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?

   Solution: The input to the Encoder is the tokenized source input sequence. For example, in a machine translation task from French to English, this input would be the tokenized French.

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

   Solution: Each Encoder layer has a Multi-Head Attention layer and a "Position-Wise" Feed-Forward Network. Position-wise is slightly ambiguous. All it means is that given an embedding with shape (B, T, C), we use a feed-forward network that applies along the C dimension, i.e. along each token's (position's) embedding.

   a. Briefly describe what they do.

      Solution: For each input embedding, the Multi-Head Attention layers add "attended" information from each other input embedding based on relevance according to some similarity function. The feed-forward network simply adds additional nonlinear complexity to the model.

3. What is the output of the Encoder?

   Solution: The Encoder's output will be a tensor with shape (B, T, C), which can then be inputted into the decoder for sequence-to-sequence tasks.

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?

   Solution: `Q`, `K`, and `V` (this is a poor convention in the literature) originally represent our *initial* embeddings, before any changes or attention operations. In self-attention, all three matrices will be the input embedding, since we are doing attention between the embedding and itself.

   In cross-attention, `Q` will be the "target" embedding (target simply means the ongoing English decoding), `K`, `V` will be from the source embedding (French). We want to query the French using English and use the values that we retrieve.

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!).

   Solution:

   a. Tokenized inputs are of shape (B, T)

   b. After applying an embedding function (just a weight matrix times one-hot encodings): our shape becomes (B, T, C).

   c. **Q1: `Q`, `K`, `V` have shape (B, T, C) when first inputted into Multi-Head Attention**. They are all equal to the embedding we computed in step (b).

      We eventually want `Q`, `K`, and `V` to have shape $(B, \text{num\_heads}, T, \text{vec\_length})$ before we apply our attention operation. Why? We want to parallelize the attention operation using tensor optimizations in the GPU. This is a common principle in PyTorch, where we will place the dimension that we want to be parallelized near the front of the tensor (i.e. `B` is the first dimension, then `num_heads` is the second dimension). BUT, we can't immediately conjure up a new dimension while doing the projection (the `Linear` layer in PyTorch). So, we first project to include all the information, then split the tensor to have 4 dimensions.

   d. **Q2: `Q`, `K`, `V` should be projected to include all the information we need for all heads.**

   e. **Q3: `Q`, `K`, `V` will have shape $(B, T, \text{num\_heads} \cdot \text{vec\_length})$ after the initial projection**.

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?

   Solution:

   Now we will split our data-full tensors so they are suitable for parallelization.

   Initial shapes: $(B, T, \text{num\_heads} \cdot \text{vec\_length})$

   Final shapes: $(B, \text{num\_heads}, T, \text{vec\_length})$

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?

   Solution: Consider one element of our dot product attention QK^T. Let's derive what this element looks like. The shapes (from previous parts) of Q and K are $(B, \text{num\_heads}, T, \text{qk\_length})$. From this point on, we will disregard B and num_heads because these are simply for parallelized computation. So we have:

$$Q, K \in \mathbb{R}^{T \times \text{ qk\_length}}$$

For a sanity check, after performing QK^T, our matrix has shape $T \times T$. If we take the very first element and think back to how we got it, we will see that we performed $q_1 \cdot k_1$, where q_1 and k_1 are the first query and key embeddings (from our projection).

$$q_1, k_1 \in \mathbb{R}^{\text{qk\_length}}$$

At the very start of training, we can assume all of the components of q_1 and k_1 are independent random variables following a standard Gaussian distribution:

$$q_{1i}, k_{1i} \sim \mathcal{N}(0, 1) \qquad i \in \{1, \ldots, \text{qk\_length}\}$$

We can ask a very simple question: what will be variance of $q_1 \cdot k_1$? What are the implications of this variance?

