

Lecture 18: Multi Head Attention Part2

Instead of maintaining 2 separate classes; MultiHeadAttentionWrapper and CausalAttention, we combine both of these into a single MultiHeadAttention class.

Step 1:

Start with the input (no. of batches, no. of tokens, $d_{in}=dim$ of the vector embedding)

```
b, num_tokens, d_in = x.shape
```

Step 2:

Decide d_{out} and number of heads.

In GPT-2 model, $d_{in}=d_{out}$.

head_dimension = $d_{out} / \text{num_of_heads}$

Step 3:

Initialize trainable weight matrices for key, query, value (W_k, W_q, W_v)

Dimension will be: $d_{in} \times d_{out}$

```
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)
```

Step 4:

Calculate Keys, Queries and Value matrix

(Input* W_k , Input* W_q , Input* W_v)

Dimension: (no of batches, num_tokens, d_{out})

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

Step 5:

Unroll last dimension of Keys, Queries and Values to include num_heads and head_dim

We know, head_dimension = $d_{out} / \text{num_of_heads}$

So, dimension of Keys, Queries and Values = (no of batches, num_tokens, num_of_heads, head_dimension)

```
Reshaped Queries matrix:
tensor([[[[-0.4888,  0.2361,  2.8463],
         [-0.2184,  5.4503, -1.8915]],

        [[-2.3531, -0.7912,  1.2578],
         [ 2.0534, -4.3369,  3.2125]],

        [[-2.5745,  0.2893,  1.1454],
         [ 0.9021,  1.5632,  0.6930]]], dtype=torch.float64)
```

Code:

```
# Unroll last dim: (b, num_tokens, d_out) -> (b, num_tokens, num_heads, head_dim)
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)
```

Step 6:

Group matrices by num_heads

(batches, num_tokens, num_heads, head_dim) → (b, num_heads, num_tokens, head_dim)

```
keys = keys.transpose(1, 2)
queries = queries.transpose(1, 2)
values = values.transpose(1, 2)
```

(1,3,2,3) → (1,2,3,3)

We do this so that we can do the parallel computing of attention scores more effectively.

Step 7:

Find attention scores: Queries*Keys Transpose (2,3)



(b, num_heads, num_tokens, head_dim) * (b, num_heads, head_dim, num_tokens) → (b, num_heads, num_tokens, num_tokens)

Step 8:

Find attention weights: Mask attention scores to implement causal attention

Divide by sqrt of head_dim

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

# Original mask truncated to the number of tokens and converted to boolean
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=-1)
attn_weights = self.dropout(attn_weights)
```

Step 9:

Context vector = Attention weights * Values

= (b, num_heads, num_tokens, num_tokens) * (b, num_heads, num_tokens, head_dim)

= (b, num_heads, num_tokens, head_dim)

But there's a problem here. We need to merge the num_heads and head_dim back together. Because the resultant context vector should have dimension d_out. The d_out dimension must be preserved.

Step 10:

Reformat context vectors:

(b, num_heads, num_tokens, head_dim) → (b, num_tokens, num_heads, head_dim)

```
tensor([[[[-3.6194,  2.0935,  1.3879],
          [ 1.5961, -0.9367, -0.3400]],

         [[-1.8728,  1.6494,  0.7163],
          [ 1.3712,  0.7805,  0.2233]],

         [[-0.3553, -0.5607, -0.1181],
          [ 0.6997, -0.4487, -0.0530]]]])
```

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

So the final shape = (b, num_tokens, d_out)

DETAILED EXPLANATION OF THE MULTI-HEAD ATTENTION CLASS

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 $\text{head_dim} = \text{d_out} / \text{num_heads}$.

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.
