Lecture 7: Causal and multi-head attention

CSC 352 / CSC 552 · 2026-09-17

Open in Colab → **File → Save a copy in Drive**. You cannot edit the class original.

Start-of-class quiz

Type answers in the Canvas attendance quiz (empty boxes only). The questions stay here.

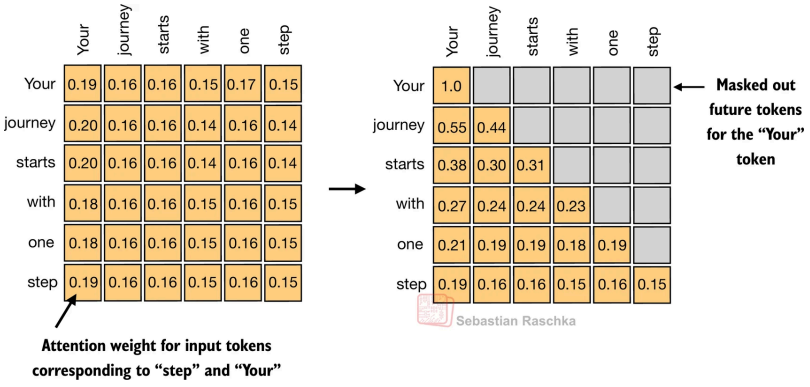
1. Self-attention mixes information from which positions?
2. Name one of the three trainable matrices in weighted self-attention.

Continued from last class

We can attend to every position. Today we hide the future, then run several heads.

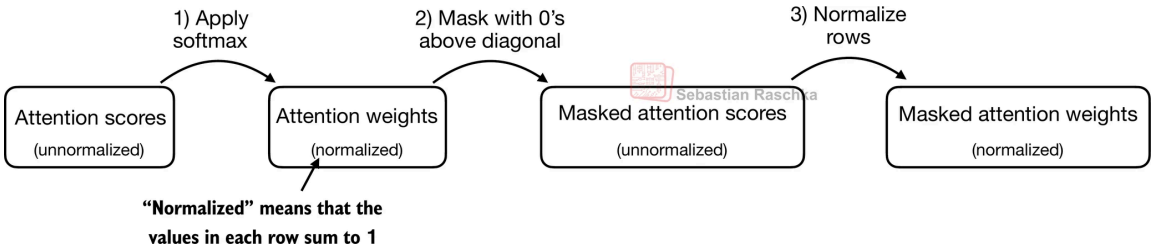
3.5 Hiding future words with causal attention

- In causal attention, the attention weights above the diagonal are masked, ensuring that for any given input, the LLM is unable to utilize future tokens while calculating the context vectors with the attention weight



3.5.1 Applying a causal attention mask

- In this section, we are converting the previous self-attention mechanism into a causal self-attention mechanism
- Causal self-attention ensures that the model's prediction for a certain position in a sequence is only dependent on the known outputs at previous positions, not on future positions
- In simpler words, this ensures that each next word prediction should only depend on the preceding words
- To achieve this, for each given token, we mask out the future tokens (the ones that come after the current token in the input text):



- To illustrate and implement causal self-attention, let's work with the attention scores and weights from the previous section:

In [108…
```
# Reuse the query and key weight matrices of the
# SelfAttention_v2 object from the previous section for convenience
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)
```

```
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>)
```

- The simplest way to mask out future attention weights is by creating a mask via PyTorch's tril function with elements below the main diagonal (including the diagonal itself) set to 1 and above the main diagonal set to 0:

In [110…
```
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.]])
```

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

In [112…
```
masked_simple = attn_weights*mask_simple
print(masked_simple)
```

```
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>)
```

- However, if the mask were applied after softmax, like above, it would disrupt the probability distribution created by softmax
- Softmax ensures that all output values sum to 1
- Masking after softmax would require re-normalizing the outputs to sum to 1 again, which complicates the process and might lead to unintended effects

- To make sure that the rows sum to 1, we can normalize the attention weights as follows:

In [115…
```
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],
        [0.5517, 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>)
```

- While we are technically done with coding the causal attention mechanism now, let's briefly look at a more efficient approach to achieve the same as above
- So, instead of zeroing out attention weights above the diagonal and renormalizing the results, we can mask the unnormalized attention scores above the diagonal with negative infinity before they enter the softmax function:

In [118…
```
mask = torch.triu(torch.ones(context_length, context_length), diagonal=1)
masked = attn_scores.masked_fill(mask.bool(), -torch.inf)
print(masked)
```

```
tensor([[0.2899,   -inf,   -inf,   -inf,   -inf,   -inf],
        [0.4656, 0.1723,   -inf,   -inf,   -inf,   -inf],
        [0.4594, 0.1703, 0.1731,   -inf,   -inf,   -inf],
        [0.2642, 0.1024, 0.1036, 0.0186,   -inf,   -inf],
        [0.2183, 0.0874, 0.0882, 0.0177, 0.0786,   -inf],
        [0.3408, 0.1270, 0.1290, 0.0198, 0.1290, 0.0078]],
       grad_fn=<MaskedFillBackward0>)
```

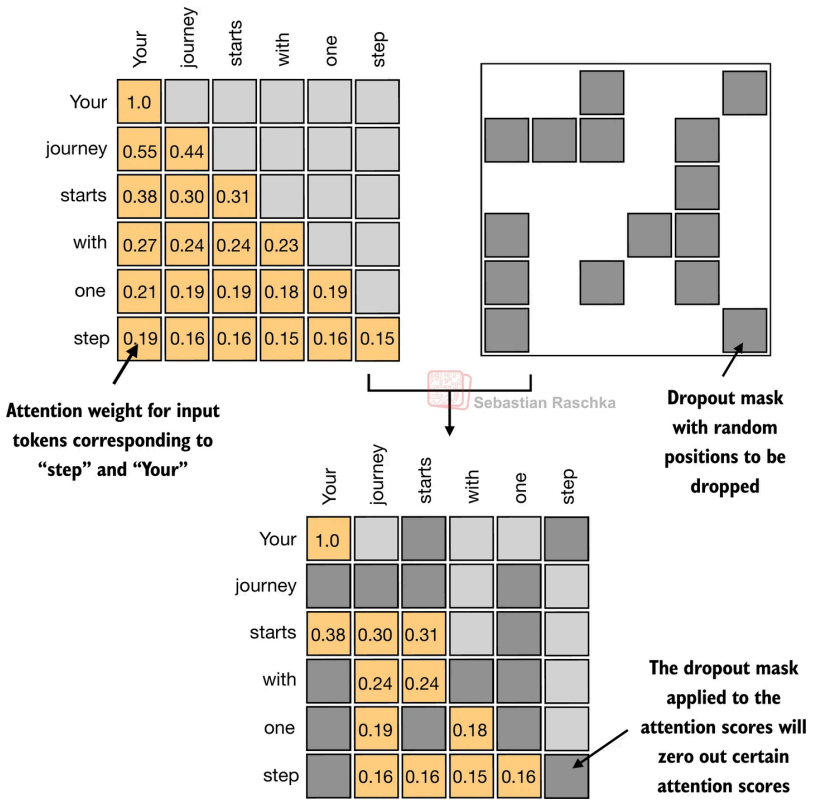
- As we can see below, now the attention weights in each row correctly sum to 1 again:

In [120…
```
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],
        [0.5517, 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=<SoftmaxBackward0>)
```

3.5.2 Masking additional attention weights with dropout

- In addition, we also apply dropout to reduce overfitting during training
- Dropout can be applied in several places:
  - for example, after computing the attention weights;
  - or after multiplying the attention weights with the value vectors
- Here, we will apply the dropout mask after computing the attention weights because it's more common
- Furthermore, in this specific example, we use a dropout rate of 50%, which means randomly masking out half of the attention weights. (When we train the GPT model later, we will use a lower dropout rate, such as 0.1 or 0.2



- If we apply a dropout rate of 0.5 (50%), the non-dropped values will be scaled accordingly by a factor of 1/0.5 = 2
- The scaling is calculated by the formula 1 / (1 - `dropout_rate`)

In [125…
```
torch.manual_seed(123)
dropout = torch.nn.Dropout(0.5) # dropout rate of 50%
example = torch.ones(6, 6) # create a matrix of ones

print(dropout(example))
```

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

The next cell runs `torch.manual_seed(123) print(dropout(attn_weights))`.

