COS 484: Natural Language Processing **(Due: 03/31/26)**

Assignment #3

Instructor: Karthik Narasimhan

Read all the instructions below carefully before you start working on the assignment and before you make a submission. The course assignment policy is available at https://princeton-nlp.github.io/cos484/. When you're ready to submit, please follow the instructions found here: http://bit.ly/COS_NLP_Submission

- This assignment contains 2 parts, a theoretical and a programming part. The former consists of 2 problems, and the latter has 1, for a total of **3 problems**.
- We *highly* recommended that you typeset your submissions in LaTeX. Use the template provided on the website for your answers. If you have never used LaTeX, you can refer to the short guide here: http://bit.ly/WorkingWithLaTeX. Include your name and NetIDs with your submission. If you wish to submit hand-written answers, you can scan and upload the pdf.
- Assignments must be uploaded to Gradescope by **11:59pm Eastern** on the due date mentioned above.
- As per the late-day policy outlined on the course website, you have 4 late days that you can use any time during the semester, with at most 3 late days per assignment. Once you run out of late days, late submissions will incur a penalty of 10% for each day, up to a maximum of 3 days beyond which submissions will not be accepted.
- All programming problems in this class should be completed in Google Colab using Python. If you would like to get familiar with this environment, you may complete the problems in this introductory Colab notebook (This will not be graded). If you've never worked with Google Colab before, take a look through this introduction guide: Working With Colab. **The answers to the written questions proposed in the programming part should be answered in your Colab notebook.**
- **LLM usage policy**: You **may not** consult a Large Language Model (LLM) when working on the **Theoretical** part of this assignment. For the **Programming** Part only, you may use coding assistants (like GitHub Copilot, Cursor, etc.) for writing code. If you do use such assistants, include the prompts you used at the end of your Colab notebook.

Theoretical Part

Submission Policy: Submit a single PDF for the answers to all questions in this part.

Problem 1: LSTMs vs Transformers (4 + 3 + 3 + 4 = 14 points)

Both LSTMs and Transformers can be thought of a transformation from a sequence of input vectors $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in \mathbb{R}^d$ to a sequence of outputs $\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_n \in \mathbb{R}^d$ (for simplicity, we assume their dimensions are the same). In this problem, we will compare the running time of an LSTM layer with a (single-head) self-attention layer in a Transformer:

- LSTM:

$$\begin{aligned} \mathbf{i}_i &= \sigma(\mathbf{W}^{(i)}\mathbf{h}_{i-1} + \mathbf{U}^{(i)}\mathbf{x}_i) & \mathbf{f}_i &= \sigma(\mathbf{W}^{(f)}\mathbf{h}_{i-1} + \mathbf{U}^{(f)}\mathbf{x}_i) \\ \mathbf{o}_i &= \sigma(\mathbf{W}^{(o)}\mathbf{h}_{i-1} + \mathbf{U}^{(o)}\mathbf{x}_i) & \mathbf{g}_i &= \tanh(\mathbf{W}^{(g)}\mathbf{h}_{i-1} + \mathbf{U}^{(g)}\mathbf{x}_i) \\ \mathbf{c}_i &= \mathbf{f}_i \odot \mathbf{c}_{i-1} + \mathbf{i}_i \odot \mathbf{g}_i & \mathbf{h}_i &= \mathbf{o}_i \odot \tanh(\mathbf{c}_i) \end{aligned}$$

  where $\mathbf{W}^{(i)}, \mathbf{W}^{(f)}, \mathbf{W}^{(o)}, \mathbf{W}^{(g)}, \mathbf{U}^{(i)}, \mathbf{U}^{(f)}, \mathbf{U}^{(o)}, \mathbf{U}^{(g)} \in \mathbb{R}^{d \times d}$ (the bias terms are omitted).

- Self-attention:

$$\begin{aligned} \mathbf{q}_i &= \mathbf{W}^{(q)}\mathbf{x}_i, \;\; \mathbf{k}_i = \mathbf{W}^{(k)}\mathbf{x}_i, \;\; \mathbf{v}_i = \mathbf{W}^{(v)}\mathbf{x}_i, \\ \mathbf{h}_i &= \mathbf{W}^{(o)} \sum_{j=1}^{n} \left(\frac{\exp(\mathbf{q}_i \cdot \mathbf{k}_j / \sqrt{d})}{\sum_{j'=1}^{n} \exp(\mathbf{q}_i \cdot \mathbf{k}_{j'} / \sqrt{d})} \mathbf{v}_j \right) \end{aligned}$$

  where $\mathbf{W}^{(q)}, \mathbf{W}^{(k)}, \mathbf{W}^{(v)}, \mathbf{W}^{(o)} \in \mathbb{R}^{d \times d}$.

