Lab 6: Language Models

Javier Porras-Valenzuela and Alejandro Ribeiro

June 2, 2025

1 Language Sequences

We can think of language as a time series. In this interpretation, each word in a sentence corresponds to the equivalent of a different point in time and the words themselves represent different vectors of the time series. Consider as an example the first lines spoken by Miranda in The Tempest[1]:

> *If by your art, my dearest father, you have put the wild waters in this roar, allay them.*

We can, as we illustrate in Figure 1, parse this sentence as a time series. In this time series the first vector is $\mathbf{x}_0 =$ "If", the second vector is $\mathbf{x}_1 =$ "by", the third is $\mathbf{x}_2 =$ "your", and so on.

If we interpret language as a time series, we can use a transformer to predict the next word in a sequence as we did in Chapter 5. If we then execute this predictor recursively, we can use it to predict several words in a sequence. This is a strategy for generating language.

The first challenge to implement this strategy is how to represent words numerically. We do that with word embeddings as we discuss in the following section.

[1] In honor of the best Miranda.

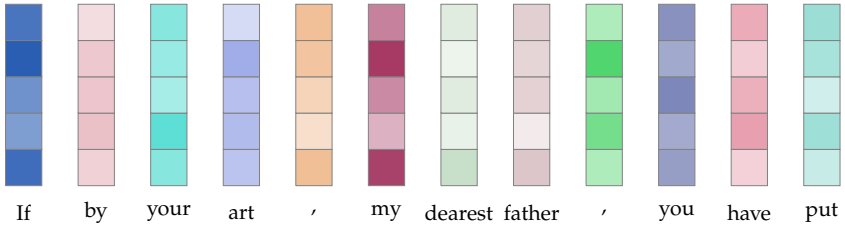

Figure 1. Language is a Time Series. We parse sentences as time series in which each word corresponds to the equivalent of a different point in time and the words themselves represent different vectors of the time series.

2 Word Embeddings

A simple approach is to encode words in the index of a long vector. Formally, suppose that we are given a collection of texts that collectively contain a total of c words. We then consider a set of c vectors $\mathbf{e}_i$ whose length is also c. These vector have all zero entries except for the ith entry which we set to 1,

$$(\mathbf{e}_i)_i = 1 \quad \text{and} \quad (\mathbf{e}_i)_j = 0 \;\text{ for }\; i \neq j. \tag{1}$$

We use the vector $\mathbf{e}_i$ to encode the ith word in the corpus.

In corpuses used in practice we have thousands of different words and anywhere between hundreds of thousands to trillions of sentences. In this lab we work with a subset of Shakespeare's plays which contains $c = 14,295$ different words and a total of 292,072 words in the corpus. But to illustrate ideas let us work with a corpus made up of just two quotes:

> *If by your art, my dearest father, you have put the wild waters in this roar, allay them.*
>
> *Sir, are not you my father?*

In this corpus we have a total of 24 different words including 3 punctuation marks. We therefore represent the words in the corpus with $c = 24$ vectors of length $c = 24$. Proceeding in the order of the sentence, the vector $\mathbf{e}_1 = [1, 0, \ldots, 0]$ represents the word "If," The vector

$\mathbf{e}_2 = [0, 1, 0, \ldots, 0]$ represents the word "by" and so on. The word "father" is the eight word that appears in the sentence and is therefore represented by the vector $\mathbf{e}_8$. This vector's value at index 8 is $(\mathbf{e}_8)_8 = 1$ and all of its other entries are zero.

When the same word appears again in the corpus, we encode it with the same vector. E.g., when the word "father" appears a second time we still encode it with the vector $\mathbf{e}_8$. This also happens with the comma (",") which appears three times and is encoded with the vector $\mathbf{e}_5$ in all three appearances and with the words "my" and "you" that appear twice and are encoded in the vectors $\mathbf{e}_6$ and $\mathbf{e}_9$. So encoded, our corpus becomes:

$$\mathbf{e}_1\ \mathbf{e}_2\ \mathbf{e}_3\ \mathbf{e}_4\ \mathbf{e}_5\ \mathbf{e}_6\ \mathbf{e}_7\ \mathbf{e}_8\ \mathbf{e}_5\ \mathbf{e}_9\ \mathbf{e}_{10}\ \mathbf{e}_{11}\ \mathbf{e}_{12}\ \mathbf{e}_{13}\ \mathbf{e}_{14}\ \mathbf{e}_{15}\ \mathbf{e}_{16}\ \mathbf{e}_{17}\ \mathbf{e}_5\ \mathbf{e}_{18}\ \mathbf{e}_{19}\ \mathbf{e}_{20}$$

$$\mathbf{e}_{21}\ \mathbf{e}_5\ \mathbf{e}_{22}\ \mathbf{e}_{23}\ \mathbf{e}_9\ \mathbf{e}_6\ \mathbf{e}_8\ \mathbf{e}_{24}$$

This is a defilement of Shakespeare's work. However, this representation of the corpus can be processed with numerical techniques.

