4 Implementing a GPT model from scratch to generate text

This chapter covers

- Coding a GPT-like large language model (LLM) that can be trained to generate human-like text
- Normalizing layer activations to stabilize neural network training
- Adding shortcut connections in deep neural networks
- Implementing transformer blocks to create GPT models of various sizes
- Computing the number of parameters and storage requirements of GPT models

You've already learned and coded the *multi-head attention* mechanism, one of the core components of LLMs. Now, we will code the other building blocks of an LLM and assemble them into a GPT-like model that we will train in the next chapter to generate human-like text.

The LLM architecture referenced in figure 4.1, consists of several building blocks. We will begin with a top-down view of the model architecture before covering the individual components in more detail.

4 实施从零开始构建GPT 模型生成文本

本章涵盖

- 编写一个类似 GPT 的大语言模型（LLM），可以训练生成类似人类的文本
- 将层激活归一化以稳定神经网络训练
- 在深度神经网络中添加快捷连接
- 实现不同尺寸的 GPT 模型所需的 Transformer 块
- 计算 GPT 模型的参数数量和存储需求

您已经学习和编写了多头注意力机制，这是LLMs的核心组件之一。现在，我们将编写LLM的其他构建模块，并将它们组装成一个类似 GPT 的模型，我们将在下一章中对其进行训练以生成类似人类的文本。

图 4.1 中提到的LLM架构由几个构建块组成。我们将从模型架构的顶层视图开始，然后再更详细地介绍各个组件。

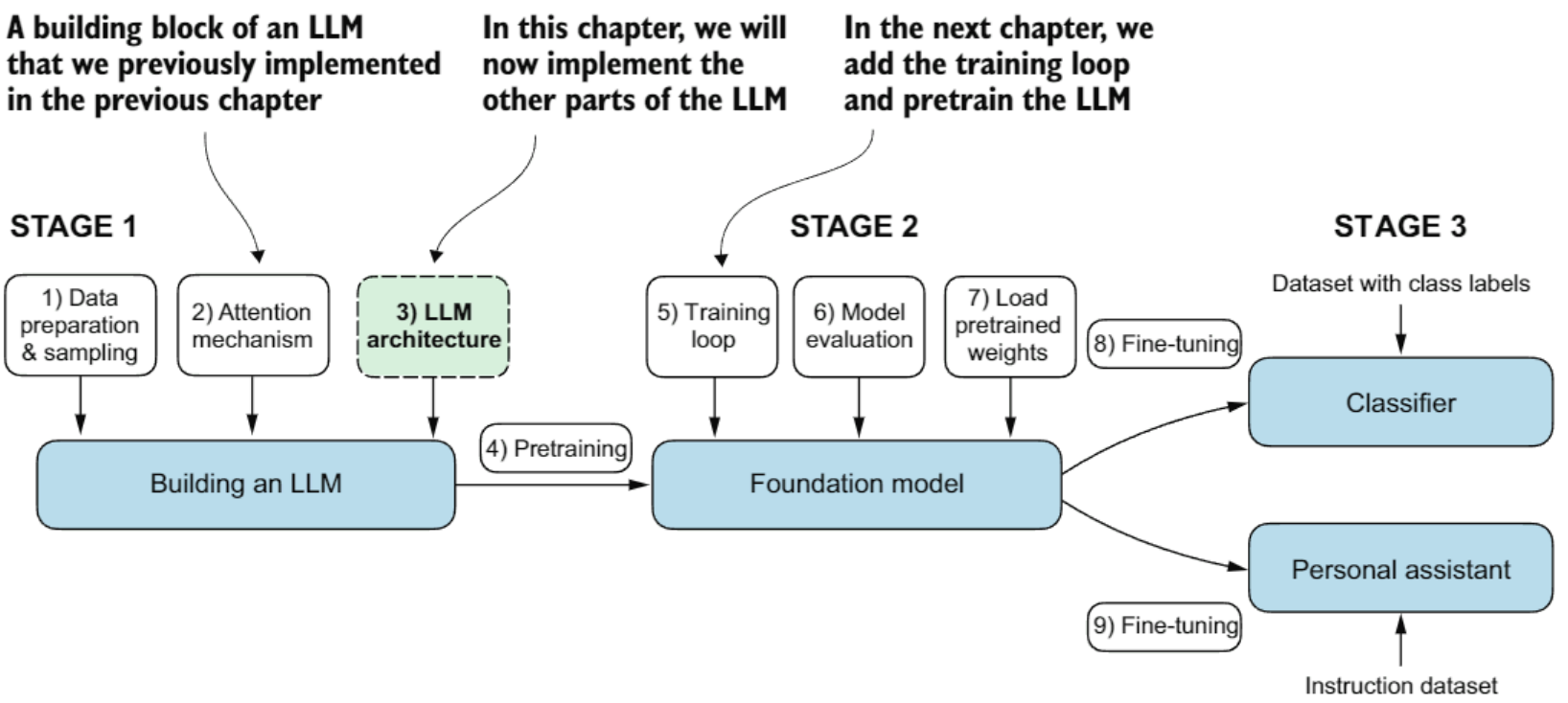

Figure 4.1 The three main stages of coding an LLM. This chapter focuses on step 3 of stage 1: implementing the LLM architecture.

4.1 Coding an LLM architecture

LLMs, such as GPT (which stands for *generative pretrained transformer*), are large deep neural network architectures designed to generate new text one word (or token) at a time. However, despite their size, the model architecture is less complicated than you might think, since many of its components are repeated, as we will see later. Figure 4.2 provides a top-down view of a GPT-like LLM, with its main components highlighted.

We have already covered several aspects of the LLM architecture, such as input tokenization and embedding and the masked multi-head attention module. Now, we will implement the core structure of the GPT model, including its *transformer blocks*, which we will later train to generate human-like text.

Previously, we used smaller embedding dimensions for simplicity, ensuring that the concepts and examples could comfortably fit on a single page. Now, we are scaling up to the size of a small GPT-2 model, specifically the smallest version with 124 million parameters, as described in "Language Models Are Unsupervised Multitask Learners," by Radford et al. (https://mng.bz/yoBq). Note that while the original report mentions 117 million parameters, this was later corrected. In chapter 6, we will focus on loading pretrained weights into our implementation and adapting it for larger GPT-2 models with 345, 762, and 1,542 million parameters.

In the context of deep learning and LLMs like GPT, the term "parameters" refers to the trainable weights of the model. These weights are essentially the internal variables of the model that are adjusted and optimized during the training process to minimize a specific loss function. This optimization allows the model to learn from the training data.

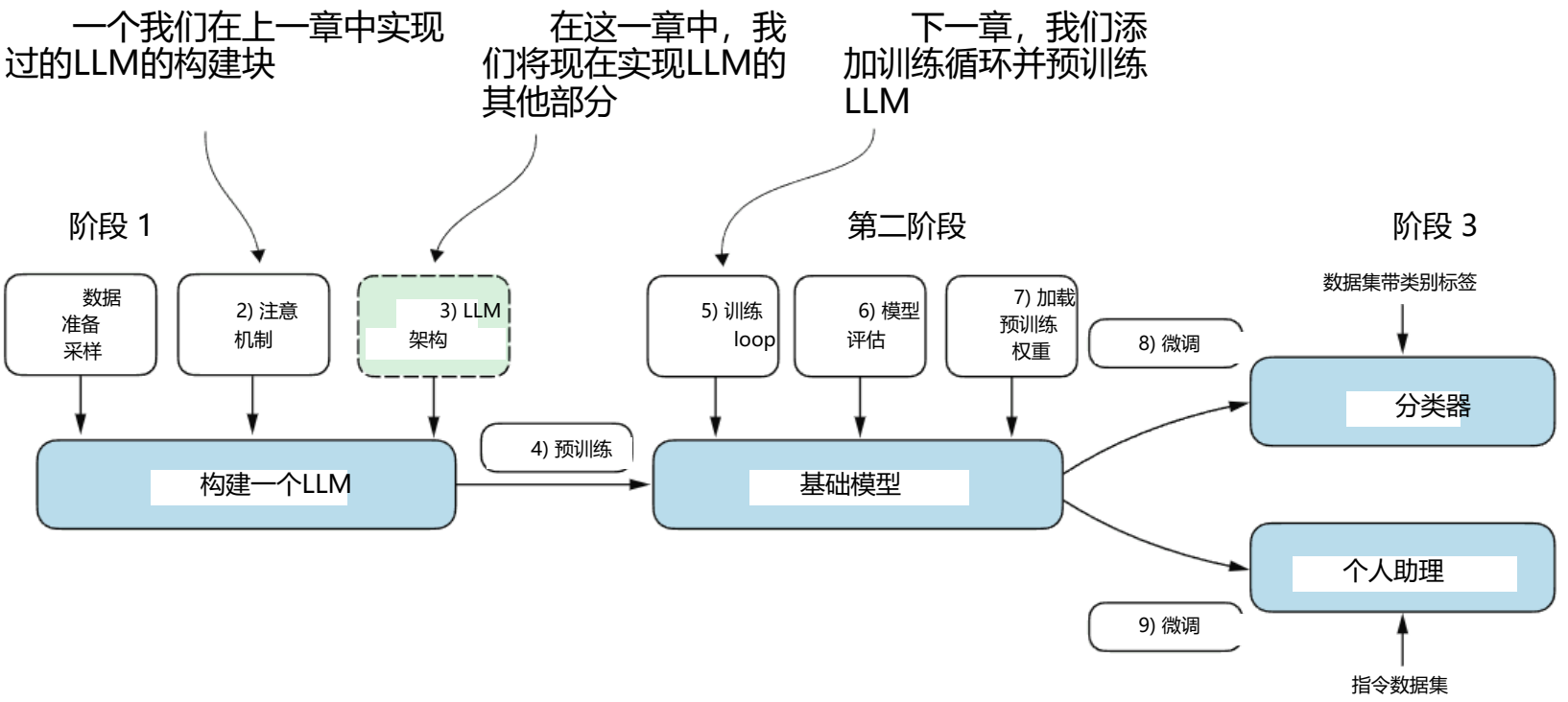

图 4.1 LLM编码的三个主要阶段。本章重点介绍第一阶段第 3 步：实现LLM架构。

4.1 编写一个LLM架构

LLMs，例如 GPT（代表生成式预训练变换器），是设计用来逐词（或标记）生成新文本的大型深度神经网络架构。然而，尽管它们的规模很大，但模型架构并没有你想象的那么复杂，因为其中许多组件是重复的，正如我们稍后将会看到的。图 4.2 提供了一个类似 GPT 的LLM的俯视图，其中其主要组件被突出显示。

我们已经涵盖了LLM架构的几个方面，例如输入分词和嵌入以及掩码多头注意力模块。现在，我们将实现 GPT 模型的核心结构，包括其变换器块，我们将在以后训练它们以生成类似人类的文本。

之前，我们为了简便使用了较小的嵌入维度，确保概念和示例可以舒适地放在一页上。现在，我们正在扩大到小型 GPT-2 模型的大小，具体是最小的 124 百万参数版本，如 Radford 等人所描述的“语言模型是无监督的多任务学习者”（https://mng.bz/yoBq）。请注意，虽然原始报告提到 117 百万参数，但这后来被更正了。在第 6 章中，我们将专注于将预训练的权重加载到我们的实现中，并适应具有 345 亿、762 亿和 15.42 亿参数的更大 GPT-2 模型。

在深度学习和 GPT 等LLMs的背景下，“参数”一词指的是模型的可训练权重。这些权重实际上是模型在训练过程中调整和优化的内部变量，以最小化特定的损失函数。这种优化使模型能够从训练数据中学习。

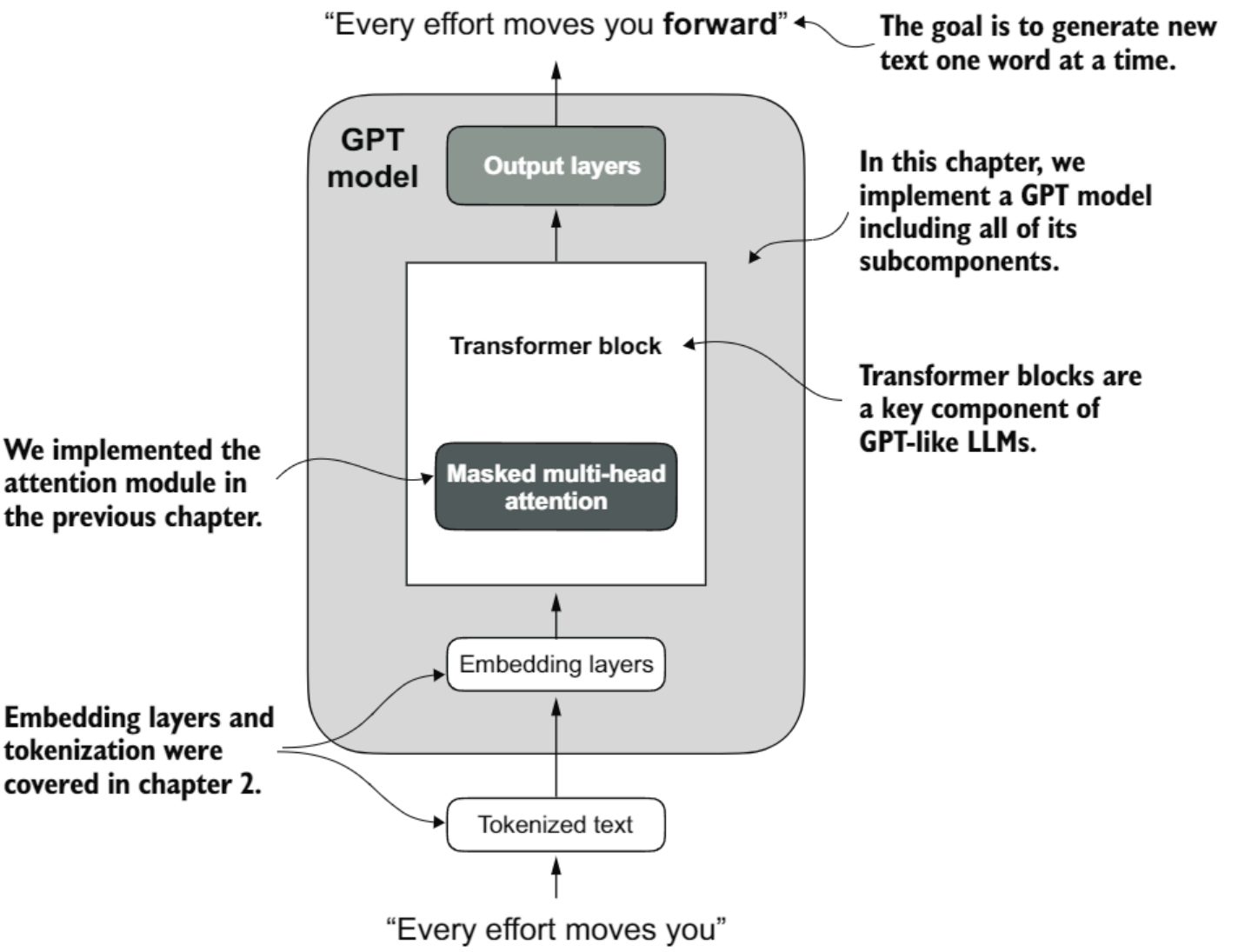

Figure 4.2 A GPT model. In addition to the embedding layers, it consists of one or more transformer blocks containing the masked multi-head attention module we previously implemented.

For example, in a neural network layer that is represented by a 2,048 × 2,048–dimensional matrix (or tensor) of weights, each element of this matrix is a parameter. Since there are 2,048 rows and 2,048 columns, the total number of parameters in this layer is 2,048 multiplied by 2,048, which equals 4,194,304 parameters.

GPT-2 vs. GPT-3

Note that we are focusing on GPT-2 because OpenAI has made the weights of the pretrained model publicly available, which we will load into our implementation in chapter 6. GPT-3 is fundamentally the same in terms of model architecture, except that it is scaled up from 1.5 billion parameters in GPT-2 to 175 billion parameters in GPT-3, and it is trained on more data. As of this writing, the weights for GPT-3 are not publicly available. GPT-2 is also a better choice for learning how to implement LLMs, as it can be run on a single laptop computer, whereas GPT-3 requires a GPU cluster for training and inference. According to Lambda Labs (https://lambdalabs.com/), it would take 355 years to train GPT-3 on a single V100 datacenter GPU and 665 years on a consumer RTX 8000 GPU.

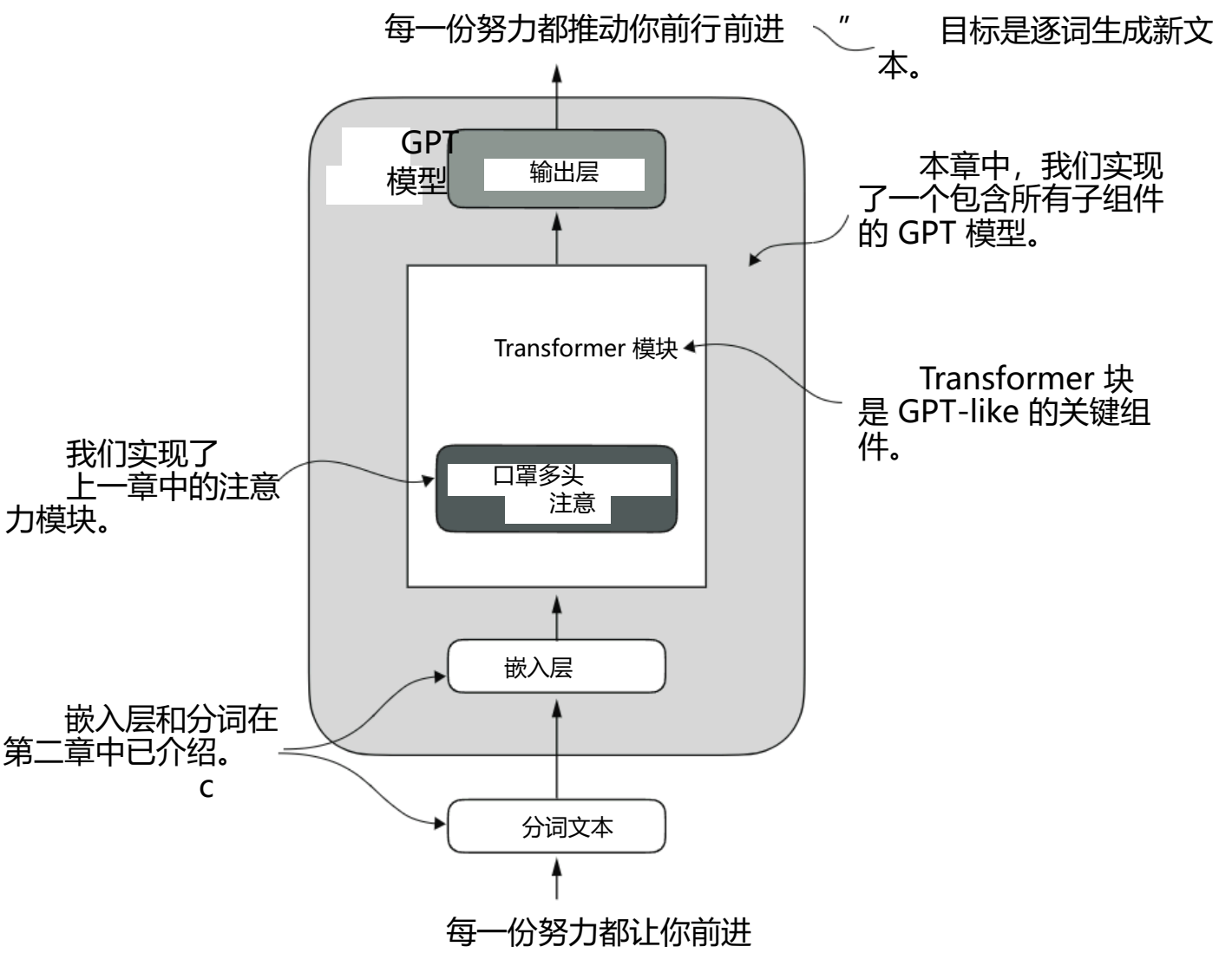

图 4.2 一个 GPT 模型。除了嵌入层之外，它还包括一个或多个包含我们之前实现的掩码多头注意力模块的 Transformer 块。

例如，在一个由 2,048 × 2,048 维度的权重矩阵（或张量）表示的神经网络层中，这个矩阵的每个元素都是一个参数。由于有 2,048 行和 2,048 列，这个层中的参数总数是 2,048 乘以 2,048，等于 4,194,304 个参数。

GPT-2 对比 GPT-3

请注意，我们专注于 GPT-2，因为 OpenAI 已将预训练模型的权重公开，我们将在第 6 章中将这些权重加载到我们的实现中。GPT-3 在模型架构方面基本上是相同的，只是它从 GPT-2 的 15 亿参数扩展到 GPT-3 的 1750 亿参数，并且训练数据更多。截至本文写作时，GPT-3 的权重尚未公开。GPT-2 也是学习如何在LLMs中实现更好的选择，因为它可以在单个笔记本电脑上运行，而 GPT-3 需要 GPU 集群进行训练和推理。根据 Lambda Labs（https://lambdalabs.com/）的数据，在单个 V100 数据中心 GPU 上训练 GPT-3 需要 355 年，在消费级 RTX 8000 GPU 上需要 665 年。

We specify the configuration of the small GPT-2 model via the following Python dictionary, which we will use in the code examples later:

```
GPT_CONFIG_124M = {
    "vocab_size": 50257,     # Vocabulary size
    "context_length": 1024,  # Context length
    "emb_dim": 768,          # Embedding dimension
    "n_heads": 12,           # Number of attention heads
    "n_layers": 12,          # Number of layers
    "drop_rate": 0.1,        # Dropout rate
    "qkv_bias": False        # Query-Key-Value bias
}
```

In the `GPT_CONFIG_124M` dictionary, we use concise variable names for clarity and to prevent long lines of code:

- `vocab_size` refers to a vocabulary of 50,257 words, as used by the BPE tokenizer (see chapter 2).
- `context_length` denotes the maximum number of input tokens the model can handle via the positional embeddings (see chapter 2).
- `emb_dim` represents the embedding size, transforming each token into a 768-dimensional vector.
- `n_heads` indicates the count of attention heads in the multi-head attention mechanism (see chapter 3).
- `n_layers` specifies the number of transformer blocks in the model, which we will cover in the upcoming discussion.
- `drop_rate` indicates the intensity of the dropout mechanism (0.1 implies a 10% random drop out of hidden units) to prevent overfitting (see chapter 3).
- `qkv_bias` determines whether to include a bias vector in the `Linear` layers of the multi-head attention for query, key, and value computations. We will initially disable this, following the norms of modern LLMs, but we will revisit it in chapter 6 when we load pretrained GPT-2 weights from OpenAI into our model (see chapter 6).

Using this configuration, we will implement a GPT placeholder architecture (`DummyGPTModel`), as shown in figure 4.3. This will provide us with a big-picture view of how everything fits together and what other components we need to code to assemble the full GPT model architecture.

The numbered boxes in figure 4.3 illustrate the order in which we tackle the individual concepts required to code the final GPT architecture. We will start with step 1, a placeholder GPT backbone we will call `DummyGPTModel`.

我们通过以下 Python 字典指定小型 GPT-2 模型的配置，该配置将在后面的代码示例中使用：

```
GPT_CONFIG_124M = {
"vocab_size": 50257, # 词汇量大小 "context_length": 1024, # 上下文长度 "emb_dim": 768, # 嵌入维度 "n_heads": 12, # 注意力头数 "n_layers": 12, # 层数 "drop_rate": 0.1, # 抖动率
"qkv_bias": False # Query-Key-Value 偏置 }
```

在 GPT_CONFIG_124M 字典中，我们使用简洁的变量名以提高清晰度并防止代码行过长：

- 词汇大小指的是一个包含 50,257 个单词的词汇表，该词汇表由 BPE 分词器使用（见第 2 章）。
- context_length 表示模型通过位置嵌入可以处理的最大输入标记数（见第 2 章）。
- emb_dim 表示嵌入大小，将每个标记转换为 768 维向量。
- n_heads 表示多头注意力机制中注意力头的数量（见第 3 章）。
- n_layers 指定了模型中 transformer 块的数目，我们将在接下来的讨论中介绍。
- drop_rate 表示 dropout 机制的强度（0.1 表示隐藏单元随机丢弃 10%以防止过拟合（见第 3 章））。
- qkv_bias 决定是否在多头注意力的查询、键和值计算中线性层的线性层中包含偏置向量。我们最初将禁用此功能，遵循现代LLMs规范，但在我们将从 OpenAI 加载预训练的 GPT-2 权重到我们的模型时（见第 6 章），我们将重新审视它。

使用此配置，我们将实现一个 GPT 占位符架构（DummyGPTModel），如图 4.3 所示。这将为我们提供一个整体视图，了解所有组件如何组合以及我们需要编写哪些其他组件来组装完整的 GPT 模型架构。

图 4.3 中的编号框展示了我们解决编码最终 GPT 架构所需各个概念的顺序。我们将从步骤 1 开始，使用一个名为 DummyGPTModel 的占位符 GPT 骨干。

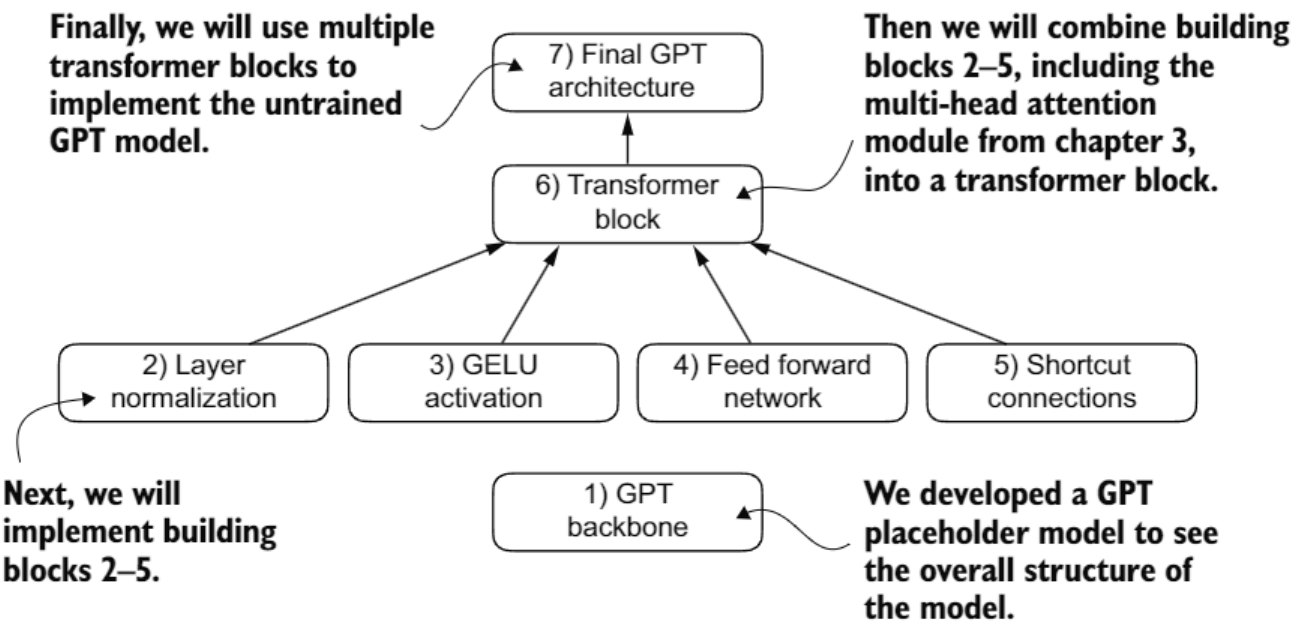

Figure 4.3 The order in which we code the GPT architecture. We start with the GPT backbone, a placeholder architecture, before getting to the individual core pieces and eventually assembling them in a transformer block for the final GPT architecture.

