3. 编写注意力机制

前言

本章包括：

- 在神经网络中使用注意力机制的原因
- 一个基本的自注意力框架，进步到一个增强的自注意力机制
- 一个允许LLM一次生成一个token的因果注意力模块
- 用dropout随机遮掩选中的注意力权重以减少过拟合
- 把多个因果注意力模块堆叠到一个多头注意力模块中

我们将看一看一个完整的LLM架构一部分，注意力机制，如图3.1。我们将主要单独研究注意力机制，并在机制层面上关注它们。然后，我们将编写围绕自注意力机制的LLM其余部分的代码，以实际观察其运行情况并创建一个生成文本的模型。

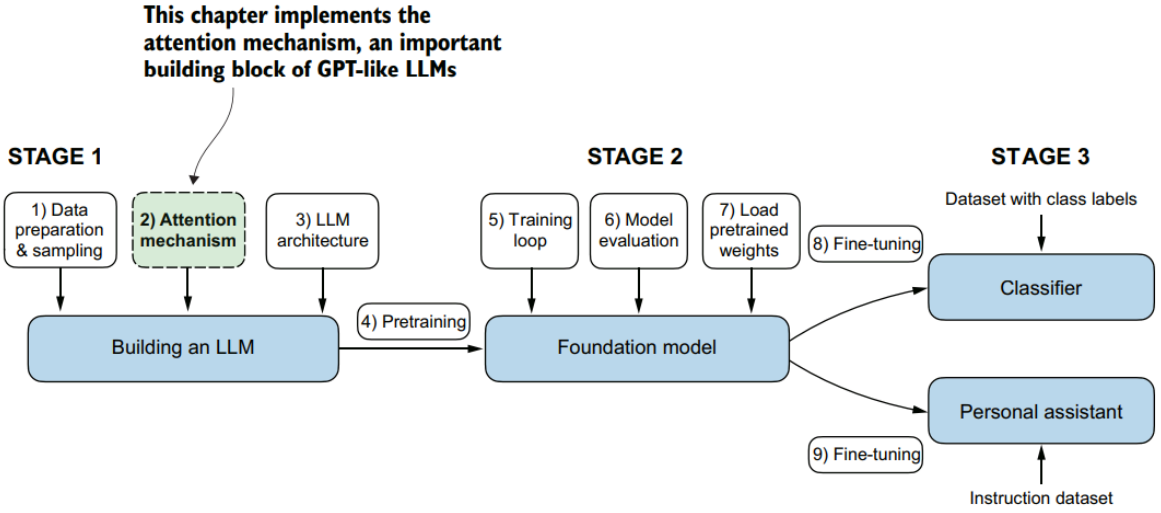

Figure 3.1 The three main stages of coding an LLM. This chapter focuses on step 2 of stage 1: implementing attention mechanisms, which are an integral part of the LLM architecture.

我们将实现四种注意力机制的变体，如图3.2所示。这些不同的变体都建立在彼此之上，目标是实现一个完整的，高效的可以插入我们将在下一章实现的LLM架构的多头注意力。

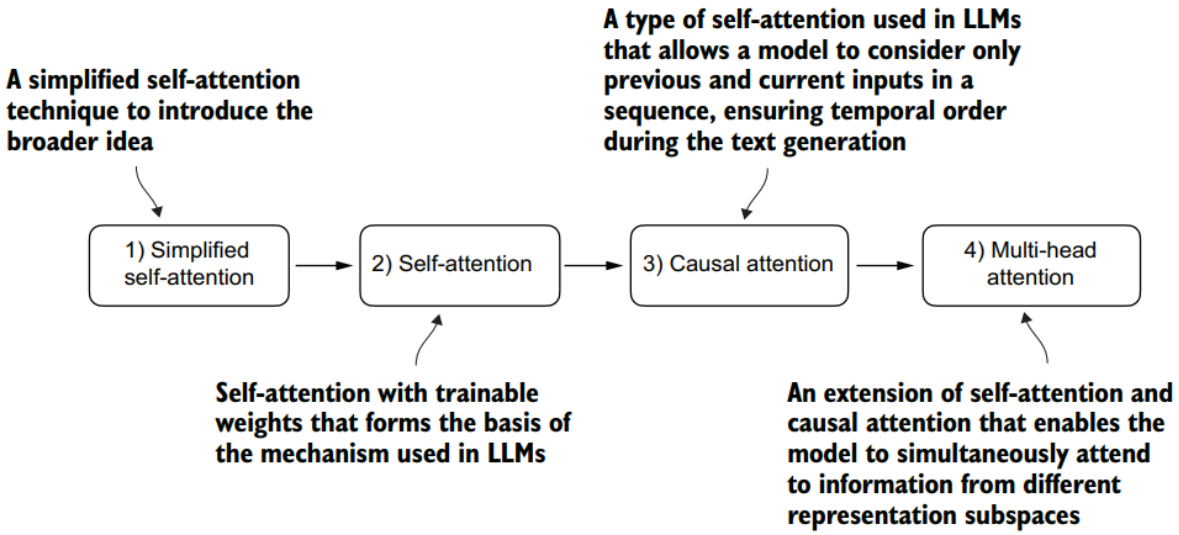

Figure 3.2 The figure depicts different attention mechanisms we will code in this chapter, starting with a simplified version of self-attention before adding the trainable weights. The causal attention mechanism adds a mask to self-attention that allows the LLM to generate one word at a time. Finally,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

3.1 建模长序列的问题

假设我们要构建一个语言翻译模型，从一种语言文本翻译到另外一种，如图3.3所示。我们不能简单地逐字翻译，由于不同语言的语法结构不同。

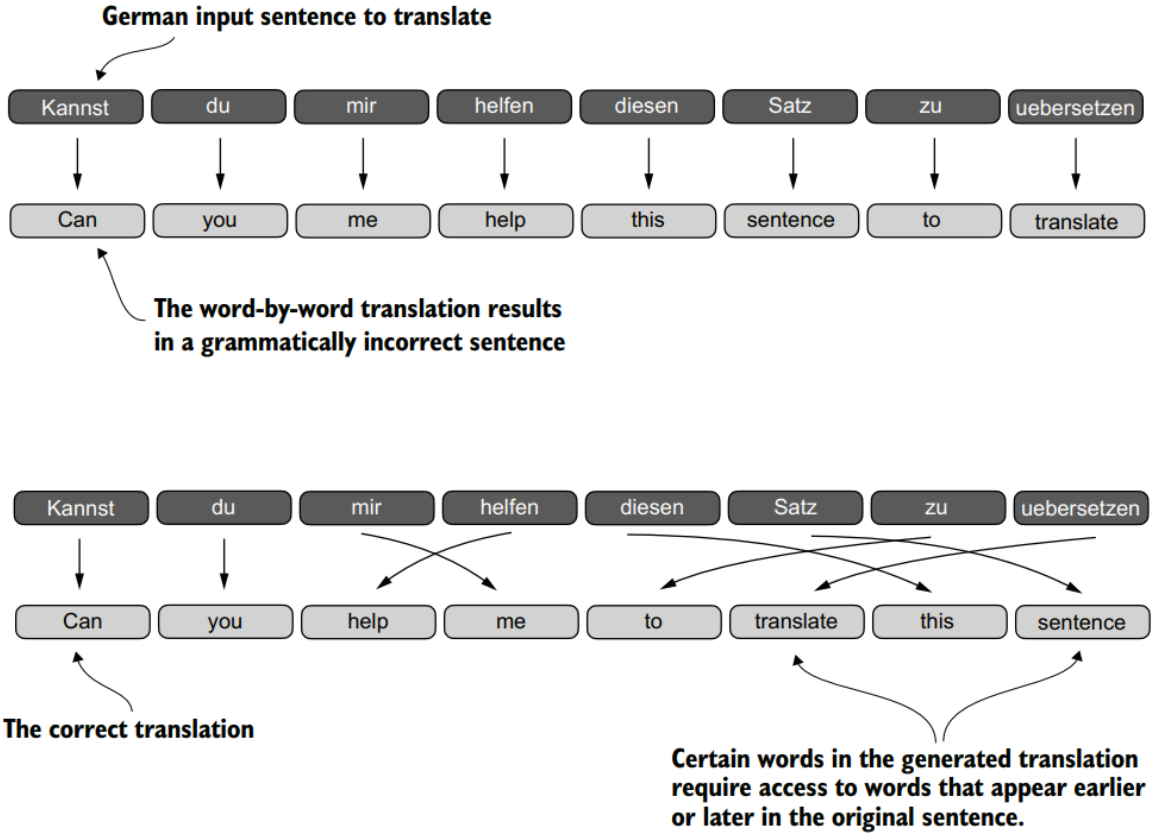

Figure 3.3 When translating text from one language to another, such as German to English, it's not possible to merely translate word by word. Instead, the translation process requires contextual understanding and grammatical alignment.

为了解决这个问题，通常是使用带有两个子模块——encoder和decoder的深度神经网络。encoder的工作是首先读取并处理整篇文本，然后decoder产生翻译后的文本。

在Transformer到来之前，循环神经网络（recurrent neural networks, RNNs）是对于语言翻译来说最流行的encoder-decoder架构。一个RNN是一类神经网络，其来自先前步骤的输出被喂给当前步骤的输入，使它们很好地适配像文本这样的序列数据。

在一个encoder-decoder的RNN中，输入文本被喂给encoder，并顺序地处理它。encoder在每一步更新它的隐藏状态（隐藏层中的内部值），在最终的隐藏状态中尝试捕获输入序列完整的意思，如图3.4所示。然后，decoder采取这个最终的隐藏状态来生成翻译后的句子，一次生成一个单词。它也在每一步更新它的隐藏状态，该状态应该携带下一个词预测所需的上下文信息。

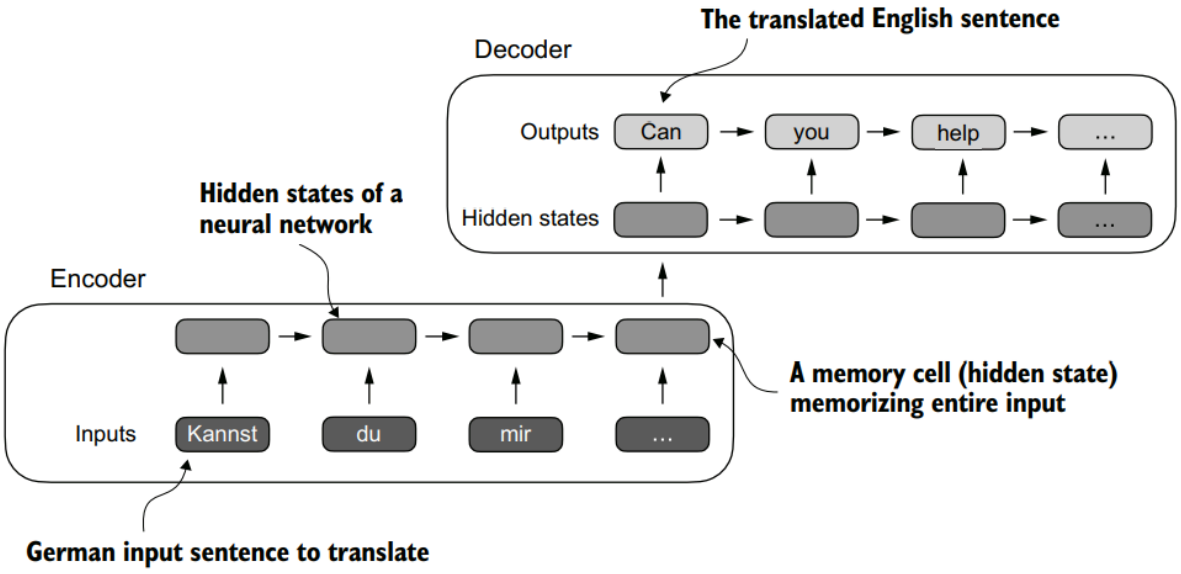

Figure 3.4 Before the advent of transformer models, encoder–decoder RNNs were a popular choice for machine translation. The encoder takes a sequence of tokens from the source language as input, where a hidden state (an intermediate neural network layer) of the encoder encodes a compressed representation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

可以把encoder处理整个原始文本得到的隐藏状态理解成一种嵌入向量。

encoder-decoder RNNs的一大限制是RNN在解码阶段不能直接从encoder获取更早的隐藏状态。因此，它只依赖于当前的隐藏状态，该隐藏状态压缩了所有的相关信息，这可能导致一种上下文的缺失，尤其是在长距离依赖的复杂句子中。

3.2 使用注意力机制来捕获数据依赖

尽管RNN对于短句翻译来说很有用，但是他们对于更长的文本不是很好用，因为它们没有直接获取输入中更早的单词。这种方法的一个主要缺点是RNN必须在一个单独的隐藏状态中记住整个编码后的输入，在它传递给decoder之前。（如图3.4）

