Homework 4 Part 1
Language Modeling using RNNs

11-785: Introduction to Deep Learning (Spring 2024)
OUT: **March 29, 2024**
EARLY BONUS DDL: **April 10, 2024, 11:59 PM**
FINAL DDL: **April 26, 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:**
  - **Submisssion Format and Grading**: All of the problems in Part 1 will be graded on Autolab. Download the starter code from Autolab.
  - **Masked Self-Attention**: Complete the forward and backward functions of the **Attention class** in the attention.py file.
  - **Single Word 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 **Masked Self-Attention** from scratch.
  - Learn how to train **Recurrent Neural Network models** for generating text.
  - Learn about various techniques to regularize Recurrent Neural Networks like **Locked Dropout**, **Embedding Dropout**, **Weight Decay**, **Weight Tying**, **Activity Regularization**.
  - You will understand the workings of Language Modeling and you will also train one to generate next-word as well as entire sequences!

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 HW4P2), but this homework itself (HW4P1) actually models language in terms of words. Please do not let this confuse you; we expect you to generalize the concepts presented using character sequences as examples to apply to word sequences.

In addition to this, you will also learn how to implement the forward and backward functions of masked 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 use transformers.

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).

1 Introduction

Key new concepts: Masked self-attention, language modeling, text generation, regularization techniques for RNNs

Mandatory implementation: Attention class, RNN-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 Masked 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.

The attention mechanism comes in different flavors. Let's look at some of them:

- **Self-attention**: Analyzes relationships within a single sequence.
- **Masked self-attention**: Similar to self-attention, but prevents certain data from influencing the current position. We can mask different parts of the sequence:
  - **Causal mask**: Prevents a sequence position from seeing future positions. It ensures predictions rely only on past data.
  - **Padding mask**: Nullifies the impact of padding tokens in sequences.
  - **Localized mask**: Restricts attention to a nearby range or "window" around each token.
- **Cross-attention**: Focuses on interactions between two different sequences, enhancing relevance and alignment in tasks like translation.
- **Multi-head attention**: Runs several attention processes in parallel, capturing diverse relationships and nuances in data.
- ... and many more.

In this homework, **you will be implementing a self-attention with a causal additive mask (i.e., masked self-attention)**. Let's discuss its key components:

- **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.
- **Additive Causal Mask**: This mask ensures a position is influenced only by past and current data, not future data. "Causal" means each position in a sequence can't see ahead. "Additive" refers to adding large negative numbers (e.g., -1e9) **above** the main diagonal of the attention matrix. These negatives become zeros after the softmax step, effectively hiding future positions. This method is key in sequence-to-sequence tasks to maintain proper information flow. We use additive rather than multiplicative masking because additive

operations maintain the stability of the softmax step, preventing numerical instability or overflow issues that can occur with the multiplication of large values.

In the following section, we provide the equations for the forward and backward functions of a masked self-attention. 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

        self.softmax = Softmax() # Already implemented

    def forward(self, X):

        self.X     = X

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

        self.A_w   = # TODO

        mask       = # TODO

        self.A_w   = # TODO (adding a causal mask)

        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
```

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

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 Forward Equations

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

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

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

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

$$A_w(w/mask) = A_w + mask \qquad \in \mathbb{R}^{B \times T \times T} \tag{5}$$

$$A_\sigma = \sigma(\frac{A_w}{\sqrt{D_k}}) \qquad \in \mathbb{R}^{B \times T \times T} \tag{6}$$

$$X_n = A_\sigma \cdot V \qquad \in \mathbb{R}^{B \times T \times D_v} \tag{7}$$

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 Backward Equations

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

$$\frac{\partial L}{\partial A_\sigma} = \left(\frac{\partial L}{\partial X_n}\right) \cdot V^T \qquad \in \mathbb{R}^{B\times T\times T} \tag{8}$$

$$\frac{\partial L}{\partial A_w} = \frac{1}{\sqrt{D_k}} \cdot \sigma^{'}(\frac{\partial L}{\partial A_\sigma}) \qquad \in \mathbb{R}^{B\times T\times T} \tag{9}$$

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

$$\frac{\partial L}{\partial V} = A_\sigma^T \cdot \left(\frac{\partial L}{\partial X_n}\right) \qquad \in \mathbb{R}^{B\times T\times D_v} \tag{10}$$

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

$$\frac{\partial L}{\partial Q} = \left(\frac{\partial L}{\partial A_w}\right) \cdot K \qquad \in \mathbb{R}^{B\times T\times D_k} \tag{12}$$

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.

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

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

$$\frac{\partial L}{\partial W_k} = \sum_{i=1}^{B} X^T \cdot \left(\frac{\partial L}{\partial K}\right) \qquad \in \mathbb{R}^{D_k\times D} \tag{15}$$

4. **Derivative wrt input:**

$$\frac{\partial L}{\partial X} = \left(\frac{\partial L}{\partial V}\right) \cdot W_v^T + \left(\frac{\partial L}{\partial K}\right) \cdot W_k^T + \left(\frac{\partial L}{\partial Q}\right) \cdot W_q^T \qquad \in \mathbb{R}^{B\times T\times D} \tag{16}$$

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 Language Modeling `[Jupyter Notebook]`

In addition to the above, you will learn how to use a Recurrent Network (ideally an LSTM-based network) to model language, and generate text. You will also learn about the various techniques we use to regularize recurrent networks and improve their performance.

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

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.

1.2.1 A note on beginnings and endings

Language 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 modeling 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,
```

