Coding Attention Mecanism Part 1

Hamza Fatnaoui

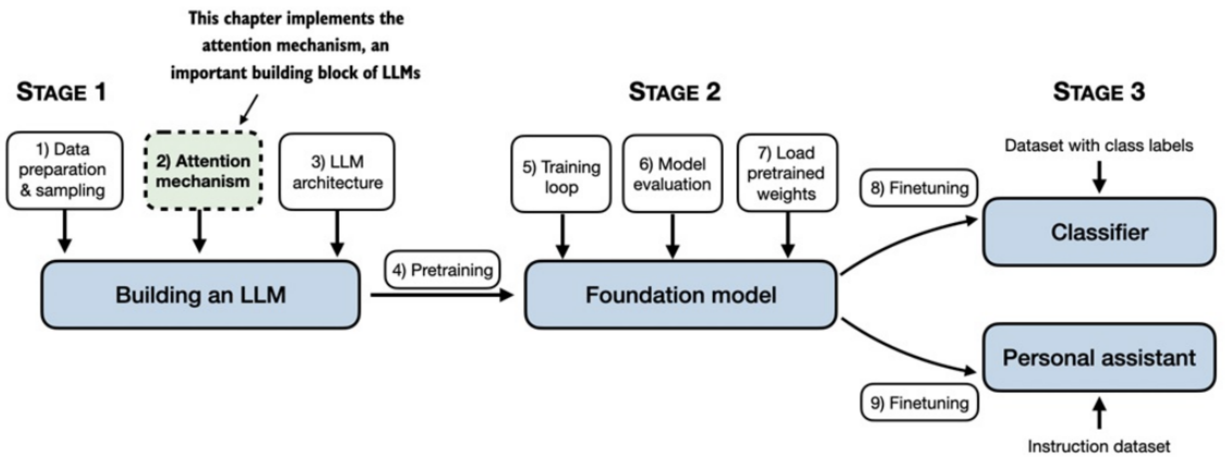
Hicham Ibn Issaghyr

Ikram Arif

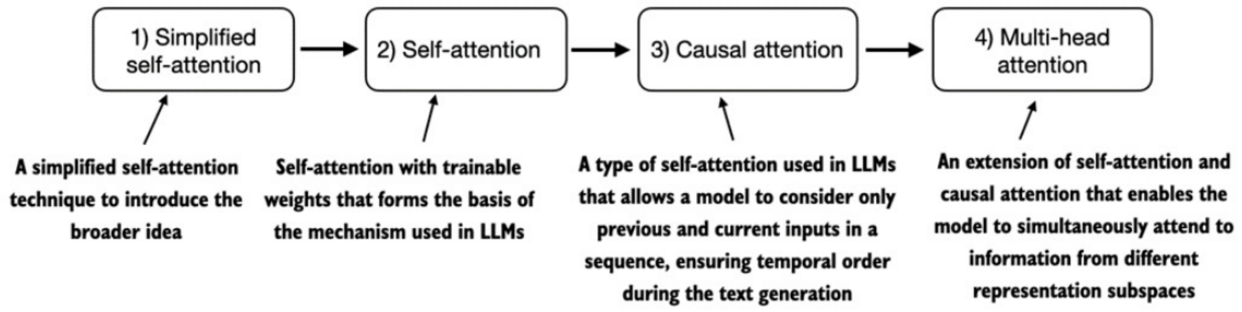
In the previous chapter, you learned how to prepare the input text for training LLMs. This involved splitting text into individual word and subword tokens, which can be encoded into vector representations, the so-called embeddings, for the LLM.

In this chapter, we will now look at an integral part of the LLM architecture itself.

A mental model of the three main stages of coding an LLM, pretraining the LLM on a general text dataset, and finetuning it on a labeled dataset. This chapter focuses on attention mechanisms, which are an integral part of an LLM architecture.



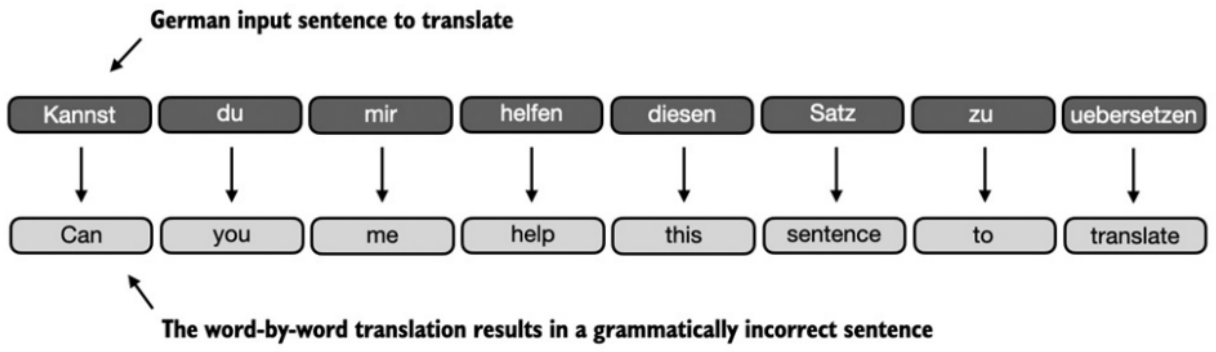
The figure depicts different attention mechanisms we will code in this chapter, starting with a simplified version of self-attention before adding the trainable weights. The causal attention mechanism adds a mask to self-attention that allows the LLM to generate one word at a time. Finally, multi-head attention organizes the attention mechanism into multiple heads, allowing the model to capture various aspects of the input data in parallel.

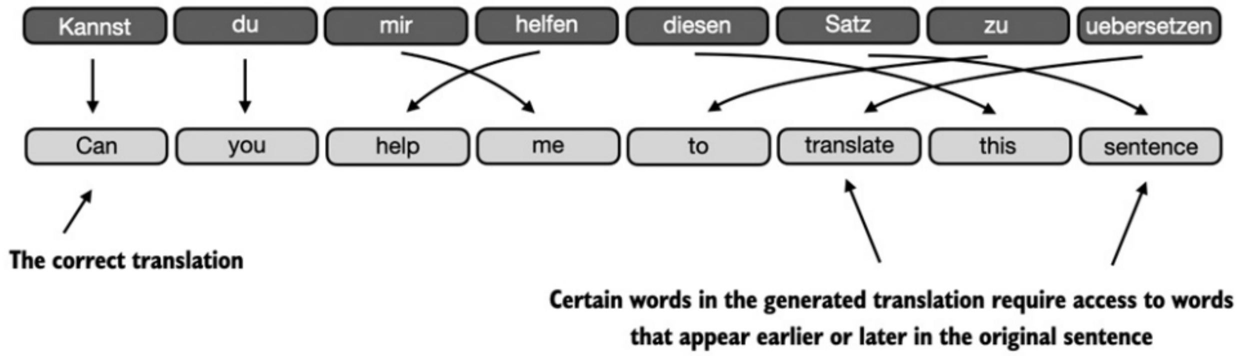

1) Simplified self-attention
2) Self-attention
3) Causal attention
4) Multi-head attention
A simplified self-attention technique to introduce the broader idea
Self-attention with trainable weights that forms the basis of the mechanism used in LLMs
A type of self-attention used in LLMs that allows a model to consider only previous and current inputs in a sequence, ensuring temporal order during the text generation
An extension of self-attention and causal attention that enables the model to simultaneously attend to information from different representation subspaces

1. The problem with modeling long sequences

Before we dive into the self-attention mechanism that is at the heart of LLMs later in this chapter, what is the problem with architectures without attention mechanisms that predate LLMs? Suppose we want to develop a language translation model that translates text from one language into another. We can't simply translate a text word by word due to the grammatical structures in the source and target language.

