Homework 4 Part 1

Language Modeling using Causal Transformer Decoders

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

OUT: **November 6, 2024**
DUE: **December 2, 2024, 11:59 PM**
Early Bonus: **November 16, 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.
  - **Sequence 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.

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.

Contents

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.

1 Introduction

Key new concepts: Language modeling, text generation, Decoder-Only Transformer Models, self-attention

Mandatory implementation: Attention class, Decoder-Only Transformer 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 either be between 2 different sets of sequences (for example encoder hidden states and decoder hidden states) which is called cross attention. Or it can be within the same set of inputs which is called self-attention. You will be implementing self-attention here.

Keys: Keys are representations of the 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 other different methods to compute 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):**

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

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

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 think about the shapes for each of the above equations. For batched inputs you can use torch.bmm if required. Use permute where necessary, when you have to take the transpose. You will need to sum along the batch dimension for the gradient of the three weight matrices as provided in the equations.

1.2 Overview of Language Modeling with a Decoder-Only Transformer

In this section, you will learn how to use a decoder-only Transformer model to perform language modeling and generate text. Language modeling involves estimating the probability distribution of sentences (or more precisely, token sequences) in a language. This statistical approach allows us to model language by predicting the likelihood of a given sequence of tokens.

Language modelling is, quite literally, the task of modelling language. From a statistical point of view, which is the one we will take, it is the problem of accurately estimating the probability distribution of sentences (or, more precisely, token sequences) in the language[2].

At its core it is no different from, say, finding the probability distribution of a loaded dice, except that unlike the usual dice with only six sides, this dice has infinite sides, one corresponding to every token sequence from the language. This little difference is what complicates matters – unlike the usual dice, where you can store the entire probability distribution in a table with six entries, the distribution for language cannot be stored in a table – we would need an infinite-sized table for it. Instead, we will learn a model for the distribution – specifically, a neural network model. We refer to such models for the distribution of word sequences in a language as a *language model*.

Here are some things you would be able to do if you knew the entire probability distribution of a discrete random variable:

1. Determine the probability of any given outcome;
2. *Draw* samples from the distribution;
3. Rank order outcomes by probability to identify the most likely or the n^{th} most likely outcome;
4. Compute various expectations and other statistics.

A model-based probability distribution enables 1 and 2 above (we can use it to compute the probability of any given token sequence; we can also use it to *draw* samples from the distribution), however 3 and 4 are infeasible[3].

In this homework we will learn how to *train* a neural-network language model from data. We will also learn how to use it to *compute the probabilities* of specified token sequence, and to *draw* (i.e. generate) token sequences from the language.

[2]Language models may be estimated over *any* type of tokens, e.g. words, characters, byte-pairs etc. In this homework we will focus on word-based LMs. However, to remain agnostic to the type of unit (character, word, byte-pair etc) whose sequences are being modelled, we will generically refer to the symbols as *tokens*.

[3]In order to rank order outcomes by probability, or to compute expectations, we must consider *all* infinite possible outcomes, which is generally infeasible.

1.2.1 A note on beginnings and endings

cLanguage models model the distribution of token sequences. A token sequence is a *complete* string of tokens that has a definitive beginning and an end. For instance, consider the following word sequence (where the tokens are words): "*four score and*". Is this a *complete* word sequence, or is it merely a part of a longer sentence? From its appearance, it looks incomplete - as it is. This sequence is the first three words of the first sentence of Abraham Lincoln's famous Gettysburg address : "Four score and seven years ago our fathers brought forth on this continent, a new nation, conceived in Liberty, and dedicated to the proposition that all men are created equal." When modelling language, we would, in fact, be attempting to model the statistical properties of the *entire* underlying sentence, not merely some arbitrarily chosen section (such as the first three words) of it. Thus, here, "Four score and" is *not* a complete sequence; it is merely a portion of a much longer, complete sequence of words.

On the other hand, it is also entirely likely that somewhere, somewhen, a novice actor playing the part of Lincoln went on stage for the first time and held forth "Four score and", got nervous, and quit and went home. Now the *complete* sentence was merely this short sequence of words.

How do we distinguish between the two? In one case, the three words did *not* constitute the entire word sequence, while in the other case they did.

We do this through the use of *start of sequence* and *end of sequence* markers. A *start of sequence* marker, often represented as <sos> indicates that a token sequence has just begun. The symbol immediately following the <sos> tag is, in fact, the first token in the sequence; the <sos> marker is merely the indicator of the start of a new sequence. Similarly, the *end of sequence* marker, often represented as <eos>, indicates the *end* of a complete sequence. Again, the <eos> tag is not the actual final token in the sequence; it is only a marker that indicates the end of a complete sequence.

Using these tags Abraham Lincoln's complete sentence would be:

```
<sos> Four score and seven years ago our fathers brought forth on this continent, a new nation,
conceived in Liberty, and dedicated to the proposition that all men are created equal.<eos>
```

Here, the section "Four score and" is clearly not the full sequence, but is a section of a longer sequence. This is also identifiable from visual inspection – "Four score and" by itself is missing the <sos> and <eos> tags, and so cannot be complete.

On the other hand, our nervous actor's complete monologue consisted of

```
<sos> Four score and <eos>
```

That was the entire sequence as indicated by the <sos> and <eos> tags.

As a matter of convenience, it is sometimes simpler to just have the same tag to indicate both the start and end of sequences. With this standard, using the symbol <eos> to tag both, our complete sequences would be.

```
<eos> Four score and seven years ago our fathers brought forth on this continent, a new nation,
conceived in Liberty, and dedicated to the proposition that all men are created equal.<eos>
<eos> Four score and <eos>
```

From the perspective of language modelling, this means that we must explicitly include the (<sos> and) <eos> tags to our sequences:

- We must explicitly include <sos> and <eos> to the neural network's vocabulary. So, for instance, if we are building a language model over characters, in addition to the alphabet (a-z and A-Z) and punctuation marks and spaces ('.', ',', ' ', etc. which are all part of your text), our vocabulary must be expanded to include <sos> and <eos> as well, so now our vocabulary would be: {a-z , A-Z , ',' , ';' , ',' , ' ' , <sos>, <eos>}.
- When *training* the LM we must include the tags in our training sequences. So, for instance, given a **''Four score and''** as a complete sequence (the product of our nervous actor), we must actually represent it as **''<sos> Four score and <eos>''**.
- When we use the LM to *compute the probability* of a (complete) token sequence, we must ensure it includes

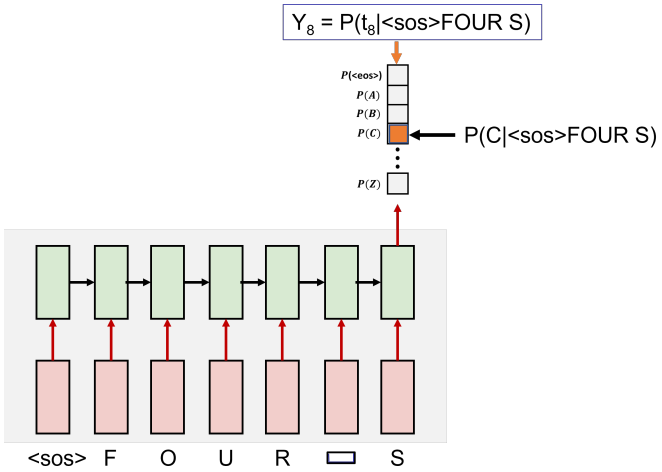

Figure 1: Given the length 7 input "<sos>FOUR S" (including the space), the RNN outputs P(T|<sos>FOUR S), the probability distribution for the eighth character in the sequence given the input. P(C|<sos>FOUR S), the actual probability of having a "C" in the 8th position must be read from this distribution, as shown by the highlighted red box. The RNN figure uses the same pictorial representation as HW3P2.

the tags. So, if we were to want to compute the probability of **''Four score and''** as a complete sequence, this must be converted to **''<sos> Four score and <eos>''**, and the probability of the longer five-token sequence must be computed.

