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.

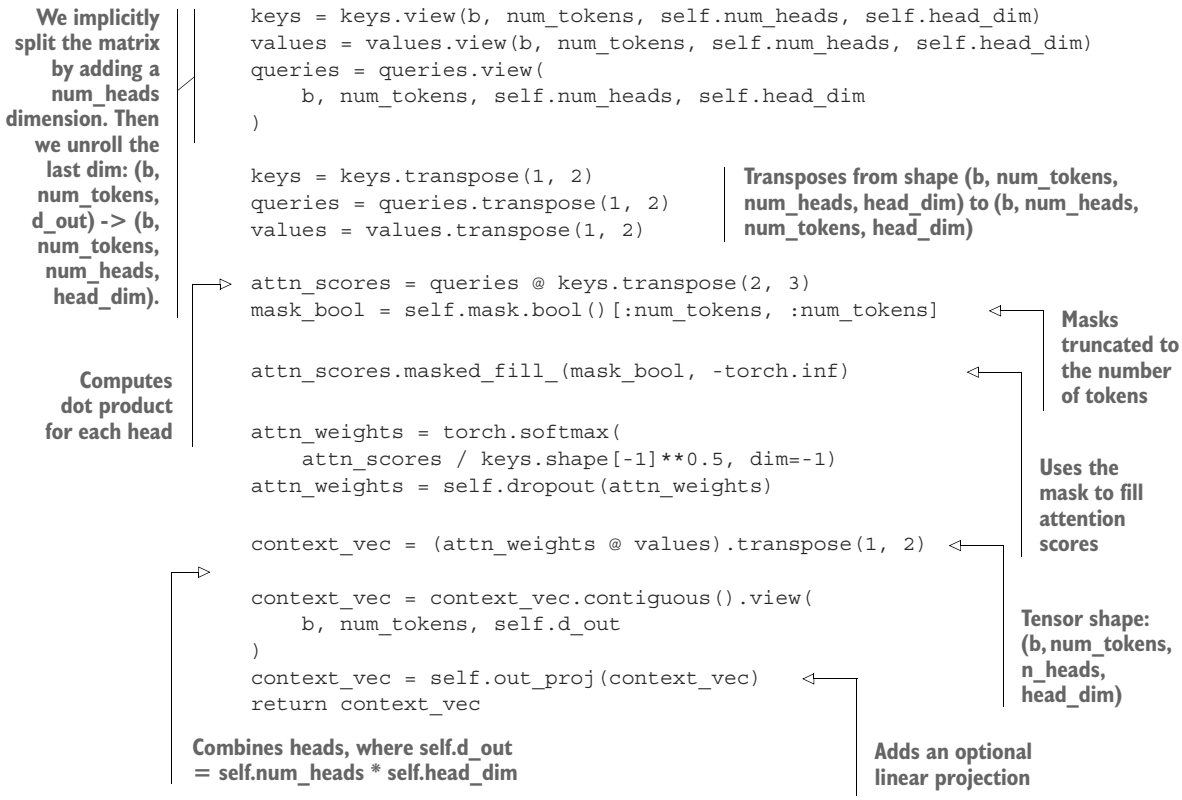
Listing 3.5 An efficient multi-head attention class

```
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)



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)`.

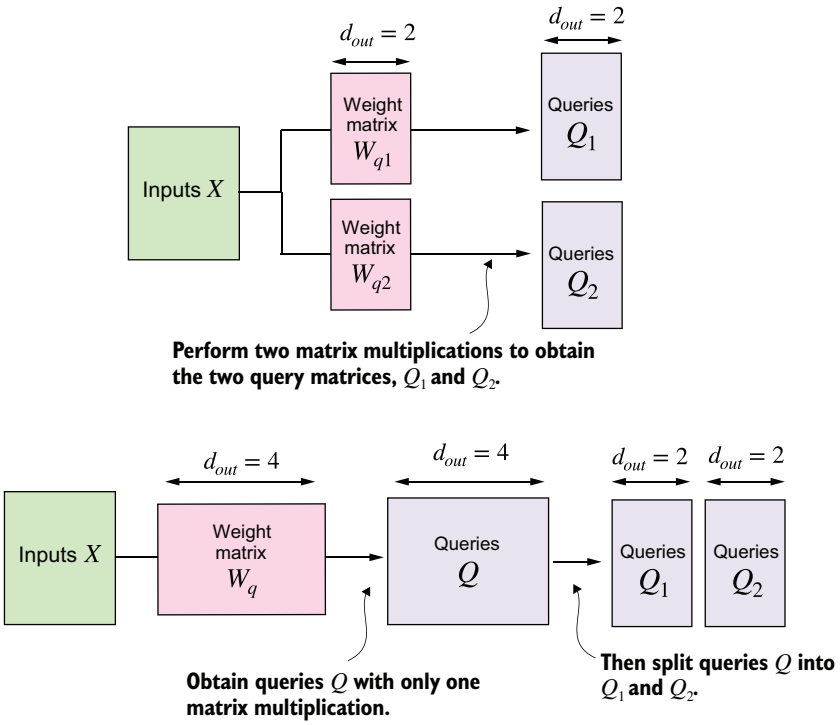


Figure 3.26 In the `MultiHeadAttentionWrapper` class with two attention heads, we initialized two weight matrices, W_{q1} and W_{q2}, and computed two query matrices, Q_1 and Q_2 (top). In the `MultiheadAttention` class, we initialize one larger weight matrix W_q, only perform one matrix multiplication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