When translating text from one language to another, such as German to English, it's not possible to merely translate word by word. Instead, the translation process requirescontextual understanding and grammar alignment.


German input sentence to translate
Kannst
du
mir
helfen
diesen
Satz
zu
uebersetzen
Can
you
me
help
this
sentence
to
translate
The word-by-word translation results in a grammatically incorrect sentence


Kannst
du
mir
helfen
diesen
Satz
zu
uebersetzen
Can
you
help
me
to
translate
this
sentence
The correct translation
Certain words in the generated translation require access to words that appear earlier or later in the original sentence

2. Attending to different parts of the input with self-attention

We'll now delve into the inner workings of the self-attention mechanism andlearn how to code it from the ground up. Self-attention serves as the cornerstone of every LLM based on the transformer architecture. It's worth noting that this topic may require a lot of focus and attention (no pun intended), but once you grasp its fundamentals

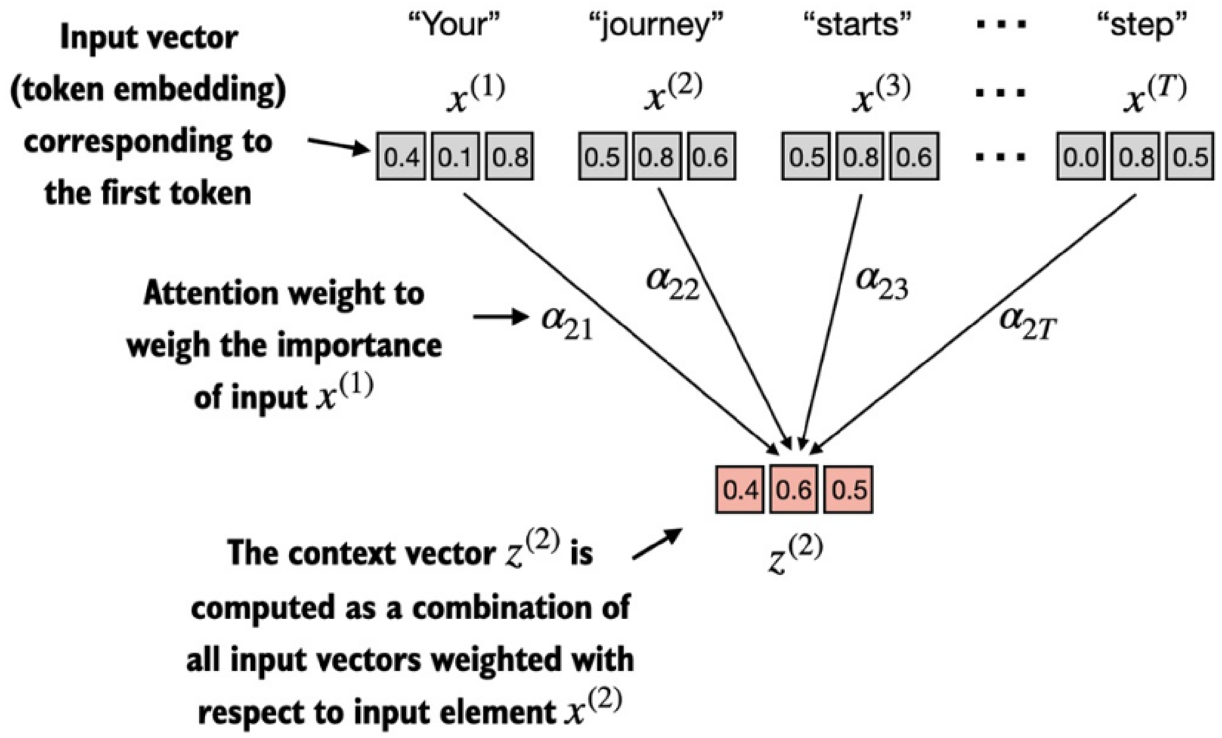
The "self" in self-attention

In self-attention, the "self" refers to the mechanism's ability to compute attention weights by relating different positions within a single input sequence. It assesses and learns the relationships and dependencies between various parts of the input itself, such as words in a sentence or pixels in an image. This is in contrast to traditional attention mechanisms, where the focus is on the relationships between elements of two different sequences, such as in sequence-to-sequence models where the attention might be between an input sequence and an output sequence

Since self-attention can appear complex, especially if you are encountering it for the first time, we will begin by introducing a simplified version of self-attention in the next subsection.

2.1. A simple self-attention mechanism without trainable weights

The goal of self-attention is to compute a context vector, for each input element, that combines information from all other input elements. In the example depicted in this figure, we compute the context vector z(2). The importance or contribution of each input element for computing z(2) is determined by the attention weights α21 to α2T. When computing z(2), the attention weights are calculated with respect to input element x(2) and all other inputs. The exact computation of these attention weights is discussed later in this section.


Input vector (token embedding) corresponding to the first token
"Your"
"journey"
"starts"
"step"
$x^{(1)}$
$x^{(2)}$
$x^{(3)}$
$x^{(T)}$
0.4 0.1 0.8
0.5 0.8 0.6
0.5 0.8 0.6
0.0 0.8 0.5
Attention weight to weigh the importance of input $x^{(1)}$
α_{21}
α_{22}
α_{23}
α_{2T}
0.4 0.6 0.5
$z^{(2)}$
The context vector $z^{(2)}$ is computed as a combination of all input vectors weighted with respect to input element $x^{(2)}$

In self-attention, our goal is to calculate context vectors z(i) for each element x(i) in the input sequence. A context vector can be interpreted as an enriched embedding vector.

To illustrate this concept, let's focus on the embedding vector of the second input element, x(2) (which corresponds to the token "journey"), and the corresponding context vector, z(2). This enhanced context vector, z(2), is an embedding that contains information about x(2) and all other input elements x(1) to x(T).

In self-attention, context vectors play a crucial role. Their purpose is to create enriched representations of each element in an input sequence (like a sentence) by incorporating information from all other elements in the sequence, as illustrated in Figure 3.7. This is essential in LLMs, which needto understand the relationship and relevance of words in a sentence to each
other. Later, we will add trainable weights that help an LLM learn to construct these context vectors so that they are relevant for the LLM to generate the next token.