- When we *generate* sequences using the LM, the generation can only be considered complete when the terminal <eos> marker is generated. So, **''<sos> Four score and''** will not be considered a complete generated sequence, but **''<sos> Four score and <eos>''** will be.

1.2.2 Computing the probability of a complete token sequence

Let us now consider how we might *compute the probability* of a token sequence $<\text{sos}>, \text{t}_1, \text{t}_2, \cdots, \text{t}_\text{N}, <\text{eos}>$ (recall that if we are just given a sequence $t_1, \cdots, t_N$ we must add <sos> at the beginning and <eos> at the end of the sequence to make it complete).

We assume that the probability is being computed by a neural-network language model. To do so, we first decompose the probability using Bayes rule as:

$$\begin{aligned} P(<\text{sos}>, \text{t}_1, \text{t}_2, \cdots, \text{t}_\text{N} <\text{eos}>) \quad &= \quad P(<\text{sos}>)\text{P}(\text{t}_1|\ <\text{sos}>)\text{P}(\text{t}_2|\ <\text{sos}>\ \text{t}_1)\cdots \\ & \quad P(t_N|\ <\text{sos}>\ \text{t}_1, \cdots, \text{t}_{\text{N}-1})\text{P}(<\text{eos}>\ |\ <\text{sos}>\ \text{t}_1 \cdots \text{t}_\text{N}) \end{aligned}$$

Here $P(t_n|\ <\text{sos}>\ \text{t}_1, \cdots, \text{t}_{\text{n}-1})$ is the probability that t_n is the n^{th} token in a sequence, given that $<\text{sos}>\ \text{t}_1, \cdots, \text{t}_{\text{n}-1}$ are the first $n-1$ tokens from the beginning (i.e. from <sos>). By this notation, $P(<\text{sos}>)$ is the probability that <sos> is the inital token in a complete sequence. Since <sos> is *always* the beginning of a complete sequence, $P(<\text{sos}>) = 1$.

The rest of the probability terms must be computed by a neural network and have the form $P(t_n|\ <\text{sos}>, \text{t}_1, \cdots, \text{t}_{\text{n}-1}; \theta)$. Here θ represents the parameters of the network. So using this mechanism, and considering that $P(<\text{sos}>)$ is always 1.0, we can write:

$$\begin{aligned} P(<\text{sos}>, \text{t}_1, \text{t}_2, \cdots, \text{t}_\text{N} <\text{eos}>; \theta) \quad &= \quad P(t_1|\ <\text{sos}>; \theta)\text{P}(\text{t}_2|\ <\text{sos}>\ \text{t}_1; \theta)\text{P}(\text{t}_3|\ <\text{sos}>\ \text{t}_1\text{t}_2; \theta) \\ & \quad \cdots P(<\text{eos}>\ |\ <\text{sos}>\ \text{t}_1 \cdots \text{t}_\text{N}; \theta) \end{aligned} \tag{16}$$

$P(t_n|\ <\text{sos}>, \text{t}_1, \cdots, \text{t}_{\text{n}-1}; \theta)$ looks familiar – it is the probability assigned to symbol t_n by a recurrent network that has obtained the sequence $<\text{sos}>, \text{t}_1, \cdots, \text{t}_{\text{n}-1}$ as inputs. This is illustrated by Figure 1.

By extrapolation, since the network is recurrent, *all* of the probabilities $P(t_1|\ <\text{sos}>; \theta)$, $P(t_2|\ <\text{sos}>\ \text{t}_1; \theta)$, $P(t_3|\ <\text{sos}>\ \text{t}_1\text{t}_2; \theta)$, $\cdots$, $P(<\text{eos}>\ |\ <\text{sos}>\ \text{t}_1 \cdots \text{t}_\text{N}; \theta)$ can be computed using the same neural network, as shown by Figure 2.

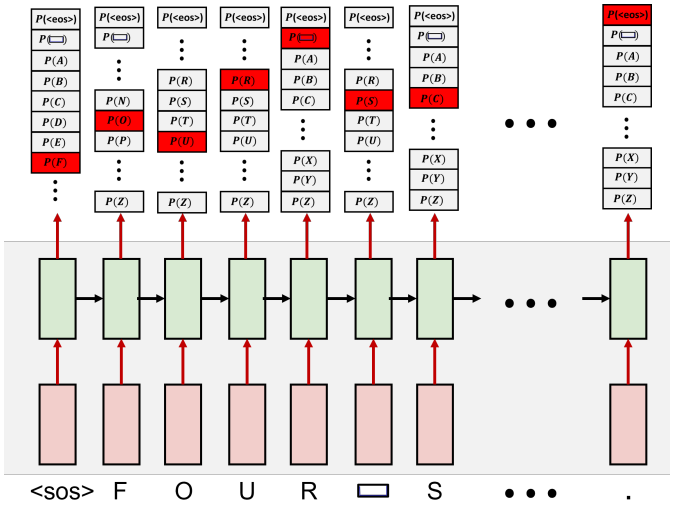

Figure 2: To compute the probability of the entire first sentence of the Gettysburg address, the entire sequences of characters (from the first <sos> until, but *not* including the final <eos>) is fed into the network. At each step, the network outputs the probability distribution for the next character in the sequence. At each time we select the probability assigned to the *next* character in the sequence (highlighted in red). The final probability selected is the probability assigned to <eos> at the final period in the sentence. The product of all of the selected (highlighted) probabilities is the total probability of the sentence.

1.2.3 The Token *Embedding*

The math and the figure above do not clarify how we actually *represent* the tokens ($< sos >$, $< eos >$ and the individual tokens, be they words or characters or whatever else) to the network.

The obvious way is to simply represent every token in the vocabulary (including <sos> and <eos>) as a one-hot vector. As discussed in class, this representation assigns the same length to the vector for any word, and the same distance between any two words (i.e. between their vectors). This way, we make no assumptions about the relative distance between words or the relative importance of words.

However, the one-hot vector approach is incredibly wasteful. The issue is the high dimensionality: For example, a vocabulary of one million words would require a million-dimensional vector! In order to deal with this problem, we typically *project* the one-hot vectors down to a lower-dimensional space, using a linear transformation. Thus, each D-dimensional one-hot vector T ($D = 1000,000$ in our example) is converted to a reduced N-dimensional vector E, by multiplying it by a $N \times D$ projection matrix P, such that $E = PT$. To implement this, The linear projection P can itself be implemented as a $N \times D$ linear layer, which applies to every input. Note that the actual input to the recurrent layers is now $E = PT$.