To compute the variance, we can write out this dot product:

$$\text{Var}(q_1 \cdot k_1) = \text{Var}\left(q_{11}k_{11} + \cdots + q_{1\ \text{qk\_length}}k_{1\ \text{qk\_length}}\right)$$

Due to independence of the components, we can "distribute" the variance operation:

$$= \text{Var}(q_{11}k_{11}) + \cdots + \text{Var}\left(q_{1\ \text{qk\_length}}k_{1\ \text{qk\_length}}\right)$$

Recall that for independent random variables, we have:

$$\text{Var}(XY) = \text{Var}(X)\text{Var}(Y) + \text{Var}(X)\mathbb{E}[Y]^2 + \text{Var}(Y)\mathbb{E}[X]^2$$

So for q_{1i}, k_{1i}, we have:

$$\text{Var}(q_{1i}k_{1i}) = \text{Var}(q_{1i})\text{Var}(k_{1i}) + \text{Var}(q_{1i})\mathbb{E}[k_{1i}]^2 + \text{Var}(k_{1i})\mathbb{E}[q_{1i}]^2$$

We know the expectation of q_{1i} and k_{1i}! From our initial assumption of standard Gaussian, both of these expectations is just 0. We also know from this assumption that each random variable's variance is 1. So we can simplify to:

$$\text{Var}(q_{1i}k_{1i}) = \text{Var}(q_{1i})\text{Var}(k_{1i}) = 1$$
$$\text{Var}(q_1 \cdot k_1) = 1 + \cdots + 1 = \text{qk\_length}$$

We get an interesting result—our variance scales with `qk_length`! So at initialization, if we set `qk_length = 1024` or some other high value, our gradients for every element will move us in some arbitrary direction just purely based on the high variance. This makes training unstable in practice, so we need to remedy this by normalizing our variance to 1. Recall the identity:

$$\text{Var}(\alpha X) = \alpha^2 \text{Var}(X)$$

Substituting X for $q_1 \cdot k_1$ and solving for α given that our overall variance should be 1:

$$\begin{aligned} 1 &= \text{Var}(\alpha(q_1 \cdot k_1)) = \alpha^2 \text{Var}(q_1 \cdot k_1) \\ 1 &= \alpha^2 \cdot \text{qk\_length} \\ \alpha^2 &= \frac{1}{\text{qk\_length}} \\ \alpha &= \frac{1}{\sqrt{\text{qk\_length}}} \end{aligned}$$

Thus, we normalize our matrix multiplication by multiplying by a factor of $\frac{1}{\sqrt{\text{qk\_length}}}$ so our initial gradients don't overshoot in arbitrary directions. This ensures stable training.

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?

   Solution: We simply undo our previous operation to only have three dimensions.

   Initial shape: $(B, \text{num\_heads}, T, \text{value\_length})$

   Final shape: $(B, T, \text{num\_heads} \cdot \text{value\_length})$

   Note: the final `Linear` layer will then take $\text{num\_heads} \cdot \text{value\_length}$ as the input hidden dimension, and output embedding_dim as the final hidden dimension.

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

Solution:

```
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
        self.W_q = Linear from embedding_dim to num_heads * qk_length
        self.W_k = Linear from embedding_dim to num_heads * qk_length
        self.W_v = Linear from embedding_dim to num_heads * value_length
        self.W_o = Linear from num_heads * value_length to embedding_dim
```

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:

Solution:

```
def scaled_dot_product_attention(self, Q: Tensor, K: Tensor, V: Tensor,
  mask: Optional[Tensor] = None) -> Tensor:
  lookup = Q times K^T
  scaled_lookup = lookup / sqrt(qk_length)
  attention = softmax(scaled_lookup)
  return attention times V
```

Write the pseudocode `split_heads` function:

Solution:

```
def split_heads(self, x: Tensor, vec_length: int) -> Tensor:
  assert correct shape for splitting
  x = view x in the shape (B, T, num_heads, vec_length)

  return permuted x in shape (B, num_heads, T, vec_length)
```

Now the `combine_heads` function:

Solution:

```
def combine_heads(self, x: Tensor) -> Tensor:
  x = permuted x in shape (B, T, num_heads, vec_length)

  return contiguous x viewed in shape (B, T, num_heads * vec_length)
```

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:

Solution:

```
def forward(self, Q: Tensor, K: Tensor, V: Tensor) -> Tensor:
  # Applying the respective layers to the inputs
  Q = self.W_q(Q)
  K = self.W_k(K)
  V = self.W_v(V)

  # Preparing the inputs for parallel processing for each attention head
  Q = split heads with Q and qk_length
  K = split heads with K and qk_length
  V = split heads with V and value_length

  # Applying the attention layer
  attention = scaled dot product attention with Q, K, V

  # Combining the mutliple "heads"
  attention = combine heads with attention

  # Applying the respective layer to the attention output
  attention = self.W_o(attentiont)

  return attention
```