Programming 1: Neural Machine Translation (26 points)

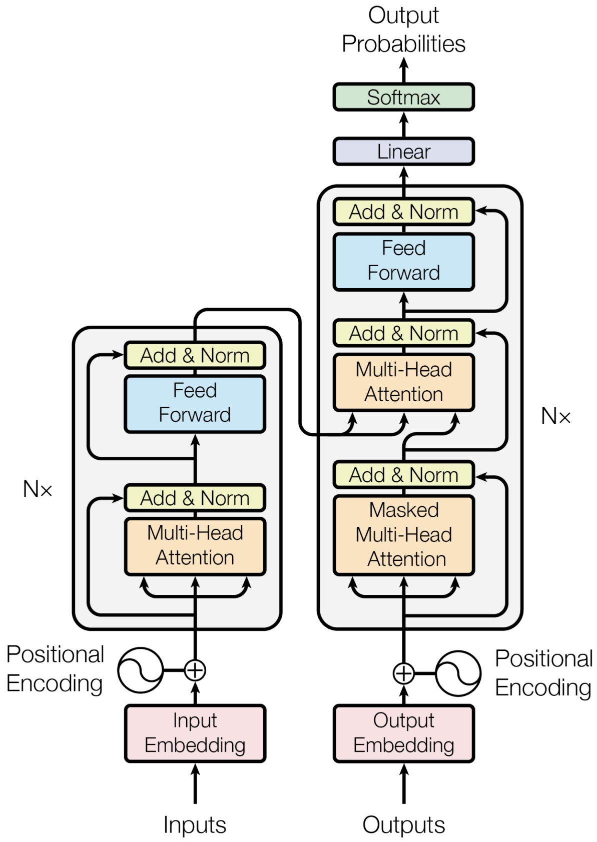
In this problem, you will implement a sequence-to-sequence (seq2seq) model based on the Transformer architecture to build a neural machine translation (NMT) system (translating from French to English).

Code setup Start from the code in this Colab notebook. Please follow the instructions in the notebook to complete TODOs and train and evaluate the seq2seq model.

Data We will use a dataset consisting of parallel French and English sentences and partitioned into training, validation and test splits. You will need to tokenize the data using French and English tokenizers which are provided with the assignment resources. We will use BPE (byte pair encoding) tokenization, which can split a less common word into multiple subword tokens. If you are interested in learning more about BPE, see the paper Neural Machine Translation of Rare Words with Subword Units, or this blog post.

Model The model we will be implementing is an encoder-decoder Transformer model. For simplicity, we make several modifications to the architecture described in this paper:

- We use learned positional embeddings instead of sinusoidal positional embeddings.[1]
- No dropout in the embedding layer and the attention weights.
- We use weight tying between the decoder's input embeddings and the output projection matrix (we have learned this in the class!).



We will describe each block of the model. Let d be the embedding dimension of input embeddings and hidden states, N the maximum sequence length and $V^{(e)}$ and $V^{(d)}$ the vocab sizes for encoder and decoder vocabulary, respectively. Let n be the length of a particular input sequence. In practice, the model will process B sequences in a batch in parallel and the sequence might contain pad tokens, which should be excluded from the attention mechanism and the loss function.

[1] The authors find that this does not impact performance in their experiments. See Table 3 in the Transformers paper.

- **Embedding:** Let $\mathbf{E}^{(e)} \in \mathbb{R}^{V^{(e)} \times d}$ be the token embedding tables and $\mathbf{P}^{(e)} \in \mathbb{R}^{N \times d}$ be the positional embedding tables for the encoder. To embed a token with token id t at position i, we look up the token embedding at index t and position embedding at index i and add the two embeddings. The same procedure is done in the decoder embedding layer with separate matrices $\mathbf{E}^{(d)} \in \mathbb{R}^{V^{(d)} \times d}$ and $\mathbf{P}^{(d)} \in \mathbb{R}^{N \times d}$. The output of each embedding layer is a sequence of vectors $\mathbf{h}_1^{(0)}, \mathbf{h}_2^{(0)}, \dots, \mathbf{h}_n^{(0)} \in \mathbb{R}^d$.