Thus, the *actual* neural network that models language would look like Figure 3, with an explicit projection layer to derive token embeddings, prior to passing them to the recurrent layers.

This gives us the following algorithm for computing the probability of any complete token sequence $t_1, \cdots, t_N$ using a recurrent neural network.

- Prepend <sos> and append <eos> to the token sequence to get $<\text{sos}>, \text{t}_1, \text{t}_2, \cdots, \text{t}_\text{N}, <\text{eos}>$.
- Pass the entire sequence $<\text{sos}>, \text{t}_1, \cdots, \text{t}_\text{N}$ through the network. It will compute a probability distribution over the vocabulary (symbol set) of the model at each time step. *This vocabulary must include <eos> (and <sos>, if it is separate from <eos>).*
- Within the network, at each timestep $n = 0, \cdots, N$ (where $t_0 =$<sos> and $t_{N+1} =$<eos>):
  - Compute the embedding for the n^{th} token
  - Pass it into the network to compute the probability distribution for the $(n+1)^{\text{th}}$ token,
  - Read out the probability for t_{n+1} from the output probability distribution. Note that the final token $t_{N+1} =$<eos>.

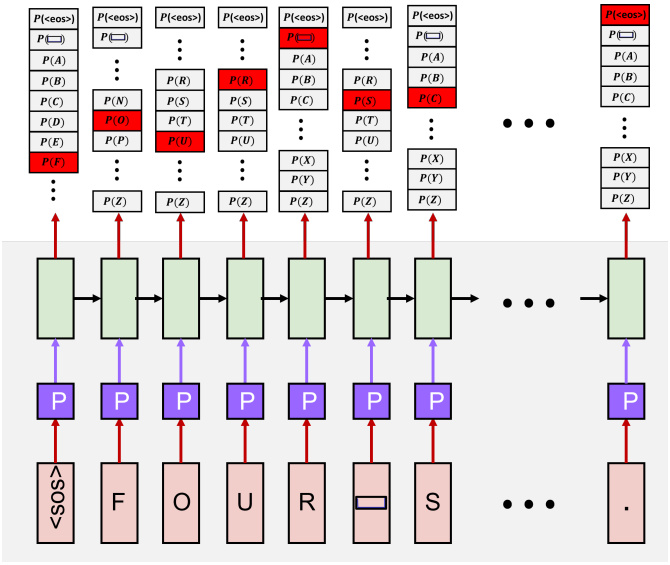

Figure 3: The orange boxes at the bottom represent one-hot embeddings of the input tokens (shown inside the boxes). These are passed through a projection layer (the purple boxes with P) to generate *embedding* vectors that are passed to the recurrent layers.

- Multiply the sequence of probabilities.

What is the projection matrix P? There are several ways to approach P, including *finite* approaches like the soft bag approach, which predicts words based on their immediate context, and the skipgram approach, which predicts adjacent words based upon the current word at hand, as explained in the lecture slides. Here, in this homework, we are using RNNs for language modeling, and P will be considered part of the model and learned along with all the other parameters of the network.

1.2.4 From LSTMs to Transformer Models

In HW3, you learned how to use a LSTM-based model to perform speech recognition. Historically, Recurrent Neural Networks (RNNs), and particularly Long Short-Term Memory (LSTM) networks, have been popular for language modeling due to their ability to handle sequential data. However, Transformer models have revolutionized the field by addressing some of the limitations of RNNs, such as the difficulty in capturing long-range dependencies.

Transformer models, with their self-attention mechanisms, provide a more efficient and scalable approach to language modeling. The self-attention mechanism allows the model to weigh the importance of different tokens in a sequence, enabling it to capture dependencies regardless of their distance. This has significantly improved the performance of language models on various NLP tasks.

There are two main types of Transformer architectures: the encoder-decoder structure and the decoder-only structure.

The encoder-decoder structure, used in tasks like machine translation, consists of two parts: an encoder that processes the input sequence and a decoder that generates the output sequence. The encoder maps the input sequence to a set of continuous representations, which the decoder then uses to produce the output sequence, attending to the encoded information as needed.

On the other hand, the decoder-only structure, which we will focus on in this assignment, is typically used for tasks like language modeling and text generation. In this approach, there is no separate encoder; instead, the model uses a single stack of decoder layers to both encode the input context and generate the output sequence. This simplifies the architecture and is particularly well-suited for autoregressive tasks where the model generates text one token at a time.

We use the decoder-only approach in this homework because it is directly aligned with the goal of language modeling—predicting the next token in a sequence based on the preceding tokens. This approach allows us to efficiently model the probability distribution of token sequences and generate coherent text by sampling from this distribution.

1.2.5 Model Architecture

The architecture used in this homework is a Transformer decoder with pre-norm layers. This means that layer normalization is applied before the main components of each decoder layer, such as self-attention and feed-forward networks. The pre-norm approach has been shown to improve training stability, particularly for deeper models.

Each layer of the decoder consists of the following subcomponents:

- **Self-attention mechanism**: This allows the model to compute attention scores for each token with respect to the others, capturing dependencies between tokens in a sequence, regardless of their relative distance. The self-attention mechanism in the decoder operates in a causal manner, meaning that tokens can only attend to preceding tokens, ensuring that future tokens do not influence the current token prediction.
- **Feed-forward network (FFN)**: After the self-attention mechanism, a position-wise feed-forward network is applied, consisting of two linear transformations with a ReLU activation in between. This component enables the model to learn complex representations for each token.
- **Pre-norm layer normalization**: Layer normalization is applied before both the self-attention and feed-forward network components. This ensures that the inputs to each subcomponent are normalized, which improves the gradient flow during backpropagation and helps stabilize training.

The Transformer decoder stack is repeated several times, with each layer learning progressively higher-level representations of the input sequence. Positional encodings are also added to the input embeddings to ensure that the model captures the sequential order of tokens, as the Transformer architecture itself lacks inherent sequence awareness.

Overall, this architecture is well-suited for autoregressive tasks such as language modeling, where the goal is to predict the next token based on the preceding context. The pre-norm Transformer decoder helps improve both training efficiency and model performance in this task.

The overall structure of the model is illustrated in Figure 4, which provides a detailed view of the Transformer decoder layers and their respective components.

1.2.6 Drawing a sample from a Language Model (a.k.a. generating text)

The other key ability required by a model for any distribution is to enable drawing samples from it. Since an LM is actually probability distributions over token sequences, it must enable us to draw samples from this distribution as follows:

$$\text{<sos>}, \text{t}_1, \text{t}_2, \cdots, \text{t}_\text{N}, \text{<eos>} \quad \sim \quad \text{P}(\text{<sos>}, \text{t}_1, \text{t}_2, \cdots, \text{t}_\text{N} \text{ <eos>}; \theta)$$

This is actually the task of *generating text*. So how exactly do we use our network to produce text?