Listing 4.1 A placeholder GPT model architecture class

```
import torch
import torch.nn as nn

class DummyGPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["emb_dim"])
        self.drop_emb = nn.Dropout(cfg["drop_rate"])
        self.trf_blocks = nn.Sequential(
            *[DummyTransformerBlock(cfg)
              for _ in range(cfg["n_layers"])]
        )
        self.final_norm = DummyLayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(
            cfg["emb_dim"], cfg["vocab_size"], bias=False
        )

    def forward(self, in_idx):
        batch_size, seq_len = in_idx.shape
        tok_embeds = self.tok_emb(in_idx)
        pos_embeds = self.pos_emb(
            torch.arange(seq_len, device=in_idx.device)
        )
        x = tok_embeds + pos_embeds
        x = self.drop_emb(x)
        x = self.trf_blocks(x)
        x = self.final_norm(x)
        logits = self.out_head(x)
        return logits
```

Uses a placeholder for TransformerBlock

Uses a placeholder for LayerNorm

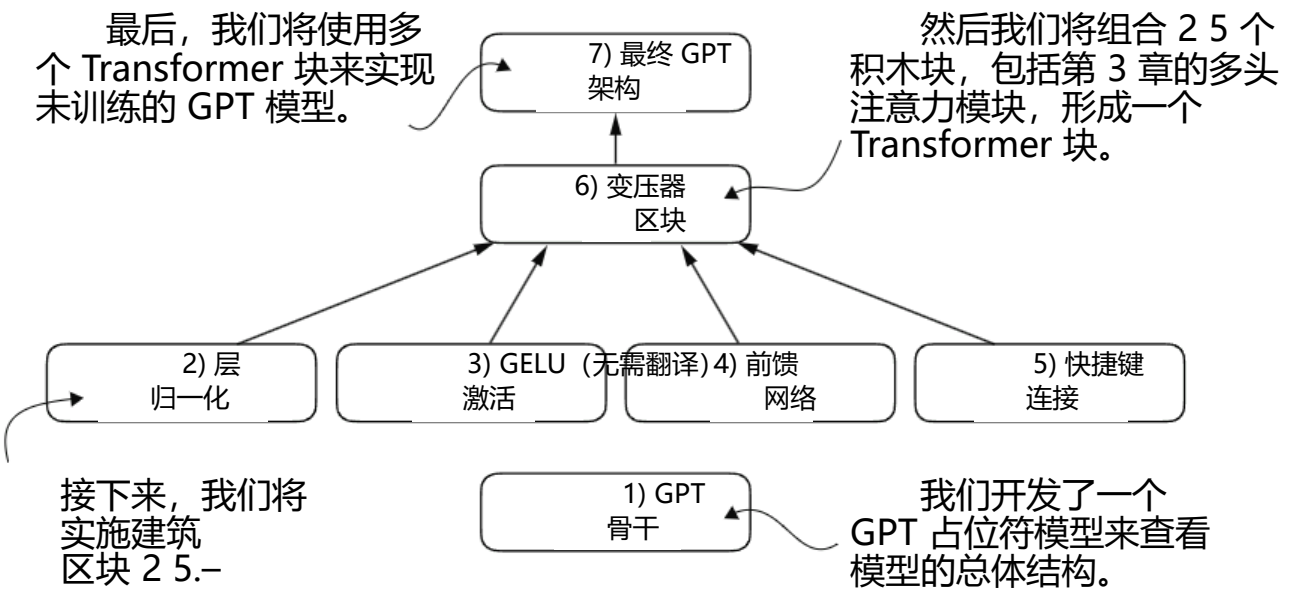

图 4.3 我们编码 GPT 架构的顺序。我们首先从 GPT 骨干架构开始，这是一个占位符架构，然后到达各个核心组件，最终将它们组装在 transformer 块中，形成最终的 GPT 架构。

列表 4.1 占位符 GPT 模型架构类

```
import torch
导入     torch.nn   作为 nn
    神经网络

class DummyGPT 模型(nn.Module):
def __init__(self, cfg): # 初始化(self, 配置)
    super().__init__()
    self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_dim"])
self.pos_emb = nn.Embedding(cfg["context_length"], cfg["emb_dim"])
self.drop_emb = nn.Dropout(cfg["drop_rate"])
self.trf_blocks = nn.Sequential()

    *[模拟 Transformer 块（cfg）for _ in
range(cfg["n_layers"])] self.final_norm = 模拟层归一化
（cfg["emb_dim"]） self.out_head = nn.Linear(

    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 logits
```

使用占位符表示 TransformerBlock

使用占位符用于 LayerNorm

```
class DummyTransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()

    def forward(self, x):
        return x

class DummyLayerNorm(nn.Module):
    def __init__(self, normalized_shape, eps=1e-5):
        super().__init__()

    def forward(self, x):
        return x
```

A simple placeholder class that will be replaced by a real TransformerBlock later

This block does nothing and just returns its input.

A simple placeholder class that will be replaced by a real LayerNorm later

The parameters here are just to mimic the LayerNorm interface.

The `DummyGPTModel` class in this code defines a simplified version of a GPT-like model using PyTorch's neural network module (`nn.Module`). The model architecture in the `DummyGPTModel` class consists of token and positional embeddings, dropout, a series of transformer blocks (`DummyTransformerBlock`), a final layer normalization (`DummyLayerNorm`), and a linear output layer (`out_head`). The configuration is passed in via a Python dictionary, for instance, the `GPT_CONFIG_124M` dictionary we created earlier.

The `forward` method describes the data flow through the model: it computes token and positional embeddings for the input indices, applies dropout, processes the data through the transformer blocks, applies normalization, and finally produces logits with the linear output layer.

The code in listing 4.1 is already functional. However, for now, note that we use placeholders (`DummyLayerNorm` and `DummyTransformerBlock`) for the transformer block and layer normalization, which we will develop later.

Next, we will prepare the input data and initialize a new GPT model to illustrate its usage. Building on our coding of the tokenizer (see chapter 2), let's now consider a high-level overview of how data flows in and out of a GPT model, as shown in figure 4.4.

To implement these steps, we tokenize a batch consisting of two text inputs for the GPT model using the tiktoken tokenizer from chapter 2:

```
import tiktoken

tokenizer = tiktoken.get_encoding("gpt2")
batch = []
txt1 = "Every effort moves you"
txt2 = "Every day holds a"

batch.append(torch.tensor(tokenizer.encode(txt1)))
batch.append(torch.tensor(tokenizer.encode(txt2)))
batch = torch.stack(batch, dim=0)
print(batch)
```

```
    class DummyTransformerBlock(nn.Module):
类 DummyTransformerBlock(nn.Module):
          def __init__(self, cfg):
               super().__init__()

          def forward(self, x): # 定义前向传播函数
               返回 x


    class DummyLayerNorm（nn.Module）：
          def __init__(self, 标准化形状, eps=1e-5):
               super().__init__()

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

一个简单的占位符类，稍后将被真正的 TransformerBlock 替换

此块不执行任何操作，仅返回其输入。

一个简单的占位符类，稍后将被真正的 LayerNorm 替换

这里的参数只是为了模仿 LayerNorm 接口。

该代码中的 DummyGPTModel 类定义了一个使用 PyTorch 的神经网络模块（nn.Module）的 GPT 类似模型的简化版本。DummyGPTModel 类中的模型架构包括标记和位置嵌入、dropout、一系列的转换器块（DummyTransformerBlock）、最终的层归一化（DummyLayerNorm）和线性输出层（out_head）。配置通过 Python 字典传入，例如我们之前创建的 GPT_CONFIG_124M 字典。

前向方法描述了模型中的数据流：它为输入索引计算标记和位置嵌入，应用 dropout，通过 Transformer 块处理数据，应用归一化，并最终通过线性输出层产生 logits。

代码列表 4.1 中的代码已经可用。然而，目前请注意，我们使用占位符（DummyLayerNorm 和 DummyTransformerBlock）来表示变换块和层归一化，这些我们将在以后开发。

接下来，我们将准备输入数据并初始化一个新的 GPT 模型以展示其用法。基于我们对分词器的编码（见第 2 章），现在让我们考虑一下数据如何在 GPT 模型中流入和流出，如图 4.4 所示。

要实现这些步骤，我们使用第 2 章中的 tiktoken 分词器对包含两个文本输入的批次进行分词，以供 GPT 模型使用

```
    导入     tiktoken

    分词器 = tiktoken.get_encoding("gpt2") 批次 =
[] txt1 = "每一点努力都让你" txt2 = "每一天都蕴含
着"



    batch.append(torch.tensor(tokenizer.encode(txt1)))
batch.append(torch.tensor(tokenizer.encode(txt2))) batch =
torch.stack(batch, dim=0)
    打印(batch)
```

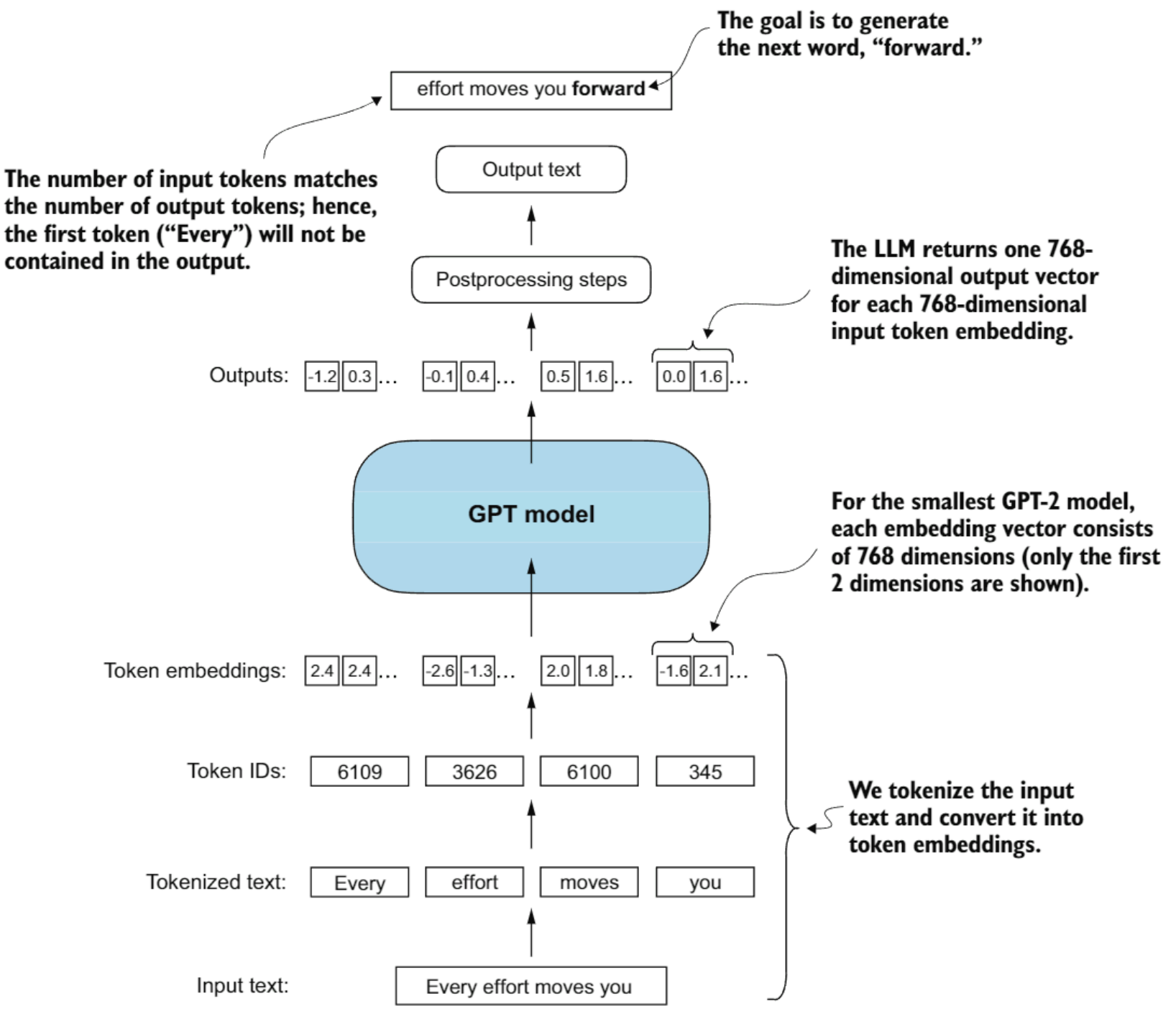

Figure 4.4 A big-picture overview showing how the input data is tokenized, embedded, and fed to the GPT model. Note that in our `DummyGPTClass` coded earlier, the token embedding is handled inside the GPT model. In LLMs, the embedded input token dimension typically matches the output dimension. The output embeddings here represent the context vectors (see chapter 3).

The resulting token IDs for the two texts are as follows:

```
tensor([[6109,  3626,  6100,   345],
        [6109,  1110,  6622,   257]])
```

The first row corresponds to the first text, and the second row corresponds to the second text.

Next, we initialize a new 124-million-parameter `DummyGPTModel` instance and feed it the tokenized `batch`:

```
torch.manual_seed(123)
model = DummyGPTModel(GPT_CONFIG_124M)
logits = model(batch)
print("Output shape:", logits.shape)
print(logits)
```

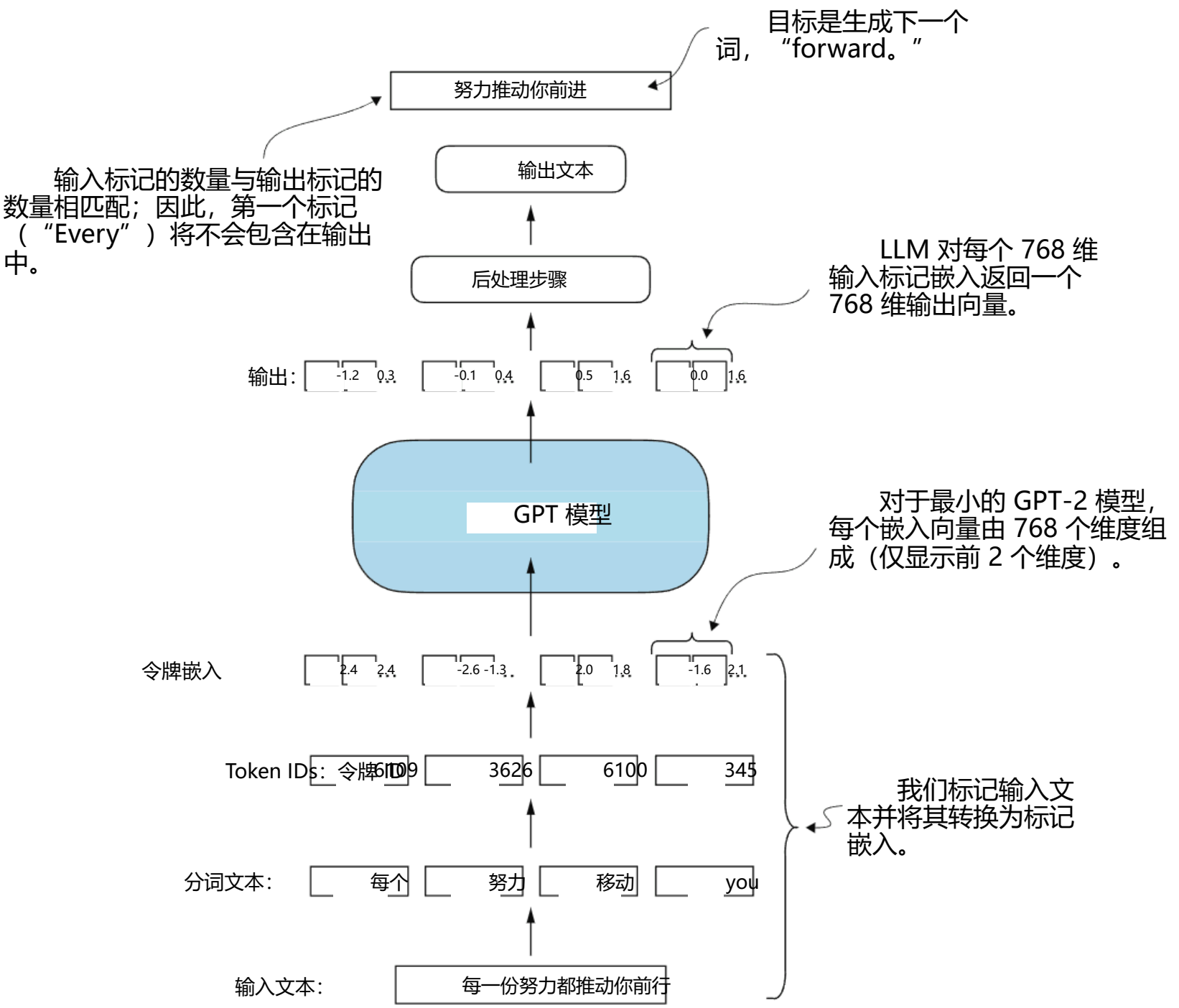

图 4.4 展示了如何对输入数据进行分词、嵌入并将其输入到 GPT 模型中的整体概述。注意，在我们之前编写的 DummyGPTClass 中，标记嵌入是在 GPT 模型内部处理的。在LLMs中，嵌入的输入标记维度通常与输出维度相匹配。这里的输出嵌入表示上下文向量（见第 3 章）。

两个文本的结果标记 ID 如下：

```
张量([[6109,  3626,  6100,  345]
      [6109, ] 1110,  6622,  257]])
```

第一行对应第一文本，第二行对应第二文本。

接下来，我们初始化一个包含 1240 万个参数的 DummyGPTModel 实例，并向其提供分词后的批次：

```
    torch 手动设置随机种子(123) 模型 =
DummyGPTModel(GPT_CONFIG_124M) logits = 模型
(批次) 打印("输出形状:", logits.shape) 打印
(logits)
```

The model outputs, which are commonly referred to as logits, are as follows:

```
Output shape: torch.Size([2, 4, 50257])
tensor([[[-1.2034,  0.3201, -0.7130,  ..., -1.5548, -0.2390, -0.4667],
         [-0.1192,  0.4539, -0.4432,  ...,  0.2392,  1.3469,  1.2430],
         [ 0.5307,  1.6720, -0.4695,  ...,  1.1966,  0.0111,  0.5835],
         [ 0.0139,  1.6755, -0.3388,  ...,  1.1586, -0.0435, -1.0400]],

        [[-1.0908,  0.1798, -0.9484,  ..., -1.6047,  0.2439, -0.4530],
         [-0.7860,  0.5581, -0.0610,  ...,  0.4835, -0.0077,  1.6621],
         [ 0.3567,  1.2698, -0.6398,  ..., -0.0162, -0.1296,  0.3717],
         [-0.2407, -0.7349, -0.5102,  ...,  2.0057, -0.3694,  0.1814]]],
       grad_fn=<UnsafeViewBackward0>)
```

The output tensor has two rows corresponding to the two text samples. Each text sample consists of four tokens; each token is a 50,257-dimensional vector, which matches the size of the tokenizer's vocabulary.

The embedding has 50,257 dimensions because each of these dimensions refers to a unique token in the vocabulary. When we implement the postprocessing code, we will convert these 50,257-dimensional vectors back into token IDs, which we can then decode into words.

Now that we have taken a top-down look at the GPT architecture and its inputs and outputs, we will code the individual placeholders, starting with the real layer normalization class that will replace the `DummyLayerNorm` in the previous code.

4.2 Normalizing activations with layer normalization

Training deep neural networks with many layers can sometimes prove challenging due to problems like vanishing or exploding gradients. These problems lead to unstable training dynamics and make it difficult for the network to effectively adjust its weights, which means the learning process struggles to find a set of parameters (weights) for the neural network that minimizes the loss function. In other words, the network has difficulty learning the underlying patterns in the data to a degree that would allow it to make accurate predictions or decisions.

> **NOTE** If you are new to neural network training and the concepts of gradients, a brief introduction to these concepts can be found in section A.4 in appendix A. However, a deep mathematical understanding of gradients is not required to follow the contents of this book.

Let's now implement *layer normalization* to improve the stability and efficiency of neural network training. The main idea behind layer normalization is to adjust the activations (outputs) of a neural network layer to have a mean of 0 and a variance of 1, also known as unit variance. This adjustment speeds up the convergence to effective weights and ensures consistent, reliable training. In GPT-2 and modern transformer architectures, layer normalization is typically applied before and after the multi-head attention module, and, as we have seen with the `DummyLayerNorm` placeholder, before

模型输出的结果，通常被称为 logits，如下所示：

```
输出形状：torch.Size([2, 4, 50257]) 张量
[[[-1.2034, 0.3201, -0.7130,]]                    ...,  -1.5548,   [-0.2390, -0.4667]
              [-0.1192,]   0.4539,   -0.4432,     ...,   0.2392,    1.3469,    1.2430]
              [ 0.5307, ]  1.6720,   -0.4695,     ...,   1.1966,    0.0111,    0.5835]
              [ 0.0139 ]   1.6755,   -0.3388,     ...,   1.1586,   -0.0435, -1.0400]]

             [[-1.0908, 0.1798, -0.9484, ..., -1.6047, 0.2439, -0.4530], [-0.7860,]
                            0.5581,   -0.0610,     ...,   0.4835,   -0.0077,    1.6621]
              [ 0.3567, ]  1.2698,   -0.6398,     ...,  -0.0162,   -0.1296,    0.3717]
            [-0.2407, -0.7349, -0.5102, ..., 2.0057, -0.3694, 0.1814]]], grad_fn=)
```

输出张量对应两个文本样本的两行。每个文本样本由四个标记组成；每个标记是一个 50,257 维度的向量，与分词器的词汇表大小相匹配。

嵌入有 50,257 个维度，因为每个维度都对应词汇表中的一个唯一标记。当我们实现后处理代码时，我们将这些 50,257 维向量转换回标记 ID，然后我们可以将其解码成单词。

现在我们已经从上到下了解了 GPT 架构及其输入输出，我们将编写单个占位符代码，从替换之前代码中的 DummyLayerNorm 的真实层归一化类开始。

4.2 将激活值进行层归一化

训练多层深度神经网络有时会因梯度消失或梯度爆炸等问题而变得具有挑战性。这些问题导致训练动态不稳定，使得网络难以有效地调整其权重，这意味着学习过程难以找到一组参数（权重）以最小化损失函数。换句话说，网络难以达到足够的学习程度，以从数据中学习到允许其做出准确预测或决策的潜在模式。

> 注意：如果您对神经网络训练和梯度概念不熟悉，可以在附录 A 的 A.4 节中找到这些概念的简要介绍。然而，理解梯度不需要深入数学知识，以理解本书的内容。

现在让我们实现层归一化来提高神经网络训练的稳定性和效率。层归一化的主要思想是将神经网络层的激活（输出）调整为均值为 0、方差为 1，也称为单位方差。这种调整加快了有效权重的收敛速度，并确保了一致、可靠的训练。在 GPT-2 和现代 Transformer 架构中，层归一化通常在多头注意力模块前后应用，并且，正如我们通过 DummyLayerNorm 占位符所看到的，在

the final output layer. Figure 4.5 provides a visual overview of how layer normalization functions.

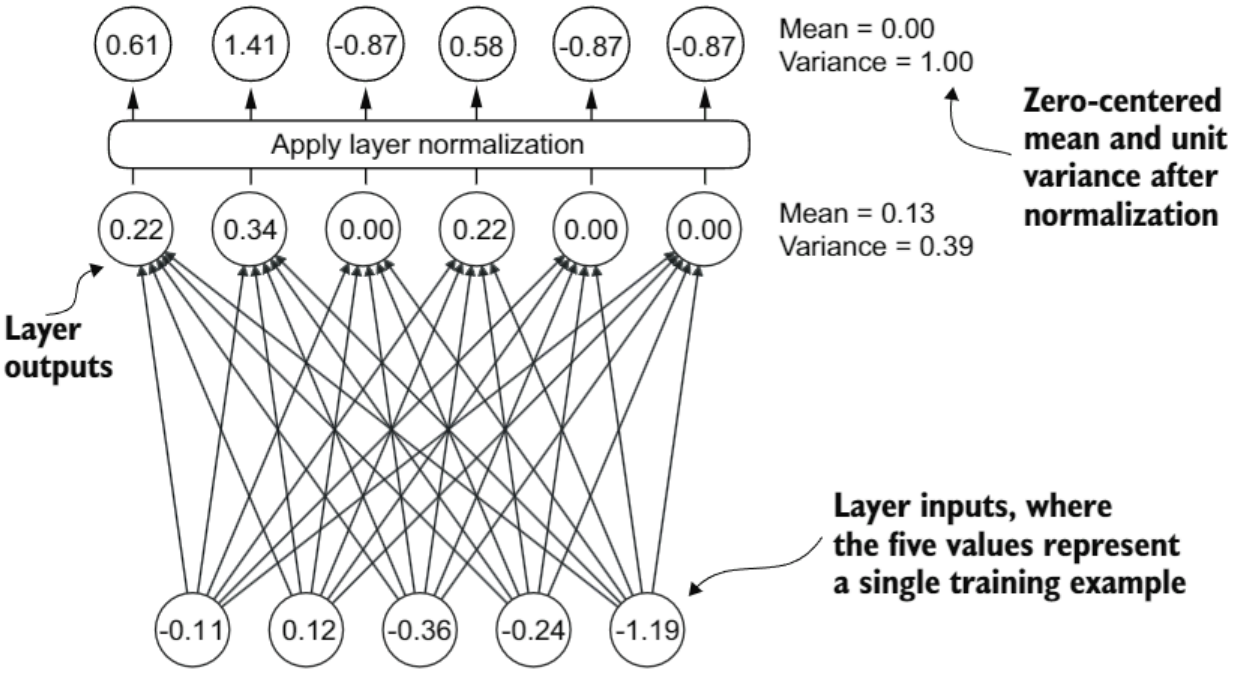

Figure 4.5 An illustration of layer normalization where the six outputs of the layer, also called activations, are normalized such that they have a 0 mean and a variance of 1.

We can recreate the example shown in figure 4.5 via the following code, where we implement a neural network layer with five inputs and six outputs that we apply to two input examples:

```
torch.manual_seed(123)
batch_example = torch.randn(2, 5)
layer = nn.Sequential(nn.Linear(5, 6), nn.ReLU())
out = layer(batch_example)
print(out)
```

Creates two training examples with five dimensions (features) each

This prints the following tensor, where the first row lists the layer outputs for the first input and the second row lists the layer outputs for the second row:

```
tensor([[0.2260, 0.3470, 0.0000, 0.2216, 0.0000, 0.0000],
        [0.2133, 0.2394, 0.0000, 0.5198, 0.3297, 0.0000]],
       grad_fn=<ReluBackward0>)
```

The neural network layer we have coded consists of a `Linear` layer followed by a nonlinear activation function, `ReLU` (short for rectified linear unit), which is a standard activation function in neural networks. If you are unfamiliar with `ReLU`, it simply thresholds negative inputs to 0, ensuring that a layer outputs only positive values, which explains why the resulting layer output does not contain any negative values. Later, we will use another, more sophisticated activation function in GPT.

在最终输出层之前。图 4.5 提供了层归一化功能的一个视觉概述。

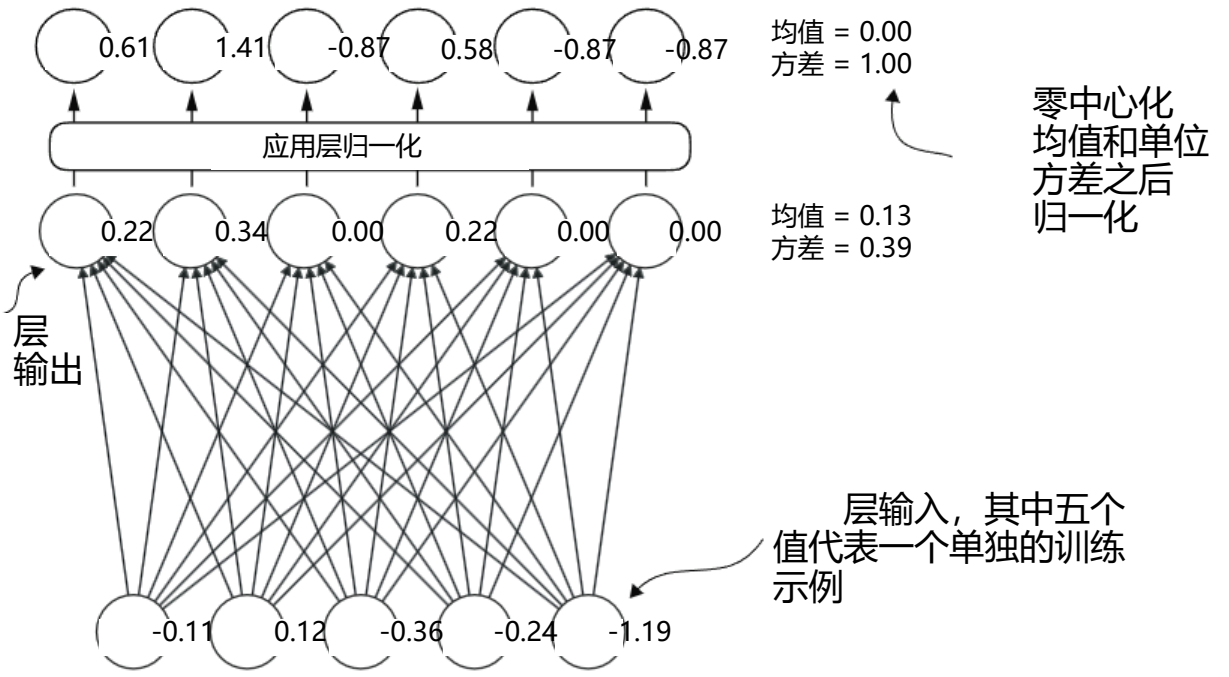

图 4.5 层归一化的示意图，其中层的六个输出，也称为激活，被归一化，使得它们具有 0 均值和 1 方差。

我们可以通过以下代码重新创建图 4.5 所示的示例，其中我们实现了一个具有五个输入和六个输出的神经网络层，并将其应用于两个输入示例：

```
torch.manual_seed(123) 批量示例 = torch.randn(2, 5) 层 = nn.Sequential(nn.Linear(5, 6), nn.ReLU()) 输出 = 层(批量示例) 打印(输出)
```

创建两个具有五个维度（特征）的培训示例

这打印了以下张量，其中第一行列出了第一个输入的层输出，第二行列出了第二行的层输出：

```
张量([[0.2260, 0.3470,         0.0000, 0.2216, 0.0000,       0.0000],
        [0.2133, 0.2394, 0.0000, 0.5198, 0.3297, 0.0000]], 梯度函数=)
```

我们编写的神经网络层由一个线性层后跟一个非线性激活函数 ReLU（即修正线性单元）组成，ReLU 是神经网络中的标准激活函数。如果您不熟悉 ReLU，它只是将负输入阈值设置为 0，确保层只输出正值，这也解释了为什么结果层的输出不包含任何负值。

稍后，我们将在 GPT 中使用另一个更复杂的激活函数。

Before we apply layer normalization to these outputs, let's examine the mean and variance:

```
mean = out.mean(dim=-1, keepdim=True)
var = out.var(dim=-1, keepdim=True)
print("Mean:\n", mean)
print("Variance:\n", var)
```

The output is

```
Mean:
  tensor([[0.1324],
          [0.2170]], grad_fn=<MeanBackward1>)
Variance:
  tensor([[0.0231],
          [0.0398]], grad_fn=<VarBackward0>)
```

The first row in the mean tensor here contains the mean value for the first input row, and the second output row contains the mean for the second input row.

Using `keepdim=True` in operations like mean or variance calculation ensures that the output tensor retains the same number of dimensions as the input tensor, even though the operation reduces the tensor along the dimension specified via `dim`. For instance, without `keepdim=True`, the returned mean tensor would be a two-dimensional vector `[0.1324, 0.2170]` instead of a 2 × 1–dimensional matrix `[[0.1324], [0.2170]]`.

The `dim` parameter specifies the dimension along which the calculation of the statistic (here, mean or variance) should be performed in a tensor. As figure 4.6 explains, for

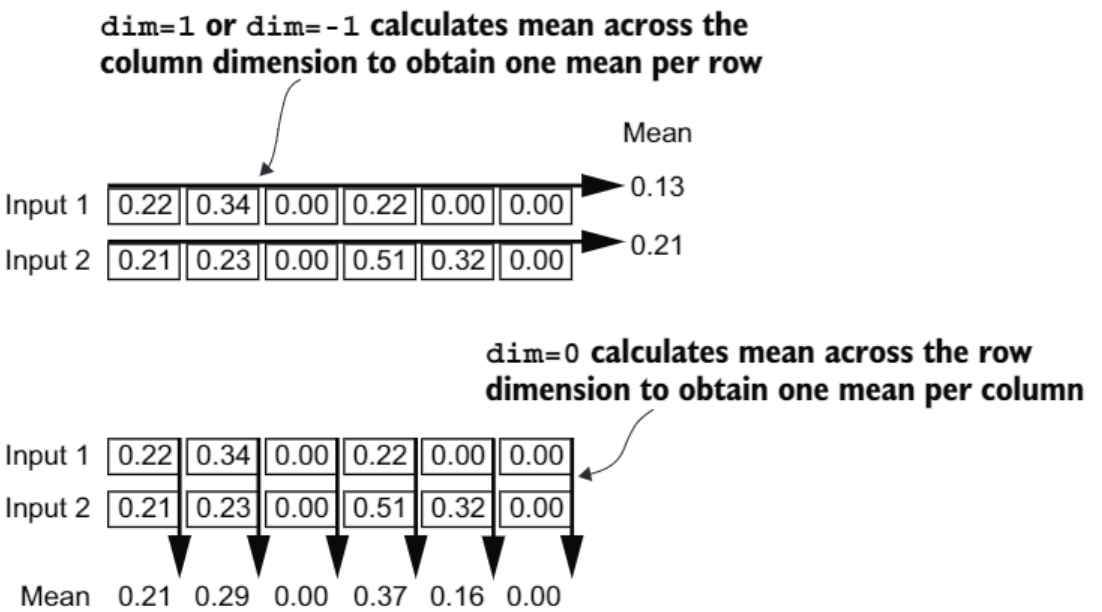

Figure 4.6 An illustration of the dim parameter when calculating the mean of a tensor. For instance, if we have a two-dimensional tensor (matrix) with dimensions `[rows, columns]`, using `dim=0` will perform the operation across rows (vertically, as shown at the bottom), resulting in an output that aggregates the data for each column. Using `dim=1` or `dim=-1` will perform the operation across columns (horizontally, as shown at the top), resulting in an output aggregating the data for each row.

在我们应用层归一化到这些输出之前，让我们先检查均值和方差：

```
    mean = out.mean(维度=-1, 保持维度=True)
var = out.var(维度=-1, 保持维度=True) 打印
("平均值:\n", mean) 打印("方差:\n", var)
```

输出结果

```
平均值：
  张量([[0.1324],
          [0.2170]]    grad_fn=<MeanBackward1>)
方差：
  张量([[0.0231],
          [0.0398]]    grad_fn=<VarBackward0>)
```

该张量第一行包含第一个输入行的平均值，第二输出行包含第二个输入行的平均值。

使用 keepdim=True 在计算均值或方差等操作中确保输出张量保留与输入张量相同的维度数，即使操作通过 dim 指定的维度减少了张量。例如，如果不使用 keepdim=True，返回的均值张量将是一个二维向量

```
[0.1324, 0.2170]
```

而不是一个 2×1 维矩阵 [[0.1324], [0.2170]].

张量中统计量（此处为均值或方差）的计算应沿哪个维度进行，由 dim 参数指定。如图 4.6 所示，对于

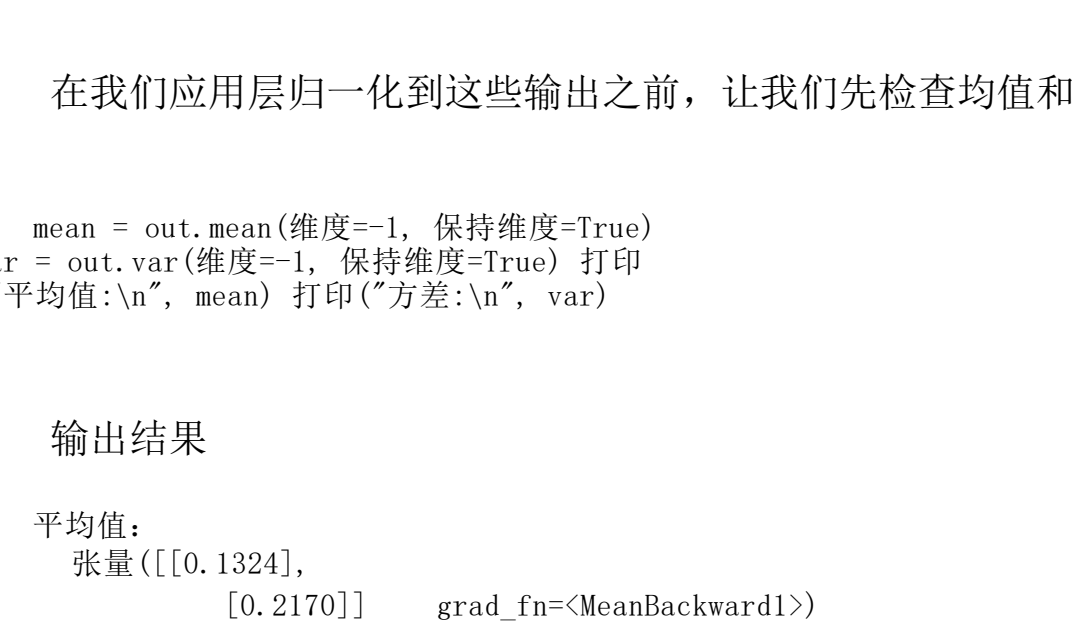

图 4.6 计算张量均值时 dim 参数的说明。例如，如果我们有一个二维张量（矩阵）具有维度[行，列]，使用 dim=0 将在行（垂直，如图底部所示）上执行操作，结果将汇总每列的数据。使用 dim=1 或 dim=-1 将在列（水平，如图顶部所示）上执行操作，结果将汇总每行的数据。

a two-dimensional tensor (like a matrix), using `dim=-1` for operations such as mean or variance calculation is the same as using `dim=1`. This is because `-1` refers to the tensor's last dimension, which corresponds to the columns in a two-dimensional tensor. Later, when adding layer normalization to the GPT model, which produces three-dimensional tensors with the shape `[batch_size, num_tokens, embedding_size]`, we can still use `dim=-1` for normalization across the last dimension, avoiding a change from `dim=1` to `dim=2`.

Next, let's apply layer normalization to the layer outputs we obtained earlier. The operation consists of subtracting the mean and dividing by the square root of the variance (also known as the standard deviation):

```
out_norm = (out - mean) / torch.sqrt(var)
mean = out_norm.mean(dim=-1, keepdim=True)
var = out_norm.var(dim=-1, keepdim=True)
print("Normalized layer outputs:\n", out_norm)
print("Mean:\n", mean)
print("Variance:\n", var)
```

As we can see based on the results, the normalized layer outputs, which now also contain negative values, have 0 mean and a variance of 1:

```
Normalized layer outputs:
 tensor([[ 0.6159,  1.4126, -0.8719,  0.5872, -0.8719, -0.8719],
        [-0.0189,  0.1121, -1.0876,  1.5173,  0.5647, -1.0876]],
       grad_fn=<DivBackward0>)
Mean:
 tensor([[-5.9605e-08],
        [1.9868e-08]], grad_fn=<MeanBackward1>)
Variance:
 tensor([[1.],
        [1.]], grad_fn=<VarBackward0>)
```

Note that the value –5.9605e-08 in the output tensor is the scientific notation for -5.9605×10^{-8}, which is –0.000000059605 in decimal form. This value is very close to 0, but it is not exactly 0 due to small numerical errors that can accumulate because of the finite precision with which computers represent numbers.

To improve readability, we can also turn off the scientific notation when printing tensor values by setting `sci_mode` to `False`:

```
torch.set_printoptions(sci_mode=False)
print("Mean:\n", mean)
print("Variance:\n", var)
```

The output is

```
Mean:
 tensor([[    0.0000],
        [    0.0000]], grad_fn=<MeanBackward1>)
```

一个二维张量（如矩阵），使用 dim=-1 进行均值或方差计算与使用 dim=1 相同。这是因为-1 指的是张量的最后一个维度，在二维张量中对应于列。后来，当将层归一化添加到 GPT 模型时，该模型产生形状为[batch_size, num_tokens, embedding_size]的三维张量，我们仍然可以使用 dim=-1 对最后一个维度进行归一化，避免从 dim=1 到

dim=2

接下来，让我们将层归一化应用于我们之前获得的层输出。该操作包括减去均值并除以方差的平方根（也称为标准差）：

```
out_norm = (out - mean) / torch.sqrt(var) mean =
out_norm.mean(dim=-1, keepdim=True) var =
out_norm.var(dim=-1, keepdim=True) 打印("标准化层输出:
\n", out_norm) 打印("均值: \n", mean) 打印("方差: \n",
var)
```

根据结果我们可以看到，归一化层输出，现在也包含负值，均值为 0，方差为 1：

```
标准化层输出：tensor([[
0.6159,                    1.4126,   -0.8719,      [0.5872, -0.8719, -0.8719]
        [-0.0189, 0.1121, -1.0876, 1.5173, 0.5647, -1.0876], grad_fn=)
均值：tensor([[-5.9605e-
08],
          [1.9868e-08]      grad_fn=<MeanBackward1>)
方差：
 tensor([[1.]])
          [1.]], grad_fn=<VarBackward0>)
```

注意，输出张量中的值-5.9605e-08 是-5.9605×10 的科学记数法，以十进制形式表示为-0.000000059605。这个值非常接近 0，但由于计算机以有限的精度表示数字，可能会积累小的数值误差，因此它不是正好等于 0。

为了提高可读性，我们还可以在打印张量值时关闭科学记数法，通过将 sci_mode 设置为 False：

```
torch 设置打印选项（科学计数法模式为
False）打印("平均值: \n", mean）打印("方差:
\n", var）
```

输出结果

```
平均值：
 张量([[         0.0000],
          [     0.0000]]    grad_fn=<MeanBackward1>)
```

```
Variance:
 tensor([[1.],
         [1.]], grad_fn=<VarBackward0>)
```

So far, we have coded and applied layer normalization in a step-by-step process. Let's now encapsulate this process in a PyTorch module that we can use in the GPT model later.

Listing 4.2 A layer normalization class

```
class LayerNorm(nn.Module):
    def __init__(self, emb_dim):
        super().__init__()
        self.eps = 1e-5
        self.scale = nn.Parameter(torch.ones(emb_dim))
        self.shift = nn.Parameter(torch.zeros(emb_dim))

    def forward(self, x):
        mean = x.mean(dim=-1, keepdim=True)
        var = x.var(dim=-1, keepdim=True, unbiased=False)
        norm_x = (x - mean) / torch.sqrt(var + self.eps)
        return self.scale * norm_x + self.shift
```

This specific implementation of layer normalization operates on the last dimension of the input tensor x, which represents the embedding dimension (`emb_dim`). The variable `eps` is a small constant (epsilon) added to the variance to prevent division by zero during normalization. The `scale` and `shift` are two trainable parameters (of the same dimension as the input) that the LLM automatically adjusts during training if it is determined that doing so would improve the model's performance on its training task. This allows the model to learn appropriate scaling and shifting that best suit the data it is processing.

Biased variance

In our variance calculation method, we use an implementation detail by setting `unbiased=False`. For those curious about what this means, in the variance calculation, we divide by the number of inputs *n* in the variance formula. This approach does not apply Bessel's correction, which typically uses *n – 1* instead of *n* in the denominator to adjust for bias in sample variance estimation. This decision results in a so-called biased estimate of the variance. For LLMs, where the embedding dimension *n* is significantly large, the difference between using *n* and *n – 1* is practically negligible. I chose this approach to ensure compatibility with the GPT-2 model's normalization layers and because it reflects TensorFlow's default behavior, which was used to implement the original GPT-2 model. Using a similar setting ensures our method is compatible with the pretrained weights we will load in chapter 6.

Let's now try the `LayerNorm` module in practice and apply it to the batch input:

```
方差：
 tensor([[1.]])
         [1.]], grad_fn=<VarBackward0>)
```

到目前为止，我们已经逐步编码并应用了层归一化。现在让我们将这个过程封装成一个 PyTorch 模块，以便以后在 GPT 模型中使用。

列表 4.2 层归一化类

```
class 层归一化(nn.Module):
     def __init__(self, emb_dim): # 初始化(self, 词嵌入维度)
          super().__init__()
          self.eps = 1e-5
          self.scale = nn.Parameter(torch.ones(emb_dim))
      self.shift =        nn.Parameter(torch.zeros(emb_dim))

     def forward(self, x): # 定义前向传播函数
          mean = x.mean(维度=-1, 保持维度=True) var = x.var(维度
      =-1, 保持维度=True, 无偏=False) norm_x = (x - mean) /
      torch.sqrt(var + self.eps) return self.scale * norm_x +
      self.shift
```

这一特定实现的层归一化操作在输入张量 x 的最后一个维度上，该维度代表嵌入维度（emb_dim）。变量 eps 是一个小的常数（epsilon），在归一化过程中添加到方差中，以防止除以零。缩放和偏移是两个可训练的参数（与输入具有相同的维度），在训练过程中，如果确定这样做会提高模型在训练任务上的性能，LLM会自动调整这些参数。这允许模型学习适当的缩放和偏移，以最好地适应其处理的数据。

偏见方差

在我们的方差计算方法中，我们通过设置实现细节来使用

无偏=False。对于那些对此表示好奇的人，在方差计算中，我们在方差公式中除以输入数量 n。这种方法不应用贝塞尔校正，通常在分母中使用 n-1 而不是 n 来调整样本方差估计中的偏差。这个决定导致了一个所谓的有偏方差估计。对于LLMs，其中嵌入维度 n 非常大，使用 n 和 n-1 之间的差异实际上可以忽略不计。我选择这种方法是为了确保与 GPT-2 模型的归一化层兼容，并且因为它反映了 TensorFlow 的默认行为，这是实现原始 GPT-2 模型所使用的。使用类似的设置确保我们的方法与第 6 章中我们将加载的预训练权重兼容。

现在让我们在实践中使用 LayerNorm 模块并将其应用于批量输入：

```
ln = LayerNorm(emb_dim=5)
out_ln = ln(batch_example)
mean = out_ln.mean(dim=-1, keepdim=True)
var = out_ln.var(dim=-1, unbiased=False, keepdim=True)
print("Mean:\n", mean)
print("Variance:\n", var)
```

The results show that the layer normalization code works as expected and normalizes the values of each of the two inputs such that they have a mean of 0 and a variance of 1:

```
Mean:
 tensor([[    -0.0000],
         [     0.0000]], grad_fn=<MeanBackward1>)
Variance:
 tensor([[1.0000],
         [1.0000]], grad_fn=<VarBackward0>)
```

We have now covered two of the building blocks we will need to implement the GPT architecture, as shown in figure 4.7. Next, we will look at the GELU activation function, which is one of the activation functions used in LLMs, instead of the traditional ReLU function we used previously.

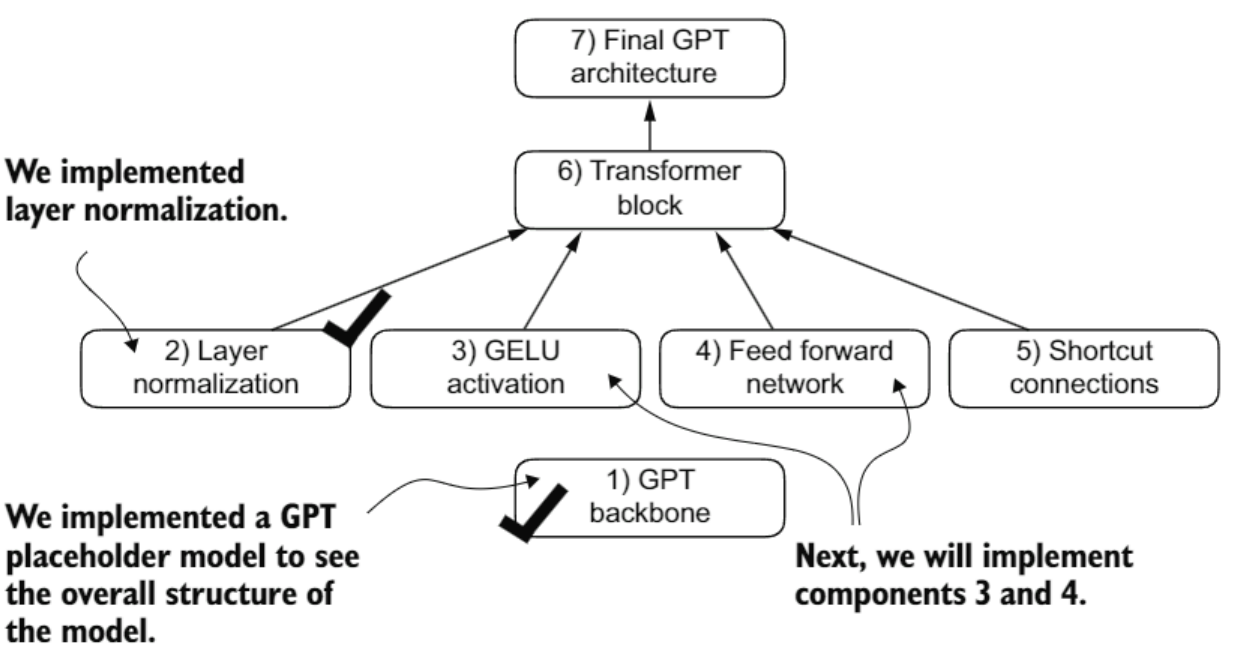

Figure 4.7 The building blocks necessary to build the GPT architecture. So far, we have completed the GPT backbone and layer normalization. Next, we will focus on GELU activation and the feed forward network.

Layer normalization vs. batch normalization

If you are familiar with batch normalization, a common and traditional normalization method for neural networks, you may wonder how it compares to layer normalization. Unlike batch normalization, which normalizes across the batch dimension, layer normalization normalizes across the feature dimension. LLMs often require significant

```
    ln = 层归一化(emb_dim=5) out_ln = ln(批量示例) mean =
out_ln.mean(dim=-1, keepdim=True) var = out_ln.var(dim=-1,
unbiased=False, keepdim=True) 打印("均值:\n", mean) 打印("方
差:\n", var)
```

结果显示，层归一化代码按预期工作，将两个输入的值归一化，使得它们的均值为 0，方差为 1：

```
平均值：
 张量([[          -0.0000]
        [          [0.0000], grad_fn=>
方差：
 张量([[1.0000],
        [1.0000], grad_fn=>
```

我们现在已经涵盖了实现 GPT 架构所需的基础块中的两个，如图 4.7 所示。接下来，我们将探讨 GELU 激活函数，这是在 LLMs 中使用的一种激活函数，而不是我们之前使用的传统 ReLU 函数。

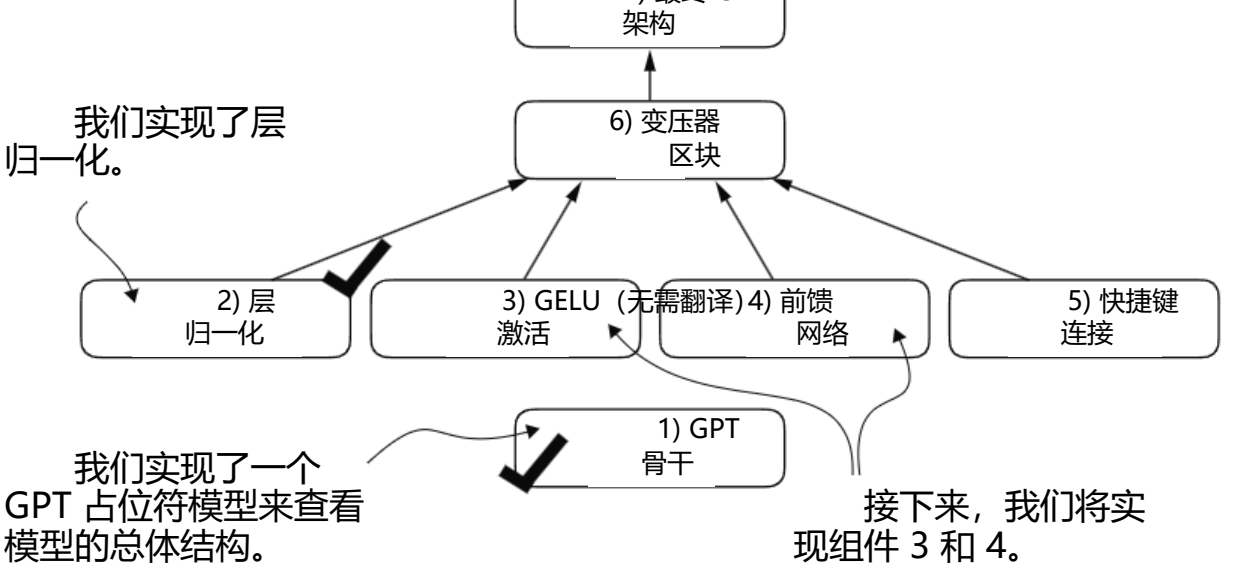

图 4.7 构建 GPT 架构所需的基石。迄今为止，我们已经完成了 GPT 骨干和层归一化。接下来，我们将专注于 GELU 激活和前馈网络。

层归一化与批归一化

如果您熟悉批归一化，这是一种常见的传统神经网络归一化方法，您可能会想知道它与层归一化相比如何。与批归一化不同，批归一化是在批量维度上进行归一化，而层归一化是在特征维度上进行归一化。LLMs通常需要显著

computational resources, and the available hardware or the specific use case can dictate the batch size during training or inference. Since layer normalization normalizes each input independently of the batch size, it offers more flexibility and stability in these scenarios. This is particularly beneficial for distributed training or when deploying models in environments where resources are constrained.

4.3 Implementing a feed forward network with GELU activations

Next, we will implement a small neural network submodule used as part of the transformer block in LLMs. We begin by implementing the *GELU* activation function, which plays a crucial role in this neural network submodule.

NOTE For additional information on implementing neural networks in PyTorch, see section A.5 in appendix A.

Historically, the ReLU activation function has been commonly used in deep learning due to its simplicity and effectiveness across various neural network architectures. However, in LLMs, several other activation functions are employed beyond the traditional ReLU. Two notable examples are GELU (*Gaussian error linear unit*) and SwiGLU (*Swish-gated linear unit*).

GELU and SwiGLU are more complex and smooth activation functions incorporating Gaussian and sigmoid-gated linear units, respectively. They offer improved performance for deep learning models, unlike the simpler ReLU.

The GELU activation function can be implemented in several ways; the exact version is defined as GELU(x) = x·Φ(x), where Φ(x) is the cumulative distribution function of the standard Gaussian distribution. In practice, however, it's common to implement a computationally cheaper approximation (the original GPT-2 model was also trained with this approximation, which was found via curve fitting):

$$GELU(x) \approx 0.5 \cdot x \cdot \left(1 + tanh\left[\sqrt{\frac{2}{\pi}} \cdot \left(x + 0.044715 \cdot x^3\right)\right]\right)$$

In code, we can implement this function as a PyTorch module.

Listing 4.3 An implementation of the GELU activation function