(a) (4 points) Compare the running time of the forward pass of these two layers by counting the total number of floating-point multiplication operations in terms of n and d. Use the following assumptions:

1. For attention, assume that $\mathbf{q}_i \cdot \mathbf{k}_j / \sqrt{d}$ is computed for all i and j and cached. This can then be re-used to compute the denominator of the softmax.
2. You can ignore the cost of dividing by $\sqrt{d}$ and dividing by the normalization constant for a softmax.
3. You can ignore the cost of activation functions and exponentials.

(b) (3 points) Suppose that we want to run these two layers on long sequences $n \gg d$ (e.g., $d = 512, n = 4096$), which layer runs faster in theory? In practice, which layer is easier to parallelize? Which layer is better at capturing long-term dependencies? Provide a brief reasoning for each answer.

(c) (3 points) Now consider the total number of learnable parameters in each layer (still ignoring bias terms). Derive the parameter count for both the LSTM layer and the self-attention layer in terms of d. Which layer has more parameters, and what does this imply about the model capacity and risk of overfitting when training data is limited?

(d) (4 points) Suppose we extend the single-head self-attention layer to *multi-head* attention with H heads. Each head uses projection matrices $\mathbf{W}_h^{(q)}, \mathbf{W}_h^{(k)}, \mathbf{W}_h^{(v)} \in \mathbb{R}^{d \times (d/H)}$ and the outputs of all heads are concatenated and projected through $\mathbf{W}^{(o)} \in \mathbb{R}^{d \times d}$. Derive the total number of floating-point multiplications for the multi-head attention layer in terms of n, d, and H (using the same assumptions as part (a)). Is the cost the same as, cheaper, or more expensive than single-head attention?

Problem 2: Attention for Time-Series Data (3 + 3 + 4 + 10 = 20 points)

We are interested in modeling time-series data and would like to use a masked self-attention layer with a single head. The masking pattern in this self-attention layer is causal, i.e. queries cannot attend to keys and values at future time steps, and it is the type of self-attention layer used in a Transformer decoder. We make a small modification to the masked self-attention layer: The layer takes inputs $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in \mathbb{R}^d$, and outputs a sequence of scalars $y_1, y_2, \ldots, y_n \in \mathbb{R}$, via the following implementation:

$$\mathbf{q}_i = \mathbf{W}^{(q)}\mathbf{x}_i, \quad \mathbf{k}_i = \mathbf{W}^{(k)}\mathbf{x}_i, \quad \mathbf{v}_i = \mathbf{W}^{(v)}\mathbf{x}_i,$$

$$A_{i,j} = \begin{cases} \exp(\mathbf{q}_i \cdot \mathbf{k}_j/\tau) / \left(\sum_{j'=1}^{i} \exp(\mathbf{q}_i \cdot \mathbf{k}_{j'}/\tau)\right) & j \le i \\ 0 & j > i \end{cases}$$

$$y_i = \sum_{j=1}^{i} A_{i,j}\mathbf{v}_j$$

where $A_{i,j}$ are the indices of the attention matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, τ is a hyper-parameter that controls the softmax temperature and $\mathbf{W}^{(q)} \in \mathbb{R}^{2 \times d}$ is the query projection matrix, $\mathbf{W}^{(k)} \in \mathbb{R}^{2 \times d}$ is the key projection matrix and $\mathbf{W}^{(v)} \in \mathbb{R}^{1 \times d}$ is the value projection matrix. We will make the following assumptions throughout the question:

1. The input vector at time step i has the following structure:

$$\mathbf{x}_i = \begin{bmatrix} u_i \\ \mathbf{p}_i \end{bmatrix}, \qquad \text{where} \quad \begin{aligned} \mathbf{p}_1 &= [1 \quad 0 \quad 0 \quad ... \quad 0]^T \\ \mathbf{p}_2 &= [0 \quad 1 \quad 0 \quad ... \quad 0]^T \\ \mathbf{p}_3 &= [0 \quad 0 \quad 1 \quad ... \quad 0]^T \\ &\vdots \\ \mathbf{p}_{d-1} &= [0 \quad 0 \quad 0 \quad ... \quad 1]^T \end{aligned}$$

   Here, $u_i \in \mathbb{R}$ is the time-series value at time step i and $\mathbf{p}_i \in \mathbb{R}^{d-1}$ encodes the positional (or temporal) information as a one-hot vector. Therefore, we can only encode sequence up to a maximum length of $N = d - 1$, i.e., the sequence length must be $1 \le n \le N$.