If you recall from Section 1.1.2, the LM models the probability of a token sequence $\text{<sos>}, \text{t}_1, \text{t}_2, \cdots, \text{t}_\text{N}, \text{<eos>}$, by incrementally decomposing it using Bayes rule as:

$$\begin{aligned} P(\text{<sos>}, \text{t}_1, \text{t}_2, \cdots, \text{t}_\text{N} \text{ <eos>}; \theta) \quad &= \quad P(t_1 | \text{ <sos>}; \theta) \text{P}(\text{t}_2 | \text{ <sos> t}_1; \theta) \cdots \\ & \quad\quad P(t_N | \text{ <sos> t}_1, \cdots, \text{t}_{\text{N}-1}; \theta) \text{P}(\text{<eos>} \mid \text{<sos> t}_1 \cdots \text{t}_\text{N}; \theta) \end{aligned}$$

Here $P(t_n | \text{ <sos> t}_1, \cdots, \text{t}_{\text{n}-1}; \theta)$ is the probability that t_n is the n^{th} token in a sequence, given that $\text{<sos> t}_1, \cdots, \text{t}_{\text{n}-1}$ are the first $n-1$ tokens from the beginning (i.e. from <sos>).

What this tells us is that the sequence $\text{<sos>}, \text{t}_1, \text{t}_2, \cdots, \text{t}_\text{N}, \text{<eos>}$ is drawn from the following probability distribution:

$$\begin{aligned} \text{<sos>}, \text{t}_1, \text{t}_2, \cdots, \text{t}_\text{N}, \text{<eos>} \quad &\sim \quad P(t_1 | \text{ <sos>}; \theta) \text{P}(\text{t}_2 | \text{ <sos> t}_1; \theta) \cdots \\ & \quad\quad P(t_N | \text{ <sos> t}_1, \cdots, \text{t}_{\text{N}-1}; \theta) \text{P}(\text{<eos>} \mid \text{<sos> t}_1 \cdots \text{t}_\text{N}; \theta) \end{aligned}$$

It follows naturally from this equation that each token in the sequence must be drawn from its corresponding distribution:

$$t_i \quad \sim \quad P(t_i | \text{ <sos> t}_1 \text{t}_2 ... \text{t}_{\text{i}-1}; \theta)$$

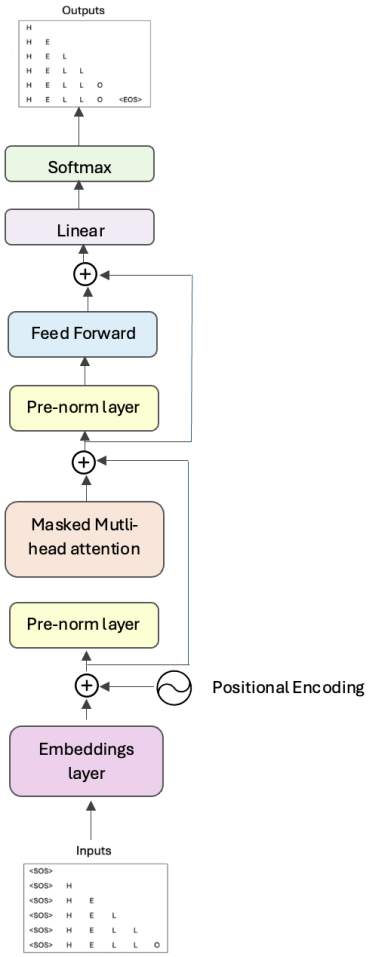

Figure 4: Transformer decoder architecture with pre-norm layers.

I.e. the generation is itself incremental : each token in the sequence is drawn from its own conditional distribution, and then used to condition the distributions of subsequent tokens (from which they are drawn). In terms of the neural-net LM, this means that the n^{th} token in the sequence must drawn from the distribution output by the network at the n^{th} timestep, and then fed back into the network as input to influence the network outputs for subsequent tokens. The generation procedure is illustrated in Figure 5. Note that the figure does not explicitly show the projection matrix P, which is always implicitly assumed.

The following pseudocode explains the procedure. Note that the first symbol is <sos>. Also note how even the generation uses the embedding matrix P. `DrawFromDistribution(Y)` is any routine for drawing a sample from a probability distribution Y. The pseudocode assumes you are directly drawing a token. In reality you would be drawing its index and recovering the actual token from a dictionary.

Pseudocode 1: Generating a token sequence

```
function generate(x, timesteps):
    # Pass input through the model to get initial probability distribution
    token_prob_dist = Decoder(x)
    next_token = DrawFromDistribution(x)
    generated_sequence = [next_token]
    for t in range(timesteps - 1):
        # Pass next_token through the model to get the next probability distribution
        next_prob_dist =Decoder(next_token)
```

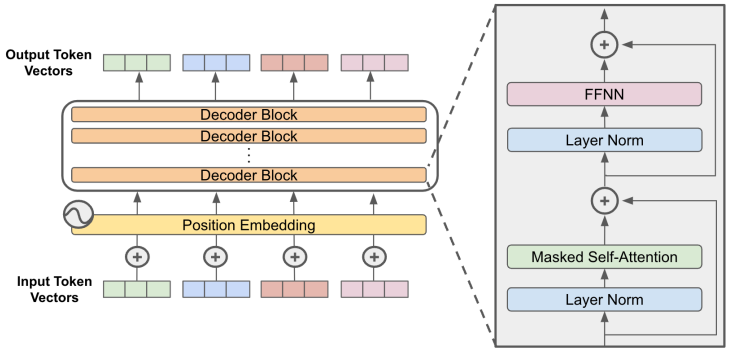

Figure 5: The structure of a decoder-only transformer

```
    # Select the most probable token from the distribution
    next_token = DrawFromDistribution(generated_sequence)
    # Append the selected token to generated_sequence
    generated_sequence.append(next_token)
    # Update x by appending next_token
    generated_sequence = concatenate(generated_sequence, next_token)
return generated_sequence
```

1.2.7 The `predict` Function

The `predict` function is designed to generate a sequence of tokens over a specified number of timesteps, starting from an initial input sequence. While the `generate` function focuses on producing a sequence of tokens by selecting the most probable token at each step, the `predict` function serves a slightly different purpose.

The key differences and the necessity of the `predict` function are outlined below:

Purpose of the `predict` Function The `predict` function provides a detailed analysis of the model's predictions at each timestep. It records the probability distribution over the vocabulary for each token in the generated sequence. This is particularly useful for understanding the model's behavior and decision-making process during the generation of text.

Method

1. Convert the input sequence `x` to a tensor and move it to the appropriate device (CPU or GPU).
2. Initialize an empty list `prob_dists` to store the probability distributions for each timestep.
3. For each timestep:
   (a) Pass the current input sequence `x` through the model to get the output probability distribution `y`.
   (b) Extract the probability distribution for the last token in the sequence (`last_prob`).
   (c) Append `last_prob` to the `prob_dists` list.
   (d) Select the most probable token from `last_prob` (`next_token`).
   (e) Append `next_token` to the input sequence `x` for use in the next iteration.
4. Stack all collected probability distributions into a tensor and return it.

Necessity of the `predict` Function While the `generate` function is ideal for generating coherent and contextually relevant text sequences, the `predict` function is crucial for debugging and analyzing the model's performance. By examining the probability distributions at each timestep, one can gain insights into how the model assigns probabilities to different tokens, which can help identify any biases or issues in the model's predictions.

1.3 Training the Language Model

Let us now consider how we would *train* a neural language model that computes $P(t_n| < eos >, t_1, \cdots, t_{n-1}; \theta)$ using a Transformer decoder.