In [127…
```
torch.manual_seed(123)
print(dropout(attn_weights))
```

```
tensor([[2.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.7599, 0.6194, 0.6206, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.4921, 0.4925, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.3966, 0.0000, 0.3775, 0.0000, 0.0000],
        [0.0000, 0.3327, 0.3331, 0.3084, 0.3331, 0.0000]],
       grad_fn=<MulBackward0>)
```

- Note that the resulting dropout outputs may look different depending on your operating system; you can read more about this inconsistency here on the PyTorch issue tracker

3.5.3 Implementing a compact causal self-attention class

- Now, we are ready to implement a working implementation of self-attention, including the causal and dropout masks
- One more thing is to implement the code to handle batches consisting of more than one input so that our `CausalAttention` class supports the batch outputs produced by the data loader we implemented in chapter 2
- For simplicity, to simulate such batch input, we duplicate the input text example:

In [131…
```
batch = torch.stack((inputs, inputs), dim=0)
print(batch.shape) # 2 inputs with 6 tokens each, and each token has embeddi
```

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

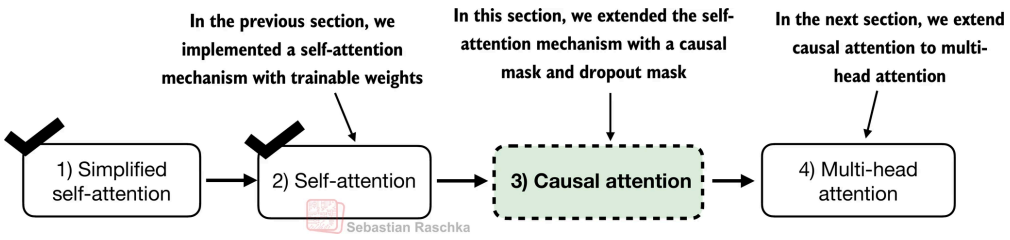
The next cell runs `class CausalAttention(nn.Module): def __init__(self, d_in, d_out, contex`.

In [133…
```
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) # New
        self.register_buffer('mask', torch.triu(torch.ones(context_length, c

    def forward(self, x):
        b, num_tokens, d_in = x.shape # New batch dimension b
        # For inputs where `num_tokens` exceeds `context_length`, this will
        # in the mask creation further below.
        # In practice, this is not a problem since the LLM (chapters 4-7) en
        # do not exceed `context_length` before reaching this forward method
        keys = self.W_key(x)
        queries = self.W_query(x)
        values = self.W_value(x)

        attn_scores = queries @ keys.transpose(1, 2) # Changed transpose
        attn_scores.masked_fill_(  # New, _ ops are in-place
            self.mask.bool()[:num_tokens, :num_tokens], -torch.inf)  # `:num
        attn_weights = torch.softmax(
            attn_scores / keys.shape[-1]**0.5, dim=-1
```

```
        )
        attn_weights = self.dropout(attn_weights) # New

        context_vec = attn_weights @ values
        return context_vec

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)
print("context_vecs.shape:", context_vecs.shape)
```

```
tensor([[[-0.4519,  0.2216],
         [-0.5874,  0.0058],
         [-0.6300, -0.0632],
         [-0.5675, -0.0843],
         [-0.5526, -0.0981],
         [-0.5299, -0.1081]],

        [[-0.4519,  0.2216],
         [-0.5874,  0.0058],
         [-0.6300, -0.0632],
         [-0.5675, -0.0843],
         [-0.5526, -0.0981],
         [-0.5299, -0.1081]]], grad_fn=<UnsafeViewBackward0>)
context_vecs.shape: torch.Size([2, 6, 2])
```

- Note that dropout is only applied during training, not during inference



Exercise (3.5 Causal mask)

A 4-step chat is ordered `[system, user, assistant, user]`. Build a boolean causal mask (`True` where token `i` may look at token `j`). Then list the steps the **last user** token may attend to. Do not count entries in a demo `masked` tensor.

In [ ]:
```
import torch

T = 4
allowed = None
last_user_can_see = None
print(allowed)
print('last user may look at', last_user_can_see)
```

Solution (3.5). Lower-triangular True; row 3 is positions 0–3.