```
class GELU(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        return 0.5 * x * (1 + torch.tanh(
            torch.sqrt(torch.tensor(2.0 / torch.pi)) *
            (x + 0.044715 * torch.pow(x, 3))
        ))
```

计算资源以及可用的硬件或特定用例可以决定训练或推理过程中的批量大小。由于层归一化独立于批量大小对每个输入进行归一化，因此在这些场景中提供了更大的灵活性和稳定性。这对于分布式训练或在资源受限的环境中部署模型特别有益。

4.3 实现具有 GELU 激活函数的前馈网络

接下来，我们将实现一个小型神经网络子模块，该模块作为LLMs中变换器块的一部分使用。我们首先实现 GELU 激活函数，它在神经网络子模块中起着至关重要的作用。

注意：有关在 PyTorch 中实现神经网络的更多信息，请参阅附录 A 中的 A.5 节。

历史上，ReLU 激活函数因其简单性和在各种神经网络架构中的有效性而被广泛用于深度学习。然而，在LLMs中，除了传统的 ReLU 之外，还使用了其他几种激活函数。两个值得注意的例子是 GELU（高斯误差线性单元）和 SwiGLU（Swish 门控线性单元）。

GELU 和 SwiGLU 是更复杂、更平滑的激活函数，分别结合高斯和 sigmoid 门控线性单元。与简单的 ReLU 不同，它们为深度学习模型提供了改进的性能。

GELU 激活函数可以通过多种方式实现；确切版本定义为 GELU(x) = x·Φ(x)，其中Φ(x)是标准高斯分布的累积分布函数。然而，在实践中，通常实现一个计算成本更低的近似（原始 GPT-2 模型也是用这个近似进行训练的，这个近似是通过曲线拟合得到的）：

$$GELU(x) \approx 0.5 \cdot x \cdot \left(1 + tanh\left[\sqrt{\frac{2}{\pi}} \cdot \left(x + 0.044715 \cdot x^3\right)\right]\right)$$

在代码中，我们可以将此函数实现为一个 PyTorch 模块。

列表 4.3 GELU 激活函数的实现

```
    class GELU(nn.Module):
类 GELU(nn.Module):
        def __init__(self):
            super().__init__()

        def forward(self, x): # 定义前向传播函数
            返回 0.5 * x * (1 + torch.tanh(
            torch.sqrt(torch.tensor(2.0 / torch.pi)) * (x +
         0.044715 * torch.pow(x, 3)))
```

Next, to get an idea of what this GELU function looks like and how it compares to the ReLU function, let's plot these functions side by side:

```
import matplotlib.pyplot as plt
gelu, relu = GELU(), nn.ReLU()

x = torch.linspace(-3, 3, 100)
y_gelu, y_relu = gelu(x), relu(x)
plt.figure(figsize=(8, 3))
for i, (y, label) in enumerate(zip([y_gelu, y_relu], ["GELU", "ReLU"]), 1):
    plt.subplot(1, 2, i)
    plt.plot(x, y)
    plt.title(f"{label} activation function")
    plt.xlabel("x")
    plt.ylabel(f"{label}(x)")
    plt.grid(True)
plt.tight_layout()
plt.show()
```

Creates 100 sample data points in the range –3 to 3

As we can see in the resulting plot in figure 4.8, ReLU (right) is a piecewise linear function that outputs the input directly if it is positive; otherwise, it outputs zero. GELU (left) is a smooth, nonlinear function that approximates ReLU but with a non-zero gradient for almost all negative values (except at approximately $x = -0.75$).

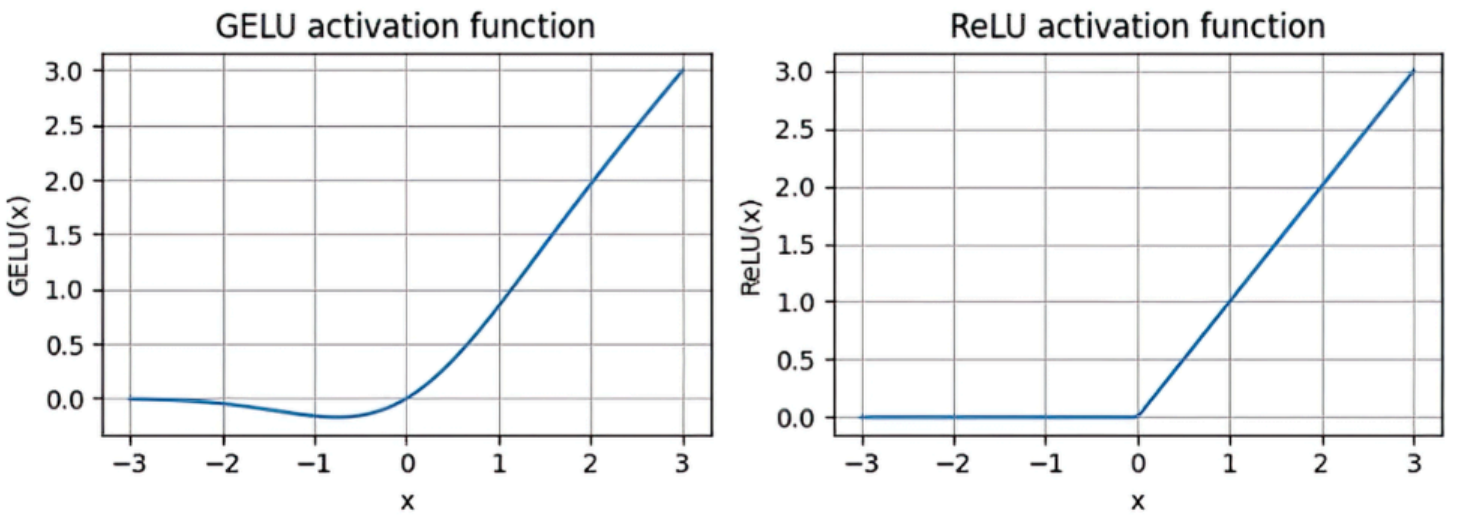

Figure 4.8 The output of the GELU and ReLU plots using matplotlib. The x-axis shows the function inputs and the y-axis shows the function outputs.

The smoothness of GELU can lead to better optimization properties during training, as it allows for more nuanced adjustments to the model's parameters. In contrast, ReLU has a sharp corner at zero (figure 4.18, right), which can sometimes make optimization harder, especially in networks that are very deep or have complex architectures. Moreover, unlike ReLU, which outputs zero for any negative input, GELU allows for a small, non-zero output for negative values. This characteristic means that during the training process, neurons that receive negative input can still contribute to the learning process, albeit to a lesser extent than positive inputs.

接下来，为了了解这个 GELU 函数的形状以及它与 ReLU 函数的比较，让我们将这些函数并排放置进行绘图：

```
    import matplotlib.pyplot as plt
gelu, relu = GELU(), nn.ReLU()


    x = torch.linspace(-3, 3, 100) y_gelu, y_relu = gelu(x), relu(x) plt.figure(figsize=(8,
3)) for i, (y, label) in enumerate(zip([y_gelu, y_relu], ["GELU", "ReLU"]), 1):



    plt.subplot(1, 2, i) plt.plot(x, y) plt.title(f"{标
签}激活函数") plt.xlabel("x") plt.ylabel(f"{标签}(x)")
plt.grid(True) plt.tight_layout() plt.show()
```

创建 100 个样本数据点在范围 -3 到 3

如图 4.8 所示的结果图中可见，ReLU（右侧）是一个分段线性函数，如果输入为正，则直接输出输入值；否则输出零。GELU（左侧）是一个平滑的非线性函数，它近似 ReLU，但几乎所有负值（除了大约 x = -0.75）都具有非零梯度。

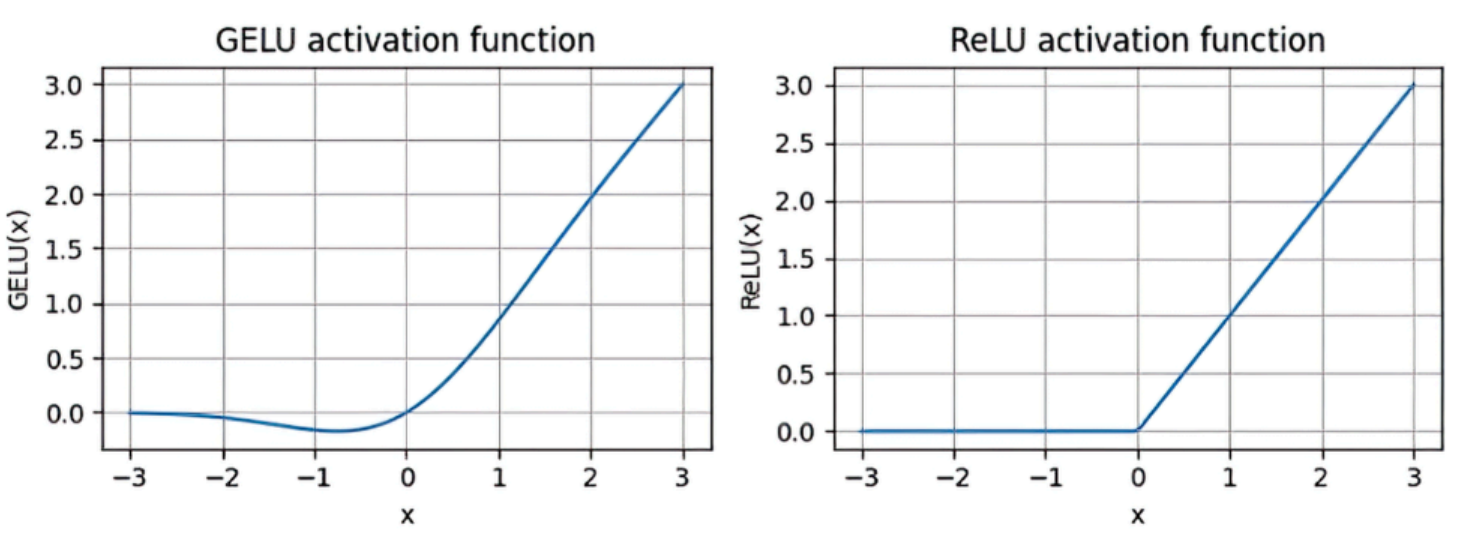

图 4.8 使用 matplotlib 绘制的 GELU 和 ReLU 图。x 轴显示函数输入，y 轴显示函数输出。

GELU 的平滑性可以在训练期间带来更好的优化特性，因为它允许对模型参数进行更细微的调整。相比之下，ReLU 在零点有一个尖锐的拐角（图 4.18，右），这有时会使优化更加困难，尤其是在非常深或具有复杂结构的网络中。此外，与 ReLU 不同，ReLU 对任何负输入都输出零，而 GELU 允许负值有小的、非零的输出。这一特性意味着在训练过程中，接收负输入的神经元仍然可以参与到学习过程中，尽管其贡献不如正输入大。

Next, let's use the GELU function to implement the small neural network module, `FeedForward`, that we will be using in the LLM's transformer block later.

Listing 4.4 A feed forward neural network module

```
class FeedForward(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(cfg["emb_dim"], 4 * cfg["emb_dim"]),
            GELU(),
            nn.Linear(4 * cfg["emb_dim"], cfg["emb_dim"]),
        )

    def forward(self, x):
        return self.layers(x)
```

As we can see, the `FeedForward` module is a small neural network consisting of two `Linear` layers and a `GELU` activation function. In the 124-million-parameter GPT model, it receives the input batches with tokens that have an embedding size of 768 each via the `GPT_CONFIG_124M` dictionary where `GPT_CONFIG_ 124M["emb_dim"] = 768`. Figure 4.9 shows how the embedding size is manipulated inside this small feed forward neural network when we pass it some inputs.

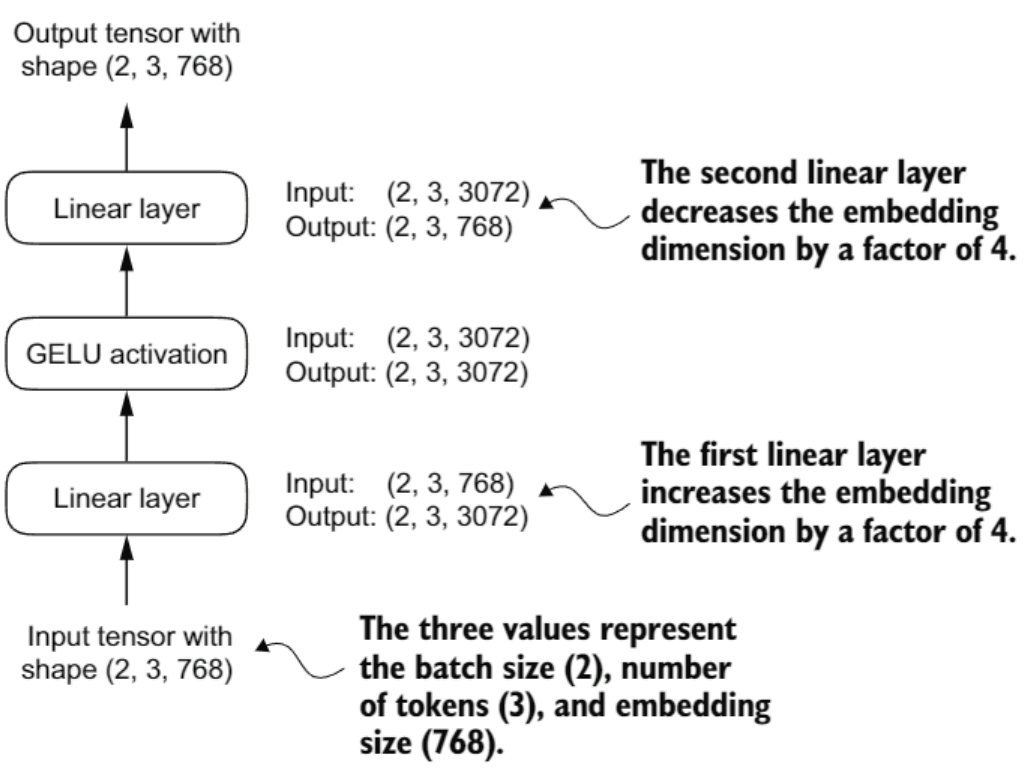

Figure 4.9 An overview of the connections between the layers of the feed forward neural network. This neural network can accommodate variable batch sizes and numbers of tokens in the input. However, the embedding size for each token is determined and fixed when initializing the weights.

接下来，让我们使用 GELU 函数来实现我们将在LLM的 transformer 块中使用的 FeedForward 小神经网络模块。

列表 4.4 前馈神经网络模块

```
class 前馈神经网络(nn.Module):
    def __init__(self, cfg):
        super().__init__() self.layers
 = nn.Sequential()
超类 __init__() self.layers =
nn.Sequential()
        nn.Linear(cfg["emb_dim"], 4 * cfg["emb_dim"]),
        GELU(), nn.Linear(4 * cfg["emb_dim"], cfg["emb_dim"]))




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

我们可以看到，FeedForward 模块是一个由两个线性层和一个 GELU 激活函数组成的小型神经网络。在 1240 万个参数的 GPT 模型中，它接收具有 768 个嵌入大小的标记输入批次。

每个通过 GPT_CONFIG_124M 字典，其中 GPT_CONFIG_ 124M["emb_dim"] = 768。

图 4.9 展示了当我们向这个小前馈神经网络传递一些输入时，嵌入大小是如何被操作的。

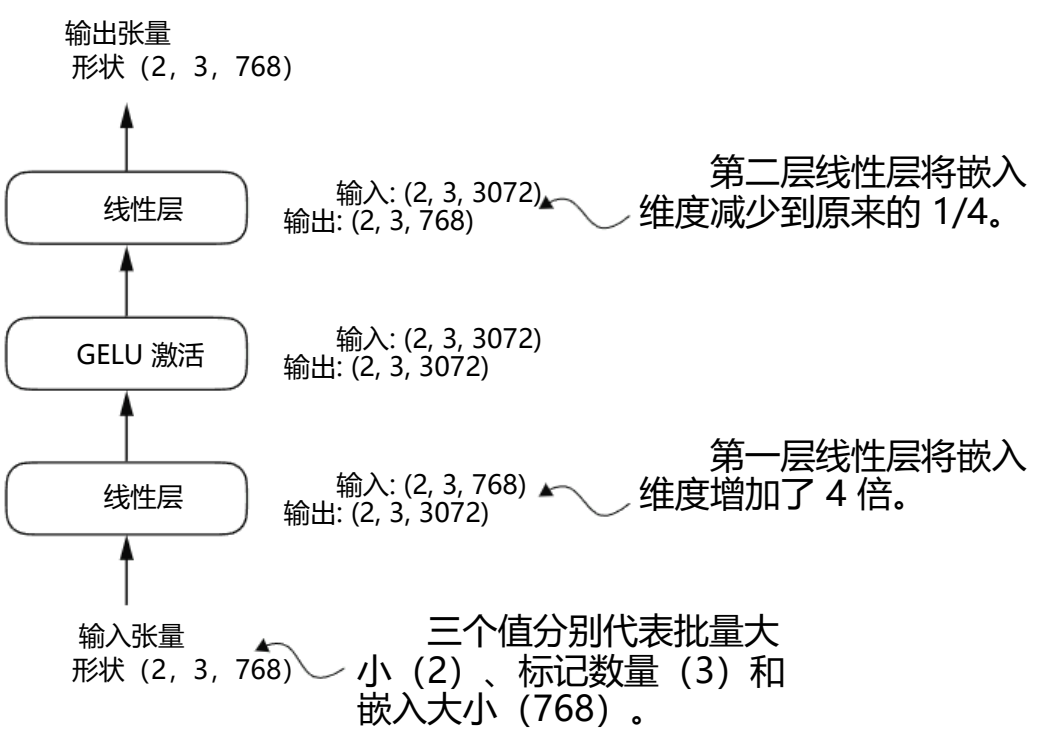

图 4.9 前馈神经网络各层之间连接的概述。该神经网络可以适应可变的批量大小和输入中的标记数量。然而，每个标记的嵌入大小在初始化权重时确定并固定。

Following the example in figure 4.9, let's initialize a new `FeedForward` module with a token embedding size of 768 and feed it a batch input with two samples and three tokens each:

```
ffn = FeedForward(GPT_CONFIG_124M)
x = torch.rand(2, 3, 768)
out = ffn(x)
print(out.shape)
```

Creates sample input with batch dimension 2

As we can see, the shape of the output tensor is the same as that of the input tensor:

```
torch.Size([2, 3, 768])
```

The `FeedForward` module plays a crucial role in enhancing the model's ability to learn from and generalize the data. Although the input and output dimensions of this module are the same, it internally expands the embedding dimension into a higher-dimensional space through the first linear layer, as illustrated in figure 4.10. This expansion is followed by a nonlinear GELU activation and then a contraction back to the original dimension with the second linear transformation. Such a design allows for the exploration of a richer representation space.

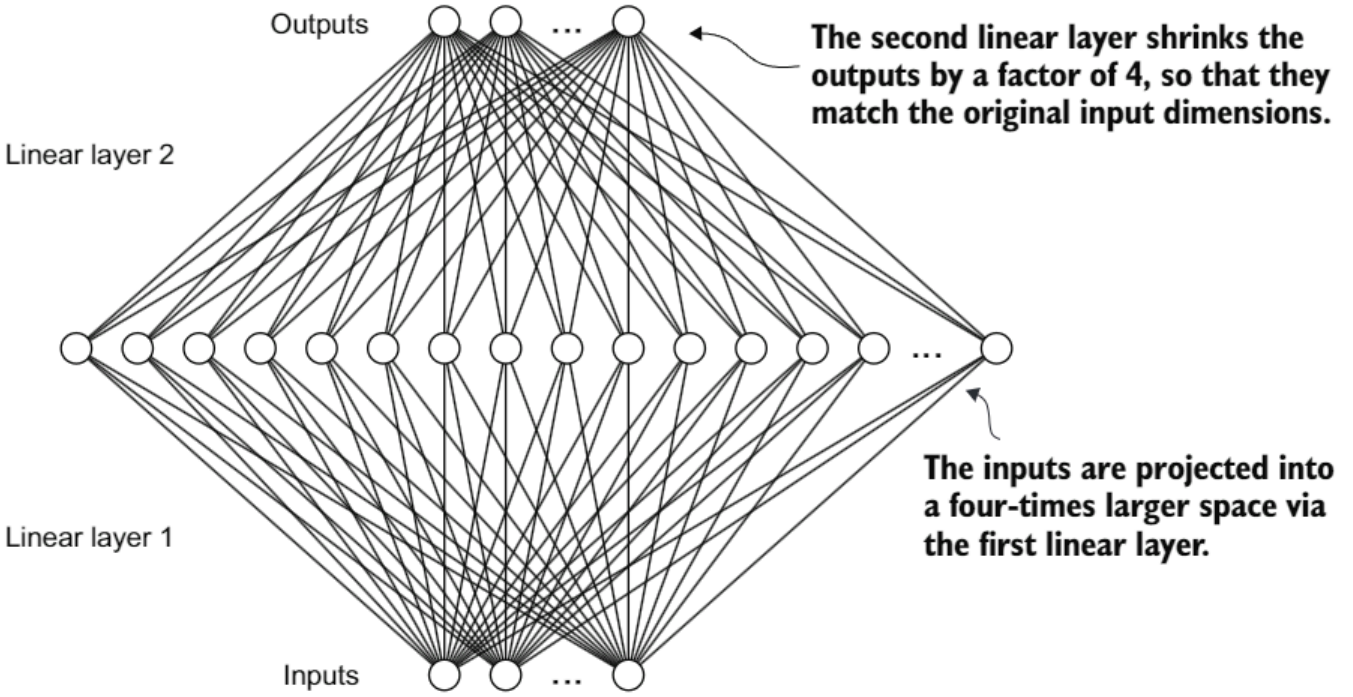

Figure 4.10 An illustration of the expansion and contraction of the layer outputs in the feed forward neural network. First, the inputs expand by a factor of 4 from 768 to 3,072 values. Then, the second layer compresses the 3,072 values back into a 768-dimensional representation.

Moreover, the uniformity in input and output dimensions simplifies the architecture by enabling the stacking of multiple layers, as we will do later, without the need to adjust dimensions between them, thus making the model more scalable.

按照图 4.9 中的示例，让我们初始化一个具有 768 个标记嵌入大小的新的前馈模块，并向其提供一个包含两个样本和每个样本三个标记的批次输入：

```
ffn = 前馈神经网络(GPT_CONFIG_124M) x
= torch.rand(2, 3, 768) out = ffn(x) 打印
(out.shape)
```

创建具有批量维度 2 的示例输入

我们可以看到，输出张量的形状与输入张量相同：

```
torch.Size([2, 3,])   768])
```

前馈模块在增强模型从数据中学习和泛化的能力中起着至关重要的作用。尽管该模块的输入和输出维度相同，但它通过第一层线性层将嵌入维度扩展到更高维空间，如图 4.10 所示。这种扩展随后通过非线性 GELU 激活，然后通过第二层线性变换收缩回原始维度。这种设计允许探索更丰富的表示空间。

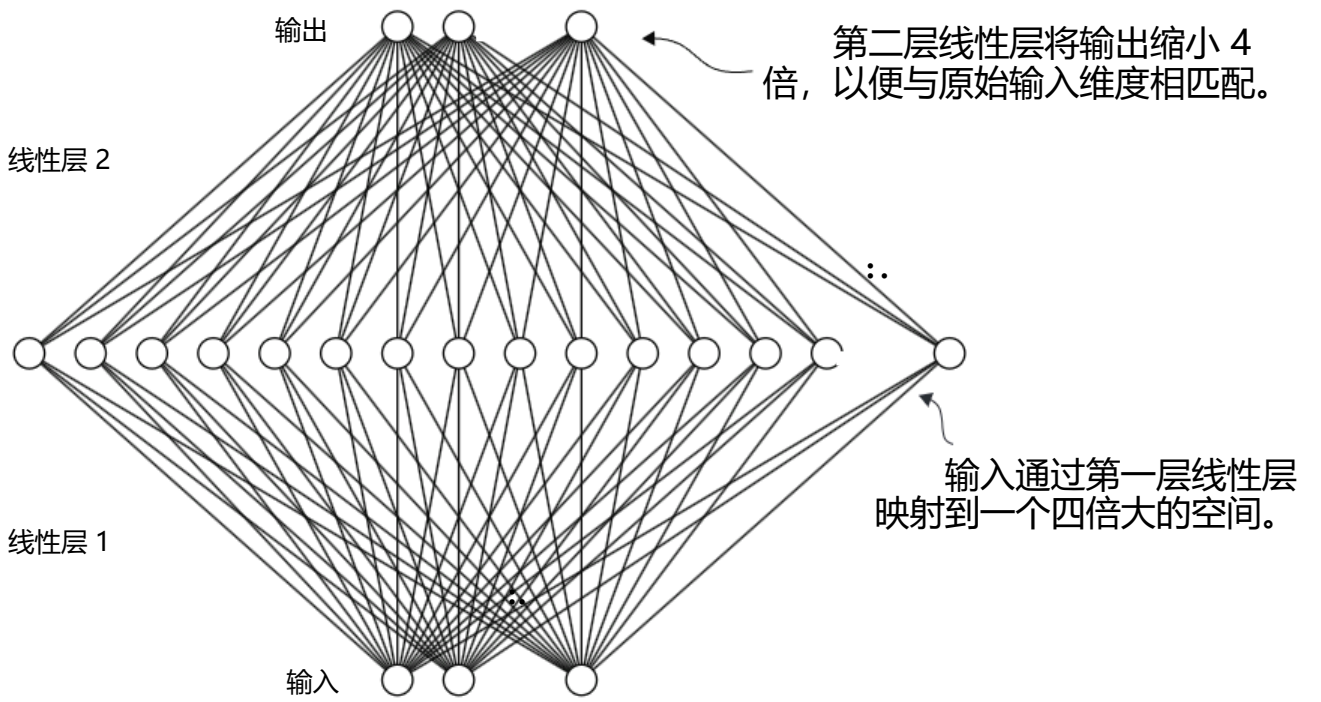

图 4.10 前馈神经网络中层输出的扩展和收缩示意图。首先，输入值以 4 倍扩展，从 768 增加到 3,072 个值。然后，第二层将 3,072 个值压缩回 768 维表示。

此外，输入和输出维度的统一通过允许堆叠多个层（正如我们稍后将要做的）简化了架构，无需调整它们之间的维度，从而使模型更具可扩展性。

As figure 4.11 shows, we have now implemented most of the LLM's building blocks. Next, we will go over the concept of shortcut connections that we insert between different layers of a neural network, which are important for improving the training performance in deep neural network architectures.