[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.

```
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 modeling, 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** (Note: Our objective in this assignment however, is to model over a combination of words and characters from the WikiText-2 Language Modeling Dataset), 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 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 $\text{<sos>}, \mathrm{t_1}, \mathrm{t_2}, \cdots, \mathrm{t_N}, \text{<eos>}$ (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 the Bayes rule as:

$$\begin{aligned} P(\text{<sos>}, \mathrm{t_1}, \mathrm{t_2}, \cdots, \mathrm{t_N}\ \text{<eos>}) &= P(\text{<sos>})\mathrm{P}(\mathrm{t_1}|\ \text{<sos>})\mathrm{P}(\mathrm{t_2}|\ \text{<sos>}\ \mathrm{t_1}) \cdots \\ &\quad P(t_N|\ \text{<sos>}\ \mathrm{t_1}, \cdots, \mathrm{t_{N-1}})\mathrm{P}(\text{<eos>}\ |\ \text{<sos>}\ \mathrm{t_1} \cdots \mathrm{t_N}) \end{aligned}$$

Here $P(t_n|\ \text{<sos>}\ \mathrm{t_1}, \cdots, \mathrm{t_{n-1}})$ is the probability that t_n is the n^{th} token in a sequence, given that $\text{<sos>}\ \mathrm{t_1}, \cdots, \mathrm{t_{n-1}}$ 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 initial 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 $P(t_n|\ < \mathrm{sos} >, \mathrm{t_1}, \cdots, \mathrm{t_{n-1}}; \theta)$. Here θ represents the parameters of the network. So using this mechanism, and considering

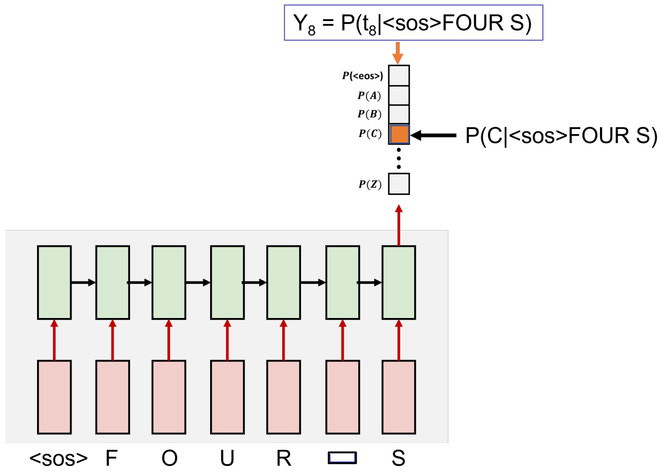


Figure 1: Given the length 7 input "<sos>FOUR S" (including the space), the RNN outputs P(T|<sos>FOUR S), the probability distribution for the eighth character in the sequence given the input. P(C|<sos>FOUR S), the actual probability of having a "C" in the 8th position must be read from this distribution,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

that $P(\text{<sos>})$ is always 1.0, we can write:

$$\begin{aligned} P(\text{<sos>}, \mathrm{t}_1, \mathrm{t}_2, \cdots, \mathrm{t}_\mathrm{N}\ \text{<eos>}; \theta) &= P(t_1|\ \text{<sos>}; \theta)\mathrm{P}(\mathrm{t}_2|\ \text{<sos>}\ \mathrm{t}_1; \theta)\mathrm{P}(\mathrm{t}_3|\ \text{<sos>}\ \mathrm{t}_1\mathrm{t}_2; \theta) \\ &\quad \cdots P(\text{<eos>} \mid \text{<sos>}\ \mathrm{t}_1 \cdots \mathrm{t}_\mathrm{N}; \theta) \end{aligned} \tag{17}$$

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

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

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 ($D = 1,000,000$ 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, before 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 $\text{<sos>}, \mathrm{t}_1, \mathrm{t}_2, \cdots, \mathrm{t}_\mathrm{N}, \text{<eos>}$.

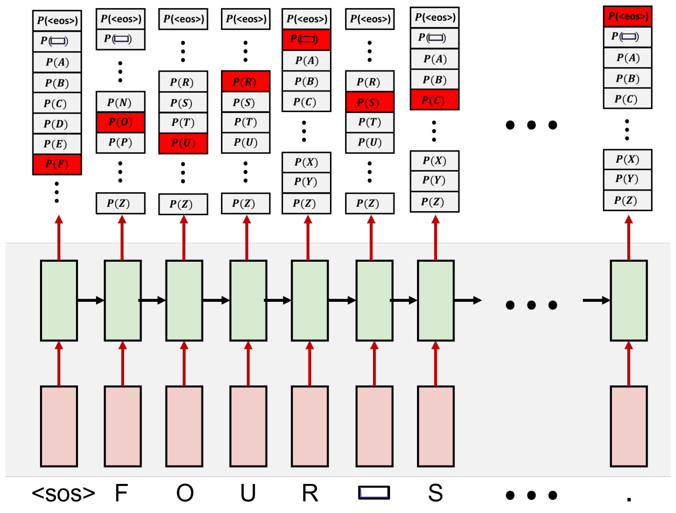


Figure 2: To compute the probability of the entire first sentence of the Gettysburg address, the entire sequences of characters (from the first <sos> until, but *not* including the final <eos>) is fed into the network. At each step, the network outputs the probability distribution for the next character 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.

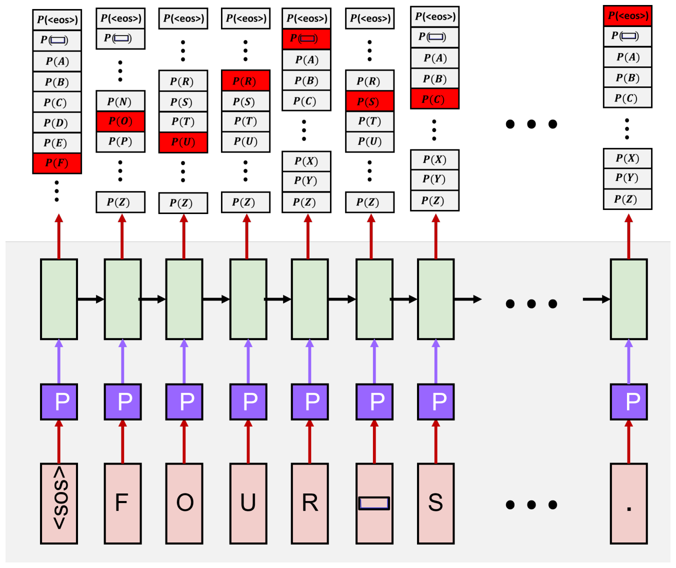


Figure 3: The orange boxes at the bottom represent one-hot embeddings of the input tokens (shown inside the boxes). These are passed through a projection layer (the purple boxes with P) to generate *embedding* vectors that are passed to the recurrent layers.

- Pass the entire sequence $\text{<sos>}, \mathrm{t}_1, \cdots, \mathrm{t}_\mathrm{N}$ 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 $t_0 = \text{<sos>}$ and $t_{N+1} = \text{<eos>}$):
  - 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 $t_{N+1} = \text{<eos>}$.
- Multiply the sequence of probabilities.

The entire operation is given by the following pseudocode. Note that we are actually computing the *log* probability to avoid underflows. Also, the pseudocode refers to words directly, rather than their dictionary indices (which is what you would have in a real implementation).

Pseudocode 1: Computing the probability of a word sequence

