Homework 4 Part 1

Language Modeling using Causal Transformer Decoder

11-785: Introduction to Deep Learning (Fall 2024)

OUT: **November 9, 2024**
DUE: **December 6, 2024, 11:59 PM**
Early Bonus: **November 22, 2024, 11:59 PM**

Start Here

- **Collaboration policy:**
  - You are expected to comply with the University Policy on Academic Integrity and Plagiarism.
  - You are allowed to help your friends debug
  - You are allowed to look at your friends code
  - You are not allowed to type code for your friend
  - You are not allowed to look at your friends code while typing your solution
  - You are not allowed to copy and paste solutions off the internet
  - You are not allowed to import pre-built or pre-trained models
  - You can share ideas but not code, you must submit your own code. All submitted code will be compared against all code submitted this semester and in previous semesters using MOSS.

  We encourage you to meet regularly with your study group to discuss and work on the homework. You will not only learn more, you will also be more efficient that way. However, as noted above, the actual code used to obtain the final submission must be entirely your own.

- **Overview/TL;DR:**
  - **Part 1**: All of the problems in Part 1 will be graded on Autolab. Download the starter code from Autolab.
  - **Self-Attention**: Complete the forward and backward functions of the Attention class in the attention.py file.
  - **Next token prediction**: Complete the function **predict** in class **LanguageModel** in the Jupyter Notebook.
  - **Generation of Sequence**: Complete the function **generate** in the class **LanguageModel** in the Jupyter Notebook.
  - **Beam Search [Optional]**: Use beam search on prediction and generation tasks.

Homework Objectives

- **If you complete this homework successfully, you would ideally have learned**
  - Learn how to implement self-attention from scratch.
  - Learn how to train a causal transformer decoder model for generating text.
  - Understand the workings of Language Modeling, and train a model to generate next-token predictions as well as entire sequences.
  - Learn about positional encoding and how it is used to incorporate sequence order in transformer models.

- Gain knowledge of multihead attention and its role in capturing multiple relationships between different parts of the input.
- Explore layer normalization, including both pre-norm and post-norm techniques, and understand their impact on training stability and performance.

IMPORTANT NOTE

The writeup is about modelling written language. Since we may consider either characters or words to be the units of written language, we generally talk in terms of "tokens", where a token may be a word or character, depending on what we model. The examples provided are in terms of characters which we will use in HW4P1 and HW4P2.

In addition to this, you will also learn how to implement the forward and backward functions of self-attention. You do not have to use this in the language model, the purpose of implementing this is to understand in-depth how attention works, which will be beneficial in HW4P2 when you will create an encoder decoder transformer based model.

The writeup also includes several listings of pseudocode. These are only intended to present the logic behind the described approaches. The pseudocode is *not* in python form, and may not even be directly translatable to python. It is merely a didactic tool; you will have to determine how to implement the logic in python.

And last, but definitely not the least: The writeup is long and detailed and explains the underlying concepts. It includes explanations and pseudocode that are not directly related to the homework, but are intended to be instructive *and assist in understanding the homework.* The actual homework problems to solve are listed in two of the sections (or subsections, or subsubsections, we aren't actually sure which). Please read the document carefully and identify these sections.

Contents

1 Introduction

Key new concepts: Language modeling, text generation, regularization techniques for Transformers, self-attention

Mandatory implementation: Attention class, transformer-based Language model, text prediction and text generation using your trained model, Greedy Decoding.

Restrictions: You may not use any data besides that provided as part of this homework; You may not use the validation data for training; You have to use at least greedy decoding (and optionally beam search) for both text prediction and text generation.

1.1 Overview of Self-Attention `[attention.py]`

At its core, attention refers to the ability of a system to focus on specific elements within a larger dataset, assigning varying degrees of importance to different parts of the input. This mechanism enables models to selectively process and weight information, facilitating more efficient and context-aware processing of complex data, be it in natural language understanding, image recognition, or numerous other machine learning tasks.

At its heart, attention typically involves three essential components: keys, queries, and values. This part is similar to your other Homework Part 1s, where you implement the code for the Attention class in a .py file. Attention can be applied between two different sets of sequences(for example, encoder hidden states and decoder hidden states), which is called cross attention. Alternatively, it can be applied within the same set of inputs, known as self-attention. Here, you will be implementing self-attention.