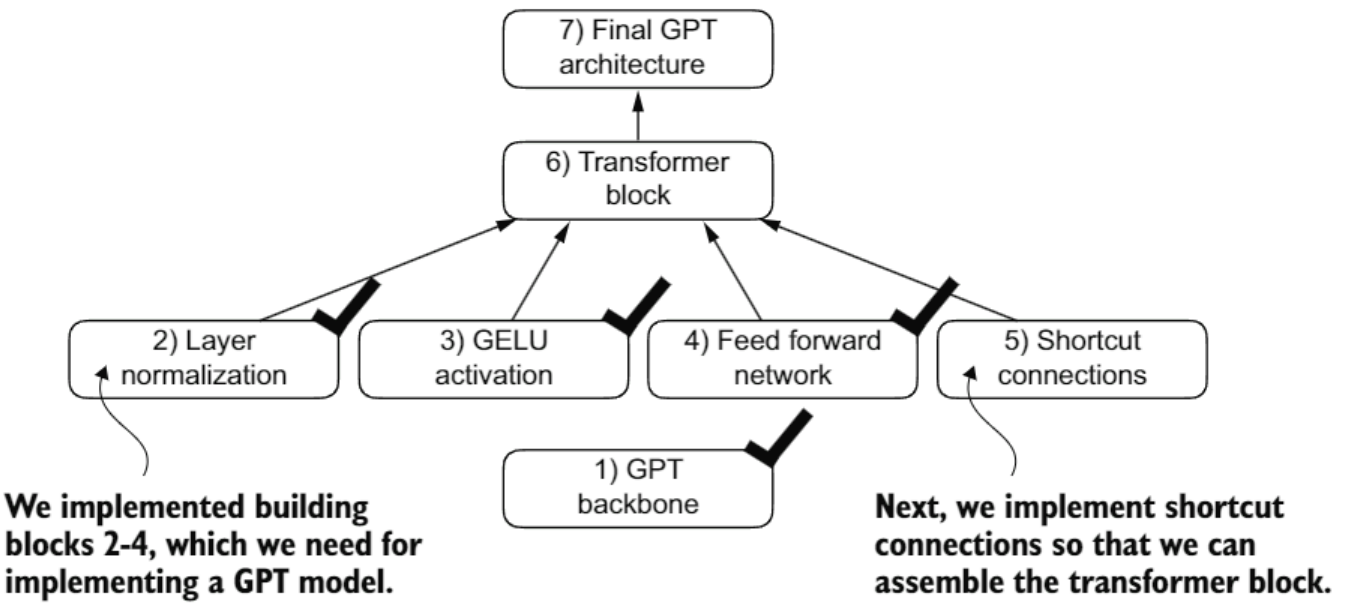

Figure 4.11 The building blocks necessary to build the GPT architecture. The black checkmarks indicating those we have already covered.

4.4 Adding shortcut connections

Let's discuss the concept behind *shortcut connections*, also known as skip or residual connections. Originally, shortcut connections were proposed for deep networks in computer vision (specifically, in residual networks) to mitigate the challenge of vanishing gradients. The vanishing gradient problem refers to the issue where gradients (which guide weight updates during training) become progressively smaller as they propagate backward through the layers, making it difficult to effectively train earlier layers.

Figure 4.12 shows that a shortcut connection creates an alternative, shorter path for the gradient to flow through the network by skipping one or more layers, which is achieved by adding the output of one layer to the output of a later layer. This is why these connections are also known as skip connections. They play a crucial role in preserving the flow of gradients during the backward pass in training.

In the following list, we implement the neural network in figure 4.12 to see how we can add shortcut connections in the `forward` method.

如图 4.11 所示，我们已实现了LLM的大部分构建模块。

接下来，我们将介绍我们在神经网络的不同层之间插入的快捷连接的概念，这对于提高深度神经网络架构的训练性能非常重要。

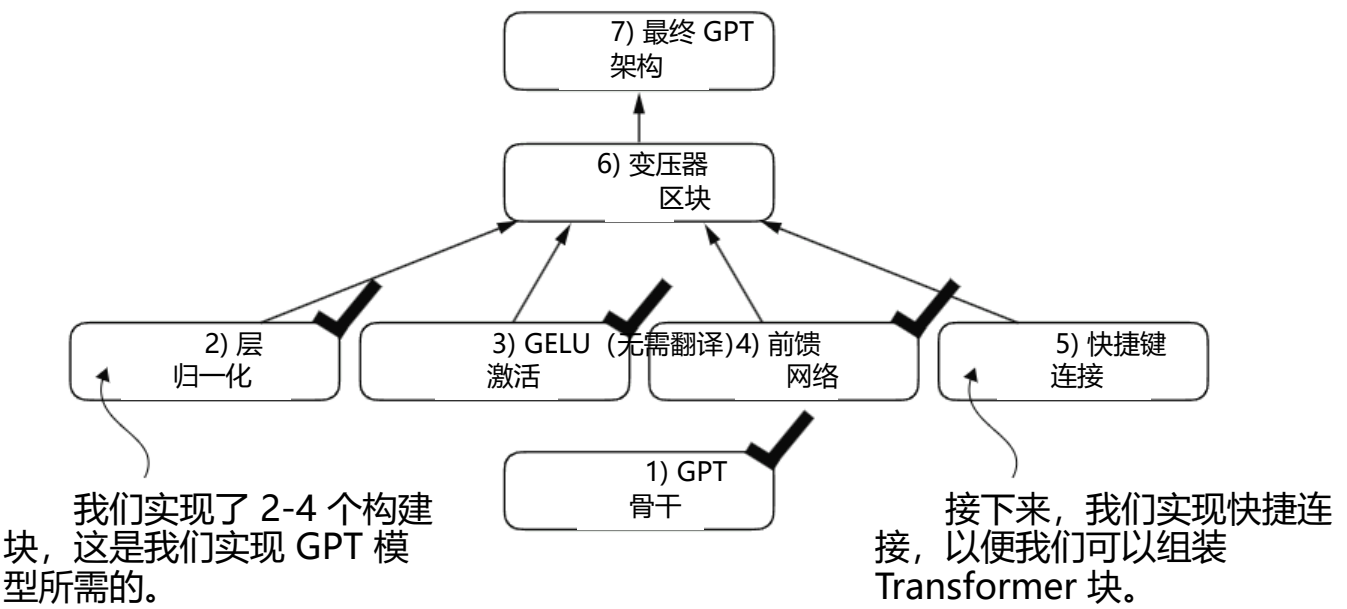

图 4.11 构建 GPT 架构所需的基石。黑色勾号表示我们已经讨论过的内容。

4.4 添加快捷连接

让我们讨论跳过连接背后的概念，也称为跳过或残差连接。最初，跳过连接是为了缓解计算机视觉中的深度网络（特别是在残差网络中）的梯度消失问题而提出的。梯度消失问题指的是梯度（在训练过程中指导权重更新的）在反向传播通过层时逐渐变小的现象，这使得有效地训练早期层变得困难。

图 4.12 显示，通过跳过一层或多层，快捷连接为梯度在网络中流动创建了一条替代的、更短的路径，这是通过将某一层的输出添加到后续层的输出中实现的。这就是为什么这些连接也被称为跳跃连接。它们在训练过程中反向传播中保持梯度流动起着至关重要的作用。

在以下列表中，我们实现了图 4.12 中的神经网络，以查看我们如何在正向方法中添加快捷连接。

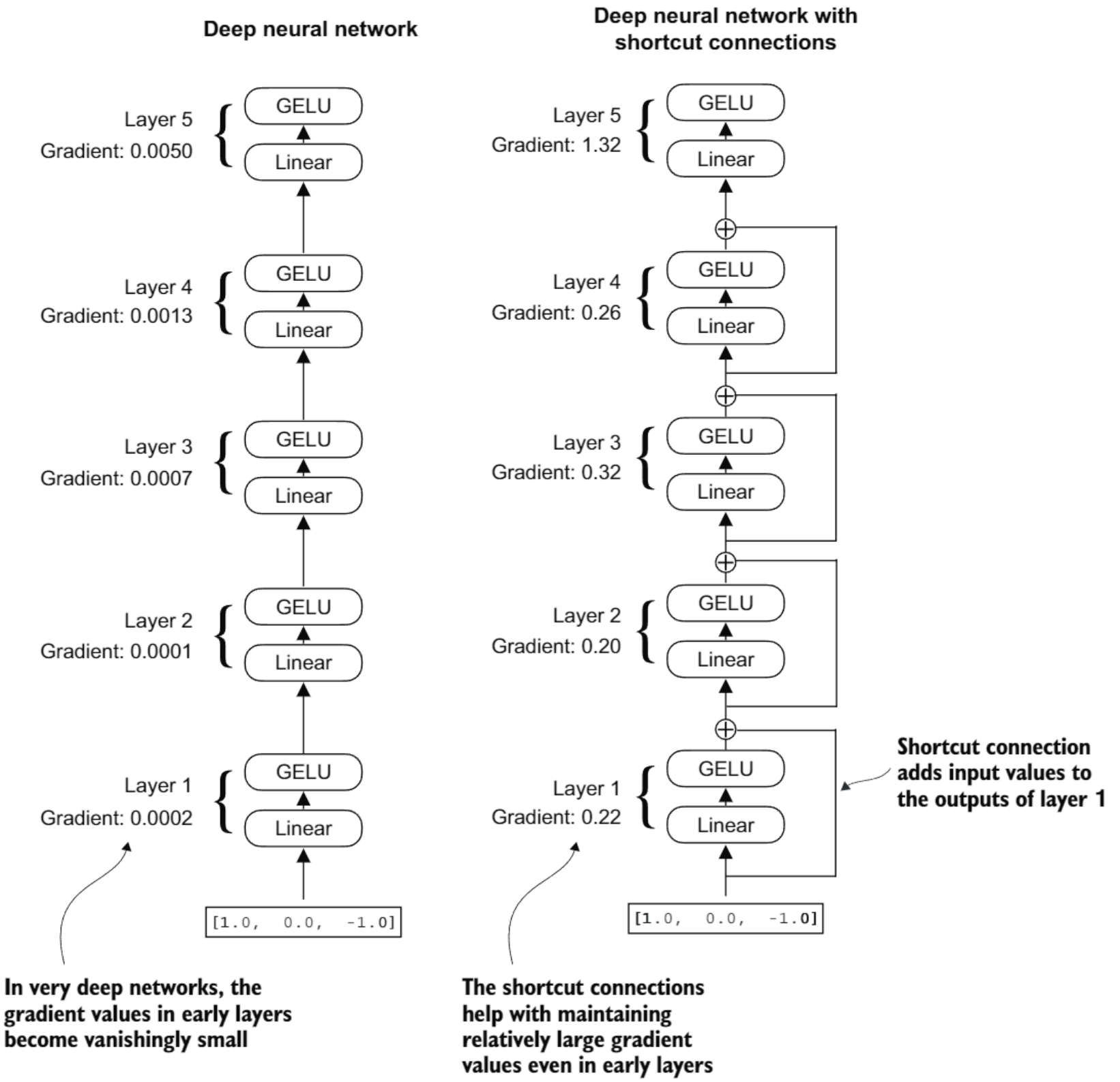

Figure 4.12 A comparison between a deep neural network consisting of five layers without (left) and with shortcut connections (right). Shortcut connections involve adding the inputs of a layer to its outputs, effectively creating an alternate path that bypasses certain layers. The gradients denote the mean absolute gradient at each layer, which we compute in listing 4.5.

Listing 4.5 A neural network to illustrate shortcut connections

```
class ExampleDeepNeuralNetwork(nn.Module):
    def __init__(self, layer_sizes, use_shortcut):
        super().__init__()
        self.use_shortcut = use_shortcut
        self.layers = nn.ModuleList([
```

Implements five layers

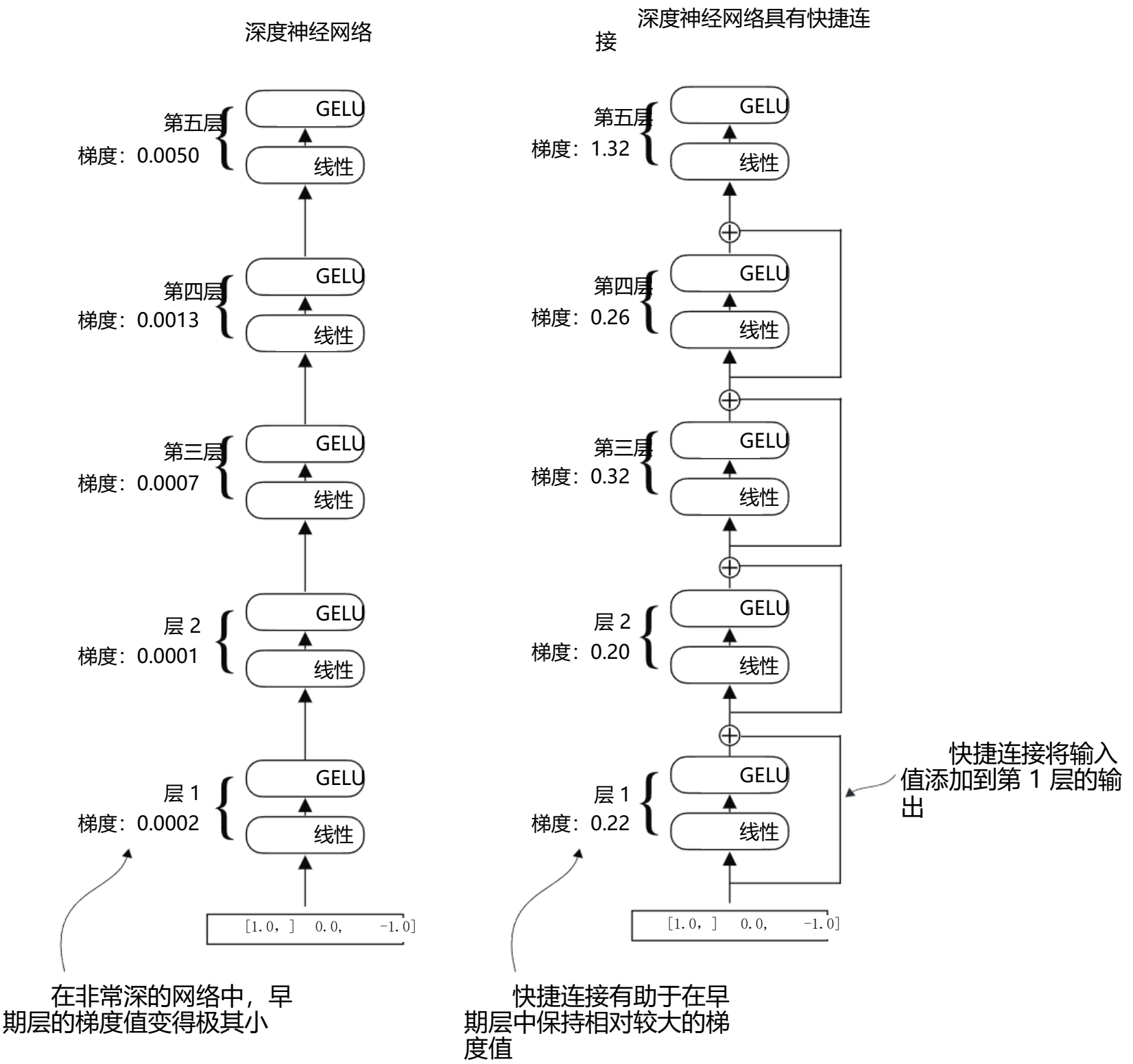

图 4.12 比较由五层组成的深度神经网络，左侧为无捷径连接，右侧为有捷径连接。捷径连接涉及将层的输入添加到其输出中，从而有效地创建一条绕过某些层的替代路径。梯度表示每层的平均绝对梯度，我们在列表 4.5 中计算。

列表 4.5 神经网络以展示快捷连接

```
class Example 深度神经网络(nn.Module):
    def __init__(self, 层数大小, 使用快捷方式):
        super().__init__()
        self.use_shortcut = use_shortcut
        self.layers = nn.ModuleList([
```

实现五层

```
            nn.Sequential(nn.Linear(layer_sizes[0], layer_sizes[1]),
                          GELU()),
            nn.Sequential(nn.Linear(layer_sizes[1], layer_sizes[2]),
                          GELU()),
            nn.Sequential(nn.Linear(layer_sizes[2], layer_sizes[3]),
                          GELU()),
            nn.Sequential(nn.Linear(layer_sizes[3], layer_sizes[4]),
                          GELU()),
            nn.Sequential(nn.Linear(layer_sizes[4], layer_sizes[5]),
                          GELU())
        ])

    def forward(self, x):
        for layer in self.layers:
            layer_output = layer(x)
            if self.use_shortcut and x.shape == layer_output.shape:
                x = x + layer_output
            else:
                x = layer_output
        return x
```

Compute the output of the current layer

Check if shortcut can be applied

The code implements a deep neural network with five layers, each consisting of a `Linear` layer and a `GELU` activation function. In the forward pass, we iteratively pass the input through the layers and optionally add the shortcut connections if the `self.use_shortcut` attribute is set to `True`.

Let's use this code to initialize a neural network without shortcut connections. Each layer will be initialized such that it accepts an example with three input values and returns three output values. The last layer returns a single output value:

```
layer_sizes = [3, 3, 3, 3, 3, 1]
sample_input = torch.tensor([[1., 0., -1.]])
torch.manual_seed(123)
model_without_shortcut = ExampleDeepNeuralNetwork(
    layer_sizes, use_shortcut=False
)
```

Specifies random seed for the initial weights for reproducibility

Next, we implement a function that computes the gradients in the model's backward pass:

```
def print_gradients(model, x):
    output = model(x)
    target = torch.tensor([[0.]])

    loss = nn.MSELoss()
    loss = loss(output, target)

    loss.backward()
```

Forward pass

Calculates loss based on how close the target and output are

Backward pass to calculate the gradients

```
            nn.Sequential(nn.Linear(layer_sizes[0],         layer_sizes[1]
            GELU()(),
        nn.Sequential(nn.Linear(layer_s                     layer_sizes[2]
        izes[1],)
            GELU()(),
        nn.Sequential(nn.Linear(layer_s                     layer_sizes[3]
        izes[2],
            GELU()(),
        nn.Sequential(nn.Linear(layer_s                     layer_sizes[4]
        izes[3],
            GELU()(),
        nn.Sequential(nn.Linear(layer_s                     layer_sizes[5]
        izes[4],                GELU()
    ])

def forward(self, x): # 定义前向传播函数
    for 层 in self.layers:
        layer_output = layer(x) 如果 self.use_shortcut 且 x.shape 等于
     layer_output.shape:
        x = x + layer_output
     else:
            x = 层输出
    返回 x
```

计算输出当前层

检查是否快捷键可以适用于

该代码实现了一个具有五层的深度神经网络，每层包含
线性层和 GELU 激活函数。在正向传播过程中，我们迭代地将输入通过层，如果 self.use_shortcut 属性设置为 True，则可选地添加快捷连接。

让我们使用此代码初始化一个不带快捷连接的神经网络。

每一层都将初始化为接受三个输入值并返回三个输出值。最后一层返回单个输出值：

```
层大小 = [3, 3, 3, 3, 3, 1] 样本输入 = torch.tensor([[1., 0., -1.]]) torch.manual_seed(123) model_without_shortcut = ExampleDeepNeuralNetwork(

层大小，使用快捷方式=False
```

指定初始权重的随机种子以实现可重复性

接下来，我们实现一个函数来计算模型反向传播中的梯度：

```
def 打印梯度(model, x):
    output = 模型(x) target = torch.tensor([[0.]])

    损失 = nn.MSELoss() 损失 = 损失(输出, 目标)

    loss.backward() 逆向传播损失
```

前向传播

根据目标和输出之间的接近程度计算损失

反向传播计算梯度

```
    for name, param in model.named_parameters():
        if 'weight' in name:
            print(f"{name} has gradient mean of {param.grad.abs().mean().item()}")
```

This code specifies a loss function that computes how close the model output and a user-specified target (here, for simplicity, the value 0) are. Then, when calling `loss.backward()`, PyTorch computes the loss gradient for each layer in the model. We can iterate through the weight parameters via `model.named_parameters()`. Suppose we have a 3 × 3 weight parameter matrix for a given layer. In that case, this layer will have 3 × 3 gradient values, and we print the mean absolute gradient of these 3 × 3 gradient values to obtain a single gradient value per layer to compare the gradients between layers more easily.

In short, the `.backward()` method is a convenient method in PyTorch that computes loss gradients, which are required during model training, without implementing the math for the gradient calculation ourselves, thereby making working with deep neural networks much more accessible.

NOTE If you are unfamiliar with the concept of gradients and neural network training, I recommend reading sections A.4 and A.7 in appendix A.

Let's now use the `print_gradients` function and apply it to the model without skip connections:

```
print_gradients(model_without_shortcut, sample_input)
```

The output is

```
layers.0.0.weight has gradient mean of 0.00020173587836325169
layers.1.0.weight has gradient mean of 0.0001201116101583466
layers.2.0.weight has gradient mean of 0.0007152041653171182
layers.3.0.weight has gradient mean of 0.001398873864673078
layers.4.0.weight has gradient mean of 0.005049646366387606
```

The output of the `print_gradients` function shows, the gradients become smaller as we progress from the last layer (`layers.4`) to the first layer (`layers.0`), which is a phenomenon called the *vanishing gradient problem*.

Let's now instantiate a model with skip connections and see how it compares:

```
torch.manual_seed(123)
model_with_shortcut = ExampleDeepNeuralNetwork(
    layer_sizes, use_shortcut=True
)
print_gradients(model_with_shortcut, sample_input)
```

The output is

```
for name, param in 模型.named_parameters():
    如果 'weight' 在 name 中:
        print(f"{name} 的梯度均值为 {param.grad.abs().mean().item()}")
```

这段代码指定了一个损失函数，用于计算模型输出与用户指定的目标（这里，为了简单起见，是值 0）之间的接近程度。然后，在调用

loss.backward()，PyTorch 计算模型中每层的损失梯度。我们可以通过 model.named_parameters()遍历权重参数。假设我们有一个给定层的 3×3 权重参数矩阵。在这种情况下，该层将有 3×3 的梯度值，我们打印这些 3×3 梯度值的平均绝对梯度，以获得每个层的单个梯度值，以便更容易地比较层之间的梯度。

简而言之，.backward() 方法是 PyTorch 中一个方便的方法，它计算损失梯度，这在模型训练期间是必需的，无需我们自己实现梯度计算的数学，从而使得与深度神经网络的工作变得更加容易。

注意：如果您不熟悉梯度概念和神经网络训练，建议您阅读附录 A 中的 A.4 和 A.7 节。

现在使用 print_gradients 函数，并将其应用于没有跳过连接的模型：

```
打印梯度(model_without_shortcut,)                    sample_input)
```

输出结果

```
layers.0.0.weight 的梯度均值为 0.00020173587836325169
layers.1.0.weight 的梯度均值为 0.0001201116101583466 layers.2.0.weight 的
梯度均值为 0.0007152041653171182 layers.3.0.weight 的梯度均值为
0.001398873864673078 layers.4.0.weight 的梯度均值为 0.005049646366387606
```

打印梯度函数的输出显示，随着我们从最后一层（layers.4）向第一层（layers.0）推进，梯度变得越小，这是一种称为梯度消失问题的现象。

现在让我们实例化一个具有跳过连接的模型，看看它的表现如何：

```
torch 手动设置随机种子(123) model_with_shortcut =
ExampleDeepNeuralNetwork(
层大小，使用快捷方式=True) 打印带有快捷
方式的模型的梯度
                                                    sample_input)
```

输出结果

```
layers.0.0.weight has gradient mean of 0.22169792652130127
layers.1.0.weight has gradient mean of 0.20694105327129364
layers.2.0.weight has gradient mean of 0.32896995544433594
layers.3.0.weight has gradient mean of 0.2665732502937317
layers.4.0.weight has gradient mean of 1.3258541822433472
```

The last layer (`layers.4`) still has a larger gradient than the other layers. However, the gradient value stabilizes as we progress toward the first layer (`layers.0`) and doesn't shrink to a vanishingly small value.

In conclusion, shortcut connections are important for overcoming the limitations posed by the vanishing gradient problem in deep neural networks. Shortcut connections are a core building block of very large models such as LLMs, and they will help facilitate more effective training by ensuring consistent gradient flow across layers when we train the GPT model in the next chapter.

Next, we'll connect all of the previously covered concepts (layer normalization, GELU activations, feed forward module, and shortcut connections) in a transformer block, which is the final building block we need to code the GPT architecture.

4.5 Connecting attention and linear layers in a transformer block

Now, let's implement the *transformer block*, a fundamental building block of GPT and other LLM architectures. This block, which is repeated a dozen times in the 124-million-parameter GPT-2 architecture, combines several concepts we have previously covered: multi-head attention, layer normalization, dropout, feed forward layers, and GELU activations. Later, we will connect this transformer block to the remaining parts of the GPT architecture.

Figure 4.13 shows a transformer block that combines several components, including the masked multi-head attention module (see chapter 3) and the `FeedForward` module we previously implemented (see section 4.3). When a transformer block processes an input sequence, each element in the sequence (for example, a word or subword token) is represented by a fixed-size vector (in this case, 768 dimensions). The operations within the transformer block, including multi-head attention and feed forward layers, are designed to transform these vectors in a way that preserves their dimensionality.