```
# Takes in token sequence T and returns its log probability
function LogProb = ComputeTokenSequenceLogProbability(T)
    LogProb = 0
    h = initial_h # Assuming that h[-1] = hinit is part of the model
    i = 0 # Assuming T[0] = <sos>
    do
        E = P*onehot(T[i]) # P is the projection matrix from one-hot vectors to embeddings
        [Y, h] = RNNstep(h, E)
        LogProb = LogProb + log(Y[T[i+1]])
        i = i + 1
    until (T[i] = <eos>) # Stop when we hit the end of the sequence
    return LogProb
end
```

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 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:

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

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

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

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

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

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

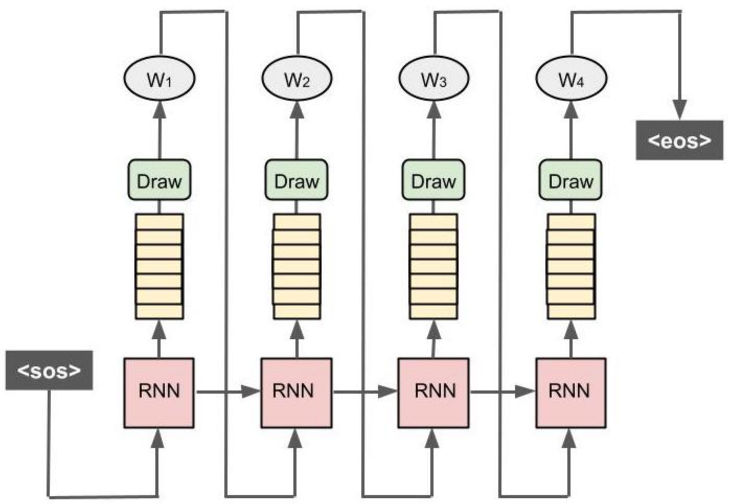


Figure 4: Generating text with the LM. At each time step a word is drawn from the output distribution and input back into the recurrent block at the next time. Although not explicitly shown, we actually input the *embedding* of the word to the RNN block.

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

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

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

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 4. 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 2: Generating a token sequence