因此，研究人员在2014年为RNNs研发了Bahdanau Attention机制，修改了encoder-decoder RNNs从而decoder可以在decoding的每一个步骤选择性获取输入序列的不同部分，如图3.5所示：

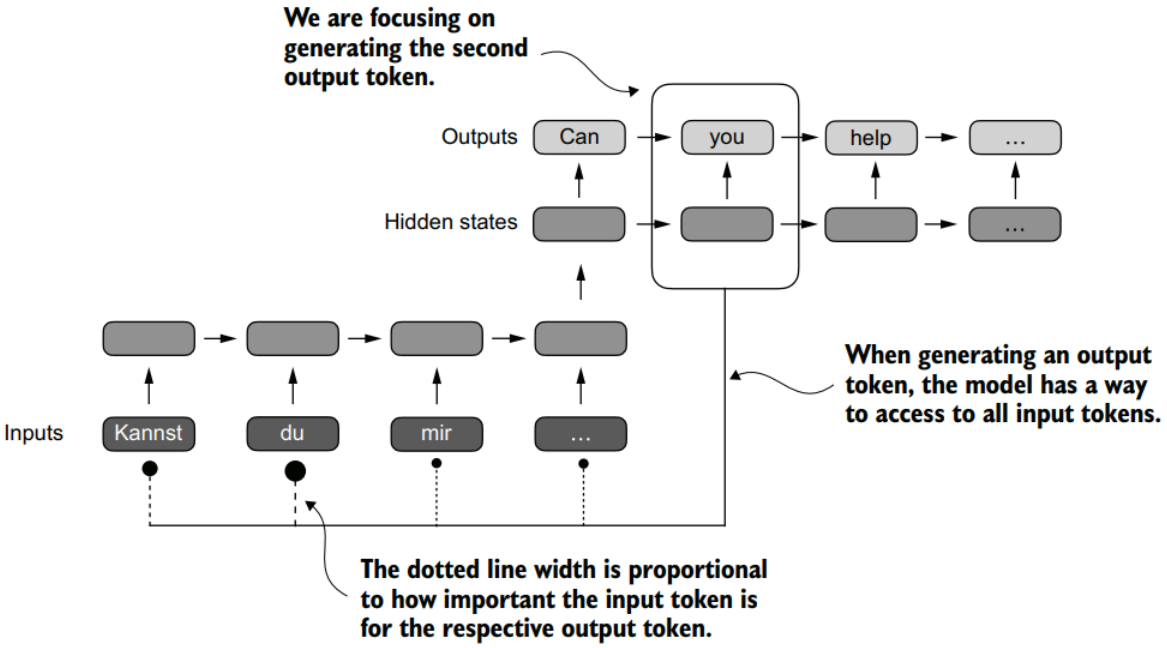

Figure 3.5 Using an attention mechanism, the text-generating decoder part of the network can access all input tokens selectively. This means that some input tokens are more important than others for generating a given output token. The importance is determined by the attention weights, which we will 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.

3年后，研究人员发现RNN对于构建自然语言处理深度学习网络不是必要的，并提出了原始的transformer架构，包括一个由Bahdanau注意力机制启发得到的自注意力机制。

自注意力是一种当计算一个序列表示时，允许输入序列的每个位置考虑同一输入序列中所有其他位置的相关性的机制。自注意力是当代基于Transformer架构的LLM的一个关键组件。

本章将编码和理解自注意力机制（如图3.6），并在下一章编写LLM的剩余部分。

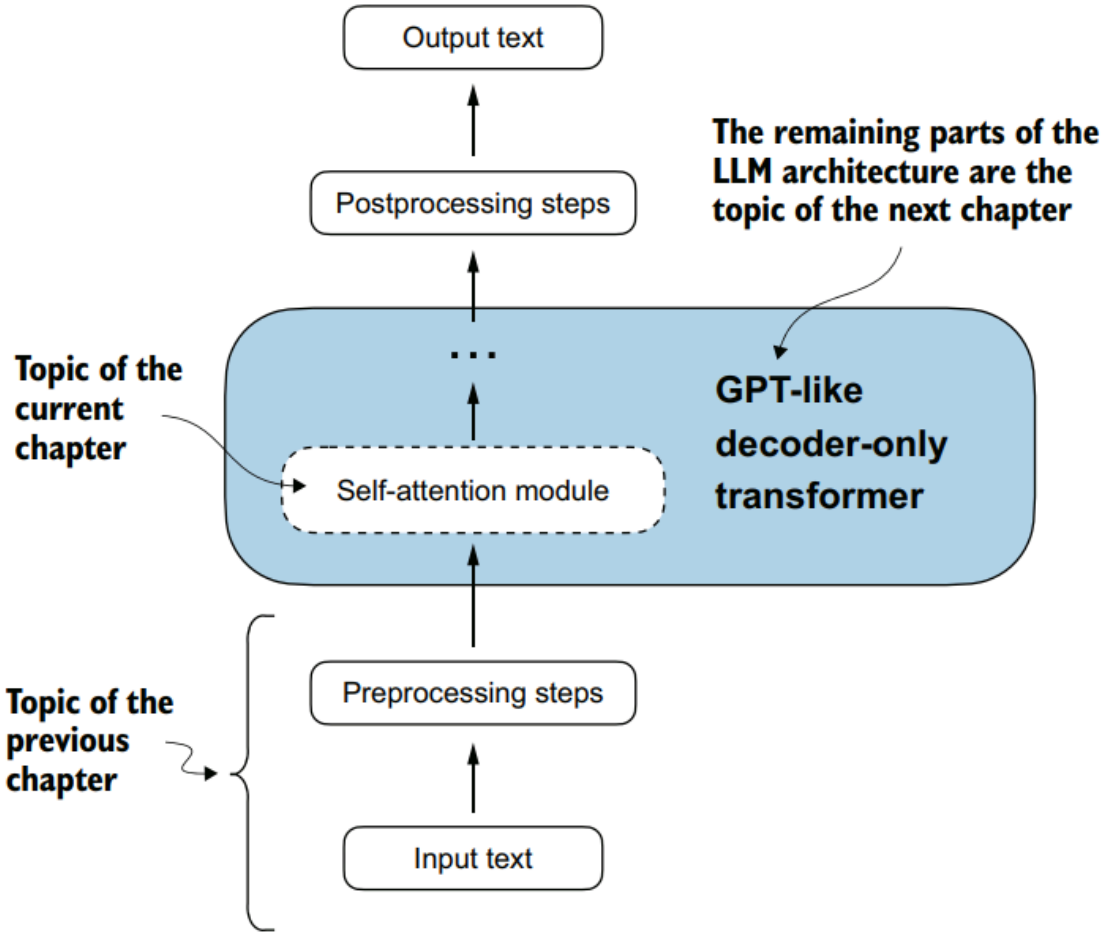

Figure 3.6 Self-attention is a mechanism in transformers used to compute more efficient input representations by allowing each position in a sequence to interact with and weigh the importance of all other positions within the same sequence. In this chapter, we will code this self-attention mechanism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

3.3 使用自注意力机制关注输入的不同部分

"self-attention"中的"self" 指的是这种机制能够通过关联一个单独的输入序列中不同位置来计算注意力权重的能力。它评估和学习输入的不同部分之间的关系和依赖性，例如句子中的单词或者图像中的像素。

这与传统注意力机制是相反的，传统注意力机制的关注点是两个不同序列的元素之间的关系，例如在seq2seq模型中注意力可能是在输入序列和一个输出序列之间，如图3.5描绘的那样。

3.3.1 一个简单的没有可训练权重的注意力机制

让我们首先实现一个简化的自注意力变体，没有任何可训练的权重，如图 3.7 所示。目标是在添加可训练的权重之前说明自注意力的一些关键概念。

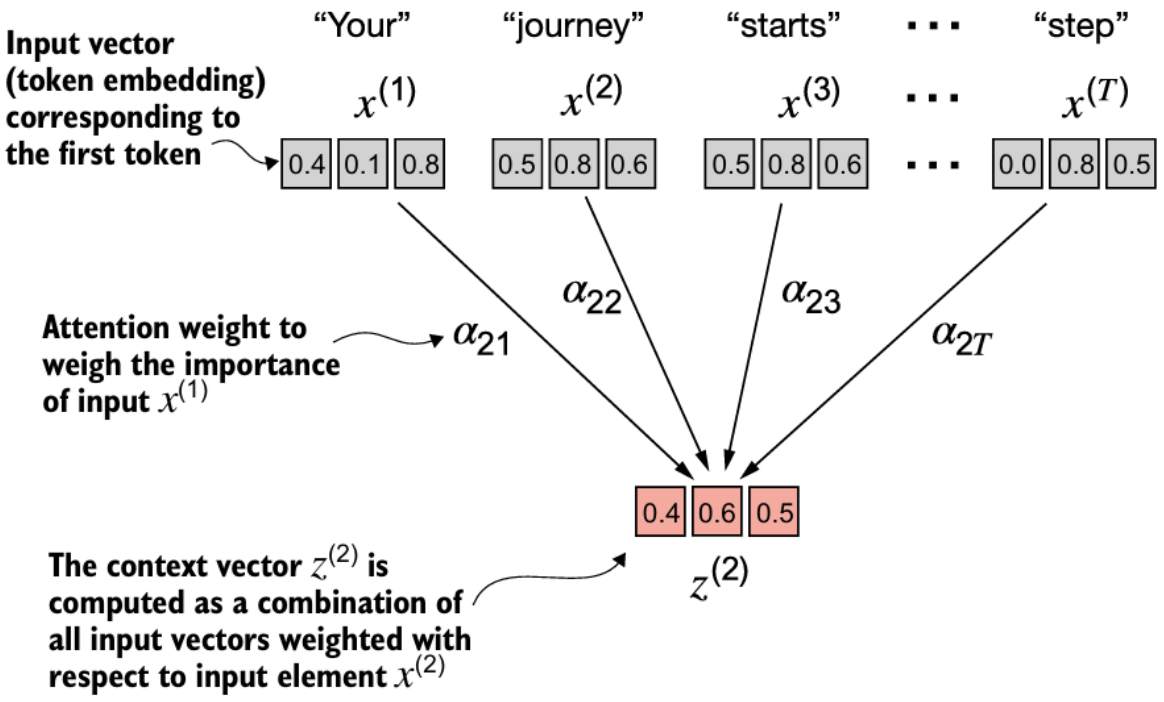

Figure 3.7 The goal of self-attention is to compute a context vector for each input element that combines information from all other input elements. In this example, we compute the context vector $z^{(2)}$. The importance or contribution of each input element for computing $z^{(2)}$ is determined by 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.

图3.7显示了一个输入序列，记作x，包含T个元素记作$x^{(1)}$到$x^{(T)}$。此序列通常表示已转换为标记嵌入的文本，例如句子。

例如，考虑输入文本像"You journey starts with one step."在这个例子中，序列的每个元素，例如$x^{(1)}$，都对应一个表示一个特定token的d维嵌入向量，例如"Your"。图3.7显示这些输入向量都是3维嵌入。

在自注意力机制中，我们的目标是对输入序列的每个元素$x^{(i)}$计算上下文向量$z^{(i)}$。上下文向量可以解释为"enriched"的嵌入向量。

为了说明这个概念，我们注意第二个输入元素的嵌入向量$x^{(2)}$ (对应token "journey")和对应的上下文向量$z^{(2)}$，这个增强的上下文向量$z^{(2)}$，是一个包含关于$x^{(2)}$和所有其它输入元素$x^{(1)}$到$x^{(T)}$的信息的嵌入。

上下文向量在自注意力机制中扮演着重要的角色，它们的目的是创建一个输入序列（如一个句子）中每个元素的enriched表达，通过合并句子中所有其它元素的信息。这对于LLM来说至关重要，因为LLM需要理解句子中单词之间的关系和相关性。稍后，我们将添加可训练的权重，帮助 LLM 学习构建这些上下文向量，以便它们与 LLM 生成下一个标记相关。但首先，让我们实现一种简化的自缩放机制，以一步一步地计算这些权重和生成的上下文向量。

考虑以下输入序列，已经被嵌入到3维向量中。

```
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)
)
```

实现自注意力的第一步是计算中间值ω，称为注意力分数，如图3.8所示。由于空间限制，该图以截断版本显示前面输入张量的值;例如，0.87 被截断为 0.8。在这个截断版本中，单词“journey”和“starts”的嵌入可能会随机出现相似。