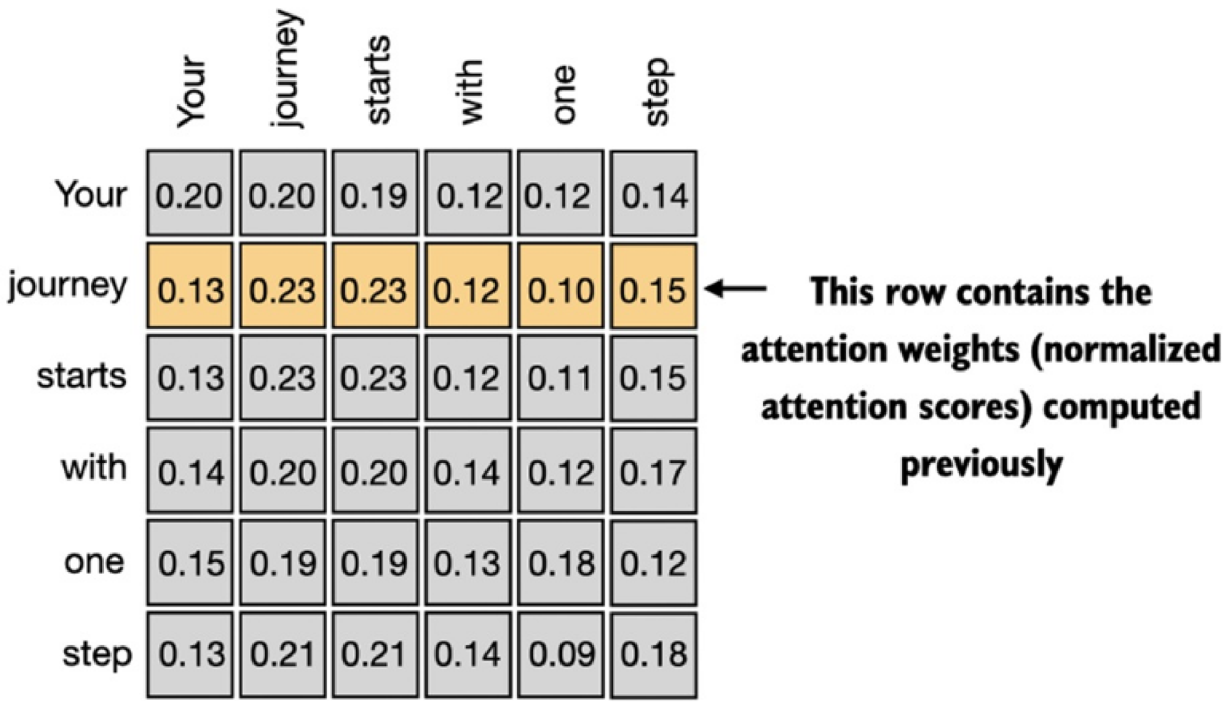
In this section, we implement a simplified self-attention mechanism to compute these weights and the resulting context vector one step at a time.

Here is the GitHub link for the implemented and documented code:

Self-Attention Code for X2

2.2. Computing attention weights for all input tokens

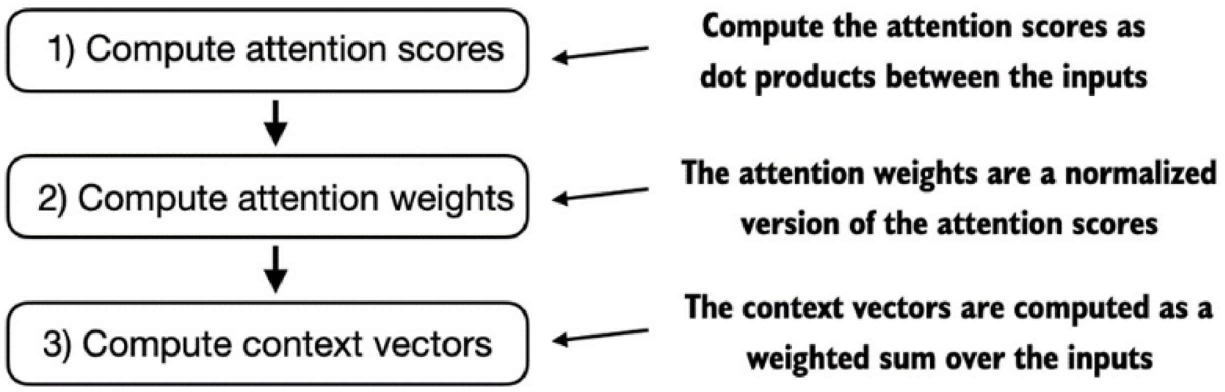
In the previous section, we computed attention weights and the context vector for input 2. Now, we are extending this computation to calculate attention weights and context vectors for all inputs. The highlighted row shows the attention weights for the second input element as a query, as we computed in the previous section. This section generalizes the computation to obtain all other attention weights.



	Your	journey	starts	with	one	step
Your	0.20	0.20	0.19	0.12	0.12	0.14
journey	0.13	0.23	0.23	0.12	0.10	0.15
starts	0.13	0.23	0.23	0.12	0.11	0.15
with	0.14	0.20	0.20	0.14	0.12	0.17
one	0.15	0.19	0.19	0.13	0.18	0.12
step	0.13	0.21	0.21	0.14	0.09	0.18

This row contains the attention weights (normalized attention scores) computed previously

We follow the same three steps as before, except that we make a few modifications in the code to compute all context vectors instead of only the second context vector, z(2).


1) Compute attention scores
Compute the attention scores as dot products between the inputs
2) Compute attention weights
The attention weights are a normalized version of the attention scores
3) Compute context vectors
The context vectors are computed as a weighted sum over the inputs

Here is the GitHub link for the implemented and documented code:

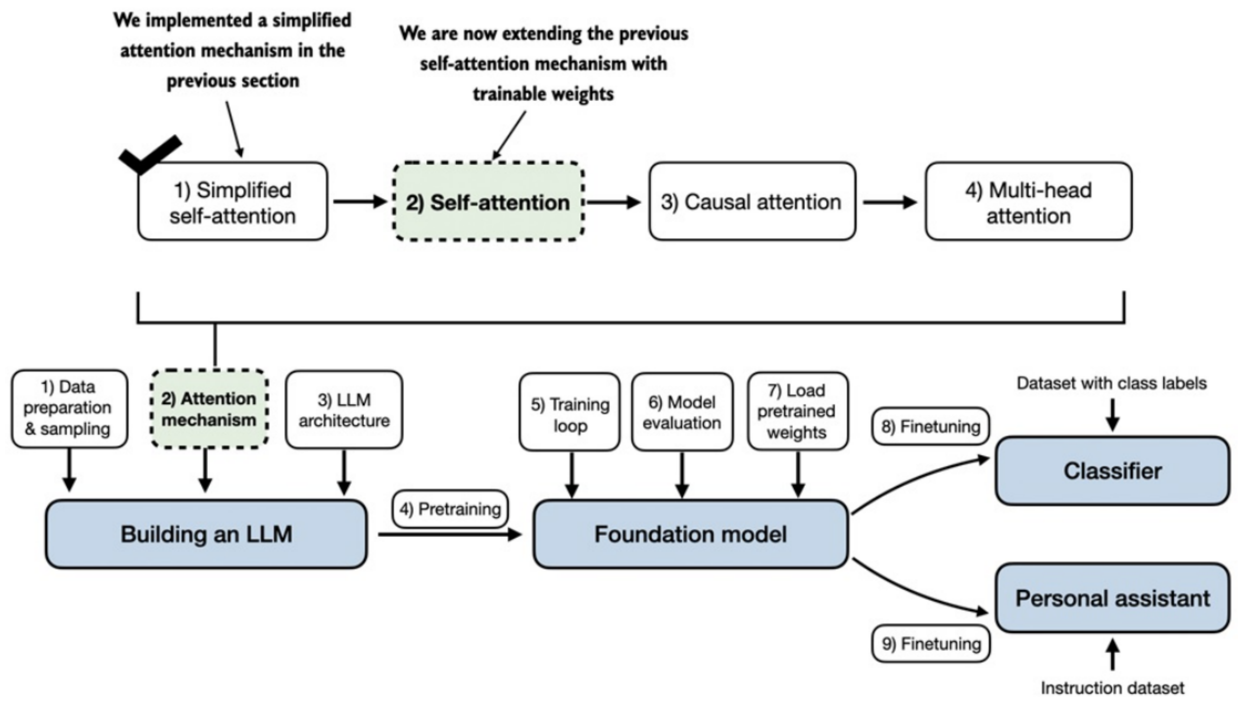
Self-Attention for All the inputs

In the next section, we will add trainable weights, enabling the LLM to learn from data and improve its performance on specific tasks.

3. Implementing self-attention with trainable weights

In this section, we are implementing the self-attention mechanism that is used in the original transformer architecture, the GPT models, and most otherpopular LLMs. This self-attention mechanism is also called scaled dot-product attention.

In the previous section, we coded a simplified attention mechanism to understand the basic mechanism behind attention mechanisms. In this section, we add trainable weights to this attention mechanism. In the upcoming sections, we will then extend this self-attention mechanism by adding a causal mask and multiple heads.


We implemented a simplified attention mechanism in the previous section
We are now extending the previous self-attention mechanism with trainable weights
1) Simplified self-attention
2) Self-attention
3) Causal attention
4) Multi-head attention
1) Data preparation & sampling
2) Attention mechanism
3) LLM architecture
Building an LLM
4) Pretraining
5) Training loop
6) Model evaluation
7) Load pretrained weights
Foundation model
8) Finetuning
9) Finetuning
Dataset with class labels
Classifier
Personal assistant
Instruction dataset

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.

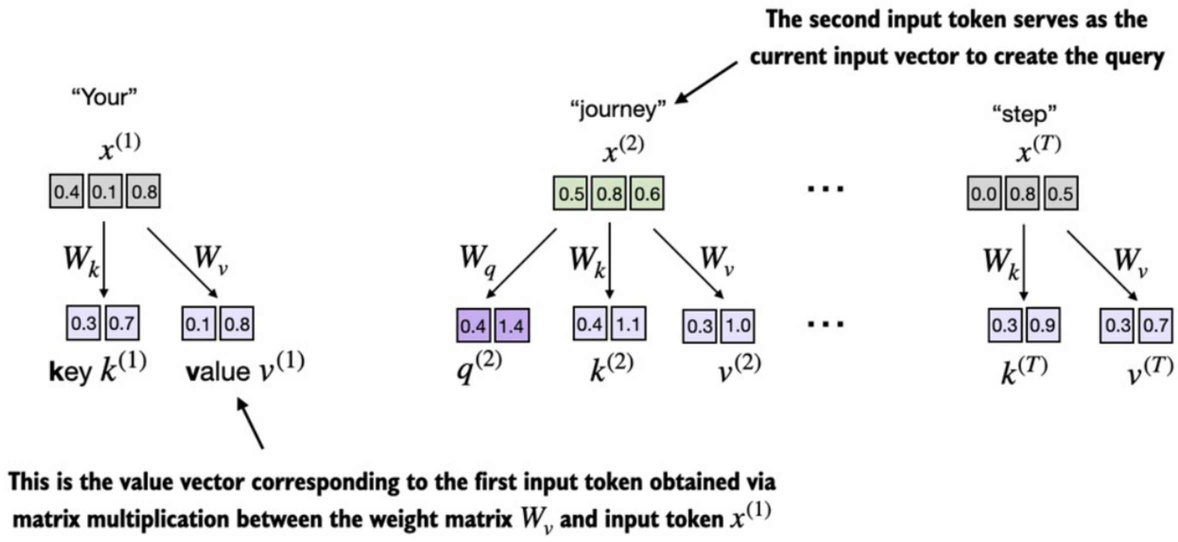
The most notable difference is the introduction of weight matrices that are updated during model training. These trainable weight matrices are crucial so that the model (specifically, the attention module inside the model) can learn to produce "good" context vectors.

We will 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 an LLM architecture, whichwe will code in the next chapter.

3.1. Computing the attention weights step by step

We will implement the self-attention mechanism step by step by introducing the three trainable weight matrices Wq, Wk, and Wv. These three matrices are used to project the embedded input tokens, x(i), into query, key, and value vectors

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 Wq. Similarly, we obtain the key and value vectors via matrix multiplication involving the weight matrices Wk and Wv.



we will start by computing only one context vector, z(2), for illustration purposes. In the next section, we will modify this code to calculate all context vectors.

The implemented and documented code: Computing the attention weights step by step

Why query, key, and value?

The terms "key," "query," and "value" in the context of attention mechanisms are borrowed from the domain of information retrieval and databases, where similar concepts are used to store, search, and retrieve information.

A "query" is analogous to a search query in a database. It represents the current item (e.g., a word or token in a sentence) the model focuses on or tries to understand. The query is used to probe the other parts of the input sequence to determine how much attention to pay to them.

The "key" is like a database key used for indexing and searching. In the attention mechanism, each item in the input sequence (e.g., each word in a sentence) has an associated key. These keys are used to match with the query.

The "value" in this context is similar to the value in a key-value pair in a database. It represents the actual content or representation of the input items. Once the model determines which keys (and thus which parts of the input) are most relevant to the query (the current focus item), it retrieves the corresponding values.