Keys: Keys represent elements in the input data that the model needs to pay attention to. These keys are derived from the input and encode specific information about the elements. In the context of natural language processing, keys might represent words or tokens in a sentence, or in computer vision, they could correspond to spatial locations in an image.

Queries: Queries are another set of representations derived from the input data. They are used to seek information from the keys. Queries encode what the model is specifically looking for in the input. For instance, in language translation, a query might represent a word in the target language, and the model uses this query to find relevant information in the source language.

Values: Values are yet another set of representations derived from the input data, and they contain the actual information the model is interested in. Values are associated with the keys, and they can be thought of as the content at those particular locations. When queries and keys match closely, the values associated with those keys are given more importance in the final output.

In the following section, equations for the forward and backward functions of self-attention are provided in detail, which you will have to implement. We will be using the scaled dot-product attention method for computing attention. (You can find additional methods for computing attention in HW4P2.)

1.1.1 Structure of attention.py

Your task is to implement the Attention class in file `attention.py`:

- Class attributes:
  - Variables stored during forward-propagation to compute derivatives during back-propagation: key `K`, query `Q`, value `V`, raw weights `A_w`, attention weights `A_sig`.
  - Variables stored during backward-propagation `dLdK, dLdQ, dLdV, dLdWk, dLdWq, dLdWv`.
- Class methods:
  - `__init__`: Store the key weights, value weights and query weights as model parameters.
  - `forward`: forward method takes in one inputs X and computes the attention context.
  - `backward`: backward method takes in input `dLdXnew`, and calculates and stores `dLdK, dLdQ, dLdV, dLdWk, dLdWq, dLdWv`, which are used to improve the model. It returns `dLdX`, how changes in the inputs affect loss to enable downstream computation.

Please consider the following class structure:

```
class Attention:

    def __init__(self, weights_keys, weights_queries, weights_values):
        self.W_k    = # TODO
        self.W_q    = # TODO
        self.W_v    = # TODO

    def forward(self, X):

        self.X     = X

        self.Q     = # TODO
        self.K     = # TODO
        self.V     = # TODO

        self.A_w   = # TODO
        self.A_sig = # TODO

        X_new  = # TODO

        return X_new

    def backward(self, dLdXnew):
        self.dLdK      = # TODO
        self.dLdQ      = # TODO
        self.dLdV      = # TODO

        self.dLdWq     = # TODO
        self.dLdWk     = # TODO
        self.dLdWv     = # TODO

        dLdX           = # TODO

        return dLdX
```

Before we move to the math of attention, please understand some notation. B is the batch size, T is sequence length of the input, and the representation of each time step is D-dimensional. The keys and queries obtained from the input have to be of the same dimensionality which is denoted by D_k. The values can have a different dimensionality denoted by D_v. Now that you know the notation, you should understand the equations to implement. To understand the shapes of different matrices and tensors involved, refer to Table 1.

1.1.2 Self-attention Forward Equations

$$K = X \cdot W_k \qquad \in \mathbb{R}^{B \times T \times D_k} \tag{1}$$

$$V = X \cdot W_v \qquad \in \mathbb{R}^{B \times T \times D_v} \tag{2}$$

$$Q = X \cdot W_q \qquad \in \mathbb{R}^{B \times T \times D_k} \tag{3}$$

$$A_w = Q \cdot K^T \qquad \in \mathbb{R}^{B \times T \times T} \tag{4}$$

$$A_\sigma = \sigma(\frac{A_w}{\sqrt{D_k}}) \qquad \in \mathbb{R}^{B \times T \times T} \tag{5}$$