Encoding language with these index vectors is not creative and does not work well. We discuss more interesting and useful word embeddings in the next section.

Task 1 Get the data for this lab from dsd.seas.upenn.edu/lab6 and load it to your environment. This is a text file containing around 40,000 lines of dialogue from Shakespeare's plays. Split the text into words, defined here to include punctuation marks and line breaks. We associate words with vectors $\mathbf{e}_i$ as in (1). Since it it would be wasteful to store vectors in which all but one entry is 0 we just store the index of the vector that represents each individual word. E.g., if "father" is represented by the index vector $\mathbf{e}_8$ we do not store $\mathbf{e}_8$ to represent this word. We just store the index $i = 8$.

Implement a function that turns a word into an index and the inverse function that turns an index into a word. We recommend that you use the code that we provide for this task. It is a somewhat cumbersome and not very enlightening activity.

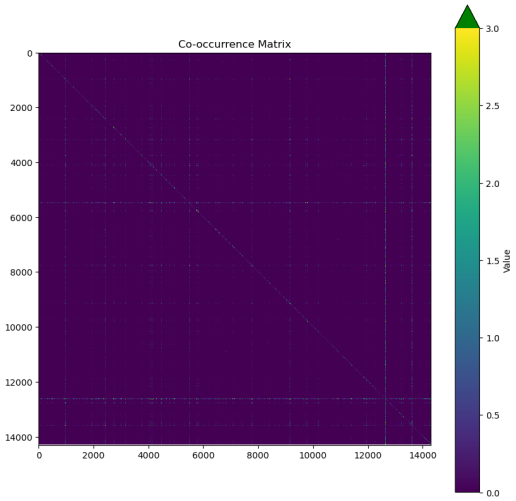

Figure 2. Heat map of the cooccurrence matrix. Pairs of words with a brighter color appear more frequently. You can see from the bright columns that some words cooccur with a lot of other words. For example, the words "the","and", "to", and special characters like ',' or '.' cooccur very frequently.

2.1 Cooccurrence Matrices

To create richer word embeddings we leverage the cooccurrence matrix $\mathbf{C}$. To construct this matrix we consider a window of length $W+1$ and scan the corpus for joint occurrences of words $\mathbf{e}_i$ and $\mathbf{e}_j$. The cooccurrence C_{ij} is the number of times that $\mathbf{e}_j$ appears in a window centered at $\mathbf{e}_i$. If we index the corpus by an index t and use $\mathbf{w}_t$ to represent the tth word in the corpus, we can write cooccurrences as,

$$C_{ij} = \sum_t \mathbb{I}(\mathbf{w}_t = \mathbf{e}_i) = \sum_{u=-W/2}^{u=W/2} \mathbb{I}(\mathbf{w}_u = \mathbf{e}_j), \tag{2}$$

where we assume that the window is even for simplicity. In (2) the first indicator function $\mathbb{I}(\mathbf{w}_t = \mathbf{e}_i) = 1$ only when the window is centered at $\mathbf{w}_t$ and $\mathbf{w}_t = \mathbf{e}_i$. The second indicator function $\mathbb{I}(\mathbf{w}_u = \mathbf{e}_j) = 1$ whenever the word $\mathbf{e}_j$ appears in the window centered at $\mathbf{w}_t$. Thus, the second sum counts the number of times that $\mathbf{e}_j$ appears centered in a window centered at $\mathbf{w}_t = \mathbf{e}_i$. The first sum is counting the number of times that $\mathbf{e}_i$ appears in the corpus.

The cooccurrence matrix $\mathbf{C}$ is relevant because related words tend to appear near each other and they also tend to appear next to words that indicate their relationships. In an extensive corpus we expect to find several cooccurrences of the words "birds" and "fly" indicating that these two words are related. We do not expect to see many cooccurrences of "dogs" and "fly" because dogs do not fly. We also expect to see cooccurrences of the words "bird" and "albatross" and of the words "bird" and "swallow," indicating that there is some relationship between "albatross" and "swallow."

