3 Coding attention mechanisms

This chapter covers

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

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

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

编码注意力机制

本章涵盖

- 神经网络中使用注意力机制的原因
- 一个基本的自注意力框架，进阶到增强的自注意力机制
- 因果注意力模块，允许LLMs逐个生成标记
- 随机遮挡选定的注意力权重以减少过拟合
- 堆叠多个因果注意力模块到多头注意力模块

此时，您已了解如何通过将文本拆分为单个单词和子词标记来准备训练输入文本LLMs，这些标记可以编码成向量表示，即嵌入，用于LLM。

现在，我们将探讨LLM架构本身的组成部分，即注意力机制，如图 3.1 所示。我们将主要关注注意力机制本身，并在机制层面进行探讨。然后，我们将编写剩余的部分。

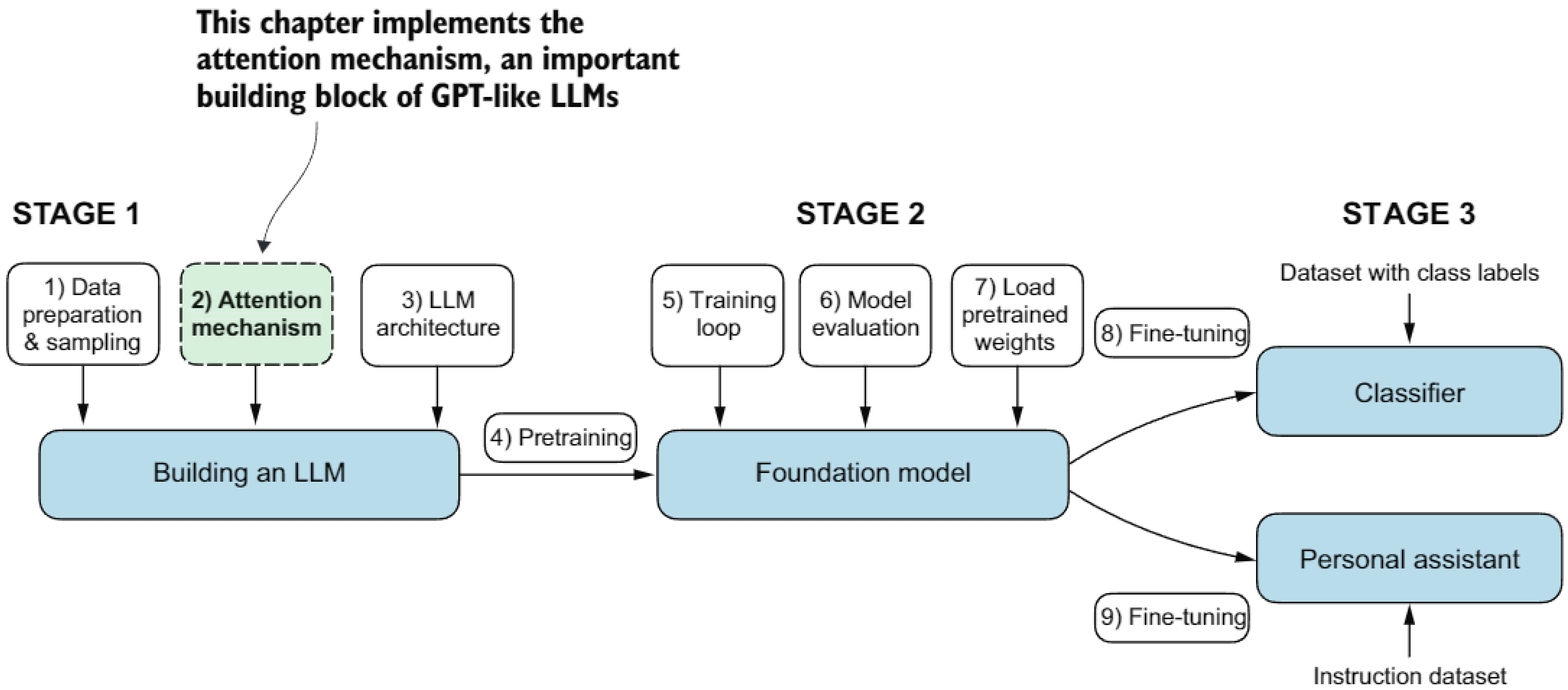

Figure 3.1 The three main stages of coding an LLM. This chapter focuses on step 2 of stage 1: implementing attention mechanisms, which are an integral part of the LLM architecture.

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

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

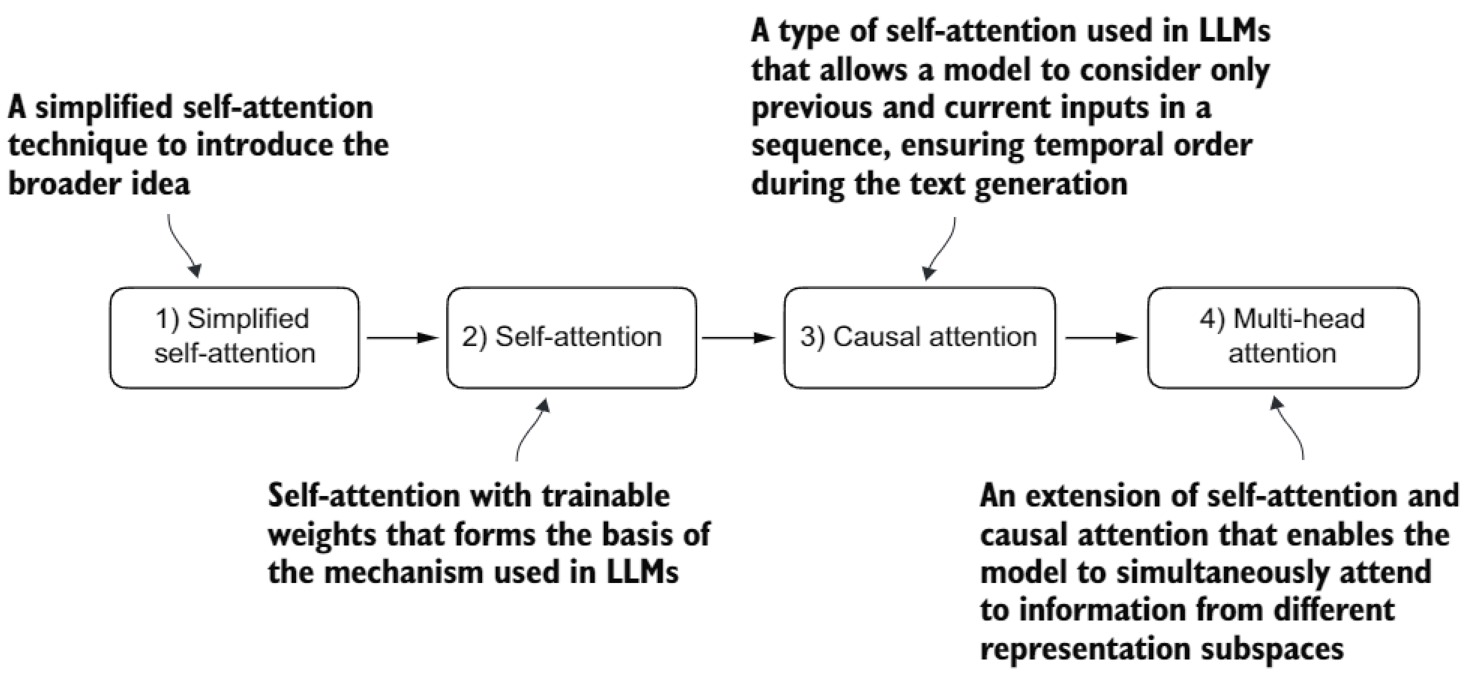

Figure 3.2 The figure depicts different attention mechanisms we will code in this chapter, starting with a simplified version of self-attention before adding the trainable weights. The causal attention mechanism adds a mask to self-attention that allows the LLM to generate one word at a time. Finally,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

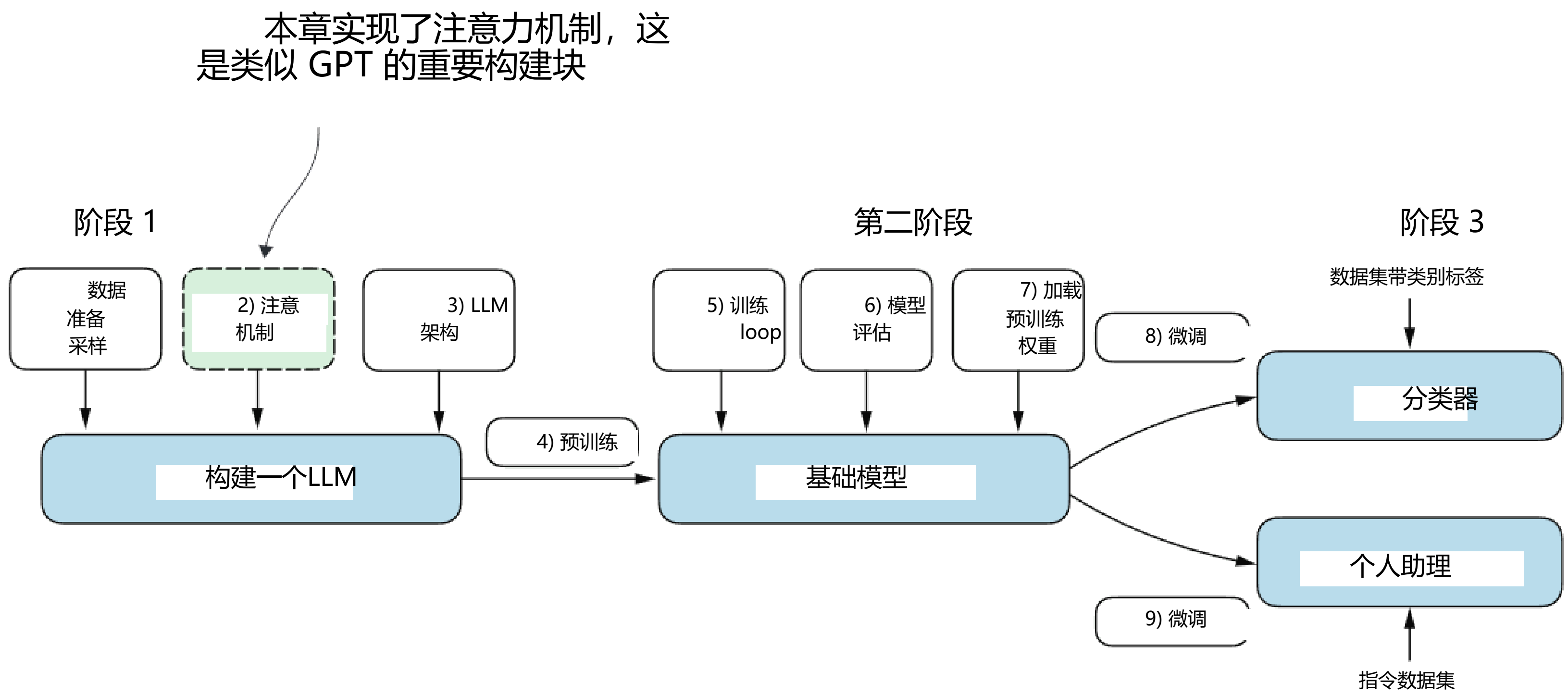

图 3.1 LLM 编码的三个主要阶段。本章重点介绍第一阶段第二步：实现注意力机制，这是 LLM 架构的组成部分。

部分围绕自注意力机制的LLM，以观察其作用并创建生成文本的模型。

我们将实现如图 3.2 所示的四种不同的注意力机制变体。这些不同的注意力变体相互构建，目标是

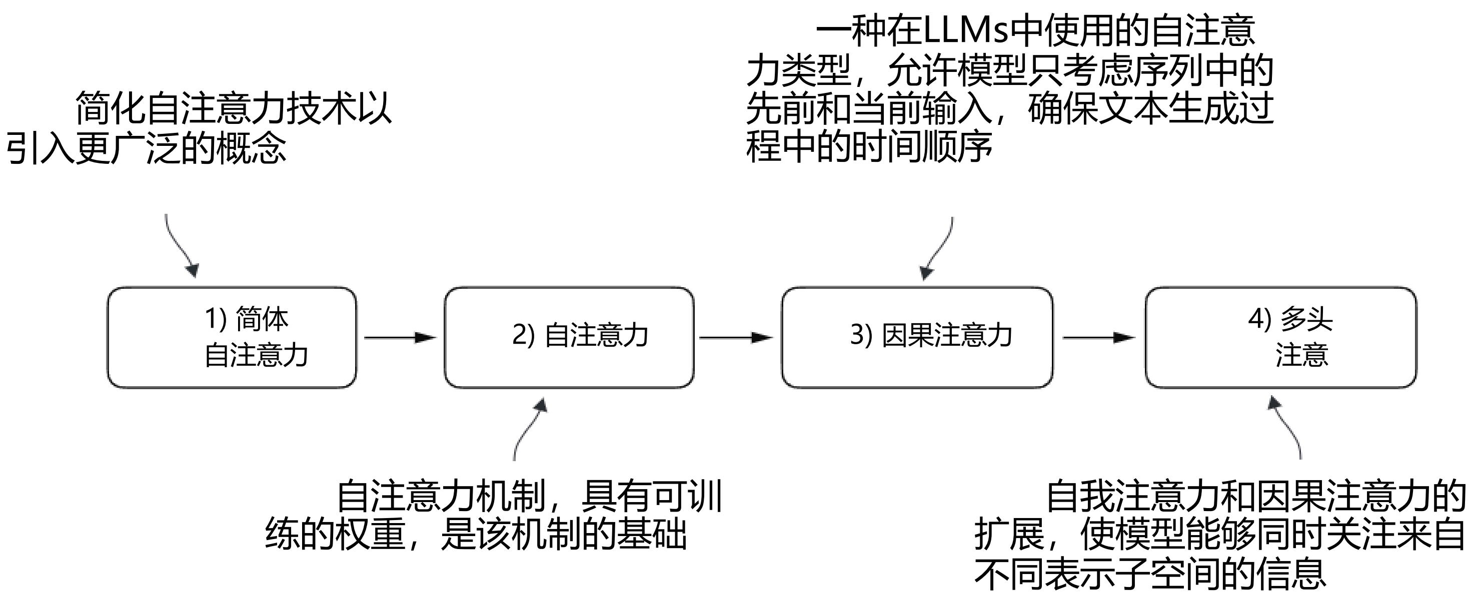

图 3.2 该图展示了本章中将编码的不同注意力机制，从简化的自注意力版本开始，然后添加可训练的权重。因果注意力机制为自注意力添加了一个掩码，允许LLM逐个生成单词。最后，多头注意力将注意力机制组织成多个头，使模型能够并行捕获输入数据的各个方面。

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

3.1 The problem with modeling long sequences

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

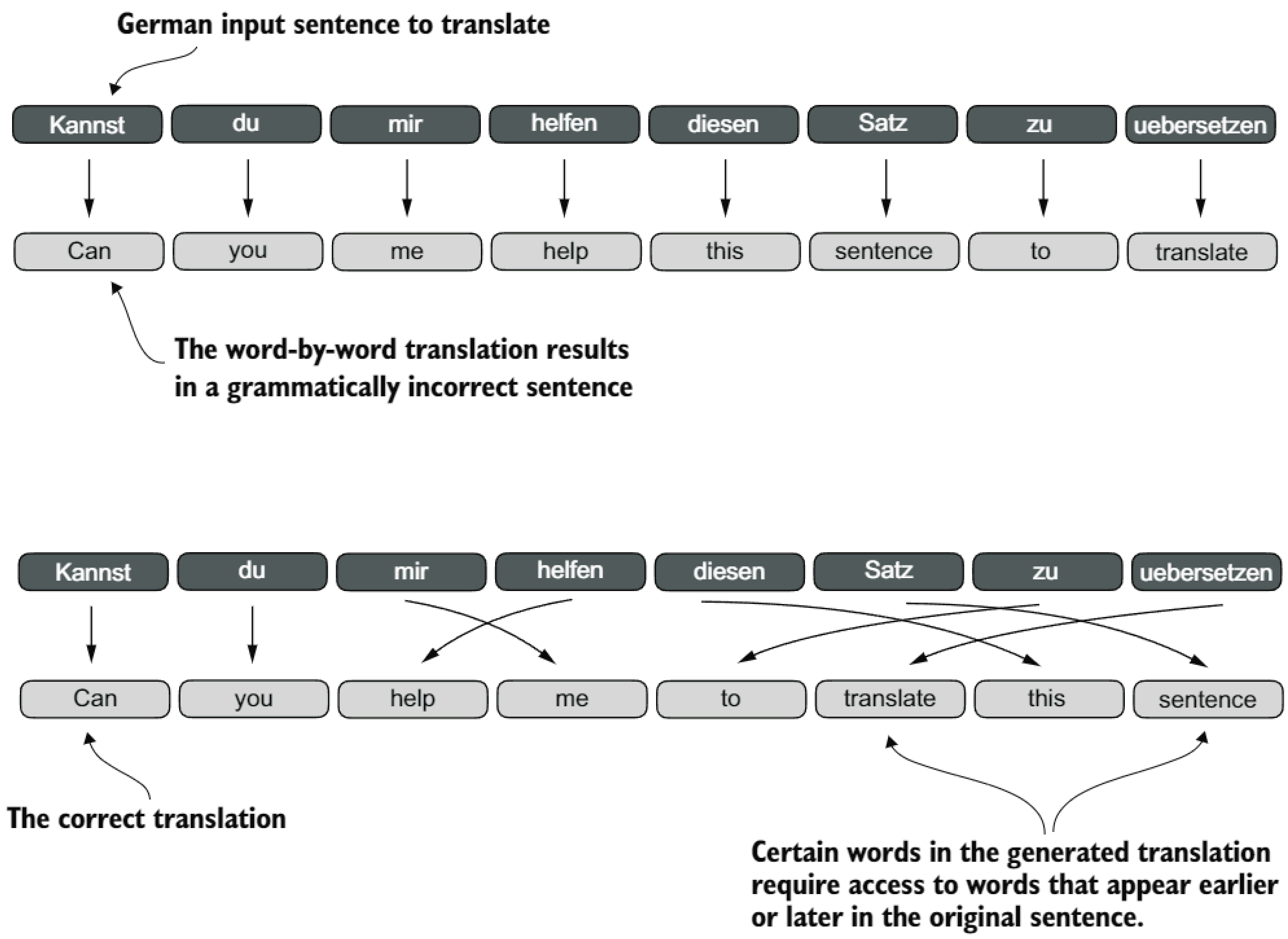

Figure 3.3 When translating text from one language to another, such as German to English, it's not possible to merely translate word by word. Instead, the translation process requires contextual understanding and grammatical alignment.

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

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

达到一个紧凑且高效的多头注意力实现，然后我们可以将其插入到下一章我们将编写的LLM架构中。

3.1 该问题在于建模长序列

在深入探讨LLMs核心的自注意力机制之前，让我们先考虑那些不包括注意力机制的预-LLM架构存在的问题。假设我们想要开发一个语言翻译模型，将一种语言的文本翻译成另一种语言。如图 3.3 所示，由于源语言和目标语言的语法结构，我们无法简单地逐词翻译文本。

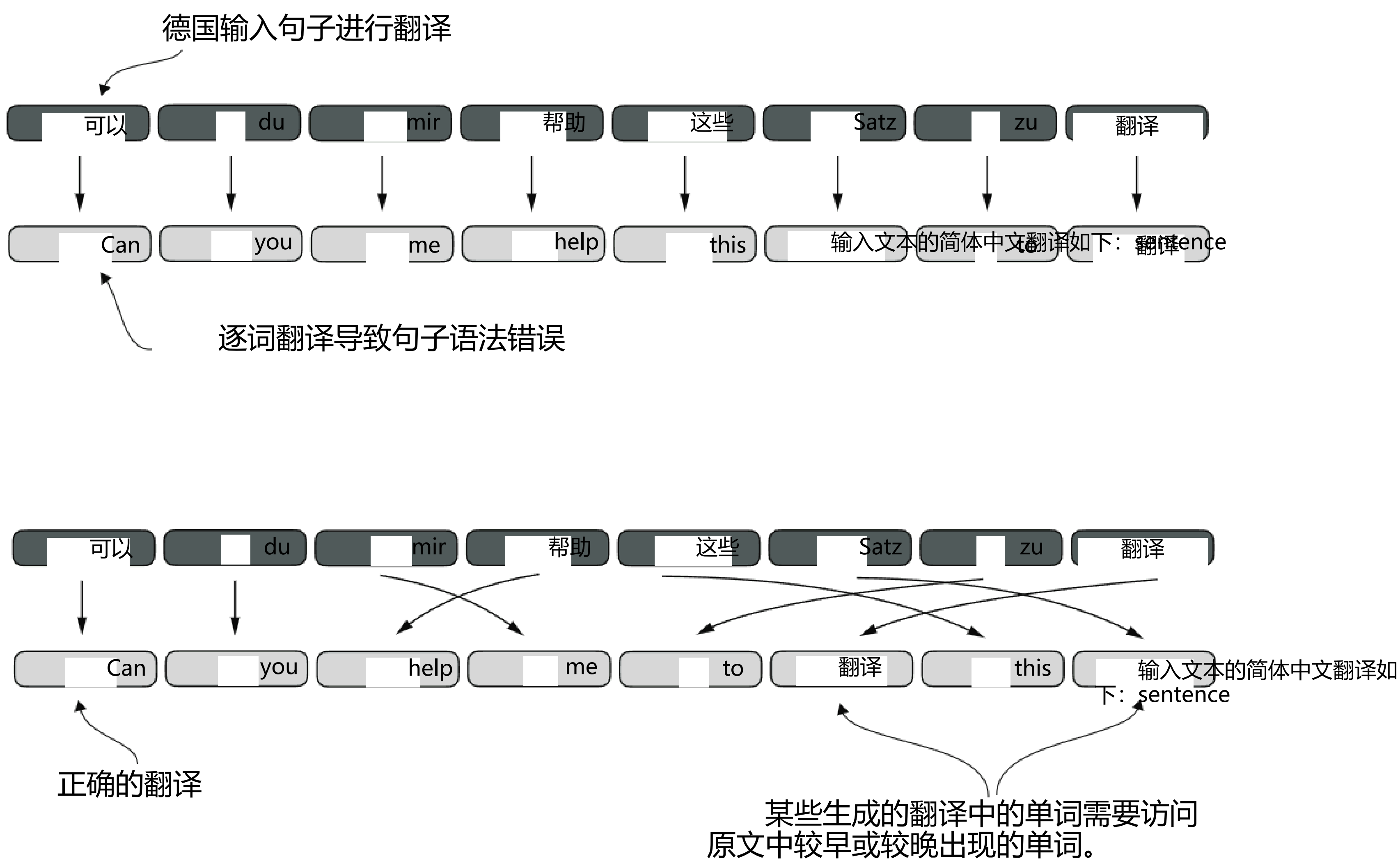

图 3.3 在将文本从一种语言翻译成另一种语言时，例如从德语翻译成英语，不能仅仅逐字翻译。相反，翻译过程需要语境理解和语法对齐。

为了解决这个问题，通常使用具有两个子模块的深度神经网络，即编码器和解码器。编码器的任务是首先读取并处理整个文本，然后解码器生成翻译后的文本。

在变压器出现之前，循环神经网络（RNN）是语言翻译中最受欢迎的编码器-解码器架构。RNN 是一种神经网络，其中前一步的输出被作为当前输入。

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

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

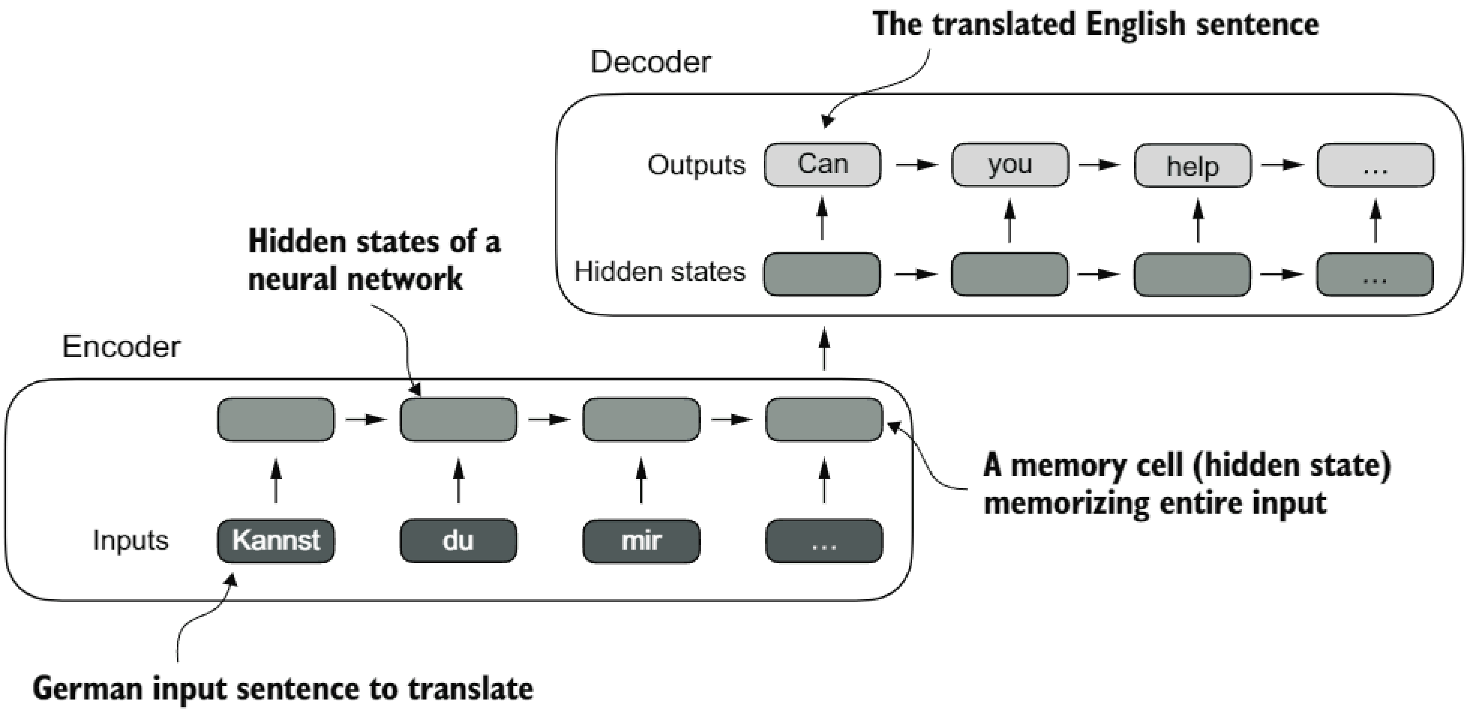

Figure 3.4 Before the advent of transformer models, encoder–decoder RNNs were a popular choice for machine translation. The encoder takes a sequence of tokens from the source language as input, where a hidden state (an intermediate neural network layer) of the encoder encodes a compressed representation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

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

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

循环神经网络是一种神经网络，其中前一步的输出被作为当前步骤的输入，这使得它们非常适合像文本这样的顺序数据。如果你对 RNN 不熟悉，不用担心——你不需要了解 RNN 的详细工作原理来跟随这次讨论；我们在这里更关注编码器-解码器设置的一般概念。

在一个编码器-解码器循环神经网络中，输入文本被送入编码器，编码器按顺序处理它。编码器在每一步更新其隐藏状态（隐藏层的内部值），试图在最终的隐藏状态中捕捉输入句子的整个含义，如图 3.4 所示。然后解码器使用这个最终的隐藏状态开始生成翻译句子，一次一个单词。它也在每一步更新其隐藏状态，这个状态应该携带进行下一个单词预测所必需的上下文。

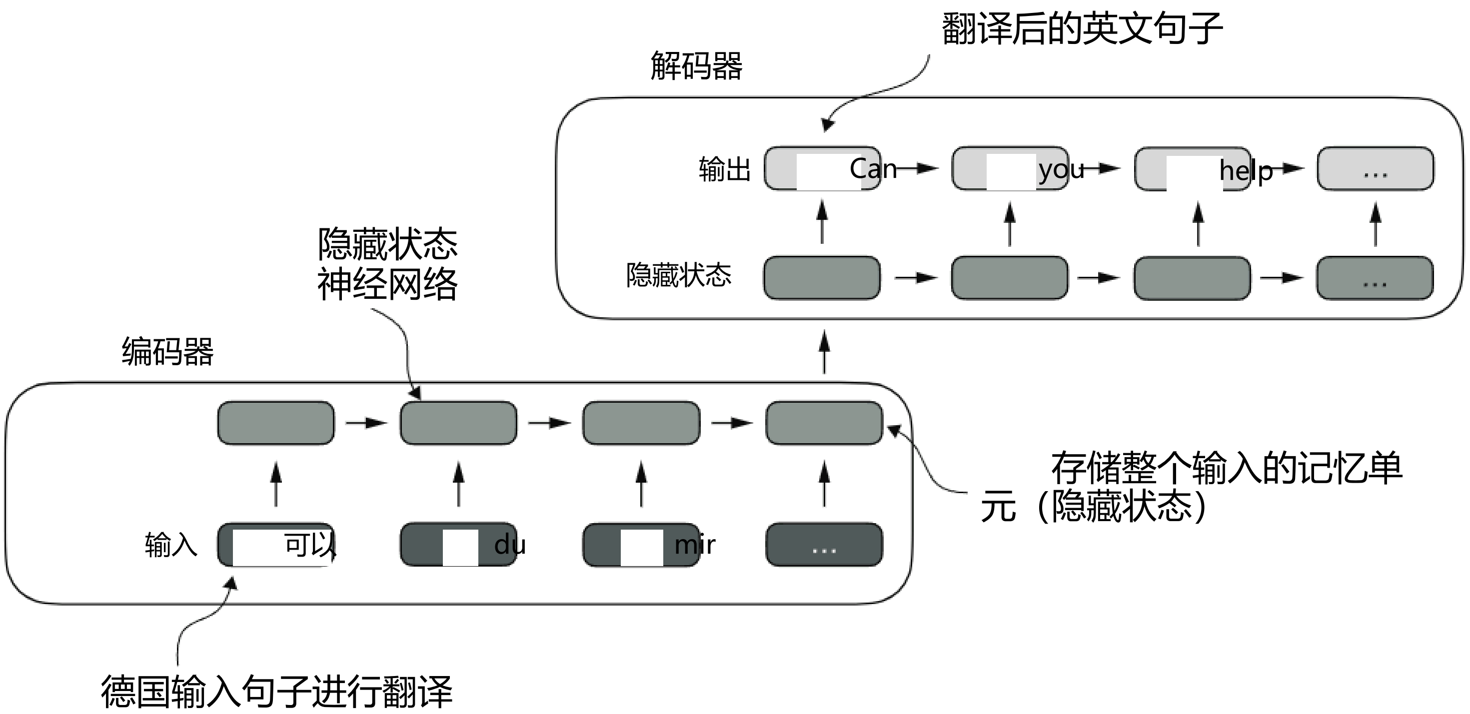

图 3.4 在 transformer 模型出现之前，编码器-解码器循环神经网络是机器翻译的一个流行选择。编码器接收源语言的一序列标记作为输入，其中编码器的隐藏状态（一个中间神经网络层）对整个输入序列进行压缩表示。然后，解码器使用其当前的隐藏状态逐个标记地开始翻译。