$$X_n = A_\sigma \cdot V \qquad \in \mathbb{R}^{B \times T \times D_v} \tag{6}$$

Table 1: Attention Components

Code Name	**Math**	**Type**	**Shape**	
`W_q`	W_q	matrix	$D \times D_k$	Weight matrix of Query
`W_k`	W_k	matrix	$D \times D_k$	Weight matrix of Key
`W_v`	W_v	matrix	$D \times D_v$	Weight matrix of Value
`X`	X	matrix	$B \times T \times D$	Input to attention
`K`	K	matrix	$B \times T \times D_k$	Key
`Q`	Q	matrix	$B \times T \times D_k$	Query
`V`	V	matrix	$B \times T \times D_v$	Value
`A_w`	A_w	matrix	$B \times T \times T$	Raw attention weights
`A_sig`	A_σ	matrix	$B \times T \times T$	Softmax of raw attention weights
`X_new`	X_n	matrix	$B \times T \times D_v$	Final attention context
`dLdX_new`	$\partial L/\partial X_{new}$	matrix	$B \times T \times D_v$	Gradient of Loss wrt Attention Context
`dLdA_sig`	$\partial L/\partial A_{sig}$	matrix	$B \times T \times T$	Gradient of Loss wrt Attention Weights
`dLdV`	$\partial L/\partial V$	matrix	$B \times T \times D_v$	Gradient of Loss wrt Values
`dLdA_w`	$\partial L/\partial A_w$	matrix	$B \times T \times T$	Gradient of Loss wrt Raw weights
`dLdK`	$\partial L/\partial K$	matrix	$B \times T \times D_k$	Gradient of Loss wrt Key
`dLdQ`	$\partial L/\partial Q$	matrix	$B \times T \times D_k$	Gradient of Loss wrt Query
`dLdW_q`	$\partial L/\partial W_q$	matrix	$D \times D_k$	Gradient of Loss wrt Query weight
`dLdW_v`	$\partial L/\partial W_v$	matrix	$D \times D_v$	Gradient of Loss wrt Value weight
`dLdW_k`	$\partial L/\partial W_k$	matrix	$D \times D_k$	Gradient of Loss wrt Key weight
`dLdX`	$\partial L/\partial X$	matrix	$B \times T \times D$	Gradient of Loss wrt Input

Remember to think about the shapes of the matrices in each of the above equations and permute the dimensions accordingly when you want to perform a transpose. σ symbol refers to the softmax function. **You can use torch.bmm** [1] **if and when required.**

1.1.3 Self-attention Backward Equations

1. **Derivatives wrt attention weights (raw and normalized):**

$$\frac{\partial L}{\partial A_\sigma} = \left(\frac{\partial L}{\partial X_n}\right) \cdot V^T \qquad \in \mathbb{R}^{B\times T\times T} \tag{7}$$

$$\frac{\partial L}{\partial A_w} = \frac{1}{\sqrt{D_k}} \cdot \sigma^{'}(\frac{\partial L}{\partial A_\sigma}) \qquad \in \mathbb{R}^{B\times T\times T} \tag{8}$$

2. **Derivatives wrt keys, queries, and values:**

$$\frac{\partial L}{\partial V} = A_\sigma^T \cdot \left(\frac{\partial L}{\partial X_n}\right) \qquad \in \mathbb{R}^{B\times T\times D_v} \tag{9}$$

$$\frac{\partial L}{\partial K} = \left(\frac{\partial L}{\partial A_w}\right)^T \cdot Q \qquad \in \mathbb{R}^{B\times T\times D_k} \tag{10}$$

$$\frac{\partial L}{\partial Q} = \left(\frac{\partial L}{\partial A_w}\right) \cdot K \qquad \in \mathbb{R}^{B\times T\times D_k} \tag{11}$$

3. **Derivatives wrt weight matrices:** Notice that all inputs in the batch influence their corresponding keys, queries, and values through the same respective matrices. This is why derivatives from each input in the batch have to be added up. Think about this in the influence diagram framework.

[1] https://pytorch.org/docs/stable/generated/torch.bmm.html

$$\frac{\partial L}{\partial W_q} = \sum_{i=1}^{B} X^T \cdot \left(\frac{\partial L}{\partial Q}\right) \qquad \in \mathbb{R}^{D_k \times D} \tag{12}$$

$$\frac{\partial L}{\partial W_v} = \sum_{i=1}^{B} X^T \cdot \left(\frac{\partial L}{\partial V}\right) \qquad \in \mathbb{R}^{D_v \times D} \tag{13}$$

$$\frac{\partial L}{\partial W_k} = \sum_{i=1}^{B} X^T \cdot \left(\frac{\partial L}{\partial K}\right) \qquad \in \mathbb{R}^{D_k \times D} \tag{14}$$