Recall that the language model is essentially a parametric model for a probability distribution. We will use the standard approach for training parametric probability-distribution models—*maximum likelihood estimation* (MLE). In the following sections, we first outline the concept of maximum likelihood estimation (MLE) and how it applies to training neural language models.

1.3.1 Maximum Likelihood Estimation

Maximum Likelihood Estimation, or MLE, is a general method for estimating a parametric probability distribution for a random variable from training data.

Let X be any random variable (e.g., the outcome of a roll of dice, the height of people in a population, or sentences from a language). Our objective is to estimate the probability distribution of X.

To estimate the probability distribution of the random variable, we assign it a *parametric* model $P(X; \theta)$ (where θ are the parameters of the model), such as a *categorical* distribution (e.g., for the roll of dice, the parameters are the probabilities of the six sides), a *Gaussian* distribution (e.g., for the height of people, the parameters are the mean and variance of the Gaussian), or, as in our case, a *neural network* (e.g., for language, where the parameters of the distribution are the parameters of the network).

Given a collection of training samples $X_1, X_2, \cdots, X_N$ of the variable (e.g., a number of rolls from the dice, the heights of a number of people from the population, or a large number of sentences from the language), we attempt to estimate the parameter θ such that $P(X; \theta)$ best explains our training data $X_1, X_2, \cdots, X_N$.

The governing principle behind maximum likelihood estimation is that the observed training data are *highly typical* of the process. In fact, the data are *so* typical that of all the models in the world, the one that assigns them the highest probability is assumed to be their underlying distribution.

So, in keeping with this principle, we choose the parameter θ such that $P(X; \theta)$ assigns the *most* probability to our training data:

$$\hat{\theta} = \arg\max_{\theta} P(X_1, X_2, \cdots, X_N; \theta)$$

where $\hat{\theta}$ is our estimate.

Often, we can assume that the individual training instances are independent (and identically distributed). In this case, $P(X_1, X_2, \cdots, X_N; \theta) = \prod_i P(X_i; \theta)$.

$$\hat{\theta} = \arg\max_{\theta} \prod_i P(X_i; \theta)$$

Maximizing a product of many terms can be difficult. To simplify matters, we note that the logarithm converts products to sums. Moreover, it is monotonic—the maximum of a function occurs at the same value of the argument as the maximum of the *log* of the function. This leads us to the following rule for maximum likelihood estimation of the model parameter θ:

$$\hat{\theta} = \arg\max_{\theta} \sum_i \log P(X_i; \theta)$$

1.3.2 Applying the MLE to Training a Transformer Decoder-Based LM

We now move to the task of applying the Maximum Likelihood Estimation principle to training a language model using a Transformer decoder.

Suppose we have a set $S = \{S_1, S_2, \cdots, S_K\}$, where each S_i is a training token sequence: $S_i = (< \text{sos} >$, $\text{T}_{\text{i},1}$, $\text{T}_{\text{i},2}, \cdots, \text{T}_{\text{i},\text{l}_\text{i}}$, <eos>), where l_i is the length of the sequence. Recall that you must attach <sos> and <eos> to the start and end of every sequence you are provided, so we assume here that those are already included in the sequences.

From Equation 16 (Section 1.2.2), the log probability of any sequence S_i is given by:

$$\log P(S_i;\theta) = \sum_{t=1}^{l_i+1} \log P(T_{i,t}| < sos >, T_{i,1}, \cdots, T_{i,t-1};\theta) \tag{17}$$

Here we have assumed that $T_{i,l_i+1} =< eos >$.

Assuming all of the training sequences are independent, the total log probability of the training data is:

$$\log P(S;\theta) = \sum_i \log P(S_i;\theta)$$

Note that since the training data S are given, this is actually just a function of θ.

By the MLE principle, our estimate of θ is given by

$$\hat{\theta} = \arg\max_\theta \sum_i \log P(S_i;\theta)$$

This is given in the form of a *maximization*. To convert it to the standard loss *minimization* format, we can define our loss as the negative of $\log P(S;\theta)$ to get:

$$Loss(\theta) = -\sum_i \log P(S_i;\theta)$$

where $\log P(S_i;\theta)$ is given by Equation 17. Note that this is just the NLL (negative log likelihood) loss. The actual optimization is performed as

$$\hat{\theta} = \arg\min_\theta Loss(\theta)$$

This can be minimized using gradient descent, as always.

The following pseudocode explains how the loss is computed. The *current* estimate of the parameter, θ, is explicitly shown in the code for clarity. The code is split into two routines, one which computes the loss for a single sequence, and the second which aggregates the loss over a set of sequences.

Pseudocode 2: LM training loss for a single sequence

```
function Loss = SequenceLoss(S, theta)
    LogProb = 0
    i = 0 # Assuming S[0] = <sos>
    do
        E = P * embedding(S[i]) # P is the projection matrix from one-hot vectors to embeddings
        Y = Decoder(E)
        LogProb = LogProb + log(Y[S[i+1]])
        i = i + 1
    until (S[i] = <eos>) # Stop when we hit the end of the sequence
    Loss = -LogProb
    return Loss
end
```

Pseudocode 3: LM training loss for a set of training sequences

```
# Given a set of sequences TrainingSequences = {S1, S2, ..., SN} and model $\theta$, computes the loss
function Loss = ComputeLoss(TrainingSequences, theta)
    Loss = 0
    for S in TrainingSequences:
        Loss = Loss + SequenceLoss(S, theta)
    end
    return Loss
end
```

1.4 Evaluating your Language Model

How exactly do you *evaluate* whether the language model you have trained is good?

There are several methods. Here are two simple techniques:

1.4.1 Predicting the Next Token

The language model (LM) learns to compute $P(T_{k+1}| < sos >, T_1, T_2, \cdots, T_k; \theta)$, i.e., the probability distribution of the *next* token in a sequence T_{k+1}, given the entire sequence up to the current token T_k. All other probabilities are composed from these incremental probabilities. So, the obvious way to test the quality of your LM is to verify how accurately it computes this term, $P(T_{k+1}| < sos >, T_1, T_2, \cdots, T_k; \theta)$.

If the LM has been well trained and has learned the structure of language well, then given the first K words of any natural sequence of tokens, it would ideally assign the highest probability (or, at least a high probability) to the *true* next token. For instance, if it were given the word sequence "<sos> Four score and seven years" and asked to compute the probability distribution of the next word, it would ideally assign the highest probability to the word "ago". In other words, $P(\text{ago}| < sos > \text{ four score and seven}; \theta)$ would ideally be assigned a high probability, if not a probability greater than $P(W| < sos > \text{ four score and seven}; \theta)$ for any other word W.

A good test would be as follows: For a large test set of partial token sequences (or word sequences) of the kind <sos> $\mathrm{T}_1\ \mathrm{T}_2\ \cdots\ \mathrm{T}_\mathrm{k}$, where we also know the next token T_{k+1} in the sequence, compute the probability distribution for the $(k+1)^{\text{th}}$ token. If, over the entire test set, the average predicted log probability for the last token (in the sequence) is high enough, the LM may have been well estimated. Note that this example is a *supervised* test. We *know* the next token T_{k+1}, and we are evaluating the LM by how well it predicts it.

The pseudocode below can be used to compute the (log) probability for any given partial token sequence. The predicted log probability of the actual next token is simply $\log Y[T_{k+1}]$. For this HW, since our vocabulary consists of characters, we predict ten characters for this task, which can be one or more words in reality.

