Next, we will make enhancements to the self-attention mechanism, focusing specifically on incorporating causal and multi-head elements. The causal aspect involves modifying the attention mechanism to prevent the model from accessing future information in the sequence, which is crucial for tasks like language modeling, where each word prediction should only depend on previous words.

The multi-head component involves splitting the attention mechanism into multiple "heads." Each head learns different aspects of the data, allowing the model to simultaneously attend to information from different representation subspaces at different positions. This improves the model's performance in complex tasks.

3.5 Hiding future words with causal attention

For many LLM tasks, you will want the self-attention mechanism to consider only the tokens that appear prior to the current position when predicting the next token in a sequence. Causal attention, also known as *masked attention*, is a specialized form of self-attention. It restricts a model to only consider previous and current inputs in a sequence when processing any given token when computing attention scores. This is in contrast to the standard self-attention mechanism, which allows access to the entire input sequence at once.

Now, we will modify the standard self-attention mechanism to create a *causal attention* mechanism, which is essential for developing an LLM in the subsequent chapters. To achieve this in GPT-like LLMs, for each token processed, we mask out the future tokens, which come after the current token in the input text, as illustrated in figure 3.19. We mask out the attention weights above the diagonal, and we

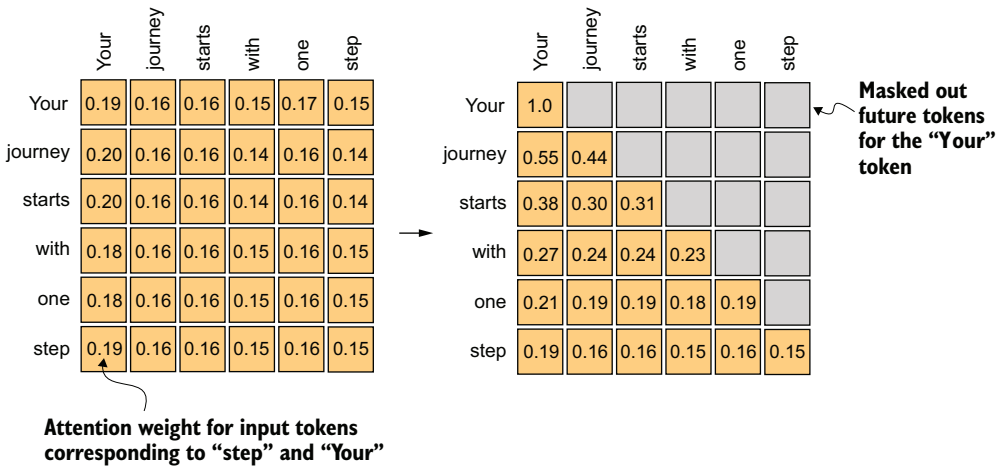

Figure 3.19 In causal attention, we mask out the attention weights above the diagonal such that for a given input, the LLM can't access future tokens when computing the context vectors using the attention weights. For example, for the word "journey" in the second row, we only keep the attention weights for the words before ("Your") and in the current position ("journey").

normalize the nonmasked attention weights such that the attention weights sum to 1 in each row. Later, we will implement this masking and normalization procedure in code.

3.5.1 Applying a causal attention mask

Our next step is to implement the causal attention mask in code. To implement the steps to apply a causal attention mask to obtain the masked attention weights, as summarized in figure 3.20, let's work with the attention scores and weights from the previous section to code the causal attention mechanism.

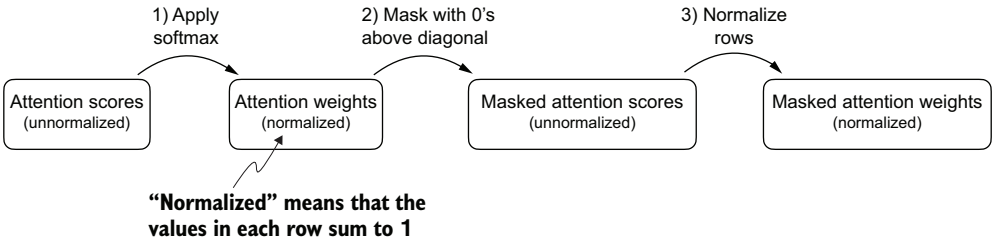

Figure 3.20 One way to obtain the masked attention weight matrix in causal attention is to apply the softmax function to the attention scores, zeroing out the elements above the diagonal and normalizing the resulting matrix.

In the first step, we compute the attention weights using the softmax function as we have done previously:

```
queries = sa_v2.W_query(inputs)
keys = sa_v2.W_key(inputs)
attn_scores = queries @ keys.T
attn_weights = torch.softmax(attn_scores / keys.shape[-1]**0.5, dim=-1)
print(attn_weights)
```

Reuses the query and key weight matrices of the SelfAttention_v2 object from the previous section for convenience

This results in the following attention weights:

```
tensor([[0.1921, 0.1646, 0.1652, 0.1550, 0.1721, 0.1510],
        [0.2041, 0.1659, 0.1662, 0.1496, 0.1665, 0.1477],
        [0.2036, 0.1659, 0.1662, 0.1498, 0.1664, 0.1480],
        [0.1869, 0.1667, 0.1668, 0.1571, 0.1661, 0.1564],
        [0.1830, 0.1669, 0.1670, 0.1588, 0.1658, 0.1585],
        [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
       grad_fn=<SoftmaxBackward0>)
```

We can implement the second step using PyTorch's `tril` function to create a mask where the values above the diagonal are zero:

```
context_length = attn_scores.shape[0]
mask_simple = torch.tril(torch.ones(context_length, context_length))
print(mask_simple)
```

The resulting mask is

```
tensor([[1., 0., 0., 0., 0., 0.],
        [1., 1., 0., 0., 0., 0.],
        [1., 1., 1., 0., 0., 0.],
        [1., 1., 1., 1., 0., 0.],
        [1., 1., 1., 1., 1., 0.],
        [1., 1., 1., 1., 1., 1.]])
```

Now, we can multiply this mask with the attention weights to zero-out the values above the diagonal:

```
masked_simple = attn_weights*mask_simple
print(masked_simple)
```

As we can see, the elements above the diagonal are successfully zeroed out:

```
tensor([[0.1921, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.2041, 0.1659, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.2036, 0.1659, 0.1662, 0.0000, 0.0000, 0.0000],
        [0.1869, 0.1667, 0.1668, 0.1571, 0.0000, 0.0000],
        [0.1830, 0.1669, 0.1670, 0.1588, 0.1658, 0.0000],
        [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
       grad_fn=<MulBackward0>)
```

The third step is to renormalize the attention weights to sum up to 1 again in each row. We can achieve this by dividing each element in each row by the sum in each row:

```
row_sums = masked_simple.sum(dim=-1, keepdim=True)
masked_simple_norm = masked_simple / row_sums
print(masked_simple_norm)
```

The result is an attention weight matrix where the attention weights above the diagonal are zeroed-out, and the rows sum to 1:

```
tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.5517, 0.4483, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.3800, 0.3097, 0.3103, 0.0000, 0.0000, 0.0000],
        [0.2758, 0.2460, 0.2462, 0.2319, 0.0000, 0.0000],
        [0.2175, 0.1983, 0.1984, 0.1888, 0.1971, 0.0000],
        [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
       grad_fn=<DivBackward0>)
```

Information leakage

When we apply a mask and then renormalize the attention weights, it might initially appear that information from future tokens (which we intend to mask) could still influence the current token because their values are part of the softmax calculation. However, the key insight is that when we renormalize the attention weights after masking,