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.

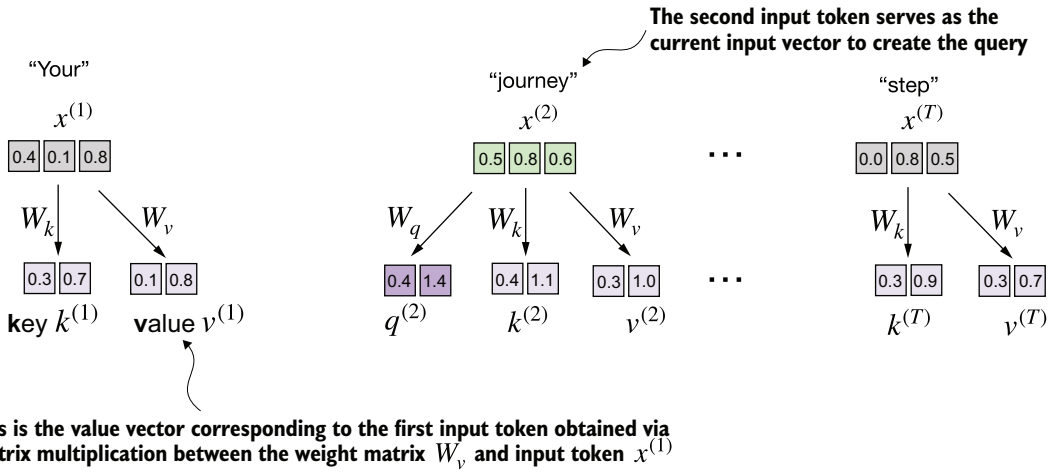
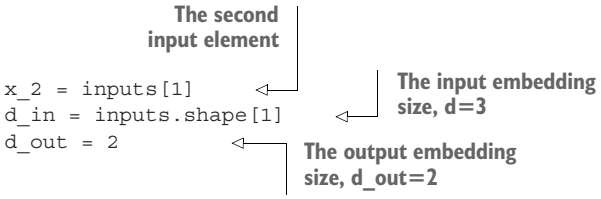


Figure 3.14 In the first step of the self-attention mechanism with trainable weight matrices, we compute query (*q*), key (*k*), and value (*v*) vectors for input elements *x*. Similar to previous sections, we designate the second input, $x^{(2)}$, as the query input. The query vector $q^{(2)}$ is obtained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

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:



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.

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.

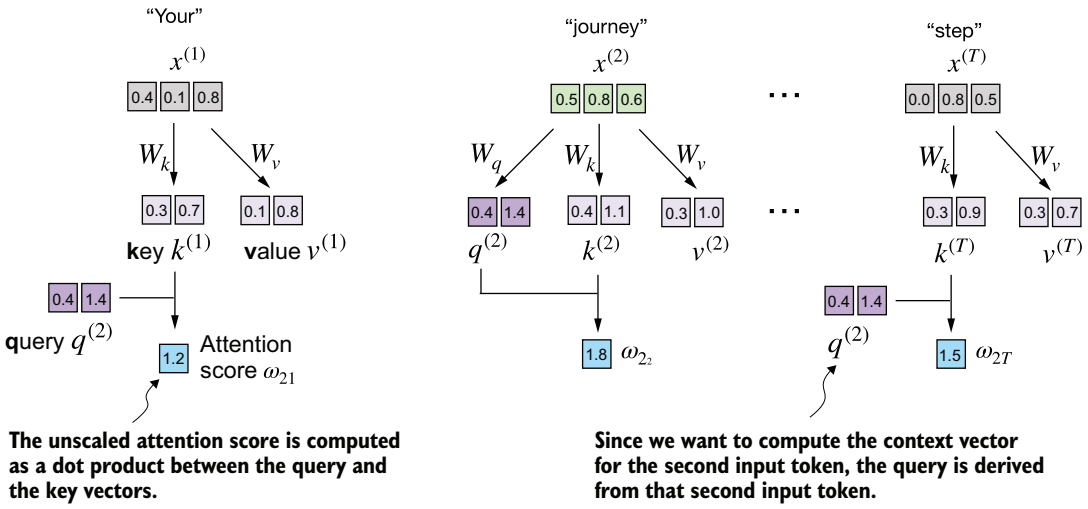


Figure 3.15 The attention score computation is a dot-product computation similar to what we used in the simplified self-attention mechanism in section 3.3. The new aspect here is that we are not directly computing the dot-product between the input elements but using the query and key obtained by transforming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