Pseudocode 4: Probability distribution of the next token

```
# Given a partial token sequence Tpartial = <sos> t1 t2 ... tK, compute probability distribution of next
    token
function Y = Predict(Tpartial, timesteps)
    # Initialize list to store probability distributions for each timestep
    prob_dist = []
    for i = 0:timesteps
        # Pass the input sequence through the model
        Y = model.forward(x)
        # Append the probability distribution to the list
        prob_dist.append(Y)
        # Select the most probable token from the distribution
        next_token = DrawToken(Y)
        # Append the selected token to the input sequence for the next prediction
        x = concatenate(x, next_token)
    end
    return Y
end
```

1.4.2 Sequence Completion

A second method to test the language model is to evaluate how well it *extends* (or completes) partial token sequences. For the purpose of explanation, we will assume tokens are words.

The test scenario: We are given a partial word sequence, e.g., "<sos> four score and". We use the LM to extend the sequence by N words (or to complete the sequence). The initial sequence of words is input to the network. We ignore the first few outputs of the network and only begin drawing words from the output at the last word in the sequence. Generation continues until the sequence has been extended by N words or until an <eos> has been encountered.

The pseudocode below explains this procedure. Note that we have not yet defined what `DrawToken()` does (but you can continue reading to find out how it's defined). The task is generally the same as predicting the next word, in which we predict 10 characters. For completing the entire sequence, we predict 30 characters.

Pseudocode 5: Completing a word sequence

```
# Given a partial word sequence Tpartial = <sos> w1 w2 ... wK, generates the next N words
function TokenSeq = Complete(Tpartial, N)
    # Convert input sequence Tpartial to tensor and move to device
    x = to_tensor(Tpartial).to(device)
    # Initialize TokenSeq with the input partial sequence
    TokenSeq = Tpartial
    # Generate the rest of the sequence
    for j = 1:N
        # Pass the current sequence through the model
        Y = model.forward(x)
        # Select the most probable token from the distribution
        NewToken = DrawToken(Y)
        # Append the new token to the sequence
        TokenSeq.append(NewToken)
        # Update x with the new token
        x = concatenate(x, NewToken)
        # Stop if the end-of-sequence token is generated
        if NewToken == <eos>: break
    end
    return TokenSeq
```

If the LM is well trained, this extension (or completion) will be *natural*. "Natural", here, is unfortunately hard to quantify. If, however, we had access to an oracle (such as another LM trained on an extremely large amount of data that we could use to score our outputs), we could have that oracle score our completion; and if that oracle were to score our completion as high, the completions could be deemed reasonably natural and our LM could be good.

For such an oracle to work, we must ideally complete the sequence *as best as we can*. In principle, even a random draw is likely to be plausible if our LM is accurate, and `DrawToken(Y)` could just be a random draw from the distribution Y (which we referred to as `DrawFromDistribution()` in Section 1.2.6).

A more reasonable approach, though, is to try to find an extension that is highly probable according to our LM. If the LM is well trained, high-probability extensions are likely to be closer to the distribution of our training data, and are hence more likely to be plausible. The `DrawToken(Y)` function here would draw the most probable token at each time:

```
function t = DrawToken(Y)
    return Token[argmax(Y)]
end
```

Alternatively, we could try to draw the *most likely* extension of the word sequence. However, as we recall from 1.2, neural-network based LMs do not facilitate identifying the rank-ordering of sentences by probability, nor do they facilitate finding the most probable sentence. There is no clean solution to guessing the sequence, and so we must use some heuristics such as greedy or beam search. For this HW we will only use the greedy approach.

2 Dataset

For this homework we will be training a Causal Transformer Decoder on the transcriptions from the LibriSpeech 1000 hours 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 consists of 281,241 rows in a csv file called transcripts.csv, each containing transcriptions formatted as sequences of character tokens. These files represent spoken utterances transcribed into character sequences. An example transcription from one of the training files is shown below, demonstrating how speech is broken down into

individual character tokens:

```
<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>
```

The vocabulary consists of a concise set of 31 items, including the entire English alphabet and special tokens that facilitate sequence processing. These special tokens are:

- `<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.
- `'`: Apostrophe - Accommodates contractions and possessive forms in text.
- `' '`: Space - Serves as a separator between words.

Here is the complete vocabulary used in our dataset:

```
VOCAB = ["<sos>", "<eos>", "A", "B", "C", "D", "E", "F", "G", "H",
"I", "J", "K", "L", "M", "N", "O", "P", "Q", "R", "S", "T", "U", "V", "W",
"X", "Y", "Z", "'", " ", "<pad>"]
```

The fixtures folder will be used for evaluation. It contains pre-processed data files specifically designed for validation and testing. Below is a detailed description of the contents and purpose of each file in the fixtures folder:

- **fixtures\prediction.npz**: This file is used for validation. It includes 128 pairs, each consisting of an input sequence of 30 characters and a corresponding output sequence of 20 characters that the model is expected to predict.
- **fixtures\prediction_test.npz**: This file is used for testing under conditions similar to those in validation. It includes 128 pairs, each consisting of an input sequence of 30 characters. Unlike validation, the corresponding 20 output characters are hidden, and the model is expected to predict these characters.
- **fixtures\generation_test.npz**: This file is used for testing for the generation task. It includes 128 pairs, each consisting of an input sequence of 40 characters. The model is expected to generate an additional 30 characters.

You don't need to worry about these files, test scripts that process them are already set up for you. Refer to Section 5 for more details about the testing procedure.

3 Problems

3.1 Self-attention (20 points)

This is a stand-alone class that has to be implemented in the provided attention.py file. You will have to implement the forward and the backward functions of this class. Use the equations provided in Section 1.1 and complete this file. You do not have to use this implementation in the language model. This is to ensure you know how attention works in detail.

3.2 Prediction of a 10 characters (40 points)

Complete the function `prediction` in class `CausalLanguageModel` in the Jupyter Notebook.

This function takes as input a batch of sequences, shaped `[batch size, sequence length]`. This function should use your trained model and perform a forward pass.

Return your model's guess of what the next 10 characters might be. You should return the score for the last timestep from the output of the model. The returned array will assign a probability score to every token in the vocabulary, so the result should be in shape of `[batch size, vocabulary size]`, of dtype float.

These input sequences will be drawn from the unseen test data. Your model will be evaluated based on the score it assigns to the actual next 10 characters in the test data. Note that scores should be raw linear output values. Do not apply softmax activation to the scores you return. **Required performance for this task is a negative log likelihood of 5.2 on the test set**. The value reported by autolab for the test dataset should be similar to

the validation NLL you calculate on the validation dataset. If these values differ greatly, you likely have a bug in your code.

3.3 Generation of a Sequence (40 points)

Complete the function `generate` in the class `CausalLanguageModel` in the Jupyter Notebook.

As before, this function takes as input a batch of sequences, shaped `[batch_size, sequence_length]`.

Instead of only scoring the next 10 characters, this function should generate an entire sequence of 30 characters from 20 characters taken as input. The length of the sequence you should generate is provided in the **timesteps** parameter. The returned shape should be `[batch_size, timesteps]`. This function requires sampling the output at one time-step and incorporate that in the input at the next time-step. You can call the function `predict` in class `CausalLanguageModel` to get probabilities of the next character. Please refer to Recitation 8 for additional details on how to perform this operation.