虽然我们不需要了解这些编码器-解码器 RNN 的内部工作原理，但关键思想是编码器部分将整个输入文本处理成一个隐藏状态（记忆单元）。然后解码器接收这个隐藏状态以生成输出。你可以将这个隐藏状态视为一个嵌入向量，这是我们第 2 章讨论的概念。

编码器-解码器 RNN 的大局限在于 RNN 在解码阶段无法直接访问编码器中的早期隐藏状态。因此，它完全依赖于当前隐藏状态，该状态封装了所有相关信息。这可能导致上下文丢失，尤其是在复杂句子中，其中依赖关系可能跨越很长的距离。

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

3.2 Capturing data dependencies with attention mechanisms

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

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

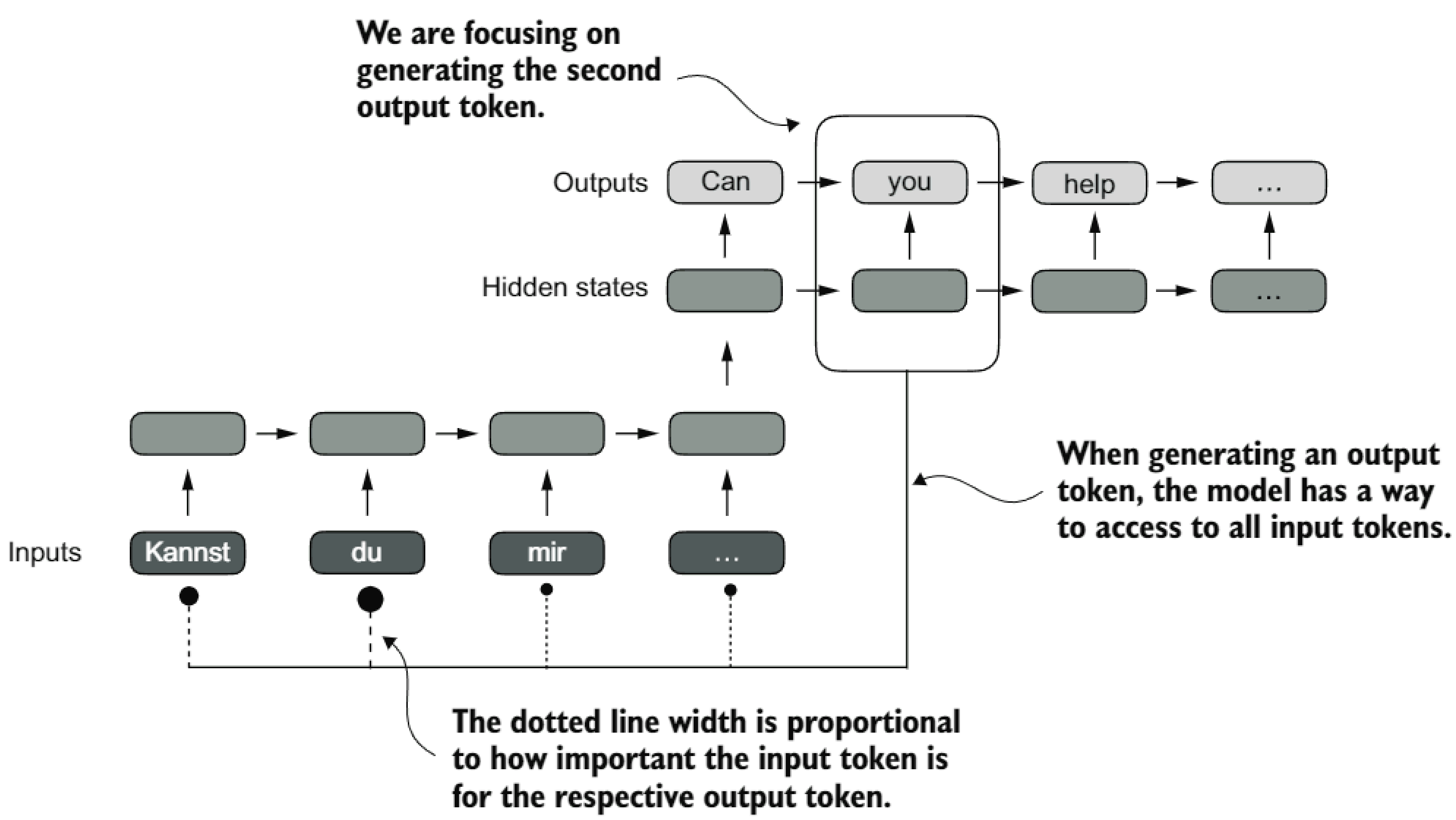

Figure 3.5 Using an attention mechanism, the text-generating decoder part of the network can access all input tokens selectively. This means that some input tokens are more important than others for generating a given output token. The importance is determined by the attention weights, which we will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

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

幸运的是，构建一个LLM并不需要理解 RNN。只需记住，编码器-解码器 RNN 存在一个缺陷，这促使了注意力机制的设计。

3.2 捕获数据依赖关系，使用注意力机制

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

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

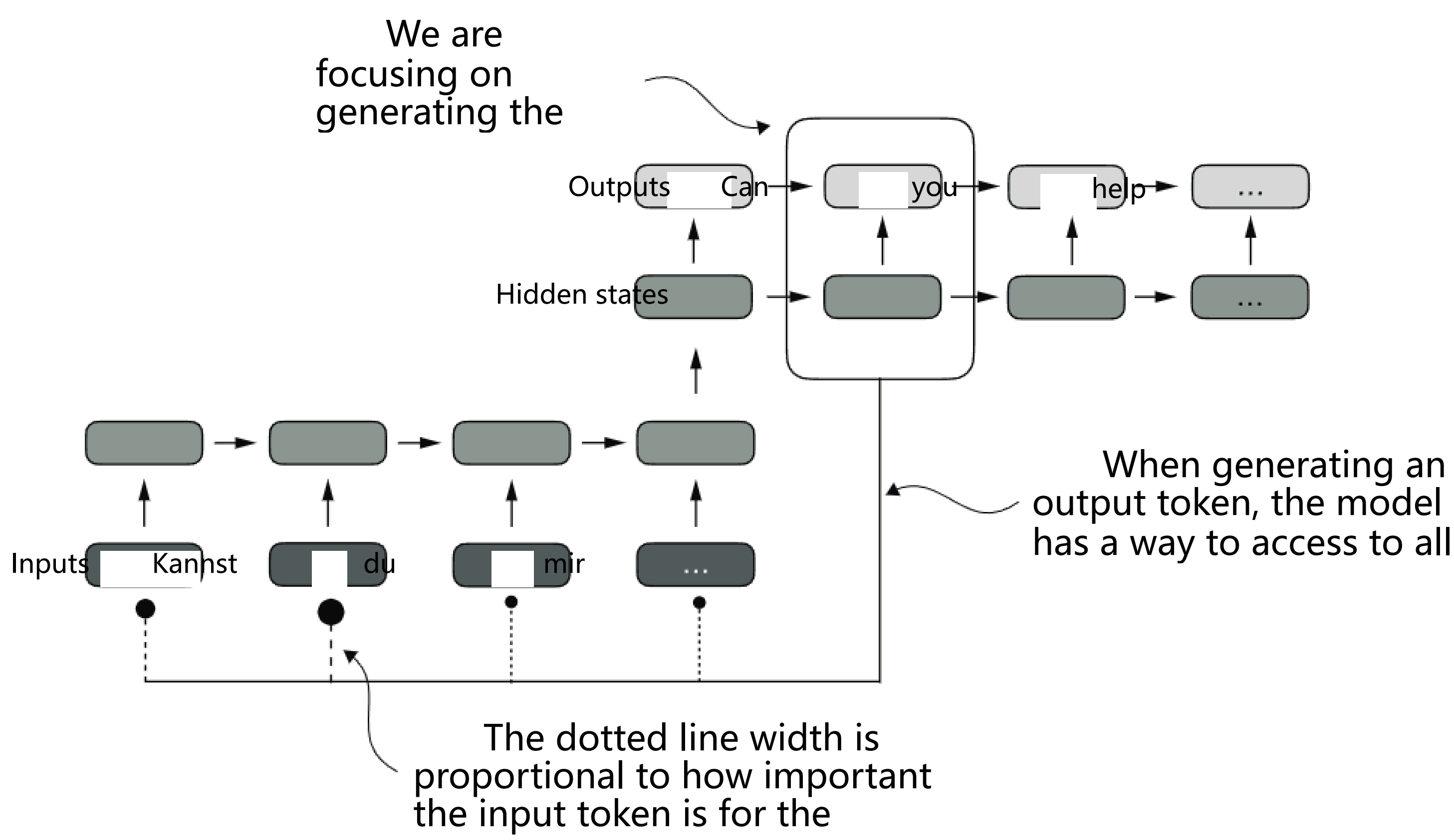

Figure 3.5 Using an attention mechanism, the text-generating decoder part of the network can access all input tokens selectively. This means that some input tokens are more important than others for generating a given output token. The importance is determined by the attention weights, which we will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

Interestingly, only three years later, researchers found that RNN architectures are not required for building deep neural networks for natural language processing and proposed the original *transformer* architecture (discussed in chapter 1) including a self-attention mechanism inspired by the Bahdanau attention mechanism.

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

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

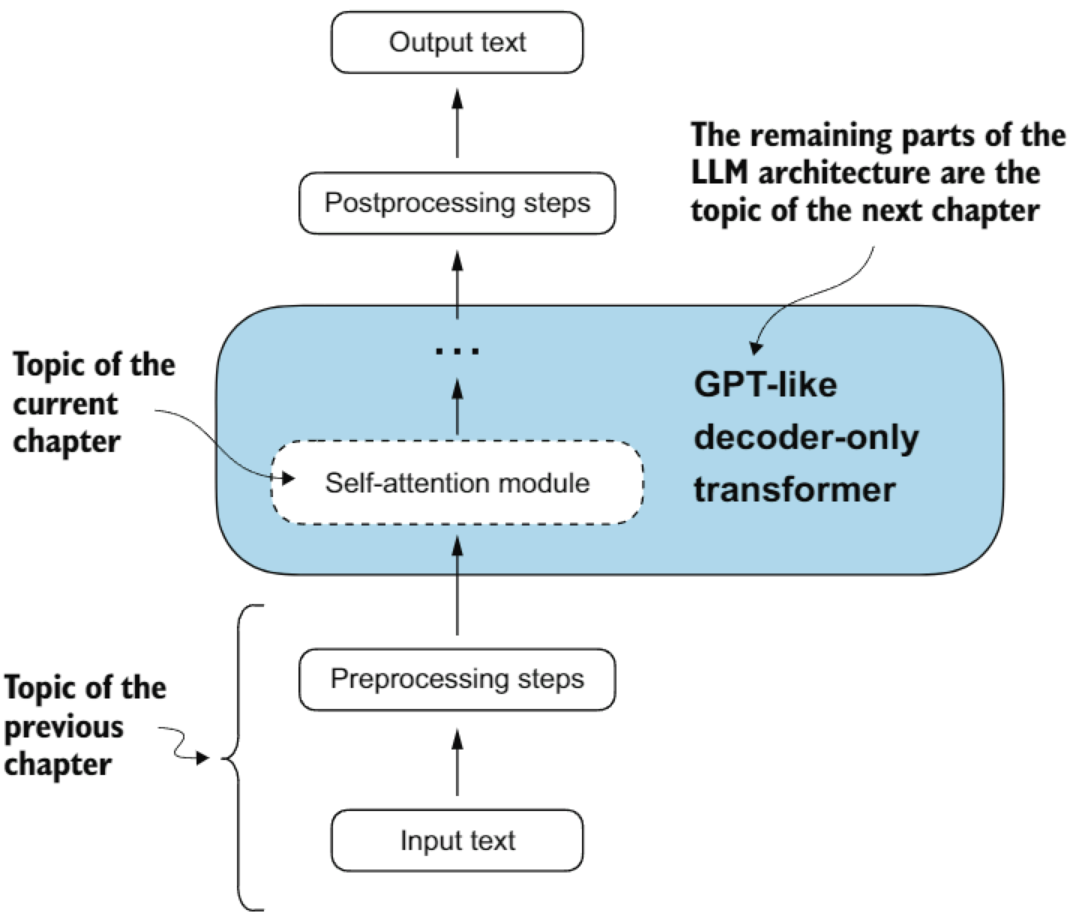

Figure 3.6 Self-attention is a mechanism in transformers used to compute more efficient input representations by allowing each position in a sequence to interact with and weigh the importance of all other positions within the same sequence. In this chapter, we will code this self-attention mechanism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

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

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

researchers found that RNN architectures are not required for building deep neural networks for natural language processing and proposed the original transformer architecture (discussed in chapter 1) including a self-attention mechanism inspired by the Bahdanau attention mechanism.

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

This chapter focuses on coding and understanding this self-attention mechanism used in GPT-like models, as illustrated in figure 3.6. In the

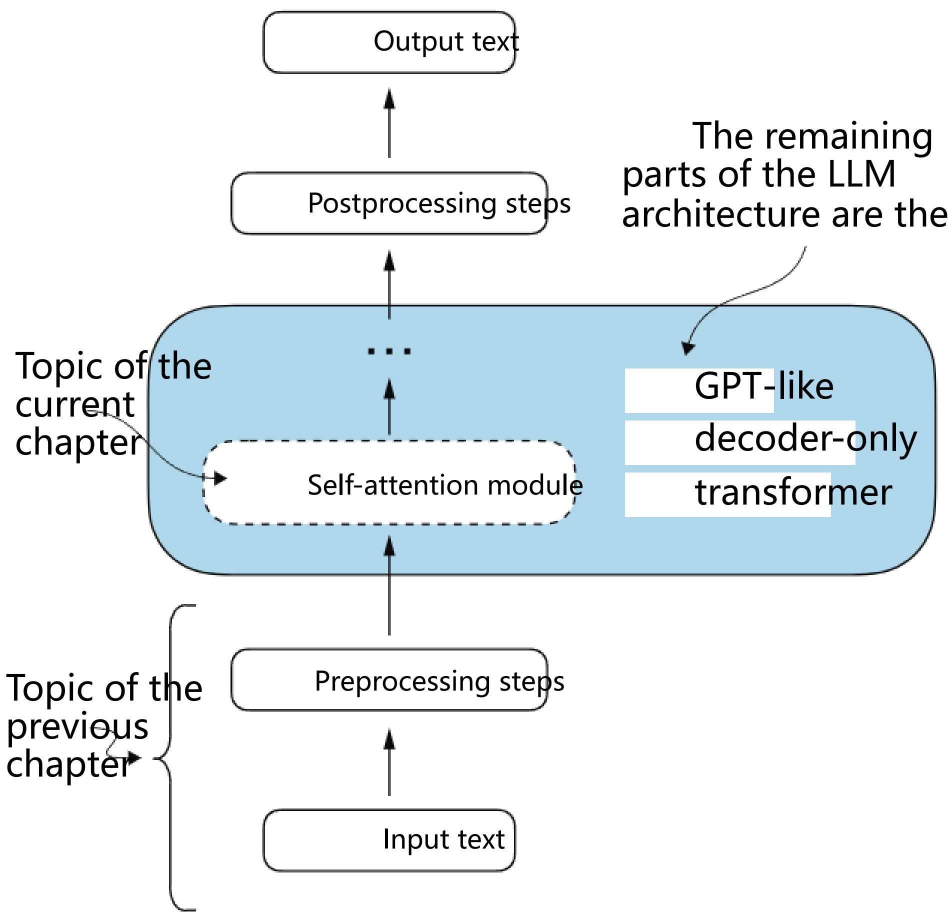

Figure 3.6 Self-attention is a mechanism in transformers used to compute more efficient input representations by allowing each position in a sequence to interact with and weigh the importance of all other positions within the same sequence. In this chapter, we will code this self-attention mechanism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

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

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

The "self" in self-attention

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

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

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

3.3.1 A simple self-attention mechanism without trainable weights

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

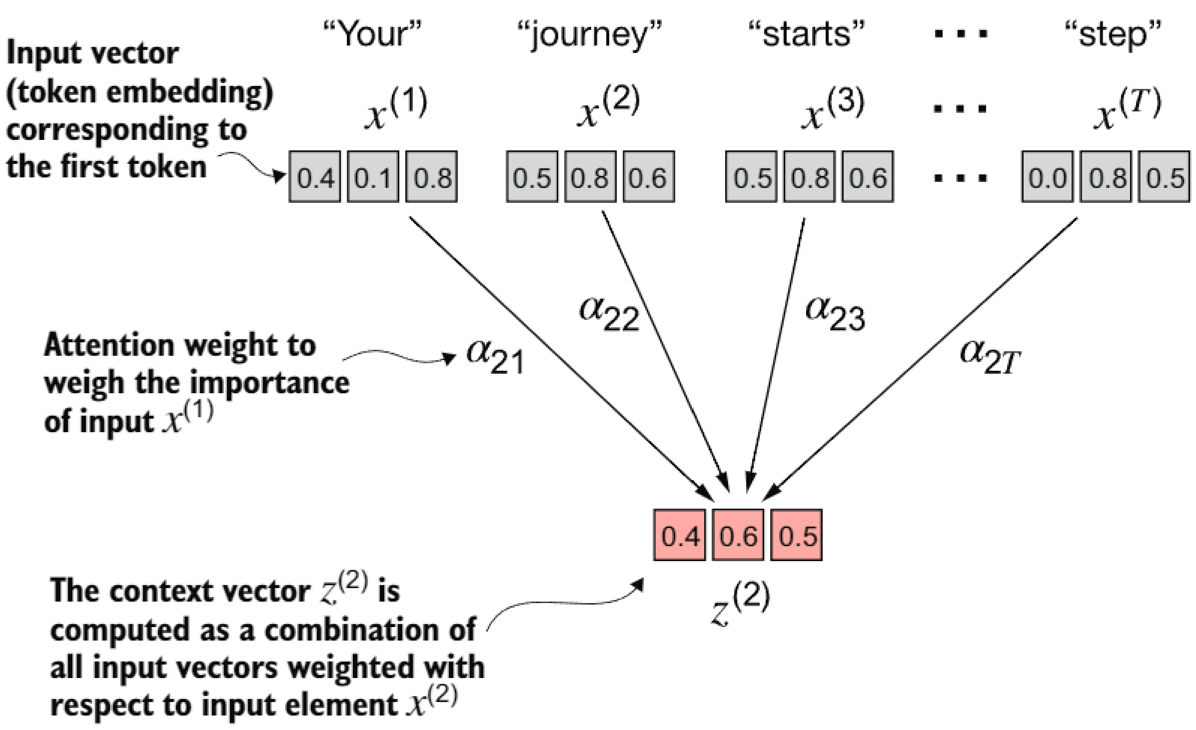

Figure 3.7 The goal of self-attention is to compute a context vector for each input element that combines information from all other input elements. In this example, we compute the context vector $z^{(2)}$. The importance or contribution of each input element for computing $z^{(2)}$ is determined by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

The "self" in self-attention

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

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

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

3.3.1 A simple self-attention mechanism without trainable weights

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

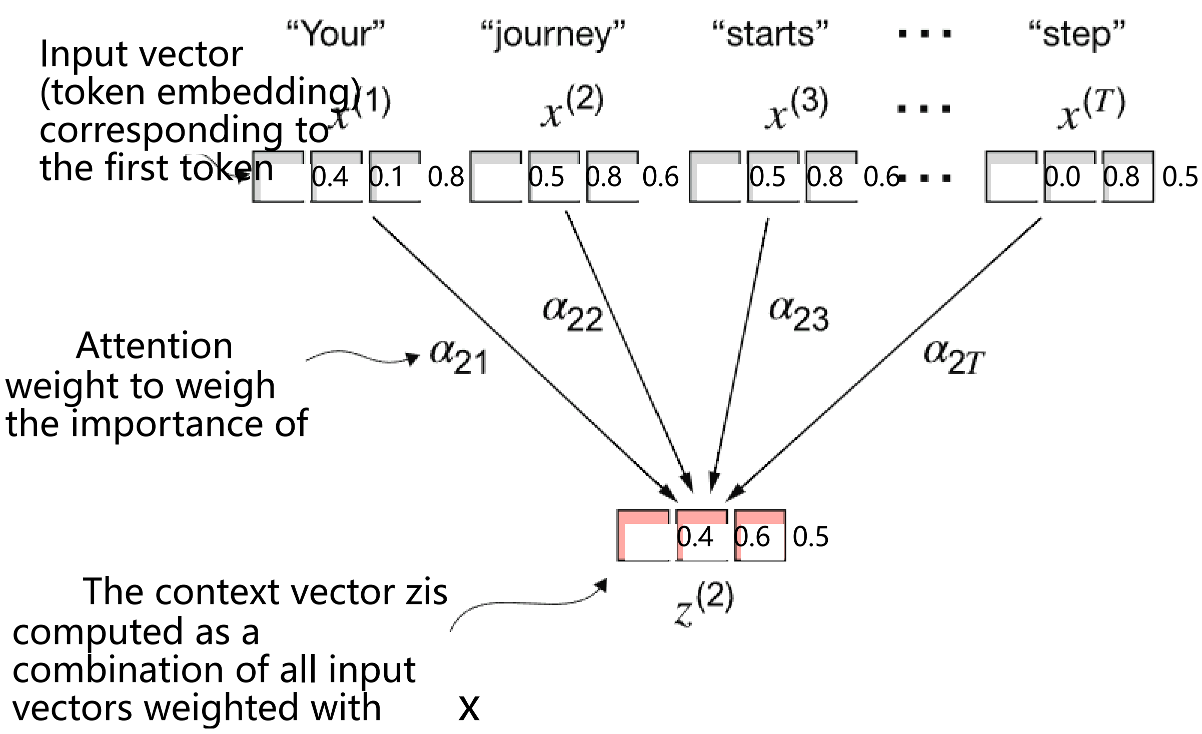

Figure 3.7 The goal of self-attention is to compute a context vector for each input element that combines information from all other input elements. In this example, we compute the context vector z. The importance or contribution of each input element for computing zis determined by the attention weights □to □. When computing z, the attention weights are calculated with respect to input element xand all other inputs.

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

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

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

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

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

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

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

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

Figure 3.7 shows an input sequence, denoted as x, consisting of T elements represented as xto x. This sequence typically represents text, such as a sentence, that has already been transformed into token embeddings.

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

In self-attention, our goal is to calculate context vectors zfor each element x in the input sequence. A context vector can be interpreted as an enriched embedding vector.

To illustrate this concept, let’s focus on the embedding vector of the second input element, x(which corresponds to the token “journey”), and the corresponding context vector, z, shown at the bottom of figure 3.7. This enhanced context vector, z, is an embedding that contains information about xand all other input elements, xto x.

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

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

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

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

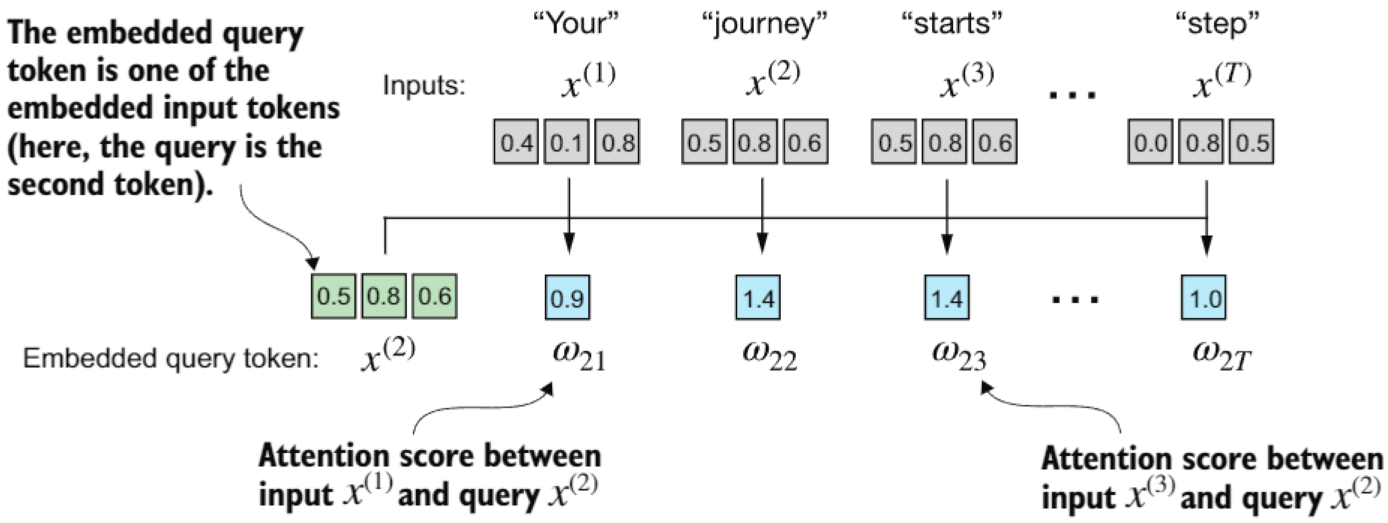

Figure 3.8 The overall goal is to illustrate the computation of the context vector $z^{(2)}$ using the second input element, $x^{(2)}$ as a query. This figure shows the first intermediate step, computing the attention scores ω between the query $x^{(2)}$ and all other input elements as a dot product.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

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

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

The second input token serves as the query.

The computed attention scores are

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

Understanding dot products

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

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

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

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

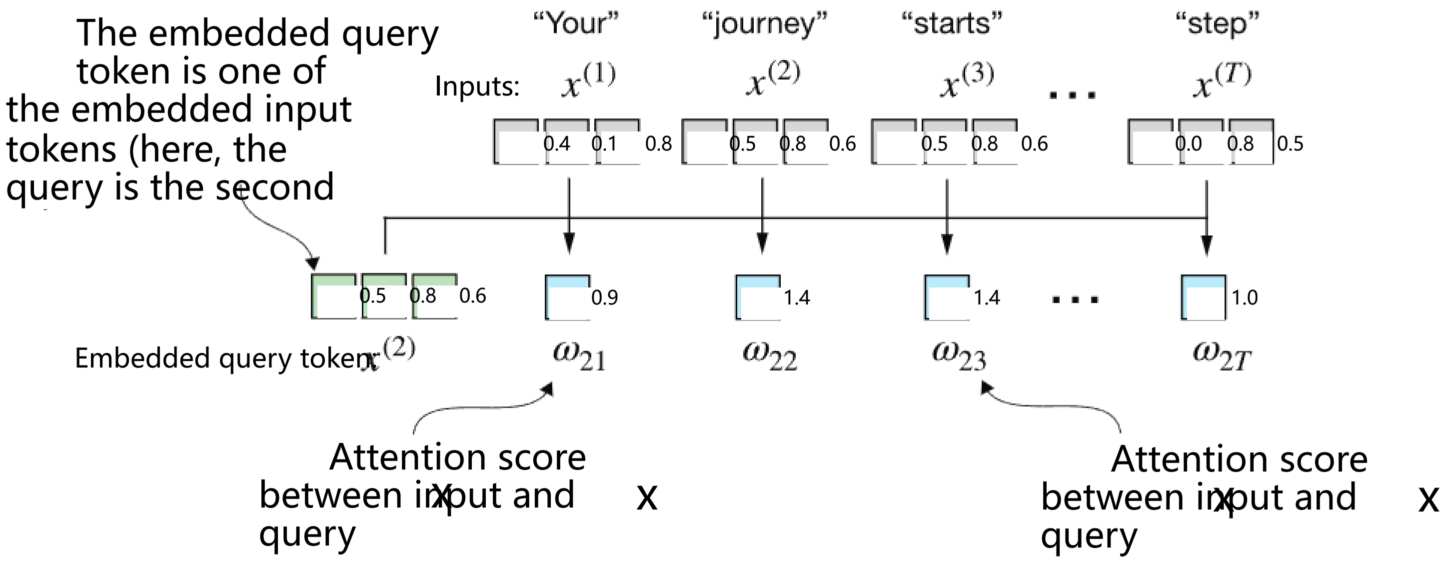

Figure 3.8 The overall goal is to illustrate the computation of the context vector zusing the second input element, xas a query. This figure shows the first intermediate step, computing the attention scores between the query xand all other input elements as a dot product. (Note that the numbers are truncated to one digit after the decimal point to reduce visual clutter.)

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

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

The second input token serves as the query.

The computed attention scores are

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

Understanding dot products

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

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

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

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

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

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

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

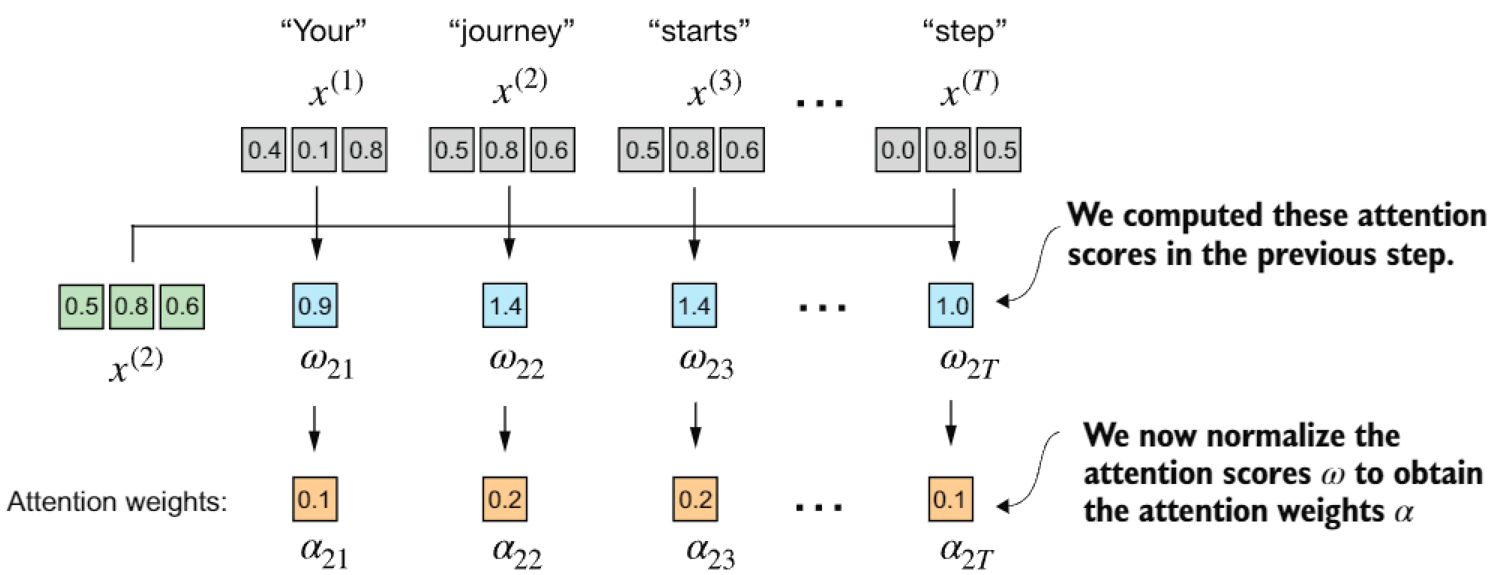

Figure 3.9 After computing the attention scores ω_{21} to ω_{2T} with respect to the input query $x^{(2)}$, the next step is to obtain the attention weights α_{21} to α_{2T} by normalizing the attention scores.