The idea is that the self-attention mechanism in the multi-head attention block identifies and analyzes relationships between elements in the input sequence. In contrast, the feed forward network modifies the data individually at each position. This combination not only enables a more nuanced understanding and processing of the input but also enhances the model's overall capacity for handling complex data patterns.

```
    layers.0.0.weight 的梯度均值为 0.22169792652130127
layers.1.0.weight 的梯度均值为 0.20694105327129364 layers.2.0.weight
的梯度均值为 0.32896995544433594 layers.3.0.weight 的梯度均值为
0.2665732502937317 layers.4.0.weight 的梯度均值为 1.3258541822433472
```

最后一层（layers.4）的梯度仍然比其他层大。然而，随着我们向第一层（layers.0）前进，梯度值趋于稳定，并不会缩小到极小的值。

总结来说，捷径连接对于克服深度神经网络中梯度消失问题带来的限制至关重要。捷径连接是像LLMs这样的大型模型的核心构建块，它们将有助于通过确保在下一章训练 GPT 模型时层间梯度流的连续性，从而促进更有效的训练。

接下来，我们将把之前介绍的所有概念（层归一化、GELU 激活、前馈模块和跳跃连接）连接到一个 Transformer 块中，这是我们编码 GPT 架构所需的最后一个构建块。

4.5 连接注意力层和线性层在 Transformer 块中

现在，让我们实现 Transformer 块，这是 GPT 和其他LLM架构的基本构建块。这个块在 124 百万参数的 GPT-2 架构中重复了十几次，结合了我们之前介绍过的几个概念：多头注意力、层归一化、dropout、前馈层和 GELU 激活。稍后，我们将把这个 Transformer 块连接到 GPT 架构的其余部分。

图 4.13 展示了一个结合了多个组件的变压器块，包括掩码多头注意力模块（见第 3 章）和之前实现的 FeedForward 模块（见第 4.3 节）。当变压器块处理一个输入序列时，序列中的每个元素（例如，一个词或子词标记）由一个固定大小的向量表示（在这种情况下，768 维）。变压器块内的操作，包括多头注意力和前馈层，旨在以保留其维度的这种方式转换这些向量。

该想法是多头注意力块中的自注意力机制识别和分析输入序列中元素之间的关系。相比之下，前馈网络在每个位置单独修改数据。这种组合不仅使对输入的理解和处理更加细腻，而且增强了模型处理复杂数据模式的整体能力。

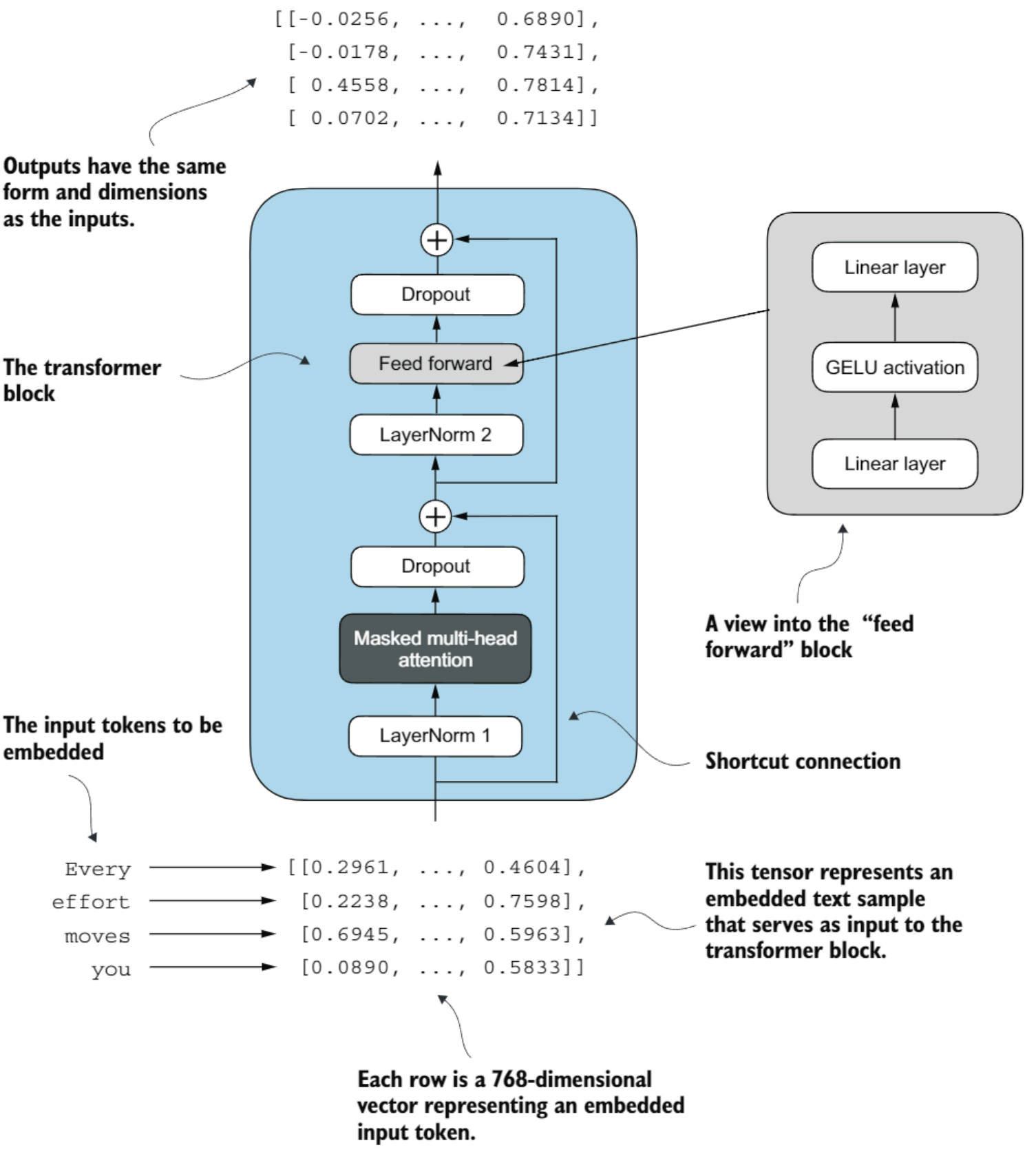

Figure 4.13 An illustration of a transformer block. Input tokens have been embedded into 768-dimensional vectors. Each row corresponds to one token's vector representation. The outputs of the transformer block are vectors of the same dimension as the input, which can then be fed into subsequent layers in an LLM.

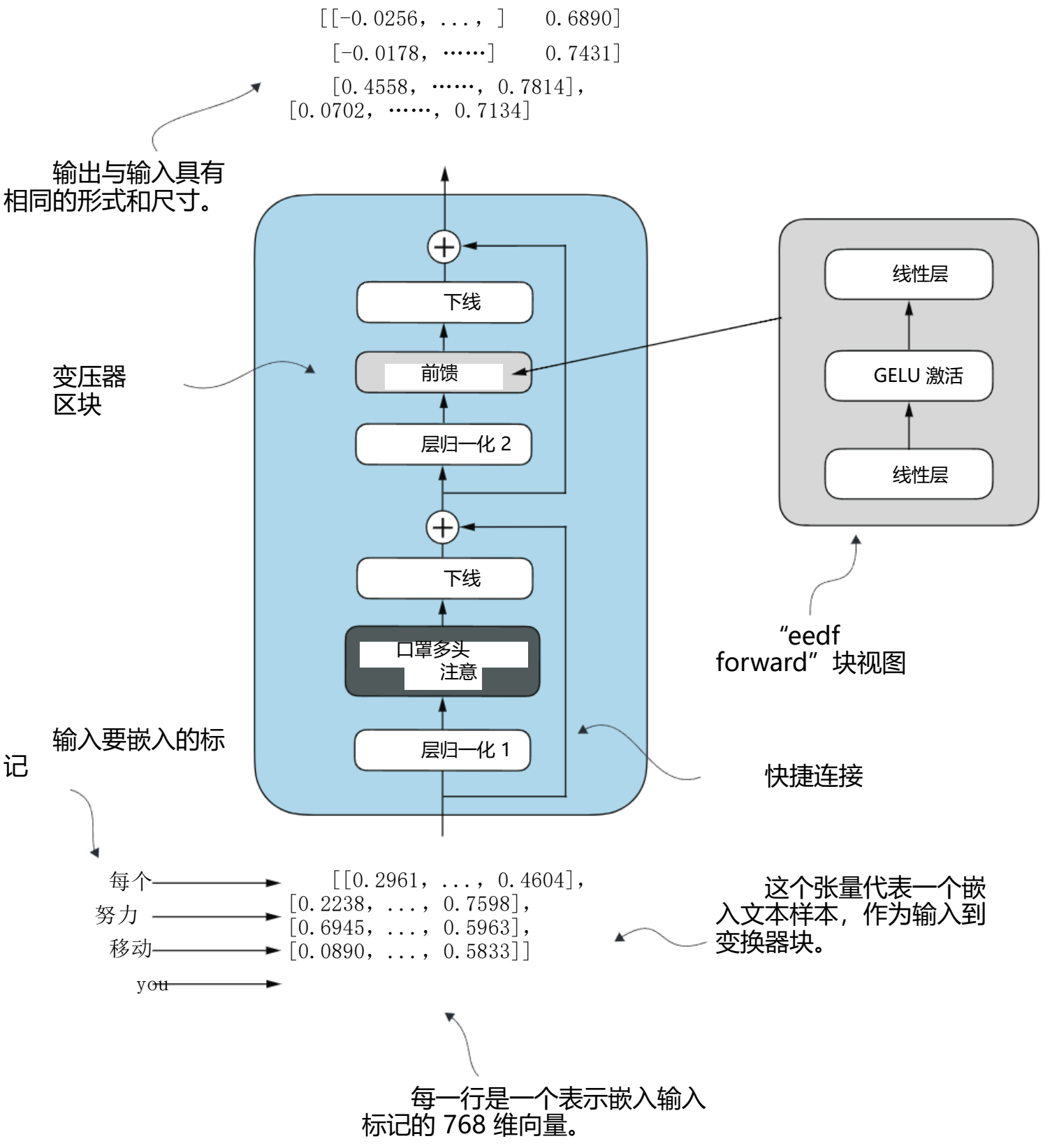

图 4.13 变压器块的示意图。输入标记已嵌入到 768 维向量中。每一行对应一个标记的向量表示。变压器块的输出与输入具有相同维度的向量，然后可以输入到后续层中。

We can create the `TransformerBlock` in code.

Listing 4.6 The transformer block component of GPT

```
from chapter03 import MultiHeadAttention

class TransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.att = MultiHeadAttention(
            d_in=cfg["emb_dim"],
            d_out=cfg["emb_dim"],
            context_length=cfg["context_length"],
            num_heads=cfg["n_heads"],
            dropout=cfg["drop_rate"],
            qkv_bias=cfg["qkv_bias"])
        self.ff = FeedForward(cfg)
        self.norm1 = LayerNorm(cfg["emb_dim"])
        self.norm2 = LayerNorm(cfg["emb_dim"])
        self.drop_shortcut = nn.Dropout(cfg["drop_rate"])

    def forward(self, x):
        shortcut = x                                  <- Shortcut connection for attention block
        x = self.norm1(x)
        x = self.att(x)
        x = self.drop_shortcut(x)
        x = x + shortcut                              <- Add the original input back

        shortcut = x                                  <- Shortcut connection for feed forward block
        x = self.norm2(x)
        x = self.ff(x)
        x = self.drop_shortcut(x)
        x = x + shortcut                              <- Adds the original input back
        return x
```

The given code defines a `TransformerBlock` class in PyTorch that includes a multi-head attention mechanism (`MultiHeadAttention`) and a feed forward network (`FeedForward`), both configured based on a provided configuration dictionary (`cfg`), such as `GPT_CONFIG_124M`.

Layer normalization (`LayerNorm`) is applied before each of these two components, and dropout is applied after them to regularize the model and prevent overfitting. This is also known as *Pre-LayerNorm*. Older architectures, such as the original transformer model, applied layer normalization after the self-attention and feed forward networks instead, known as *Post-LayerNorm*, which often leads to worse training dynamics.

我们可以用代码创建 TransformerBlock。

列表 4.6 GPT 的 Transformer 块组件

```
from  第三章 导入 MultiHeadAttention


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"], 上下文长度
           =cfg["context_length"], 头数=cfg["n_heads"],
           dropout=cfg["drop_rate"]


          qkv_bias=cfg["qkv_bias"]) self.ff = FeedForward(cfg)
       self.norm1 = LayerNorm(cfg["emb_dim"]) self.norm2 =
       LayerNorm(cfg["emb_dim"]) self.drop_shortcut =
       nn.Dropout(cfg["drop_rate"])



     def forward(self, x): # 定义前向传播函数
                                                  快捷连接
                                                  for 注意块
          快捷键 = x
          x = self.norm1(x)
          x = self.att(x)
          x = self.drop_shortcut(x) x
= x + shortcut                                    添加原始
                                                  输入回退

          快捷键 = x                               快捷连接前馈块
          x = self.norm2(x)
          x = self.ff(x)
          x = self.drop_shortcut(x) x
= x + shortcut return x                           添加原始输
                                                  入
```

给定的代码在 PyTorch 中定义了一个 TransformerBlock 类，该类包含一个多头注意力机制（MultiHeadAttention）和一个前馈网络（FeedForward），两者均基于提供的配置字典（cfg）进行配置

GPT_CONFIG_124M.

层归一化（LayerNorm）应用于这两个组件之前，并在它们之后应用 dropout 以正则化模型并防止过拟合。这也被称为预层归一化。较老的架构，如原始的 Transformer 模型，将层归一化应用于自注意力和前馈网络之后，称为后层归一化，这通常会导致更差的训练动态。

The class also implements the forward pass, where each component is followed by a shortcut connection that adds the input of the block to its output. This critical feature helps gradients flow through the network during training and improves the learning of deep models (see section 4.4).

Using the `GPT_CONFIG_124M` dictionary we defined earlier, let's instantiate a transformer block and feed it some sample data:

```
torch.manual_seed(123)
x = torch.rand(2, 4, 768)                          <-- Creates sample input of shape [batch_size, num_tokens, emb_dim]
block = TransformerBlock(GPT_CONFIG_124M)
output = block(x)

print("Input shape:", x.shape)
print("Output shape:", output.shape)
```

The output is

```
Input shape: torch.Size([2, 4, 768])
Output shape: torch.Size([2, 4, 768])
```

As we can see, the transformer block maintains the input dimensions in its output, indicating that the transformer architecture processes sequences of data without altering their shape throughout the network.

The preservation of shape throughout the transformer block architecture is not incidental but a crucial aspect of its design. This design enables its effective application across a wide range of sequence-to-sequence tasks, where each output vector directly corresponds to an input vector, maintaining a one-to-one relationship. However, the output is a context vector that encapsulates information from the entire input sequence (see chapter 3). This means that while the physical dimensions of the sequence (length and feature size) remain unchanged as it passes through the transformer block, the content of each output vector is re-encoded to integrate contextual information from across the entire input sequence.

With the transformer block implemented, we now have all the building blocks needed to implement the GPT architecture. As illustrated in figure 4.14, the transformer block combines layer normalization, the feed forward network, GELU activations, and shortcut connections. As we will eventually see, this transformer block will make up the main component of the GPT architecture.

该类还实现了前向传递，其中每个组件后面都跟着一个快捷连接，该连接将块的输入添加到其输出中。这一关键特性有助于在训练期间使梯度在网络中流动，并提高深度模型的学习效果（见第 4.4 节）。

使用我们之前定义的 GPT_CONFIG_124M 字典，让我们实例化一个 Transformer 块，并向它提供一些样本数据：

```
torch.manual_seed(123) x = torch.rand(2, 4, 768) block = TransformerBlock(GPT_CONFIG_124M) output = block(x)

打印("输入形状：", x.shape) 打印("输出形状：")
```

创建形状为[batch_size, num_tokens, emb_dim]的样本输入

输出形状)

输出结果

```
输入形状：torch.Size([2, 4, 768]) 输出形状：torch.Size([2, 4,])
```

768])

如您所见，Transformer 块在其输出中保持了输入维度，这表明 Transformer 架构在处理数据序列时不会改变它们的形状，在整个网络中保持其形状。

形状在整个变换器块架构中的保留并非偶然，而是其设计的关键方面。这种设计使其能够有效地应用于广泛的序列到序列任务中，其中每个输出向量直接对应于一个输入向量，保持一对一的关系。然而，输出是一个上下文向量，它封装了整个输入序列的信息（见第 3 章）。这意味着，尽管序列的物理维度（长度和特征大小）在通过变换器块时保持不变，但每个输出向量的内容被重新编码，以整合整个输入序列的上下文信息。

随着 transformer 块的实现，我们现在拥有了实现 GPT 架构所需的所有构建块。如图 4.14 所示，transformer 块结合了层归一化、前馈网络、GELU 激活和快捷连接。正如我们最终将看到的，这个 transformer 块将构成 GPT 架构的主要组成部分。

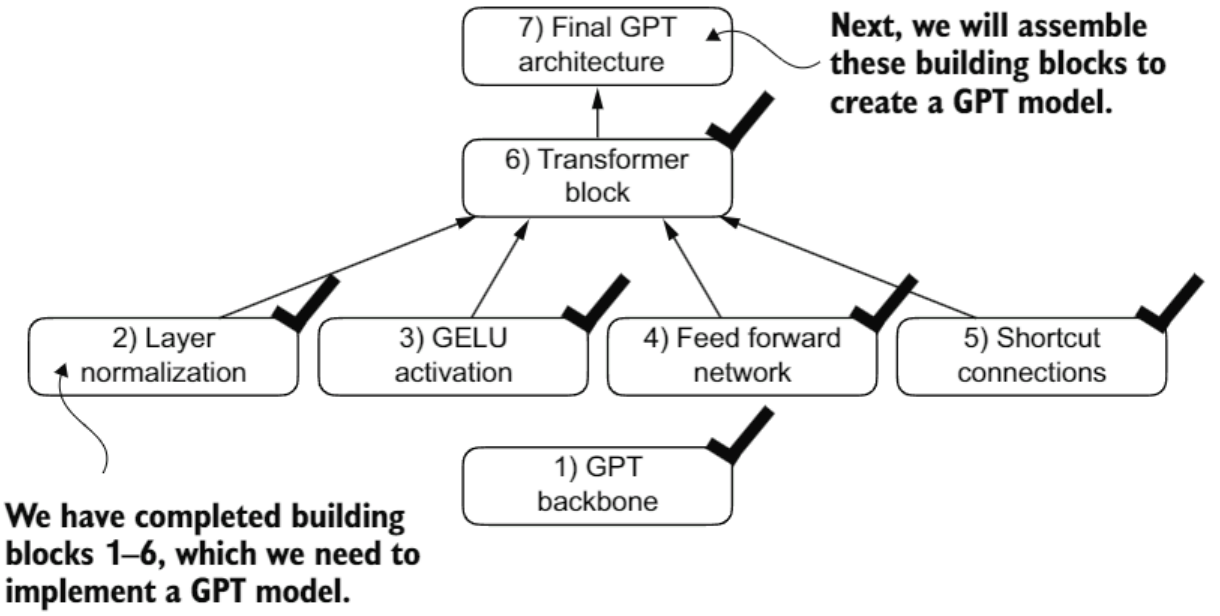

Figure 4.14 The building blocks necessary to build the GPT architecture. The black checks indicate the blocks we have completed.

4.6 Coding the GPT model

We started this chapter with a big-picture overview of a GPT architecture that we called `DummyGPTModel`. In this `DummyGPTModel` code implementation, we showed the input and outputs to the GPT model, but its building blocks remained a black box using a `DummyTransformerBlock` and `DummyLayerNorm` class as placeholders.

Let's now replace the `DummyTransformerBlock` and `DummyLayerNorm` placeholders with the real `TransformerBlock` and `LayerNorm` classes we coded previously to assemble a fully working version of the original 124-million-parameter version of GPT-2. In chapter 5, we will pretrain a GPT-2 model, and in chapter 6, we will load in the pretrained weights from OpenAI.

Before we assemble the GPT-2 model in code, let's look at its overall structure, as shown in figure 4.15, which includes all the concepts we have covered so far. As we can see, the transformer block is repeated many times throughout a GPT model architecture. In the case of the 124-million-parameter GPT-2 model, it's repeated 12 times, which we specify via the `n_layers` entry in the `GPT_CONFIG_124M` dictionary. This transform block is repeated 48 times in the largest GPT-2 model with 1,542 million parameters.

The output from the final transformer block then goes through a final layer normalization step before reaching the linear output layer. This layer maps the transformer's output to a high-dimensional space (in this case, 50,257 dimensions, corresponding to the model's vocabulary size) to predict the next token in the sequence.

Let's now code the architecture in figure 4.15.

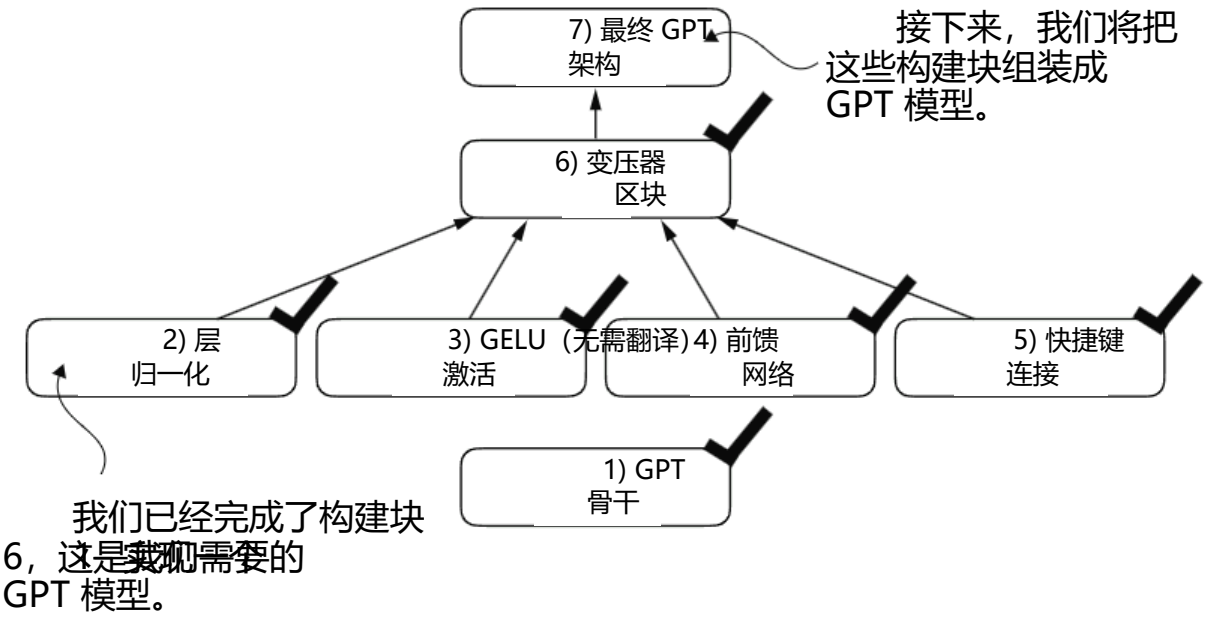

图 4.14 构建 GPT 架构所需的构建块。黑色勾号表示我们已完成的模块。

4.6 编码 GPT 模型

我们以对名为 DummyGPTModel 的 GPT 架构的整体概述开始了这一章节。在这个 DummyGPTModel 代码实现中，我们展示了 GPT 模型的输入和输出，但其构建模块仍然是一个黑盒

使用 DummyTransformerBlock 和 DummyLayerNorm 类作为占位符。

现在将 DummyTransformerBlock 和 DummyLayerNorm 占位符替换为我们之前编写的真实 TransformerBlock 和 LayerNorm 类，以组装一个完全工作的原始 1.24 亿参数版本 GPT-2 的完整版本。在第 5 章，我们将预训练一个 GPT-2 模型，在第 6 章，我们将从 OpenAI 加载预训练的权重。

在将 GPT-2 模型代码组装之前，让我们先看看其整体结构，如图 4.15 所示，其中包含了我们迄今为止所涉及的所有概念。正如我们所见，在整个 GPT 模型架构中，transformer 块被重复多次。在 1.24 亿参数的 GPT-2 模型中，它被重复了 12 次，我们通过 GPT_CONFIG_124M 字典中的 n_layers 条目来指定。在具有 15.42 亿参数的最大 GPT-2 模型中，这个转换块被重复了 48 次。

最终变换器块的输出随后经过最终层归一化步骤，然后到达线性输出层。该层将变换器的输出映射到一个高维空间（在这种情况下，为 50,257 维，对应于模型的词汇量）以预测序列中的下一个标记。

现在让我们编写图 4.15 中的架构代码。

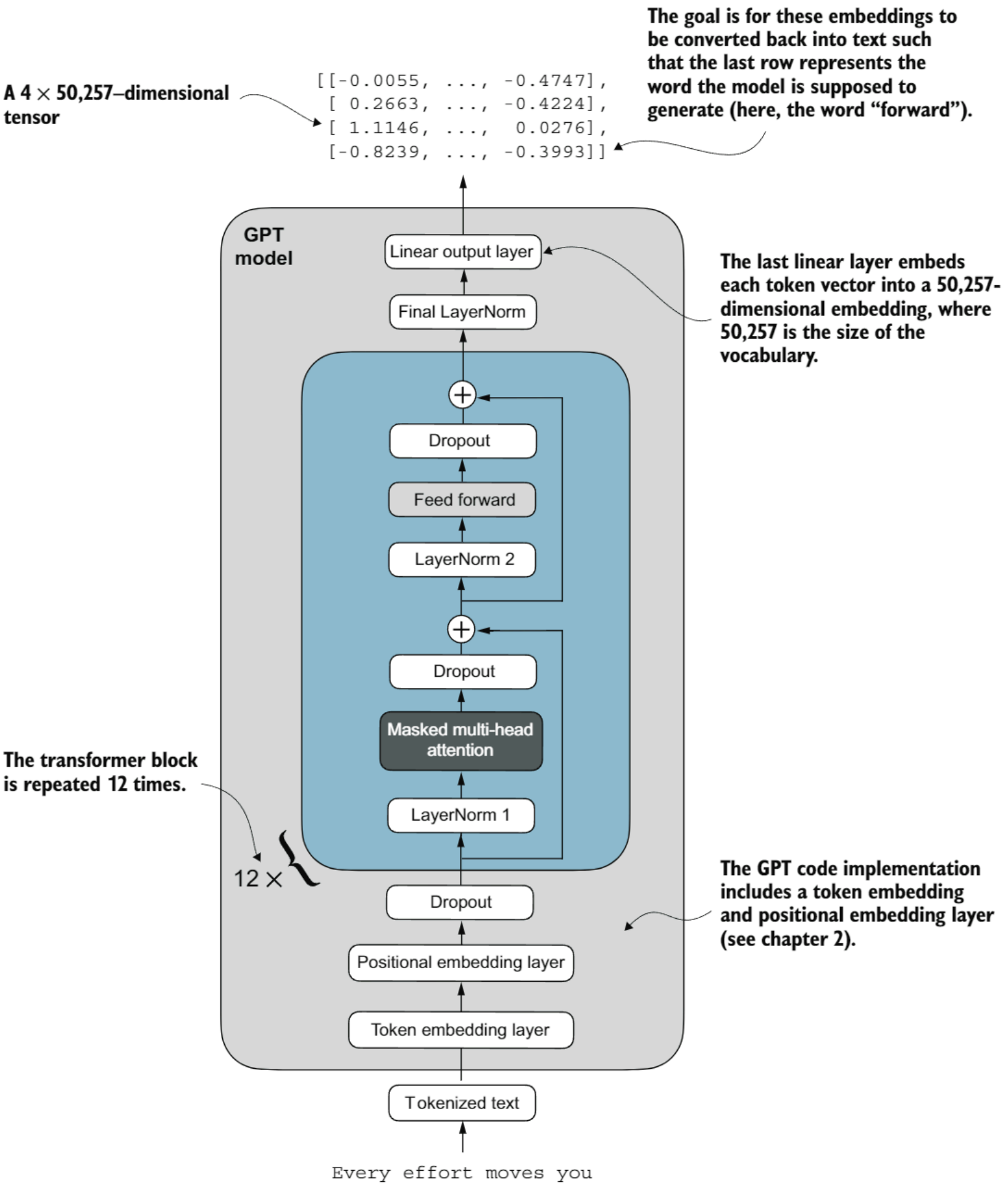

Figure 4.15 An overview of the GPT model architecture showing the flow of data through the GPT model. Starting from the bottom, tokenized text is first converted into token embeddings, which are then augmented with positional embeddings. This combined information forms a tensor that is passed through a series of transformer blocks shown in the center (each containing multi-head attention and feed forward neural network layers with dropout and layer normalization), which are stacked on top of each other and repeated 12 times.