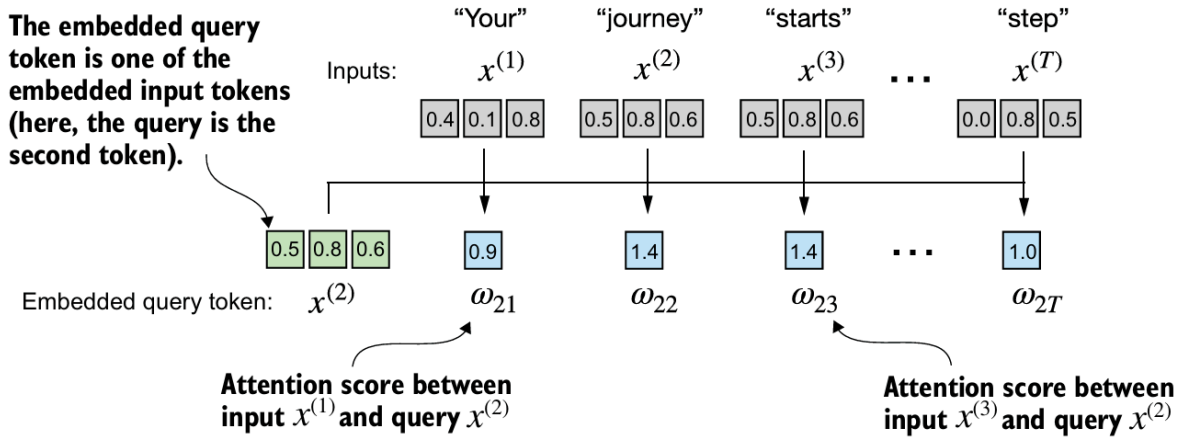

Figure 3.8 The overall goal is to illustrate the computation of the context vector $z^{(2)}$ using the second input element, $x^{(2)}$ as a query. This figure shows the first intermediate step, computing the attention scores ω between the query $x^{(2)}$ and all other input elements as a dot product.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

图3.8说明了如何计算query与各个输入token之间的注意力分数，我们通过计算query即$x^{(2)}$ 与每个（包括自己）token 的点积：

```
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)
```

输出：

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

除了将点积运算视为一种将两个向量组合起来产生标量值的数学工具之外，点积还是相似性的度量，因为它量化了两个向量对齐的紧密程度：较高的点积表示向量之间的对齐或相似程度越高。在自注意力机制的上下文中，点积决定了序列中每个元素关注或“关注”任何其他元素的程度：点积越高，两个元素之间的相似性和注意力分数就越高。

下一步，如图3.9所示，我们正则化我们之前计算的每个注意力分数，正则化的主要目的是取得和为1的注意力分数，正则化是一种有助于在LLM中interpretation和取得训练稳定性的惯例。

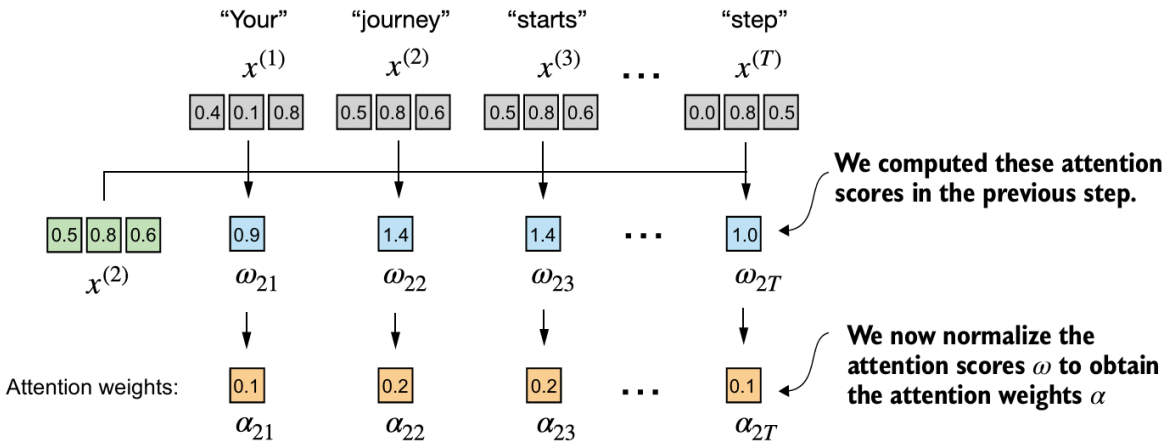

Figure 3.9 After computing the attention scores ω_{21} to ω_{2T} with respect to the input query $x^{(2)}$, the next step is to obtain the attention weights α_{21} to α_{2T} by normalizing the attention scores.

有一种直接的正则化方法：

```
attn_weights_2_tmp = attn_scores_2 / attn_scores_2.sum()
print(f"Attention weights:{attn_weights_2_tmp}")
print(f"Sum:{attn_weights_2_tmp.sum()}")
```

输出：

```
Attention weights:tensor([0.1455, 0.2278, 0.2249, 0.1285, 0.1077, 0.1656])
Sum:1.0000001192092896
```

在实践中，更常用的方法是使用softmax函数来正则化。这个方法在训练期间更擅长处理极端数据和提供更有利的梯度属性：

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

attn_weights_2_naive = softmax_naive(attn_scores_2)
print(f"Attention weights:{attn_weights_2_naive}")
print(f"Sum:{attn_weights_2_naive.sum()}")
```

输出：

```
Attention weights:tensor([0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581])
Sum:1.0
```

此外，softmax 功能确保注意力权重始终为正。这使得输出可以解释为概率或相对重要性，其中权重越高表示重要性越高。

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

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

输出：

```
Attention weights:tensor([0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581])
Sum:1.0
```

现在，我们计算出了正则化的注意力权重，我们来到了最后一步，如图3.10:计算上下文向量$z^{(2)}$——通过将嵌入的输入tokens （即 $x^{(i)}$ ）与对应的注意力权重相乘，然后对结果向量求和。因此，上下文向量 $z^{(2)}$ 是所有输入向量的带权和，由每个输入向量与它对应的注意力权重相乘得到：

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

输出：

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

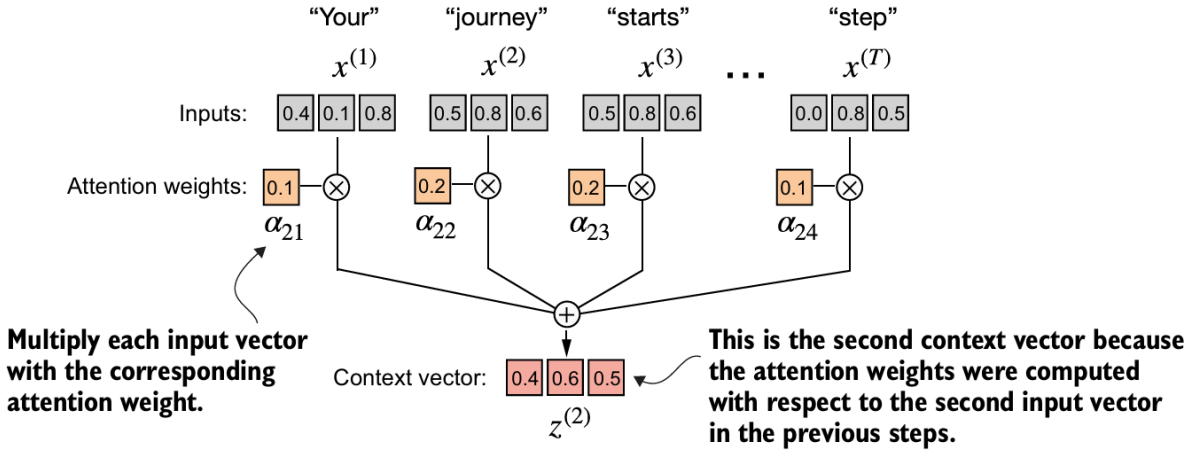

Figure 3.10 The final step, after calculating and normalizing the attention scores to obtain the attention weights for query $x^{(2)}$, is to compute the context vector $z^{(2)}$. This context vector is a combination of all input vectors $x^{(1)}$ to $x^{(T)}$ weighted by the attention weights.

接下来，我们将推广此计算上下文向量的过程，以同时计算所有上下文向量。

3.3.2 对所有输入tokens计算注意力权重

到目前为止，我们已经计算了输入 2 的注意力权重和上下文向量，如图 3.11 中高亮的行所示。现在让我们扩展此计算以计算所有输入的注意力权重和上下文向量。

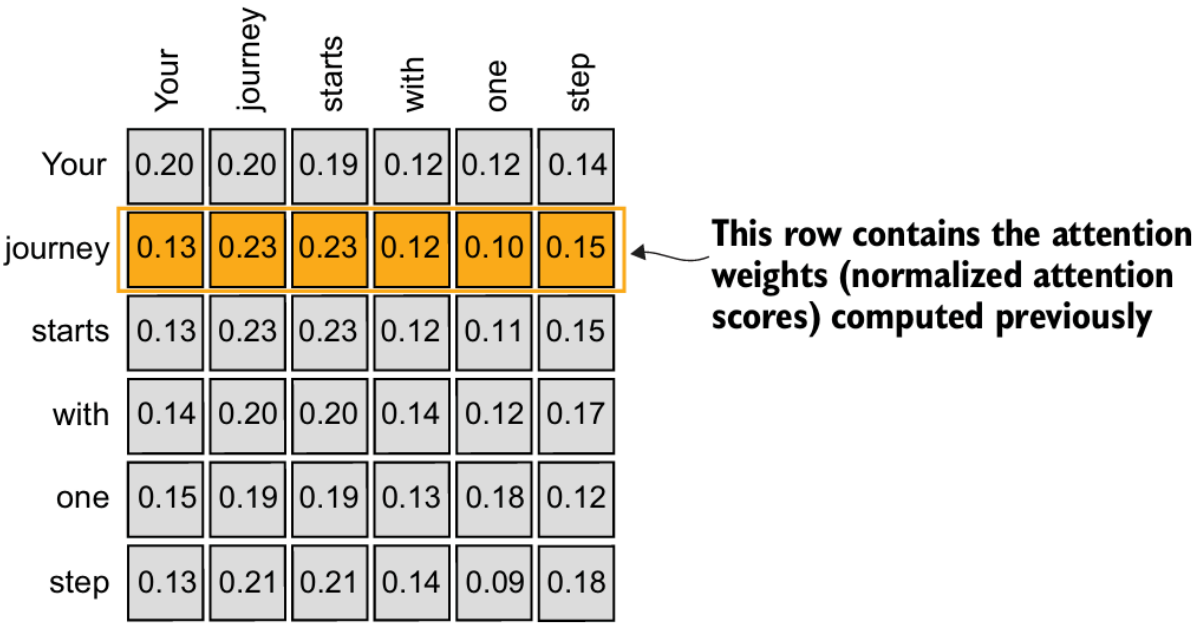

Figure 3.11 The highlighted row shows the attention weights for the second input element as a query. Now we will generalize the computation to obtain all other attention weights. (Please note that the numbers in this figure are truncated to two digits after the decimal point to reduce visual clutter.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

我们遵循着如图3.12的三个步骤来计算所有输入向量的注意力分数:

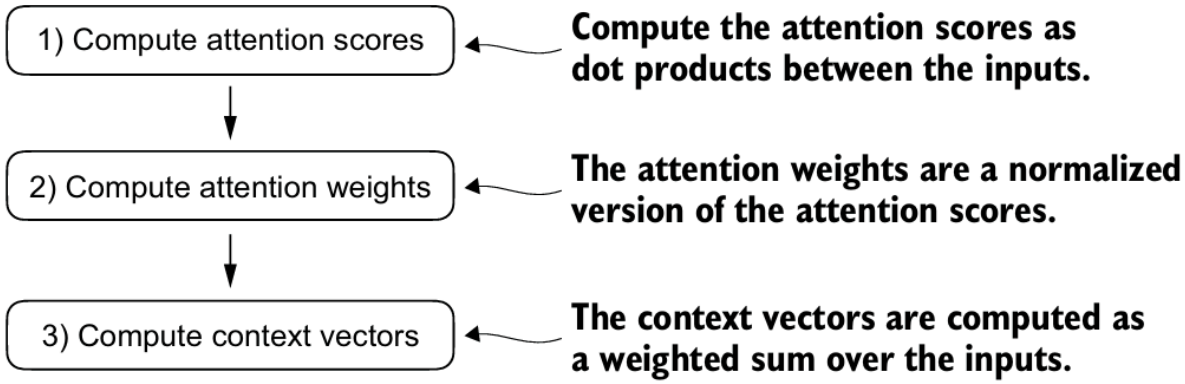

Figure 3.12 In step 1, we add an additional `for` loop to compute the dot products for all pairs of inputs.