3.2. Implementing a compact self-attention Python class

In the previous sections, we have gone through a lot of steps to compute the self-attention outputs. This was mainly done for illustration purposes so we could go through one step at a time. In practice, with the LLM implementation in the next chapter in mind, it is helpful to organize this code
into a Python class as follows:

```python
import torch.nn as nn
class SelfAttention_v1(nn.Module):
  def __init__(self, d_in, d_out):
    super().__init__()
    self.d_out = d_out
    self.W_query = nn.Parameter(torch.rand(d_in, d_out))
    self.W_key = nn.Parameter(torch.rand(d_in, d_out))
    self.W_value = nn.Parameter(torch.rand(d_in, d_out))
  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, which 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.

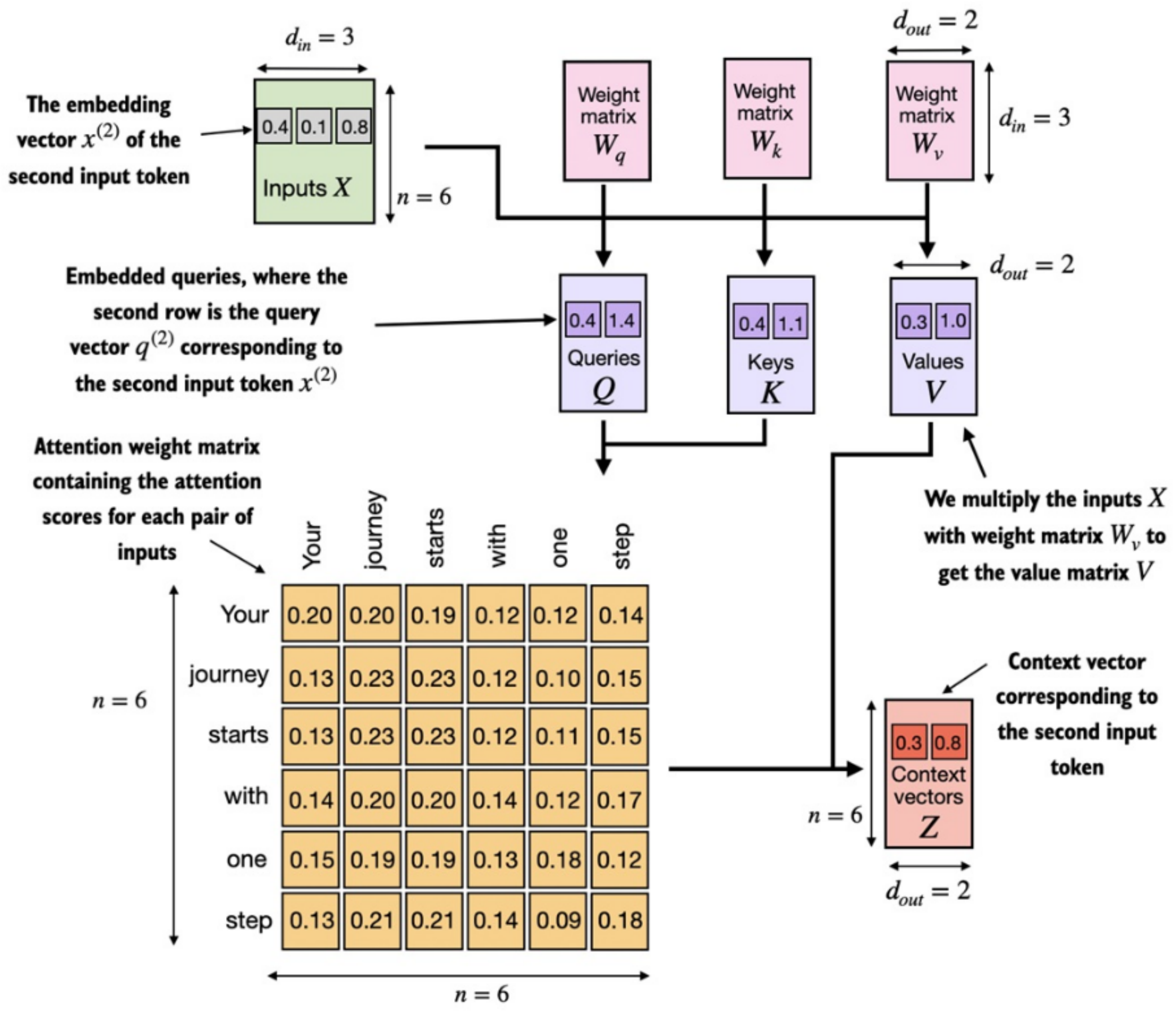
We can use this class as follows:

```python
torch.manual_seed(123)
sa_v1 = SelfAttention_v1(d_in, d_out)
print(sa_v1(inputs))
```

```
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>)
```

In self-attention, we transform the input vectors in the input matrix X with the three weight matrices, Wq, Wk, and Wv. 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 in this figure, not a batch of multiple inputs.

Consequently, the 3D input tensor is simplified to a 2D matrix in this context. This approach allows for a more straightforward visualization and understanding of the processes involved.)



$d_{in} = 3$
The embedding vector $x^{(2)}$ of the second input token
0.4 0.1 0.8
Inputs X
$n = 6$
Weight matrix W_q
Weight matrix W_k
Weight matrix W_v
$d_{out} = 2$
$d_{in} = 3$
Embedded queries, where the second row is the query vector $q^{(2)}$ corresponding to the second input token $x^{(2)}$
0.4 1.4
Queries Q
0.4 1.1
Keys K
0.3 1.0
Values V
$d_{out} = 2$
Attention weight matrix containing the attention scores for each pair of inputs
We multiply the inputs X with weight matrix W_v to get the value matrix V
Context vector corresponding to the second input token
0.3 0.8
Context vectors Z
$n = 6$
$d_{out} = 2$
$n = 6$
$n = 6$

	Your	journey	starts	with	one	step
Your	0.20	0.20	0.19	0.12	0.12	0.14
journey	0.13	0.23	0.23	0.12	0.10	0.15
starts	0.13	0.23	0.23	0.12	0.11	0.15
with	0.14	0.20	0.20	0.14	0.12	0.17
one	0.15	0.19	0.19	0.13	0.18	0.12
step	0.13	0.21	0.21	0.14	0.09	0.18

Self-attention involves the trainable weight matrices Wq, Wk, and Wv. These matrices transform input data into queries, keys, and values, which are crucial components of the attention mechanism. As the model is exposed to more data during training, it adjusts these trainable
weights.

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 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.

A self-attention class using PyTorch's Linear layers

```
import torch.nn as nn
class SelfAttention_v2(nn.Module):
  def __init__(self, d_in, d_out, qkv_bias=False):
    super().__init__()
    self.d_out = d_out
    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))
```

```
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.

In the next section, 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.

