！”

—瓦希德·米尔贾利利博士，高级数据 科学家，FM 全球

查看第一页

ISBN-13: 978-1-63343-716-6