

```

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`

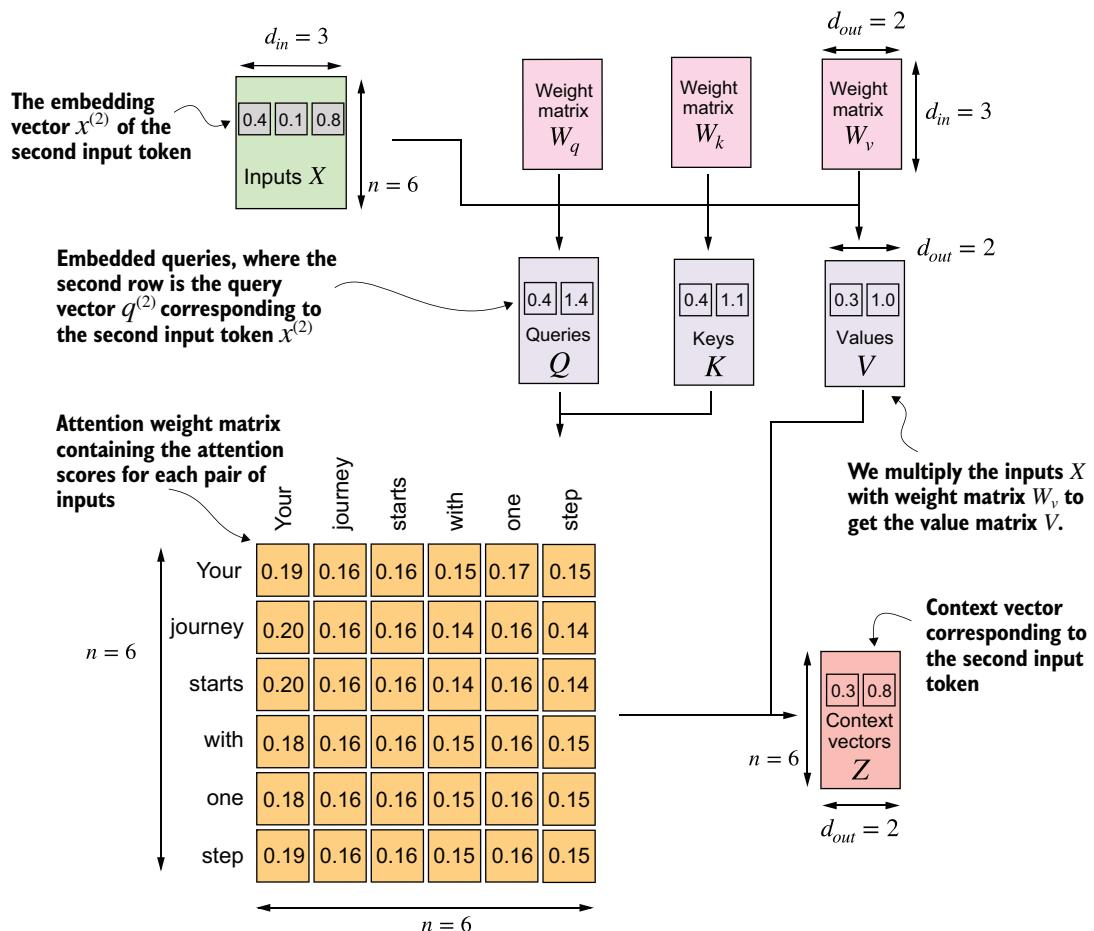


Figure 3.18 In self-attention, we transform the input vectors in the input matrix X with the three weight matrices, W_q , W_k , and W_v . Then we compute the attention weight matrix based on the resulting queries (Q) and keys (K). Using the attention weights and values (V), we then compute the context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