4. Hiding future words with causal attention

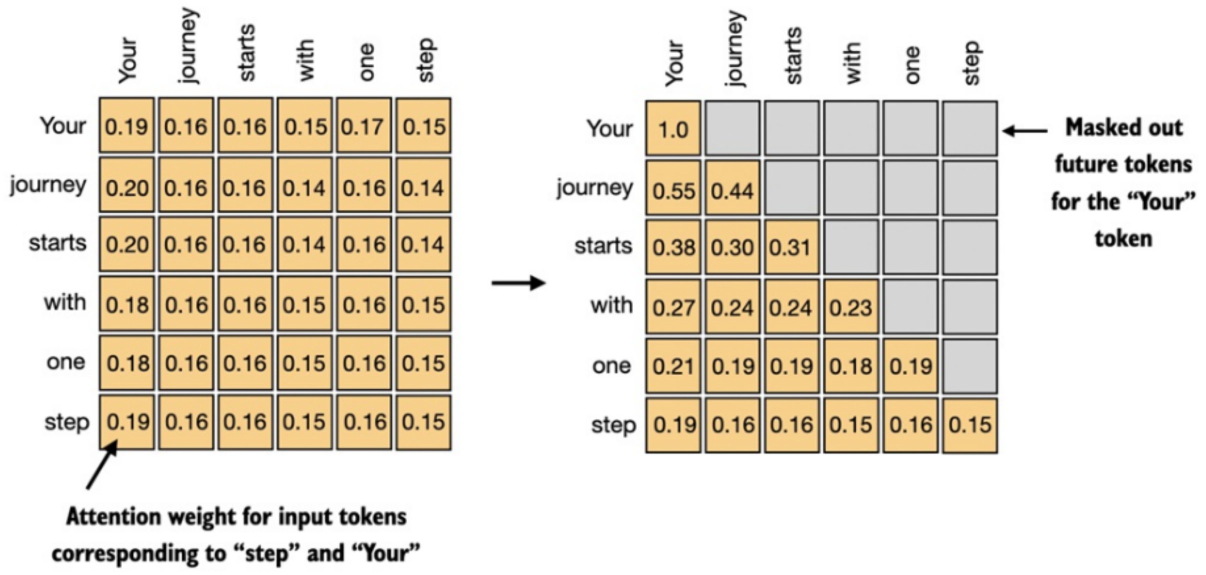
In this section, we modify the standard self-attention mechanism to create a causal attention mechanism, which is essential for developing an LLM in the subsequent chapters.

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. This is in contrast to the standard self-attention mechanism, which allows access to the entire input sequence at once.

Consequently, when computing attention scores, the causal attention mechanism ensures that the model only factors in tokens that occur at or before the current token in the sequence.

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

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").

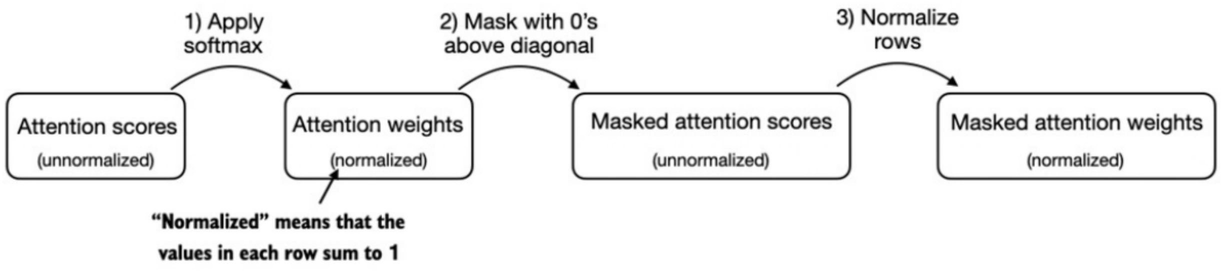


As illustrated, we mask out the attention weights above the diagonal, and we normalize the non-masked attention weights, such that the attention weights sum to 1 in each row. In the next section, we will implement this masking and normalization procedure in code.

4.1. Applying a causal attention mask

In this section, we implement the causal attention mask in code.

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.


1) Apply softmax
2) Mask with 0's above diagonal
3) Normalize rows
Attention scores (unnormalized)
Attention weights (normalized)
Masked attention scores (unnormalized)
Masked attention weights (normalized)
"Normalized" means that the values in each row sum to 1

we compute the attention weights using the softmax function as we have done in previous sections:

```
queries = sa_v2.W_query(inputs) #A
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)
```

```
tensor([[0.1690, 0.1607, 0.1609, 0.1706, 0.1710, 0.1677],
        [0.1615, 0.1555, 0.1563, 0.1762, 0.1844, 0.1662],
        [0.1614, 0.1558, 0.1566, 0.1759, 0.1841, 0.1661],
        [0.1636, 0.1596, 0.1601, 0.1727, 0.1776, 0.1664],
        [0.1602, 0.1651, 0.1655, 0.1693, 0.1748, 0.1652],
        [0.1649, 0.1559, 0.1565, 0.1752, 0.1805, 0.1670]],
       grad_fn=<SoftmaxBackward0>)
```

We can use 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)
```

```
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)
```

```
tensor([[0.1690, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.1615, 0.1555, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.1614, 0.1558, 0.1566, 0.0000, 0.0000, 0.0000],
        [0.1636, 0.1596, 0.1601, 0.1727, 0.0000, 0.0000],
        [0.1602, 0.1651, 0.1655, 0.1693, 0.1748, 0.0000],
        [0.1649, 0.1559, 0.1565, 0.1752, 0.1805, 0.1670]],
       grad_fn=<MulBackward0>)
```

The next 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)
```

```
tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.5096, 0.4904, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.3406, 0.3289, 0.3305, 0.0000, 0.0000, 0.0000],
        [0.2494, 0.2433, 0.2441, 0.2633, 0.0000, 0.0000],
        [0.1919, 0.1978, 0.1982, 0.2028, 0.2093, 0.0000],
        [0.1649, 0.1559, 0.1565, 0.1752, 0.1805, 0.1670]],
       grad_fn=<DivBackward0>)
```

Information leakage

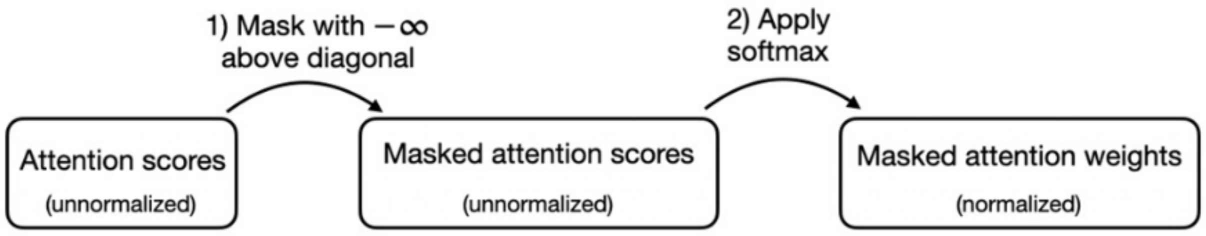
When we apply a mask and then renormalize the attention weights, it might initially appear that information from future tokens (which we intend to mask) could still influence the current token because their values are part of the softmax calculation. However, the key insight is that when we
renormalize the attention weights after masking, what we're essentially doing is recalculating the softmax over a smaller subset (since masked positions don't contribute to the softmax value).