2. The weights of the key projection matrix are defined by:

$$\mathbf{W}^{(k)} = \begin{bmatrix} 0 & \cos\frac{2\pi(1)}{N} & \cos\frac{2\pi(2)}{N} & ... & \cos\frac{2\pi(N-1)}{N} & \cos\frac{2\pi(N)}{N} \\ 0 & \sin\frac{2\pi(1)}{N} & \sin\frac{2\pi(2)}{N} & ... & \sin\frac{2\pi(N-1)}{N} & \sin\frac{2\pi(N)}{N} \end{bmatrix},$$

   where $N = d - 1$ is again the maximum sequence length. This ignores the time-series values and maps the positional information to an $\mathbb{R}^2$ vector. The weights of the value projection matrix are defined by:

$$\mathbf{W}^{(v)} = [1 \quad 0 \quad 0 \quad ... \quad 0 \quad 0].$$

   This selects the information of the time-series values and ignores the positional information.

(a) (3 points) Assume that the maximum sequence length is $N = 8$. Since the key vectors $\mathbf{k}_1, \mathbf{k}_2, ..., \mathbf{k}_n$ only use the positional information of the input sequence, we can compute the key vectors for any valid input sequence which follows our assumptions. Sketch these key vectors on a 2-d plane and label the vectors by their time-step. *(You do not need to show numeric values for these vectors.)*

(b) (3 points) Consider a layer where the weights of the query matrix are all zero:

$$\mathbf{W}^{(q)} = \begin{bmatrix} 0 & 0 & 0 & ... & 0 & 0 \\ 0 & 0 & 0 & ... & 0 & 0 \end{bmatrix},$$

Describe the resulting attention matrix for an input sequence of length n. If the input contains a time-series with values $u_1, u_2, \ldots, u_n$, what would the output of the layer be?

Hint: Remember that the attention mechanism is causal!

(c) (4 points) Describe the behavior of the attention mechanism in the limit of $\tau \to 0$. Pay attention to the special case of "ties", when multiple entries have the same value, i.e. $A_{i,j} = A_{i,j'}$ for $j \neq j'$.

(d) (10 points) We now assume that the layer operates in the limit of $\tau \to 0$. Consider the following time-series operations and, for each, either:

A. Show how the operation can be implemented with the attention layer by finding the corresponding attention matrix $\mathbf{A}$ and a query projection matrix $\mathbf{W}^{(q)}$, or

B. Give a reason why the operation cannot be implemented with the attention layer under our assumptions.

(i) Always select the value at the m'th time step, where m is a hyperparameter and greater than 1:

$$y_i = u_m$$

(ii) A moving average of the current and the previous time-series value:

$$y_i = \begin{cases} u_1 & \text{if } i = 1 \\ \frac{u_i + u_{i-1}}{2} & \text{otherwise} \end{cases}$$

Hint: You need to use the special case of "ties" from your answer in (c).

(iii) An exponentially-weighted moving average of the current and all previous time-series values, defined by:

$$y_i = \left(\sum_{j=1}^{i} (2^{j-i}) u_j \right) / \left(\sum_{j=1}^{i} (2^{j-i}) \right)$$

Programming Part

Submission Policy: The answers to the written questions for this part should be answered in the Colab notebook. **Do not include the answers to this part in the same PDF as the theory part.**

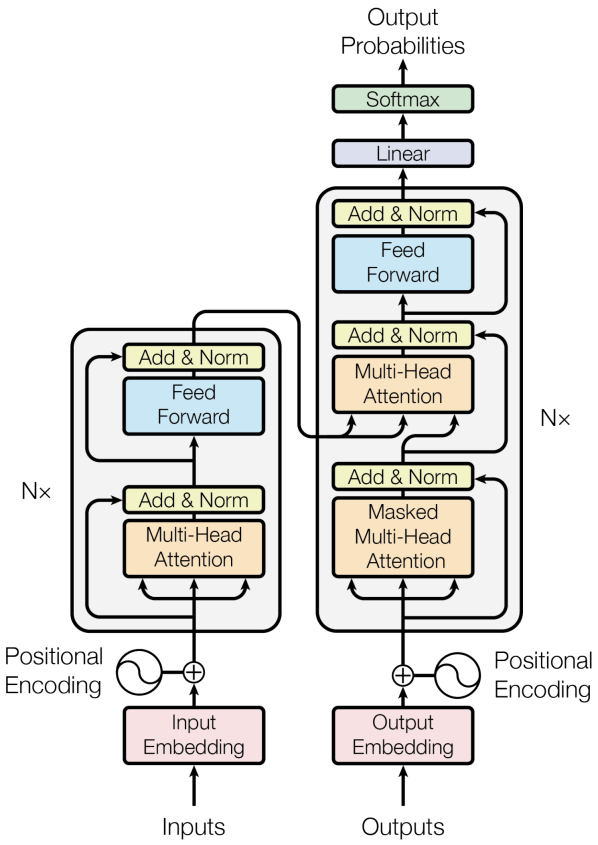
Problem 1: Neural Machine Translation (20 + 2 + 4 + 10 = 36 points)

