Coding Attention Mecanism Part 2

> Hamza Fatnaoui

> Hicham Ibn Issaghyr

> Ikram Arif

5. Extending single-head attention to multi-head attention

In this section of this chapter, we are extending the previously implemented causal attention class over multiple-heads. This is also called multi-head attention.

The term "multi-head" refers to dividing the attention mechanism into multiple "heads," each operating independently. In this context, a single causal attention module can be considered single-head attention, where there is only one set of attention weights processing the input sequentially.

In the following subsections, we will tackle this expansion from causal attention to multi-head attention. The first subsection will intuitively build a multi-head attention module by stacking multiple CausalAttention modules for illustration purposes. The second subsection will then implement the same multi-head attention module in a more complicated but computationally more efficient way.

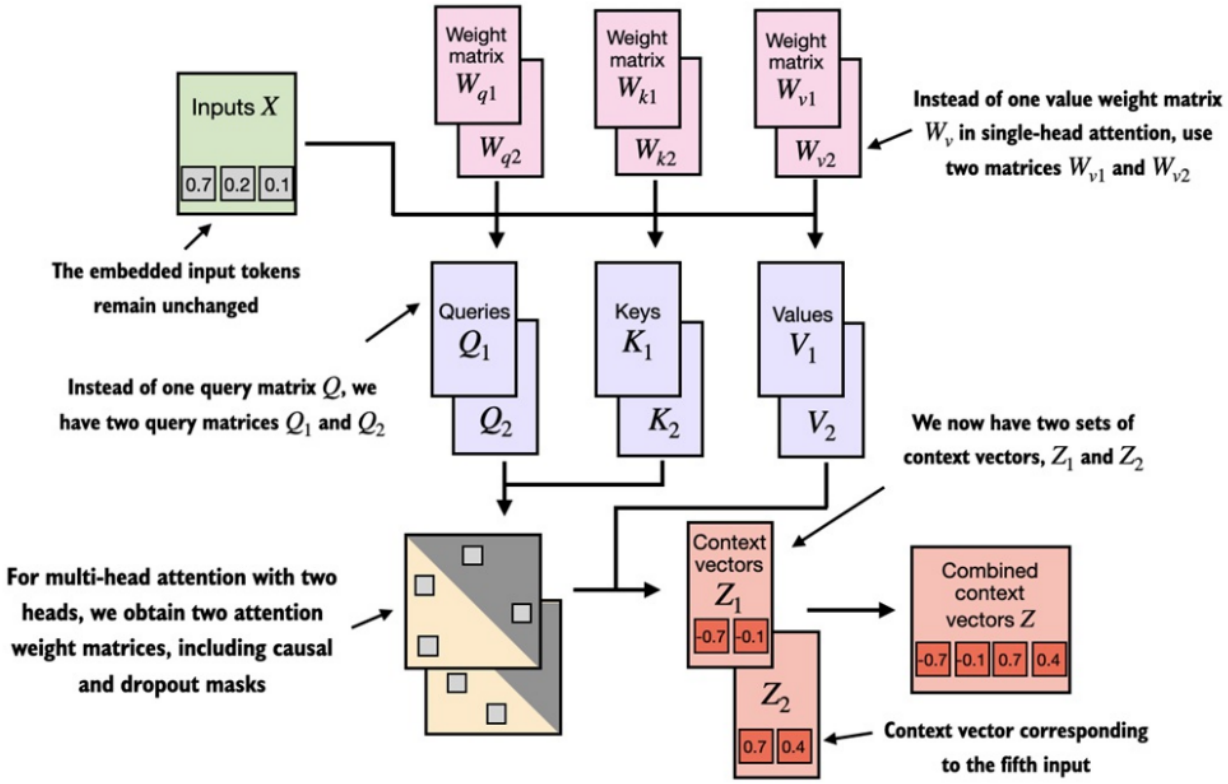
5.1. Stacking multiple single-head attention layers

In practical terms, implementing multi-head attention involves creating multiple instances of the self-attention mechanism, each with its own weights, and then combining their outputs. Using multiple instances of the self-attention mechanism

can be computationally intensive, but it's crucial for the kind of complex pattern recognition that models like transformer-based LLMs are known for.