- **Multi-head attention:** The input to this layer are a sequence of hidden state vectors from the previous layer $\mathbf{h}_1^{(l-1)}, \mathbf{h}_2^{(l-1)}, \dots, \mathbf{h}_n^{(l-1)} \in \mathbb{R}^d$. We project each of these vectors to query, key and value vectors:

  $$\mathbf{q}_i^{(l)} = \mathbf{W}^{(q)}\mathbf{h}_i^{(l-1)}, \ \ \mathbf{k}_i^{(l)} = \mathbf{W}^{(k)}\mathbf{h}_i^{(l-1)}, \ \ \mathbf{v}_i^{(l)} = \mathbf{W}^{(v)}\mathbf{h}_i^{(l-1)},$$

  where $\mathbf{q}_i^{(l)}, \mathbf{k}_i^{(l)}, \mathbf{v}_i^{(l)} \in \mathbb{R}^d$. For the sake of clarity, we are omitting the layer index (l) from the weight matrices and intermediate variables. In cross-attention, the input to key and value matrices would be the output states of the encoder, whereas the queries would be computed from the hidden states of the decoder.

  We then split these vectors into H separate vectors, where H is the number of attention heads:

  $$\mathbf{q}_i = \begin{bmatrix} \mathbf{q}_{i,1} \\ \mathbf{q}_{i,2} \\ \vdots \\ \mathbf{q}_{i,H} \end{bmatrix}, \quad \mathbf{k}_i = \begin{bmatrix} \mathbf{k}_{i,1} \\ \mathbf{k}_{i,2} \\ \vdots \\ \mathbf{k}_{i,H} \end{bmatrix}, \quad \mathbf{v}_i = \begin{bmatrix} \mathbf{v}_{i,1} \\ \mathbf{v}_{i,2} \\ \vdots \\ \mathbf{v}_{i,H} \end{bmatrix},$$

  where $\mathbf{q}_{i,h}, \mathbf{k}_{i,h}, \mathbf{v}_{i,h} \in \mathbb{R}^{d_H}$ and $d_H = \frac{d}{H}$ is the head dimension. For each head h, we compute a scaled dot product attention:

  $$\mathbf{y}_{i,h} = \sum_{j=1}^{n} \left(\frac{\exp(\mathbf{q}_{i,h} \cdot \mathbf{k}_{j,h} / \sqrt{d_H})}{\sum_{j'=1}^{n} \exp(\mathbf{q}_{i,h} \cdot \mathbf{k}_{j',h} / \sqrt{d_H})} \mathbf{v}_{j,h} \right),$$

  where each $\mathbf{y}_{i,h} \in \mathbb{R}^{d_H}$.

  We need to make sure to avoid attending to pad tokens, which should not affect the model's output. In practice, this is achieved by setting the attention score $\mathbf{q}_i^{(h)} \cdot \mathbf{k}_j^{(h)} / \sqrt{d_H}$ to a very large negative value if there is a pad token at position j. When computing the self-attention in the decoder, we use causal masking to avoid attending to future values:

  $$\mathbf{y}_{i,h} = \sum_{j=1}^{i} \left(\frac{\exp(\mathbf{q}_{i,h} \cdot \mathbf{k}_{j,h} / \sqrt{d_H})}{\sum_{j'=1}^{i} \exp(\mathbf{q}_{i,h} \cdot \mathbf{k}_{j',h} / \sqrt{d_H})} \mathbf{v}_{j,h} \right),$$

  Finally we stack the attention outputs and project each token to obtain a sequence of output vectors $\mathbf{h}_1^{(l)}, \mathbf{h}_2^{(l)}, \dots, \mathbf{h}_n^{(l)} \in \mathbb{R}^d$:

  $$\mathbf{h}_i^{(l)} = \mathbf{W}^{(o)} \begin{bmatrix} \mathbf{y}_{i,1} \\ \mathbf{y}_{i,2} \\ \vdots \\ \mathbf{y}_{i,H} \end{bmatrix}$$