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

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

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

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

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

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

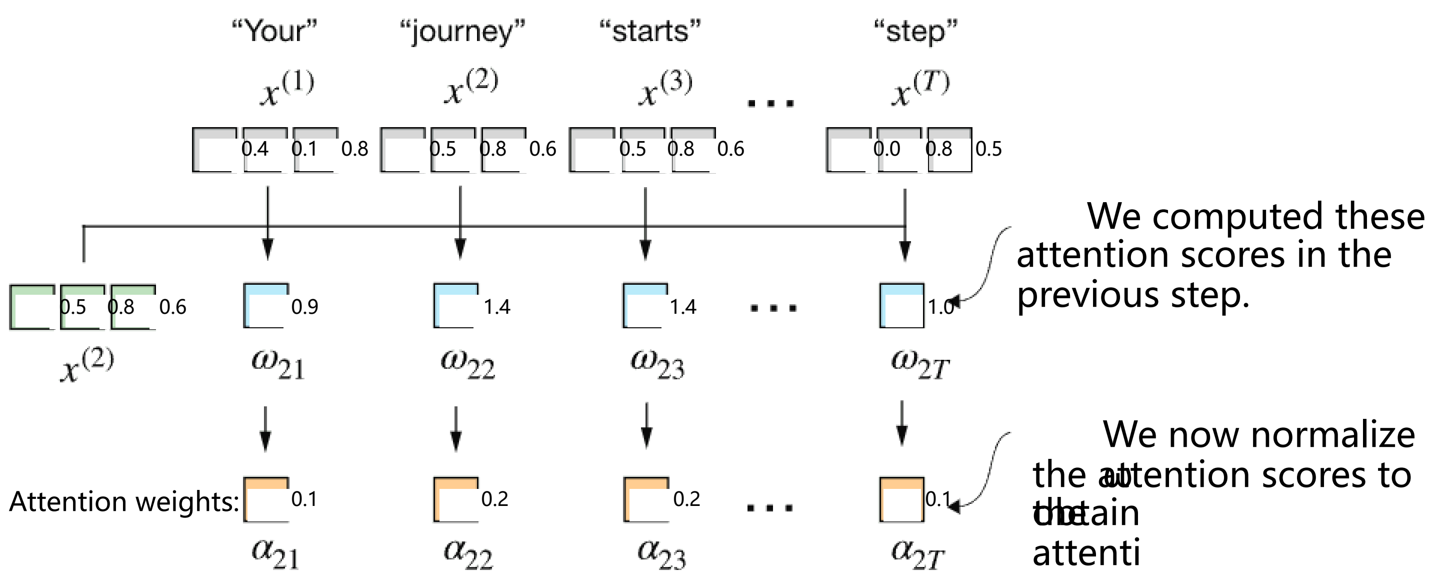

Figure 3.9 After computing the attention scores □to □with respect to the input query x , the next step is to obtain the attention weights □to □by normalizing the attention scores.

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

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

In practice, it's more common and advisable to use the softmax function for normalization. This approach is better at managing extreme values and offers more favorable gradient properties during training. The following is a basic implementation of the softmax function for normalizing the attention scores:

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

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

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

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

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

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

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

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

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

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

```
query = inputs[1]                               # The second input token is the query.
context_vec_2 = torch.zeros(query.shape)
for i,x_i in enumerate(inputs):
    context_vec_2 += attn_weights_2[i]*x_i
print(context_vec_2)
```

The results of this computation are

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

This approach is better at managing extreme values and offers more favorable gradient properties during training. The following is a basic implementation of the softmax function for normalizing the attention scores:

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

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

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

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

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

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

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

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

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

Now that we have computed the normalized attention weights, we are ready for the final step, as shown in figure 3.10: calculating the context vector zby multiplying the embedded input tokens, x, with the corresponding attention weights and then summing the resulting vectors. Thus, context vector zis the weighted sum of all input vectors, obtained by multiplying each input vector by its corresponding attention weight:

```
query = inputs[1]                                   # The second input token is the query.
context_vec_2 = torch.zeros(query.shape) for
i,x_i in enumerate(inputs):
    context_vec_2 += attn_weights_2[i]*x_i
print(context_vec_2)
```

The results of this computation are

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

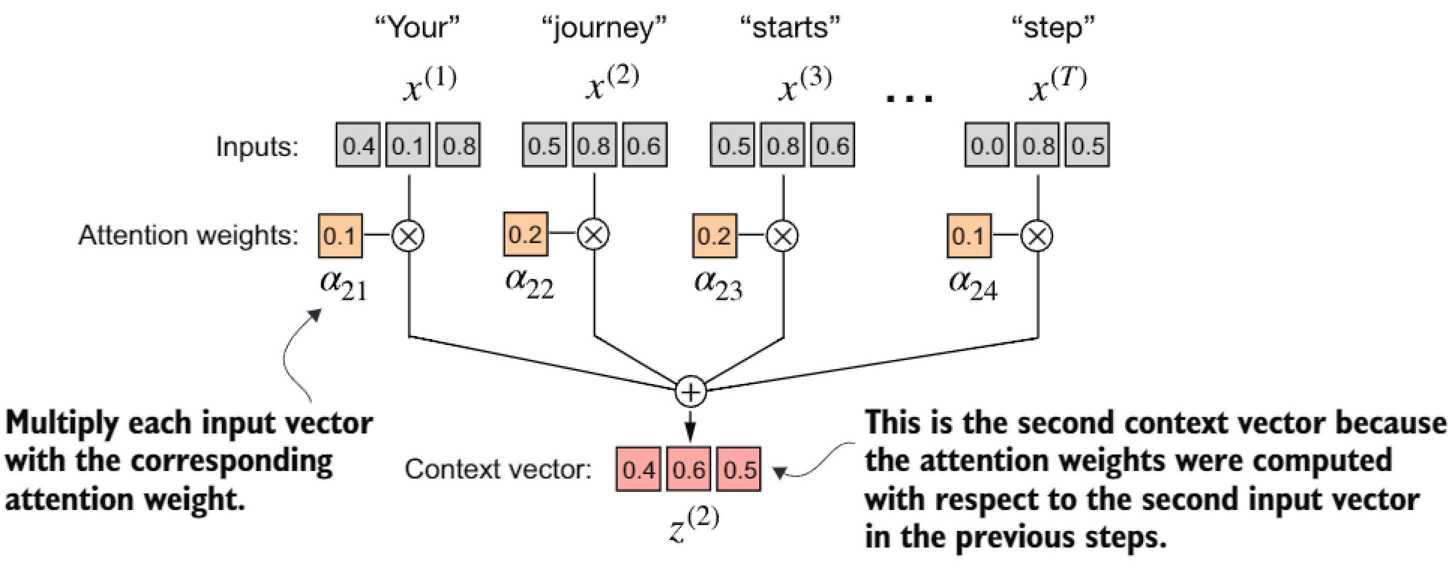

Figure 3.10 The final step, after calculating and normalizing the attention scores to obtain the attention weights for query $x^{(2)}$, is to compute the context vector $z^{(2)}$. This context vector is a combination of all input vectors $x^{(1)}$ *to* $x^{(T)}$ weighted by the attention weights.

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

3.3.2 Computing attention weights for all input tokens

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

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

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

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

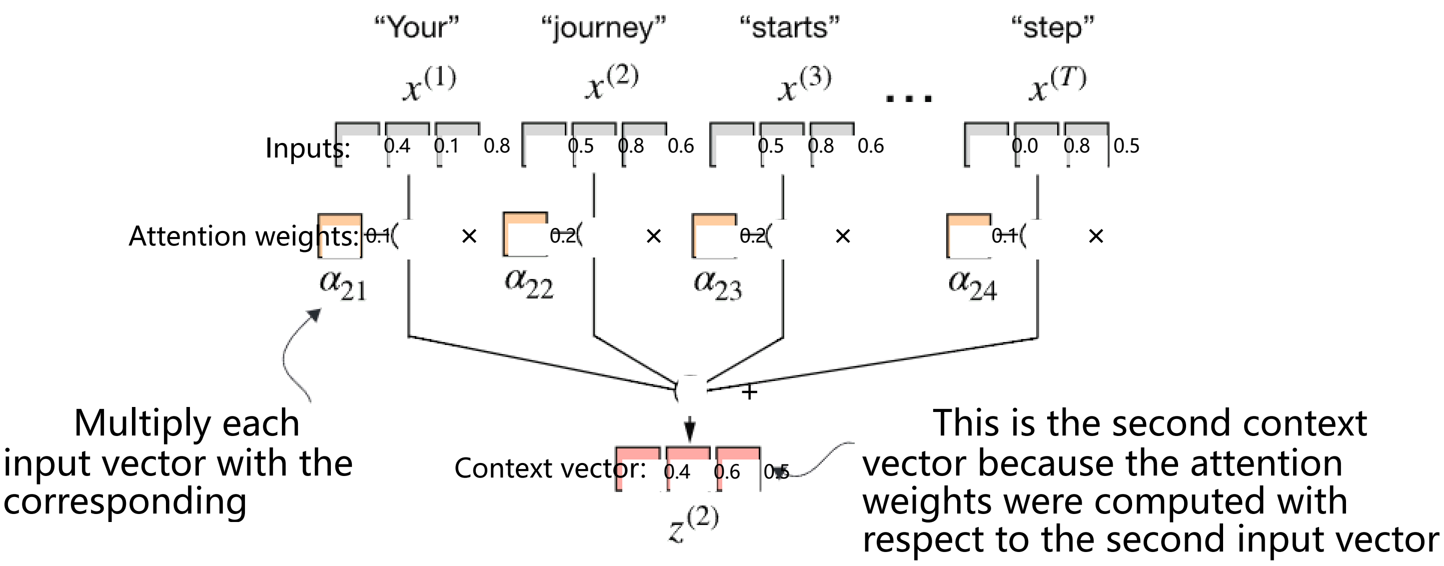

Figure 3.10 The final step, after calculating and normalizing the attention scores to obtain the attention weights for query x, is to compute the context vector z. This context vector is a combination of all input vectors xto xweighted by the attention weights.

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

3.3.2 Computing attention weights for all input tokens

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

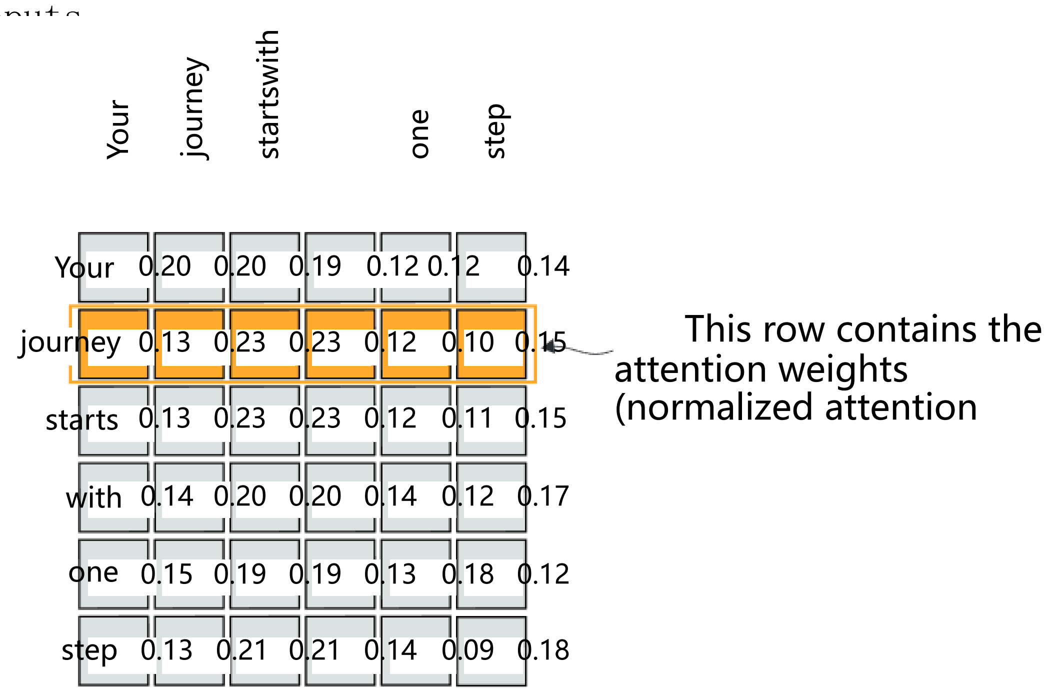

Figure 3.11 The highlighted row shows the attention weights for the second input element as a query. Now we will generalize the computation to obtain all other attention weights. (Please note that the numbers in this figure are truncated to two digits after the decimal point to reduce visual clutter.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

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

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

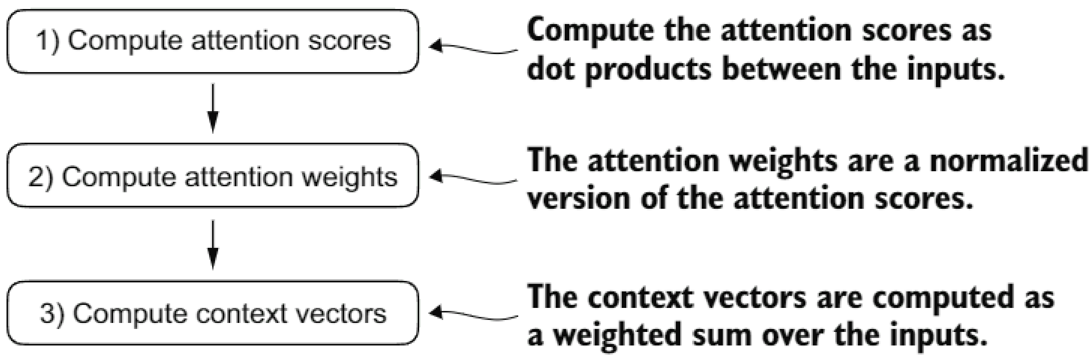

Figure 3.12 In step 1, we add an additional `for` loop to compute the dot products for all pairs of inputs.

The resulting attention scores are as follows:

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

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

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

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

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

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

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

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

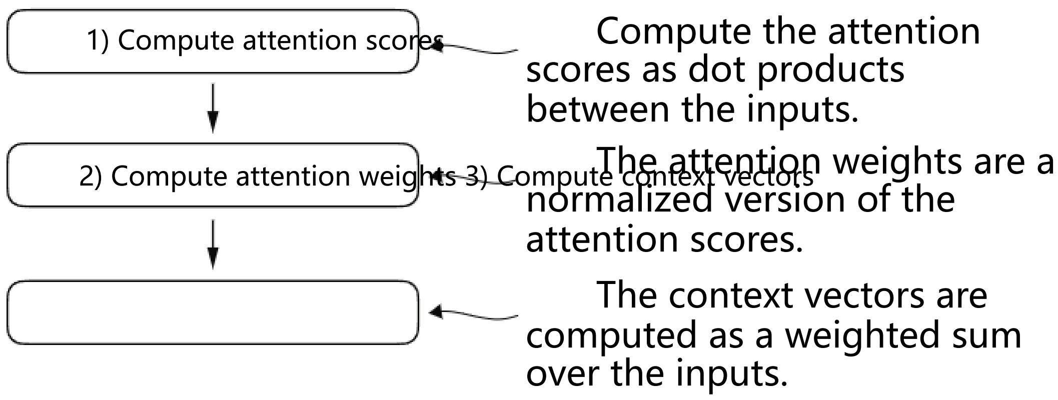

Figure 3.12 In step 1, we add an additional for loop to compute the dot products for all pairs of inputs.

The resulting attention scores are as follows:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The result is

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

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

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

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

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

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

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

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

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

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

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

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

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

The result is

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

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

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

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

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

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

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

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

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

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

3.4 Implementing self-attention with trainable weights

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

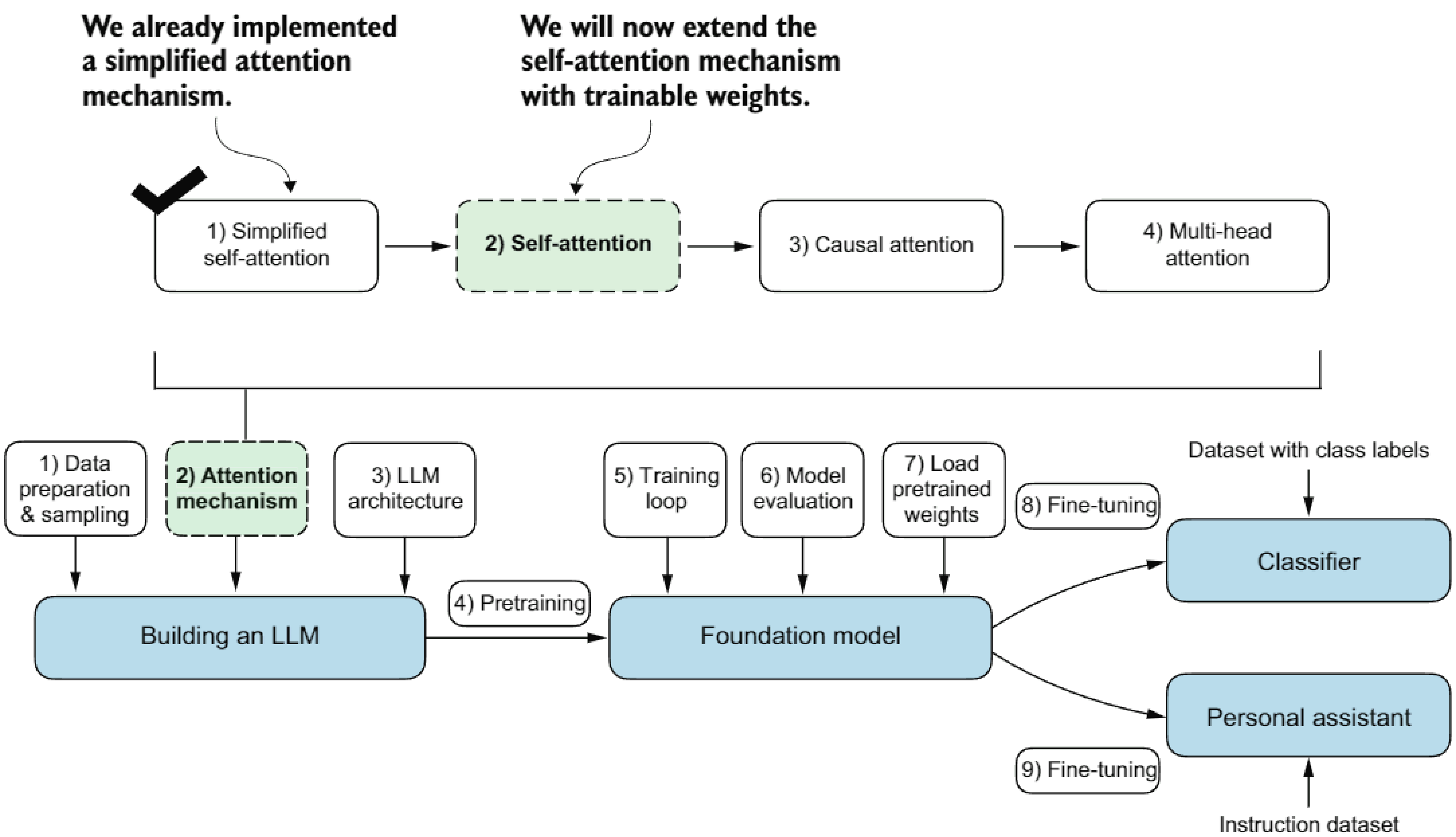

Figure 3.13 Previously, we coded a simplified attention mechanism to understand the basic mechanism behind attention mechanisms. Now, we add trainable weights to this attention mechanism. Later, we will extend this self-attention mechanism by adding a causal mask and multiple heads.

We can double-check that the code is correct by comparing the second row with the context vector zthat we computed in section 3.3.1:

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

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

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

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

3.4 Implementing self-attention with trainable weights

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

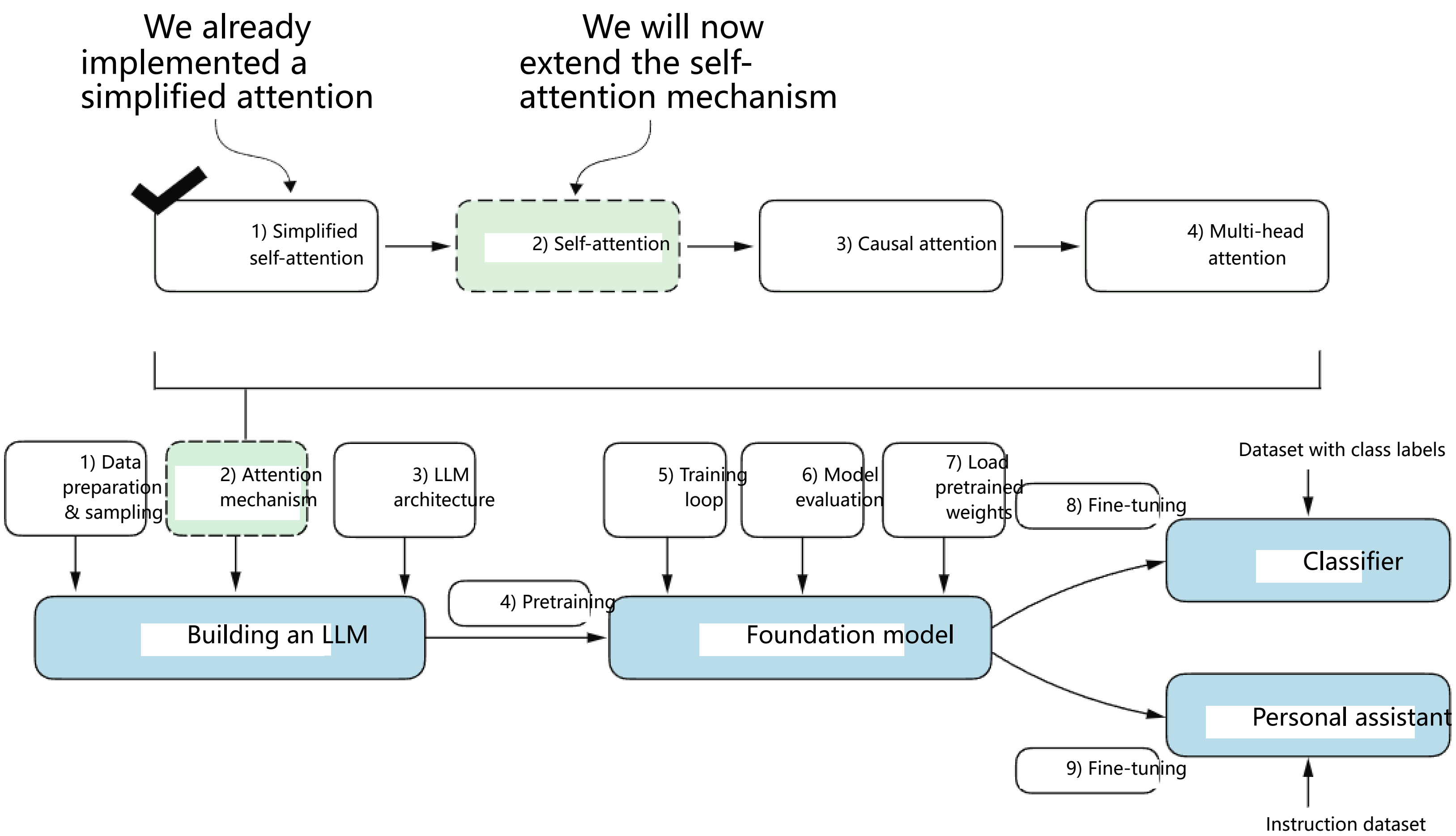

Figure 3.13 Previously, we coded a simplified attention mechanism to understand the basic mechanism behind attention mechanisms. Now, we add trainable weights to this attention mechanism. Later, we will extend this self-attention mechanism by adding a causal mask and multiple heads.

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

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

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

3.4.1 Computing the attention weights step by step

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

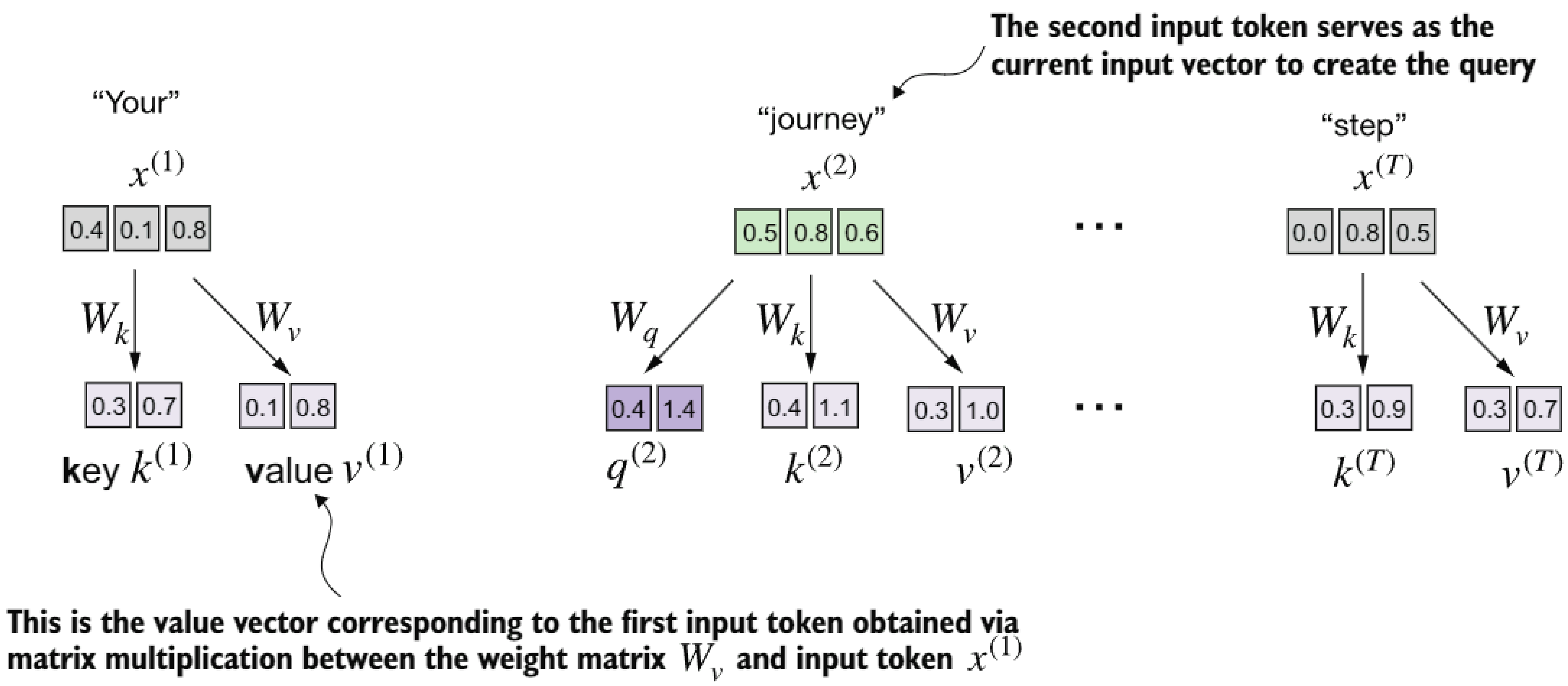

Figure 3.14 In the first step of the self-attention mechanism with trainable weight matrices, we compute query (q), key (k), and value (v) vectors for input elements x. Similar to previous sections, we designate the second input, $x^{(2)}$, as the query input. The query vector $q^{(2)}$ is obtained via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

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

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

The most notable difference is the introduction of weight matrices that are updated during model training. These trainable weight matrices are crucial so that the model (specifically, the attention module inside the model) can learn to produce "good" context vectors. (We will train the LLM in chapter 5.) We will tackle this self-attention mechanism in the two subsections. First, we will code it step by step as before. Second, we will organize the code into a compact Python class that can be imported into the LLM architecture.

3.4.1 Computing the attention weights step by step

We will implement the self-attention mechanism step by step by introducing the three trainable weight matrices W, W, and W. These three matrices are used to project the embedded input tokens, x, into query, key, and value vectors, respectively, as illustrated in figure 3.14.

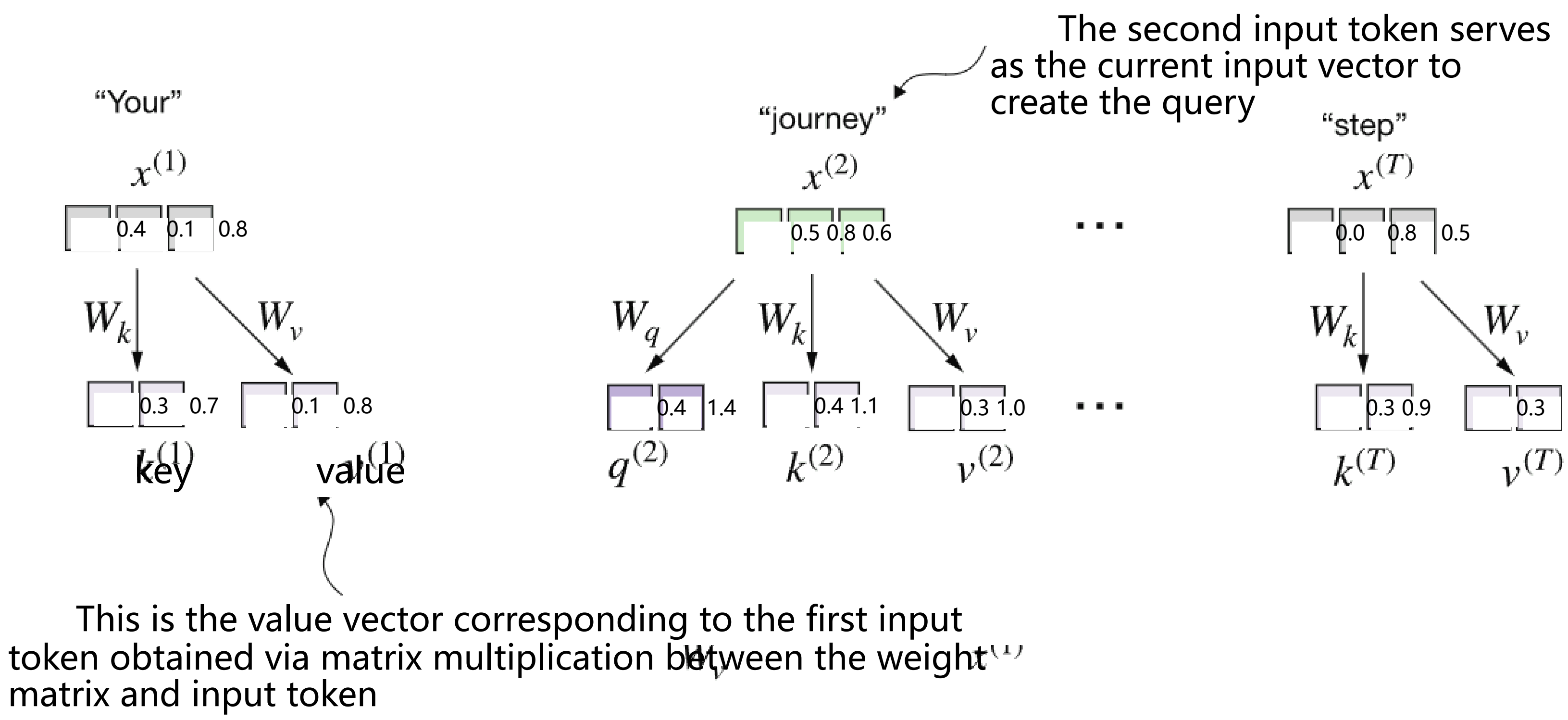

Figure 3.14 In the first step of the self-attention mechanism with trainable weight matrices, we compute query (q), key (k), and value (v) vectors for input elements x. Similar to previous sections, we designate the second input, x, as the query input. The query vector qis obtained via matrix multiplication between the input xand the weight matrix W. Similarly, we obtain the key and value vectors via matrix multiplication involving the weight matrices Wand W.