Machine translation is the task of automatically translating text from one language (the *source*) to another (the *target*). Modern neural machine translation (NMT) systems use an *encoder-decoder* framework: an encoder reads the source sentence and produces a sequence of contextual representations, and a decoder generates the target sentence one token at a time, conditioning on both the encoder's output and the tokens it has generated so far. The decoder attends to the encoder's representations through *cross-attention*, which allows it to focus on relevant parts of the source sentence at each generation step — effectively learning a soft alignment between source and target words. We evaluate translation quality using **BLEU** (Bilingual Evaluation Understudy), a standard metric that measures the overlap of n-grams between the model's output and one or more reference translations. A higher BLEU score indicates closer agreement with the reference.

In this problem, you will implement a sequence-to-sequence model based on the Transformer architecture to build an NMT system that translates from French to English. You will implement the core components of the model (embeddings, multi-head attention, encoder, decoder), train it on a parallel corpus, evaluate it using BLEU, and visualize what the model learns through its attention weights. To get started, open this Colab notebook.

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 you will be implementing is an encoder-decoder Transformer model.



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.

- **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 $\mathbf{h}_1^{(0)}, \mathbf{h}_2^{(0)}, \ldots, \mathbf{h}_n^{(0)} \in \mathbb{R}^d$.

- **Multi-head attention:** The input to this layer is a sequence of hidden state vectors from the previous layer $\mathbf{h}_1^{(l-1)}, \mathbf{h}_2^{(l-1)}, \ldots, \mathbf{h}_n^{(l-1)} \in \mathbb{R}^d$. 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 $\mathbf{h}_1^{(l)}, \mathbf{h}_2^{(l)}, \ldots, \mathbf{h}_n^{(l)} \in \mathbb{R}^d$:

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

- **Feedforward layers:** The input to this layer are a sequence of hidden state vectors from the previous layer $\mathbf{h}_1^{(l-1)}, \mathbf{h}_2^{(l-1)}, \ldots, \mathbf{h}_n^{(l-1)} \in \mathbb{R}^d$. 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:** 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

- 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 a template in the Colab for each part you need to implement. You should make use of PyTorch's documentation when needed https://pytorch.org/docs/stable/. If you get stuck, don't hesitate to leverage AI coding tools to help you debug.

(a) (20 points) Complete the code implementation in order:

1. Complete the data loading and 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 this. For full credit, avoid using python loops.
3. Before moving on to completing the other sub-modules, code up a sanity check for the multi-head attention (see colab for exact details). 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. Implement the transformer block, which combines the multi-head attention and embedding layers either for the encoder or the decoder.
6. Put all the sub-modules together by implementing the main `EncoderDecoderModel`.

(b) (2 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 hyperparameters 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 the end of the notebook to the following questions:

(i) What vocabulary size are you 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?

(c) (4 points) Load the trained model and evaluate the model on the test set. Report the corpus-level BLEU score (using `nltk.translate.bleu_score.corpus_bleu`) 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) (10 points) Extract and visualize the **cross-attention** weights from the **last decoder layer**, averaged across all attention heads. This should produce a single 2-D heatmap per example, where the rows correspond to the generated English (target) tokens and the columns correspond to the French (source) tokens.

Produce attention heatmaps for the following two French sentences:

1. `Le chat noir est assis sur le tapis rouge.`

2. `Elle n'a pas encore pris une décision importante concernant son avenir.`

For each example, answer the following questions:

(i) Does the attention pattern resemble a word-level alignment between the source and target sentences? Where does it deviate from a diagonal pattern, and why might that be the case? *(Hint: Consider word order differences between French and English.)*

(ii) Does the attention heatmap reveal anything about how the model handles the translation? For instance, are there tokens where attention is diffuse or concentrated, and can you relate this to the structure of the sentence?