```
# Generates a sequence TokenSeq
function TokenSeq = Generate()
    TokenSeq[0] = <sos>
    h = hinit # Initial value of hidden state
    i = 0  # Assuming TokenSeq[0] = <sos>
    do
        E = P*onehot(TokenSeq[i]) # P is the projection matrix from one-hot vectors to embeddings
        [Y, h] = RNNstep(h, E)
        NewToken = DrawFromDistribution(Y)
        i = i + 1
        TokenSeq[i] = NewToken
    until (TokenSeq[i] = <eos>) # Stop when we hit the end of the sequence
    return TokenSeq
end
```

1.2.5 Training the Language Model

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

Recall that the LM is actually just a parametric model for a probability distribution. So we will use the standard approach for training parametric probability-distribution models – *maximum likelihood estimation.* In the following sections we first outline the concept of maximum-likelihood estimation (MLE) and how it applies to training neural LMs.

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

Then, 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, or 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 whatever training data we have observed 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 the one we will assume 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. Morever, 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:

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

Applying the MLE to training an LM

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

Suppose we have a set $S = \{S_1, S_2, \cdots, S_K\}$, where each S_i is a training token sequence: $S_i = (<\text{sos}>, \text{T}_{\text{i},1}, \text{T}_{\text{i},2}, ..., \text{T}_{\text{i},\text{l}_\text{i}}, <\text{eos}>)$, 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 17) (Section 1.2.2) the log probability of any sequence S_i is given by:

$$\log P(S_i;\theta) = \sum_{t=1}^{l_i+1} \log P(T_{i,t}| < sos >, T_{i,1}, \cdots, T_{i,t-1}; \theta) \tag{18}$$

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 $\theta$.

By the MLE principle, our estimate of $\theta$ 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 18. 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, $\theta$, 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 loss over a set of sequences. Note that the first routine `SequenceLoss()` is nearly identical to `ComputeSequenceProbability()` from Section 1.2.3.

Pseudocode 3: LM training loss for a single sequence

```
function Loss = SequenceLoss(S, theta)
    LogProb = 0
    h = initial_h # Assuming that h[-1] = hinit is part of the model (and maybe in theta)
    i = 0 # Assuming S[0] = <sos>
    do
        E = P*onehot(S[i]) # P is the projection matrix from one-hot vectors to embeddings
        [Y, h] = RNNstep(h, E, theta)
        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 4: LM training loss for a set of training sequences

```
# Give 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
```

1.2.6 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:

Predicting the next token

The 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 until the current token T_k. All of the other probabilities are composed from these incremental probabilities. So the obvious way to test the quality of your LM is to simply verify how accurately it computes this term, $P(T_{k+1}| < sos >, T_1, T_2, \cdots, T_k; \theta)$.

Say that 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 (ref. Figure 1), it would ideally assign the highest probability to the word "ago". (*Four score and seven* is part of a quote from the Gettysburg address, which begins with "<sos> Four score and seven years ago".) In other words, $P(ago| < sos > \ four\ score\ and\ seven; \theta)$ would ideally be assigned a high probability, if not a probability greater than $P(W| < sos > \ four\ score\ and\ seven; \theta)$ for any other word W.

So, a good test would be as follows:

For a large test set of partial token sequences (or word sequences) of the kind <sos> $\mathrm{T_1\ T_2\ \cdots\ T_k}$, 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 large 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 $\log Y[T_{k+1}]$.