The mathematical elegance of softmax is that despite initially including all positions in the denominator, after masking and renormalizing, the effect of the masked positions is nullified — they don't contribute to the softmax score in any meaningful way.

In simpler terms, after masking and renormalization, the distribution of attention weights is as if it was calculated only among the unmasked positions to begin with. This ensures there's no information leakage from future (or otherwise masked) tokens as we intended.

While we could be technically done with implementing causal attention at this point, we can take advantage of a mathematical property of the softmax function and implement the computation of the masked attention weights more efficiently in fewer steps.

A more efficient way to obtain the masked attention weight matrix in causal attention is to mask the attention scores with negative infinity values before applying the softmax function.


1) Mask with −∞ above diagonal
2) Apply softmax
Attention scores (unnormalized)
Masked attention scores (unnormalized)
Masked attention weights (normalized)

The softmax function converts its inputs into a probability distribution. When negative infinity values (-∞) are present in a row, the softmax function treats them as zero probability. (Mathematically, this is because e-∞ approaches 0.)

We can implement this more efficient masking "trick" by creating a mask with 1's above the diagonal and then replacing these 1's with negative infinity (-inf) values:

```
mask = torch.triu(torch.ones(context_length, context_length), diagonal=1)
masked = attn_scores.masked_fill(mask.bool(), -torch.inf)
print(masked)
```

```
tensor([[-0.1193,    -inf,    -inf,    -inf,    -inf,    -inf],
        [-0.3330, -0.3871,    -inf,    -inf,    -inf,    -inf],
        [-0.3263, -0.3761, -0.3691,    -inf,    -inf,    -inf],
        [-0.2094, -0.2444, -0.2400, -0.1329,    -inf,    -inf],
        [-0.1138, -0.0713, -0.0680, -0.0359,  0.0092,    -inf],
        [-0.2850, -0.3639, -0.3583, -0.1993, -0.1564, -0.2669]],
       grad_fn=<MaskedFillBackward0>)
```

Now, all we need to do is apply the softmax function to these masked results, and we are done:

```
attn_weights = torch.softmax(masked / keys.shape[-1]**0.5, dim=1)
print(attn_weights)
```

```
tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.5096, 0.4904, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.3406, 0.3289, 0.3305, 0.0000, 0.0000, 0.0000],
        [0.2494, 0.2433, 0.2441, 0.2633, 0.0000, 0.0000],
        [0.1919, 0.1978, 0.1982, 0.2028, 0.2093, 0.0000],
        [0.1649, 0.1559, 0.1565, 0.1752, 0.1805, 0.1670]],
       grad_fn=<SoftmaxBackward0>)
```

We could now use the modified attention weights to compute the context vectors via context_vec = attn_weights @ values. However, in the next section, we first cover another minor tweak to the
causal attention mechanism that is useful for reducing overfitting when training LLMs.

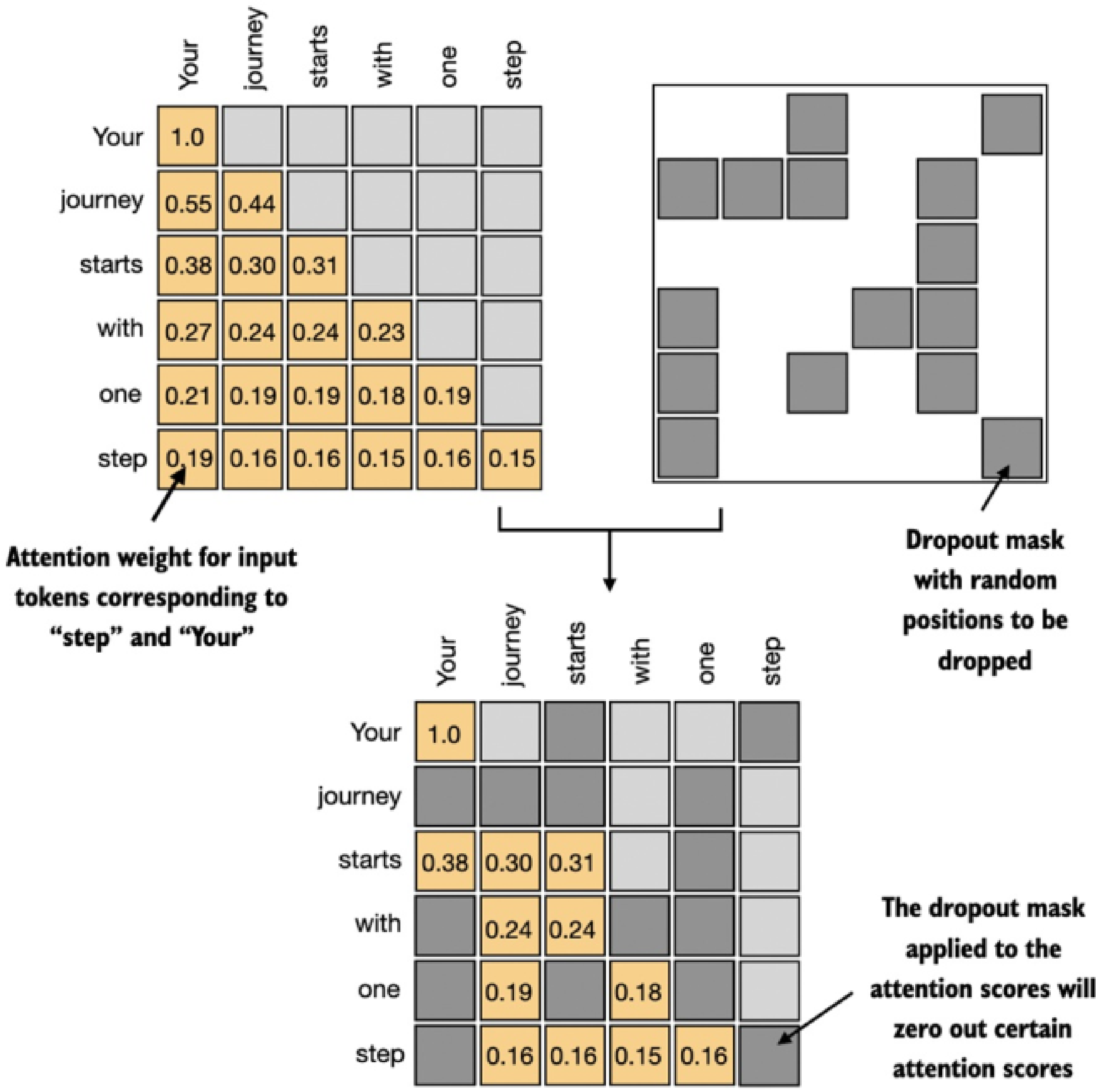
4.2. Masking additional attention weights with dropout

Dropout in deep learning is a technique where randomly selected hidden layer units are ignored during training, effectively "dropping" them out. This method helps prevent overfitting by ensuring that a model does not become overly reliant on any specific set of hidden layer units. It's important to emphasize that dropout is only used during training and is disabled afterward.