We highlight that the cooccurrence matrix $\mathbf{C}$ is symmetric,

$$\mathbf{C} = \mathbf{C}^T \quad \Leftrightarrow \quad C_{ij} = C_{ji} \tag{3}$$

This is because whenever the word $\mathbf{e}_j$ appears in a window centered at an occurrence of the word $\mathbf{e}_i$, these two words are less than $W/2$ words apart. This implies that the word $\mathbf{e}_i$ must appear in a window centered at an occurrence of the word $\mathbf{e}_j$

Task 2 Compute the cooccurrence matrix for the Shakespeare corpus loaded in Task 1. Use a window of length $W = 10$.

2.2 Eigenvector Embeddings

A vector $\mathbf{v}_k$ is said to be an eigenvector of the cooccurrence matrix $\mathbf{C}$ if there exist a constant λ_k such that

$$\mathbf{C}\mathbf{v}_k = \lambda_k \mathbf{v}_k. \tag{4}$$

Eigenvectors are peculiar vectors because the matrix multiplication $\mathbf{C}\mathbf{e}$ yields a vectors that is, in general, quite different from $\mathbf{e}$. In the case of an eigenvector, the product $\mathbf{C}\mathbf{v}_k = \lambda_k \mathbf{v}_k$ is a simple scaling of $\mathbf{v}_k$. All of the components of $\mathbf{v}_k$ are multiplied by the same number.

It is known that the symmetric matrix $\mathbf{C} \in \mathbb{R}^{c \times c}$ has c distinct eigenvectors. It is customary to order the corresponding c eigenvalues from largest to smallest so that $\lambda_k \geq \lambda_\ell$ when $k < \ell$. Since eigenvector $\mathbf{v}_k$ is associated with eigenvalue λ_k, the eigenvectors inherit this order. When $k < \ell$ eigenvector $\mathbf{v}_k$ is associated with an eigenvalue that is not smaller than the eigenvalue associated with eigenvector $\mathbf{v}_\ell$ – it is most often larger. We

will say that eigenvector $\mathbf{v}_k$ is not smaller than eigenvector $\mathbf{v}_\ell$ or that $\mathbf{v}_k$ is larger than $\mathbf{v}_\ell$ if $\lambda_k > \lambda_\ell$.

It is also customary to group eigenvectors in the eigenvector matrix

$$\mathbf{V} = [\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_c], \tag{5}$$

in which Column k registers the value of Eigenvector $\mathbf{v}_k$. The eigenvector matrix is an $n \times n$ matrix. It has c columns representing c distinct eigenvectors which have c rows each.

We consider now a number $n \leq c$ and define the *dimensionality reduction* matrix $\mathbf{V}_n$ grouping the first n eigenvectors of $\mathbf{C}$,

$$\mathbf{V}_n = [\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n]. \tag{6}$$

This is a tall matrix because it has c rows but only n columns. These columns coincide with the first n columns of $\mathbf{V}$. Instead of storing all eigenvectors, we are storing only the n largest eigenvectors of $\mathbf{C}$.

We use the dimensionality reduction matrix $\mathbf{V}_n$ to construct representations of vectors $\mathbf{e} \in \mathbb{R}^c$ in a space of dimensionality n. These representations are

$$\mathbf{x} = \mathbf{V}_n^T \mathbf{e}. \tag{7}$$

We say that this is a dimensionality reduction because $\mathbf{x} \in \mathbb{R}^n$ is a vector with n components, which is (much) smaller than the number of components c of the vector $\mathbf{e} \in \mathbb{R}^c$.

We use dimensionality reduction to compute word embeddings. Given the collection of words $\mathbf{e}_i$, we transform them into the collection of embeddings

$$\mathbf{x}_i = \mathbf{V}_n^T \mathbf{e}_i. \tag{8}$$

Representations $\mathbf{x}_i$ are preferable to representations $\mathbf{e}_i$ because they have smaller dimensionality. They also turn out to capture some semantic properties in the sense that vectors $\mathbf{x}_i$ and $\mathbf{x}_j$ that are close represent similar words. This is different from the index embeddings $\mathbf{e}_i$ in which comparisons between different vectors $\mathbf{e}_i$ and $\mathbf{e}_j$ have no meaning.