Earlier, we defined the second input element xas the query when we computed the simplified attention weights to compute the context vector z. Then we generalized this to compute all context vectors z... zfor the six-word input sentence "Your journey starts with one step."

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

Let's begin by defining a few variables:

```
x_2 = inputs[1]          # The second input element
d_in = inputs.shape[1]   # The input embedding size, d=3
d_out = 2                # The output embedding size, d_out=2
```

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

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

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

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

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

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

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

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

Weight parameters vs. attention weights

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

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

Similarly, we start here by computing only one context vector, z, for illustration purposes. We will then modify this code to calculate all context vectors.

Let's begin by defining a few variables:

```
x_2 = inputs[1]
d_in = inputs.shape[1]
d_out = 2
```

The second input element

The input embedding size, d=3

The output embedding size, d_out=2

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

Next, we initialize the three weight matrices W, W, and Wshown in figure 3.14:

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

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

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

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

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

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

Weight parameters vs. attention weights

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

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

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

We can obtain all keys and values via matrix multiplication:

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

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

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

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

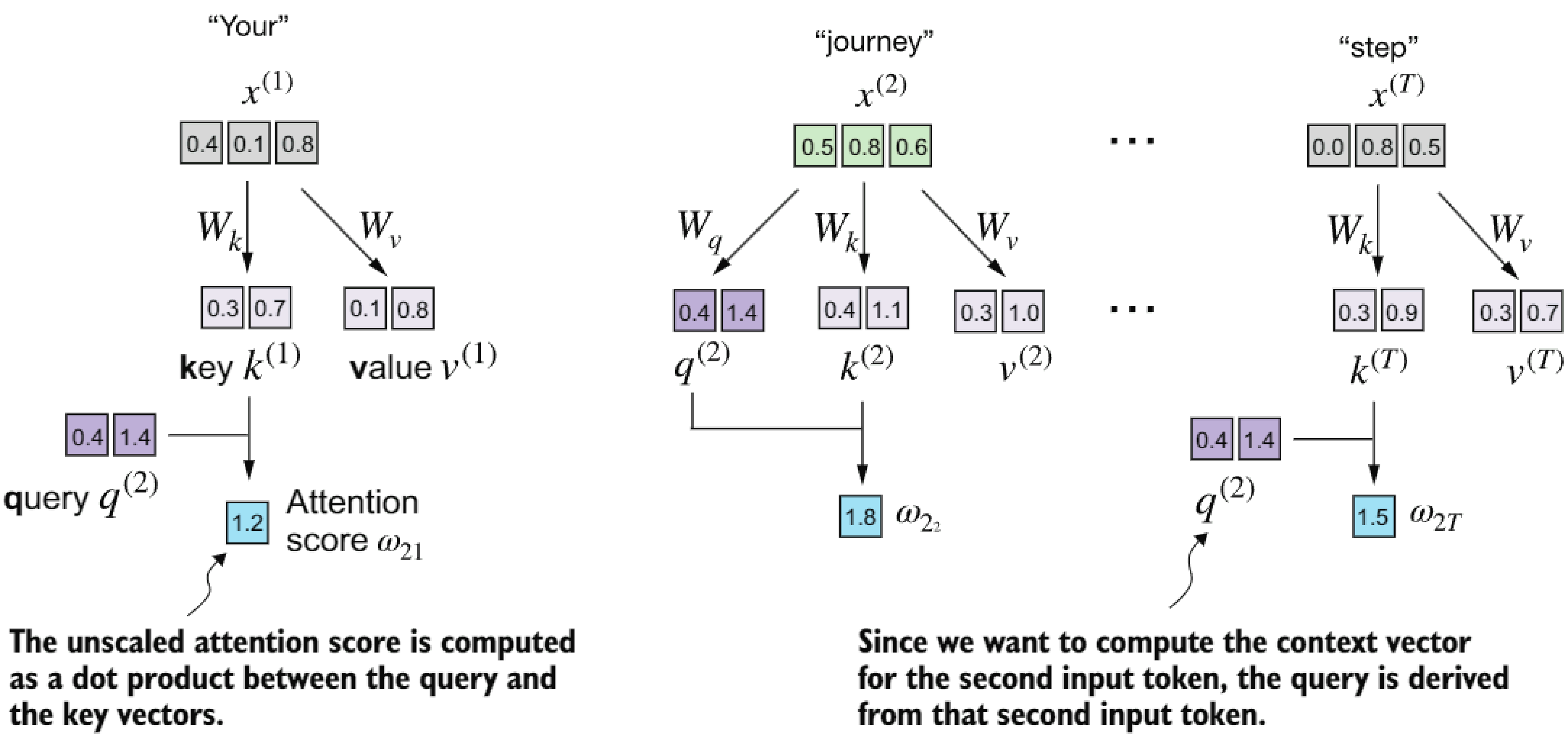

Figure 3.15 The attention score computation is a dot-product computation similar to what we used in the simplified self-attention mechanism in section 3.3. The new aspect here is that we are not directly computing the dot-product between the input elements but using the query and key obtained by transforming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

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

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

Remember that Python starts indexing at 0.

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

We can obtain all keys and values via matrix multiplication:

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

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

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

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

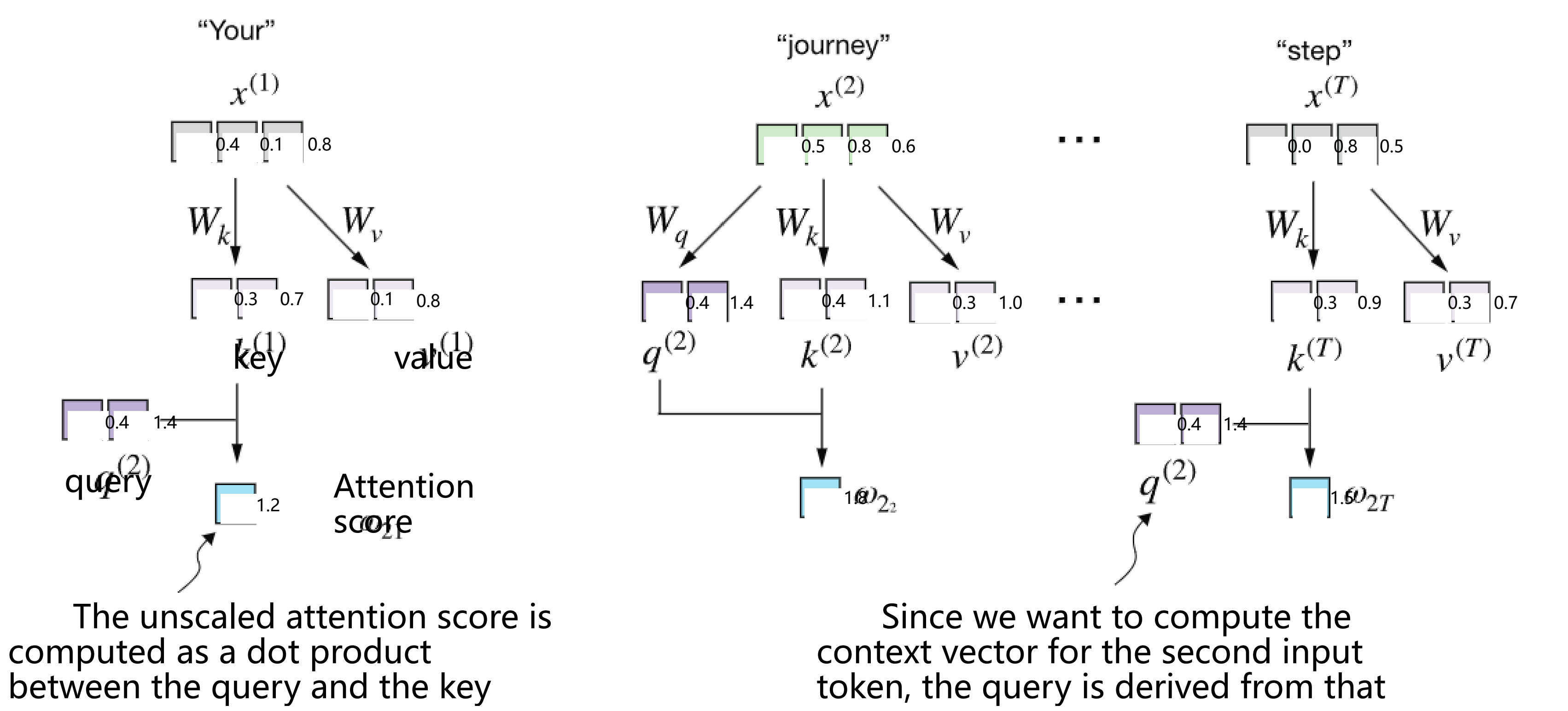

Figure 3.15 The attention score computation is a dot-product computation similar to what we used in the simplified self-attention mechanism in section 3.3. The new aspect here is that we are not directly computing the dot-product between the input elements but using the query and key obtained by transforming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

First, let's compute the attention score ω:

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

Remember that Python starts indexing at 0.

The result for the unnormalized attention score is

```
tensor(1.8524)
```

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

```
attn_scores_2 = query_2 @ keys.T
print(attn_scores_2)
```

All attention scores for given query

As we can see, as a quick check, the second element in the output matches the `attn_score_22` we computed previously:

```
tensor([1.2705, 1.8524, 1.8111, 1.0795, 0.5577, 1.5440])
```

Now, we want to go from the attention scores to the attention weights, as illustrated in figure 3.16. We compute the attention weights by scaling the attention scores and using the softmax function. However, now we scale the attention scores by dividing them by the square root of the embedding dimension of the keys (taking the square root is mathematically the same as exponentiating by 0.5):

```
d_k = keys.shape[-1]
attn_weights_2 = torch.softmax(attn_scores_2 / d_k**0.5, dim=-1)
print(attn_weights_2)
```

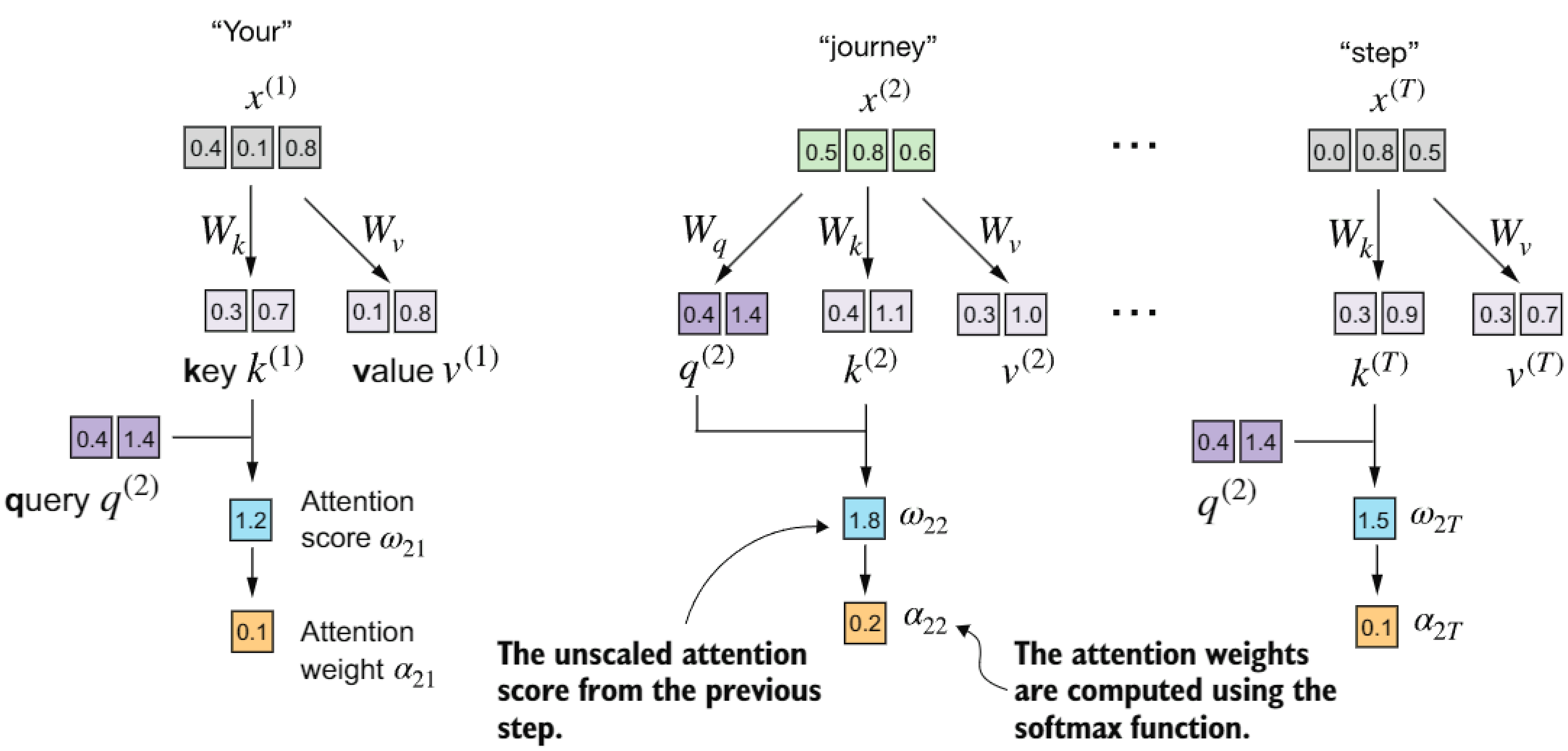

Figure 3.16 After computing the attention scores ω, the next step is to normalize these scores using the softmax function to obtain the attention weights α.

The result for the unnormalized attention score is

```
tensor(1.8524)
```

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

```
attn_scores_2 = query_2 @ keys.T
print(attn_scores_2)
```

All attention scores for given query

As we can see, as a quick check, the second element in the output matches the `attn_score_22` we computed previously:

```
tensor([1.2705, 1.8524, 1.8111, 1.0795, 0.5577, 1.5440])
```

Now, we want to go from the attention scores to the attention weights, as illustrated in figure 3.16. We compute the attention weights by scaling the attention scores and using the softmax function. However, now we scale the attention scores by dividing them by the square root of the embedding dimension of the keys (taking the square root is mathematically the same as exponentiating by 0.5):

```
d_k = keys.shape[-1] attn_weights_2 = torch.softmax(attn_scores_2 /
d_k**0.5, dim=-1) print(attn_weights_2)
```

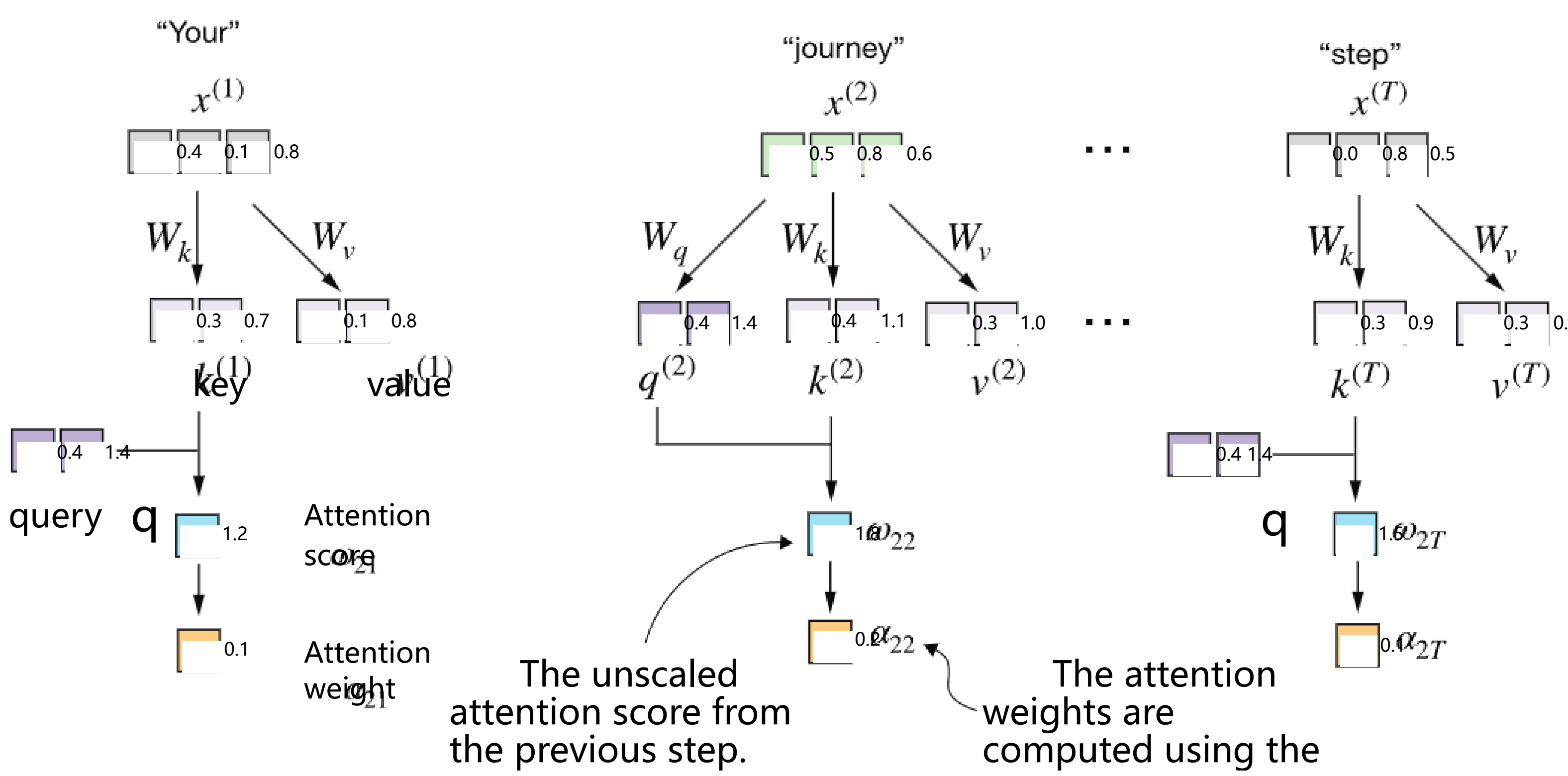

Figure 3.16 After computing the attention scores , the next step is to normalize these scores using the softmax function to obtain the attention weights .

The resulting attention weights are

```
tensor([0.1500, 0.2264, 0.2199, 0.1311, 0.0906, 0.1820])
```

The rationale behind scaled-dot product attention

The reason for the normalization by the embedding dimension size is to improve the training performance by avoiding small gradients. For instance, when scaling up the embedding dimension, which is typically greater than 1,000 for GPT-like LLMs, large dot products can result in very small gradients during backpropagation due to the softmax function applied to them. As dot products increase, the softmax function behaves more like a step function, resulting in gradients nearing zero. These small gradients can drastically slow down learning or cause training to stagnate.

The scaling by the square root of the embedding dimension is the reason why this self-attention mechanism is also called scaled-dot product attention.

Now, the final step is to compute the context vectors, as illustrated in figure 3.17.

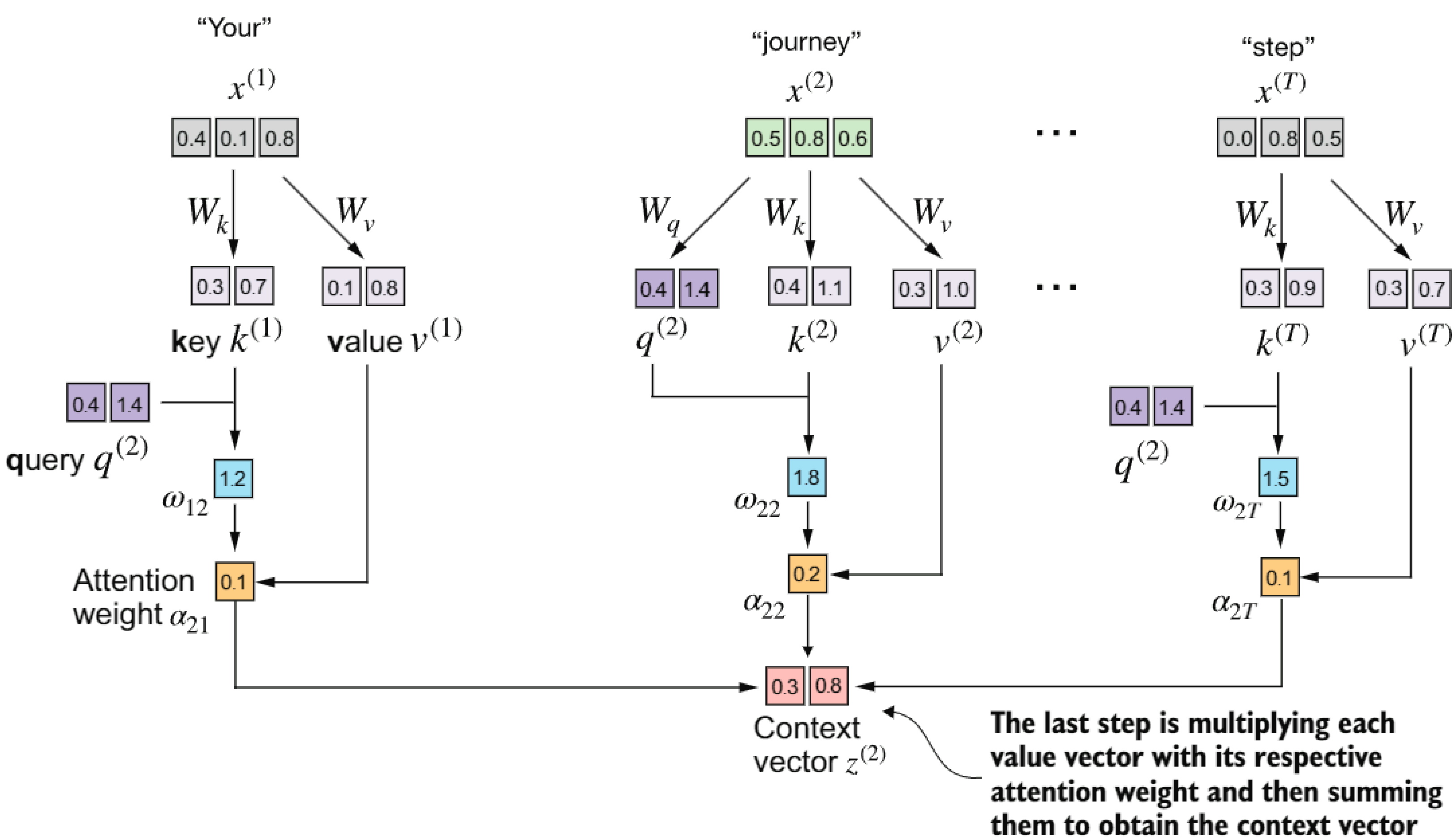

Figure 3.17 In the final step of the self-attention computation, we compute the context vector by combining all value vectors via the attention weights.

Similar to when we computed the context vector as a weighted sum over the input vectors (see section 3.3), we now compute the context vector as a weighted sum over the value vectors. Here, the attention weights serve as a weighting factor that weighs

The resulting attention weights are

```
tensor([0.1500, 0.2264,    0.2199, 0.1311,    0.0906, 0.1820])
```

The rationale behind scaled-dot product attention

The reason for the normalization by the embedding dimension size is to improve the training performance by avoiding small gradients. For instance, when scaling up the embedding dimension, which is typically greater than 1,000 for GPT-like LLMs, large dot products can result in very small gradients during backpropagation due to the softmax function applied to them. As dot products increase, the softmax function behaves more like a step function, resulting in gradients nearing zero. These small gradients can drastically slow down learning or cause training to stagnate.

The scaling by the square root of the embedding dimension is the reason why this self-attention mechanism is also called scaled-dot product attention.

Now, the final step is to compute the context vectors, as illustrated in figure 3.17.

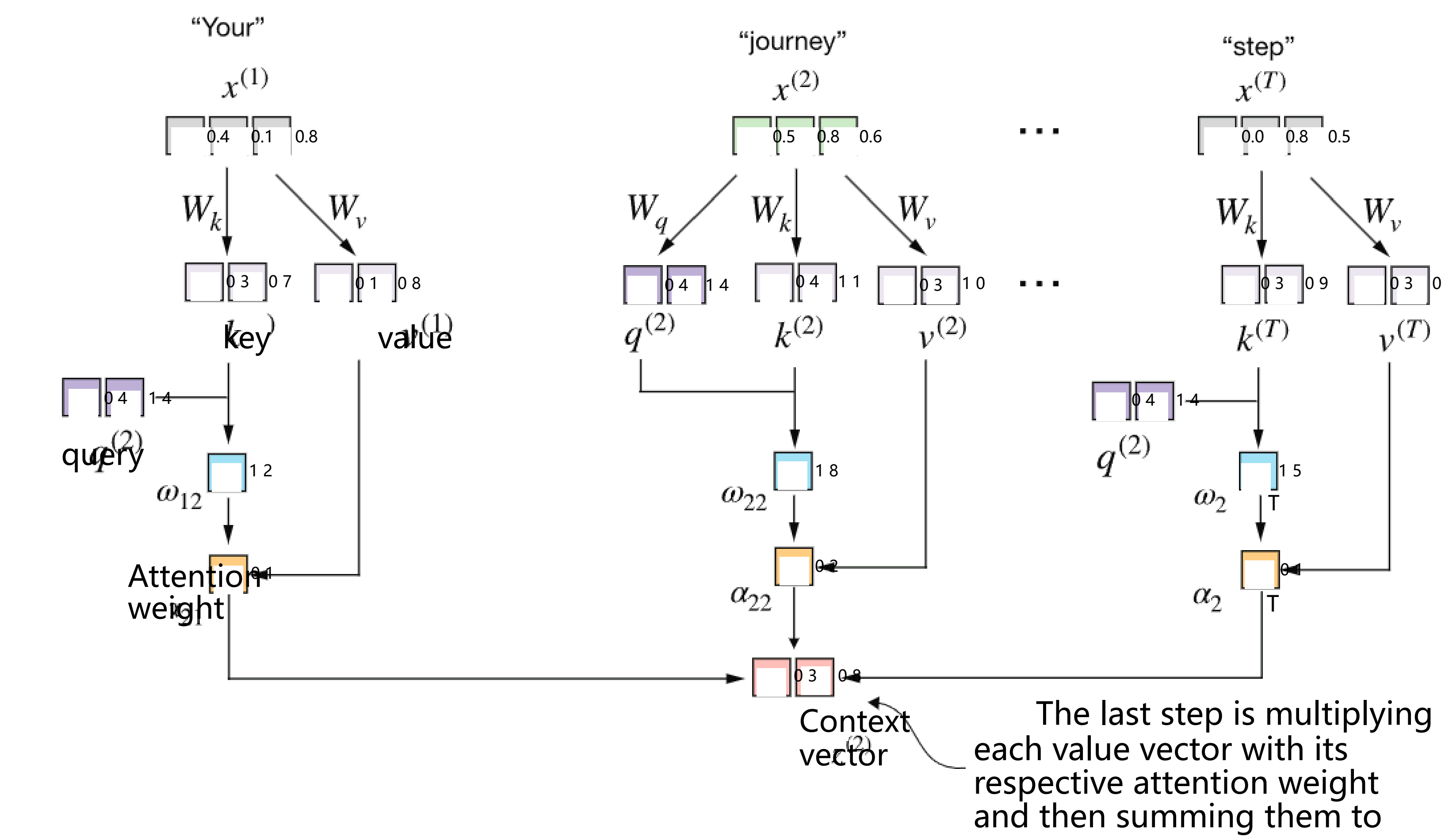

Figure 3.17 In the final step of the self-attention computation, we compute the context vector by combining all value vectors via the attention weights.

Similar to when we computed the context vector as a weighted sum over the input vectors (see section 3.3), we now compute the context vector as a weighted sum over the value vectors. Here, the attention weights serve as a weighting factor that weighs the respective importance of each value vector. Also as before, we can use matrix multiplication to obtain the output in one step:

```
context_vec_2 = attn_weights_2 @ values
print(context_vec_2)
```

The contents of the resulting vector are as follows:

```
tensor([0.3061, 0.8210])
```

So far, we've only computed a single context vector, $z^{(2)}$. Next, we will generalize the code to compute all context vectors in the input sequence, $z^{(1)}$ to $z^{(T)}$.

Why query, key, and value?

The terms "key," "query," and "value" in the context of attention mechanisms are borrowed from the domain of information retrieval and databases, where similar concepts are used to store, search, and retrieve information.

A *query* is analogous to a search query in a database. It represents the current item (e.g., a word or token in a sentence) the model focuses on or tries to understand. The query is used to probe the other parts of the input sequence to determine how much attention to pay to them.

The *key* is like a database key used for indexing and searching. In the attention mechanism, each item in the input sequence (e.g., each word in a sentence) has an associated key. These keys are used to match the query.

The *value* in this context is similar to the value in a key-value pair in a database. It represents the actual content or representation of the input items. Once the model determines which keys (and thus which parts of the input) are most relevant to the query (the current focus item), it retrieves the corresponding values.

3.4.2 Implementing a compact self-attention Python class

At this point, we have gone through a lot of steps to compute the self-attention outputs. We did so mainly for illustration purposes so we could go through one step at a time. In practice, with the LLM implementation in the next chapter in mind, it is helpful to organize this code into a Python class, as shown in the following listing.

Listing 3.1 A compact self-attention class

```
import torch.nn as nn
class SelfAttention_v1(nn.Module):
    def __init__(self, d_in, d_out):
        super().__init__()
        self.W_query = nn.Parameter(torch.rand(d_in, d_out))
        self.W_key   = nn.Parameter(torch.rand(d_in, d_out))
        self.W_value = nn.Parameter(torch.rand(d_in, d_out))
```

the attention weights serve as a weighting factor that weighs the respective importance of each value vector. Also as before, we can use matrix multiplication to obtain the output in one step:

```
    context_vec_2 = attn_weights_2 @ values
print(context_vec_2)
```

The contents of the resulting vector are as follows:

```
tensor([0.3061,     0.8210])
```

So far, we' ve only computed a single context vector, z. Next, we will generalize the code to compute all context vectors in the input sequence, zto z.

Why query, key, and value?

