Now we perform a batched matrix multiplication between the tensor itself and a view of the tensor where we transposed the last two dimensions, `num_tokens` and `head_dim`:

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

The result is

```
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]]]])
```

In this case, the matrix multiplication implementation in PyTorch handles the four-dimensional input tensor so that the matrix multiplication is carried out between the two last dimensions `(num_tokens, head_dim)` and then repeated for the individual heads.

For instance, the preceding becomes a more compact way to compute the matrix multiplication for each head separately:

```
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)
```

The results are exactly the same results as those we obtained when using the batched matrix multiplication `print(a @ a.transpose(2, 3))`:

```
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]])
```

Continuing with `MultiHeadAttention`, after computing the attention weights and context vectors, the context vectors from all heads are transposed back to the shape `(b, num_tokens, num_heads, head_dim)`. These vectors are then reshaped (flattened) into the shape `(b, num_tokens, d_out)`, effectively combining the outputs from all heads.

Additionally, we added an output projection layer (`self.out_proj`) to `MultiHeadAttention` after combining the heads, which is not present in the `CausalAttention` class. This output projection layer is not strictly necessary (see appendix B for

more details), but it is commonly used in many LLM architectures, which is why I added it here for completeness.

Even though the `MultiHeadAttention` class looks more complicated than the `MultiHeadAttentionWrapper` due to the additional reshaping and transposition of tensors, it is more efficient. The reason is that we only need one matrix multiplication to compute the keys, for instance, `keys = self.W_key(x)` (the same is true for the queries and values). In the `MultiHeadAttentionWrapper`, we needed to repeat this matrix multiplication, which is computationally one of the most expensive steps, for each attention head.

The `MultiHeadAttention` class can be used similar to the `SelfAttention` and `CausalAttention` classes we implemented earlier:

```
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)
```

The results show that the output dimension is directly controlled by the `d_out` argument:

```
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])
```

We have now implemented the `MultiHeadAttention` class that we will use when we implement and train the LLM. Note that while the code is fully functional, I used relatively small embedding sizes and numbers of attention heads to keep the outputs readable.

For comparison, the smallest GPT-2 model (117 million parameters) has 12 attention heads and a context vector embedding size of 768. The largest GPT-2 model (1.5 billion parameters) has 25 attention heads and a context vector embedding size of 1,600. The embedding sizes of the token inputs and context embeddings are the same in GPT models (`d_in = d_out`).

Exercise 3.3 Initializing GPT-2 size attention modules

Using the `MultiHeadAttention` class, initialize a multi-head attention module that has the same number of attention heads as the smallest GPT-2 model (12 attention heads). Also ensure that you use the respective input and output embedding sizes similar to GPT-2 (768 dimensions). Note that the smallest GPT-2 model supports a context length of 1,024 tokens.

Summary

- Attention mechanisms transform input elements into enhanced context vector representations that incorporate information about all inputs.
- A self-attention mechanism computes the context vector representation as a weighted sum over the inputs.
- In a simplified attention mechanism, the attention weights are computed via dot products.
- A dot product is a concise way of multiplying two vectors element-wise and then summing the products.
- Matrix multiplications, while not strictly required, help us implement computations more efficiently and compactly by replacing nested `for` loops.
- In self-attention mechanisms used in LLMs, also called scaled-dot product attention, we include trainable weight matrices to compute intermediate transformations of the inputs: queries, values, and keys.
- When working with LLMs that read and generate text from left to right, we add a causal attention mask to prevent the LLM from accessing future tokens.
- In addition to causal attention masks to zero-out attention weights, we can add a dropout mask to reduce overfitting in LLMs.
- The attention modules in transformer-based LLMs involve multiple instances of causal attention, which is called multi-head attention.
- We can create a multi-head attention module by stacking multiple instances of causal attention modules.
- A more efficient way of creating multi-head attention modules involves batched matrix multiplications.