Pseudocode 5: 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)
    h = initial_h # Initial value of hidden state
    # The partial sequence has symbols Tpartial[0] ... Tpartial[K}, where Tpartia[0] = <sos>
    for i = 0:K
        E = P*onehot(Tpartial[i])
        [Y, h] = RNNstep(h, E)
    end
    return Y
```

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 N words in the sequence (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. (Figure 5 illustrates this process.)

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).

Pseudocode 6: Completing a word sequence

```
# Given a partial word sequence Tpartial = <sos> w1 w2 ... wK, generates the next N words
function TokenSeq = Complete(Tpartial)
```

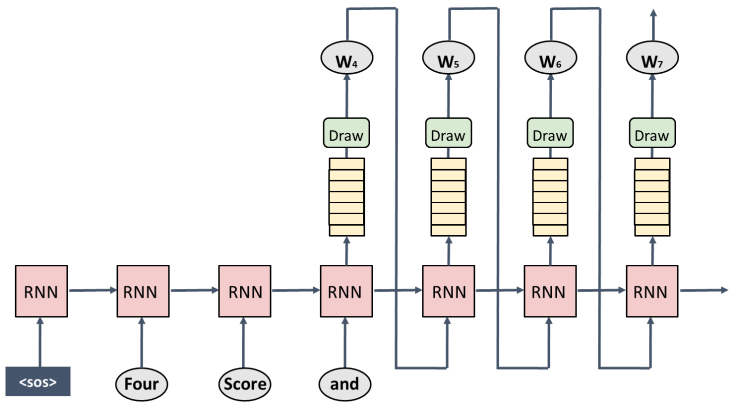


Figure 5: Given input sequence "<sos> four score and" the network must extend or complete the sequence. It does so by feeding in "<sos> four score and" in the first four time steps to the net. When the final word "and" is fed in, we begin drawing words from the output distribution for subsequent words w4, w5 etc.

```
    h = initial_h # Initial value of hidden state
    # First run the provided partial sequence through the net
    # The partial sequence has symbols Tpartial[0] ... Tpartial[K}, where Tpartia[0] = <sos>
    for i = 0:K
        E = P*onehot(Tpartial[i])
        [Y, h] = RNNstep(h, E)
    end

    TokenSeq[0:K] = Tpartial[0:K] # The first K tokens are just copied over
    # Now draw the rest
    i = K
    for j = 1:N
        E = P*onehot(TokenSeq[i]) # P is the projection matrix from one-hot vectors to embeddings
        [Y, h] = RNNstep(h, E)
        NewToken = DrawToken(Y) # PLEASE SEE BELOW FOR AN EXPLANATION OF THIS FUNCTION
        i = i + 1
        TokenSeq[i] = NewToken
        if Tokenseq[i] == <eos>: break
    end
    return TokenSeq
end
```

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 1.2.4).

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 a beam search. We defer the discussion of beam search to HW4P2, as you will not be using it here.

2 Dataset

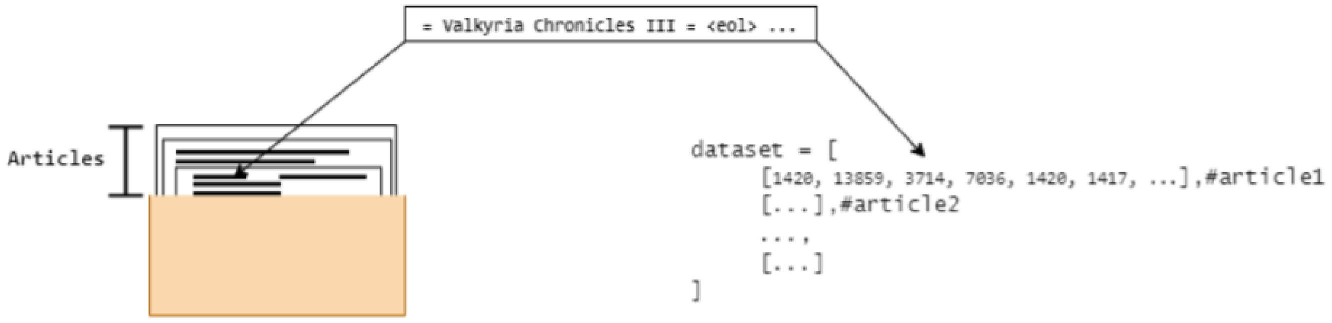
We will be training an RNN Language Model on the WikiText-2 Language Modeling Dataset, which has been pre-processed for HW4p1 and included in the template tarball:

- `vocab.npy`: a NumPy file containing the words in the vocabulary
- `vocab.csv`: a human-readable CSV file listing the vocabulary
- `wiki.train.npy`: a NumPy file containing training text

We also provide test files in the fixture directory which will make up the test cases for the prediction and generation functions you will implement. You don't need to worry about these files, test scripts that process them are already set up for you.

The train file contains an array of articles. Each article is an array of integers, corresponding to words in the vocabulary. There are 579 articles in the training set.

As an example, the first article in the training set contains 3803 integers. The first 6 integers of the first article are `[1420 13859 3714 7036 1420 1417]`. Looking up these integers in the vocabulary reveals the first line to be: `= Valkyria Chronicles III = <eol>`