Code in the notebook will write your generations to a file as the model gets trained. The generations will be evaluated using an LLM provided by Mistral AI. The code to run this evaluation is present in the notebook in a cell. You are encouraged to understand the code to introduce yourself to the Mistral AI API. **You have to run this cell with the runid, epoch number, and your API key. Then submit the perplexity value obtained.** More instructions about submitting this are covered in Section 5.

Perplexity[4] is defined as the exponentiated average negative log-likelihood of a sequence. The models you will be able to train using the Decoder-only model won't be nearly as powerful as the LLM we will be evaluating your generations with. You might consider this LLM as an "oracle" model mentioned in Section 1.4.2. **For reasonable enough generations, the perplexity will be less than 1400. This is the required performance to get credit for the generation task.**

Note: Unlike Section 1.4.2, the generation function which you are required to implement should not draw *<sos>* or *<eos>* from the vocabulary, although your vocabulary and the embedding layer will include it. We instead require you to generate for a fixed number of timesteps T. Hence, when you are drawing your tokens for completion, please draw from the probability distribution without including *<sos>* and *<eos>*.
Hint: You can do this by slicing your probability distribution by the actual number of words in the vocabulary before you draw from the distribution.

4 Implementation

We break down the problem to these steps. First, implement self-attention since this is a stand-alone problem. Next start with the data loader. Then, set up the model with predict and generate functions. After we have both components, we provide guidance on training techniques for the language model.

4.1 Self-attention

See Sections 1.1 and 3.1 for more details. Complete the attention.py file using the equations provided in Section 1.1. Run the autograder using the command `python hw4/hw4p1_autograder.py` from the handout directory. While we will test on some hidden tests on Autolab, generally full points locally will ensure full points on Autolab. While creating the handin file, ensure the handin has this file. The makefile used to create the handin should take care of this. More instructions about submission are in Section 5.

4.2 Brief Note on the Language Modeling Task

You have been provided a starter notebook to complete the language modeling task. You will need to build and train a language model which will require a GPU. If you have a GPU on your local machine, that's fine. However, if you use a cloud resource (Colab, GCP, AWS, etc.), please upload the handout directory to the cloud environment as the notebook accesses other files in the handout too. The command to generate the handin is also present in the notebook and will access the required files. In case you are using Colab, you might want to upload the handout directory to your drive and mount the drive in Colab. **This homework is doable on free Colab/Kaggle notebooks. So, you might want to save on credits by not setting it up on GCP/AWS.**

[4] https://huggingface.co/docs/transformers/perplexity

4.3 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>.".

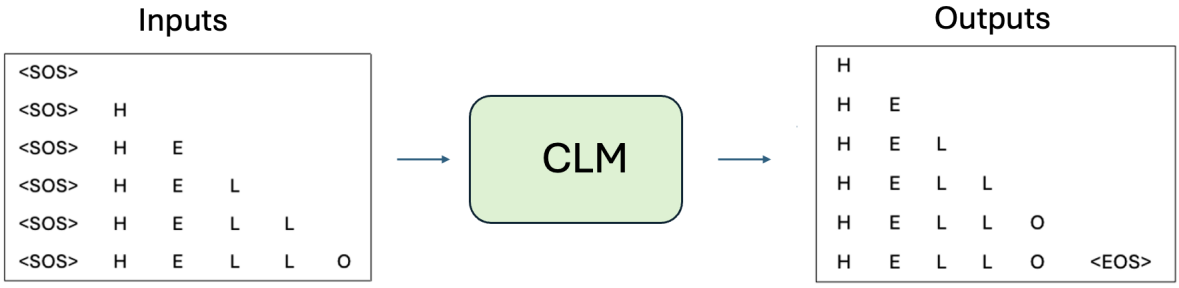

Figure 6: The shifted input sequence, beginning with a start-of-sentence token (<SOS>) and excluding the end-of-sentence token (<EOS>), is used by the decoder to predict the subsequent token in the sequence. The target sequence is the true next token the model aims to predict at each timestep, including the <EOS>token but excluding the initial <SOS>.

4.4 Model Description

Your next task is to fill out the `CausalLanguageModel` class. You will complete the implementation of the `CausalLanguageModel` class using a decoder-only transformer architecture. These models are essential for text generation, handling tasks ranging from generating conversation replies to creating original content. Unlike Encoder-Only Models, which are excellent at understanding, Decoder-Only Models specialize in producing coherent and contextually appropriate text.

- **What Are Decoder-Only Models?**
  Decoder-only transformer architectures are pivotal in prominent language models such as GPT-3, ChatGPT, GPT-4, PaLM, LaMDa, and Falcon. These models uniquely handle language by focusing on generating new text, unlike Encoder-Only Models, which excel at interpreting and analyzing existing text. The concept of transformers, combining both encoder and decoder components, was first introduced in the 2017 paper "Attention is All You Need." However, recent trends with GPT models highlight the shift towards decoder-only models due to their outstanding performance in text generation.

  What distinguishes these models is their operational approach. Decoder-only models take input, ranging from simple prompts to complex data sets, and generate text that is relevant to the input. This process is similar to responding in a conversation or writing an essay on a given topic. The model uses the input as a starting point to produce coherent and contextually appropriate text.

The strength of Decoder-Only Models lies in their ability not only to mimic human-like text but also to exhibit creativity in their responses. They can generate stories, answer questions, and engage in natural and fluid dialogue. This versatility makes them incredibly valuable in various applications, including chatbots, digital assistants, content creation, abstractive summarization, and storytelling.

- **How Decoder Only Models work?**
  To better understand these models, let's look at the original Transformer architecture, which included both Encoder and Decoder components. Recently, there has been a trend towards models specializing in either encoding, such as Google's BERT, or decoding, such as GPT. Decoder-Only Models belong to the latter category.

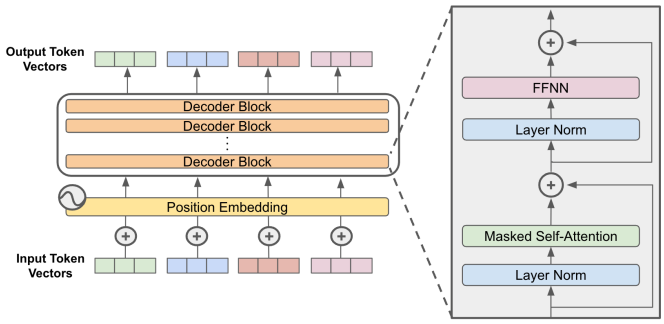

Figure 7: Decoder-Only Architecture

The basic structure of a Decoder-Only Model is relatively simple and typically includes the following components:

1. **Token Input Layer:** This layer receives the input sequence of tokens.
2. **Decoder Block Layer:** This is the model's core and includes several components:
   - *Masked Multi-Head Self-Attention Layer:* Essential for understanding the input sequence. The 'masked' feature ensures that predictions for a token do not consider future tokens, preserving the autoregressive nature of the model.
   - *Fully Connected Layers:* These layers further process the information.
   - Positional Encoding: Adds information about the position of each token in the sequence, which is crucial for understanding the order of words.
3. **Output Layer:** A fully connected layer followed by a SoftMax function is applied over the entire vocabulary to predict the next word in the sequence.