4. **Derivative wrt input:**

$$\frac{\partial L}{\partial X} = \left(\frac{\partial L}{\partial V}\right) \cdot W_v^T + \left(\frac{\partial L}{\partial K}\right) \cdot W_k^T + \left(\frac{\partial L}{\partial Q}\right) \cdot W_q^T \qquad \in \mathbb{R}^{B \times T \times D} \tag{15}$$

Again, consider the shapes for each of the above equations. For batched inputs, you can use torch.bmm if needed, and permute when taking the transpose. To compute the gradients of the three weight matrices, sum along the batch dimension as shown in the equations

1.2 Overview of Language Modeling

1.2.1 Structure of a Language Model

A decoder-only language model leverages the transformer architecture, which is highly effective for autoregressive language modeling. The structure primarily focuses on generating sequences based on previously seen tokens, making it ideal for tasks such as text generation and next-token prediction. One of the key features of this architecture is its use of **multihead attention**, which enhances the model's ability to capture various aspects of the input sequence.

Input Embedding Layer Each token in the input sequence is first converted into a dense vector representation through an embedding layer. This transformation allows the model to learn and represent complex semantic relationships between tokens.

Positional Encoding Transformers lack an inherent understanding of token order, so positional encodings are added to the input embeddings. These encodings inform the model about the sequential nature of the data and help differentiate between tokens based on their positions.

Multihead Attention Mechanism Multihead attention extends the standard self-attention mechanism by allowing the model to project the input into multiple attention heads. Each head independently performs the attention operation, capturing different relationships in the input data. The outputs from all heads are then concatenated and projected into a single representation. This process helps the model capture diverse and complementary features across the sequence.

Feed-Forward Network After the multihead attention block, the output is processed by a feed-forward neural network comprising two linear layers separated by a non-linear activation function (e.g., ReLU). This step enriches the representation by adding more complexity to the learned features.

Layer Normalization and Residual Connections To stabilize training and facilitate deeper networks, each multihead attention and feed-forward block is followed by layer normalization and residual connections. These components ensure that gradients flow effectively through the network, aiding in faster convergence and preventing vanishing gradient problems.

Output Layer The final output from the transformer blocks is passed through a linear layer, mapping it to the size of the vocabulary. The softmax function then produces a probability distribution over possible next tokens, enabling the model to predict the next token in the sequence.

