Lab 1: The One Where You Learn to Attend

Introduction

In recent years, self-attention has emerged as one of the most groundbreaking concepts in Natural Language Processing (NLP), becoming the foundation for models like Transformers, BERT, and GPT. Unlike traditional sequence models that process information sequentially, self-attention allows each word in a sequence to focus on all other words simultaneously, capturing global dependencies regardless of distance in the text.

This laboratory practice aims to give you hands-on experience in understanding self-attention from scratch, visualizing attention scores, and exploring the role of positional encodings in sequence models.

Throughout this practice, you will:

- Implement self-attention from scratch to deepen your comprehension of how it works under the hood.
- Visualize the attention matrices using `BertViz` to see how different words relate to each other in a sentence.
- Investigate the impact of positional encodings on model performance and how they allow self-attention models to differentiate between different positions in a sequence.

The concepts covered here are central to modern NLP and are applicable in a wide range of tasks, from language modeling and machine translation to text classification and question answering.

Learning Outcomes

Upon completion of this laboratory exercise, participants will have acquired the ability to:

- Understand the mechanics of self-attention, including the calculation of attention scores and how these scores guide model focus.
- Explore positional encodings and understand their importance in self-attention mechanisms, where input sequences are processed without inherent order.

Deliverables

1. `models.py` - This file must define the necessary encoder blocks for a Transformer model. Your implementation should include the AttentionHead, MultiHeadAttention, FeedForward, TransformerEncoderLayer, and TransformerEncoder classes.

These files should be uploaded to Github maintaining the format handle in the laboratory practice.

Exercises

Exercise 1: Pay (single) attention!

In this exercise, you will implement the `AttentionHead` class. This class calculates the attention scores for a single head and applies them to the input data.

1. Implement the `AttentionHead` class in `models.py`. It should include linear layers to project the input into query (W^Q), key (W^K), and value (W^V) vectors.

2. Define the `scaled_dot_product_attention` method to calculate attention scores using these vectors.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$

   where Q (queries), K (keys), and V (values) are the projected vectors, and d_k is the dimension of the key vectors.

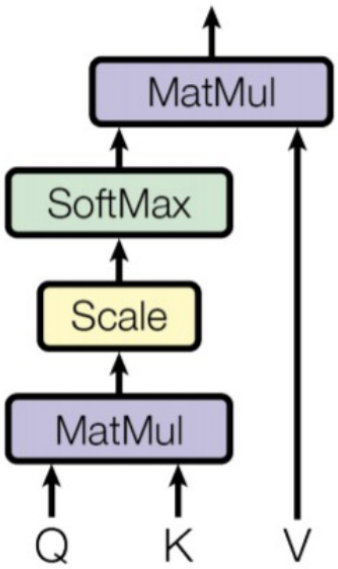

Figure 1: Scaled dot product attention diagram

3. Ensure that the `forward` method processes the input correctly and returns the attention-weighted output. Remember to project the inputs before calculating the attention scores.

Exercise 2: Multi-Tasking is real.

In this exercise, you will implement the `MultiHeadAttention` class, which combines multiple attention heads to enhance the model's ability to focus on different parts of the input simultaneously.

1. Implement the `MultiHeadAttention` class in `models.py`. This class should:

   - Initialize multiple `AttentionHead` instances, each with its own set of parameters.

- Concatenate the outputs from each attention head.
- Apply a final linear transformation to the concatenated output.

2. The attention mechanism is defined as follows:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_h)W^O \tag{2}$$

where each attention head head_i is computed as:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{3}$$

where W_i^Q, W_i^K, W_i^V are the learned projections for queries, keys, and values for head_i, and W^O is the learned projection after concatenating the heads. Although the size of the query, key and value vectors (d_k, d_q and d_v) in each head do not have to be smaller than the number of embedding dimensions of the tokens (d_{model}), in practice, we often use $d_k = d_q = d_v$, and set this value to be a fraction of d_{model} so that the computation across each head is constant. For example, BERT has 12 attention heads, so the dimension of each head is $768/12 = 64$. You should also set the sizes of the key, query and value vectors in this way: $d_k = d_q = d_v = \frac{d_{model}}{num_{heads}}$.

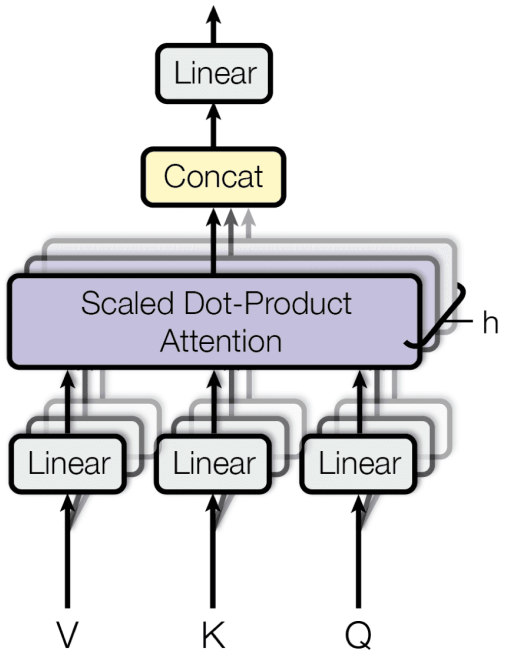

Figure 2: Multihead Attention diagram

3. Ensure that the `forward` method processes the input across all heads and returns the final output.

Exercise 3: Deep Learning haunts you

In this exercise, you will implement the `FeedForward` block. This block applies two linear transformations with a GELU activation function in between, followed by a dropout layer. The original transformer from "Attention is all you need" used ReLU, but GELU are commonly used as well.