```
dataset = [
    [1420, 13859, 3714, 7036, 1420, 1417, ...],#article1
    [...],#article2
    ...,
    [...]
]
```

Figure 6: A little visualization of the dataset

`<eol>` (end of line) is a special token added to the vocabulary to specify the end of a given line.

A preview of the raw dataset (Wikitext 2) is also available on **huggingface.co** via the following link: https://huggingface.co/datasets/wikitext/viewer/wikitext-2-raw-v1/train.

The dataset for this homework is provided as a collection of articles. So you may concatenate those articles to perform batching as described in the paper. It is advised to shuffle articles between epochs if you take this approach.

You will also receive a vocabulary file containing an array of strings. Each string is a word in the vocabulary. There are 33,278 vocabulary items.

Refer to Section 5 for more details about the testing procedure.

3 Problems

3.1 Masked Self-Attention (20 points) `[attention.py]`

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 task is designed for you to ensure you know how attention works in detail.

3.2 Language Modeling - Prediction (40 points) `[Jupyter Notebook]`

Complete the function `prediction` in class `LanguageModel` 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 token 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 word in the test data. Note that scores should be raw linear output values. Do not apply softmax activation to the scores you return.

3.3 Language Modeling - Generation (40 points) `[Jupyter Notebook]`

Complete the function `generate` in the class `LanguageModel` 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 word, this function should generate an entire sequence of words. 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 incorporating that in the input at the next time-step. You can call the function `predict` in class `LanguageModel` to get probabilities of the next word. Please refer to Recitation 7 (RNN Basics) 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 OpenAI. 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 OpenAI 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 RNNs/LSTMs 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.

Note: 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] https://huggingface.co/docs/transformers/perplexity

4.1 Masked Self-Attention `[attention.py]`

Complete the attention.py file using the equations provided in Section 1.1. Then, 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 Language Modeling `[Jupyter Notebook]`

4.2.1 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.**

4.2.2 Data Processing for Language Modeling

As mentioned in section 2, the train files for the language modeling part of this assignment contain an array of articles. Each article is in turn an array of integers, corresponding to words in the vocabulary. The arrays of "words" are token sequences that our language model aims to model the distribution of.

Recall from section 1.2.1 that we need to distinguish partial and complete word sequences through the use of start of sequence and end of sequence markers. <sos>, the start of sequence marker, indicates that a token sequence has just begun and the end of sequence marker, <eos>, indicates the end of a complete sequence.

We could use different tokens for <sos> and <eos>, or we can also use the same tag, say <eos>, to indicate both the start and end of sequences.

Before we proceed with further data loading tasks, add these markers to each text segment in the training corpus.

4.2.3 DataLoader

As explained earlier, your task is to build a language model that you will use to perform text generation, which is the task of predicting the next word given a sequence of preceding words. Thus, your samples should be formatted in a certain way so as to successfully train for that task. Lets see an example.

Article: *I eat a banana and an apple everyday including today.*

The following may be considered as sample hyperparameters that you can tune at your convenience.

- sequence_length = 3
- batch_size = 2

Then one batch consists of 2 of pairs of sequences which are pairs of (input sequence, target sequence).

A batch would have 2 training samples with sequence length 3. See how the first batch looks like in Figure 7.

Pytorch's `DataLoader` class allows you to format your samples the way you desire. To do so, you will have to overwrite the `__iter__` method (See starter notebook).

The `__iter__` method should:

1. Randomly shuffle all the articles from the WikiText-2 dataset.
2. Concatenate all articles in the dataset.
3. Divide the complete corpus into inputs and targets, where targets are shifted by a window of 1 word.
4. Resize your inputs and targets into batches of articles based on `batchsize`.

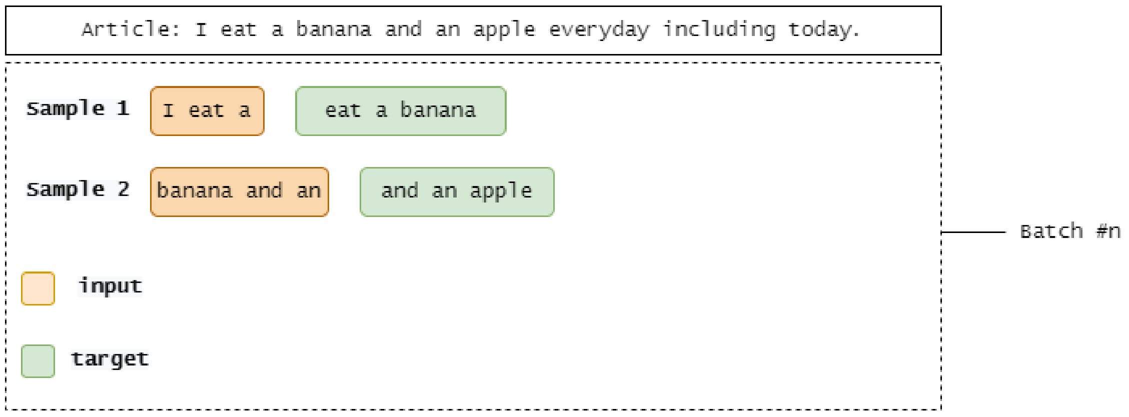


Figure 7: An example of batching and data loading

5. Run a loop that yields a tuple of `(input, target)` in every iteration based on sequence length, with both these components being in shape `(batchsize, seqlen)`. Look at iterators in python if this sounds unfamiliar.

4.2.4 Model Description

Your next task is to fill out the `LanguageModel` class. The `LanguageModel` class needs to have the following attributes, which are listed for you in the starter code:

1. `self.token_embedding` : As we mentioned in Section 1.2.3, we need to represent the tokens somehow; we need to compute token embeddings. Use the Embedding class from PyTorch to define `self.token_embedding`. This class is a simple lookup table that stores embeddings of a fixed dictionary and size, and takes in `num_embeddings`, the size of the dictionary of embeddings and `embedding_dim`, the size of each embedding vector. Think about the parameters that the `LanguageModel` takes in (you should add more parameters!) and define `num_embeddings` and `embedding_dim` accordingly. The embeddings are randomly initialized and update as your model trains. You can use the instance of the nn.Embedding class (`self.token_embedding`) to get the corresponding embedding for an input sequence `x`.

2. `self.lstm_cells`: LSTM cells are recurrent units that are used as the building blocks for the layers of our RNN. PyTorch offers an LSTMCell class (different from the LSTM in HW3P2) which you can use to define these LSTM cells. An LSTM layer is equivalent to an LSTMCell that is looped over T timesteps on a batch of sequences. We use LSTMCells instead of LSTMs to break down the operations involved in generating probability distributions over the vocabulary more atomically (this will feed into HW4P2, where you will use the attention mechanism to add context to generate your probability distributions over vocaubulary). We have pre-written a single LSTMCell layer in `self.lstm_cells` in the starter code, but you are encouraged to add more layers of LSTM cells to the sequential container. Please refer to the documentation in PyTorch to understand it's parameters. Once again, consider the parameters that the `LanguageModel` takes in (you should add more parameters!) and define `input_size` and `hidden_size` accordingly.

3. `self.token_probability`: At any timestep n, our model needs to compute the probability distribution for the (n+1)th token. The output from the `rnn_step` method is the next hidden state in our RNN, which will have some `hidden_size`. This tensor will need to be projected to dimension `vocab_size`, to obtain a "score" for each token in our vocabulary, which `self.token_probability` will do, after which we will typically perform a softmax operation to obtain a probability distribution over the vocabulary. `self.token_probability` is defined as a linear layer in our starter code, which takes in two parameters, `in_features`, the size of each input sample and `out_features`, the size of each output sample. You are **not** required to use a softmax function on the raw logits returned by `self.token_probability`, as after 3 HWs you would be knowing that PyTorch implicitly handles softmax within its `CrossEntropyLoss` function. Think about what dimensions we are projecting from and into and define `in_features` and `out_features` accordingly.

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

We will first discuss the `rnn_step` method. The `rnn_step` method iterates over our LSTMCell layers, defined in `self.lstm_cells`. At each iteration, our LSTMCell layers take in a token embedding at time step `t` and the hidden state at each layer and computes the next hidden state. The method returns the final hidden state output, and our final list of hidden states.

We now come to the `predict` and `generate` methods. 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 1.2.6 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. In this function, the model takes in one batch of inputs, `x`, from the dataloader, which has shape `[batchsize, seqlen]`.

We have defined two lists for you, `token_prob_distribution`, which will contain probability distributions for all timesteps, and `hidden_states_list`, to store the hidden states of our LSTM cells at timestep `t`.

We need to perform the following steps in `forward`:

1. The first step is to obtain the embedding of input `x`. Think about what attribute of the `LanguageModel` class stores our embeddings and how we would obtain these embeddings for the input `x`.
2. Following this, we iterate over our the tokens in our sequences, and perform the following steps. For each timestep `t` $=$ `0`, $\cdots$, `N`, where `N` is the sequence length of our batch of token sequences:
   (a) Obtain the token embeddings for the input `x` (all samples in our batch) at the time `t`.
   (b) We then need to return the final hidden state output, `rnn_out`, and our final list of hidden states, `hidden_states_list`. Which method allows us to do this?
   (c) As mentioned when previously, `rnn_out` is the final hidden state in our RNN. This needs to be projected to dimension `vocab_size`, to obtain a "score" for each token in our vocabulary. Therefore, this step will map the RNN output to the vocabulary's dimension and store it in `token_prob_dist_t`, the token probability distribution at time `t`. Think about what attribute of the `LanguageModel` allows us to perform this projection.
   (d) Finally, append `token_prob_dist_t` to a list of token probability distributions.
3. Return the list of token probability distributions and the list of hidden states.

4.2.5 Training and Regularization Techniques

