By: Tejas Prabhune, Ayushi Batwara, Tim Xie, Derek Xu

Overview

In the last two lectures, we've thoroughly covered the popular Transformer architecture for sequence-to-sequence modeling, representation learning, and autoregressive generation. Before we jump into **HW 3: Transformers**, let's warm up with reviewing and implementing the logic of the architecture in pseudocode.

At any point in time, feel free to review both the Attention Is All You Need paper or the Homework 3 informational document that was sent to you. We **highly** recommend not using LLM-based tools for this assignment—this is purely for your understanding, and you'll greatly benefit from thinking through all the nitty-gritty details yourself.

If you have ANY questions at all, do **not** hesitate to ask any Edu staff! We hope this will aid your intuition of the architectural concepts before starting to code!

Background Questions

1. What is the input to the Encoder?

2. What are the two main components in an Encoder layer?

    a. Briefly describe what they do.

3. What is the output of the Encoder?

Multi-Head Attention

You are given the following class and `__init__` function signature on the next page.

First, briefly answer the following questions about Multi-Head Attention:

1. What do `Q`, `K`, and `V` represent?

2. What are the shapes of the `Q`, `K`, and `V` matrices when first inputted into the Multi-Head Attention module? Since we want to repeat this operation in parallel `num_heads` times, how should we project (i.e., apply a linear transformation to) each `Q`, `K`, and `V` matrices? What will be the shape after the projections?

    Note: Assume that you have two values: `qk_length` and `value_length` that represent the length of your query/key embeddings and value embeddings. You can generalize this length to be `vec_length`, which will be `qk_length` or `value_length` based on if you are working on `Q, K` or `V`.

    Note: While during lecture it makes sense to consider our embeddings as having a size of (B, C, T), we prefer to use the inverse format of (B, T, C) in code (hint: look at the `torch.nn.Linear` documentation for why this may be the case—if unclear, please ask!).

3. After these projections, we need to make each of the `Q`, `K`, `V` tensors suitable for parallel processing of each head. To do this, we use a `split_heads` function. What are the initial and final shapes of `Q`, `K`, and `V` tensors for this function?

4. Why do we scale the output of the dot product attention by a factor of $\frac{1}{\sqrt{\text{qk\_length}}}$? What would happen if we didn't have this scaling factor?

5. After we have the scaled dot-product attention result (reminder: shape will be $(B, \text{num\_heads}, T, \text{value\_length})$ since we've also multiplied with the `V` tensor), we need to complete the `Concat` and `Linear` part of the module. We implement the `Concat` part as a `combine_heads` function. What will be the shape before and after this function?

Now based on your understanding of Multi-Head Attention layers, write the remaining pseudocode for the `__init__` function:

```
class MultiHeadAttention(nn.Module):
    def __init__(self,
                 num_heads: int,
                 embedding_dim: int,
                 qk_length: int,
                 value_length: int
                 ):
        """
        The Multi-Head Attention layer will take in Q, K, and V
        matrices and will output an attention matrix of shape <TODO>.

        First, Q, K, and V should be projected to have
        a shape of (B, T, C) where C = num_heads * qk_length. You are
        then expected to split the C dimension into num_heads
        different heads, each with shape (B, T, qk_length).

        Next, you will compute the scaled dot-product attention
        between Q, K, and V.

        Finally, you will concatenate the heads and project the
        output to have a shape of (B, T, C).
        """
        super().__init__()

        self.num_heads = num_heads
        self.embedding_dim = embedding_dim
        self.qk_length = qk_length
        self.value_length = value_length

        # Define layers you'll need in the forward pass (hint: there are 4 lol)
        # Note: these are learnable parameters and will change
        # throughout the model training process
        ________________________________________________________________
        ________________________________________________________________
        ________________________________________________________________
        ________________________________________________________________
```

Recall our discussion about the attention layer and how it performs a "lookup" given a query against our set of keys. Then, we get the corresponding values based on our lookup. Write the remaining pseudocode for the `scaled_dot_product_attention` function:

```
def scaled_dot_product_attention(self, _________, _________, _________, _________):
  lookup = ___________________________
  scaled_lookup = ___________________________
  attention = ___________________________
  return ___________________________
```

Write the pseudocode `split_heads` function:

```
def split_heads(self, _________, _________) -> torch.Tensor:
  ________________________________________________________________
  ________________________________________________________________
  ________________________________________________________________
  ________________________________________________________________
```

Now the `combine_heads` function:

```
def combine_heads(self, _________) -> torch.Tensor:
  ________________________________________________________________
  ________________________________________________________________
  ________________________________________________________________
  ________________________________________________________________
```

Now that we have all the components for a forward pass through our Encoder, let's combine them in our `forward` function. Write the remaining pseudocode:

```
def forward(self, _________, _________, _________) -> torch.Tensor:
  # Applying the respective layers to the inputs
  ________________________________________________________________
  ________________________________________________________________
  ________________________________________________________________

  # Preparing the inputs for parallel processing for each attention head
  ________________________________________________________________
  ________________________________________________________________
  ________________________________________________________________

  # Applying the attention layer
  attention = ____________________________________________________

  # Combining the mutliple "heads"
  attention = ____________________________________________________

  # Applying the respective layer to the attention output
  attention = ____________________________________________________

  return attention
```