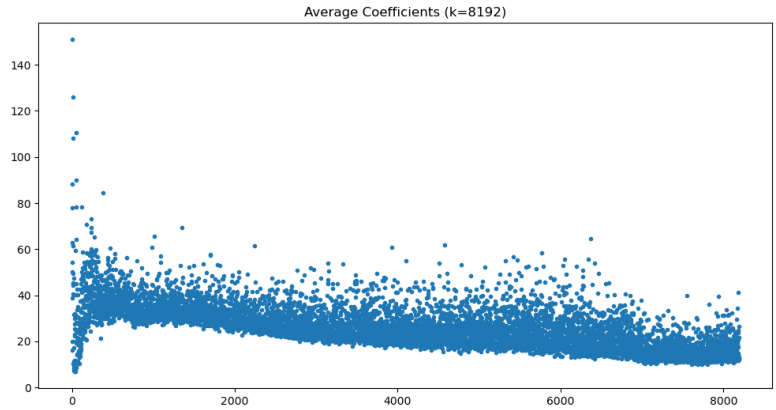

Figure 3. Principal component analysis (PCA) transform coefficients of word vectors. For word vectors $\mathbf{e}_i$, PCA transform coefficients with large index k are small. This means that we can capture most of the information contained in a word vector $\mathbf{e}_i$ in its reduced dimension representation $\mathbf{x}_i = \mathbf{V}_n^T \mathbf{e}_i$ [cf. (8)].

Task 3 Compute the first $n = 256$ eigenvectors of the cooccurrence matrix computed in Task 2. Use these eigenvectors to compute the eigenvector embeddings of all of the c words in the corpus loaded in Task 1. Store the corpus using these eigenvector embeddings. This is the time series with which we will work in subsequent tasks.

2.3 Principal Component Analysis

The principal component analysis (PCA) transform of a vector $\mathbf{e}$ is its projection in the eigenvector space of $\mathbf{C}$,

$$\mathbf{y} \;=\; \mathbf{V}^T \mathbf{e}. \tag{9}$$

This is similar to the dimensionality reduction operation in (7) except that we are using all c eigenvectors instead of the largest n.

The PCA representation in (9) has the important property that it can be undone by multiplication with the eigenvector matrix. I.e, given the PCA transform $\mathbf{y}$ in (9), we can recover the original vector $\mathbf{e}$ as

$$\mathbf{e} \;=\; \mathbf{V}\mathbf{y}. \tag{10}$$

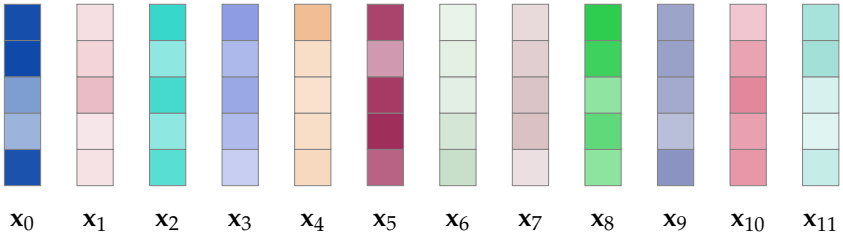

Figure 4. Language time series with eigenvector encodings. The $\mathbf{x}_i$ vectors in this time series are eigenvector encodings of the words in Figure 1. These vectors represent words in a manner that can be processed with linear algebra operations.

The combination of (9) and (10) indicates that $\mathbf{e}$ and $\mathbf{y}$ are equivalent representations of the same information. Given $\mathbf{e}$ we can compute $\mathbf{y}$ and given $\mathbf{y}$ we can compute $\mathbf{e}$.

The same is not true of the dimensionality reduction transformation in (7). When going from $\mathbf{e}$ to $\mathbf{x}$ we loose information precisely because we are reducing dimensionality. In this context it is interesting to implement the dimensionality recovery operation,

$$\tilde{\mathbf{e}} \;=\; \mathbf{V}_n\mathbf{x} \;=\; \mathbf{V}_n\Big(\,\mathbf{V}_n^T\mathbf{e}\,\Big), \tag{11}$$

and ask the question of how close the recovered vector $\mathbf{e}$ is to the original $\tilde{\mathbf{e}}$. The answer is that for word vectors $\mathbf{e}_i$ the error is small. That is, for most word vectors,

$$\tilde{\mathbf{e}}_i \;=\; \mathbf{V}_n\mathbf{x}_i \;=\; \mathbf{V}_n\Big(\,\mathbf{V}_n^T\mathbf{e}_i\,\Big) \;\approx\; \mathbf{e}_i. \tag{12}$$

We have a good theoretical understanding of why this happens. In the context of this lab it suffices for us to verify that this is true empirically. In Figure 3 we illustrate the values of the PCA components y_{ik} of some representative word vectors $\mathbf{e}_i$. These PCA components are the entries of the PCA transform vectors $\mathbf{y}_i \;=\; \mathbf{V}^T\mathbf{e}_i$ [cf. (9)] using the eigenvectors of the cooccurrence matrix $\mathbf{C}$ of the Shakespeare corpus that we use in this lab. We see that the value of PCA components y_{ik} decreses with increasing k.