The terms "key," "query," and "value" in the context of attention mechanisms are borrowed from the domain of information retrieval and databases, where similar concepts are used to store, search, and retrieve information.

A query is analogous to a search query in a database. It represents the current item (e.g., a word or token in a sentence) the model focuses on or tries to understand. The query is used to probe the other parts of the input sequence to determine how much attention to pay to them.

The key is like a database key used for indexing and searching. In the attention mechanism, each item in the input sequence (e.g., each word in a sentence) has an associated key. These keys are used to match the query.

The value in this context is similar to the value in a key-value pair in a database. It represents the actual content or representation of the input items. Once the model determines which keys (and thus which parts of the input) are most relevant to the query (the current focus item), it retrieves the corresponding values.

3.4.2 Implementing a compact self-attention Python class

At this point, we have gone through a lot of steps to compute the self-attention outputs. We did so mainly for illustration purposes so we could go through one step at a time. In practice, with the LLM implementation in the next chapter in mind, it is helpful to organize this code into a Python class, as shown in the following listing.

Listing 3.1 A compact self-attention class

```
    import torch.nn as nn class
SelfAttention_v1(nn.Module):
        def __init__(self, d_in, d_out):
            super().__init__()
            self.W_query = nn.Parameter(torch.rand(d_in, d_out))
         self.W_key = nn.Parameter(torch.rand(d_in, d_out)) self.W_value
         = nn.Parameter(torch.rand(d_in, d_out))
```

```
    def forward(self, x):
        keys = x @ self.W_key
        queries = x @ self.W_query
        values = x @ self.W_value
        attn_scores = queries @ keys.T # omega
        attn_weights = torch.softmax(
            attn_scores / keys.shape[-1]**0.5, dim=-1
        )
        context_vec = attn_weights @ values
        return context_vec
```

In this PyTorch code, `SelfAttention_v1` is a class derived from `nn.Module`, which is a fundamental building block of PyTorch models that provides necessary functionalities for model layer creation and management.

The `__init__` method initializes trainable weight matrices (`W_query`, `W_key`, and `W_value`) for queries, keys, and values, each transforming the input dimension `d_in` to an output dimension `d_out`.

During the forward pass, using the forward method, we compute the attention scores (`attn_scores`) by multiplying queries and keys, normalizing these scores using softmax. Finally, we create a context vector by weighting the values with these normalized attention scores.

We can use this class as follows:

```
torch.manual_seed(123)
sa_v1 = SelfAttention_v1(d_in, d_out)
print(sa_v1(inputs))
```

Since `inputs` contains six embedding vectors, this results in a matrix storing the six context vectors:

```
tensor([[0.2996, 0.8053],
        [0.3061, 0.8210],
        [0.3058, 0.8203],
        [0.2948, 0.7939],
        [0.2927, 0.7891],
        [0.2990, 0.8040]], grad_fn=<MmBackward0>)
```

As a quick check, notice that the second row (`[0.3061, 0.8210]`) matches the contents of `context_vec_2` in the previous section. Figure 3.18 summarizes the self-attention mechanism we just implemented.

Self-attention involves the trainable weight matrices W_q, W_k, and W_v. These matrices transform input data into queries, keys, and values, respectively, which are crucial components of the attention mechanism. As the model is exposed to more data during training, it adjusts these trainable weights, as we will see in upcoming chapters.

We can improve the `SelfAttention_v1` implementation further by utilizing PyTorch's `nn.Linear` layers, which effectively perform matrix multiplication when the bias units are disabled. Additionally, a significant advantage of using `nn.Linear`

```
def forward(self, x):
    keys = x @ self.W_key queries = x @
 self.W_query values = x @ self.W_value
 attn_scores = queries @ keys.T # omega
 attn_weights = torch.softmax(

    attn_scores / keys.shape[-1]**0.5, dim=-1 )
 context_vec = attn_weights @ values return context_vec
```

In this PyTorch code, SelfAttention_v1 is a class derived from nn.Module, which is a fundamental building block of PyTorch models that provides necessary functionalities for model layer creation and management.

The __init__ method initializes trainable weight matrices (W_query, W_key, and W_value) for queries, keys, and values, each transforming the input dimension d_in to an output dimension d_out.

During the forward pass, using the forward method, we compute the attention scores (attn_scores) by multiplying queries and keys, normalizing these scores using softmax. Finally, we create a context vector by weighting the values with these normalized attention scores.

We can use this class as follows:

```
torch.manual_seed(123) sa_v1 =
SelfAttention_v1(d_in, d_out)
print(sa_v1(inputs))
```

Since inputs contains six embedding vectors, this results in a matrix storing the six context vectors:

```
tensor([[0.2996,    0.8053],
        [0.3061, 0.8210],
        [0.3058, 0.8203],
        [0.2948, 0.7939],
        [0.2927, 0.7891],
        [0.2990,   0.8040]], grad_fn=<MmBackward0>)
```

As a quick check, notice that the second row ([0.3061, 0.8210]) matches the contents of context_vec_2 in the previous section. Figure 3.18 summarizes the self-attention mechanism we just implemented.

Self-attention involves the trainable weight matrices W, W, and W. These matrices transform input data into queries, keys, and values, respectively, which are crucial components of the attention mechanism. As the model is exposed to more data during training, it adjusts these trainable weights, as we will see in upcoming chapters.

We can improve the SelfAttention_v1 implementation further by utilizing PyTorch’s nn.Linear layers, which effectively perform matrix multiplication when the bias units are disabled. Additionally, a significant advantage of using nn.Linear

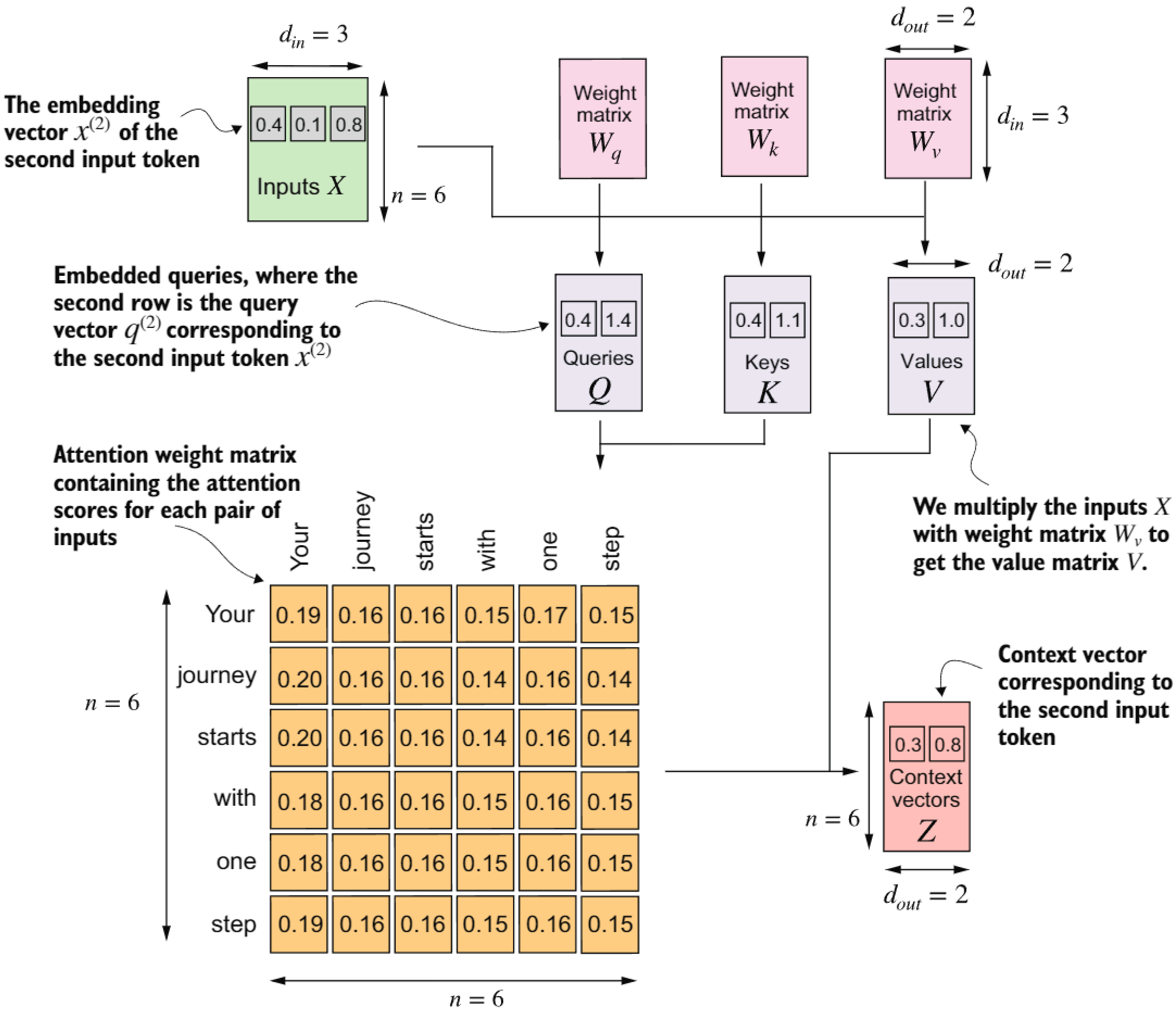

Figure 3.18 In self-attention, we transform the input vectors in the input matrix X with the three weight matrices, W_q, W_k, and W_v. The new compute the attention weight matrix based on the resulting queries (Q) and keys (K). Using the attention weights and values (V), we then compute the context vectors (Z). For visual clarity, we focus on a single input text with *n* tokens, not a batch of multiple inputs. Consequently, the three-dimensional input tensor is simplified to a two-dimensional matrix in this context. This approach allows for a more straightforward visualization and understanding of the processes involved. For consistency with later figures, the values in the attention matrix do not depict the real attention weights. (The numbers in this figure are truncated to two digits after the decimal point to reduce visual clutter. The values in each row should add up to 1.0 or 100%.)

instead of manually implementing `nn.Parameter(torch.rand(...))` is that `nn.Linear` has an optimized weight initialization scheme, contributing to more stable and effective model training.

Listing 3.2 A self-attention class using PyTorch's Linear layers

```
class SelfAttention_v2(nn.Module):
    def __init__(self, d_in, d_out, qkv_bias=False):
```

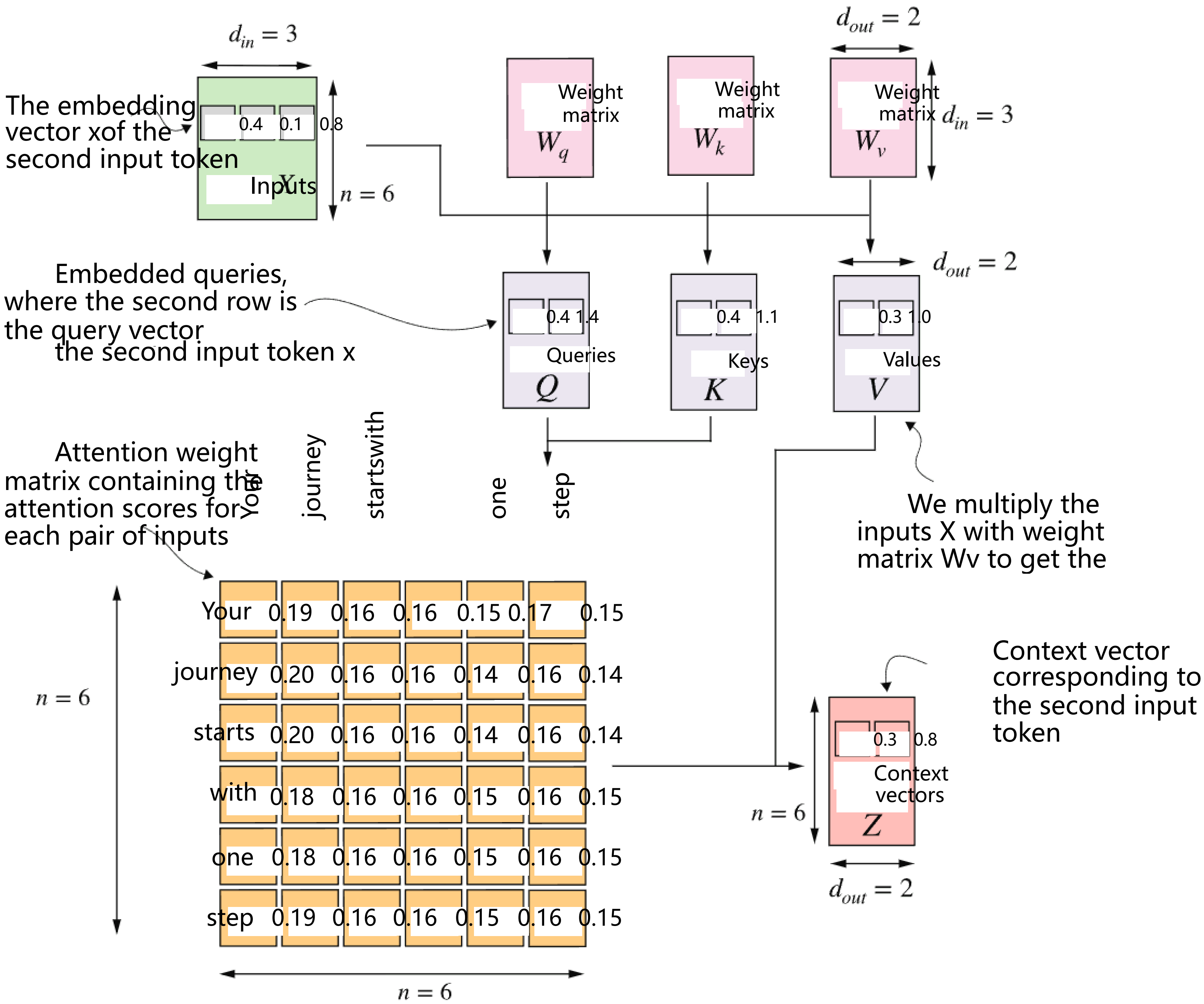

Figure 3.18 In self-attention, we transform the input vectors in the input matrix X with the three weight matrices, W , W , and W . The new compute the attention weight matrix based on the resulting queries (Q) and keys (K). Using the attention weights and values (V), we then compute the context vectors (Z). For visual clarity, we focus on a single input text with n tokens, not a batch of multiple inputs. Consequently, the three-dimensional input tensor is simplified to a two-dimensional matrix in this context. This approach allows for a more straightforward visualization and understanding of the processes involved. For consistency with later figures, the values in the attention matrix do not depict the real attention weights. (The numbers in this figure are truncated to two digits after the decimal point to reduce visual clutter. The values in each row should add up to 1.0 or 100%.)

instead of manually implementing nn.Parameter(torch.rand(...)) is that nn.Linear has an optimized weight initialization scheme, contributing to more stable and effective model training.

Listing 3.2 A self-attention class using PyTorch's Linear layers

```
class SelfAttention_v2(nn.Module):
    def __init__(self,     d_in, d_out, qkv_bias=False):
```

```
        super().__init__()
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key   = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)

    def forward(self, x):
        keys = self.W_key(x)
        queries = self.W_query(x)
        values = self.W_value(x)
        attn_scores = queries @ keys.T
        attn_weights = torch.softmax(
            attn_scores / keys.shape[-1]**0.5, dim=-1
        )
        context_vec = attn_weights @ values
        return context_vec
```

You can use the `SelfAttention_v2` similar to `SelfAttention_v1`:

```
torch.manual_seed(789)
sa_v2 = SelfAttention_v2(d_in, d_out)
print(sa_v2(inputs))
```

The output is

```
tensor([[-0.0739,  0.0713],
        [-0.0748,  0.0703],
        [-0.0749,  0.0702],
        [-0.0760,  0.0685],
        [-0.0763,  0.0679],
        [-0.0754,  0.0693]], grad_fn=<MmBackward0>)
```

Note that `SelfAttention_v1` and `SelfAttention_v2` give different outputs because they use different initial weights for the weight matrices since `nn.Linear` uses a more sophisticated weight initialization scheme.

Exercise 3.1 Comparing SelfAttention_v1 and SelfAttention_v2

Note that `nn.Linear` in `SelfAttention_v2` uses a different weight initialization scheme as `nn.Parameter(torch.rand(d_in, d_out))` used in `SelfAttention_v1`, which causes both mechanisms to produce different results. To check that both implementations, `SelfAttention_v1` and `SelfAttention_v2`, are otherwise similar, we can transfer the weight matrices from a `SelfAttention_v2` object to a `SelfAttention_v1`, such that both objects then produce the same results.

Your task is to correctly assign the weights from an instance of `SelfAttention_v2` to an instance of `SelfAttention_v1`. To do this, you need to understand the relationship between the weights in both versions. (Hint: `nn.Linear` stores the weight matrix in a transposed form.) After the assignment, you should observe that both instances produce the same outputs.

```
        super().__init__()
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
    self.W_key = nn.Linear(d_in, d_out, bias=qkv_bias) self.W_value
    = nn.Linear(d_in, d_out, bias=qkv_bias)


    def forward(self, x):
        keys = self.W_key(x) queries =
    self.W_query(x) values =
    self.W_value(x) attn_scores = queries
    @ keys.T attn_weights =
    torch.softmax(

        attn_scores / keys.shape[-1]**0.5, dim=-1 )
    context_vec = attn_weights @ values return context_vec
```

You can use the SelfAttention_v2 similar to SelfAttention_v1:

```
torch.manual_seed(789) sa_v2 =
SelfAttention_v2(d_in, d_out)
print(sa_v2(inputs))
```

The output is

```
tensor([[-0.0739,     0.0713],
        [-0.0748, 0.0703],
        [-0.0749, 0.0702],
        [-0.0760, 0.0685],
        [-0.0763, 0.0679],
        [-0.0754,    0.0693]], grad_fn=<MmBackward0>)
```

Note that SelfAttention_v1 and SelfAttention_v2 give different outputs because they use different initial weights for the weight matrices since nn.Linear uses a more sophisticated weight initialization scheme.

Exercise 3.1 Comparing SelfAttention_v1 and SelfAttention_v2

Note that nn.Linear in SelfAttention_v2 uses a different weight initialization scheme as nn.Parameter(torch.rand(d_in, d_out)) used in SelfAttention_v1, which causes both mechanisms to produce different results. To check that both implementations, SelfAttention_v1 and SelfAttention_v2, are otherwise similar, we can transfer the weight matrices from a SelfAttention_v2 object to a SelfAttention_v1, such that both objects then produce the same results.

Your task is to correctly assign the weights from an instance of SelfAttention_v2 to an instance of SelfAttention_v1. To do this, you need to understand the relationship between the weights in both versions. (Hint: nn.Linear stores the weight matrix in a transposed form.) After the assignment, you should observe that both instances produce the same outputs.

Next, we will make enhancements to the self-attention mechanism, focusing specifically on incorporating causal and multi-head elements. The causal aspect involves modifying the attention mechanism to prevent the model from accessing future information in the sequence, which is crucial for tasks like language modeling, where each word prediction should only depend on previous words.

The multi-head component involves splitting the attention mechanism into multiple "heads." Each head learns different aspects of the data, allowing the model to simultaneously attend to information from different representation subspaces at different positions. This improves the model's performance in complex tasks.

3.5 Hiding future words with causal attention

For many LLM tasks, you will want the self-attention mechanism to consider only the tokens that appear prior to the current position when predicting the next token in a sequence. Causal attention, also known as *masked attention*, is a specialized form of self-attention. It restricts a model to only consider previous and current inputs in a sequence when processing any given token when computing attention scores. This is in contrast to the standard self-attention mechanism, which allows access to the entire input sequence at once.

Now, we will modify the standard self-attention mechanism to create a *causal attention* mechanism, which is essential for developing an LLM in the subsequent chapters. To achieve this in GPT-like LLMs, for each token processed, we mask out the future tokens, which come after the current token in the input text, as illustrated in figure 3.19. We mask out the attention weights above the diagonal, and we

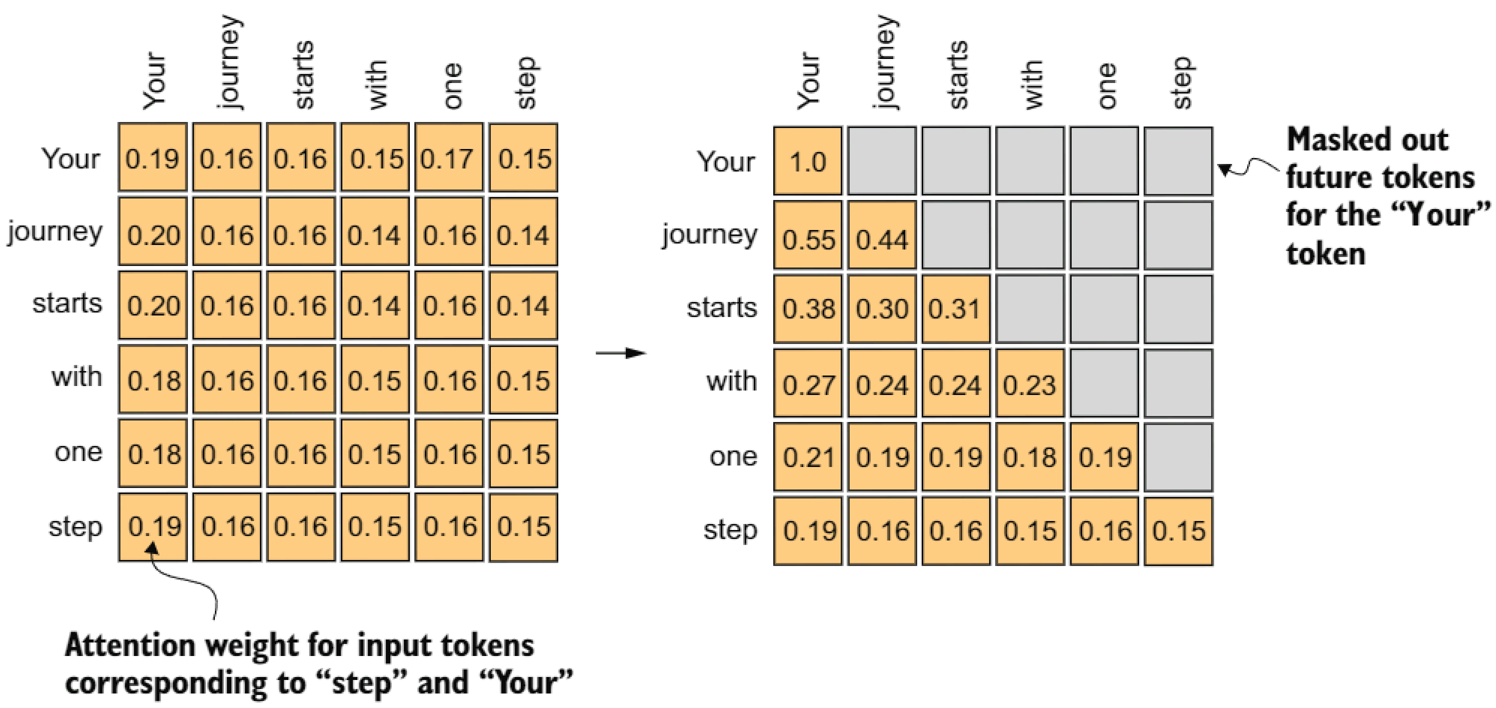

Figure 3.19 In causal attention, we mask out the attention weights above the diagonal such that for a given input, the LLM can't access future tokens when computing the context vectors using the attention weights. For example, for the word "journey" in the second row, we only keep the attention weights for the words before ("Your") and in the current position ("journey").

Next, we will make enhancements to the self-attention mechanism, focusing specifically on incorporating causal and multi-head elements. The causal aspect involves modifying the attention mechanism to prevent the model from accessing future information in the sequence, which is crucial for tasks like language modeling, where each word prediction should only depend on previous words.

The multi-head component involves splitting the attention mechanism into multiple “heads.” Each head learns different aspects of the data, allowing the model to simultaneously attend to information from different representation subspaces at different positions. This improves the model’s performance in complex tasks.

3.5 Hiding future words with causal attention

For many LLM tasks, you will want the self-attention mechanism to consider only the tokens that appear prior to the current position when predicting the next token in a sequence. Causal attention, also known as masked attention, is a specialized form of selfattention. It restricts a model to only consider previous and current inputs in a sequence when processing any given token when computing attention scores. This is in contrast to the standard self-attention mechanism, which allows access to the entire input sequence at once.

Now, we will modify the standard self-attention mechanism to create a causal attention mechanism, which is essential for developing an LLM in the subsequent chapters. To achieve this in GPT-like LLMs, for each token processed, we mask out the future tokens, which come after the current token in the input text, as illustrated in figure 3.19. We mask out the attention weights above the diagonal, and we

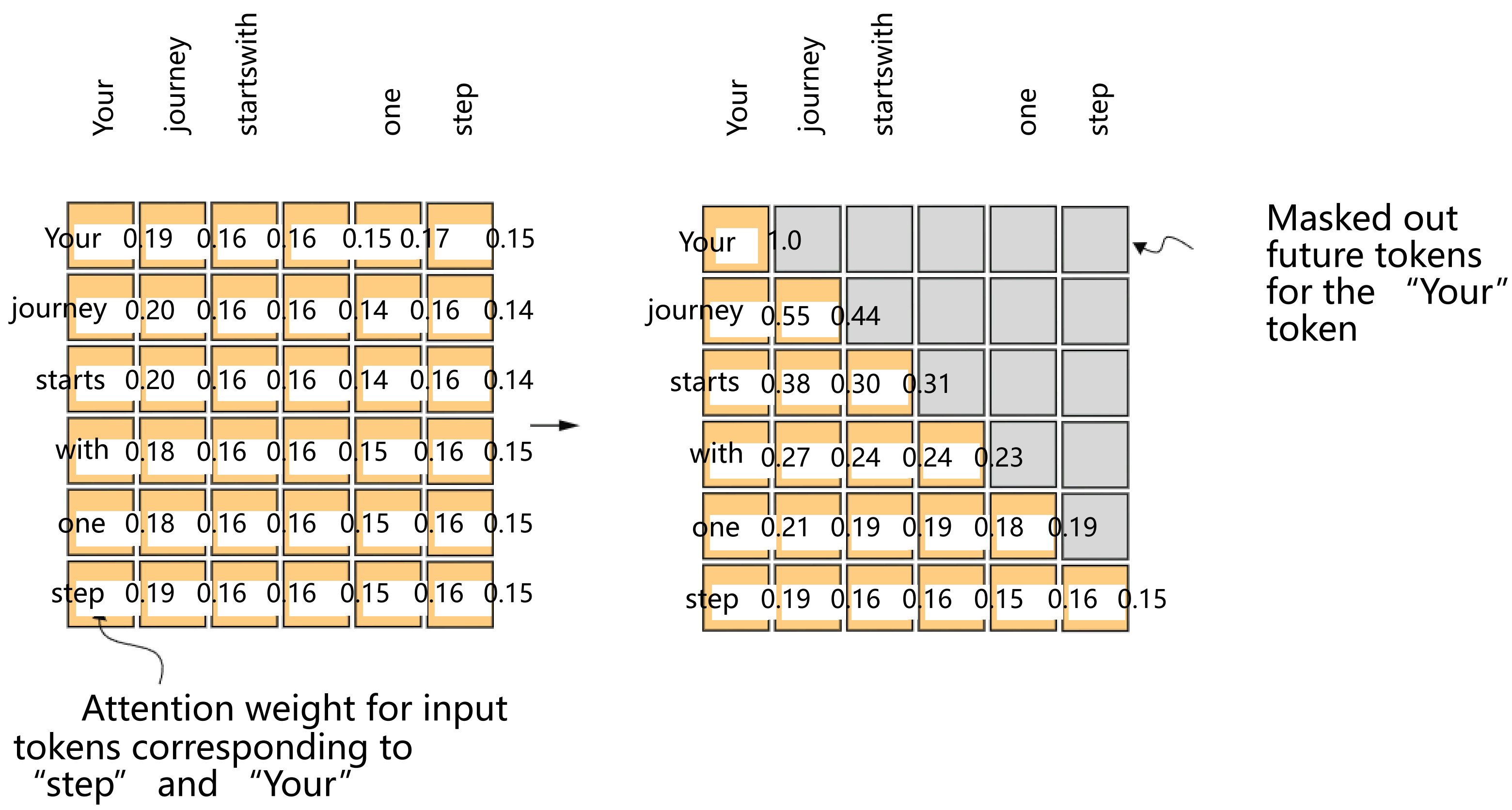

Figure 3.19 In causal attention, we mask out the attention weights above the diagonal such that for a given input, the LLM can’t access future tokens when computing the context vectors using the attention weights. For example, for the word “journey” in the second row, we only keep the attention weights for the words before (“Your”) and in the current position (“journey”).

normalize the nonmasked attention weights such that the attention weights sum to 1 in each row. Later, we will implement this masking and normalization procedure in code.

3.5.1 Applying a causal attention mask

Our next step is to implement the causal attention mask in code. To implement the steps to apply a causal attention mask to obtain the masked attention weights, as summarized in figure 3.20, let's work with the attention scores and weights from the previous section to code the causal attention mechanism.

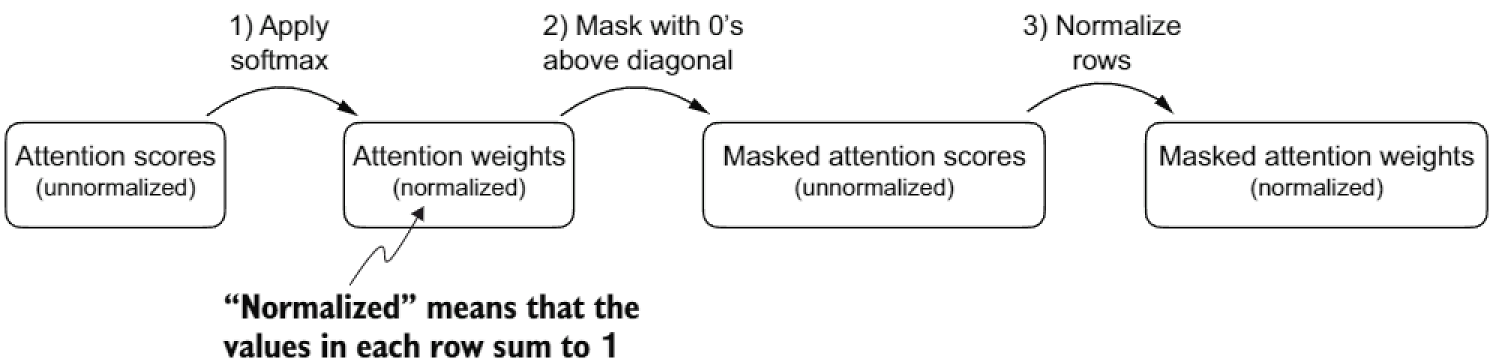

Figure 3.20 One way to obtain the masked attention weight matrix in causal attention is to apply the softmax function to the attention scores, zeroing out the elements above the diagonal and normalizing the resulting matrix.