1. Implement the `FeedForward` class in `models.py`. This class should:
   - Apply a linear transformation to the input.
   - Pass the result through a GELU activation function.
   - Apply a second linear transformation.
2. You do not need to implement the backward method. For now.

Hint #1:

Remember that the dimensions of the input and output should remain consistent across the layers, except within the hidden dimension inside the feedforward network.

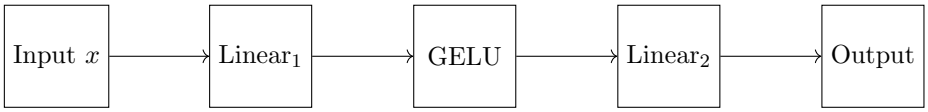

Figure 3: FeedForward Layer Structure

Exercise 4: Hold your positions!

In this exercise, you will implement the `Embeddings` block, which is the first step in the Transformer model. This block combines token embeddings with positional encodings to provide the model with both word meanings and their positions in the sequence.

1. Implement the `Embeddings` class in `models.py`. This class should:
   - Apply token embeddings to convert input token indices into dense vectors.
   - Add positional embeddings to these token embeddings, helping the model understand the order of tokens.
   - Apply layer normalization to the combined embeddings. The original "Attention is all you need" paper did not apply layer norm and dropout here, but is commonly done as well.
2. The embeddings can be represented mathematically as:

$$\text{Embeddings}(x) = \text{Dropout}(\text{LayerNorm}(\text{TokenEmbeddings}(x) + \text{PositionalEmbeddings}(x))) \quad (4)$$

where x is the input token indices.

3. Ensure that the `forward` method applies these operations in the correct sequence and returns the final embeddings.

Hint #2:

Consider the maximum length of the input sequence when creating positional embeddings. Ensure that the dimensions of the token and positional embeddings match.

Exercise 5: And these are your eyes!

In this exercise, you will implement the `TransformerEncoderLayer` class, which forms the core of the Transformer encoder. This layer includes a multi-head attention mechanism followed by a feedforward network, with residual connections and layer normalization applied at each step.

1. Implement the `TransformerEncoderLayer` class in `models.py`. This class should:
   - Apply layer normalization to the input.
   - Apply multi-head attention to the normalized input.
   - Add a residual connection from the input to the output of the multi-head attention.
   - Apply layer normalization to the result.
   - Pass the normalized result through a feedforward network.
   - Add a residual connection from the result of the multi-head attention to the output of the feedforward network.
2. Ensure that the `forward` method applies these operations in the correct sequence and returns the final output.

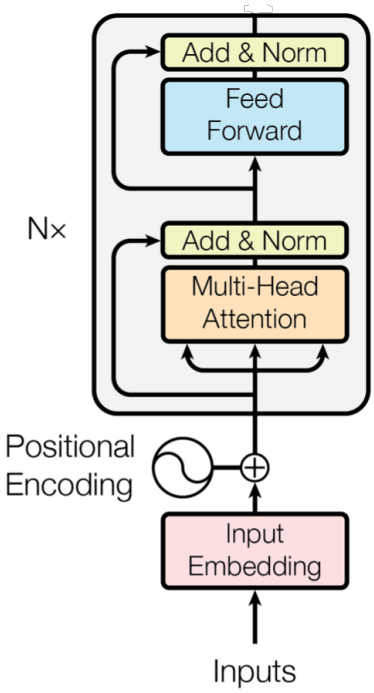

Figure 4: Encoder diagram

Hint #3:

Pay attention to the order of operations, especially with layer normalization and residual connections. These are crucial for the model's performance and stability.

Exercise 6: What are you looking at?!

In this final exercise, you will explore the `attention_visualization.ipynb` notebook, which demonstrates how to visualize the attention mechanisms within a Transformer model using the BertViz tool.

Open the `attention_visualization.ipynb` notebook and follow the instructions provided within. The notebook contains pre-loaded visualizations that show how the Transformer model attends to different parts of the input sequence across various layers and heads. Answer the questions embedded in the notebook.

Test your work!

Specific test functions have been developed to test your work. These tests are designed to automatically check the correctness of your implementations, from calculating attention to ensuring dimensions for the model.

Useful links

1. **torch.bmm**: Performs a batch matrix-matrix product of matrices stored in `input` and `mat2`. This function is crucial for implementing attention mechanisms where matrix multiplication is performed across batches.
   Documentation: `https://pytorch.org/docs/stable/generated/torch.bmm.html`

2. **torch.nn.GELU**: Applies the Gaussian Error Linear Unit (GELU) activation function, which is smoother than ReLU and commonly used in Transformer architectures.
   Documentation: `https://pytorch.org/docs/stable/generated/torch.nn.GELU.html`

3. **torch.nn.LayerNorm**: Applies layer normalization over a mini-batch of inputs. Layer normalization is essential in Transformer models for stabilizing the training process and improving convergence.
   Documentation: `https://pytorch.org/docs/stable/generated/torch.nn.LayerNorm.html`

4. **torch.nn.ModuleList**: Holds submodules in a list. 'ModuleList' is useful for storing a list of layers or blocks that can be iterated over during the forward pass. Each submodule can be indexed like a regular Python list.
   Documentation: `https://pytorch.org/docs/stable/generated/torch.nn.ModuleList.html`