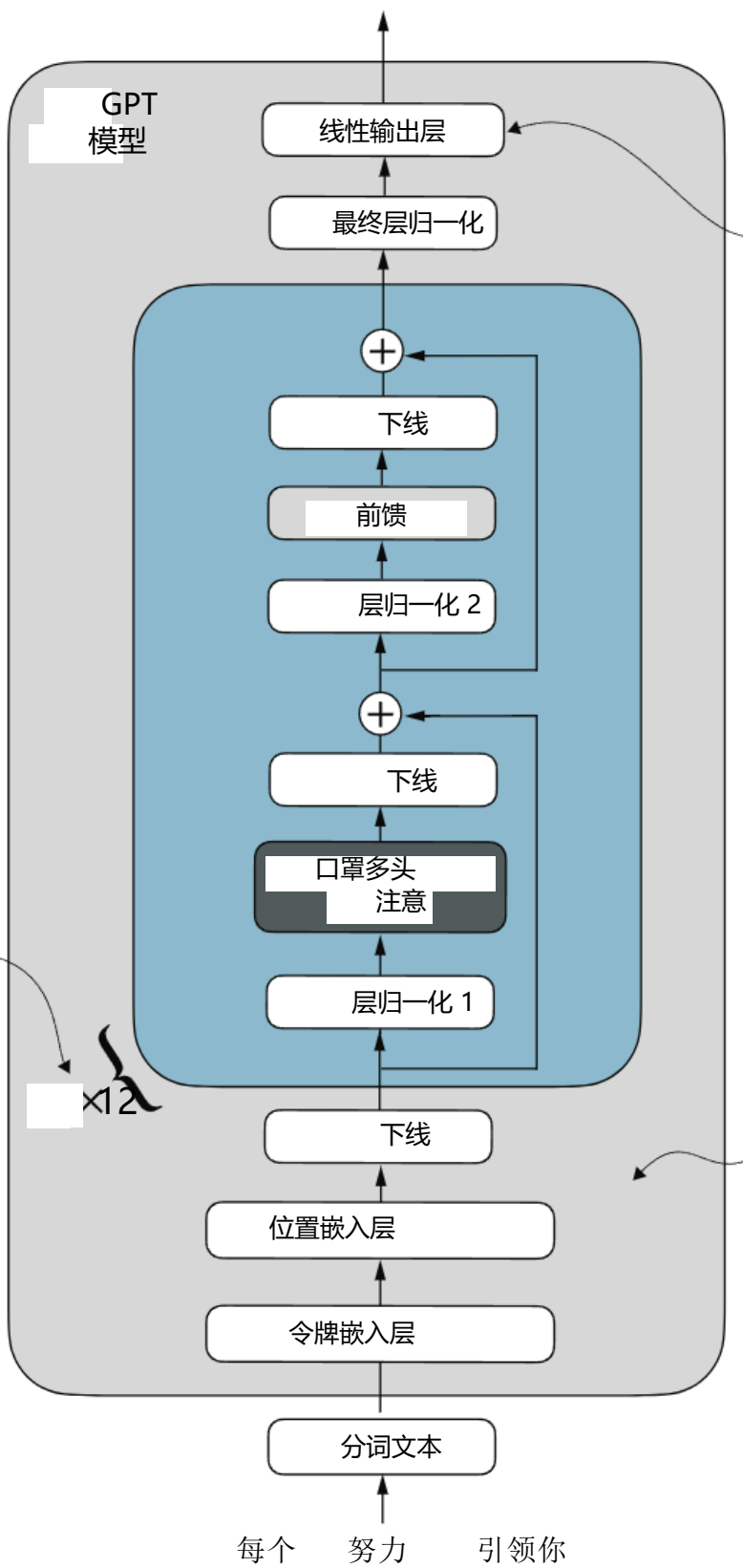

图 4.15 GPT 模型架构概述，展示了数据在 GPT 模型中的流动。从底部开始，标记化文本首先被转换为标记嵌入，然后添加位置嵌入。这些组合信息形成一个张量，通过中心显示的一系列变换器块（每个包含多头注意力和具有 dropout 和层归一化的前馈神经网络层），这些块堆叠在一起并重复 12 次。

Listing 4.7 The GPT model architecture implementation

```
class GPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["emb_dim"])
        self.drop_emb = nn.Dropout(cfg["drop_rate"])

        self.trf_blocks = nn.Sequential(
            *[TransformerBlock(cfg) for _ in range(cfg["n_layers"])])

        self.final_norm = LayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(
            cfg["emb_dim"], cfg["vocab_size"], bias=False
        )

    def forward(self, in_idx):
        batch_size, seq_len = in_idx.shape
        tok_embeds = self.tok_emb(in_idx)

        pos_embeds = self.pos_emb(
            torch.arange(seq_len, device=in_idx.device)
        )
        x = tok_embeds + pos_embeds
        x = self.drop_emb(x)
        x = self.trf_blocks(x)
        x = self.final_norm(x)
        logits = self.out_head(x)
        return logits
```

The device setting will allow us to train the model on a CPU or GPU, depending on which device the input data sits on.

Thanks to the `TransformerBlock` class, the `GPTModel` class is relatively small and compact.

The `__init__` constructor of this `GPTModel` class initializes the token and positional embedding layers using the configurations passed in via a Python dictionary, `cfg`. These embedding layers are responsible for converting input token indices into dense vectors and adding positional information (see chapter 2).

Next, the `__init__` method creates a sequential stack of `TransformerBlock` modules equal to the number of layers specified in `cfg`. Following the transformer blocks, a `LayerNorm` layer is applied, standardizing the outputs from the transformer blocks to stabilize the learning process. Finally, a linear output head without bias is defined, which projects the transformer's output into the vocabulary space of the tokenizer to generate logits for each token in the vocabulary.

The forward method takes a batch of input token indices, computes their embeddings, applies the positional embeddings, passes the sequence through the transformer blocks, normalizes the final output, and then computes the logits, representing the next token's unnormalized probabilities. We will convert these logits into tokens and text outputs in the next section.

列表 4.7 GPT 模型架构实现

```
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"])


        self.trf_blocks = nn.Sequential()
            *[Transformer 块（cfg）]    for  _ in  range(cfg["n_layers"])])


        self.final_norm = 层归一化(cfg["emb_dim"])
 self.out_head = 线性层(nn.Linear(
        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)


        pos_embeds = 自身.pos_emb(
        torch.arange(seq_len, device=in_idx.device) ) x =
 tok_embeds + pos_embeds x = self.drop_emb(x) x =
 self.trf_blocks(x) x = self.final_norm(x) logits =
 self.out_head(x) return logits
```

设备设置将允许我们在 CPU 或 GPU 上训练模型，具体取决于输入数据所在的设备。

得益于 TransformerBlock 类，GPTModel 类相对较小且紧凑。

该 GPTModel 类的 __init__ 构造函数使用通过 Python 字典 cfg 传入的配置初始化标记和位置嵌入层。这些嵌入层负责将输入标记索引转换为密集向量并添加位置信息（见第 2 章）。

接下来，__init__方法创建了一个等于 cfg 中指定层数的 TransformerBlock 模块的顺序堆栈。在 transformer 块之后，应用了一个 LayerNorm 层，将 transformer 块的输出标准化以稳定学习过程。最后，定义了一个不带偏置的线性输出头，将 transformer 的输出投影到分词器的词汇空间，为词汇表中的每个标记生成 logits。

前向方法接收一批输入标记索引，计算它们的嵌入，应用位置嵌入，将序列通过 Transformer 块，对最终输出进行归一化，然后计算 logits，表示下一个标记的非归一化概率。我们将在下一节将这些 logits 转换为标记和文本输出。

Let's now initialize the 124-million-parameter GPT model using the `GPT_CONFIG_124M` dictionary we pass into the `cfg` parameter and feed it with the batch text input we previously created:

```
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)

out = model(batch)
print("Input batch:\n", batch)
print("\nOutput shape:", out.shape)
print(out)
```

This code prints the contents of the input batch followed by the output tensor:

```
Input batch:
 tensor([[6109,  3626,  6100,   345],        <- Token IDs of text 1
         [6109,  1110,  6622,   257]])       <- Token IDs of text 2

Output shape: torch.Size([2, 4, 50257])
tensor([[[ 0.3613,  0.4222, -0.0711,  ...,  0.3483,  0.4661, -0.2838],
         [-0.1792, -0.5660, -0.9485,  ...,  0.0477,  0.5181, -0.3168],
         [ 0.7120,  0.0332,  0.1085,  ...,  0.1018, -0.4327, -0.2553],
         [-0.1076,  0.3418, -0.1190,  ...,  0.7195,  0.4023,  0.0532]],

        [[-0.2564,  0.0900,  0.0335,  ...,  0.2659,  0.4454, -0.6806],
         [ 0.1230,  0.3653, -0.2074,  ...,  0.7705,  0.2710,  0.2246],
         [ 1.0558,  1.0318, -0.2800,  ...,  0.6936,  0.3205, -0.3178],
         [-0.1565,  0.3926,  0.3288,  ...,  1.2630, -0.1858,  0.0388]]],
       grad_fn=<UnsafeViewBackward0>)
```

As we can see, the output tensor has the shape `[2, 4, 50257]`, since we passed in two input texts with four tokens each. The last dimension, `50257`, corresponds to the vocabulary size of the tokenizer. Later, we will see how to convert each of these 50,257-dimensional output vectors back into tokens.

Before we move on to coding the function that converts the model outputs into text, let's spend a bit more time with the model architecture itself and analyze its size. Using the `numel()` method, short for "number of elements," we can collect the total number of parameters in the model's parameter tensors:

```
total_params = sum(p.numel() for p in model.parameters())
print(f"Total number of parameters: {total_params:,}")
```

The result is

```
Total number of parameters: 163,009,536
```

Now, a curious reader might notice a discrepancy. Earlier, we spoke of initializing a 124-million-parameter GPT model, so why is the actual number of parameters 163 million?

现在使用我们传递给 cfg 参数的 GPT_CONFIG_124M 字典初始化 124 百万参数的 GPT 模型，并用我们之前创建的批文本输入进行喂养：

```
torch 手动设置随机种子
(123)模型  GPT 模型（GPT_CONFIG_124M）

输出 = 模型（批次）打印（"输入批次：
\n"，批次）打印（"\n 输出形状："，输
出.shape）打印（输出）
```

这段代码打印输入批次的内容，然后是输出张量：

```
输入批次：
 张量([[6109,        3626,    6100,      345]          <-- 文本 1 的标记 ID
          [6109，]  1110,    6622,      257]])        <-- 文本 2 的标记 ID

输出形状：torch.Size([2, 4, 50257]) 张量([[[
0.3613, 0.4222, -0.0711,                                ...,    0.3483,    0.4661,   -0.2838]
          [-0.1792, -0.5660,]    -0.9485,    ...,    0.0477,    0.5181,   -0.3168]
          [0.7120, 0.0332, 0.1085, ..., 0.1018, -0.4327, -0.2553], [-1.0076,
       0.3418, -0.1190, ..., 0.7195, 0.4023,]                                0.0532]]

          [-0.2564,]    0.0900,    0.0335,    ...,    0.2659,    0.4454,   -0.6806]
           [ 0.1230,    0.3653,   -0.2074,    ...,    0.7705,    0.2710,    0.2246]
           [ 1.0558,    1.0318,   -0.2800,    ...,    0.6936,    0.3205,   -0.3178]
          [-0.1565, 0.3926, 0.3288, ..., 1.2630, -0.1858, 0.0388]]], grad_fn=)
```

如您所见，输出张量的形状为[2, 4, 50257]，因为我们输入了两个每个包含四个标记的文本。最后一个维度，50257，对应于分词器的词汇量。稍后，我们将看到如何将这些 50,257 维输出向量转换回标记。

在继续编写将模型输出转换为文本的功能之前，让我们花更多的时间来分析模型架构本身及其大小。使用 numel()方法，即“元素数量”，我们可以收集模型参数张量中的总参数数：

```
总参数数：{total_params:,}
```

结果是

```
总计     数量          参数：          163,009,536
```

现在，一个好奇的读者可能会注意到一个差异。之前，我们提到了初始化一个 1.24 亿参数的 GPT 模型，那么为什么实际的参数数量是 1.63 亿呢？

The reason is a concept called *weight tying*, which was used in the original GPT-2 architecture. It means that the original GPT-2 architecture reuses the weights from the token embedding layer in its output layer. To understand better, let's take a look at the shapes of the token embedding layer and linear output layer that we initialized on the `model` via the `GPTModel` earlier:

```
print("Token embedding layer shape:", model.tok_emb.weight.shape)
print("Output layer shape:", model.out_head.weight.shape)
```

As we can see from the print outputs, the weight tensors for both these layers have the same shape:

```
Token embedding layer shape: torch.Size([50257, 768])
Output layer shape: torch.Size([50257, 768])
```

The token embedding and output layers are very large due to the number of rows for the 50,257 in the tokenizer's vocabulary. Let's remove the output layer parameter count from the total GPT-2 model count according to the weight tying:

```
total_params_gpt2 = (
    total_params - sum(p.numel()
    for p in model.out_head.parameters())
)
print(f"Number of trainable parameters "
      f"considering weight tying: {total_params_gpt2:,}"
)
```

The output is

```
Number of trainable parameters considering weight tying: 124,412,160
```

As we can see, the model is now only 124 million parameters large, matching the original size of the GPT-2 model.

Weight tying reduces the overall memory footprint and computational complexity of the model. However, in my experience, using separate token embedding and output layers results in better training and model performance; hence, we use separate layers in our `GPTModel` implementation. The same is true for modern LLMs. However, we will revisit and implement the weight tying concept later in chapter 6 when we load the pretrained weights from OpenAI.

Exercise 4.1 Number of parameters in feed forward and attention modules

Calculate and compare the number of parameters that are contained in the feed forward module and those that are contained in the multi-head attention module.

原因是称为权重绑定的概念，这在原始 GPT-2 架构中被使用。这意味着原始 GPT-2 架构在其输出层中重新使用了标记嵌入层的权重。为了更好地理解，让我们看一下我们通过 GPTModel 先前初始化在模型上的标记嵌入层和线性输出层的形状：

```
    打印("标记嵌入层形状：", 模型.tok_emb.weight.shape) 打印("输出层形状：", 模
型.out_head.weight.shape)
```

从打印输出中我们可以看到，这两个层的权重张量具有相同的形状：

```
    令牌嵌入层形状：torch.Size([50257, 768]) 输出层形状：
torch.Size([50257, 768])
```

令牌嵌入和输出层非常大，因为分词器词汇表中有 50,257 行。根据权重绑定，让我们从总 GPT-2 模型计数中减去输出层参数计数。

```
    total_params_gpt2 = (
    总参数数 - sum(p.numel() for p in
model.out_head.parameters()) ) 打印(f"可训练参数数
量 ")
    考虑权重绑定：{total_params_gpt2:,}
```

输出结果

```
数字      of  可训练参数                    考虑到          重量      绑定：124,412,160
```

如您所见，该模型现在只有 1.24 亿个参数，与原始的 GPT-2 模型大小相匹配。

权重绑定减少了模型的总体内存占用和计算复杂度。然而，根据我的经验，使用独立的标记嵌入和输出层可以获得更好的训练和模型性能；因此，我们在 GPTModel 实现中使用了独立的层。这一点对现代LLMs同样适用。然而，当我们从 OpenAI 加载预训练权重时，我们将在第 6 章中重新审视并实现权重绑定概念。

练习 4.1 前馈和注意力模块的参数数量

计算并比较前馈模块中包含的参数数量以及多头注意力模块中包含的参数数量。

Lastly, let's compute the memory requirements of the 163 million parameters in our `GPTModel` object:

```
total_size_bytes = total_params * 4
total_size_mb = total_size_bytes / (1024 * 1024)
print(f"Total size of the model: {total_size_mb:.2f} MB")
```

Calculates the total size in bytes (assuming float32, 4 bytes per parameter)

Converts to megabytes

The result is

```
Total size of the model: 621.83 MB
```

In conclusion, by calculating the memory requirements for the 163 million parameters in our `GPTModel` object and assuming each parameter is a 32-bit float taking up 4 bytes, we find that the total size of the model amounts to 621.83 MB, illustrating the relatively large storage capacity required to accommodate even relatively small LLMs.

Now that we've implemented the `GPTModel` architecture and saw that it outputs numeric tensors of shape `[batch_size, num_tokens, vocab_size]`, let's write the code to convert these output tensors into text.

Exercise 4.2 Initializing larger GPT models

We initialized a 124-million-parameter GPT model, which is known as "GPT-2 small." Without making any code modifications besides updating the configuration file, use the `GPTModel` class to implement GPT-2 medium (using 1,024-dimensional embeddings, 24 transformer blocks, 16 multi-head attention heads), GPT-2 large (1,280-dimensional embeddings, 36 transformer blocks, 20 multi-head attention heads), and GPT-2 XL (1,600-dimensional embeddings, 48 transformer blocks, 25 multi-head attention heads). As a bonus, calculate the total number of parameters in each GPT model.

4.7 Generating text

We will now implement the code that converts the tensor outputs of the GPT model back into text. Before we get started, let's briefly review how a generative model like an LLM generates text one word (or token) at a time.

Figure 4.16 illustrates the step-by-step process by which a GPT model generates text given an input context, such as "Hello, I am." With each iteration, the input context grows, allowing the model to generate coherent and contextually appropriate text. By the sixth iteration, the model has constructed a complete sentence: "Hello, I am a model ready to help." We've seen that our current `GPTModel` implementation outputs tensors with shape `[batch_size, num_token, vocab_size]`. Now the question is: How does a GPT model go from these output tensors to the generated text?

The process by which a GPT model goes from output tensors to generated text involves several steps, as illustrated in figure 4.17. These steps include decoding the

最后，让我们计算我们 1.63 亿参数模型的内存需求

```
GPT 模型对象：

    总大小（字节）= 总参数数 * 4 总大小（MB）= 总大小（字节）/ (1024 * 1024) 打印(f"Total")
                    模型大小：{total_size_mb:.2f} MB"
```

计算总大小（假设为 float32，每个参数 4 字节）

转换为兆字节

结果是

```
总计   size  of  the  模型：   621.83  MB
```

总结来说，通过计算 GPTModel 对象中 1.63 亿个参数的内存需求，并假设每个参数是一个占用 4 字节的 32 位浮点数，我们发现该模型的总大小为 621.83 MB，这说明了即使相对较小的模型也需要较大的存储容量来容纳。

现在我们已经实现了 GPTModel 架构并看到它输出形状为[batch_size, num_tokens, vocab_size]的数值张量，让我们编写代码将这些输出张量转换为文本。

练习 4.2 初始化更大的 GPT 模型

我们初始化了一个 1.24 亿参数的 GPT 模型，被称为“GPT-2 小”。除了更新配置文件外，不进行任何代码修改，使用 GPTModel 类实现 GPT-2 中（使用 1,024 维嵌入，24 个 transformer 块，16 个多头注意力头），GPT-2 大（1,280 维嵌入，36 个 transformer 块，20 个多头注意力头），和 GPT-2 XL（1,600 维嵌入，48 个 transformer 块，25 个多头注意力头）。作为额外奖励，计算每个 GPT 模型的总参数数。

4.7 生成文本

我们将现在实现将 GPT 模型的张量输出转换回文本的代码。在我们开始之前，让我们简要回顾一下像 LLM 这样的生成模型是如何逐词（或标记）生成文本的。

图 4.16 展示了 GPT 模型在给定输入上下文（如“你好，我是。”）的情况下逐步生成文本的过程。每次迭代，输入上下文都会增长，使模型能够生成连贯且上下文适当的文本。到第六次迭代时，模型已经构建了一个完整的句子：“你好，我是一个准备帮助的模型。”我们已经看到，我们当前的 GPTModel 实现输出形状为[batch_size, num_token, vocab_size]的张量。现在的问题是：GPT 模型是如何将这些输出张量转换为生成的文本的？

GPT 模型从输出张量到生成文本的过程涉及多个步骤，如图 4.17 所示。这些步骤包括解码输出张量等。

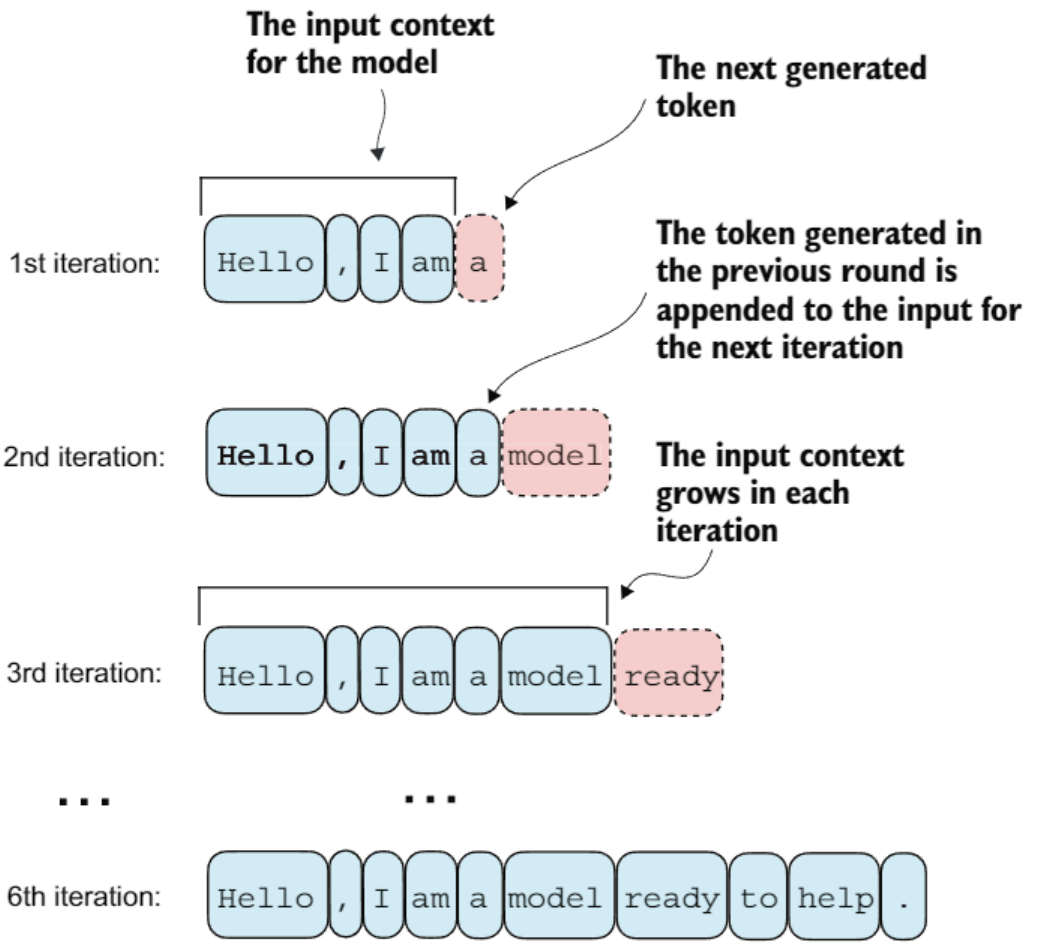

Figure 4.16 The step-by-step process by which an LLM generates text, one token at a time. Starting with an initial input context ("Hello, I am"), the model predicts a subsequent token during each iteration, appending it to the input context for the next round of prediction. As shown, the first iteration adds "a," the second "model," and the third "ready," progressively building the sentence.

output tensors, selecting tokens based on a probability distribution, and converting these tokens into human-readable text.

The next-token generation process detailed in figure 4.17 illustrates a single step where the GPT model generates the next token given its input. In each step, the model outputs a matrix with vectors representing potential next tokens. The vector corresponding to the next token is extracted and converted into a probability distribution via the `softmax` function. Within the vector containing the resulting probability scores, the index of the highest value is located, which translates to the token ID. This token ID is then decoded back into text, producing the next token in the sequence. Finally, this token is appended to the previous inputs, forming a new input sequence for the subsequent iteration. This step-by-step process enables the model to generate text sequentially, building coherent phrases and sentences from the initial input context.

In practice, we repeat this process over many iterations, such as shown in figure 4.16, until we reach a user-specified number of generated tokens. In code, we can implement the token-generation process as shown in the following listing.