The figure illustrates the structure of a multi-head attention module, which consists of multiple single-head attention modules

The multi-head attention module in this figure depicts two single-head attention modules stacked on top of each other. So, instead of using a single matrix Wv for computing the value matrices, in a multi-head attention module with two heads, we now have two value weight matrices: Wv1 and Wv2. The same applies to the other weight matrices, Wq and Wk. We obtain two sets of context vectors Z1 and Z2 that we can combine into a single context vector matrix Z.



As mentioned before, the main idea behind multi-head attention is to run the attention mechanism multiple times (in parallel) with different, learned linear projections -- the results of multiplying the input data (like the query, key, and value vectors in attention mechanisms) by a weight matrix.

In code, we can achieve this by implementing a simple MultiHeadAttentionWrapper class that stacks multiple instances of ourpreviously

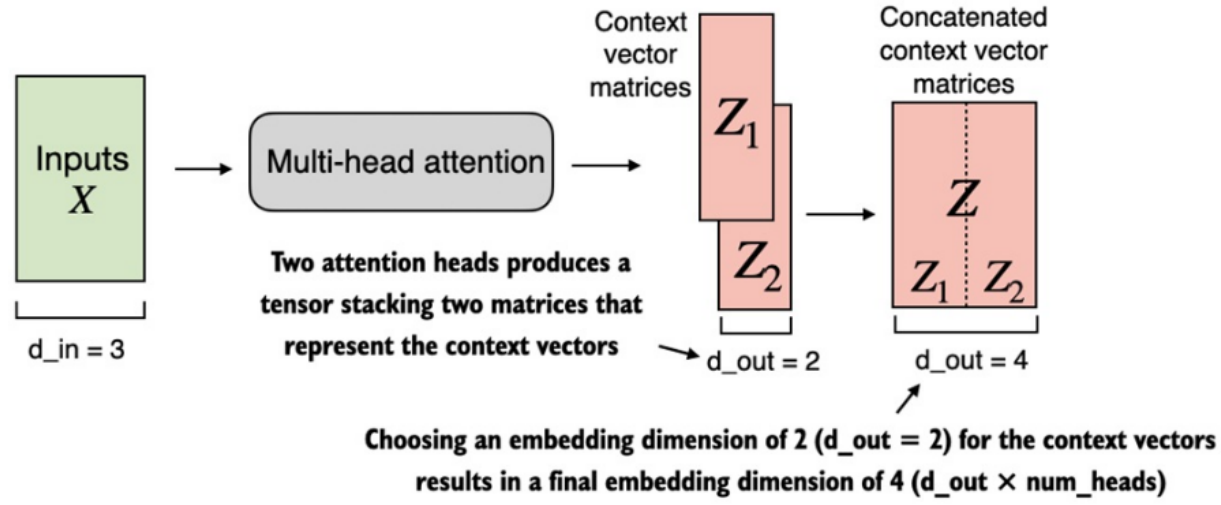
implemented CausalAttention module:

A wrapper class to implement multi-head attention

```
import torch.nn as nn
class MultiHeadAttentionWrapper(nn.Module):
  def __init__(self, d_in, d_out, context_length,
    dropout, num_heads, qkv_bias=False):
    super().__init__()
    self.heads = nn.ModuleList(
      [CausalAttention(d_in, d_out, context_length, dropout, qkv_bias)
    for _ in range(num_heads)]
    )
  def forward(self, x):
    return torch.cat([head(x) for head in self.heads], dim=-1)
```

For example, if we use this MultiHeadAttentionWrapper class with two attention heads (via num_heads=2) and CausalAttention output dimension d_out=2, this results in a 4-dimensional context vectors (d_out*num_heads=4)

Using the MultiHeadAttentionWrapper, we specified the number of attention heads (num_heads). If we set num_heads=2, as shown in this figure, we obtain a tensor with two sets of context vector matrices. In each context vector matrix, the rows represent the context vectors corresponding to the tokens, and the columns correspond to the embedding dimension specified via d_out=4. We concatenate these context vector matrices along the column dimension. Since we have 2 attention heads and an embedding dimension of 2, the final embedding dimension is 2 × 2 = 4.