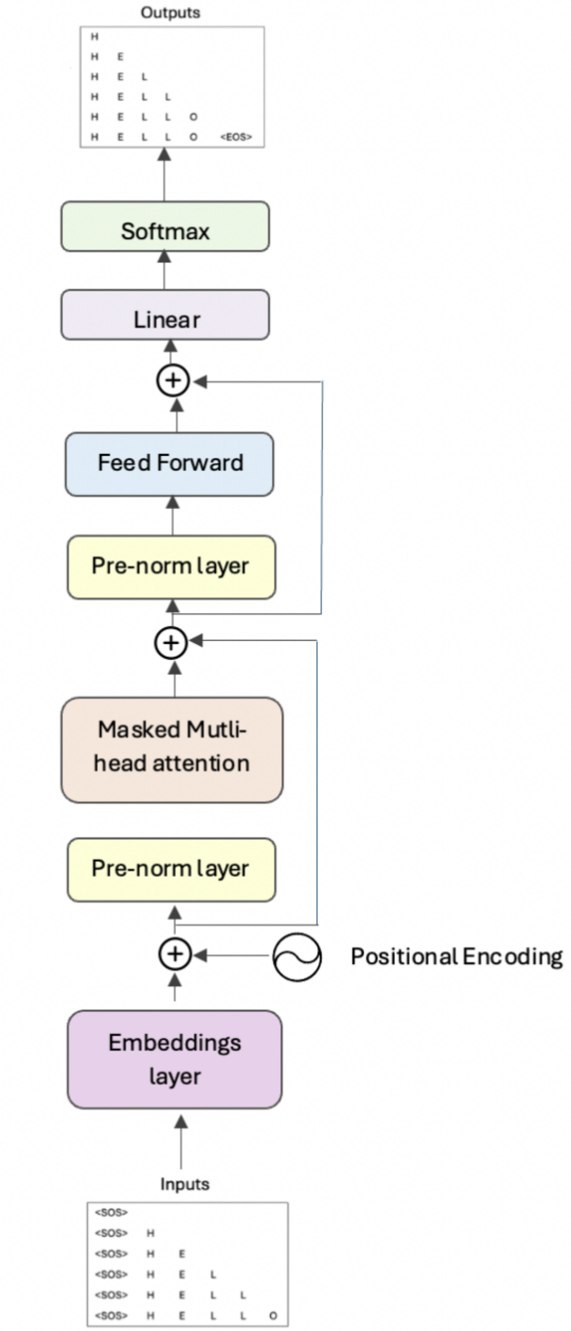

Figure 1: Structure of a Decoder-Only Language Model

1.2.2 Training the Language Model

To train the neural language model, we employ **Maximum Likelihood Estimation (MLE)**, which is a general method for estimating a parametric probability distribution from training data. The goal is to find the parameter θ that maximizes the likelihood of the observed training data.

Let $S = \{S_1, S_2, \ldots, S_K\}$ be the set of training sequences, where each sequence S_i consists of tokens and is represented as $S_i = (\texttt{<sos>}, T_{i,1}, T_{i,2}, \ldots, T_{i,l_i}, \texttt{<eos>})$, where l_i is the length of the sequence. The MLE objective is to maximize the total probability of the training data:

$$\theta = \arg\max_{\theta} \prod_{i=1}^{K} P(S_i; \theta)$$

Since computing the product of probabilities can be numerically unstable, we typically work with the log-probability, converting the product into a sum:

$$\theta = \arg\max_{\theta} \sum_{i=1}^{K} \log P(S_i; \theta)$$

This can be rewritten in terms of the **Negative Log-Likelihood (NLL) Loss**:

$$\text{Loss}(\theta) = -\sum_{i=1}^{K} \log P(S_i; \theta)$$

where the probability of each sequence S_i is calculated by breaking it down into a sequence of conditional probabilities for each token, based on the tokens that came before it:

$$P(S_i; \theta) = P(T_{i,1} | \texttt{<sos>}; \theta) P(T_{i,2} | \texttt{<sos>}, T_{i,1}; \theta) \cdots P(\texttt{<eos>} | \texttt{<sos>}, T_{i,1}, \ldots, T_{i,l_i}; \theta)$$

The task is to train the model to minimize this NLL loss by adjusting θ to increase the probability assigned to the observed sequences.

Note: Although the inference process of the language model is autoregressive, the training process is not. During training, the model does not generate tokens one by one; instead, it uses *teacher forcing* by taking in the entire input sequence at once and predicting each token based on its preceding context. This approach allows for *parallel processing*, where the model simultaneously learns to predict each token in the sequence, making training more efficient than the sequential, step-by-step generation used during inference.

1.2.3 Evaluating your language model

Evaluation consists of two main tasks:

1. **Next-Token Prediction:** In this task, the language model is tested on its ability to predict the probability distribution of the next token given a partial sequence of text. This is a fundamental evaluation method, as it directly assesses the model's understanding of language patterns and contextual cues. For example, in the sentence "The cat sat on the," the model should assign a higher probability to words like "mat" or "couch" than to unrelated words. The performance in next-token prediction is typically evaluated using metrics like NLL metioned above, which quantifies how well the model assigns high probabilities to the correct next token. The lower the NLL, the better the model's performance, as it indicates a stronger ability to predict the correct token in context.

2. **Sequence Completion:** In this task, the model generates tokens to complete an initial partial sequence, such as `<sos> Today is a`. This task tests the model's ability to produce coherent, contextually appropriate, and grammatically correct text that aligns with the style and topic of the initial sequence. Sequence completion is often evaluated both qualitatively, by examining the relevance and fluency of the generated text, and quantitatively, using metrics like **perplexity**. Perplexity provides a measure of the model's uncertainty in its predictions, with lower perplexity values indicating more confident and likely coherent completions. Effective sequence completion demonstrates that the model can capture long-range dependencies and maintain context over extended sequences, which is crucial for applications like text generation, dialogue systems, and language translation.