In the first step, we compute the attention weights using the softmax function as we have done previously:

```
queries = sa_v2.W_query(inputs)
keys = sa_v2.W_key(inputs)
attn_scores = queries @ keys.T
attn_weights = torch.softmax(attn_scores / keys.shape[-1]**0.5, dim=-1)
print(attn_weights)
```

Reuses the query and key weight matrices of the SelfAttention_v2 object from the previous section for convenience

This results in the following attention weights:

```
tensor([[0.1921, 0.1646, 0.1652, 0.1550, 0.1721, 0.1510],
        [0.2041, 0.1659, 0.1662, 0.1496, 0.1665, 0.1477],
        [0.2036, 0.1659, 0.1662, 0.1498, 0.1664, 0.1480],
        [0.1869, 0.1667, 0.1668, 0.1571, 0.1661, 0.1564],
        [0.1830, 0.1669, 0.1670, 0.1588, 0.1658, 0.1585],
        [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
       grad_fn=<SoftmaxBackward0>)
```

We can implement the second step using PyTorch's `tril` function to create a mask where the values above the diagonal are zero:

```
context_length = attn_scores.shape[0]
mask_simple = torch.tril(torch.ones(context_length, context_length))
print(mask_simple)
```

将非掩码的注意力权重归一化，使得每行的注意力权重之和为 1。稍后，我们将用代码实现这个掩码和归一化过程。

3.5.1 应用因果注意力掩码

我们的下一步是在代码中实现因果注意力掩码。为了实现应用因果注意力掩码以获得掩码注意力权重的步骤，如图 3.20 所示，让我们使用上一节中的注意力得分和权重来编写因果注意力机制。

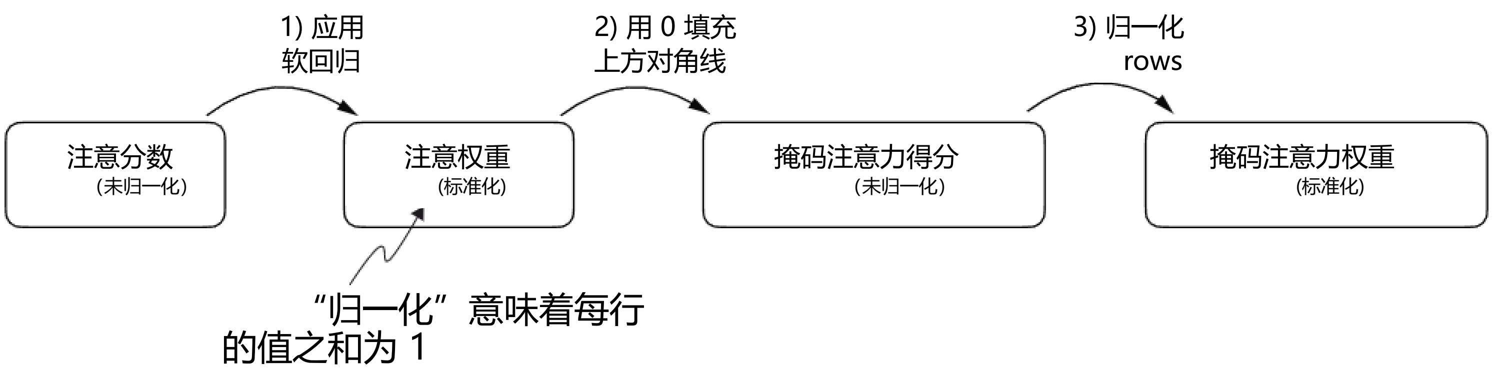

图 3.20 获取因果注意力中掩码注意力权重矩阵的一种方法是将注意力分数应用 softmax 函数，将上三角元素置零，并对得到的矩阵进行归一化。

在第一步，我们使用 softmax 函数计算注意力权重，就像我们之前做的那样：

复用上一节中 SelfAttention_v2 对象的查询和键权重矩阵以方便使用

```
查询 = sa_v2.W_query(inputs)
键 = sa_v2.W_key(inputs)
注意力分数 = 查询 @ 键.T
注意力权重 = torch.softmax(注意力分数 / 键.shape[-1]**0.5, dim=-1)
打印(注意力权重)
```

这导致以下注意力权重：

```
张量([[0.1921,       [0.1646, 0.1652, 0.1550, 0.1721, 0.1510]
         [0.2041, 0.1659, 0.1662, 0.1496, 0.1665, 0.1477],
    [0.2036, 0.1659, 0.1662, 0.1498, 0.1664, 0.1480], [0.1869,
    0.1667, 0.1668, 0.1571, 0.1661, 0.1564], [0.1830, 0.1669,
    0.1670, 0.1588, 0.1658, 0.1585], [0.1935, 0.1663, 0.1666,
    0.1542, 0.1666, 0.1529]], grad_fn=)
```

我们可以使用 PyTorch 的 tril 函数来实现第二步，创建一个对角线上方值为零的掩码：

```
context_length = attn_scores.shape[0] mask_simple =
torch.tril(torch.ones(context_length, context_length)) 打印(mask_simple)
```

The resulting mask is

```
tensor([[1., 0., 0., 0., 0., 0.],
        [1., 1., 0., 0., 0., 0.],
        [1., 1., 1., 0., 0., 0.],
        [1., 1., 1., 1., 0., 0.],
        [1., 1., 1., 1., 1., 0.],
        [1., 1., 1., 1., 1., 1.]])
```

Now, we can multiply this mask with the attention weights to zero-out the values above the diagonal:

```
masked_simple = attn_weights*mask_simple
print(masked_simple)
```

As we can see, the elements above the diagonal are successfully zeroed out:

```
tensor([[0.1921, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.2041, 0.1659, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.2036, 0.1659, 0.1662, 0.0000, 0.0000, 0.0000],
        [0.1869, 0.1667, 0.1668, 0.1571, 0.0000, 0.0000],
        [0.1830, 0.1669, 0.1670, 0.1588, 0.1658, 0.0000],
        [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
       grad_fn=<MulBackward0>)
```

The third step is to renormalize the attention weights to sum up to 1 again in each row. We can achieve this by dividing each element in each row by the sum in each row:

```
row_sums = masked_simple.sum(dim=-1, keepdim=True)
masked_simple_norm = masked_simple / row_sums
print(masked_simple_norm)
```

The result is an attention weight matrix where the attention weights above the diagonal are zeroed-out, and the rows sum to 1:

```
tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.5517, 0.4483, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.3800, 0.3097, 0.3103, 0.0000, 0.0000, 0.0000],
        [0.2758, 0.2460, 0.2462, 0.2319, 0.0000, 0.0000],
        [0.2175, 0.1983, 0.1984, 0.1888, 0.1971, 0.0000],
        [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
       grad_fn=<DivBackward0>)
```

Information leakage

When we apply a mask and then renormalize the attention weights, it might initially appear that information from future tokens (which we intend to mask) could still influence the current token because their values are part of the softmax calculation. However, the key insight is that when we renormalize the attention weights after masking,

生成的掩码是

```
context_length)) 打印(mask_simple) 矩阵[[1., 0., 0., 0., [0., 0.,
          [1., 1., 0., 0., 0., 0.],
      [1., 1., 1., 0., 0., 0.], [1.,
      1., 1., 1., 0., 0.], [1., 1.,
      1., 1., 1., 0.], [1., 1., 1.,
      1., 1., 1.]
                              1., 1.]])
```

现在，我们可以将这个掩码与注意力权重相乘，以将上三角线以上的值置零：

```
    masked_simple = attn_weights * mask_simple 打
印(masked_simple)
```

如您所见，对角线以上的元素已成功置零：

```
张量([[0.1921, 0.0000,         0.0000, 0.0000, 0.0000,       0.0000],
        [0.2041, 0.1659, 0.0000, 0.0000, 0.0000, 0.0000],
    [0.2036, 0.1659, 0.1662, 0.0000, 0.0000, 0.0000], [0.1869,
    0.1667, 0.1668, 0.1571, 0.0000, 0.0000], [0.1830, 0.1669,
    0.1670, 0.1588, 0.1658, 0.0000], [0.1935, 0.1663, 0.1666,
    0.1542, 0.1666, 0.1529]], grad_fn=)
```

第三步是将注意力权重重新归一化，使每行的总和再次为 1。我们可以通过将每行中的每个元素除以该行的总和来实现这一点：

```
    row_sums = 遮蔽简单求和(dim=-1, 保持 dim=True) 遮蔽简单归
一化 = 遮蔽简单 / 行求和 print(遮蔽简单归一化)
```

结果是一个注意力权重矩阵，其中对角线以上的注意力权重被置零，并且行和为 1：

```
张量([[1.0000, 0.0000,         0.0000, 0.0000, 0.0000,       0.0000],
        [0.5517, 0.4483, 0.0000, 0.0000, 0.0000, 0.0000],
    [0.3800, 0.3097, 0.3103, 0.0000, 0.0000, 0.0000], [0.2758,
    0.2460, 0.2462, 0.2319, 0.0000, 0.0000], [0.2175, 0.1983,
    0.1984, 0.1888, 0.1971, 0.0000], [0.1935, 0.1663, 0.1666,
    0.1542, 0.1666, 0.1529]], grad_fn=)
```

信息泄露

当我们应用掩码然后重新归一化注意力权重时，最初可能看起来未来标记（我们打算掩码的）的信息仍然可能影响当前标记，因为它们的值是 softmax 计算的一部分。然而，关键洞察是，当我们掩码后重新归一化注意力权重时，

> what we're essentially doing is recalculating the softmax over a smaller subset (since masked positions don't contribute to the softmax value).
>
> The mathematical elegance of softmax is that despite initially including all positions in the denominator, after masking and renormalizing, the effect of the masked positions is nullified—they don't contribute to the softmax score in any meaningful way.
>
> In simpler terms, after masking and renormalization, the distribution of attention weights is as if it was calculated only among the unmasked positions to begin with. This ensures there's no information leakage from future (or otherwise masked) tokens as we intended.

While we could wrap up our implementation of causal attention at this point, we can still improve it. Let's take a mathematical property of the softmax function and implement the computation of the masked attention weights more efficiently in fewer steps, as shown in figure 3.21.

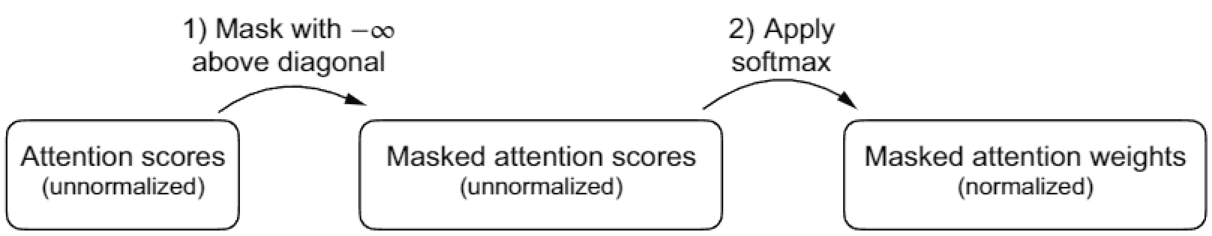

Figure 3.21 A more efficient way to obtain the masked attention weight matrix in causal attention is to mask the attention scores with negative infinity values before applying the softmax function.

The softmax function converts its inputs into a probability distribution. When negative infinity values ($-\infty$) are present in a row, the softmax function treats them as zero probability. (Mathematically, this is because $e^{-\infty}$ approaches 0.)

We can implement this more efficient masking "trick" by creating a mask with 1s above the diagonal and then replacing these 1s with negative infinity (`-inf`) values:

```
mask = torch.triu(torch.ones(context_length, context_length), diagonal=1)
masked = attn_scores.masked_fill(mask.bool(), -torch.inf)
print(masked)
```

This results in the following mask:

```
tensor([[0.2899,   -inf,   -inf,   -inf,   -inf,   -inf],
        [0.4656, 0.1723,   -inf,   -inf,   -inf,   -inf],
        [0.4594, 0.1703, 0.1731,   -inf,   -inf,   -inf],
        [0.2642, 0.1024, 0.1036, 0.0186,   -inf,   -inf],
        [0.2183, 0.0874, 0.0882, 0.0177, 0.0786,   -inf],
        [0.3408, 0.1270, 0.1290, 0.0198, 0.1290, 0.0078]],
       grad_fn=<MaskedFillBackward0>)
```

我们在本质上是在一个更小的子集上重新计算 softmax（因为掩码位置不贡献于 softmax 值）。

软最大化数学上的优雅之处在于，尽管最初在分母中包含所有位置，但在掩码和重新归一化之后，掩码位置的影响被消除——它们以任何有意义的方式都不会对软最大化得分做出贡献。

在更简单的说法中，在掩码和归一化之后，注意力权重的分布就像一开始只计算未掩码的位置一样。这确保了未来（或其他掩码）的标记不会泄露信息，正如我们所期望的。

虽然我们可以在这一点上完成因果注意力的实现，但我们仍然可以改进它。让我们采用 softmax 函数的一个数学属性，并在更少的步骤中更有效地计算掩码注意力权重，如图 3.21 所示。

图 3.21 获取因果注意力中掩码注意力权重矩阵的一种更有效的方法是在应用 softmax 函数之前，用负无穷大值掩码注意力得分。

软最大化函数将其输入转换为概率分布。当一行中出现负无穷大值（-∞）时，软最大化函数将其视为零概率。（从数学上讲，这是因为 e 趋近于 0。）我们可以通过创建一个对角线以上的 1s 组成的掩码，然后将这些 1s 替换为负无穷大（-inf）值来实现这个更高效的“技巧”：

```
mask = torch.triu(torch.ones(上下文长度, 上下文长度), 对角线=1) masked =
attn_scores.masked_fill(mask.bool(), -torch.inf) print(masked)
```

这导致以下掩码：

```
张量([[0.2899,          - 无效，或表示负无穷大
        [0.4656, 0.1723]        - 无效，或表示负无穷大
        [0.4594, 0.1703, 0.1731]        - 无效，或表示负无穷大
        [0.2642, 0.1024, 0.1036, 0.0186]        - 无效，或表示负无穷大
        [0.2183, 0.0874, 0.0882, 0.0177, 0.0786]        [-inf],
       [0.3408, 0.1270, 0.1290, 0.0198, 0.1290, 0.0078]],
    grad_fn=)
```

Now all we need to do is apply the softmax function to these masked results, and we are done:

```
attn_weights = torch.softmax(masked / keys.shape[-1]**0.5, dim=1)
print(attn_weights)
```

As we can see based on the output, the values in each row sum to 1, and no further normalization is necessary:

```
tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.5517, 0.4483, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.3800, 0.3097, 0.3103, 0.0000, 0.0000, 0.0000],
        [0.2758, 0.2460, 0.2462, 0.2319, 0.0000, 0.0000],
        [0.2175, 0.1983, 0.1984, 0.1888, 0.1971, 0.0000],
        [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
       grad_fn=<SoftmaxBackward0>)
```

We could now use the modified attention weights to compute the context vectors via `context_vec = attn_weights @ values`, as in section 3.4. However, we will first cover another minor tweak to the causal attention mechanism that is useful for reducing overfitting when training LLMs.

3.5.2 Masking additional attention weights with dropout

Dropout in deep learning is a technique where randomly selected hidden layer units are ignored during training, effectively "dropping" them out. This method helps prevent overfitting by ensuring that a model does not become overly reliant on any specific set of hidden layer units. It's important to emphasize that dropout is only used during training and is disabled afterward.

In the transformer architecture, including models like GPT, dropout in the attention mechanism is typically applied at two specific times: after calculating the attention weights or after applying the attention weights to the value vectors. Here we will apply the dropout mask after computing the attention weights, as illustrated in figure 3.22, because it's the more common variant in practice.

In the following code example, we use a dropout rate of 50%, which means masking out half of the attention weights. (When we train the GPT model in later chapters, we will use a lower dropout rate, such as 0.1 or 0.2.) We apply PyTorch's dropout implementation first to a 6×6 tensor consisting of 1s for simplicity:

```
torch.manual_seed(123)
dropout = torch.nn.Dropout(0.5)
example = torch.ones(6, 6)
print(dropout(example))
```

We choose a dropout rate of 50%.

Here, we create a matrix of 1s.

现在我们只需要将这些掩码结果应用 softmax 函数，就完成了

```
attn_weights = torch.softmax(masked / keys.shape[-1]**0.5, dim=1) 打印
(attn_weights)
```

根据输出我们可以看到，每行的值之和为 1，无需进一步归一化：

```
张量([[1.0000, 0.0000,         0.0000, 0.0000, 0.0000,       0.0000],
        [0.5517, 0.4483, 0.0000, 0.0000, 0.0000, 0.0000],
    [0.3800, 0.3097, 0.3103, 0.0000, 0.0000, 0.0000], [0.2758,
    0.2460, 0.2462, 0.2319, 0.0000, 0.0000], [0.2175, 0.1983,
    0.1984, 0.1888, 0.1971, 0.0000], [0.1935, 0.1663, 0.1666,
    0.1542, 0.1666, 0.1529]], grad_fn=)
```

我们现在可以使用修改后的注意力权重来通过 context_vec = attn_weights @ values 计算上下文向量，如第 3.4 节所述。然而，我们首先将介绍对因果注意力机制的一个小调整，这对于在训练时减少过拟合是有用的。

3.5.2 掩码额外的注意力权重与 dropout

深度学习中 dropout 是一种技术，在训练过程中随机选择隐藏层单元被忽略，实际上“丢弃”它们。这种方法通过确保模型不会过度依赖任何特定的隐藏层单元集来帮助防止过拟合。重要的是强调 dropout 仅在训练期间使用，之后被禁用。

在 Transformer 架构中，包括 GPT 等模型，通常在两个特定时间应用注意力机制中的 dropout：在计算注意力权重之后或应用注意力权重到值向量之后。在这里，我们将计算注意力权重后应用 dropout 掩码，如图 3.22 所示，因为在实践中这是更常见的变体。

在下面的代码示例中，我们使用 50%的丢弃率，这意味着屏蔽掉一半的注意力权重。（当我们后面章节训练 GPT 模型时，我们将使用较低的丢弃率，例如 0.1 或 0.2。）我们首先将 PyTorch 的丢弃实现应用于一个由 1 组成的 6×6 张量，以简化操作：

```
torch.manual_seed(123) dropout =
torch.nn.Dropout(0.5) example =
torch.ones(6, 6) 打印
(dropout(example))
```

我们选择 50%的丢弃率。

这里，我们创建一个全为 1 的矩阵。

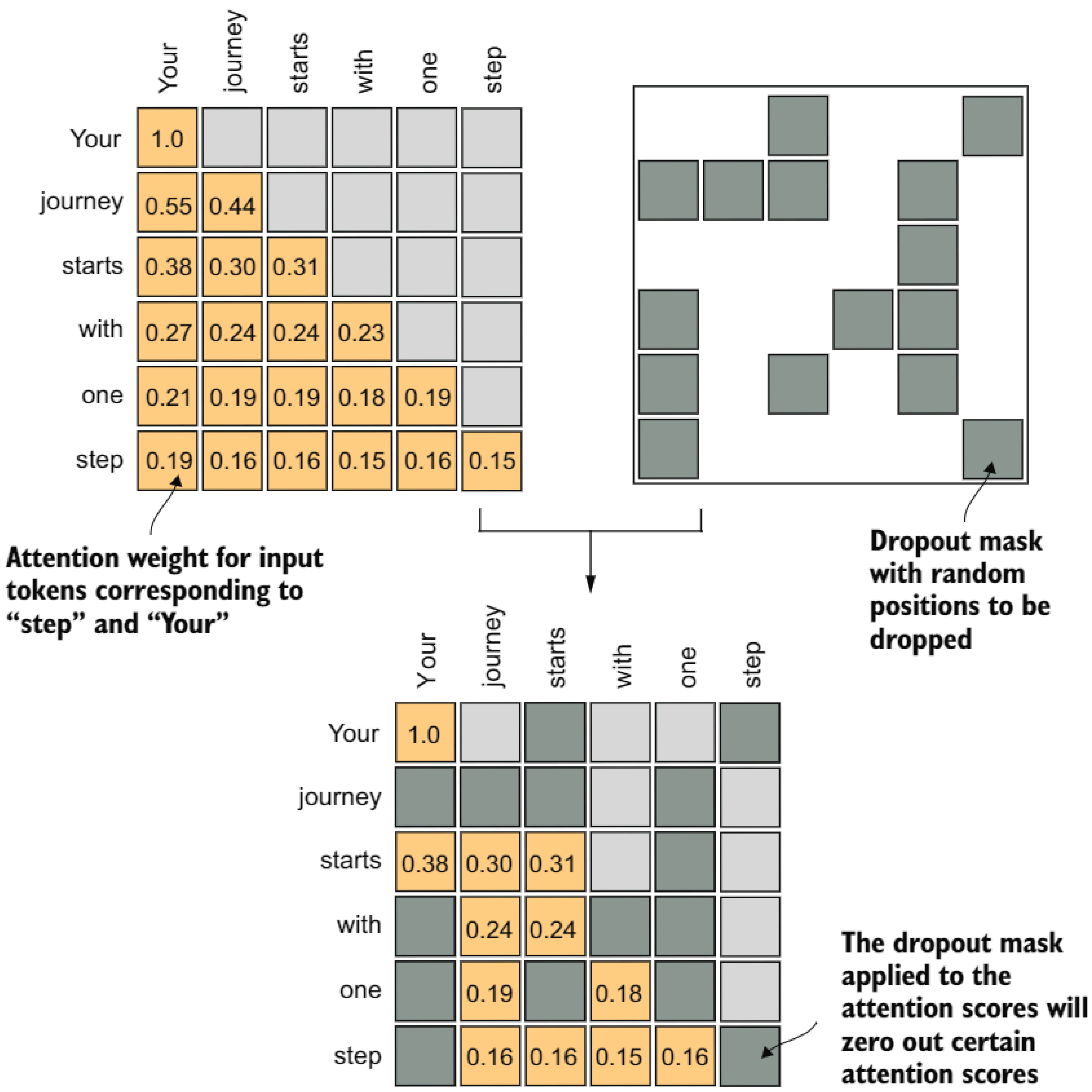

Figure 3.22 Using the causal attention mask (upper left), we apply an additional dropout mask (upper right) to zero out additional attention weights to reduce overfitting during training.

As we can see, approximately half of the values are zeroed out:

```
tensor([[2., 2., 0., 2., 2., 0.],
        [0., 0., 0., 2., 0., 2.],
        [2., 2., 2., 2., 0., 2.],
        [0., 2., 2., 0., 0., 2.],
        [0., 2., 0., 2., 0., 2.],
        [0., 2., 2., 2., 2., 0.]])
```

When applying dropout to an attention weight matrix with a rate of 50%, half of the elements in the matrix are randomly set to zero. To compensate for the reduction in active elements, the values of the remaining elements in the matrix are scaled up by a factor of 1/0.5 = 2. This scaling is crucial to maintain the overall balance of the atten-

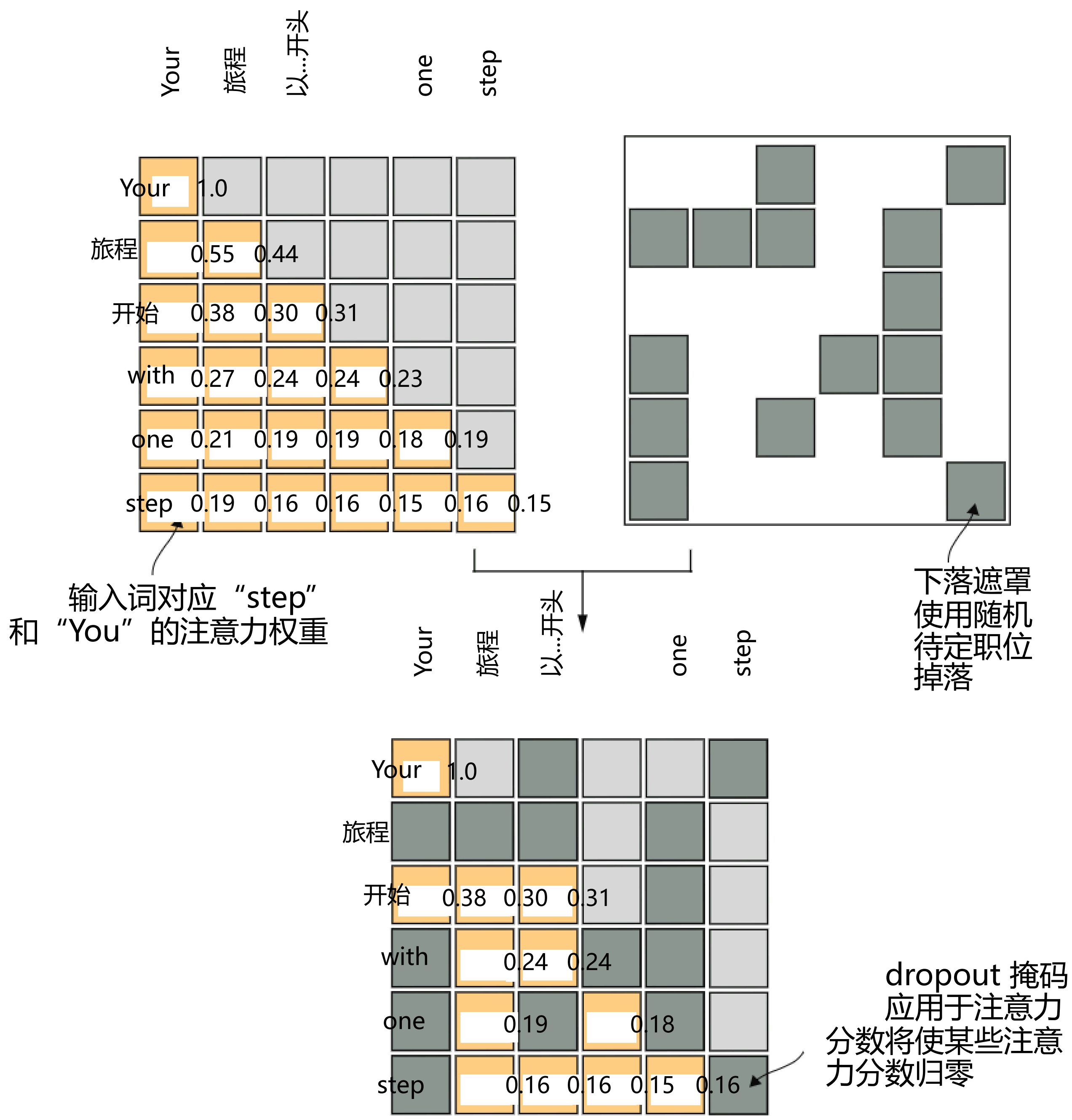

图 3.22 使用因果注意力掩码（左上角），我们在右上角应用额外的 dropout 掩码以零化额外的注意力权重，以减少训练过程中的过拟合。

我们可以看到，大约一半的值被置零：

```
张量([[2., 2.,       0.,  [2., 2., 0.],
          [0., 0., 0., 2., 0., 2.],
      [2., 2., 2., 2., 0., 2.], [0.,
      2., 2., 0., 0., 2.], [0., 2.,
      0., 2., 0., 2.], [0., 2., 2.,]

                         2., 2., 0.]])
```

当以 50%的比率应用 dropout 到注意力权重矩阵时，矩阵中一半的元素被随机设置为 0。为了补偿活跃元素数量的减少，矩阵中剩余元素的价值按 1/0.5=2 的因子放大。这种缩放对于保持整体平衡至关重要。

tion weights, ensuring that the average influence of the attention mechanism remains consistent during both the training and inference phases.

Now let's apply dropout to the attention weight matrix itself:

```
torch.manual_seed(123)
print(dropout(attn_weights))
```

The resulting attention weight matrix now has additional elements zeroed out and the remaining 1s rescaled:

```
tensor([[2.0000, 0.0000, 0 .0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.7599, 0.6194, 0.6206, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.4921, 0.4925, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.3966, 0.0000, 0.3775, 0.0000, 0.0000],
        [0.0000, 0.3327, 0.3331, 0.3084, 0.3331, 0.0000]],
       grad_fn=<MulBackward0>
```

Note that the resulting dropout outputs may look different depending on your operating system; you can read more about this inconsistency here on the PyTorch issue tracker at https://github.com/pytorch/pytorch/issues/121595.

Having gained an understanding of causal attention and dropout masking, we can now develop a concise Python class. This class is designed to facilitate the efficient application of these two techniques.

3.5.3 Implementing a compact causal attention class

We will now incorporate the causal attention and dropout modifications into the `SelfAttention` Python class we developed in section 3.4. This class will then serve as a template for developing *multi-head attention*, which is the final attention class we will implement.

But before we begin, let's ensure that the code can handle batches consisting of more than one input so that the `CausalAttention` class supports the batch outputs produced by the data loader we implemented in chapter 2.

For simplicity, to simulate such batch inputs, we duplicate the input text example:

```
batch = torch.stack((inputs, inputs), dim=0)
print(batch.shape)
```

Two inputs with six tokens each; each token has embedding dimension 3.

This results in a three-dimensional tensor consisting of two input texts with six tokens each, where each token is a three-dimensional embedding vector:

```
torch.Size([2, 6, 3])
```

The following `CausalAttention` class is similar to the `SelfAttention` class we implemented earlier, except that we added the dropout and causal mask components.

此缩放对于保持注意力权重的整体平衡至关重要，确保注意力机制在训练和推理阶段的影响保持一致。

现在让我们将 dropout 应用于注意力权重矩阵本身：

```
    torch 手动设置随机种子(123) 打
印 dropout(attn_weights)
```

结果注意力权重矩阵现在有额外的元素被置零，剩余的 1 被重新缩放：

```
张量([[2.0000, 0.0000,        0 .0000, 0.0000,     [0.0000, 0.0000]
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
     [0.7599, 0.6194, 0.6206, 0.0000, 0.0000, 0.0000], [0.0000,
     0.4921, 0.4925, 0.0000, 0.0000, 0.0000], [0.0000, 0.3966,
     0.0000, 0.3775, 0.0000, 0.0000], [0.0000, 0.3327, 0.3331,
     0.3084, 0.3331, 0.0000]], grad_fn=
```

请注意，由于操作系统不同，生成的 dropout 输出可能看起来不同；您可以在 PyTorch 问题跟踪器中了解更多关于这种不一致的信息，链接为 https://github.com/pytorch/pytorch/issues/121595。

理解了因果注意力机制和 dropout 掩码后，我们现在可以开发一个简洁的 Python 类。这个类旨在促进这两种技术的有效应用。

3.5.3 实现紧凑因果注意力类

我们现在将因果注意力机制和 dropout 修改整合到我们在 3.4 节中开发的 SelfAttention Python 类中。这个类将作为开发多头注意力的模板，这是我们最终要实现的注意力类。

但在我们开始之前，让我们确保代码可以处理包含多个输入的批次，以便 CausalAttention 类支持我们在第 2 章中实现的数据加载器产生的批量输出。

为了简化，为了模拟此类批量输入，我们复制输入文本示例：

```
    batch = torch.stack((inputs, inputs), dim=0) 打印
(batch.shape)
```

两个输入，每个输入包含六个标记；每个标记的嵌入维度为 3。

这导致了一个由两个输入文本组成的三维张量，每个文本包含六个标记，每个标记是一个三维嵌入向量：

```
    torch.Size([2, 6,])   3])
```

以下 CausalAttention 类与我们之前实现的 SelfAttention 类相似，除了我们添加了 dropout 和因果掩码组件。

Listing 3.3 A compact causal attention class

```
class CausalAttention(nn.Module):
    def __init__(self, d_in, d_out, context_length,
                dropout, qkv_bias=False):
        super().__init__()
        self.d_out = d_out
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key   = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.dropout = nn.Dropout(dropout)
        self.register_buffer(
           'mask',
           torch.triu(torch.ones(context_length, context_length),
           diagonal=1)
        )

    def forward(self, x):
        b, num_tokens, d_in = x.shape
        keys = self.W_key(x)
        queries = self.W_query(x)
        values = self.W_value(x)

        attn_scores = queries @ keys.transpose(1, 2)
        attn_scores.masked_fill_(
            self.mask.bool()[:num_tokens, :num_tokens], -torch.inf)
        attn_weights = torch.softmax(
            attn_scores / keys.shape[-1]**0.5, dim=-1
        )
        attn_weights = self.dropout(attn_weights)

        context_vec = attn_weights @ values
        return context_vec
```

Compared to the previous SelfAttention_v1 class, we added a dropout layer.