3 Language Transformers

Using the eigenvector embeddings of Section 2.2 the language time series of Figure 1 becomes the time series of Figure 4. Points in time in the former are associated with words. Points in time in the latter are associated with vectors $\mathbf{x}_t \in \mathbb{R}^n$. These vectors represent words in a manner that can be processed with linear algebra operations. This is not different from the time series we encountered in Chapter 5. That the time series represents language is irrelevant for its processing.

We use here a softmax attention transformer with multiple heads to process language sequences. For reference, a transformer with multiple heads is defined by the recursion,

$$\mathbf{A}_\ell^h = \text{sm}\Big((\mathbf{Q}_\ell^h \mathbf{X}_{\ell-1})^T (\mathbf{K}_\ell^h \mathbf{X}_{\ell-1}) \Big), \tag{13}$$

$$\mathbf{Y}_\ell^h = \mathbf{W}_\ell^{hT} \mathbf{V}_\ell^h \mathbf{X}_{\ell-1} \mathbf{A}_\ell^{hT}, \tag{14}$$

$$\mathbf{X}_\ell = \mathbf{X}_{\ell-1} + \sigma\Big(\sum_{h=1}^{H} \mathbf{Y}_\ell^h \Big). \tag{15}$$

The input to the transformer is a sequence of T eigenvector word embeddings $\mathbf{X}_0 = \mathbf{X}$ and its output $\mathbf{X}_L = \Phi(\mathbf{X}, \mathcal{A})$ is another sequence of T eigenvector word embeddings. The trainable tensor $\mathcal{A} = \{\mathbf{Q}_\ell^h, \mathbf{K}_\ell^h, \mathbf{V}_\ell^h, \mathbf{W}_\ell^h\}$ contains all of the query, key, value, and dimension recovery matrices of all heads and layers. We use the output sequence to predict the next word, $\mathbf{x}_T$, in the sequence in Section 4.

Equation (13) is the computation of softmax attention coefficients $\mathbf{A}_\ell^h$ for Layer ℓ and Head h. We use these attention coefficients to create contextual representations $\mathbf{Y}_\ell^h$ in (14). The output of Layer ℓ is computed in (15) by summing all heads and passing the output through a nonlinear operation. We also add the skip connection $\mathbf{X}_{\ell-1}$ to the output of Layer ℓ of the transformer.

Recall that in (13) and (14) we create the intermediate representations $\mathbf{Q}_\ell^h \mathbf{X}_{\ell-1}$ (queries), $\mathbf{K}_\ell^h \mathbf{X}_{\ell-1}$ (keys), and $\mathbf{V}_\ell^h \mathbf{X}_{\ell-1}$ (values) which are of dimension $m \ll n$. In this lab and in language models in general the reduction of dimension is aggressive. We have here that $n = 256$ at the input and choose $m = 32$ for intermediate representations.

Task 4 Code a Pytorch module to implement a the language transformer as specified by (13)-(15). This transformer takes sequences of length T and dimension n as inputs and produces sequences of length T and dimension n as outputs. Make the number of layers L and the number of heads H parameters of the transformer. Queries, keys and values are of dimension m, which is also a parameter of the transformer. Use relu nonlinearities at each layer.

This is the same transformer of Lab 5. It is a copy and paste task. That the time series represents language is irrelevant.

4 Next Word Prediction

To predict word $\mathbf{x}_T$ we read the output $\mathbf{X}_L = \Phi(\mathbf{X}, \mathcal{A})$ of the transformer. A possible approach is to take the average across time. To set up this readout strategy let $\mathbf{X}_u$ denote a sequence of T words – in the form of eigenvector embeddings – starting at time u,

$$\mathbf{X}_u \;=\; \big[\, \mathbf{x}_u, \mathbf{x}_{u+1}, \ldots, \mathbf{x}_{T+u-1} \,\big] \;=\; \mathbf{x}_{u:u+T-1}. \tag{16}$$

This is a recorded history of the language sequence. Our goal is to predict the next word $\mathbf{x}_{u+T}$ using this recorded history. We do that using the average of the output of the transformer in (13)-(15),

$$\hat{\mathbf{x}}_{u+T} = \Big[\, \Phi(\mathbf{X}_u, \mathcal{A}) \,\Big]\mathbf{1}. \tag{17}$$