```
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)
```

输出:

```
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]])
```

每个元素都表示一对输入之间的注意力分数。

然而，for循环是很慢的，因此我们可以用矩阵乘法取得相同但更快的结果：

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

输出：

```
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]])
```

接下来就是正则化Attention分数，得到Attention权重，如图3.12所示的那样：

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

输出：

```
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]])
```

第三步也是最后一步，我们通过矩阵乘法，使用这些注意力权重计算上下文向量：

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

输出：

```
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]])
```

接下来，我们将加入可训练权重使得LLM可以从数据中学习并在特定任务上改善表现。

3.4 实现带可训练权重的自注意力

我们接下来要实现在原始Transformer架构LLM中的自注意力机制，也被称为缩放的点积注意力。

图 3.13 显示了这种自注意力机制如何适应实施 LLM 的更广泛背景。

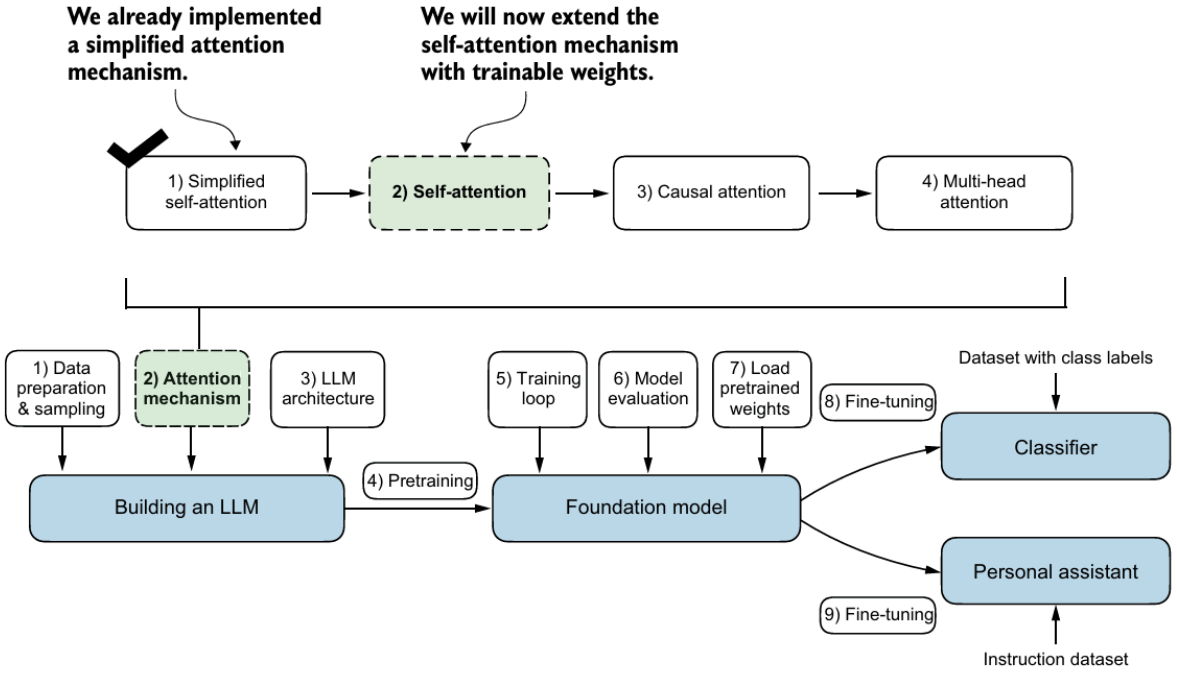

Figure 3.13 Previously, we coded a simplified attention mechanism to understand the basic mechanism behind attention mechanisms. Now, we add trainable weights to this attention mechanism. Later, we will extend this self-attention mechanism by adding a causal mask and multiple heads.

如图3.13所示，带可训练权重的自注意力机制建立在先前的概念上：我们希望将上下文向量计算为特定于某个输入元素的输入向量的加权和。如你将会见到的那样，相较于我们之前实现的简化的注意力机制只会有轻微的不同。

最值得注意的不同点是权重矩阵的引入，在模型训练期间更新。（我们将在第五章训练LLM）

我们将会分两个子节来处理自注意力机制。首先，我们将如之前那样逐步编写代码，然后，我们将把代码整理到一个完整的Python类中，可以被引入LLM架构里。

3.4.1 逐步计算Attention权重

我们将通过引入三个可训练权重矩阵W_q ,W_k , W_v 来逐步实现自注意力机制。这三个矩阵分别被用来将嵌入的输入文本$x^{(i)}$ 投影到query, key, value向量中。如图3.14所示：

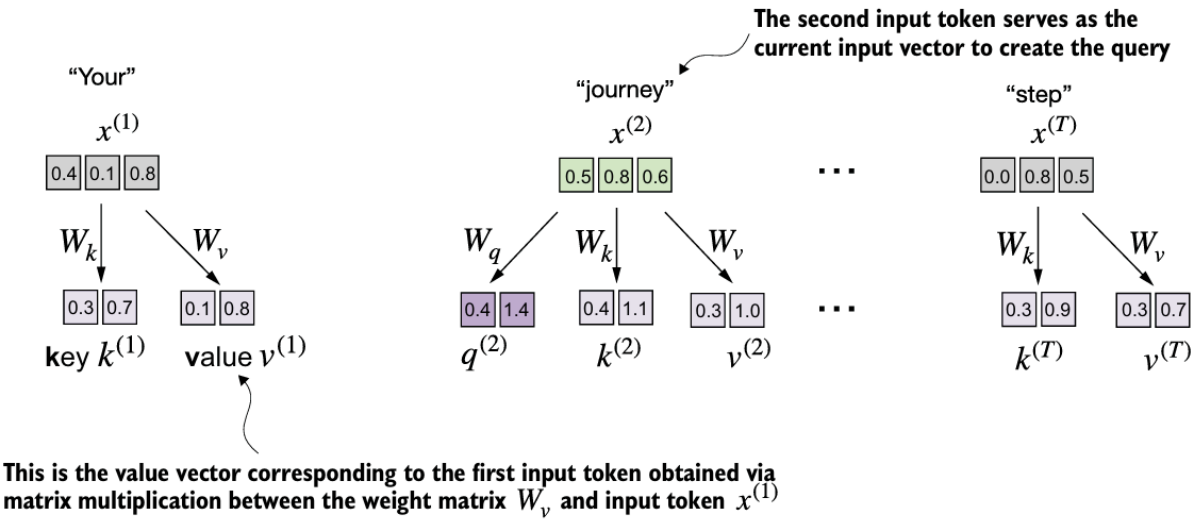

Figure 3.14 In the first step of the self-attention mechanism with trainable weight matrices, we compute query (q), key (k), and value (v) vectors for input elements x. Similar to previous sections, we designate the second input, $x^{(2)}$, as the query input. The query vector $q^{(2)}$ is obtained via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

之前，我们将$x^{(2)}$定义为query来计算简化的注意力权重以计算上下文向量$z^{(2)}$。然后我们泛化到计算所有的上下文向量$z^{(i)}$。相似地，我们这里也只计算一个上下文向量$z^{(2)}$用于演示。然后再修改为计算所有的上下文向量。

首先定义一些变量：

```
x_2 = inputs[1]
d_in = inputs.shape[1] # 输入嵌入的size:3
d_out = 2 # 输出嵌入的size:2
```

注意，GPT-like的模型大多是输入输出维度相同的，这里为了便于跟踪计算，故使用不同的。

然后，我们初始化W_q,W_k,W_v：

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

注意：我们设置 requires_grad=False 以减少输出中的混乱，但如果我们要使用权重矩阵进行模型训练，我们将设置 requires_grad=True 以在模型训练期间更新这些矩阵。

接下来，计算query、key、value向量：

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

输出：

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

由于我们设置的d_out是2，所以输出是两维的。

即便我们的目标仅仅是计算上下文向量$z^{(2)}$，我们仍然需要对于所有输入的key和value值，因为他们被包含在计算关于$q^{(2)}$的注意力权重中，

```
keys = inputs @ W_key
values = inputs @ W_value
print(f"keys.shape is :{keys.shape}")
print(f"values.shape is :{values.shape}")
```

输出：

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

输出表明，我们成功将6个输入从3维投影到了一个2维嵌入空间中。

第二步是计算注意力分数，如图3.15所示：

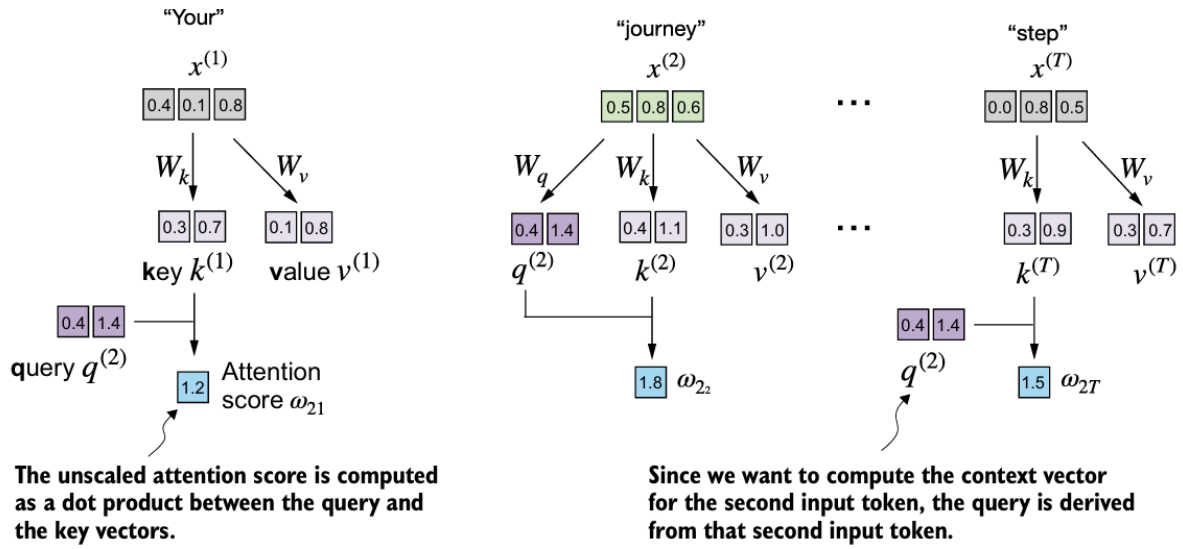

Figure 3.15 The attention score computation is a dot-product computation similar to what we used in the simplified self-attention mechanism in section 3.3. The new aspect here is that we are not directly computing the dot-product between the input elements but using the query and key obtained by transforming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

首先，让我们计算注意力分数$\omega_{2,2}$：

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

输出：