The register_buffer call is also a new addition (more information is provided in the following text).

We transpose dimensions 1 and 2, keeping the batch dimension at the first position (0).

In PyTorch, operations with a trailing underscore are performed in-place, avoiding unnecessary memory copies.

While all added code lines should be familiar at this point, we now added a `self.register_buffer()` call in the `__init__` method. The use of `register_buffer` in PyTorch is not strictly necessary for all use cases but offers several advantages here. For instance, when we use the `CausalAttention` class in our LLM, buffers are automatically moved to the appropriate device (CPU or GPU) along with our model, which will be relevant when training our LLM. This means we don't need to manually ensure these tensors are on the same device as your model parameters, avoiding device mismatch errors.

We can use the `CausalAttention` class as follows, similar to `SelfAttention` previously:

```
torch.manual_seed(123)
context_length = batch.shape[1]
ca = CausalAttention(d_in, d_out, context_length, 0.0)
context_vecs = ca(batch)
print("context_vecs.shape:", context_vecs.shape)
```

列表 3.3 紧凑因果注意力类

```
class CausalAttention(nn.Module):
类因果注意力(nn.模块): def __init__(self, 输入维度, 输出维度, context_length, dropout, qkv_bias=False): super().__init__()
        self.d_out = d_out self.W_query = nn.Linear(d_in, d_out,
        bias=qkv_bias) self.W_key = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias) self.dropout =
        nn.Dropout(dropout) self.register_buffer('mask',
        torch.triu(torch.ones(context_length, context_length), diagonal=1))
```

与比较
到上一行
SelfAttention_v1
class, 我们添加了
dropout 层。

寄存器缓冲区调用也是一个新增功能（更多详细信息见下文）。

```
    def forward(self, x): # 定义前向传播函数
        b, num_tokens, d_in = x.shape
        keys = self.W_key(x) queries =
        self.W_query(x) values =
        self.W_value(x)

        attn_scores = queries @ keys.transpose(1, 2)
        attn_scores.masked_fill_(
            self.mask.bool()[:num_tokens, :num_tokens], -torch.inf) attn_weights
        = torch.softmax(
            注意分数 / keys.shape[-1]**0.5, dim=-1 ) 注意权重 =
        self.dropout(注意权重)

        context_vec = attn_weights @ values 返
        回 context_vec
```

我们转置
维度 1 和 2
保持批维度在第
一个位置（0）。

在 PyTorch 中，带
有尾随下划线的操作是
在原地执行的，避免了
不必要的内存复制。

尽管所有新增的代码行在此阶段都应该是熟悉的，但我们现在增加了一个自 .register_buffer() 在 __init__ 方法中的调用。在 PyTorch 中，register_buffer 并非所有用例都严格必要，但在此处提供了几个优点。例如，当我们使用 CausalAttention 类时，缓冲区会自动与我们的模型（CPU 或 GPU）一起移动到适当的设备，这在训练 LLM 时将是有意义的。这意味着我们不需要手动确保这些张量与模型参数位于同一设备上，从而避免了设备不匹配错误。

我们可以使用 因果注意力类如下，类似于自注意力之前：

```
torch 手动设置随机种子(123) context_length = batch.shape[1] ca
= 因果注意力(d_in, d_out, context_length, 0.0) context_vecs =
ca(batch) 打印("context_vecs.shape:", context_vecs.shape)
```

The resulting context vector is a three-dimensional tensor where each token is now represented by a two-dimensional embedding:

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

Figure 3.23 summarizes what we have accomplished so far. We have focused on the concept and implementation of causal attention in neural networks. Next, we will expand on this concept and implement a multi-head attention module that implements several causal attention mechanisms in parallel.

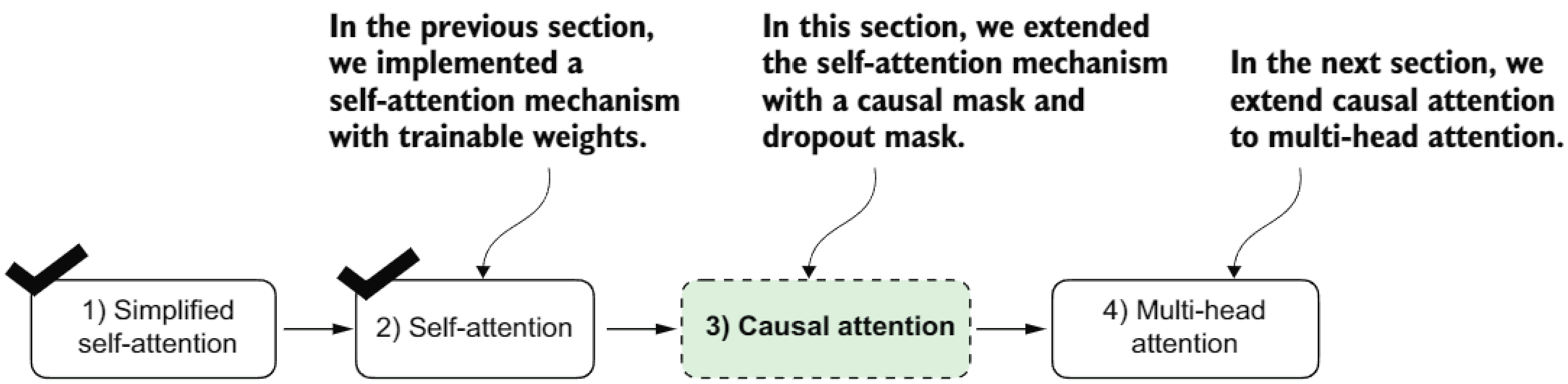

Figure 3.23 Here's what we've done so far. We began with a simplified attention mechanism, added trainable weights, and then added a causal attention mask. Next, we will extend the causal attention mechanism and code multi-head attention, which we will use in our LLM.

3.6 Extending single-head attention to multi-head attention

Our final step will be to extend the previously implemented causal attention class over multiple heads. This is also called *multi-head attention*.

The term "multi-head" refers to dividing the attention mechanism into multiple "heads," each operating independently. In this context, a single causal attention module can be considered single-head attention, where there is only one set of attention weights processing the input sequentially.

We will tackle this expansion from causal attention to multi-head attention. First, we will intuitively build a multi-head attention module by stacking multiple `CausalAttention` modules. Then we will then implement the same multi-head attention module in a more complicated but more computationally efficient way.

3.6.1 Stacking multiple single-head attention layers

In practical terms, implementing multi-head attention involves creating multiple instances of the self-attention mechanism (see figure 3.18), each with its own weights, and then combining their outputs. Using multiple instances of the self-attention mechanism can be computationally intensive, but it's crucial for the kind of complex pattern recognition that models like transformer-based LLMs are known for.

结果上下文向量是一个三维张量，其中每个标记现在由一个二维嵌入表示：

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

图 3.23 总结了我们迄今为止所取得的成果。我们专注于神经网络中因果注意力的概念和实现。接下来，我们将扩展这一概念并实现一个多头注意力模块，该模块并行实现多个因果注意力机制。

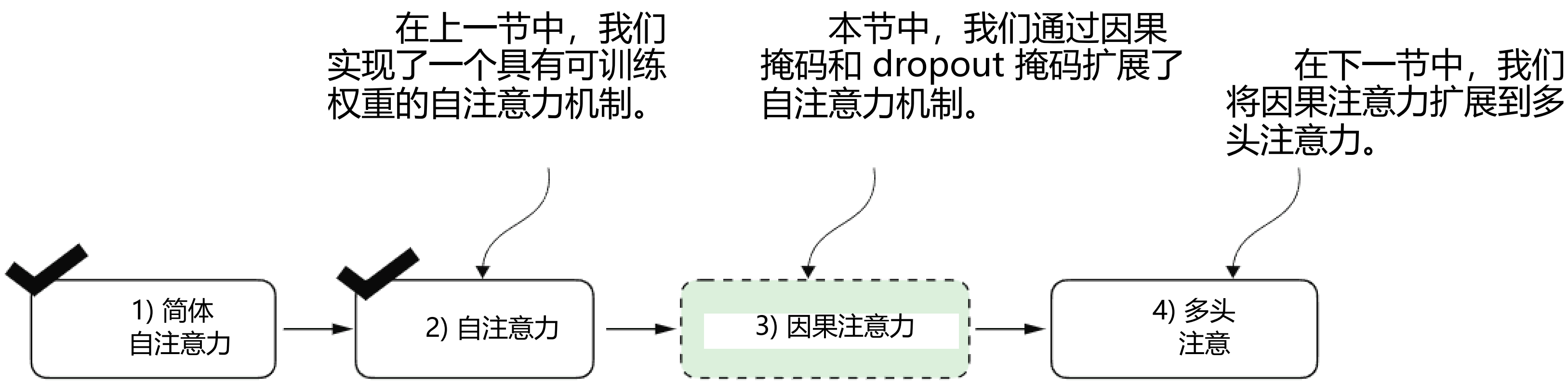

图 3.23 到目前为止我们已经做了什么。我们从一个简化的注意力机制开始，添加了可训练的权重，然后添加了因果注意力掩码。接下来，我们将扩展因果注意力机制和代码多头注意力，我们将在我们的LLM中使用。

3.6 扩展单头注意力到多头注意力

我们的最终步骤将是将之前实现的因果注意力类扩展到多个头。这也被称为多头注意力。

“多头”这个术语指的是将注意力机制划分为多个“头”，每个头独立运行。在这种情况下，一个单一因果注意力模块可以被视为单头注意力，其中只有一个注意力权重集依次处理输入。

我们将从因果注意力扩展到多头注意力。首先，我们将通过堆叠多个因果注意力模块直观地构建一个多头注意力模块。然后，我们将以更复杂但更计算高效的方式实现相同的多头注意力模块。

3.6.1 堆叠多个单头注意力层

在实用层面上，实现多头注意力机制涉及创建多个自注意力机制的实例（见图 3.18），每个实例都有自己的权重，然后将它们的输出组合起来。使用多个自注意力机制的实例可能会计算密集，但对于像基于 transformer 的LLMs这样的模型所擅长的复杂模式识别来说，这是至关重要的。

Figure 3.24 illustrates the structure of a multi-head attention module, which consists of multiple single-head attention modules, as previously depicted in figure 3.18, stacked on top of each other.

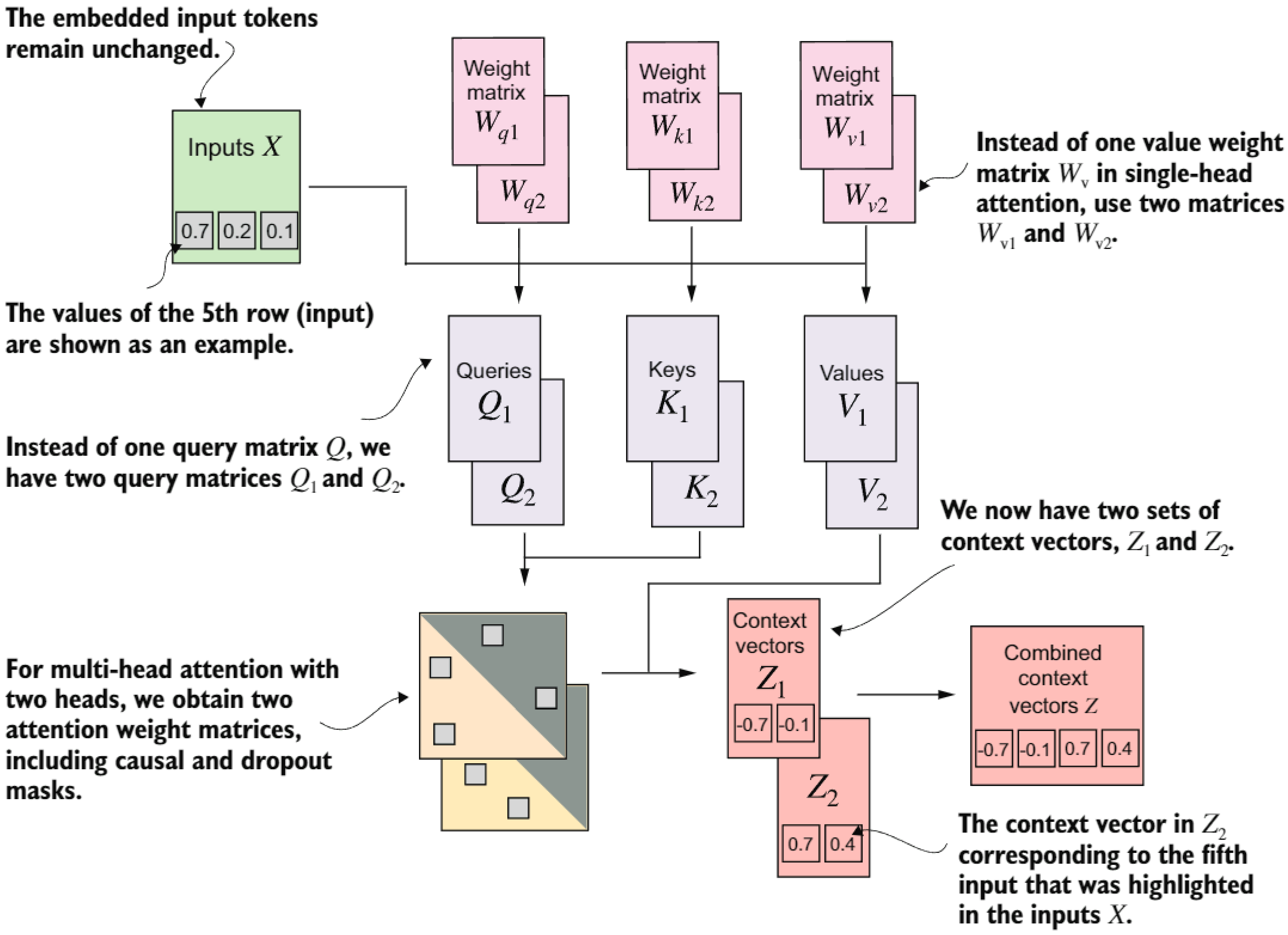

Figure 3.24 The multi-head attention module includes two single-head attention modules stacked on top of each other. So, instead of using a single matrix W_v for computing the value matrices, in a multi-head attention module with two heads, we now have two value weight matrices: W_{v1} and W_{v2}. The same applies to the other weight matrices, W_Q and W_k. We obtain two sets of context vectors Z_1 and Z_2 that we can combine into a single context vector matrix Z.

As mentioned before, the main idea behind multi-head attention is to run the attention mechanism multiple times (in parallel) with different, learned linear projections—the results of multiplying the input data (like the query, key, and value vectors in attention mechanisms) by a weight matrix. In code, we can achieve this by implementing a simple `MultiHeadAttentionWrapper` class that stacks multiple instances of our previously implemented `CausalAttention` module.

图 3.24 展示了多头注意力模块的结构，该模块由多个单头注意力模块组成，如之前图 3.18 中所示，这些模块层层堆叠。

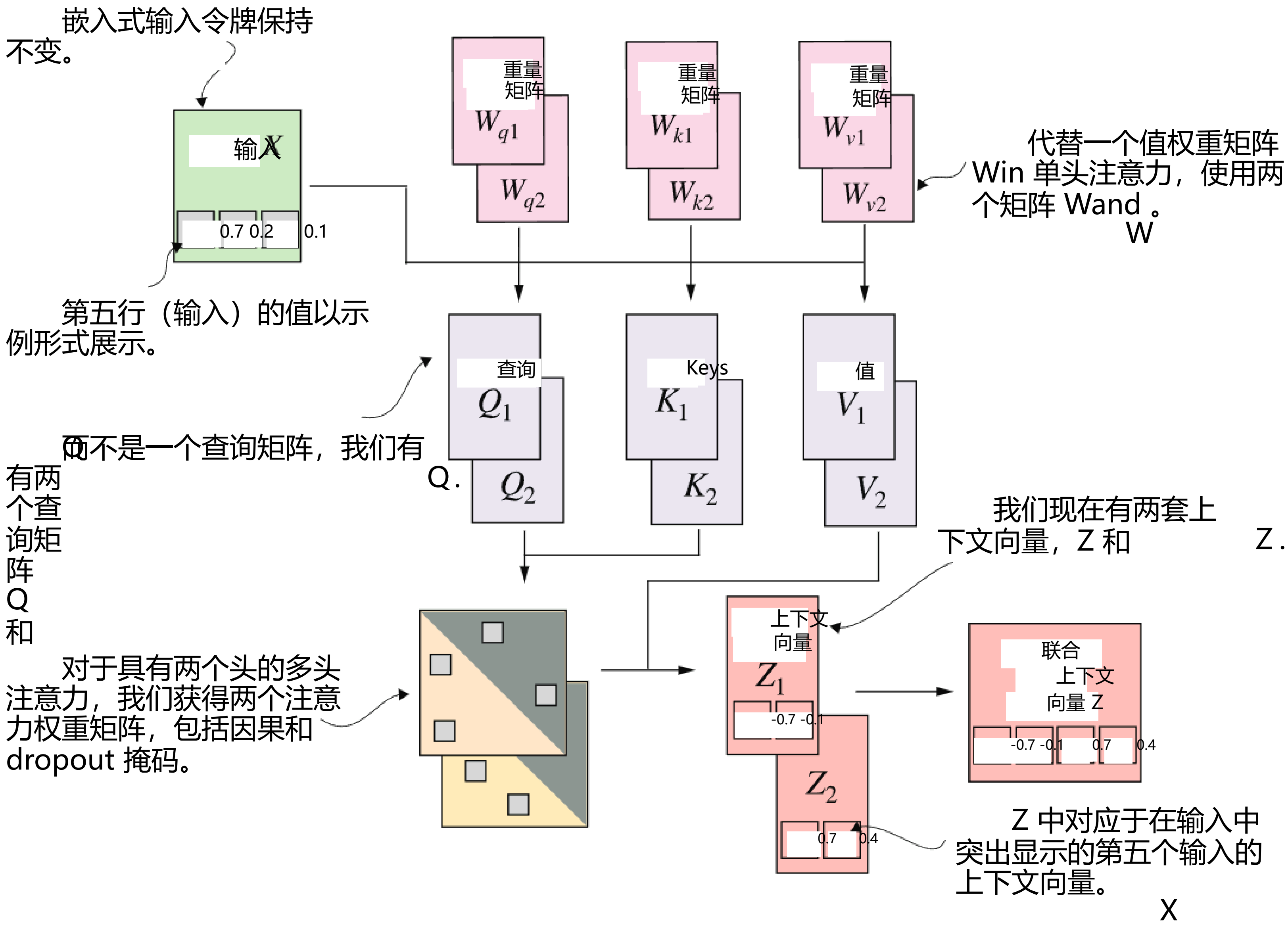

图 3.24 多头注意力模块包括两个堆叠在一起的单一注意力模块。因此，在具有两个头的多头注意力模块中，我们现在有两个值权重矩阵：W 和 W。同样适用于其他权重矩阵，W 和 W。我们获得两组上下文向量 Z 和 Z，我们可以将它们组合成一个单一的上下文向量矩阵 Z。

如前所述，多头注意力的主要思想是通过不同的、学习到的线性投影（在并行中）多次运行注意力机制——通过将输入数据（如注意力机制中的查询、键和值向量）与权重矩阵相乘的结果。在代码中，我们可以通过实现一个简单的 MultiHeadAttentionWrapper 类来实现这一点，该类堆叠了我们之前实现的 CausalAttention 模块的多个实例。

Listing 3.4 A wrapper class to implement multi-head attention

```
class MultiHeadAttentionWrapper(nn.Module):
    def __init__(self, d_in, d_out, context_length,
                 dropout, num_heads, qkv_bias=False):
        super().__init__()
        self.heads = nn.ModuleList(
            [CausalAttention(
                 d_in, d_out, context_length, dropout, qkv_bias
             )
             for _ in range(num_heads)]
        )

    def forward(self, x):
        return torch.cat([head(x) for head in self.heads], dim=-1)
```

For example, if we use this `MultiHeadAttentionWrapper` class with two attention heads (via `num_heads=2`) and `CausalAttention` output dimension `d_out=2`, we get a four-dimensional context vector (`d_out*num_heads=4`), as depicted in figure 3.25.

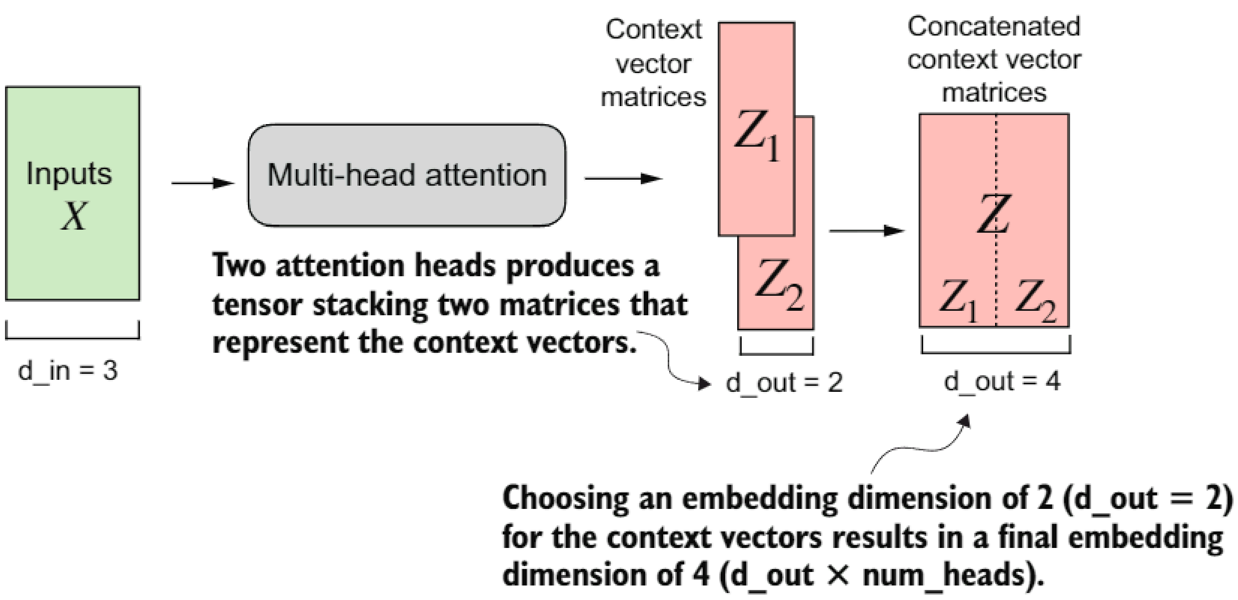

Figure 3.25 Using the `MultiHeadAttentionWrapper`, we specified the number of attention heads (`num_heads`). If we set `num_heads=2`, as in this example, we obtain a tensor with two sets of context vector matrices. In each context vector matrix, the rows represent the context vectors corresponding to the tokens, and the columns correspond to the embedding dimension specified via `d_out=4`. We concatenate these context vector matrices along the column dimension. Since we have two attention heads and an embedding dimension of 2, the final embedding dimension is 2 × 2 = 4.

To illustrate this further with a concrete example, we can use the `MultiHeadAttentionWrapper` class similar to the `CausalAttention` class before:

```
torch.manual_seed(123)
context_length = batch.shape[1] # This is the number of tokens
d_in, d_out = 3, 2
```

列表 3.4 一个实现多头注意力的包装类

```
    class MultiHeadAttentionWrapper(nn.Module):
类 多头注意力包装器(nn.Module):
        def __init__(self, d_in, d_out, 上下文长度,
                     dropout, 头数                     qkv 偏置=False):
            super().__init__()
            self.heads = nn.ModuleList()
                因果注意力
            d_in, d_out, context_length, dropout, qkv_bias) for _ in
        range(num_heads)]


        def forward(self, x): # 定义前向传播函数
            返回 torch.cat([head(x) for head in self.heads],)            dim=-1)
```

例如，如果我们使用这个 MultiHeadAttentionWrapper 类，并设置两个注意力头（通过 num_heads=2）以及因果注意力输出维度 d_out=2，我们将得到一个四维上下文向量（d_out*num_heads=4），如图 3.25 所示。

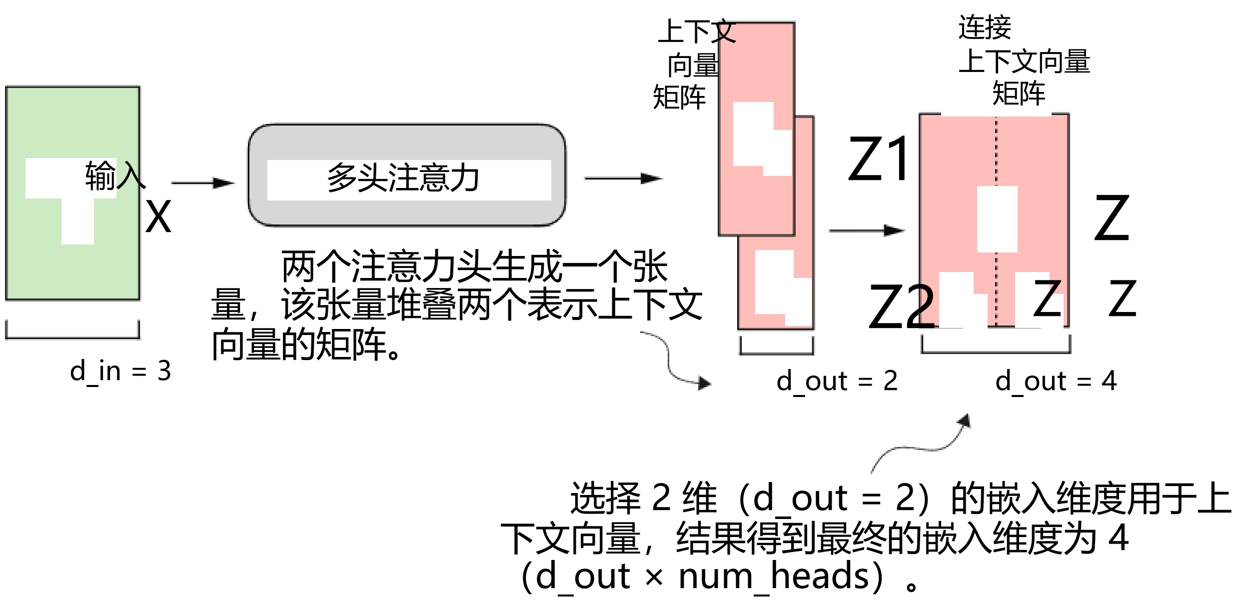

图 3.25 使用 MultiHeadAttentionWrapper，我们指定了注意力头数（num_heads）。如果我们设置 num_heads=2，如本例所示，我们获得一个包含两套上下文向量矩阵的张量。在每个上下文向量矩阵中，行表示与标记对应的上下文向量，列对应通过 d_out=4 指定的嵌入维度。我们沿着列维度将这些上下文向量矩阵连接起来。由于我们有两个注意力头和一个嵌入维度为 2，最终的嵌入维度是 2×2=4。

为了进一步用具体例子说明，我们可以使用 MultiHeadAttentionWrapper 类，类似于之前的 CausalAttention 类：

```
    torch 手动设置随机种子(123) context_length = batch.shape[1] # 这是
token 的数量 d_in, d_out = 3, 2
```

```
mha = MultiHeadAttentionWrapper(
    d_in, d_out, context_length, 0.0, num_heads=2
)
context_vecs = mha(batch)

print(context_vecs)
print("context_vecs.shape:", context_vecs.shape)
```

This results in the following tensor representing the context vectors:

```
tensor([[[-0.4519,  0.2216,  0.4772,  0.1063],
         [-0.5874,  0.0058,  0.5891,  0.3257],
         [-0.6300, -0.0632,  0.6202,  0.3860],
         [-0.5675, -0.0843,  0.5478,  0.3589],
         [-0.5526, -0.0981,  0.5321,  0.3428],
         [-0.5299, -0.1081,  0.5077,  0.3493]],

        [[-0.4519,  0.2216,  0.4772,  0.1063],
         [-0.5874,  0.0058,  0.5891,  0.3257],
         [-0.6300, -0.0632,  0.6202,  0.3860],
         [-0.5675, -0.0843,  0.5478,  0.3589],
         [-0.5526, -0.0981,  0.5321,  0.3428],
         [-0.5299, -0.1081,  0.5077,  0.3493]]], grad_fn=<CatBackward0>)
context_vecs.shape: torch.Size([2, 6, 4])
```

The first dimension of the resulting `context_vecs` tensor is 2 since we have two input texts (the input texts are duplicated, which is why the context vectors are exactly the same for those). The second dimension refers to the 6 tokens in each input. The third dimension refers to the four-dimensional embedding of each token.

Exercise 3.2 Returning two-dimensional embedding vectors

Change the input arguments for the `MultiHeadAttentionWrapper(..., num_heads=2)` call such that the output context vectors are two-dimensional instead of four dimensional while keeping the setting `num_heads=2`. Hint: You don't have to modify the class implementation; you just have to change one of the other input arguments.