In the transformer architecture, including models like GPT, dropout in the attention mechanism is typically applied in two specific areas: after calculating the attention scores or after applying the attention weights to thevalue vectors.

Here, we will apply the dropout mask after computing the attention weights, because it's the more common variant in practice.

Using the causal attention mask (upper left), we apply an additional dropout mask (upper right) to zero out additional attention weights to reduce overfitting during training.


Your
journey
starts
with
one
step
1.0
0.55 0.44
0.38 0.30 0.31
0.27 0.24 0.24 0.23
0.21 0.19 0.19 0.18 0.19
0.19 0.16 0.16 0.15 0.16 0.15
Attention weight for input tokens corresponding to "step" and "Your"
Dropout mask with random positions to be dropped
1.0
0.38 0.30 0.31
0.24 0.24
0.19 0.18
0.16 0.16 0.15 0.16
The dropout mask applied to the attention scores will zero out certain attention scores

In the following code example, we use a dropout rate of 50%, which means masking out half of the attention weights.

```
torch.manual_seed(123)
dropout = torch.nn.Dropout(0.5)
example = torch.ones(6, 6)
print(dropout(example))
```

```
tensor([[2., 2., 2., 2., 2., 2.],
        [0., 2., 0., 0., 0., 0.],
        [0., 0., 2., 0., 2., 0.],
        [2., 2., 0., 0., 0., 2.],
        [2., 0., 0., 0., 0., 2.],
        [0., 2., 0., 0., 0., 0.]])
```

When applying dropout to an attention weight matrix with a rate of 50%, half of the elements in the matrix are randomly set to zero. To compensate for the reduction in active elements, the values of the remaining elements in the matrix are scaled up by a factor of 1/0.5 =2 (1/1-p). This scaling is crucial to maintain the overall balance of the attention weights, ensuring that the average influence of the attention mechanism remains consistent during both the training and inference phases.

Now, let's apply dropout to the attention weight matrix itself:

```
torch.manual_seed(123)
print(dropout(attn_weights))
```

```
tensor([[2.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.9809, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.6610, 0.0000, 0.0000, 0.0000],
        [0.4988, 0.4866, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.3838, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.3118, 0.0000, 0.0000, 0.0000, 0.0000]],
       grad_fn=<MulBackward0>)
```

Having gained an understanding of causal attention and dropout masking, we will develop a concise Python class in the following section. This class is designed to facilitate the efficient application of these two techniques.

4.3. Implementing a compact causal attention class

In this section, we will now incorporate the causal attention and dropout modifications into the SelfAttention Python class. This class will then serve as a template for developing multi-head
attention in the part 2, which is the final attention class we implement in this chapter.

But before we begin, one more thing is to ensure that the code can handle batches consisting of more than one input so that the CausalAttention class supports the batch outputs produced by the data loader.

For simplicity, to simulate such batch inputs, we duplicate the input text example:

```
batch = torch.stack((inputs, inputs), dim=0)
print(batch.shape)
```

```
torch.Size([2, 6, 3])
```

The following CausalAttention class is similar to the SelfAttention class we implemented earlier, except that we now added the dropout and causal mask components as highlighted in the following code:

A compact causal attention class

```
class CausalAttention(nn.Module):
  def __init__(self, d_in, d_out, context_length, dropout, qkv_bias=False):
    super().__init__()
    self.d_out = d_out
    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.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
    #New batch dimension b
    keys = self.W_key(x)
    queries = self.W_query(x)
    values = self.W_value(x)
    attn_scores = queries @ keys.transpose(1, 2)
    attn_scores.masked_fill_(
      self.mask.bool()[:num_tokens, :num_tokens], -torch.inf)
    attn_weights = torch.softmax(attn_scores / keys.shape[-1]**0.5, dim=-1)
    attn_weights = self.dropout(attn_weights)
    context_vec = attn_weights @ values
    return context_vec
```

While all added code lines should be familiar from previous sections, we now added a self.register_buffer() call in the **init** method. The use of register_buffer in PyTorch is not strictly necessary for all use cases but offers several advantages here. For instance, when we use the

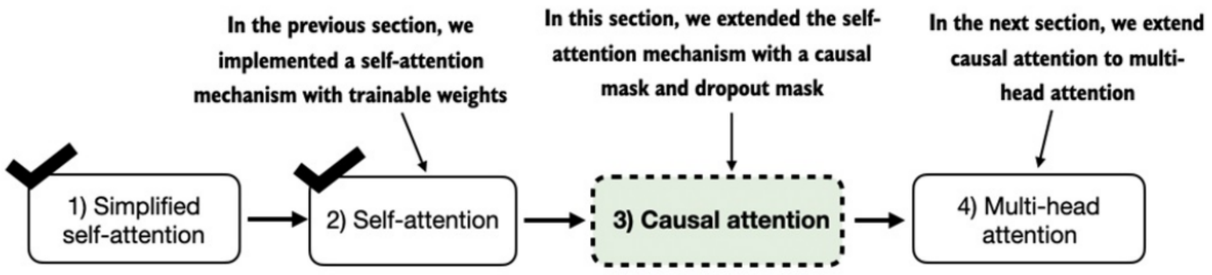
CausalAttention class in our LLM, buffers are automatically moved to the appropriate device (CPU or GPU) along with our model, which will be relevant when training the LLM in future chapters. This means we don't need to manually ensure these tensors are on the same device as your model parameters, avoiding device mismatch errors.

We can use the CausalAttention class as follows, similar to SelfAttention previously:

```
torch.manual_seed(123)
context_length = batch.shape[1]
ca = CausalAttention(d_in, d_out, context_length, 0.0)
context_vecs = ca(batch)
print("context_vecs.shape:", context_vecs.shape)
```

```
context_vecs.shape: torch.Size([2, 6, 2])
```

A mental model summarizing the four different attention modules we are coding inthis chapter. We began with a simplified attention mechanism, added trainable weights, and then added a casual attention mask. In the remainder of this chapter, we will extend the causal attention mechanism and code multi-head attention, which is the final module we will use in the LLM implementation in the next chapter.


In the previous section, we implemented a self-attention mechanism with trainable weights
In this section, we extended the self-attention mechanism with a causal mask and dropout mask
In the next section, we extend causal attention to multi-head attention
1) Simplified self-attention
2) Self-attention
3) Causal attention
4) Multi-head attention

In this section, we focused on the concept and implementation of causal attention in neural networks. In the second part, we will expand on this concept and implement a multi-head attention module that implements several of such causal attention mechanisms in parallel.