1.2.4 A word on Tokenizers

It is important to **tokenize** our input in language modeling. Models do not understand natural language as they are, so we need to find a way to convert them to numerical values. A **tokenizer** converts raw text into a sequence of unique **token_id**s with some pre-determined mapping. For example, we may use a tokenizer to split the sentence "Hello world!" into smaller parts, like individual words or subwords, assigning each a unique identifier or **token_id** based on a pre-defined vocabulary. This vocabulary is built from a large corpus of text and maps words, subwords, or characters to specific numerical IDs.

There are different types of tokenization strategies, such as:

- **Word-level tokenization:** Each unique word in the vocabulary is assigned a token_id. However, this can lead to an excessively large vocabulary, as each variation (like plurals or different verb forms) gets a unique ID.
- **Subword tokenization (e.g., Byte-Pair Encoding or WordPiece):** Words are split into smaller, reusable subword units. This approach allows for a compact vocabulary that still captures uncommon or rare words by representing them as a combination of subwords.
- **Character-level tokenization:** Each character in the language is treated as a token, resulting in a small vocabulary but longer token sequences for each sentence.

Selecting an appropriate tokenization strategy is crucial, as it affects the model's vocabulary size, memory efficiency, and handling of rare or out-of-vocabulary words.

For this homework, we provide you with three options: char-level tokenizer, tokenizer with 1,000 vocab size, and tokenizer with 10,000 vocab size. You can experiment with different tokenizers to see which performs better on the task. Note: although we are using the pre-trained decoder from part 1 in part 2, a different tokenizer may work better, as the tasks are not the same.

2 Dataset

For this homework we will be training a Causal Transformer Decoder on the transcriptions from the LibriSpeech train-clean-100 dataset. This dataset is specifically chosen to align with our objectives for training a Transformer-based Automatic Speech Recognition (ASR) model in part 2 of this assignment.

The dataset for part 1 contains transcriptions(text) from the orignal speech-to-text dataset. These files represent spoken utterances transcribed into natrual language. In this homework, you will explore different strategies to tokenize the raw text and compare their performances.

As already mentioned, we give you three options for tokenizing the text: char, 1k-vocab, and 10k-vocab.

For a char-level tokenization, the tokenized result is like the following:

`<sos> W H O H A D   U N T I L   N O W   L I S T E N E D   I N   S I L E N C E <eos>`

For a 1k-vocab tokenization, the tokenized result might look like this:

`<sos> WHO H AD UN TIL NOW LIST EN ED IN SIL EN CE <eos>`

For a 10k-vocab tokenization, the tokenized result could include less granular tokens, as shown below:

`<sos> WHO HAD UNTIL NOW LISTEN ED IN SILEN CE <eos>`

In addition to the standard tokens, the tokenizer also has the following special tokens:

- `<sos>`: Start of Sequence - Indicates the beginning of a text sequence.
- `<eos>`: End of Sequence - Marks the end of a text sequence.
- `<pad>`: Padding - Used to equalize the length of sequences for batch processing.

The fixtures folder will be used for evaluation. It contains pre-processed data files specifically designed for validation and testing. We provide validation and test files for all three tokenizer options. The pre-implemented test and validation code will choose the right test and validation files for you, so you don't need to manage these files yourself. Refer to Section 3 for more details about the testing procedure.

2.1 DataLoader and Data Processing

As mentioned in section 2, the train files for the language modeling part of this assignment consist of 28,539 .npy files, each containing transcriptions formatted as sequences of character tokens from the LibriSpeech train-clean-100 dataset. These character sequences are used by our language model to learn the distribution of spoken language as character sequences, which is useful for part 2 of this homework.

The dataloader prepares the input for the causal decoder transformer by shifting the transcriptions. Each decoder input sequence starts with a <SOS> token to signal the beginning and is used to process the subsequent characters. Conversely, the target sequence for comparison is shifted in the opposite direction to include the <EOS> token at the end, indicating the end of a sentence. This setup represents a full teacher forcing strategy, where the actual target sequences are always provided to the decoder during training. The decoder constructs the autoregressive matrix using an attention mechanism and a triangular mask to prevent future characters from influencing the prediction of the current character. The dataloaders will supply you with the shifted target sequence, e.g., “<SOS> H E L L O,” and the corresponding golden target, e.g., “H E L L O <EOS>.”.