```
tensor(1.8524)
```

然后我们可以泛化到计算所有的注意力分数，通过矩阵乘法：

```
attn_scores_2 = query_2 @ keys.T # query_2的所有注意力分数
print(attn_scores_2)
```

输出：

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

接下来我们想要从注意力分数到注意力权重，如图3.16所示。我们通过使用softmax函数缩放注意力分数来得到注意力权重，然而，现在我们通过将注意力分数除以keys嵌入维度的平方根来缩放注意力分数：

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

输出：

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

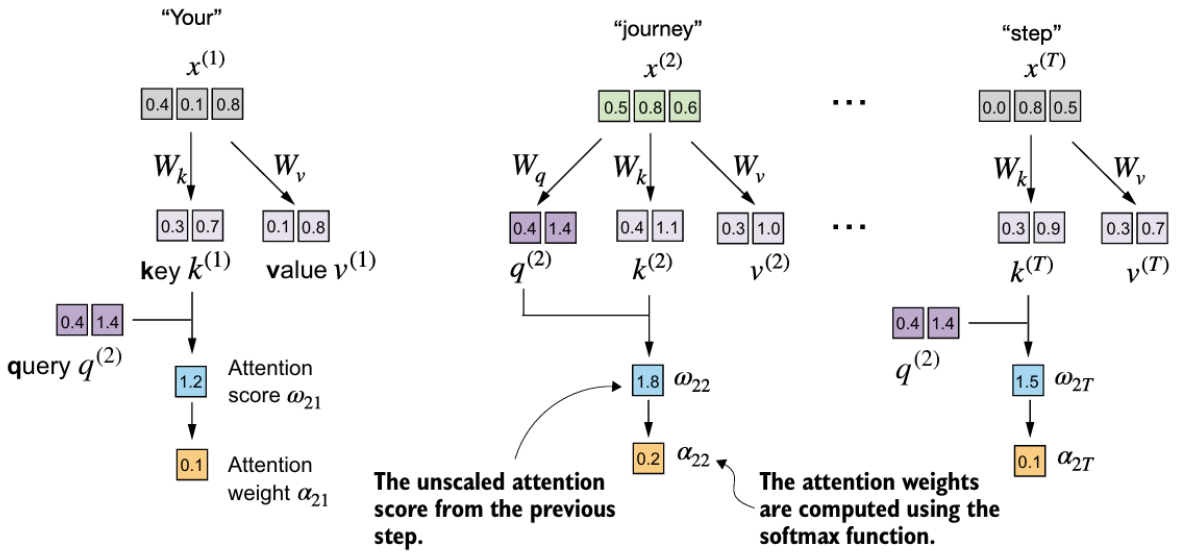

Figure 3.16 After computing the attention scores ω, the next step is to normalize these scores using the softmax function to obtain the attention weights α.

通过嵌入维度大小进行归一化的原因是为了通过避免小梯度来提高训练性能。例如，当放大嵌入维度时（对于类似 GPT 的 LLM，嵌入维度通常大于 1,000），由于应用了 softmax 函数，大点积在反向传播过程中可能会导致非常小的梯度。随着点积的增加，softmax 函数的行为更像是阶跃函数，导致梯度接近于零。这些小梯度会大大减慢学习速度或导致培训停滞不前。嵌入维度的平方根缩放是这种自注意力机制也被称为缩放点积注意力的原因。

最后一步是计算上下文向量，如图3.17所示：

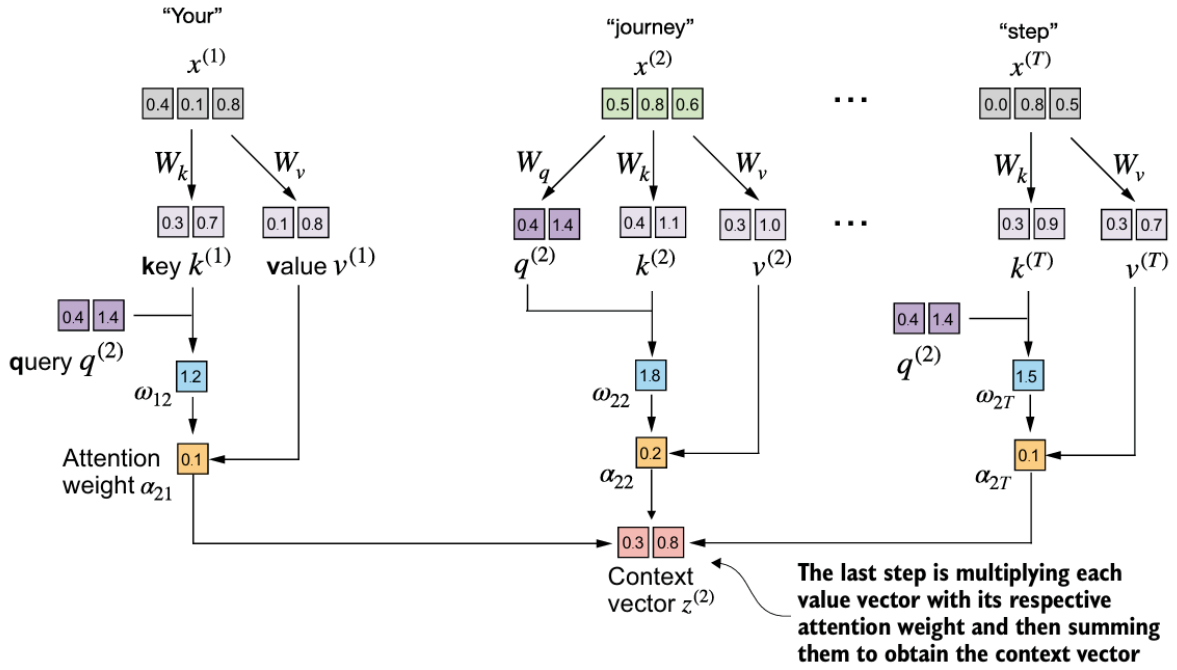

Figure 3.17 In the final step of the self-attention computation, we compute the context vector by combining all value vectors via the attention weights.

我们之前的简化版计算上下文向量的方法是输入向量和注意力权重的加权和，这里我们使用value向量和注意力权重的加权和，其中注意力权重是权重因子，衡量着每个值向量各自的重要性：

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

结果：

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

现在我们计算出了$z^{(2)}$，接下来我们可以计算从$z^{(1)}$到$z^{(T)}$。

为什么是查询、键和值?

注意力机制上下文中的术语"键"、"查询"和"值"是从信息检索和数据库领域借来的，其中类似的概念用于存储、搜索和检索信息。

查询类似于数据库中的搜索查询。它表示模型关注或试图理解的当前项目（例如，句子中的单词或标记）。该查询用于探测输入序列的其他部分，以确定对它们的关注程度。

键类似于用于索引和搜索的数据库键。在注意力机制中，输入序列中的每个项目（例如，句子中的每个单词）都有一个附加的键。这些键用于匹配查询。

此上下文中的值类似于数据库中键值对中的值。它表示输入项的实际内容或表示形式。一旦模型确定了哪些键（以及输入的哪些部分）与查询（当前焦点项）最相关，它就会检索相应的值。

3.4.2 实现一个完整的自注意力Python类

Listing 3.1 一个完整的自注意力Python类

```
import torch
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):
        queries = x @ self.W_query
        keys = x @ self.W_key
        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
```

在此 PyTorch 代码中，SelfAttention_v1 是派生自 nn 的类Module。，它是 PyTorch 模型的基本构建块，为模型层创建和管理提供必要的功能。__init__ 方法初始化查询、键和值的可训练权重矩阵（W_query、W_key 和 W_value），每个矩阵将输入维度d_in转换为输出维度d_out。在前向传递过程中，我们使用 forward 方法，通过乘以查询和键来计算注意力分数 （attn_scores），并使用 softmax 对这些分数进行归一化。最后，我们通过使用这些正则化注意力分数对值进行加权来创建一个上下文向量。

我们可以如下使用：

```
torch.manual_seed(123)

sa_v1 = SelfAttention_v1(d_in,d_out)
print(sa_v1(inputs))
```

输出：

```
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>)
```

图3.18总结了我们刚刚实现的Self-Attention机制：

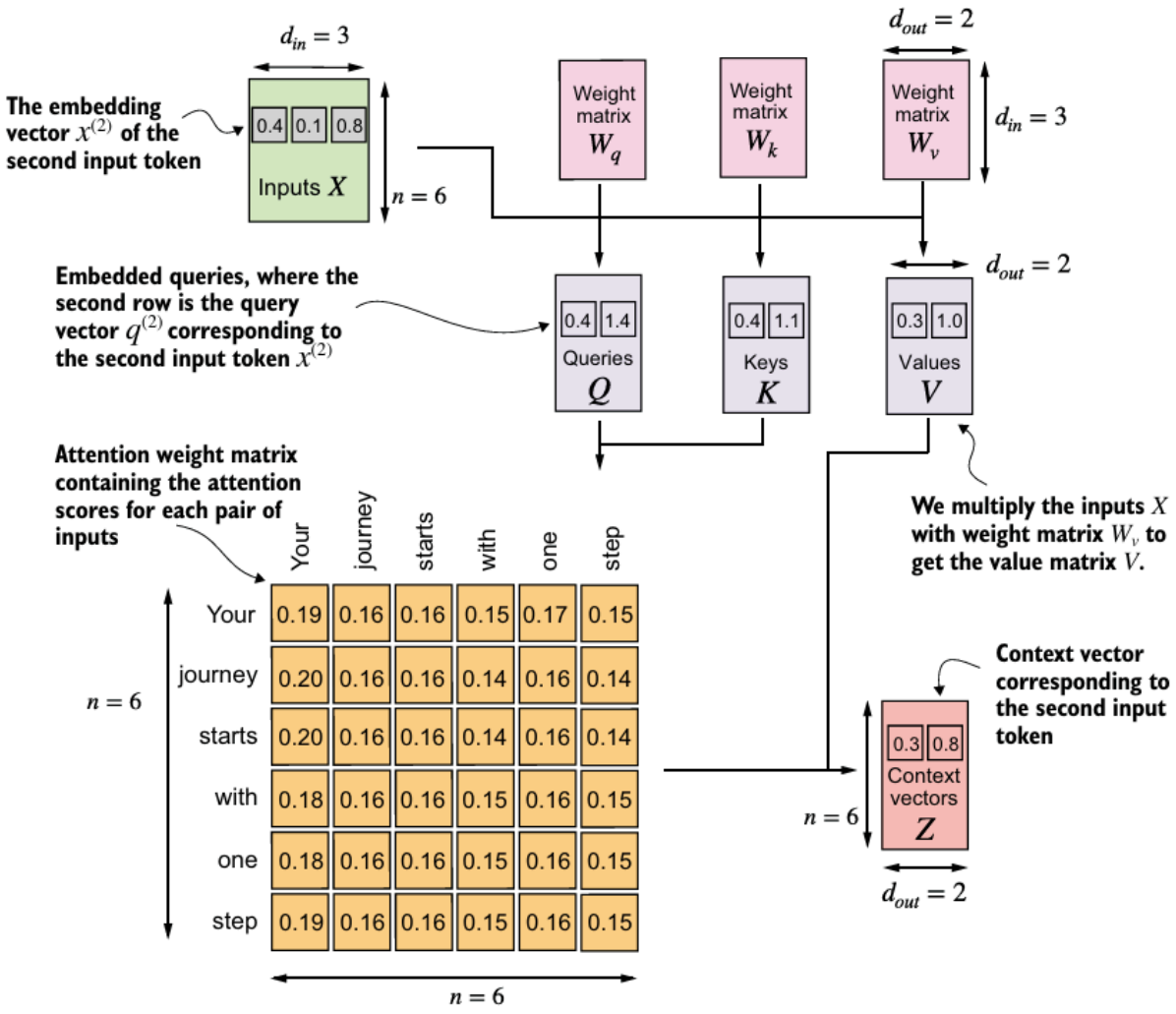

Figure 3.18 In self-attention, we transform the input vectors in the input matrix X with the three weight matrices, W_q, W_k, and W_v. The new compute the attention weight matrix based on the resulting queries (Q) and keys (K). Using the attention weights and values (V), we then compute the context vectors (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%.)

自注意力涉及可训练权重矩阵 W_q、W_k 和 W_v。这些矩阵分别将输入数据转换为查询、键和值，它们是注意力机制的关键组成部分。随着模型在训练过程中接触到更多数据，它会调整这些可训练的权重，我们将在接下来的章节中看到。

我们SelfAttention_v1可以通过利用PyTorch的nn.Linear层进一步改进，当偏置单元被禁用时，它有效地执行矩阵乘法。另外，一个显著的优点是使用nn.Linear不需要手动实现nn.Parameter(torch.rand(...))是因为nn.Linear有一个优化过的权重初始化模式，有助于更稳定、更有效的模型训练。

Listing 3.2 一个使用PyTorch的线性层的自注意力类

```
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):
        queries = self.W_query(x)
        keys = self.W_key(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
```

可以使用如下代码进行测试：

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

输出：

```
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>)
```

请注意，SelfAttention_v1 和 SelfAttention_v2 给出不同的输出，因为它们对权重矩阵使用不同的初始权重，因为 nn.线性使用更复杂的权重初始化方案。

Exercise 3.1

