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)
```

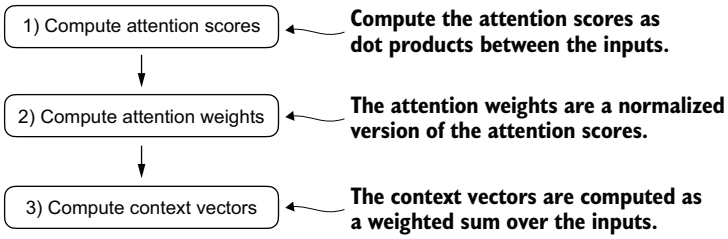

Figure 3.12 In step 1, we add an additional `for` loop to compute the dot products for all pairs of inputs.

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],
```

```
        [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.

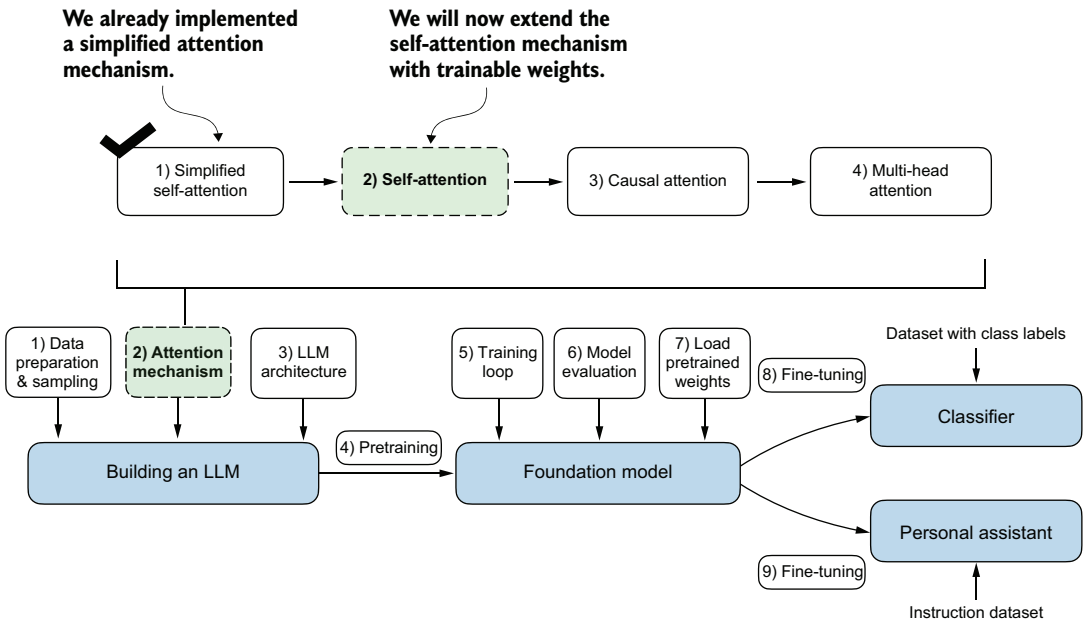

Figure 3.13 Previously, we coded a simplified attention mechanism to understand the basic mechanism behind attention mechanisms. Now, we add trainable weights to this attention mechanism. Later, we will extend this self-attention mechanism by adding a causal mask and multiple heads.