We then train the tensor $\mathcal{A} = \{\mathbf{Q}_\ell^h, \mathbf{K}_\ell^h, \mathbf{V}_\ell^h, \mathbf{W}_\ell^h\}$ to maximize prediction accuracy over the corpus. Utilizing a mean squared loss (MSE), the prediction task reduces to

$$\mathcal{A}^* \;=\; \underset{\mathcal{A}}{\text{argmin}}\; \frac{1}{C} \sum_{u=0}^{C-1} \Big\| \,\Phi(\,\mathbf{X}_u,\, \mathcal{A}\,)\mathbf{1} \,-\, \mathbf{x}_{u+T} \,\Big\|^2. \tag{18}$$

In (18) we compare the estimate $\hat{\mathbf{x}}_{u+T}$ read out from the transformer's output as per (17) with the true next word $\mathbf{x}_{u+T}$. We average the resulting MSE loss over the corpus and seek the tensor $\mathcal{A}^*$ that minimizes it. Notice that to simplify notation we sum over the whole corpus. In reality, we can't predict the last T words because we are using histories $\mathbf{X}_u$ of length T. In fact, we have several other limitations in the construction of the

training dataset. We may, e.g., want to avoid running over the end of a play, or the end of an act. We choose to ignore these practicalities as they have little effect.

Task 5 Split the corpus loaded in Task 3 into a training set containing 90% of the total number of words and a test set containing 10% of the words. Recall that this is a time series of word embeddings. Use this training set to train a transformer that predicts the next word embedding using the loss in (18). Use $T = 64$ for the length of the history $\mathbf{X}_u$. Transformer parameters are your choice. If you need a recommendation, use $L = 6$ $H = 8$ and $m = 32$.

Evaluate the test MSE and compare it to the train MSE. Both of these MSE values indicate good prediction. However, this does not mean that we are making good predictions of the next word in the sequence. Explain.

4.1 Probability Readout

The predictions $\hat{\mathbf{x}}_{u+T}$ in (17) may have a small MSE when compared to the observed words $\mathbf{x}_{u+T}$ but they are not a good strategy for estimating the next word. This is because $\hat{\mathbf{x}}_T$ need not be a valid word. Indeed, it most likely will not be a valid word.

Word $\mathbf{e}_i$ is represented by the eigenvector encoding $\mathbf{x}_i = \mathbf{V}_n^T \mathbf{e}_i$ as stated in (8). Since there are a total of c words in our corpus, there are a total of c vectors $\mathbf{x}_i$ that represent valid words. The vectors at the output of the transformer are most unlikely to be one of these vectors and the estimate $\hat{\mathbf{x}}_T$ in (17) is just as unlikely unless we manage to drive the train and test MSEs to zero.

To solve this problem we must force the readout to be a valid word. We do that with readout layer whose output is a vector of $\tilde{n}$ probabilities for each of the $\tilde{n}$ words in the corpus. This readout layer is a softmax applied to the output of a fully connected layer that acts on the output of the transformer at time $T-1$,

$$\boldsymbol{\pi}(\mathbf{X}) = \text{sm}\Big[\, \mathbf{A} \Big(\Phi(\mathbf{X}, \mathcal{A}) \Big)_{T-1} \Big]. \tag{19}$$

The matrix $\mathbf{A}$ is a trainable parameter with n columns and c rows. After applying the softmax normalization the entries of the output $\boldsymbol{\pi}(\mathbf{X})$ add

up to one and can be interpreted as as set of probabilities that dictate the likelihood of the next word in the sequence. The ith entry $\boldsymbol{\pi}_i(\mathbf{X})$ is the predicted probability that the next word is $\mathbf{e}_i$.

We refer to the probabilities in (19) as a policy. To train this policy we minimize the crossentropy loss between the true word at time $u+T$ and the probabilities $\boldsymbol{\pi}(\mathbf{X})$,

$$\mathcal{A}^*, \mathbf{A}^* \;=\; \underset{\mathcal{A},\,\mathbf{A}}{\operatorname{argmin}} \; \frac{1}{C} \sum_{u=0}^{C-1} \big(\mathbf{e}_{u+T}\big)^T \big(\log \boldsymbol{\pi}(\mathbf{X}_u)\big). \tag{20}$$

Notice that in (20) the vector $\mathbf{e}_{u+T}$ is the index encoding of the word at time $u+T$. This is a vector with all zeros except that it has a 1 at the entry that corresponds to the index of the word that is observed at time $u+T$. Is is therefore a valid probability index that we can incorporate into a crossentropy comparison.