```
sa_v1.W_query = nn.Parameter(sa_v2.W_query.weight.T)
sa_v1.W_key = nn.Parameter(sa_v2.W_key.weight.T)
sa_v1.W_value = nn.Parameter(sa_v2.W_value.weight.T)
print(sa_v1(inputs))
```

接下来，我们将对自注意力机制进行增强，特别关注纳入因果和多头元素。

因果方面涉及修改注意力机制以防止模型访问序列中的未来信息，这对于语言建模等任务至关重要，因为每个单词的预测应该仅依赖于前面的单词。

多头组件涉及将注意力机制分成多个“头”。每个头部学习数据的不同方面，使模型能够同时关注来自不同位置的不同表示子空间的信息。这提高了模型在复杂任务中的性能。

3.5 使用因果注意力隐藏未来的单词

对于许多LLM任务来说，当预测一个序列的下一个token时，你希望自注意力机制只考虑在之前出现到当前位置的tokens。因果注意力（causal attention），也称为masked attention是自注意力的一种特殊形式，当计算注意力分数时，当处理任何给定的token时，它限制模型只考虑一个序列中先前和当前的输入。这与标准的自注意力机制是相反的，标注的自注意力机制允许一次获取整个输入序列。

现在，我们将修改标准的自注意力机制来创建一个因果注意力机制，对于在随后的章节中开发一个大模型是必要的。为了在GPT-like的LLM中实现这个，对于每个处理后的token，我们遮掩未来的tokens，这些tokens在输入文本中在当前token之后到来，如图3.19所示：

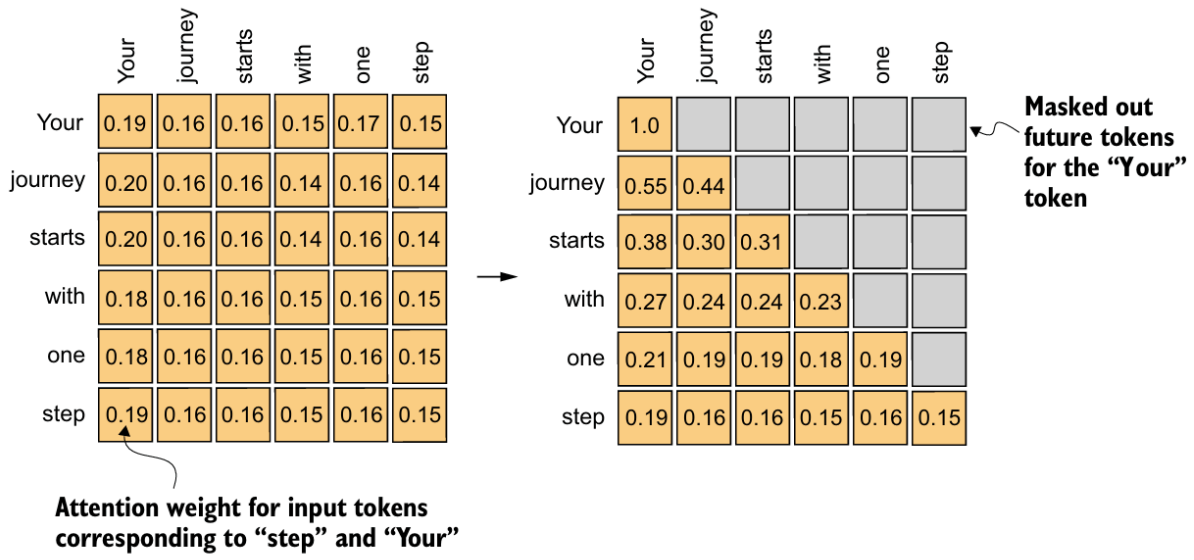

Figure 3.19 In causal attention, we mask out the attention weights above the diagonal such that for a given input, the LLM can't access future tokens when computing the context vectors using the attention weights. For example, for the word "journey" in the second row, we only keep the attention weights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

我们遮掩了对角线上方的注意力权重，然后我们正则化没有被遮掩的注意力权重，从而每一行之和仍然是1。我们后续将实现这个过程。

3.5.1 应用一个因果注意力遮掩

下一步是实现因果注意力的遮掩代码。为了实现应用一个因果注意力遮掩来获得遮掩的注意力权重步骤，如图3.20总结的那样，让我们使用上一节中的注意力分数和权重来编码因果注意力机制。

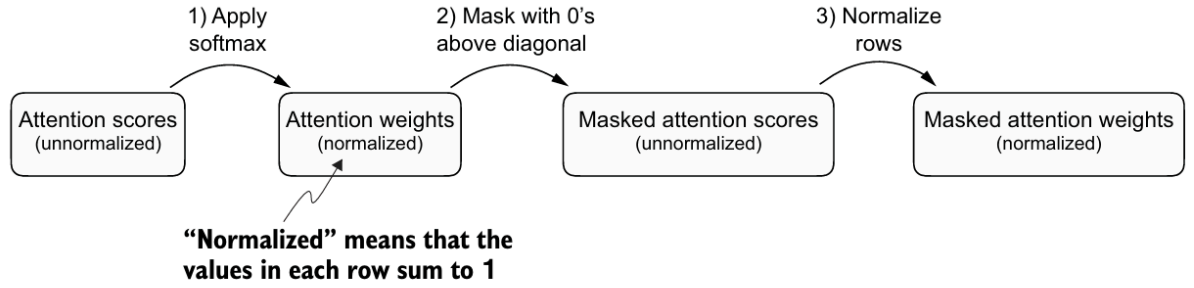

Figure 3.20 One way to obtain the masked attention weight matrix in causal attention is to apply the softmax function to the attention scores, zeroing out the elements above the diagonal and normalizing the resulting matrix.

首先，我们像之前那样获取注意力权重，但是先不用获取上下文向量。

```
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)
```

输出：

```
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>)
```

我们可以通过PyTorch的tril方法来创建一个mask，其中上三角部分都是0，来实现第二步：

```
context_length = len(attn_scores)
mask_simple = torch.tril(torch.ones(context_length, context_length))
print(mask_simple)
```

输出：

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

现在我们可以用这个mask乘注意力权重，使上三角部分变为0：

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

输出：

```
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>)
```

第三步是重新正则化注意力权重以让每一行的和重新变为1，我们可以通过将每行中的每个元素除以每行中的总和来实现这一点：

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

输出：

```
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>)
```

信息泄露

当我们应用掩码然后重新正则化注意力权重时，最初可能看起来来自未来的tokens（我们打算屏蔽）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，关键的见解是，当我们在掩蔽后重新归一注意力权重时，我们本质上正在做的是重新计算较小子集的 SoftMax（因为掩码位置不会影响 SoftMax 值）。

softmax 的数学优雅之处在于，尽管最初包括分母中的所有位置，但在掩蔽和重整化之后，掩蔽位置的影响被抵消了——它们不会以任何有意义的方式对 softmax 分数做出贡献。

简单来说，在掩蔽和重整化之后，注意力权重的分布就好像它只是在一开始就在未掩蔽的位置之间计算的。这确保了未来（或其他被屏蔽的）tokens不会像我们预期的那样泄露信息。

虽然我们可以在这时结束因果注意力的实现，但我们仍然可以改进它。让我们采用 softmax 函数的数学属性，并在更少的步骤中更有效地实现掩码注意力权重的计算，如图 3.21 所示。

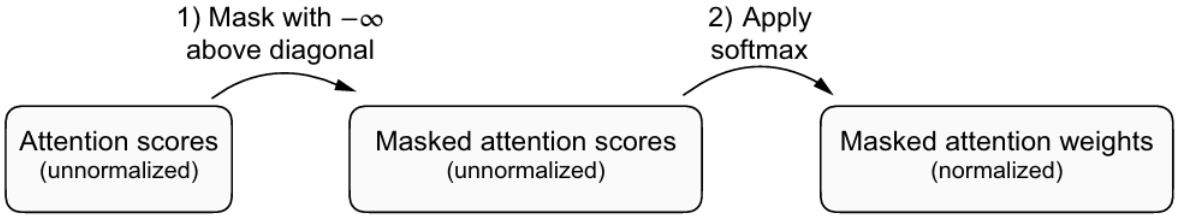

Figure 3.21 A more efficient way to obtain the masked attention weight matrix in causal attention is to mask the attention scores with negative infinity values before applying the softmax function.

softmax 函数将其输入转换为概率分布。当负无穷大值 （-∞） 存在于一行中时，softmax 函数将它们视为零概率。（从数学上讲，这是因为 $e^{-\infty}$接近 0。）

我们可以通过创建一个在对角线上方为 1 的掩码，然后用负无穷大 （-inf） 值替换这些 1 来实现这种更有效的掩码“技巧”：

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

输出：

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

```

```
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>)
```

现在我们需要做的就是将 softmax 函数应用于这些屏蔽结果，我们就完成了：

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

输出：

```
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>)
```

我们现在可以使用修改后的注意力权重通过 context_vec = attn_weights @ values来计算上下文向量，如第 3.4 节所示。然而，我们将首先介绍对因果注意力机制的另一个小调整，该调整有助于在训练 LLM 时减少过度拟合。

3.5.2 使用dropout遮掩额外的注意力权重

深度学习中的 Dropout 是一种在训练过程中忽略随机选择的隐藏层单元的技术，从而有效地将它们“丢弃”。该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通风过度拟合。需要强调的是，Dropout仅在训练期间使用，之后会禁用。

在 Transformer 架构中，包括 GPT 等模型，注意力机制中的dropout通常在两个特定时间应用：在计算注意力权重之后或将注意力权重应用于values向量之后。在这里，我们将在计算注意力权重后应用 dropout掩码，如图 3.22 所示，因为它是实践中更常见的变体。

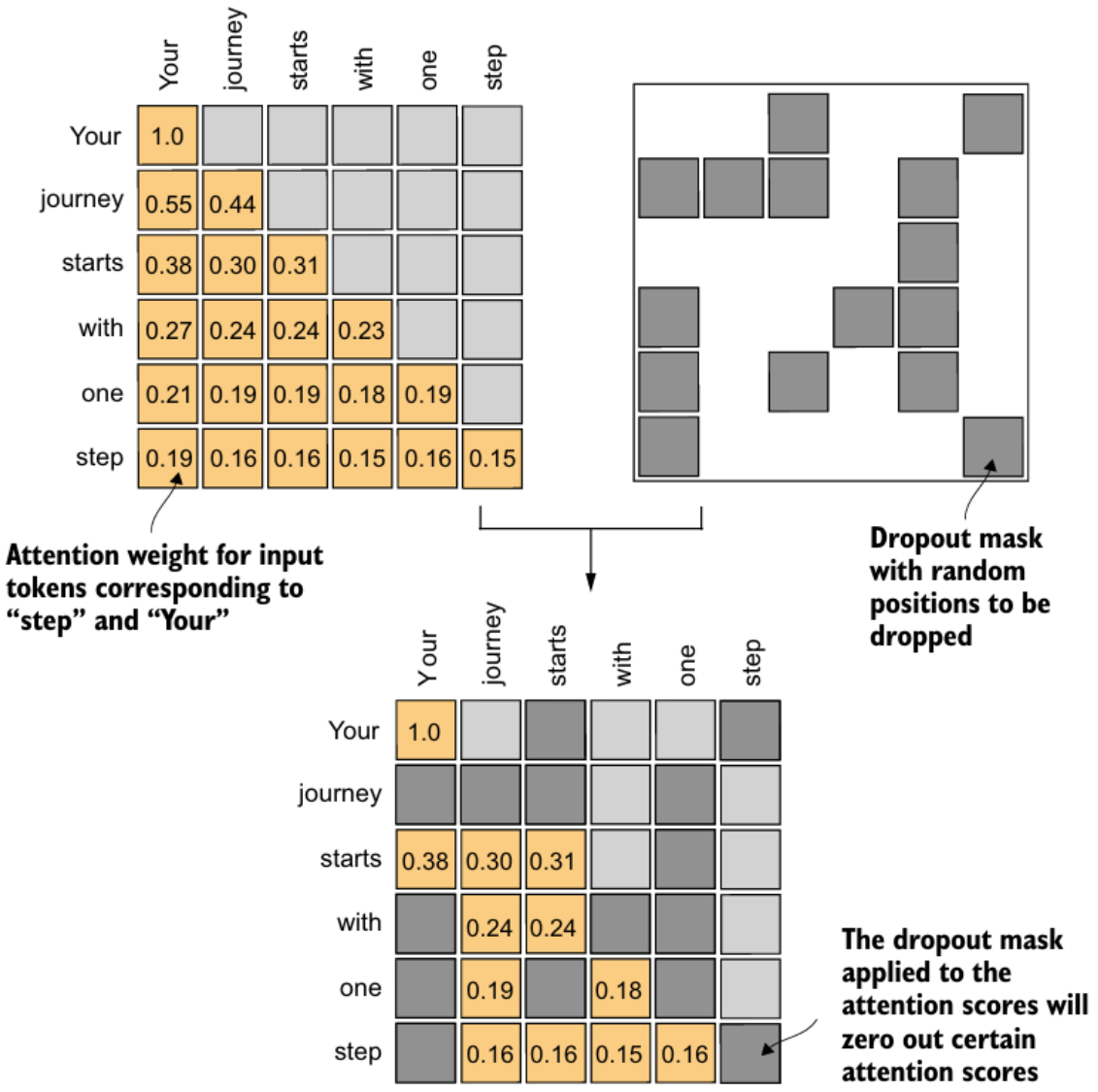

Figure 3.22 Using the causal attention mask (upper left), we apply an additional dropout mask (upper right) to zero out additional attention weights to reduce overfitting during training.