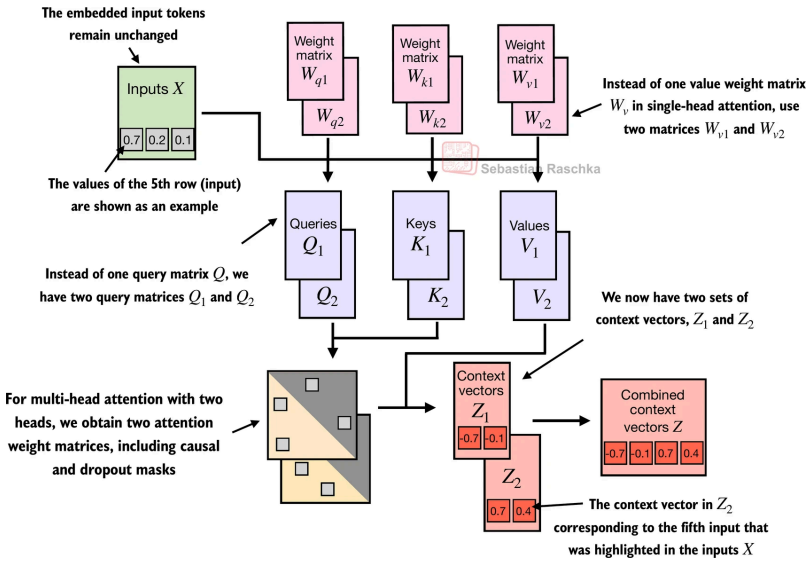
In [1]:
```
import torch

T = 4
allowed = torch.tril(torch.ones(T, T, dtype=torch.bool))
last_user_can_see = torch.nonzero(allowed[3], as_tuple=False).flatten().toli
print(allowed)
print('last user may look at', last_user_can_see)
```

```
tensor([[ True, False, False, False],
        [ True,  True, False, False],
        [ True,  True,  True, False],
        [ True,  True,  True,  True]])
last user may look at [0, 1, 2, 3]
```

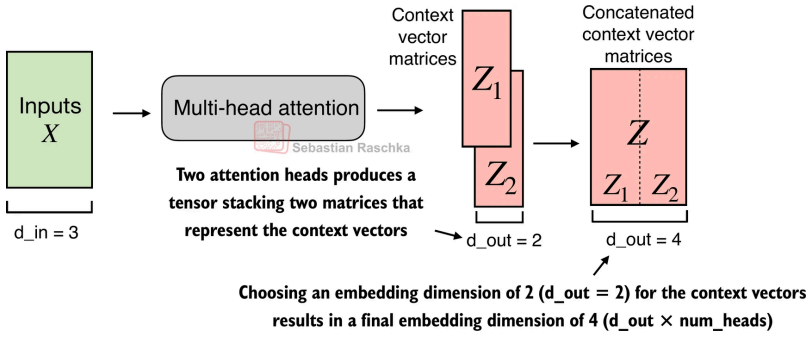
3.6 Extending single-head attention to multi-head attention

3.6.1 Stacking multiple single-head attention layers

- Below is a summary of the self-attention implemented previously (causal and dropout masks not shown for simplicity)
- This is also called single-head attention:



- We simply stack multiple single-head attention modules to obtain a multi-head attention module:



- The main idea behind multi-head attention is to run the attention mechanism multiple times (in parallel) with different, learned linear projections. This allows the model to jointly attend to information from different representation subspaces at different positions.

In [143…

```
class MultiHeadAttentionWrapper(nn.Module):

    def __init__(self, d_in, d_out, context_length, dropout, num_heads, qkv_
        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)


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])
```

- In the implementation above, the embedding dimension is 4, because we `d_out=2` as the embedding dimension for the key, query, and value vectors as well as the context vector. And since we have 2 attention heads, we have the output embedding dimension 2*2=4

3.6.2 Implementing multi-head attention with weight splits

- While the above is an intuitive and fully functional implementation of multi-head attention (wrapping the single-head attention `CausalAttention` implementation from earlier), we can write a stand-alone class called `MultiHeadAttention` to achieve the same
- We don't concatenate single attention heads for this stand-alone `MultiHeadAttention` class
- Instead, we create single W_query, W_key, and W_value weight matrices and then split those into individual matrices for each attention head:

In [147…

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_in, d_out, context_length, dropout, num_heads, qkv_
        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 # Reduce the projection dim to ma

        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)  # Linear layer to combine h
        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
        # As in `CausalAttention`, for inputs where `num_tokens` exceeds `co
        # this will result in errors in the mask creation further below.
        # In practice, this is not a problem since the LLM (chapters 4-7) en
        # do not exceed `context_length` before reaching this forward method

        keys = self.W_key(x) # Shape: (b, num_tokens, d_out)
        queries = self.W_query(x)
        values = self.W_value(x)

        # We implicitly split the matrix by adding a `num_heads` dimension
        # Unroll last dim: (b, num_tokens, d_out) -> (b, num_tokens, num_hea
        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)

        # Transpose: (b, num_tokens, num_heads, head_dim) -> (b, num_heads,
        keys = keys.transpose(1, 2)
        queries = queries.transpose(1, 2)
        values = values.transpose(1, 2)

        # Compute scaled dot-product attention (aka self-attention) with a c
        attn_scores = queries @ keys.transpose(2, 3)  # Dot product for each

        # Original mask truncated to the number of tokens and converted to b
        mask_bool = self.mask.bool()[:num_tokens, :num_tokens]

        # Use the mask to fill attention scores
        attn_scores.masked_fill_(mask_bool, -torch.inf)

        attn_weights = torch.softmax(attn_scores / keys.shape[-1]**0.5, dim=
        attn_weights = self.dropout(attn_weights)

        # Shape: (b, num_tokens, num_heads, head_dim)
        context_vec = (attn_weights @ values).transpose(1, 2)

        # Combine heads, where self.d_out = self.num_heads * self.head_dim
        context_vec = context_vec.contiguous().view(b, num_tokens, self.d_ou
        context_vec = self.out_proj(context_vec) # optional projection

        return context_vec
```

```
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])
```

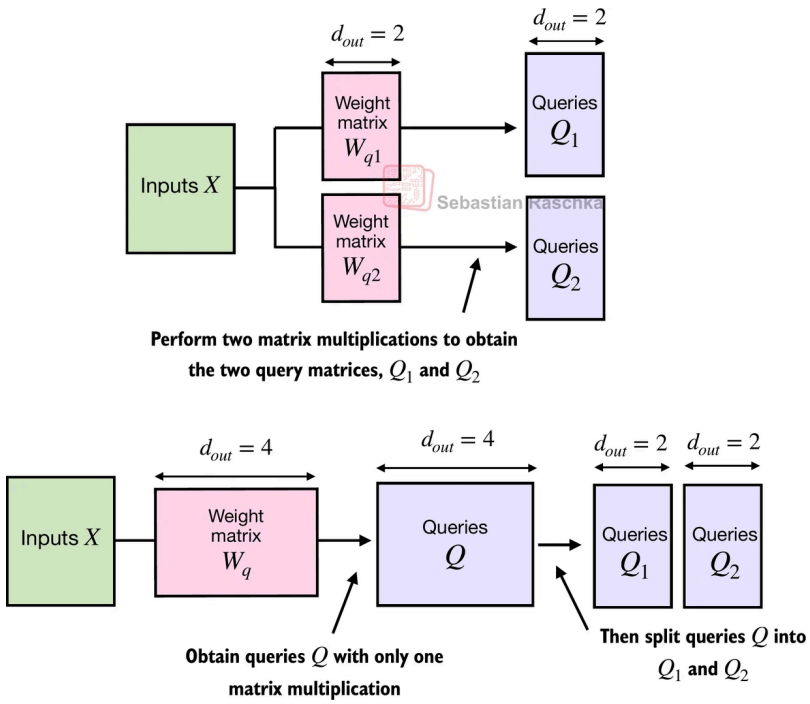
- Note that the above is essentially a rewritten version of `MultiHeadAttentionWrapper` that is more efficient
- The resulting output looks a bit different since the random weight initializations differ, but both are fully functional implementations that can be used in the GPT class we will implement in the upcoming chapters

---

A note about the output dimensions

- In the `MultiHeadAttention` above, I used `d_out=2` to use the same setting as in the `MultiHeadAttentionWrapper` class earlier
- The `MultiHeadAttentionWrapper`, due the the concatenation, returns the output head dimension `d_out * num_heads` (i.e., `2*2 = 4`)
- However, the `MultiHeadAttention` class (to make it more user-friendly) allows us to control the output head dimension directly via `d_out`; this means, if we set `d_out = 2`, the output head dimension will be 2, regardless of the number of heads
- In hindsight, as readers pointed out, it may be more intuitive to use `MultiHeadAttention` with `d_out = 4` so that it produces the same output dimensions as `MultiHeadAttentionWrapper` with `d_out = 2`.

- Note that in addition, we added a linear projection layer (`self.out_proj`) to the `MultiHeadAttention` class above. This is simply a linear transformation that doesn't change the dimensions. It's a standard convention to use such a projection layer in LLM implementation, but it's not strictly necessary (recent research has shown that it can be removed without affecting the modeling performance; see the further reading section at the end of this chapter)



- Note that if you are interested in a compact and efficient implementation of the above, you can also consider the `torch.nn.MultiheadAttention` class in PyTorch

- Since the above implementation may look a bit complex at first glance, let's look at what happens when executing `attn_scores = queries @ keys.transpose(2, 3)` :

In [154…
```
# (b, num_heads, num_tokens, head_dim) = (1, 2, 3, 4)
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]]]])