Decoder-Only Models can be constructed with multiple Decoder Blocks in a vertical arrangement, depending on the specific needs. The core component driving these models is the Masked Self-Attention mechanism, which enables the model to selectively focus on different input sequence parts when generating each token, resulting in contextually appropriate text.

These models typically undergo pre-training using an extensive collection of language data, often encompassing a large portion of internet-available text. During this initial training phase, the model's primary objective is to predict the subsequent word in each text sequence, allowing it to develop a comprehensive understanding of language and generate human-like text.

Following pre-training, the models can be adapted for particular tasks through fine-tuning. This process, which may involve techniques such as Instruction Tuning or Reinforcement Learning from Human Feedback (RHLF), prepares the model for specific applications like question-answering systems, virtual assistants, or dialogue-based platforms. In practical applications, Decoder-Only Models utilize algorithms such as Greedy Search or Sampling to select the most suitable words when generating text. This text generation capability makes these models particularly valuable for creating content that demands a high level of contextual understanding and coherence, making them well-suited for applications involving human-like interaction and content creation.

Within the provided code, you have each of the layers discussed as explain in the following lines:

1. `Embedding Layer` (`self.token_embedding`): This layer maps input tokens to their corresponding embeddings. Use the `nn.Embedding` class from PyTorch to define `self.token_embedding`. It takes in `num_embeddings`, the size of the vocabulary, and `embedding_dim`, the size of each embedding vector. These parameters should be defined based on the size of the vocabulary and the embedding dimension you choose.
2. `Decoder Layers (self.layers)`: The decoder consists of multiple `DecoderLayer` instances stacked together. Each `DecoderLayer` includes a self-attention mechanism and a feed-forward network. The number of decoder layers, attention heads, and the dimensions of the feed-forward network are parameters you can customize.
3. `Linear Projection Layer (self.fc)`: This layer projects the output from the decoder to the vocabulary size, providing a score for each token. It is defined as a linear layer with `in_features` equal to the embedding dimension and `out_features` equal to the vocabulary size.
4. `Positional Encoding (self.pos_encoder)`: This layer adds positional information to the token embeddings. It helps the model to understand the order of tokens in the input sequence.

There are also a number of functions within our `LanguageModel` that you are expected to fill out. These are as follows: `predict`, `generate`, and `forward`.

How the `predict` and `generate` methods work? As mentioned earlier in Sections 3.2 and 3.3, `predict` returns our model's best guess of what the next token might possibly be, and `generate` generates an entire sequence of words to extend the current input sequence. Section 1.4 goes into great detail on the steps needed to perform these high-level tasks, so please refer to it as well as Section 3 when you are implementing these methods.

We finally come to the `forward` method in our `LanguageModel` class. This method processes a batch of input sequences by performing the following steps:

1. Obtain the embeddings for the input tokens.
2. Add positional encodings to the embeddings.
3. Pass the embeddings through the decoder layers.
4. Project the output to the vocabulary size using the linear layer.
5. Return the token probability distributions.

4.5 Training and Regularization Techniques

Since the LibriSpeech data set for HW4p1 has been pre-processed, we can focus on breaking the text down into tokens for training the decoder. As you may have noticed in HW3, a good initialization significantly improves training time and the final validation error.

To achieve better performance on your language model, you will have to use regularization methods such as Weight tying, Embedding Dropout, and also Data Augmentation. For structuring and training your model, we would like you to follow the protocols in Continual Learning Optimizations for Auto-regressive Decoder of Multilingual ASR systems as closely as you are able to, in order to guarantee maximal performance. You are not expected to implement every method in this paper, and the regularization techniques (below) will be sufficient to achieve performance to pass on Autolab.

Ensure your implementation is consistent with the batching strategy you choose (batch first or batch last), since you may be required to transpose the outputs of your prediction and/or generation models.

4.5.1 Embedding Dropout

Embedding dropout (Gal & Ghahramani (2016)) is to perform dropout on the embedding matrix at a word level. The dropout is broadcasted across the entire embedding vector of the word. The remaining non-dropped-out word embeddings are scaled by $\frac{1}{1-p_e}$ where p_e is the probability of embedding dropout. Embedding dropout is performed for a full forward and backward pass, which means all occurrences of a specific word will disappear within that pass.

Note that if you use torch.nn.Embedding for the embedding layer, the input to that layer is an integer and not a vector. Therefore, to implement embedding dropout, we need to first convert the input to a one-hot vector (look

into torch packages for how), apply regular Dropout on it, and then manually multiply it with the weights of the embedding layer to complete the conversion to embeddings.

4.5.2 Weight Tying

Weight tying (Inan et al., 2016; Press & Wolf, 2016) shares the weights between the embedding and softmax layer, substantially reducing the total parameter count in the model. The technique has theoretical motivation (Inan et al., 2016) and prevents the model from having to learn a one-to-one correspondence between the input and output, resulting in substantial improvements to language models.

To implement weight tying, you can manually link the weight component of two different layers to be the same object so that they will have the same weights.

5 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 for the homework.

5.1 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, the self-attention portion will finally be graded on some hidden tests on Autolab like other Part 1s.

5.2 Language Modeling - the Notebook

Within the starter Jupyter Notebook, `hw4/training.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.

5.3 Language Modeling - Prediction

The notebook for the language modeling portion has code to measure validation NLL on the prediction task as you train your model. You can use this validation metric to understand how your model is doing.

The notebook also saves the logits returned by the predict method for the test set of the prediction task. These logits will be part of your handin and graded on Autolab. **The NLL on the test set should be less than 3.2 to get credit for the prediction task.**

The average model will likely take around 10 or more epochs, to achieve **a validation NLL below 1.6** on the prediction task. We have seen the **validation NLL to be around 0.5-0.7 lower** than that on the test set.

With a good set of hyperparameters and some well chosen regularization techniques (mentioned earlier in subsection 4.5), you can get a validation NLL below 3.0 in no more than 3 epochs.

5.4 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 using the *GPT-2 Medium Language Model* (`gpt2-medium`) from HuggingFace. You are required to fill in the best `runid` and `epoch number` in this cell and execute it to compute the perplexity. **For this part of the assignment, achieving a perplexity below 100 is necessary to pass, indicating that the generated text meets a minimum standard of coherence and fluency, as judged by the GPT-2 Medium model.**

5.5 Create the handin for submission

Once you have completed attention.py, trained a model with good enough validation NLL, and obtained 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.

The handout directory has a makefile that will create the submission handin.tar. You can use the following command from the handout directory to generate the submission file.

`make runid=<your run id> epoch=<best epoch number of that run> ppl=<generation perplexity>`

For example, `make runid=1698907689 epoch=7 ppl=1058.2837014084269`

You can find the run ID in your Jupyter notebook (its just the CPU time when you ran your experiment). You can choose the best epoch by looking at the validation NLL during training. You will then have to submit the file **handin.tar** to Autolab to get your score for the homework.

You will get only 5 submissions for this homework. So ensure that you submit the handin you are confident will cross the cut-offs mentioned in the writeup.

5.6 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 using only a subset of the methods specified in section 4.6.

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 !