在下面的代码示例中，我们使用 50% 的dropout率，这意味着屏蔽了一半的注意力权重。（当我们在后面的章节中训练 GPT 模型时，我们将使用较低的dropout率，例如 0.1 或 0.2。为了简单起见，我们首先将 PyTorch 的 dropout 实现应用于由全 1 组成的 6 × 6 张量：

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

输出：

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

当以 50% 的dropout率将 dropout 应用于注意力权重矩阵时，矩阵中一半的元素被随机设置为零。为了补偿活性元素的减少，矩阵中剩余元素的值被放大 1/0.5 = 2 的系数。

这种扩展对于保持注意力权重的整体平衡至关重要，确保注意力机制的平均影响在训练和推理阶段保持一致。

现在让我们将 dropout 应用于注意力权重矩阵本身：

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

输出：

```
tensor([[2.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.8966, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.6206, 0.0000, 0.0000, 0.0000],
        [0.5517, 0.4921, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.4350, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.3327, 0.0000, 0.0000, 0.0000, 0.0000]],
       grad_fn=<MulBackward0>)
```

在了解了因果注意力和dropout掩蔽之后，我们现在可以开发一个简洁的 Python 类。本课程旨在促进这两种技术的有效应用。

3.5.3 实现一个完整的因果注意力类

现在，我们将把因果注意力和dropout修改合并到我们在第 3.4 节中开发的 SelfAttention Python 类中。然后，该类将作为开发多头注意力的模板，这是我们将实现的最终注意力类。

但在开始之前，让我们确保代码可以处理由多个输入组成的批处理，以便 CausalAttention 类支持我们在第 2 章中实现的数据加载器生成的批处理输出。为简单起见，为了模拟此类批处理输入，我们复制了输入文本示例：

```
batch = torch.stack((inputs,inputs),dim=0) # 由两个inputs堆叠而成，每个inputs有6个token，每个token有一个3维的嵌入
print(batch.shape)
```

输出：

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

以下 CausalAttention 类与我们之前实现的 SelfAttention 类类似，只是我们添加了 dropout 和 causal mask 组件。

Listing 3.3 一个完整的因果注意力类

```
class CausalAttention(nn.Module):
    def __init__(self, d_in, d_out, context_length,
                 dropout, 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)
```

```
        self.dropout = nn.Dropout(dropout) # 增加了一个dropout层
        self.register_buffer( # 后续会解释
            'mask',
            torch.triu(torch.ones(context_length, context_length),
            diagonal=1)
        )

    def forward(self, x):
        b, num_tokens, d_in = x.shape # b:批次的数量； num_tokens:每一批有几个tokens; d_in:输入维度，即每个token是几维的嵌入
        queries = self.W_query(x)
        keys = self.W_key(x)
        values = self.W_value(x)
            # x.shape: b,n,d_in ; W_query.shape: d_in, d_out
            # keys = x @ W_key, keys.shape: b,n,d_out
            # queries.shape: b,n,d_out
            # attn_scores = queries @ keys.transpose(1,2), attn_scores.shape: b,n,n
        attn_scores = queries @ keys.transpose(1,2) # b,n,n
        # 在 PyTorch 中，带有尾随下划线的作就地执行，避免不必要的内存副本。
        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
```

虽然此时所有添加的代码行都应该很熟悉，但我们现在在 __init__ 方法中添加了一个 self .register_buffer（） 调用。在 PyTorch 中使用 register_buffer 并不是所有用例都必须的，但在这里提供了几个优势。例如，当我们在 LLM 中使用 CausalAttention 类时，缓冲区会与我们的模型一起自动移动到适当的设备（CPU 或 GPU），这意味着我们不需要手动确保这些张量与您的模型参数位于同一设备上，从而避免设备不匹配错误。

可以如下使用：

```
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)
```

输出：

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

生成的上下文向量是一个三维张量，其中每个标记现在由二维嵌入表示

图 3.23 总结了我们迄今为止所取得的成就。我们专注于神经网络中因果注意力的概念和实现。接下来，我们将扩展这个概念，实现一个并行实现多种因果注意力机制的多头注意力模块

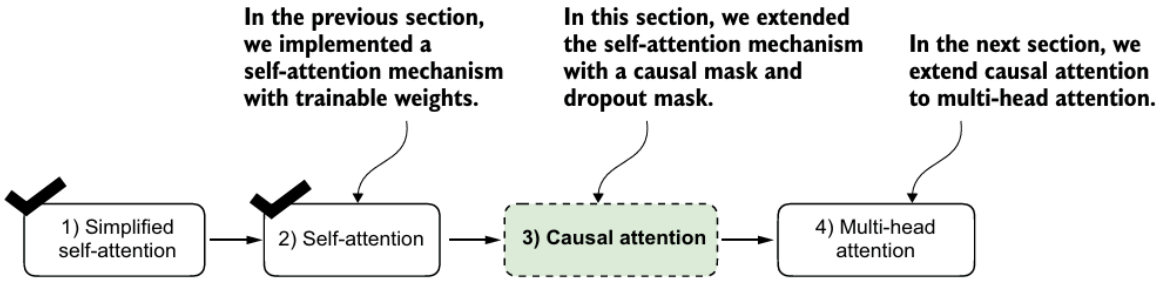

Figure 3.23 Here's what we've done so far. We began with a simplified attention mechanism, added trainable weights, and then added a causal attention mask. Next, we will extend the causal attention mechanism and code multi-head attention, which we will use in our LLM.

我们的最后一步是将先前实现的因果注意力类扩展到多个头部。这也称为多头注意力。

术语"多头"是指将注意力机制划分为多个"头"，每个"头"独立运行。在这种情况下，单个因果注意力模式可以被认为是单头注意力，其中只有一组注意力权重按顺序处理输入。

我们将解决这种从因果注意力到多头注意力的扩展。首先，我们将通过堆叠多个因果注意力模块来直观地构建一个多头注意力模块。然后，我们将以更复杂但计算效率更高的方式实现相同的多头注意力模块。

3.6.1 堆叠多个单头注意力层

实际上，实现多头注意力涉及创建多个自注意力机制实例（见图 3.18），每个实例都有自己的权重，然后组合它们的输出。使用自注意力机制的多个实例可能会产生大量计算，但对于基于 Transformer 的 LLM 等模型所熟知的复杂模式识别至关重要。

图 3.24 说明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，如图 3.18 所示，堆叠在一起。

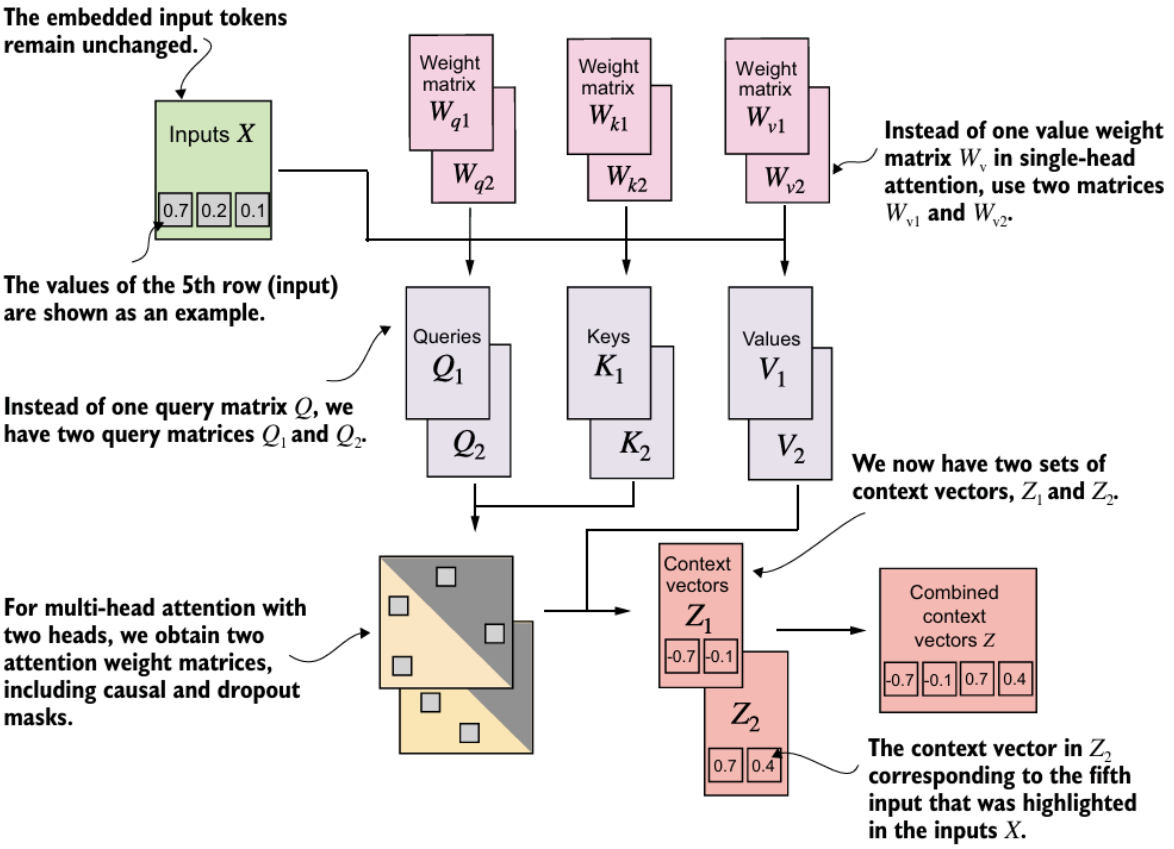

Figure 3.24 The multi-head attention module includes two single-head attention modules stacked on top of each other. So, instead of using a single matrix W_v for computing the value matrices, in a multi-head attention module with two heads, we now have two value weight matrices: W_{v1} and W_{v2}. The 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.

如前所述，多头注意力背后的主要思想是使用不同的、学习的线性投影（将输入数据（如注意力机制中的查询、键和值向量）乘以权重矩阵的结果，多次（并行）运行注意力机制。在代码中，我们可以通过实现一个模拟的 MultiHeadAttentionWrapper 类来实现这一点，该类堆叠我们之前实现的 CausalAttention 模块的多个实例。

Listing 3.4 实现多头注意力的包装类

```
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)
```

例如，如果我们将这个 MultiHeadAttentionWrapper 类与两个注意力头（通过 num_heads=2）和 CausalAttention 输出维度 d_out=2 一起使用，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。

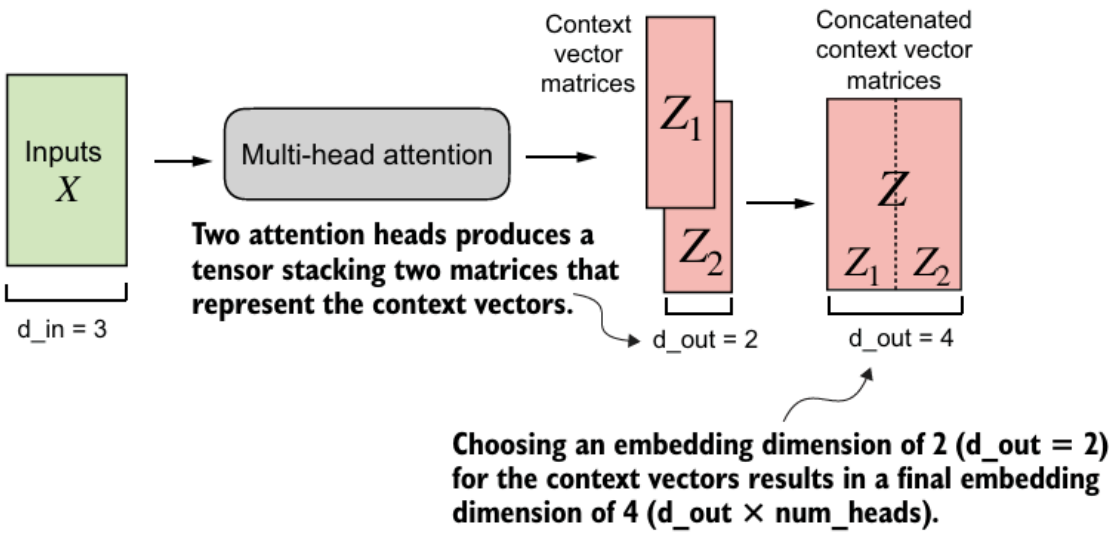

Figure 3.25 Using the `MultiHeadAttentionWrapper`, we specified the number of attention heads (`num_heads`). If we set `num_heads=2`, as in this example, we obtain a tensor with two sets of context vector matrices. In each context vector matrix, the rows represent the context vectors corresponding to 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.