Further notice that the optimization is joint over the trainable parameters $\mathcal{A}$ of the transformer and the readout matrix $\mathbf{A}$. These two parameters are implicit in (20). They appear because $\boldsymbol{\pi}(\mathbf{X}_u)$ depends on $\mathbf{A}$ and $\mathcal{A}$. In the hope that it is revealing to make this dependance explicit we instantiate $\mathbf{X} = \mathbf{X}_u$ in (19) and substitute the result in (20) to write

$$\mathcal{A}^*, \mathbf{A}^* \;=\; \underset{\mathcal{A},\,\mathbf{A}}{\operatorname{argmin}} \; \frac{1}{C} \sum_{u=0}^{C-1} \Big[\mathbf{e}_{u+T}\Big]^T \Big[\log \operatorname{sm}\Big[\mathbf{A} \operatorname{vec}\big(\Phi(\mathbf{X}_u, \mathcal{A}) \big) \Big] \Big]. \tag{21}$$

We solve this empirical risk minimization (ERM) problem to predict the next word in a sequence of text. This prediction is based on observing a history of length T that is processed by a transformer [cf. (13)-(15)] with a probability readout layer [cf. (19)]. Different from the readout strategy in (17) and the training procedure in (18), the ERM problem in (21) produces parameters $\mathcal{A}^*$ and $\mathbf{A}^*$ that map directly to predictions of actual words

Task 6 Modify the transformer of Task 4 to add the readout layer in (19).

Task 7 Split the corpuses loaded in Task 1 and Task 3 into a training set containing 90% of the total number of words and a test set containing 10% of the words. Recall that these two are equivalent time series except that the information is encoded differently. In Task 1 we store words using index encodings and in Task 3 we store words using eigenvector

embeddings. We are loading both here because the eigenvector encodings are the input to the transformer and the index encodings are needed for the crossentropy comparison in (21). Make sure that time indexes match in your data.

Use the training set to train a transformer that predicts next word probabilities using the transformer with readout of Task 6. Use $T = 64$ for the length of the history $\mathbf{X}_u$. Transformer parameters are your choice. If you need a recommendation, use $L = 6$ $H = 8$ and $m = 32$.

Evaluate the crossentropy loss in the test set and compare it to the crossentropy loss in the training set.

4.2 Model Sampling

After solving the ERM problem in (21) we have trained values $\mathcal{A}^*$ for the transformer and $\mathbf{A}^*$ for the probability readout layer. With these trained values we can execute (19) for any given text sequence $\mathbf{X}$ of length T. The result is the (optimal) vector of probabilities

$$\boldsymbol{\pi}^*(\mathbf{X}) = \text{sm}\Big[\, \mathbf{A}^* \,\text{vec}\big(\Phi(\mathbf{X}, \mathcal{A}^*)\big) \,\Big]. \tag{22}$$

This is not yet a word. It is a vector of probabilities that assigns probabilities to each of the c words in the corpus. To generate a word we need to implement a *sampling* strategy.

Let us denote as $\pi^*(\mathbf{e}_i|\mathbf{X})$ the probability of choosing word $\mathbf{e}_i$. This is the ith entry of the vector of probabilities $\boldsymbol{\pi}$. A possible sampling strategy is to sample the word $\mathbf{e}_i$ with the highest probability,

$$\hat{\mathbf{e}} = \underset{\mathbf{e}_i}{\text{argmax}}\, \pi^*(\mathbf{e}_i|\mathbf{X}) \tag{23}$$

Alternatively, we can sample predictions randomly by choosing different words according to their corresponding probabilities. We write

$$\hat{\mathbf{e}} = \mathbf{e}_i \sim \pi^*(\mathbf{e}_i|\mathbf{X}) \tag{24}$$

to signify that we choose $\hat{\mathbf{e}} = \mathbf{e}_i$ with probability $\pi^*(\mathbf{e}_i|\mathbf{X})$.

Sampling according to the largest probability [cf. (23)] is a good strategy if we want to actually predict the next word in the sequence. Sampling

randomly according to word probabilities [cf (24)] is a better strategy for generating text. Random sampling is a better imitation of the natural variability of human language. We will use random sampling.

Task 8 Given trained parameters $\mathcal{A}^*$ and $\mathbf{A}^*$ implement: (a) A transformer with parameters $\mathcal{A}^*$ that takes language sequences $\mathbf{X}$ of length T as inputs. (b) A readout layer that postprocesses the output of the transformer to yield a vector of probabilities $\boldsymbol{\pi}^*(\mathbf{X})$. (c) A sampler that takes probabilities $\boldsymbol{\pi}^*(\mathbf{X})$ as inputs and returns words $\hat{\mathbf{e}}$ sampled according to (24).