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

```
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 will handle the 4-dimensional input tensor so that the matrix multiplication is carried out between the 2 last dimensions (num_tokens, head_dim) and then repeated for the individual heads
- For instance, the following becomes a more compact way to compute the matrix multiplication for each head separately:

In [156…
```python
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)
```

```
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]])
```

Exercise (3.6 Multi-head)

Two heads each produced a `(1, 3, 2)` context. Concatenate them the way the class does, then recover **head 1** at token 0 and print the sum of that vector. (If you only print a shape, the split is not done.)

In [ ]:
```python
import torch

head0 = torch.tensor([[[1.0, 2.0], [0.0, 1.0], [3.0, 0.0]]])
head1 = torch.tensor([[[4.0, 0.0], [1.0, 1.0], [0.0, 2.0]]])
concat = None
recovered = None
print(recovered)
```

Solution (3.6). Concatenate on the last dim, view into heads, pick head 1 / token 0.

In [1]:
```
import torch

head0 = torch.tensor([[[1.0, 2.0], [0.0, 1.0], [3.0, 0.0]]])
head1 = torch.tensor([[[4.0, 0.0], [1.0, 1.0], [0.0, 2.0]]])
concat = torch.cat([head0, head1], dim=-1)
b, t, c = concat.shape
heads = concat.view(b, t, 2, c // 2)
recovered = heads[0, 0, 1]
print(recovered)
print(float(recovered.sum()))
```

```
tensor([4., 0.])
4.0
```

Comprehensive exercises

Exercise (comprehensive: causal + heads)

A 5-token window, 2 heads of size 2. Build a causal mask, apply it to a 5×5 matrix of ones (`-inf` on the future), softmax, and print **row 0** (only the current token should have mass). Then print the concatenated last-dimension width of the two heads.

In [ ]:
```
import torch

T = 5
n_heads = 2
head_dim = 2
row0 = None
concat_width = None
print('row 0', row0)
print('concat width', concat_width)
```

Solution. Row 0 of a causal softmax is one-hot on position 0. Concat width is `n_heads * head_dim`.

In [1]:
```
import torch

T = 5
n_heads = 2
head_dim = 2
mask = torch.tril(torch.ones(T, T, dtype=torch.bool))
scores = torch.ones(T, T)
scores = scores.masked_fill(~mask, float('-inf'))
row0 = torch.softmax(scores, dim=-1)[0]
concat_width = n_heads * head_dim
print('row 0', row0)
print('concat width', concat_width)
```

```
row 0 tensor([1., 0., 0., 0., 0.])
concat width 4
```

Takeaways

1. **Causal** masking sets the future to `-inf` before softmax.
2. **Multi-head** runs several attentions and concatenates.
3. Next class: those heads sit inside a GPT block with LayerNorm and a feed-forward.