为了通过一个具体的例子进一步说明这一点，我们可以使用类似于之前的 CausalAttention 类的 MultiHeadAttention Wrapper 类：

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

输出：

```
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])
```

生成的context_vecs张量的第一个维度是 2，因为我们有两个输入文本（输入文本是重复的，这就是为什么这些文本的上下文向量完全相同）。第二个维度是指每个输入中的 6 个tokens。第三个维度是指每个token的四维嵌入。

练习 3.2 返回二维嵌入向量

更改 MultiHeadAttentionWrapper（...， num_ heads=2） 调用的输入参数，使输出上下文向量是二维的，而不是四维的，同时保持设置 num_heads=2。提示：您不必修改类实现;您只需更改其他输入参数之一。

答：更改d_out为1即可。

到目前为止，我们已经实现了一个 MultiHeadAttentionWrapper，它结合了多个单头注意力模块。但是，这些是通过 forward 方法中的 [head（x） for head in self.heads] 按顺序处理的。我们可以通过并行处理头来改进此实现。实现这一目标的一种方法是通过矩阵乘法同时放置所有注意力头的输出。

3.6.2 使用权重分解实现多头注意力

到目前为止，我们已经创建了一个 MultiHeadAttentionWrapper，通过堆叠多个单头注意力模块来实现多头注意力。这是通过实例化和组合多个 CausalAttention 对象来完成的。

我们可以将这些概念组合到一个 MultiHeadAttention 类中，而不是维护两个单独的类，即 MultiHeadAttentionWrapper 和 CausalAttention。此外，除了将 MultiHeadAttentionWrapper 与 Causal Attention 代码合并之外，我们还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 CausalAttention 对象 （self.heads） 列表来实现多个头，每个对象代表一个单独的 attention head。CausalAttention 类独立执行注意力机制，每个头的结果都是串联的。相比之下，以下 MultiHeadAttention 类将多头功能集成到单个类中。它通过重塑投影的查询、键和值张量将输入拆分为多个头，然后在计算注意力后组合这些头的结果。在进一步讨论之前，让我们先看看 MultiHeadAttention 类。

Listing 3.5 一个高效的多头注意力类

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

        self.d_out = d_out
        self.num_heads = num_heads
        self.head_dim = d_out // num_heads # 减少投影维度以匹配期望的输出维度

        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) # 使用一个线性层来整合各个头的输出
        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
        queries = self.W_query(x)
        keys = self.W_key(x)
        values = self.W_value(x)
        # queries, keys, values的形状都是b, num_tokens, d_out
        # 把d_out拆解成num_heads和head_dim, 即: d_out = num_heads * head_dim
        # 拆解后的queries,keys和values的形状变为b, num_tokens, num_heads,
head_dim
        queries = queries.view(b, num_tokens, self.num_heads, self.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)
        # 调换num_tokens和num_heads的位置，便于后续的计算
        # 调换后形状变为: b,num_heads,num_tokens,head_dim
        queries = queries.transpose(1,2)
        keys = keys.transpose(1,2)
        values = values.transpose(1,2)
        # 对每一个头进行点积，计算注意力分数
        attn_scores = queries @ keys.transpose(2,3) # b,nh,nt,d @ b,nh,d,nt
= b,nh,nt,nt
        mask_bool = self.mask.bool()[:num_tokens, :num_tokens] # 掩码被截断为
token数,mask_bool是一个上三角为true的矩阵

        attn_scores.masked_fill_(mask_bool, -torch.inf) # 使用掩码填充注意力分数

        attn_weights = torch.softmax(
            attn_scores / keys.shape[-1]**0.5 , dim=-1
        )
        attn_weights = self.dropout(attn_weights) # b,nh,nt,nt
        # 计算上下文向量并把形状还原回去
        context_vec = (attn_weights @ values).transpose(1,2) # b,nh,nt,nt @
b,nh,nt,d = b,nh,nt,d -> b,nt,nh,d
        # 把num_heads和head_dim重新结合回d_out
        context_vec = context_vec.contiguous().view(
            b, num_tokens, self.d_out
        )
        # 增加一个可选的线性投影
        context_vec = self.out_proj(context_vec)

        return context_vec
```

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 HeadAttention 类实现了与前面的 MultiHeadAttentionWrapper相同的概念。

总体上，在之前的 MultiHeadAttentionWrapper 中，我们堆叠了多个单头注意力层，并将其组合成一个多头注意力层。MultiHeadAttention 类采用集成方法。它从多头层开始，然后在内部将该层拆分为单独的注意力头，如图 3.26 所示。

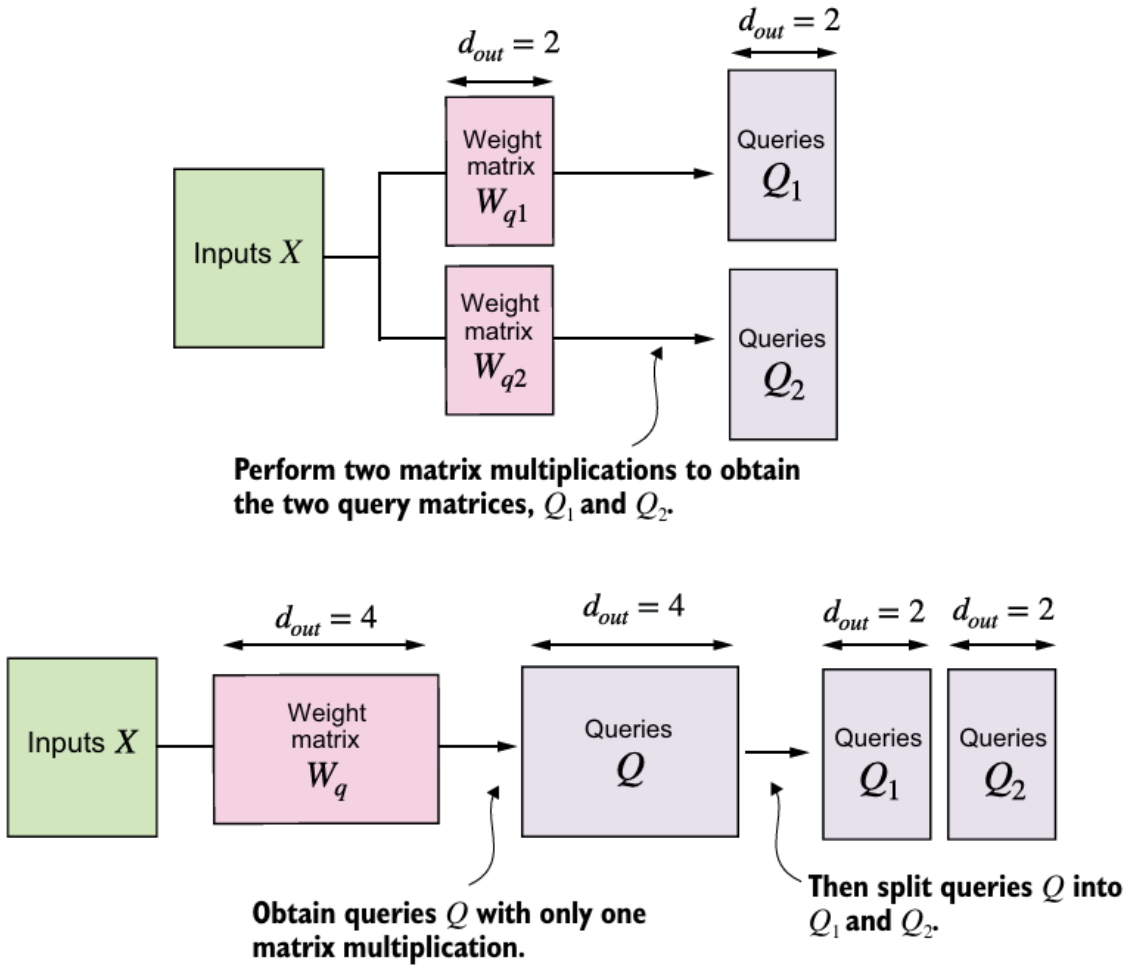

Figure 3.26 In the `MultiHeadAttentionWrapper` class with two attention heads, we initialized two weight matrices, W_{q1} and W_{q2}, and computed two query matrices, Q_1 and Q_2 (top). In the `MultiheadAttention` class, we initialize one larger weight matrix W_q, only perform one matrix multiplication 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.

查询、键和值张量的拆分是通过使用 PyTorch 的 .view 和 .transpose 方法进行张量重组和转置作来实现的。首先转换输入（通过查询、键和值的线性层），然后重塑以表示多个头。

关键做法是将d_out维拆分为num_heads和head_dim，其中head_dim = d_out / num_heads。然后使用 .view 方法实现这种拆分：维度张量 （b， num_tokens， d_out） 被重塑为维度 （b， num_tokens， num_heads， head_dim）。

然后转置张量以将num_heads维带到num_标记维度之前，从而产生 （b， num_heads， num_tokens， head_dim） 的形状。这种转置对于正确对齐不同头部的查询、键和值以及有效地执行批量矩阵乘法至关重要。

为了说明这种批处理矩阵乘法，假设我们有以下张量：

```
a = torch.tensor([[[[0.2745, 0.6584, 0.2775, 0.8573],
                    [0.8993, 0.0390, 0.9268, 0.7388],
                    [0.7179, 0.7058, 0.9156, 0.4340]],
                   [[0.0772, 0.3565, 0.1479, 0.5331],
                    [0.4066, 0.2318, 0.4545, 0.9737],
                    [0.4606, 0.5159, 0.4220, 0.5786]]]])
```

这个张量的形状是： (b, num_heads, num_tokens, head_dim) = (1, 2, 3, 4).

现在，我们在张量本身和张量视图之间执行批量矩阵乘法，我们在其中转置了最后两个维度，num_tokens 和 head_dim：

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

输出：

```
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]]]])
```

在这种情况下，PyTorch 中的矩阵乘法实现处理四维输入张量，以便在最后两个维度（num_tokens、head_dim）之间执行矩阵乘法，然后对各个头部重复。

例如，上述方法成为单独计算每个头的矩阵乘法的更紧凑的方法：

```
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)
```

输出：

```
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]])
```

结果与我们使用批量矩阵乘法打印（a @ a.transpose（2， 3））时获得的结果完全相同。

继续使用 MultiHeadAttention，在计算注意力权重和 上下文向量后，将来自所有头部的上下文向量转置回形状 （b,num_tokens,num_heads,head_dim）。然后，这些向量被重塑（压平）为（b，num_tokens，d_out）形状，有效地组合了所有头的输出。

此外，我们在组合头部后向 Multi HeadAttention 添加了一个输出投影层 （self.out_proj），这在 Causal Attention 类中不存在。此输出投影层不是绝对必要的,但它在许多 LLM 架构中很常用，这就是为什么为了完整起见，我在这里添加了它。

尽管由于张量的额外重塑和转置，MultiHeadAttention 类看起来比 MultiHeadAttentionWrapper 更复杂，但它效率更高。原因是我们只需要一个矩阵乘法来计算键，例如，keys = self.W_key（x）（查询和值也是如此）。在 MultiHeadAttentionWrapper 中，我们需要对每个注意力头重复此矩阵乘法，这是计算上最昂贵的步骤之一。

MultiHeadAttention 类的使用类似于我们之前实现的 SelfAttention 和 CausalAttention 类：

```
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)
```

输出：

```
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])
```

我们现在已经实现了 MultiHeadAttention 类，我们将在实现和训练 LLM 时使用该类。请注意，虽然代码功能齐全，但我使用了相对较小的嵌入大小和注意力头的数量来保持输出的可读性。

相比之下，最小的 GPT-2 模型（1.17 亿个参数）有 12 个 attention head，上下文向量嵌入大小为 768。最大的GPT-2模型（15亿个参数）有25个注意力头，上下文向量嵌入大小为1600。在 GPT 模型中，标记输入和上下文嵌入的嵌入大小相同 （d_in = d_out）。

练习 3.3 初始化 GPT-2 大小的注意力模块

使用 MultiHeadAttention 类，初始化一个多头注意力模块，该模块的注意力头数与最小的 GPT-2 模型（12 个注意力头）相同。此外，请确保使用类似于 GPT-2（768 维）的相应输入和输出嵌入大小。请注意，最小的 GPT-2 模型支持 1,024 个令牌的上下文长度。

```
mhagpt = MultiHeadAttention(768, 768, 1024, 0.0, num_heads=12)
```

总结

- 注意力机制将输入元素转换为增强的上下文向量表示，其中包含有关所有输入的信息。
- 自注意力机制将上下文向量表示计算为输入的加权和。
- 在简化的注意力机制中，注意力权重是通过点积计算的。
- 点积是一种将两个向量逐元素相乘，然后对乘积求和的简洁方法。
- 矩阵乘法虽然不是严格要求的，但通过替换嵌套的 for 循环，帮助我们更高效、更紧凑地实现计算。

- 在 LLM 中使用的自注意力机制（也称为缩放点乘积注意力）中，我们包括可训练的权重矩阵来计算输入的中间转换：查询、值和键。
- 当使用从左到右读取和生成文本的 LLM 时，我们添加了一个因果注意力掩码，以防止 LLM 访问未来的标记。
- 除了对零注意力权重的因果注意力掩码外，我们还可以添加一个 dropout 掩码来减少 LLM 中的过度拟合。
- 基于 transformer 的 LLM 中的注意力模块涉及多个因果注意力实例，称为多头注意力。
- 我们可以通过堆叠多个因果注意力模块实例来创建一个多头注意力模块。
- 创建多头注意力模块的更有效方法涉及批量矩阵乘法。