Up to this point, we have implemented a `MultiHeadAttentionWrapper` that combined multiple single-head attention modules. However, these are processed sequentially via `[head(x) for head in self.heads]` in the forward method. We can improve this implementation by processing the heads in parallel. One way to achieve this is by computing the outputs for all attention heads simultaneously via matrix multiplication.

```
2 mha = MultiHeadAttentionWrapper( 输入维度, 输出维度,
上下文长度, 0.0, 头数=2 ) context_vecs = mha(批次)
```

```
打印上下文向量，打印("上下文向量形状：",
context_vecs.shape)
```

这导致以下张量表示上下文向量：

```
张量([[[-0.4519,        0.2216,     0.4772,     0.1063]
         [-0.5874,]    0.0058,     0.5891,     0.3257]
         [-0.6300，] -0.0632,     0.6202,     0.3860]
         [-0.5675,]  -0.0843,     0.5478,     0.3589]
         [-0.5526]   -0.0981,     0.5321,     0.3428]
         [-0.5299]   -0.1081,     0.5077,     0.3493]]

        [-0.4519,]    0.2216,     0.4772,     0.1063]
         [-0.5874,]    0.0058,     0.5891,     0.3257]
         [-0.6300，] -0.0632,     0.6202,     0.3860]
         [-0.5675,]  -0.0843,     0.5478,     0.3589]
         [-0.5526]   -0.0981,     0.5321,     0.3428]
         [-0.5299]   -0.1081,     0.5077,     0.3493]]], 梯度函数=)
context_vecs.shape: torch.Size([2, 6,])       4])
```

结果 context_vecs 张量的第一维是 2，因为我们有两个输入文本（输入文本被重复，这就是为什么那些上下文向量完全相同）。第二维指的是每个输入中的 6 个标记。第三维指的是每个标记的四维嵌入。

练习 3.2 返回二维嵌入向量

修改 MultiHeadAttentionWrapper(..., num_heads=2)调用中的输入参数，使得输出上下文向量是二维的，而不是四维的，同时保持 num_heads=2 的设置。提示：您不需要修改类实现；只需更改其他输入参数之一。

截至目前，我们已实现了一个 MultiHeadAttentionWrapper，它结合了多个单头注意力模块。然而，这些模块是按顺序处理的。

在 forward 方法中，[head(x) for head in self.heads]。我们可以通过并行处理 heads 来改进这个实现。实现这一目标的一种方法是通过矩阵乘法同时计算所有注意力 heads 的输出。

3.6.2 Implementing multi-head attention with weight splits

So far, we have created a `MultiHeadAttentionWrapper` to implement multi-head attention by stacking multiple single-head attention modules. This was done by instantiating and combining several `CausalAttention` objects.

Instead of maintaining two separate classes, `MultiHeadAttentionWrapper` and `CausalAttention`, we can combine these concepts into a single `MultiHeadAttention` class. Also, in addition to merging the `MultiHeadAttentionWrapper` with the `CausalAttention` code, we will make some other modifications to implement multi-head attention more efficiently.

In the `MultiHeadAttentionWrapper`, multiple heads are implemented by creating a list of `CausalAttention` objects (`self.heads`), each representing a separate attention head. The `CausalAttention` class independently performs the attention mechanism, and the results from each head are concatenated. In contrast, the following `MultiHeadAttention` class integrates the multi-head functionality within a single class. It splits the input into multiple heads by reshaping the projected query, key, and value tensors and then combines the results from these heads after computing attention.

Let's take a look at the `MultiHeadAttention` class before we discuss it further.

Listing 3.5 An efficient multi-head attention class

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_in, d_out,
                 context_length, dropout, num_heads, qkv_bias=False):
        super().__init__()
        assert (d_out % num_heads == 0), \
            "d_out must be divisible by num_heads"

        self.d_out = d_out
        self.num_heads = num_heads
        self.head_dim = d_out // num_heads
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.out_proj = nn.Linear(d_out, d_out)
        self.dropout = nn.Dropout(dropout)
        self.register_buffer(
            "mask",
            torch.triu(torch.ones(context_length, context_length),
                       diagonal=1)
        )

    def forward(self, x):
        b, num_tokens, d_in = x.shape
        keys = self.W_key(x)
        queries = self.W_query(x)
        values = self.W_value(x)
```

Reduces the projection dim to match the desired output dim

Uses a Linear layer to combine head outputs

Tensor shape: (b, num_tokens, d_out)

3.6.2 实现多头注意力机制，权重拆分

截至目前，我们已创建了一个多头注意力包装器实现多头

通过堆叠多个单头注意力模块进行注意力。这是通过实例化和组合多个 CausalAttention 对象来实现的。

代替维护两个独立的类， 多头注意力包装器和

因果注意力，我们可以将这些概念合并成一个单一的 MultiHeadAttention 类。此外，除了将 MultiHeadAttentionWrapper 与 Causal-

注意代码，我们将进行一些其他修改以更有效地实现多头注意力。

在 MultiHeadAttentionWrapper 中，通过创建一个 CausalAttention 对象列表（self.heads）来实现多个头部，每个对象代表一个独立的注意力头部。CausalAttention 类独立执行注意力机制，并将每个头部的结果连接起来。相比之下，以下

多头注意力类将多头功能集成在一个类中。它通过重塑投影查询、键和值张量将输入分割成多个头，然后在计算注意力后结合这些头的输出。

让我们在进一步讨论之前看看 MultiHeadAttention 类。

列表 3.5 高效的多头注意力类

```
class 多头注意力(nn.Module):
    def __init__(self, d_in, d_out,
                 context_length,                 num_heads, qkv_bias=False):
                 dropout
        super().__init__() 断言 (d_out %
       num_heads == 0),
            d_out 必须可被整除             by  num_heads" 头数"

        self.d_out = d_out
        self.num_heads = num_heads self.head_dim = d_out //
       num_heads self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
       self.W_key = nn.Linear(d_in, d_out, bias=qkv_bias) self.W_value
       = nn.Linear(d_in, d_out, bias=qkv_bias) self.out_proj =
       nn.Linear(d_out, d_out) self.dropout = nn.Dropout(dropout)
       self.register_buffer(
            "mask", torch.triu(torch.ones(上下文长度, 上下文长度),
                    对角线=1)
        )

    def forward(self, x): # 定义前向传播函数
        b, num_tokens, d_in = x.shape
       keys = self.W_key(x) queries =
       self.W_query(x) values =
       self.W_value(x)
```

降低投影维度以匹配所需的输出维度

使用线性层组合头部输出

张量形状：(b, num_tokens, d_out)

```
        keys = keys.view(b, num_tokens, self.num_heads, self.head_dim)
        values = values.view(b, num_tokens, self.num_heads, self.head_dim)
        queries = queries.view(
            b, num_tokens, self.num_heads, self.head_dim
        )

        keys = keys.transpose(1, 2)
        queries = queries.transpose(1, 2)
        values = values.transpose(1, 2)

        attn_scores = queries @ keys.transpose(2, 3)
        mask_bool = self.mask.bool()[:num_tokens, :num_tokens]

        attn_scores.masked_fill_(mask_bool, -torch.inf)

        attn_weights = torch.softmax(
            attn_scores / keys.shape[-1]**0.5, dim=-1)
        attn_weights = self.dropout(attn_weights)

        context_vec = (attn_weights @ values).transpose(1, 2)

        context_vec = context_vec.contiguous().view(
            b, num_tokens, self.d_out
        )
        context_vec = self.out_proj(context_vec)
        return context_vec
```

We implicitly split the matrix by adding a num_heads dimension. Then we unroll the last dim: (b, num_tokens, d_out) -> (b, num_tokens, num_heads, head_dim).

Transposes from shape (b, num_tokens, num_heads, head_dim) to (b, num_heads, num_tokens, head_dim)

Computes dot product for each head

Masks truncated to the number of tokens

Uses the mask to fill attention scores

Tensor shape: (b, num_tokens, n_heads, head_dim)

Combines heads, where self.d_out = self.num_heads * self.head_dim

Adds an optional linear projection

Even though the reshaping (`.view`) and transposing (`.transpose`) of tensors inside the `MultiHeadAttention` class looks very mathematically complicated, the `MultiHeadAttention` class implements the same concept as the `MultiHeadAttentionWrapper` earlier.

On a big-picture level, in the previous `MultiHeadAttentionWrapper`, we stacked multiple single-head attention layers that we combined into a multi-head attention layer. The `MultiHeadAttention` class takes an integrated approach. It starts with a multi-head layer and then internally splits this layer into individual attention heads, as illustrated in figure 3.26.

The splitting of the query, key, and value tensors is achieved through tensor reshaping and transposing operations using PyTorch's `.view` and `.transpose` methods. The input is first transformed (via linear layers for queries, keys, and values) and then reshaped to represent multiple heads.

The key operation is to split the `d_out` dimension into `num_heads` and `head_dim`, where `head_dim = d_out / num_heads`. This splitting is then achieved using the `.view` method: a tensor of dimensions `(b, num_tokens, d_out)` is reshaped to dimension `(b, num_tokens, num_heads, head_dim)`.

```
    keys = keys.view(b, num_tokens, self.num_heads, self.head_dim)
values = values.view(b, num_tokens, self.num_heads, self.head_dim)
queries = queries.view(
    b, num_tokens, self.num_heads, self.head_dim

    keys = keys 转置(1, 2) queries =
queries 转置(1, 2) values = values 转置
(1, 2)

    attn_scores = 查询 @ 键的转置(2, 3) mask_bool =
self.mask.bool()[:num_tokens,                  : 累计令牌数

    attn_scores.masked_fill_(mask_bool, -无穷)

    注意权重 = torch.softmax(
    注意分数 / keys.shape[-1]**0.5, dim=-1) 注意权重 =
self.dropout(注意权重)

    context_vec = (注意力权重)     @ values).转置(1,      2)

    context_vec = context_vec 连续(). 视图()
    b, num_tokens, self.d_out) context_vec =
self.out_proj(context_vec) return context_vec
```

我们隐式地分割矩阵通过添加数字头维度。然后我们展开 last dim: (b, num_tokens, d_out) -> (b, num_tokens, num_heads, head_dim).

将形状从 (b, num_tokens, num_heads, head_dim) 转换为 (b, num_heads, num_tokens, head_dim)

口罩截断为数字代币

计算点积对于每个头部

使用口罩填充注意分数

张量形状：(b, num_tokens, num_heads, head_dim)

结合头部，其中 self.d_out = self.num_heads * self.head_dim

添加一个可选的线性投影

尽管 MultiHeadAttention 类中张量的重塑（.view）和转置（.transpose）在数学上看起来非常复杂，但 Multi-

头部注意力 类实现了与相同的概念 多头注意力

包装器 更早。

在宏观层面上，在之前的 MultiHeadAttentionWrapper 中，我们将多个单头注意力层堆叠起来，组合成一个多头注意力层。MultiHeadAttention 类采用了一种综合方法。它从一个多头层开始，然后内部将这个层拆分为单独的注意力头，如图 3.26 所示。

查询、键和值张量的拆分是通过使用 PyTorch 的.view 和.transpose 方法进行张量重塑和转置操作实现的。输入首先通过线性层（用于查询、键和值）进行转换，然后重塑以表示多个头。

关键操作是将 d_out 维度拆分为 num_heads 和 head_dim，其中 head_dim = d_out / num_heads。然后使用 .view 方法实现拆分：将维度为 (b, num_tokens, d_out) 的张量重塑为维度

(b, num_tokens, num_heads, head_dim)

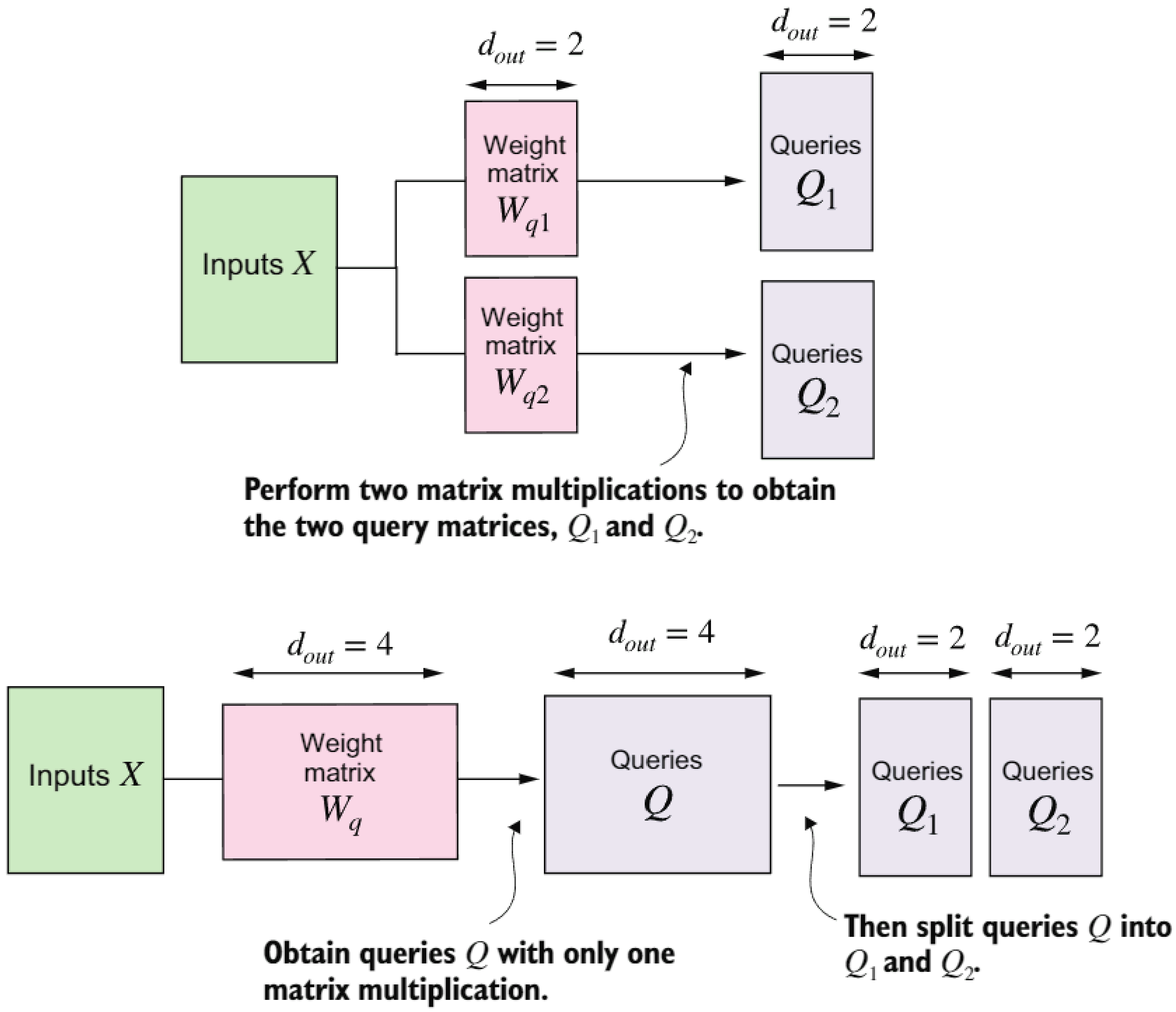

Figure 3.26 In the `MultiHeadAttentionWrapper` class with two attention heads, we initialized two weight matrices, W_{q1} and W_{q2}, and computed two query matrices, Q_1 and Q_2 (top). In the `MultiheadAttention` class, we initialize one larger weight matrix W_q, only perform one matrix multiplication with the inputs to obtain a query matrix Q, and then split the query matrix into Q_1 and Q_2 (bottom). We do the same for the keys and values, which are not shown to reduce visual clutter.

The tensors are then transposed to bring the `num_heads` dimension before the `num_tokens` dimension, resulting in a shape of `(b, num_heads, num_tokens, head_dim)`. This transposition is crucial for correctly aligning the queries, keys, and values across the different heads and performing batched matrix multiplications efficiently.

To illustrate this batched matrix multiplication, suppose we have the following tensor:

```
a = torch.tensor([[[[0.2745, 0.6584, 0.2775, 0.8573],
                    [0.8993, 0.0390, 0.9268, 0.7388],
                    [0.7179, 0.7058, 0.9156, 0.4340]],

                   [[0.0772, 0.3565, 0.1479, 0.5331],
                    [0.4066, 0.2318, 0.4545, 0.9737],
                    [0.4606, 0.5159, 0.4220, 0.5786]]]])
```

The shape of this tensor is (b, num_heads, num_tokens, head_dim) = (1, 2, 3, 4).

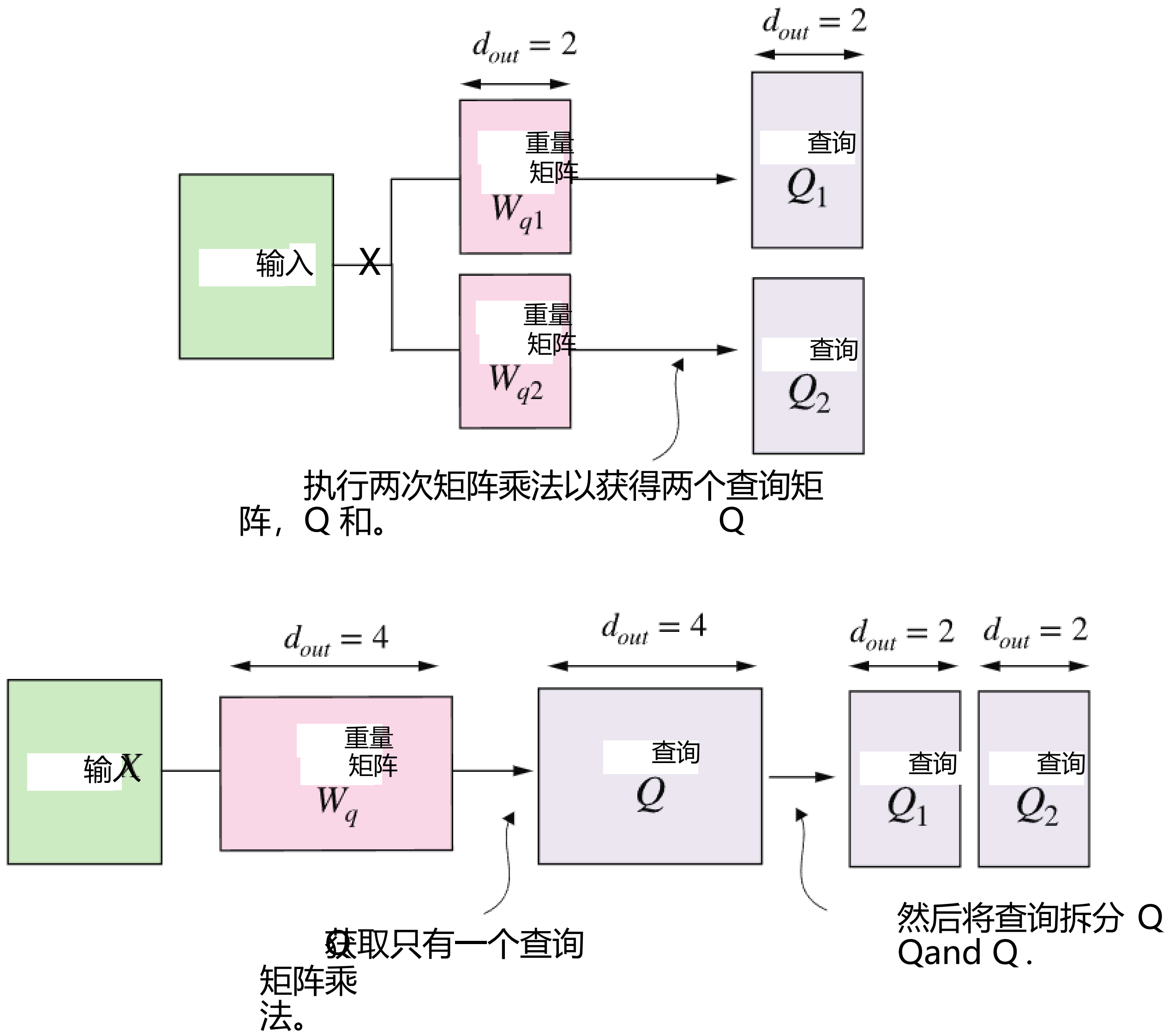

图 3.26 在具有两个注意力头的 MultiHeadAttentionWrapper 类中，我们初始化了两个权重矩阵 W 和 W，并计算了两个查询矩阵 Q 和 Q(顶部)。在 MultiheadAttention 类中，我们初始化了一个更大的权重矩阵 W，仅对输入执行一次矩阵乘法以获得查询矩阵 Q，然后将查询矩阵分割为 Q 和 Q(底部)。对于键和值，我们执行相同的操作，为了减少视觉混乱，这里没有展示。

张量随后被转置，将 num_heads 维度置于 num_ 前面
tokens 维度，结果形状为 (b, num_heads, num_tokens, head_dim)。这种转置对于正确地对齐不同头部的查询、键和值以及高效执行批量矩阵乘法至关重要。

为了说明这种批量矩阵乘法，假设我们有一个以下张量：

```
a = torch.tensor([[[[0.2745, 0.6584,          [0.2775, 0.8573]
                    [0.8993, 0.0390, 0.9268, 0.7388],
                    [0.7179, 0.7058,]          [0.9156, 0.4340]]

                    [0.0772, 0.3565, 0.1479, 0.5331],
                   [0.4066, 0.2318, 0.4545, 0.9737], [0.4606,
                   0.5159,]                    0.4220, 0.5786]]]])
```

这个张量的形状是 (b, num_heads, num_tokens, head_dim) = (1, 2, 3, Now we perform a batched matrix multiplication between the tensor itself and a view of the tensor where we transposed the last two dimensions, `num_tokens` and `head_dim`:

```
print(a @ a.transpose(2, 3))
```

The result is

```
tensor([[[[1.3208, 1.1631, 1.2879],
          [1.1631, 2.2150, 1.8424],
          [1.2879, 1.8424, 2.0402]],

         [[0.4391, 0.7003, 0.5903],
          [0.7003, 1.3737, 1.0620],
          [0.5903, 1.0620, 0.9912]]]])
```

In this case, the matrix multiplication implementation in PyTorch handles the four-dimensional input tensor so that the matrix multiplication is carried out between the two last dimensions `(num_tokens, head_dim)` and then repeated for the individual heads.

For instance, the preceding becomes a more compact way to compute the matrix multiplication for each head separately:

```
first_head = a[0, 0, :, :]
first_res = first_head @ first_head.T
print("First head:\n", first_res)

second_head = a[0, 1, :, :]
second_res = second_head @ second_head.T
print("\nSecond head:\n", second_res)
```

The results are exactly the same results as those we obtained when using the batched matrix multiplication `print(a @ a.transpose(2, 3))`:

```
First head:
 tensor([[1.3208, 1.1631, 1.2879],
        [1.1631, 2.2150, 1.8424],
        [1.2879, 1.8424, 2.0402]])

Second head:
 tensor([[0.4391, 0.7003, 0.5903],
        [0.7003, 1.3737, 1.0620],
        [0.5903, 1.0620, 0.9912]])
```

Continuing with `MultiHeadAttention`, after computing the attention weights and context vectors, the context vectors from all heads are transposed back to the shape `(b, num_tokens, num_heads, head_dim)`. These vectors are then reshaped (flattened) into the shape `(b, num_tokens, d_out)`, effectively combining the outputs from all heads.

Additionally, we added an output projection layer (`self.out_proj`) to `MultiHeadAttention` after combining the heads, which is not present in the `CausalAttention` class. This output projection layer is not strictly necessary (see appendix B for

现在我们在张量本身和转置了最后两个维度（num_tokens 和 head_dim）的张量视图之间执行批量矩阵乘法：

```
打印(a @)   a.transpose(2, 3)
```

结果是

```
张量([[[1.3208,        [1.1631, 1.2879]
             [1.1631, 2.2150, 1.8424],
        [1.2879,       [1.8424, 2.0402]]

            [0.4391, 0.7003, 0.5903],
       [0.7003, 1.3737, 1.0620],
       [0.5903, 1.0620]1.0620, 0.9912]]]])
```

在这种情况下，PyTorch 中的矩阵乘法实现处理四维输入张量，以便在两个最后维度（num_tokens, head_dim）之间执行矩阵乘法，然后对各个头进行重复。

例如，上述方法成为分别计算每个头矩阵乘法的一种更紧凑的方式：

```
    first_head = a[0, 0, :, :] first_res =
first_head @ first_head.T 打印("First
head:\n", first_res)

    second_head = a[0, 1, :, :] second_res =
second_head @ second_head.T 打印("\nSecond
head:\n", second_res)
```

结果与我们使用批量处理获得的结果完全相同

矩阵乘法打印(a @ a.transpose(2, 3))：

```
第一头：tensor([[1.3208,
1.1631,                          1.2879]
          [1.1631, 2.2150, 1.8424],
       [1.2879,       1.8424, 2.0402]]]

第二头：tensor([[0.4391,
0.7003,                          0.5903]
          [0.7003, 1.3737, 1.0620],
       [0.5903,]      1.0620, 0.9912]])
```

继续使用 MultiHeadAttention，在计算注意力权重和上下文向量之后，所有头的上下文向量被转置回形状(b, num_tokens, num_heads, head_dim)。然后这些向量被重塑（展平）为形状(b, num_tokens, d_out)，有效地结合了所有头的输出。

此外，我们在合并头部后向 MultiHeadAttention 添加了一个输出投影层（self.out_proj），这在原始版本中是不存在的。 因果-

注意班级。此输出投影层并非绝对必要（参见附录 B）

more details), but it is commonly used in many LLM architectures, which is why I added it here for completeness.

Even though the `MultiHeadAttention` class looks more complicated than the `MultiHeadAttentionWrapper` due to the additional reshaping and transposition of tensors, it is more efficient. The reason is that we only need one matrix multiplication to compute the keys, for instance, `keys = self.W_key(x)` (the same is true for the queries and values). In the `MultiHeadAttentionWrapper`, we needed to repeat this matrix multiplication, which is computationally one of the most expensive steps, for each attention head.

The `MultiHeadAttention` class can be used similar to the `SelfAttention` and `CausalAttention` classes we implemented earlier:

```
torch.manual_seed(123)
batch_size, context_length, d_in = batch.shape
d_out = 2
mha = MultiHeadAttention(d_in, d_out, context_length, 0.0, num_heads=2)
context_vecs = mha(batch)
print(context_vecs)
print("context_vecs.shape:", context_vecs.shape)
```

The results show that the output dimension is directly controlled by the `d_out` argument:

```
tensor([[[0.3190, 0.4858],
         [0.2943, 0.3897],
         [0.2856, 0.3593],
         [0.2693, 0.3873],
         [0.2639, 0.3928],
         [0.2575, 0.4028]],

        [[0.3190, 0.4858],
         [0.2943, 0.3897],
         [0.2856, 0.3593],
         [0.2693, 0.3873],
         [0.2639, 0.3928],
         [0.2575, 0.4028]]], grad_fn=<ViewBackward0>)
context_vecs.shape: torch.Size([2, 6, 2])
```

We have now implemented the `MultiHeadAttention` class that we will use when we implement and train the LLM. Note that while the code is fully functional, I used relatively small embedding sizes and numbers of attention heads to keep the outputs readable.

For comparison, the smallest GPT-2 model (117 million parameters) has 12 attention heads and a context vector embedding size of 768. The largest GPT-2 model (1.5 billion parameters) has 25 attention heads and a context vector embedding size of 1,600. The embedding sizes of the token inputs and context embeddings are the same in GPT models (`d_in = d_out`).

这一输出投影层并非绝对必要（更多详情见附录 B），但它被广泛应用于许多LLM架构中，因此我为了完整性在此添加了它。

尽管如此 多头注意力 类看起来比类更复杂

多头注意力包装器由于额外的张量重塑和转置，效率更高。原因是我们只需要一次矩阵乘法来计算键，例如，keys = self.W_key(x)（对于查询和值也是如此）。在 MultiHeadAttentionWrapper 中，我们需要对每个注意力头重复这个矩阵乘法，这是计算上最昂贵的步骤之一。

多头注意力类可以像我们之前实现的 SelfAttention 和 CausalAttention 类一样使用：

```
torch 手动设置随机种子(123) 批量大小，上下文长度，输入维度 d_in = 批量形状 d_out =
2 mha = 多头注意力(d_in, d_out, 上下文长度, 0.0, num_heads=2) 上下文向量 = mha(批量)
打印(上下文向量) 打印("上下文向量.shape:", 上下文向量.shape)
```

结果显示，输出维度直接受控于 d_out

论证：

```
张量([[0.3190,      0.4858]
         [0.2943, 0.3897]
         [0.2856, 0.3593]
         [0.2693, 0.3873]
         [0.2639, 0.3928]
         [0.2575, ] 0.4028]]

        [0.3190, 0.4858]
         [0.2943, 0.3897]
         [0.2856, 0.3593]
         [0.2693, 0.3873]
         [0.2639, 0.3928]
0.4028]]], grad_fn=[0.2575, )]context_vecs.shape: torch.Size([2, 6,
2])
```

我们现在已实现了 MultiHeadAttention 类，在实现和训练LLM时将使用它。请注意，虽然代码完全可用，但我使用了相对较小的嵌入尺寸和注意力头数量，以保持输出可读。

与比较，最小的 GPT-2 模型（117 百万参数）有 12 个注意力头和 768 大小的上下文向量嵌入。最大的 GPT-2 模型（15 亿参数）有 25 个注意力头和 1600 大小的上下文向量嵌入。在 GPT 模型中，标记输入和上下文嵌入的大小相同（d_in = d_out）。

> **Exercise 3.3 Initializing GPT-2 size attention modules**
>
> Using the `MultiHeadAttention` class, initialize a multi-head attention module that has the same number of attention heads as the smallest GPT-2 model (12 attention heads). Also ensure that you use the respective input and output embedding sizes similar to GPT-2 (768 dimensions). Note that the smallest GPT-2 model supports a context length of 1,024 tokens.

Summary

- Attention mechanisms transform input elements into enhanced context vector representations that incorporate information about all inputs.
- A self-attention mechanism computes the context vector representation as a weighted sum over the inputs.
- In a simplified attention mechanism, the attention weights are computed via dot products.
- A dot product is a concise way of multiplying two vectors element-wise and then summing the products.
- Matrix multiplications, while not strictly required, help us implement computations more efficiently and compactly by replacing nested `for` loops.
- In self-attention mechanisms used in LLMs, also called scaled-dot product attention, we include trainable weight matrices to compute intermediate transformations of the inputs: queries, values, and keys.
- When working with LLMs that read and generate text from left to right, we add a causal attention mask to prevent the LLM from accessing future tokens.
- In addition to causal attention masks to zero-out attention weights, we can add a dropout mask to reduce overfitting in LLMs.
- The attention modules in transformer-based LLMs involve multiple instances of causal attention, which is called multi-head attention.
- We can create a multi-head attention module by stacking multiple instances of causal attention modules.
- A more efficient way of creating multi-head attention modules involves batched matrix multiplications.

练习 3.3 初始化 GPT-2 大小注意力模块

使用 MultiHeadAttention 类，初始化一个具有与最小 GPT-2 模型相同数量注意力头（12 个注意力头）的多头注意力模块。同时确保使用与 GPT-2 相似的相应输入和输出嵌入大小（768 维）。请注意，最小 GPT-2 模型支持 1,024 个标记的上下文长度。

摘要

- 注意力机制将输入元素转换为增强的上下文向量表示，其中包含关于所有输入的信息。
- 自注意力机制将上下文向量表示计算为输入的加权和。
- 在一个简化的注意力机制中，注意力权重通过点积计算。
- 点积是逐元素相乘两个向量然后求和的简洁方式。
- 矩阵乘法，虽然不是严格必需的，但通过替换嵌套循环，有助于我们更高效、更紧凑地实现计算。
- 在LLMs中使用的自注意力机制，也称为缩放点积注意力，我们包括可训练的权重矩阵来计算输入的中间变换：查询、值和键。
- 当与从左到右读取和生成文本的LLMs一起工作时，我们添加因果注意力掩码以防止LLM访问未来的标记。
- 除了将注意力权重置零的因果注意力掩码外，我们还可以添加一个 dropout 掩码来减少LLMs中的过拟合。
- Transformer-based LLMs中的注意力模块涉及多个因果注意力实例，这被称为多头注意力。
- 我们可以通过堆叠多个因果注意力模块来创建一个多头注意力模块。
- 更高效地创建多头注意力模块的方法涉及批量矩阵乘法。