- **Feedforward layers:** *These are already implemented for you in the notebook!* The input to this layer are a sequence of hidden state vectors from the previous layer $\mathbf{h}_1^{(l-1)}, \mathbf{h}_2^{(l-1)}, \dots \mathbf{h}_n^{(l-1)} \in \mathbb{R}^d$. Each feedforward layer learns two feedforward matrices: $\mathbf{W}^{(1)} \in \mathbb{R}^{d_I \times d}$ and $\mathbf{W}^{(2)} \in \mathbb{R}^{d \times d_I}$ (we are omitting the layer index again). These matrices are used to project each input vector to a larger intermediate dimension d_I, apply a ReLU activation and then project back to the original embedding space. Finally, dropout is applied.

  $$\mathbf{h}_i^{(l)} = \text{Dropout}(\mathbf{W}^{(2)}\text{ReLU}(\mathbf{W}^{(1)}\mathbf{h}_i^{(l-1)}))$$

- **Add & Norm:** *These are already implemented for you in the notebook!* After each attention and feedforward block, we add a residual network connection and perform layer normalization. This helps with optimization issues of training deep Transformer networks.

- **Output layer:** We will re-use the token input embeddings and perform next token prediction at each position i in the decoder:

  $$logits_i = \mathbf{E}^{(d)}\mathbf{h}_i^{(L)} \in \mathbb{R}^{V^{(d)}}$$

  Note that in Pytorch, we do not have to compute the softmax when using the cross entropy loss function.

Tips

- Carefully read the function signatures and docstrings for the functions that you need to implement, particularly the type hints and tensor shapes.
- Before you get started on implementing the missing parts of the model implementation, make sure you read the rest of the provided code carefully. This can be useful to understand how each module is going to be used.
- Before starting to train models, make sure you visualize the attention weights to convince yourself that attention masking is working correctly. You are encouraged to implement additional tests and sanity checks for the rest of the code.
- Although this is an extensive coding assignment, we provide detailed documentation for each part you need to implement. The total number of lines that you need to implement can be below 100 lines. You should make use of PyTorch's documentation when needed https://pytorch.org/docs/stable/.

(a) (16 points) Complete the TODO items in the code in order:

1. Complete the tokenization code to produce tokenized datasets.
2. Implement the multi-head attention module. You are not allowed to use `nn.MultiheadAttention` or `nn.functional.scaled_dot_product_attention`. For full credit, avoid using python loops.
3. Before moving on to completing the other sub-modules, complete the sanity check for the multi-head attention. The generated plots should show the correct attention masking patterns. Make sure that they are included in your submitted notebook.
4. Implement the embedding layer, which computes and sums the token embeddings and the positional embeddings.
5. Put all the sub-modules together by implementing the main `EncoderDecoderModel`.

(b) (4 points) Now you should be ready to train an NMT system on the real data. Start the training process using the model that you just completed. Take a look at the hyperameters defined in colab (don't change them!) and observe the training progress in the training log. Make sure to include the training log in your answer. Provide answers at then end of the notebook to the following questions:

(i) What vocabulary size are we using for the source and target language including special tokens?

(ii) Approximately how many source and target tokens are on average contained in a training batch? What proportion of these tokens are `<pad>` tokens on average?

(iii) What is the specific purpose of saving the model parameters in a file `model.pt` throughout training in the code we provide?

You can alter the provided code to obtain the answers, but be careful not to break anything!

(c) (2 points) Load the trained model and evaluate the model on the test set. Report the BLEU score you've obtained on the test set. Manually look at some results and compare them with the gold answers. What do you think of the quality of the translations? Are these grammatical English sentences? Can you identify any common mistakes?

(d) (4 points) Consider the provided beam search method. This implementation is not efficient and performs a lot of repeated computation. Identify the issue and propose how you can fix it. In particular, describe how would you have to change the arguments and return values of your EncoderDecoderModel and its sub-modules?