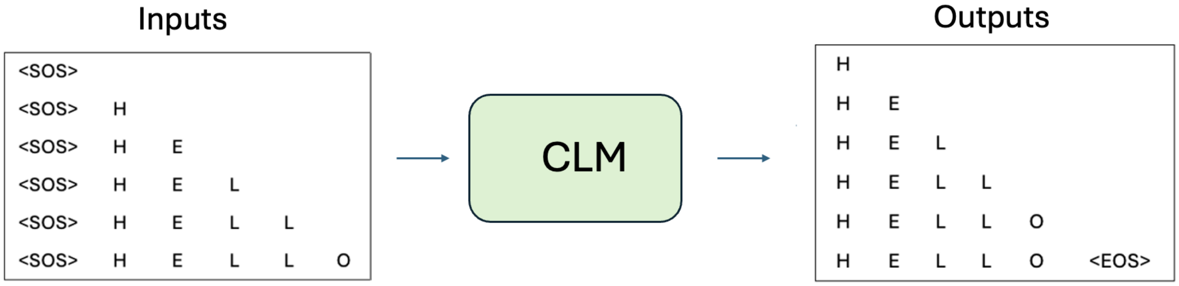

Figure 2: The shifted input sequence, beginning with a start-of-sentence token (<SOS>) and excluding the end-of-sentence token (<EOS>), is used by the decoder to predict the subsequent token in the sequence. The target sequence is the true next token the model aims to predict at each timestep, including the <EOS>token but excluding the initial <SOS>.

3 Testing and Submission

In the handout you will find a `attention.py` file and a starter Jupyter notebook `hw4p1.ipynb` in the hw4 folder. You have to fill in these 2 files to complete the homework.

3.1 Self-attention

For the self-attention part, there are local tests in `hw4/hw4p1_autograder.py` that you can run using `python hw4/hw4p1_autograder.py`. However, final grade of the self-attention portion will be determined by some hidden tests on Autolab, like previous HW Part 1s.

3.2 Language Modeling - the Notebook

Within the starter Jupyter Notebook, `hw4/hw4p1.ipynb`, there are `TODO` sections that you need to complete.

Every time you run training, the notebook creates a new experiment folder under `experiments/` with a `run_id` (which is CPU clock time for uniqueness). All of your model weights and predictions will be saved there.

The notebook trains the model, prints the Negative Log Likelihood (NLL) on the prediction validation set and creates the generation and prediction files on the test dataset.

3.3 Language Modeling - Generation

For the generation task, the notebook includes code specifically designed to generate and save continuations for a test set comprising incomplete sequences. Additionally, there is a dedicated code cell for evaluating the perplexity of your generated texts on the test dataset. A perplexity lower than 5 indicates that you have correctly implemented everything in the language model.

3.4 Create the handin for submission

Once you have completed `attention.py`, trained a model and obtained enough generation test perplexity, you are ready to make a submission. Before that, make sure that the completed notebook is present in the hw4 folder of the handout directory.

You can use the following command from the hw4 directory under handout to generate the submission file.

tar -cvf handin.tar attention.py test_perplexity.txt hw4p1.ipynb

A correct implementation of `attention.py` is worth 40 points, and a completed notebook with test perplexity lower than 5 is worth 60 points.

You will get only 10 submissions for this homework. So ensure that you submit the handin you are confident will cross the cut-offs mentioned in the writeup.

3.5 Some final words

Our tests for the language model are not overly strict, so you can work your way to a performance that is sufficient to pass Autolab by using only a subset of methods.

While the prediction task is evaluated on Autolab, we ask you to submit the final metric for the generation task. Please be honest in reporting this value. Use the same runid and epoch number that for handin creation and for evaluating generations. We will randomly check some submission using the same evaluation code and a big difference in perplexity will result in an AIV.

Warning: The classes provided for training your model are given to help you organize your training code. You shouldn't need to change the rest of the notebook, as these classes should run the training, save models/predictions and also generate plots. If you do choose to diverge from our given code (maybe implement early stopping for example), be careful.

Good luck !