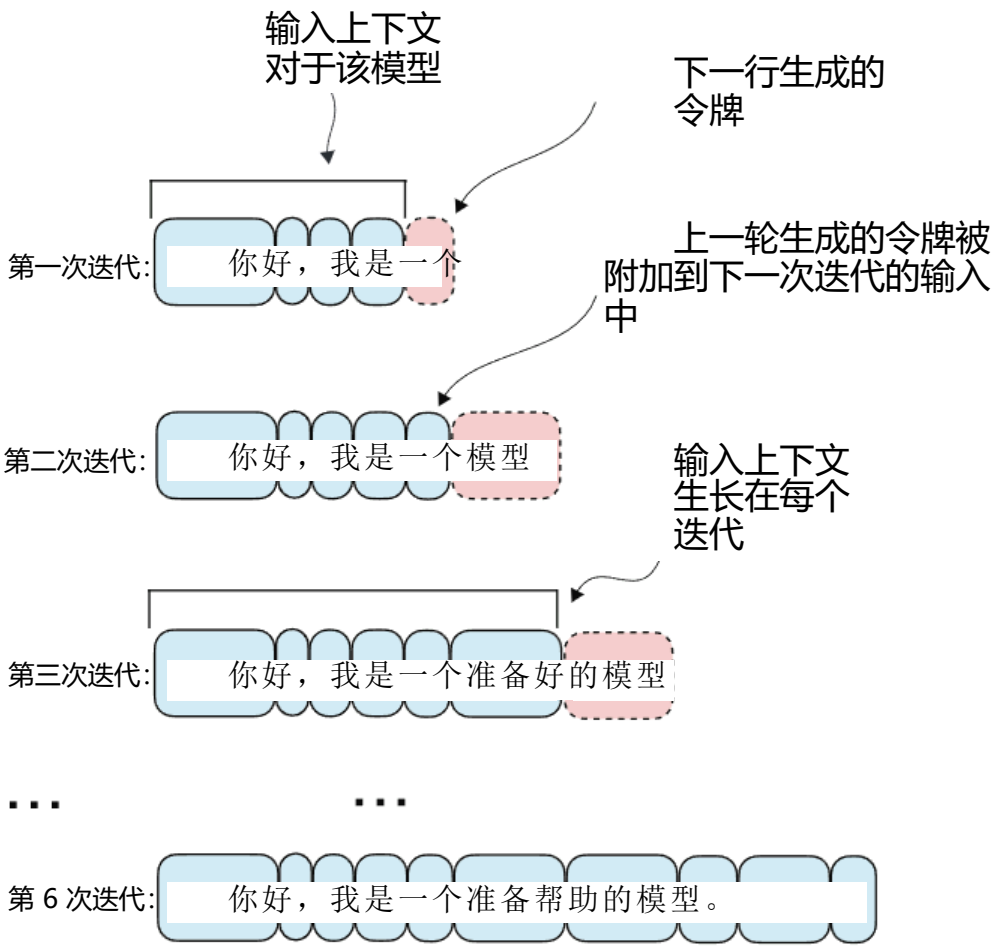

图 4.16 LLM逐个生成文本的步骤过程。从初始输入上下文（“你好，我是”）开始，模型在每次迭代中预测后续的标记，并将其附加到输入上下文中，以便进行下一轮预测。如图所示，第一次迭代添加了“一个”，第二次添加了“模型”，第三次添加了“准备”，逐步构建句子。

这些步骤包括解码输出张量、根据概率分布选择标记，并将这些标记转换为可读文本。

下一行文本的简体中文翻译如下：

图 4.17 详细说明了 GPT 模型在给定输入的情况下生成下一个标记的单一步骤。在每一步中，模型输出一个矩阵，矩阵中的向量代表潜在的下一个标记。提取与下一个标记对应的向量，并通过 softmax 函数将其转换为概率分布。在包含结果概率分数的向量中，找到最高值的索引，这对应于标记 ID。然后将此标记 ID 解码回文本，生成序列中的下一个标记。最后，将此标记附加到之前的输入上，形成新的输入序列，用于后续迭代。这一步步的过程使模型能够顺序生成文本，从初始输入上下文中构建连贯的短语和句子。

实际上，我们重复这个过程很多次，如图 4.16 所示，直到达到用户指定的生成标记数。在代码中，我们可以将标记生成过程实现如下所示。

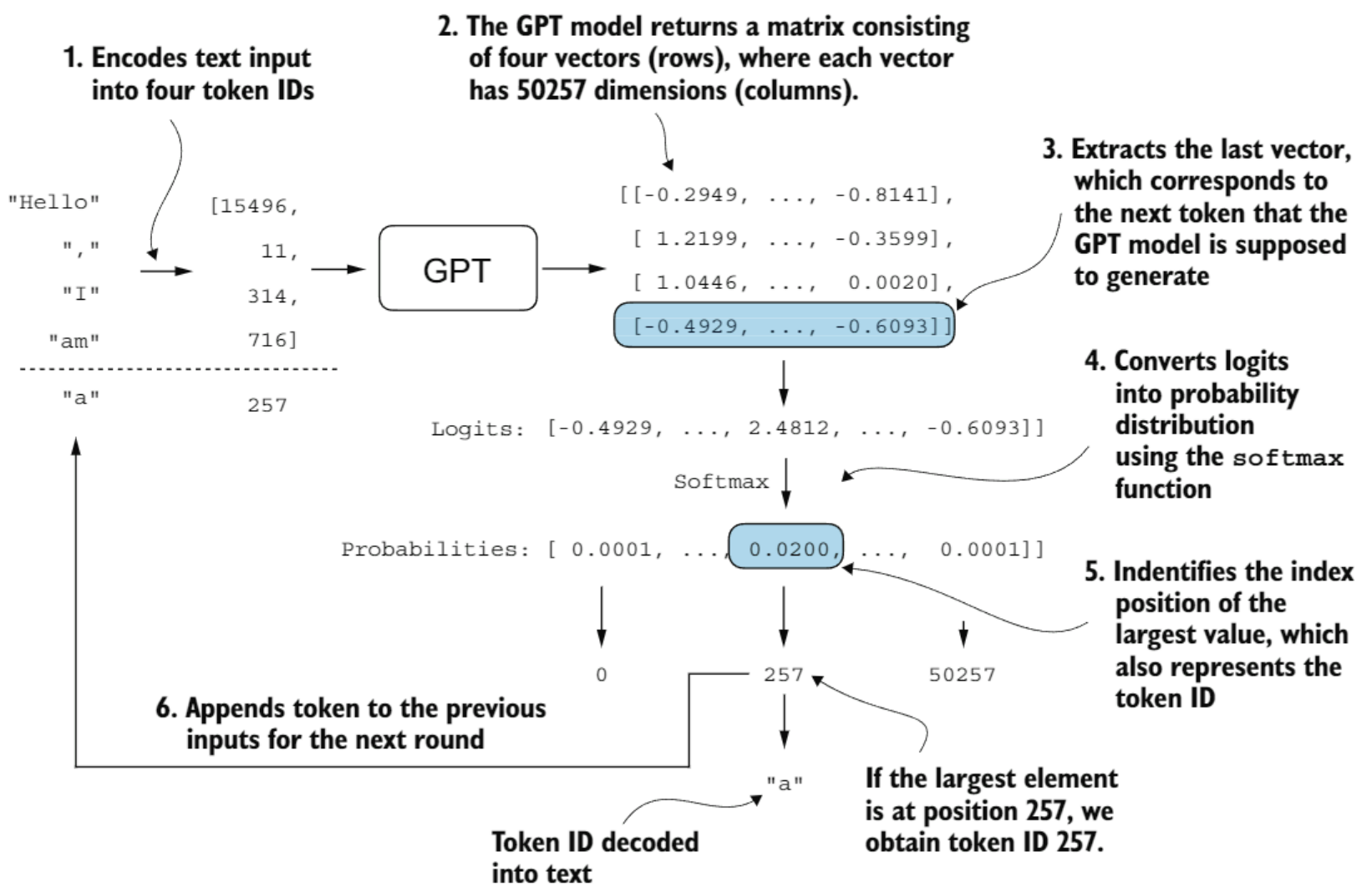

Figure 4.17 The mechanics of text generation in a GPT model by showing a single iteration in the token generation process. The process begins by encoding the input text into token IDs, which are then fed into the GPT model. The outputs of the model are then converted back into text and appended to the original input text.

Listing 4.8 A function for the GPT model to generate text

```
def generate_text_simple(model, idx,
                         max_new_tokens, context_size):
    for _ in range(max_new_tokens):
        idx_cond = idx[:, -context_size:]
        with torch.no_grad():
            logits = model(idx_cond)

        logits = logits[:, -1, :]
        probas = torch.softmax(logits, dim=-1)
        idx_next = torch.argmax(probas, dim=-1, keepdim=True)
        idx = torch.cat((idx, idx_next), dim=1)

    return idx
```

Crops current context if it exceeds the supported context size, e.g., if LLM supports only 5 tokens, and the context size is 10, then only the last 5 tokens are used as context

idx is a (batch, n_tokens) array of indices in the current context.

Focuses only on the last time step, so that (batch, n_token, vocab_size) becomes (batch, vocab_size)

probas has shape (batch, vocab_size).

Appends sampled index to the running sequence, where idx has shape (batch, n_tokens+1)

idx_next has shape (batch, 1).

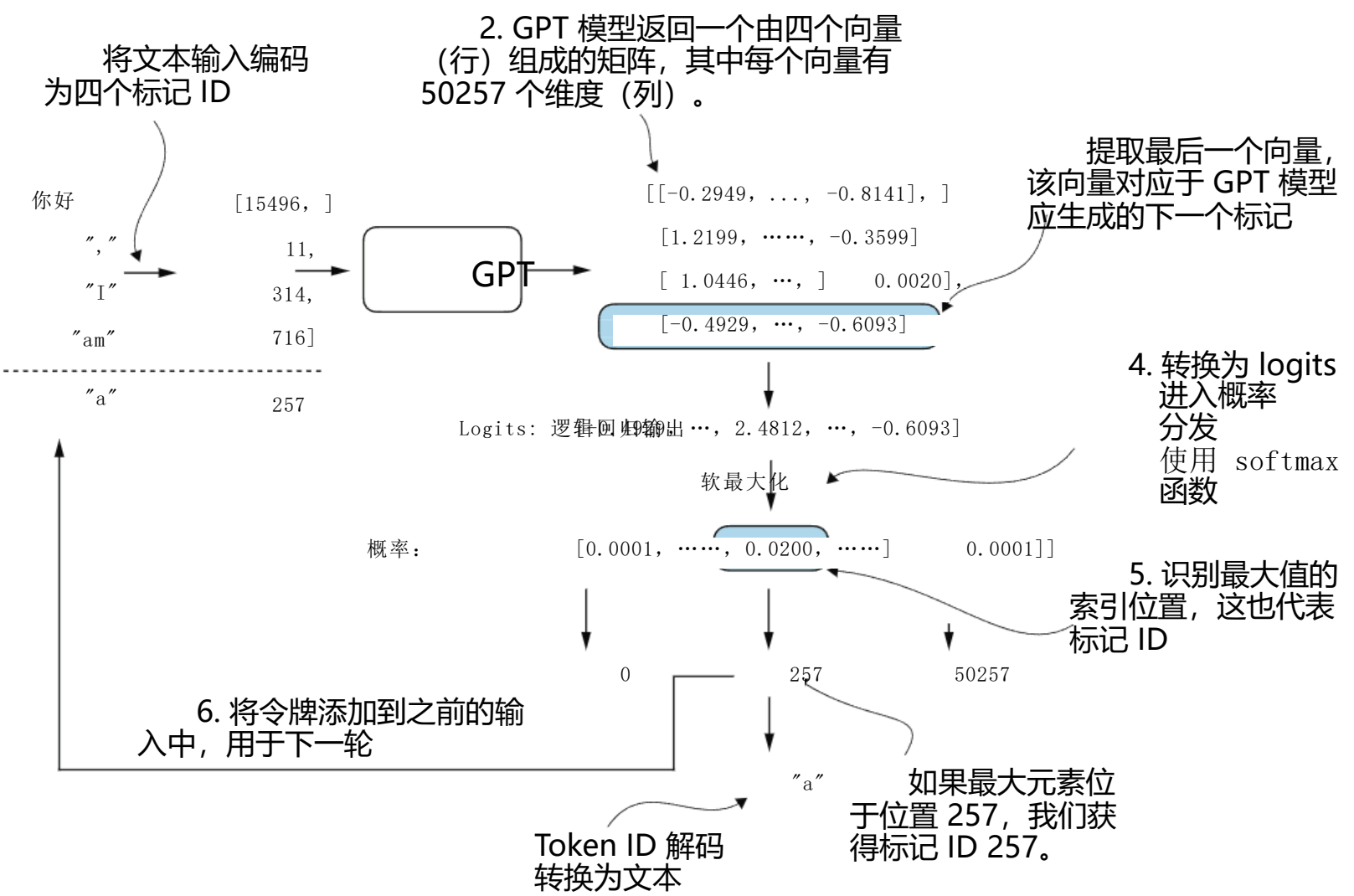

图 4.17 GPT 模型中文本生成机制，通过展示 token 生成过程中的单次迭代。过程从将输入文本编码为 token ID 开始，然后将这些 ID 输入到 GPT 模型中。模型的输出随后被转换回文本，并附加到原始输入文本上。

列表 4.8 GPT 模型生成文本的函数

作物当前上下文，如果超过支持的上下文大小，例如，如果LLM仅支持 5 个标记，并且上下文大小为 10，则仅使用最后 5 个标记作为上下文

idx 是当前上下文中索引的 (batch, n_tokens) 数组。

```
def 生成文本简单(model, idx, max_new_tokens, context_size): 对于 _ 在范围内 range(max_new_tokens):
    idx_cond = idx[:, -context_size:] 使用 torch.no_grad():
        logits = 逻辑模型(idx_cond) 概率 = torch.softmax(logits, dim=-1) 下一个索引 = torch.argmax(probas, dim=-1, keepdim=True) 索引 = torch.cat((索引, 索引_下一个), dim=1)
返回 idx
```

仅关注最后一个时间步，因此（批量，n_token，词汇大小）变为（批量，词汇大小）

概率
形状（批次，）
vocab_size).

idx_next 具有
形状（批次，1）

将采样索引追加到运行序列中，其中 idx 有
形状（批次，n_tokens+1）

This code demonstrates a simple implementation of a generative loop for a language model using PyTorch. It iterates for a specified number of new tokens to be generated, crops the current context to fit the model's maximum context size, computes predictions, and then selects the next token based on the highest probability prediction.

To code the `generate_text_simple` function, we use a `softmax` function to convert the logits into a probability distribution from which we identify the position with the highest value via `torch.argmax`. The `softmax` function is monotonic, meaning it preserves the order of its inputs when transformed into outputs. So, in practice, the softmax step is redundant since the position with the highest score in the softmax output tensor is the same position in the logit tensor. In other words, we could apply the `torch.argmax` function to the logits tensor directly and get identical results. However, I provide the code for the conversion to illustrate the full process of transforming logits to probabilities, which can add additional intuition so that the model generates the most likely next token, which is known as *greedy decoding*.

When we implement the GPT training code in the next chapter, we will use additional sampling techniques to modify the softmax outputs such that the model doesn't always select the most likely token. This introduces variability and creativity in the generated text.

This process of generating one token ID at a time and appending it to the context using the `generate_text_simple` function is further illustrated in figure 4.18. (The token ID generation process for each iteration is detailed in figure 4.17.) We generate the token IDs in an iterative fashion. For instance, in iteration 1, the model is provided with the tokens corresponding to "Hello, I am," predicts the next token (with ID 257, which is "a"), and appends it to the input. This process is repeated until the model produces the complete sentence "Hello, I am a model ready to help" after six iterations.

Let's now try out the `generate_text_simple` function with the `"Hello, I am"` context as model input. First, we encode the input context into token IDs:

```
start_context = "Hello, I am"
encoded = tokenizer.encode(start_context)
print("encoded:", encoded)
encoded_tensor = torch.tensor(encoded).unsqueeze(0)
print("encoded_tensor.shape:", encoded_tensor.shape)
```

Adds batch dimension

The encoded IDs are

```
encoded: [15496, 11, 314, 716]
encoded_tensor.shape: torch.Size([1, 4])
```

这段代码展示了使用 PyTorch 实现的一个简单的语言模型生成循环的示例。它迭代指定数量的新标记以生成，裁剪当前上下文以适应模型的最大上下文大小，计算预测，然后根据最高概率预测选择下一个标记。

将 generate_text_simple 函数编码时，我们使用 softmax 函数将 logits 转换为概率分布，然后通过 torch.argmax 识别出值最高的位置。softmax 函数是单调的，这意味着它在转换为输出时保留了输入的顺序。因此，在实践中，softmax 步骤是多余的，因为 softmax 输出张量中得分最高的位置与 logit 张量中相同位置。换句话说，我们可以直接对 logits 张量应用 torch.argmax 函数并得到相同的结果。然而，我提供了转换的代码，以说明将 logits 转换为概率的完整过程，这可以增加额外的直观性，以便模型生成最可能的下一个标记，这被称为贪婪解码。

当我们下一章实现 GPT 训练代码时，我们将使用额外的采样技术来修改 softmax 输出，使得模型不会总是选择最可能的标记。这为生成的文本引入了可变性和创造性。

这个过程通过使用 generate_text_simple 函数一次生成一个 token ID 并将其附加到上下文中进一步在图 4.18 中说明。（每个迭代的 token ID 生成过程在图 4.17 中详细说明。）我们以迭代方式生成 token IDs。例如，在迭代 1 中，模型被提供与“Hello, I am,”对应的 token，预测下一个 token（ID 为 257，即“a”），并将其附加到输入中。这个过程重复进行，直到模型在六次迭代后生成完整的句子“Hello, I am a model ready to help”。

现在让我们尝试使用"Hello, I am"作为模型输入来运行 generate_text_simple 函数。首先，我们将输入上下文编码为标记 ID：

```
    开始上下文 = "你好，我是" 编码 = 分词器.encode(开始上下文)
打印("编码：", 编码) 编码张量 = torch.tensor(编
码).unsqueeze(0) 打印("编码张量.shape：")
                                                                    添加批
                                                                    量维度
                                                encoded_tensor.shape) -> encoded_tensor.shape)
```

编码的 ID 是

```
    编码：[15496,   11,  314,
编码：编码张量
形状：
torch.Size([1, 4])
```

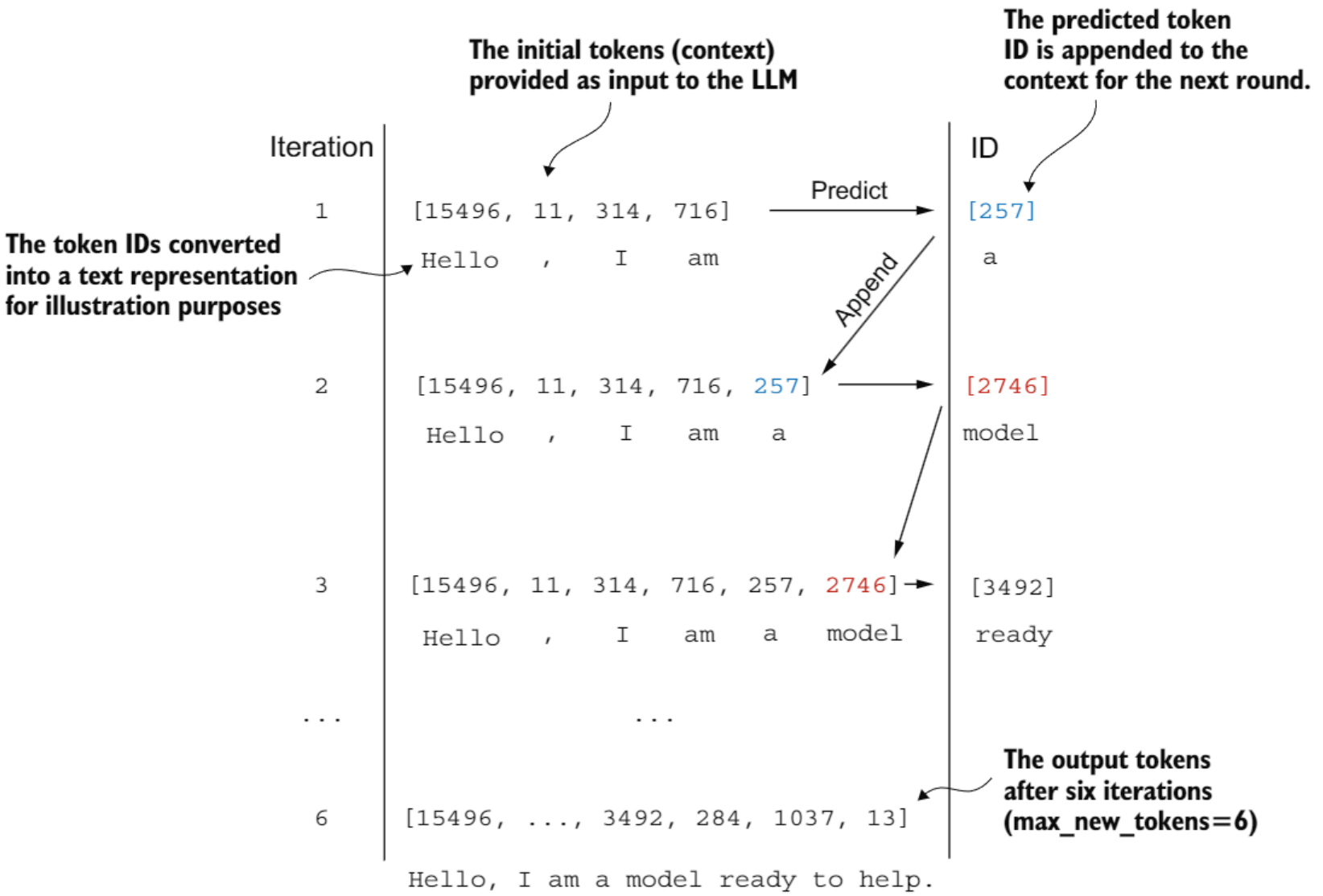

Figure 4.18 The six iterations of a token prediction cycle, where the model takes a sequence of initial token IDs as input, predicts the next token, and appends this token to the input sequence for the next iteration. (The token IDs are also translated into their corresponding text for better understanding.)

Next, we put the model into `.eval()` mode. This disables random components like dropout, which are only used during training, and use the `generate_text_simple` function on the encoded input tensor:

```
model.eval()
out = generate_text_simple(
    model=model,
    idx=encoded_tensor,
    max_new_tokens=6,
    context_size=GPT_CONFIG_124M["context_length"]
)
print("Output:", out)
print("Output length:", len(out[0]))
```

Disables dropout since we are not training the model

The resulting output token IDs are

```
Output: tensor([[15496,    11,   314,   716, 27018, 24086, 47843,
30961, 42348,  7267]])
Output length: 10
```

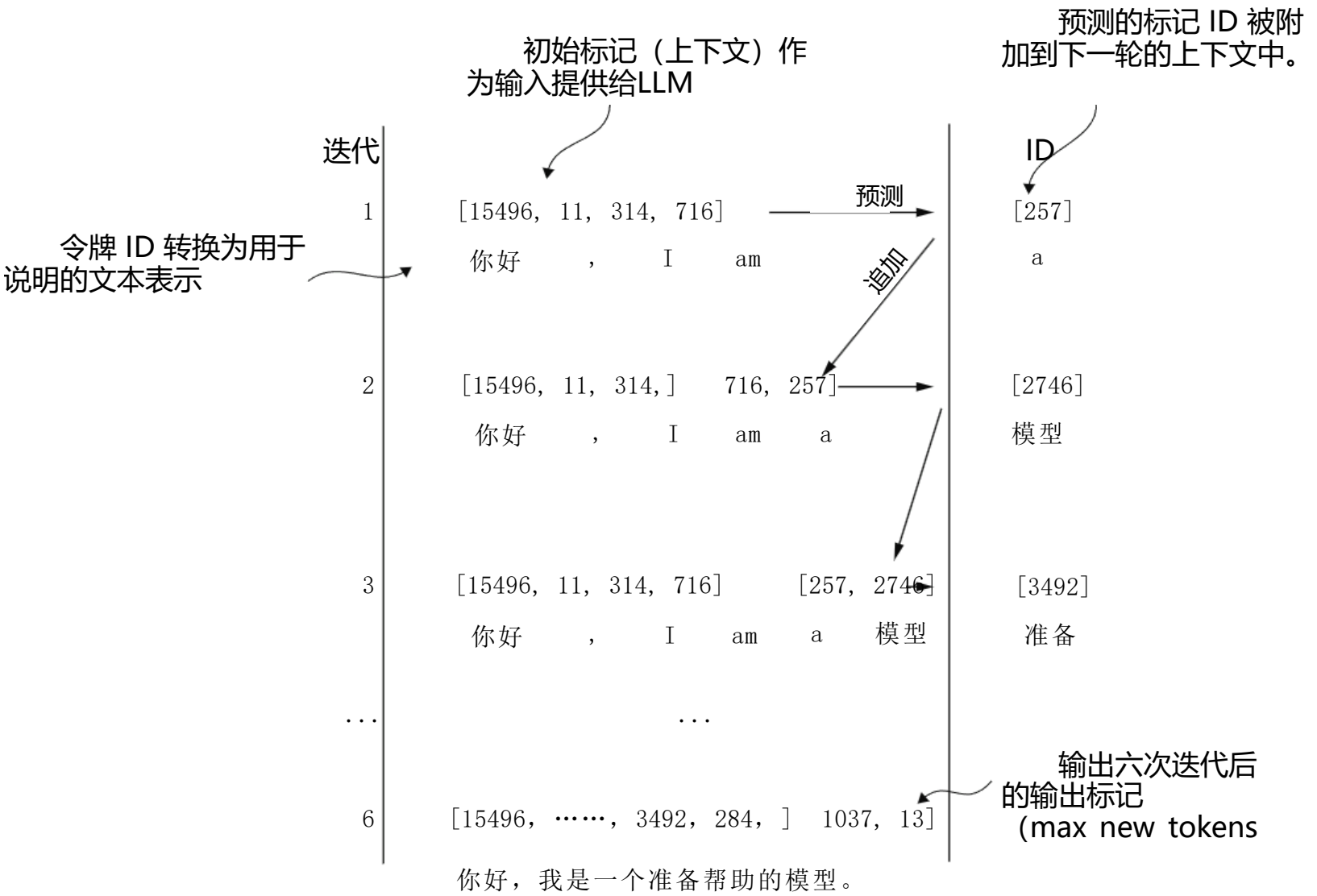

图 4.18 模型进行六次标记预测周期的迭代，其中模型以一系列初始标记 ID 作为输入，预测下一个标记，并将此标记附加到输入序列以进行下一次迭代。（标记 ID 也转换为相应的文本以更好地理解。）

接下来，我们将模型置于.eval()模式。这会禁用仅在训练期间使用的随机组件，如 dropout，并在编码后的输入张量上使用 generate_text_simple 函数：

```
model.eval()
输出 = generate_text_simple(
    model=model,
    idx=encoded_tensor
max_new_tokens=6, 上下文大小=GPT_CONFIG_124M["context_length"] ) 打印("输出:", out) 打印("输出长度:", len(out[0]))
```

禁用 dropout，因为我们没有在训练模型

结果输出标记 ID 是

```
输出: tensor([[15496, 11, 314, 716, 27018, 24086, 47843, 30961, 42348, 7267]]) 输出长度: 10
```

Using the `.decode` method of the tokenizer, we can convert the IDs back into text:

```
decoded_text = tokenizer.decode(out.squeeze(0).tolist())
print(decoded_text)
```

The model output in text format is

```
Hello, I am Featureiman Byeswickattribute argue
```

As we can see, the model generated gibberish, which is not at all like the coherent text `Hello, I am a model ready to help`. What happened? The reason the model is unable to produce coherent text is that we haven't trained it yet. So far, we have only implemented the GPT architecture and initialized a GPT model instance with initial random weights. Model training is a large topic in itself, and we will tackle it in the next chapter.

Exercise 4.3 Using separate dropout parameters

At the beginning of this chapter, we defined a global `drop_rate` setting in the `GPT_CONFIG_124M` dictionary to set the dropout rate in various places throughout the `GPTModel` architecture. Change the code to specify a separate dropout value for the various dropout layers throughout the model architecture. (Hint: there are three distinct places where we used dropout layers: the embedding layer, shortcut layer, and multi-head attention module.)

Summary

- Layer normalization stabilizes training by ensuring that each layer's outputs have a consistent mean and variance.
- Shortcut connections are connections that skip one or more layers by feeding the output of one layer directly to a deeper layer, which helps mitigate the vanishing gradient problem when training deep neural networks, such as LLMs.
- Transformer blocks are a core structural component of GPT models, combining masked multi-head attention modules with fully connected feed forward networks that use the GELU activation function.
- GPT models are LLMs with many repeated transformer blocks that have millions to billions of parameters.
- GPT models come in various sizes, for example, 124, 345, 762, and 1,542 million parameters, which we can implement with the same `GPTModel` Python class.
- The text-generation capability of a GPT-like LLM involves decoding output tensors into human-readable text by sequentially predicting one token at a time based on a given input context.
- Without training, a GPT model generates incoherent text, which underscores the importance of model training for coherent text generation.

使用分词器的.decode 方法，我们可以将 ID 转换回文本：

```
    decoded_text = tokenizer.decode(out.squeeze(0).tolist()) 打印
(decoded_text)
```

模型输出为文本格式

```
    你好，   I am  Featureiman   比斯威克属性论证
```

我们可以看到，该模型生成了乱码，与连贯的文本完全不同
你好，我是一个准备帮助的模型。发生了什么？模型无法生成连贯文本的原因是我们还没有对其进行训练。到目前为止，我们只实现了 GPT 架构并初始化了一个带有初始随机权重的 GPT 模型实例。

模型训练本身就是一个大主题，我们将在下一章中探讨它。

练习 4.3 使用单独的 dropout 参数

At the beginning of this chapter, we defined a global drop_rate setting in the GPT_ CONFIG_124M dictionary to set the dropout rate in various places throughout the GPTModel architecture. Change the code to specify a separate dropout value for the various dropout layers throughout the model architecture. (Hint: there are three distinct places where we used dropout layers: the embedding layer, shortcut layer, and multi-head attention module.)

摘要

- 层归一化通过确保每一层的输出具有一致的均值和方差来稳定训练。
- 快捷连接是通过将一层或多层的输出直接馈送到更深的一层，从而跳过一层或多层连接，这有助于缓解训练深度神经网络时梯度消失的问题，例如LLMs。
- Transformer 块是 GPT 模型的核心结构组件，结合了带掩码的多头注意力模块和采用 GELU 激活函数的全连接前馈网络。
- GPT 模型包含许多重复的 Transformer 块，这些块具有数百万到数十亿个参数。
- GPT 模型有多种大小，例如，124、345、762 和 15.42 亿个参数，我们可以使用相同的 GPTModel Python 类来实现。
- GPT-like LLM 的文本生成能力涉及通过按顺序预测一个标记一次，根据给定的输入上下文将输出张量解码成人类可读的文本。
- 没有训练，GPT 模型生成的文本不连贯，这突出了模型训练对于生成连贯文本的重要性。