We can use the MultiHeadAttentionWrapper class similar to the CausalAttention class before:

```
torch.manual_seed(123)
context_length = batch.shape[1] # This is the number of tokens
d_in, d_out = 3, 2
mha = MultiHeadAttentionWrapper(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)
```

```
tensor([[[-0.4519,  0.2216,  0.4772,  0.1063],
         [-0.5874,  0.0058,  0.5891,  0.3257],
         [-0.6300, -0.0632,  0.6202,  0.3860],
         [-0.5675, -0.0843,  0.5478,  0.3589],
         [-0.5526, -0.0981,  0.5321,  0.3428],
         [-0.5299, -0.1081,  0.5077,  0.3493]],

        [[-0.4519,  0.2216,  0.4772,  0.1063],
         [-0.5874,  0.0058,  0.5891,  0.3257],
         [-0.6300, -0.0632,  0.6202,  0.3860],
         [-0.5675, -0.0843,  0.5478,  0.3589],
         [-0.5526, -0.0981,  0.5321,  0.3428],
         [-0.5299, -0.1081,  0.5077,  0.3493]]], grad_fn=<CatBackward0>)
context_vecs.shape: torch.Size([2, 6, 4])
```

The first dimension of the resulting context_vecs tensor is 2 since we have two input texts (the input texts are duplicated, which is why the context vectors are exactly the same for those). The second dimension refers to the 6 tokens in each input. The third dimension refers to the 4-dimensional embedding of each token.

In this section, we implemented a MultiHeadAttentionWrapper that combined multiple single-head attention modules. However, note that theseare processed sequentially via [head(x) for head in self.heads] in the forward method. We can improve this implementation by processing the
heads in parallel. One way to achieve this is by computing the outputs for all attention heads simultaneously via matrix multiplication, as we will explore in the next section.

5.2. Implementing multi-head attention with weight splits

In the previous section, we 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 both of these concepts into a single MultiHeadAttention class. Also, in addition to just 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:

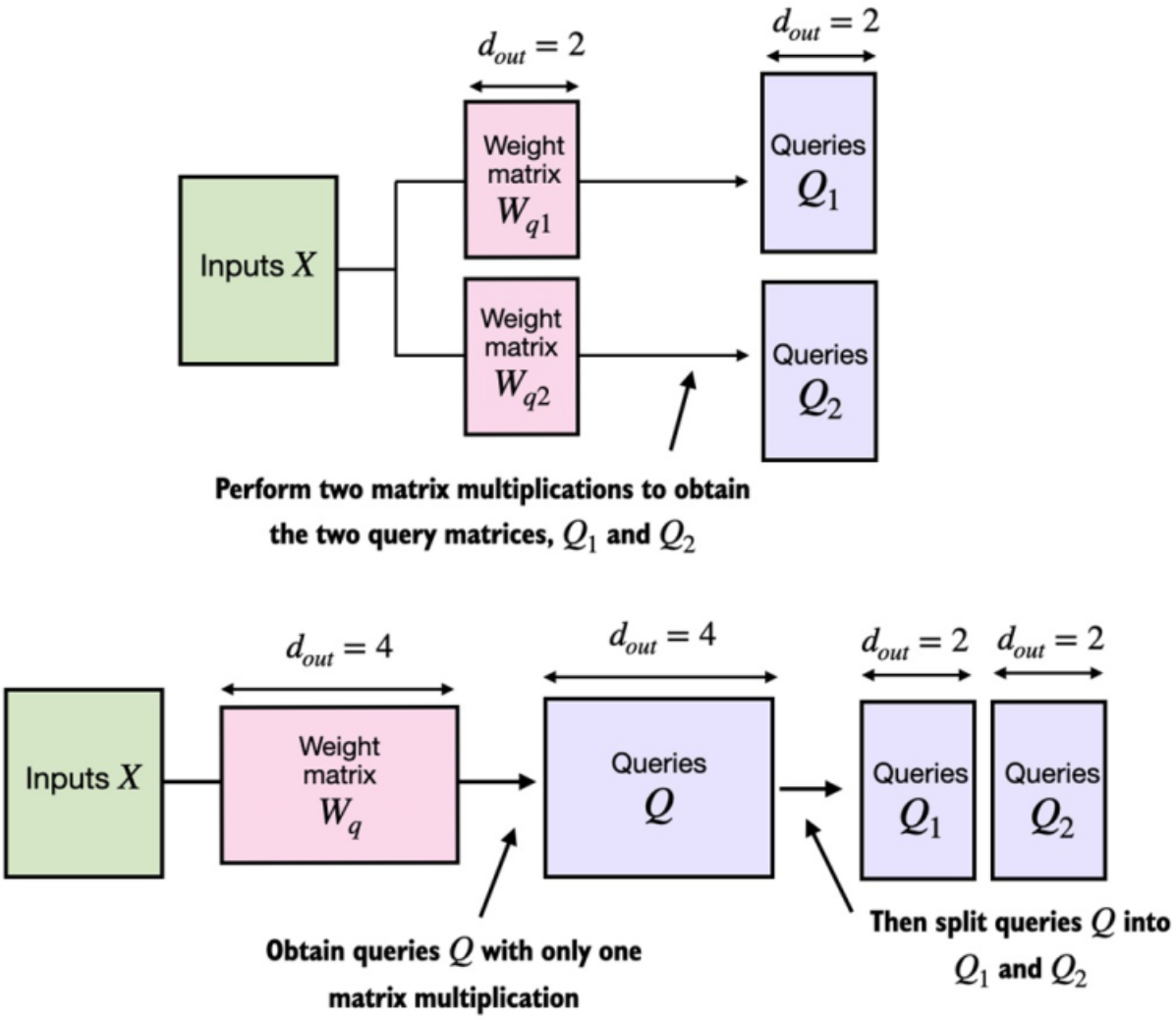
```python
class MultiHeadAttention(nn.Module):
  def __init__(self, d_in, d_out,context_length, dropout, num_heads, qkv_bias=False):
    super().__init__()
    assert d_out % num_heads == 0, "d_out must be divisible by num_head "
    self.d_out = d_out
    self.num_heads = num_heads
    self.head_dim = 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)
    keys = keys.view(b, num_tokens, self.num_heads, self.head_dim)
    values = values.view(b, num_tokens, self.num_heads, self.head_dim)
    queries = queries.view(b, num_tokens, self.num_heads, self.head_dim)
    keys = keys.transpose(1, 2)
    queries = queries.transpose(1, 2)
    values = values.transpose(1, 2)
    attn_scores = queries @ keys.transpose(2, 3)
    mask_bool = self.mask.bool()[:num_tokens, :num_tokens]
    attn_scores.masked_fill_(mask_bool, -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).transpose(1, 2)
    context_vec = context_vec.contiguous().view(b, num_tokens, self.d_out)
    context_vec = self.out_proj(context_vec)
    return context_vec
```

Even though the reshaping (.view) and transposing (.transpose) of tensors inside the MultiHeadAttention class looks very complicated, mathematically, 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.

In the MultiheadAttentionWrapper class with two attention heads, we initialized two weight matrices Wq1 and Wq2 and computed two query matrices Q1 and Q2

as illustrated at the top of this figure. In the MultiheadAttention class, we initialize one larger weight matrix Wq , only perform one matrix multiplication with the inputs to obtain a query matrix Q, and then split the query matrix into Q1 and Q2 as shown at the bottom of this figure. We do the same for the keys and values, which are not shown to reduce visual clutter.



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).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.

We added a so-called 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 the References section in Appendix B for more details), but it
is commonly used in many LLM architectures, which is why we 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)
```

```
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])
```

In this section, we implemented the MultiHeadAttention class that we will use in the upcoming sections when implementing and training the LLM itself. Note that while the code is fully functional, we 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 1600. Note that the embedding sizes of the token inputs and context embeddings are the same in GPT models (d_in = d_out).

6. Summary

- Attention mechanisms transform input elements into enhanced context vector representations that incorporate information about all inputs.
- 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 just a concise way of multiplying two vectors element-wise and then summing the products.

- Matrix multiplications, while not strictly required, help us to implement computations more efficiently and compactly by replacing nested for-loops.
- In self-attention mechanisms that are 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.
- Next to causal attention masks to zero out attention weights, we can also 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.