Since the WikiText-2 data set for HW4p1 has been pre-processed, we can focus on breaking the text down into tokens for training the RNN. 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 Locked Dropout, Weight tying, Embedding Dropout, and also Data Augmentation. For structuring and training your model, we would like you to follow the protocols in Regularizing and Optimizing LSTM Language Models 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.

Locked Dropout

In standard dropout, a new binary dropout mask is sampled every time when the dropout function is called. For example, the input x_0 to an LSTM at timestep $t = 0$ receives a different dropout mask than the input x_1 fed to the same LSTM at $t = 1$.

A variant of this, locked dropout, available in torchnlp, also called variational dropout (Gal & Ghahramani, 2016), samples a binary dropout mask only once upon the first call. It then repeatedly uses that locked dropout mask for all inputs and outputs of the LSTM within the forward and backward pass.

A thing to notice is that each sample within the mini-batch should use a unique dropout mask. It will ensure the diversity in the elements dropped out. Please also think about on which dimension you should apply the mask.

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.

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 the standard LSTM language model.

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.

Activation Regularization (AR)

Activation Regularization is regularization performed on activation functions. Different from L2-Regularization, which is performed on weights, AR penalizes activation functions that are significantly larger than 0. It is defined as :

$$\alpha L_2(m \odot h_t)$$

where m is the dropout mask, L_2 is L2 norm, h_t is the output of the RNN at timestep t, and α is a scaling coefficient.

Temporal Activation Regularization (TAR)

Temporal Activation Regularization is a kind of slowness regularization for RNNs that penalizes the model from producing large changes in the hidden state. It is defined as:

$$\beta L_2(h_t - h_{t+1})$$

where L_2 is L2 norm, h_t is the output of the RNN at timestep t, and β is a scaling coefficient.

To implement AR and TAR, you need to calculate these regularization terms from layers you'd like to be regularized, and add any such term as an additional loss to the raw loss accumulated over the training samples. The AR and TAR term should only be applied to the output of the final RNN layer as opposed to being applied to all layers.

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 Masked Self-Attention [`attention.py`]

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 [`Jupyter Notebook`]

5.2.1 About Jupyter 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.

5.2.2 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 5.3 to get credit for the prediction task.**

The average model will likely take around 10 or more epochs, to achieve **a validation NLL below 4.6** on the prediction task. We have seen the **validation NLL to be around 0.5-0.7 lower** than that on the test set.

Performance reported in the paper[5] is 4.18, so you have room for error. However, with a good set of hyperparameters and some well chosen regularization techniques (mentioned earlier in subsection 4.2.5), you can get a validation NLL below 5.0 in no more than 3 epochs.

5.2.3 Language Modeling - Generation

For the generation task, the notebook has code to generate and save continuations for a test set comprising some incomplete sequences. There is also a code cell to evaluate the perplexity of your generations using an OpenAI LLM. Fill in the best runid, epoch number, and your OpenAI API key in this cell, and run it to get perplexity. **This perplexity should be below 1400 to get credit for the generation part.**

To obtain your API key, follow the following steps:

1. Go to `https://openai.com/blog/openai-api` and click on 'Log in' on top right.
2. You can then continue with a Google account (CMU ID or other) or signup with another email address and then log back in.
3. After logging in, you might see a choice between "ChatGPT" and "API". Click on API.
4. Under "Personal" (top right), go to "View API keys".
5. Click on "Create a new secret key" and copy the generated key into the generation evaluation cell.
6. You will need to add some credits to your OpenAI account by loading some money into your account. We estimate that the minimum permissible amount of $5 should be enough.
7. Under "Personal" (top right), go to "Manage account" and click on the "Billing" tab on the left pane.
8. If you've never added a payment method, click on "Add payment details" , followed by "Individual", and add your card details. Click on "Continue".
9. Enter the amount as $5 and preferably turn automatic recharge off. Click on "Continue" and then on "Confirm payment".
10. The credits should now reflect in the "Billing" tab.

[5] Regularizing and Optimizing LSTM Language Models reports a perplexity of 65.8, which translates to a $NLL = ln(65.8)$

5.3 Final Submission on Autolab

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.

5.4 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 the aforementioned paper.

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 !