The transformer and readout implementations are just instances of the transformer and readout modules of Tasks 4 and Task 6. The only new piece here is the sampler.

Try your sampler for a few input sequences.

5 Language Generation

In Section 4 we adopted a transformer to predict the next word of a sequence of length T. We adapt this model to language generation with a rolling execution.

Begin with a language sequence entered by a user, which we call a prompt. From the prompt we construct a time series $\mathbf{X}_0$ with the eigenvector encodings of its words

$$\mathbf{X}_0 = [\mathbf{x}_0, \ldots, \mathbf{x}_{T-1}]. \tag{25}$$

We assume, for simplicity, that this prompt has length T. Using this prompt we predict the next word in the sequence using the policy $\boldsymbol{\pi}^*$,

$$\mathbf{x}_T \sim \boldsymbol{\pi}^*(\mathbf{X}_0). \tag{26}$$

Although the input $\mathbf{x}_T$ has been *generated* by the policy $\boldsymbol{\pi}^*$, we reinterpret it as a *given* word. We then roll the prompt backward and append the generated word $\mathbf{x}_T$ to construct the series

$$\mathbf{X}_1 = [\mathbf{x}_1, \ldots, \mathbf{x}_{T-1}, \mathbf{x}_T]. \tag{27}$$

In this sequence the first $T-1$ entries are part of the user prompt. The last one, $\mathbf{x}_T$, has been generated. We ignore this distinction and proceed to estimate word $T+1$ as

$$\mathbf{x}_{T+1} \sim \boldsymbol{\pi}^*(\mathbf{X}_1) \tag{28}$$

We then proceed to append this generated word to the time series in (27) and roll the series backward. This procedure yields the time series,

$$\mathbf{X}_2 = [\mathbf{x}_2, \ldots, \mathbf{x}_{T-1}, \mathbf{x}_T, \mathbf{x}_{T+1}] \tag{29}$$

In this time series we have the last $T-2$ words of the user prompt and two words that have been generated by policy $\boldsymbol{\pi}^*$. These are the words $\mathbf{x}_T$ and $\mathbf{x}_{T+1}$ generated in (26) and (28). We, again, ignore this distinction and generate the next word as,

$$\mathbf{x}_{T+2} \sim \boldsymbol{\pi}^*(\mathbf{X}_2) \tag{30}$$

We append word $\mathbf{x}_{T+2}$ to the time series, roll the time series backwards, and use the updated series to predict the next word in the sequence. In general, at generative step u we take as an input the time series

$$\mathbf{X}_u = [\mathbf{x}_u, \ldots, \mathbf{x}_{T-1+u}], \tag{31}$$

in which the last u samples have been generated – it can be that all of the samples are generated if $u \geq T$. From this time series we generate the word in position $T+u$ as,

$$\mathbf{x}_{T+u} \sim \boldsymbol{\pi}^*(\mathbf{X}_u). \tag{32}$$

The output of the generative language model the string of text $[\mathbf{x}_T, \ldots, \mathbf{x}_{T_{\text{max}}}]$ where T_{max} is a prespecified limit for the length of the language sequence to be generated. Of course, rather than returning the eigenvector embeddings $[\mathbf{x}_T, \ldots, \mathbf{x}_{T_{\text{max}}}]$ we return the sequence of corresponding words.

Task 9 Implement the generative language model as specified by the recursion (31)-(32). Take prompts of length $T = 64$ as inputs and generate language sequences of length $T_{\text{max}} = 500$. To make this task more interesting, modify your implementation to accept prompts of any length $T' \leq T$. This is not difficult because absent words in the prompt can be set to zero.

Try your generative model for some prompts.

6 Positional Encoding

The output of a transformer is equivariant to a permutation of the entries of the time series. If we exchange the positions of $\mathbf{x}_t$ and $\mathbf{x}_u$ the output of the transformer is the the same except that the corresponding outputs of the transformer also exchange places. To be precise, let $\mathbf{Q}$ be a permutation matrix that exchanges the position of $\mathbf{x}_t$ and $\mathbf{x}_u$ in the time series $\mathbf{X}$. Then, the time series $\mathbf{X}$ and $\mathbf{X} \times \mathbf{Q}$ are the same except for a switch of places between $\mathbf{x}_t$ and $\mathbf{x}_u$ and the outputs of the transformer satisfy

$$\Phi\Big(\mathbf{X} \times \mathbf{Q}, \mathcal{A}\Big) \;=\; \Phi\Big(\mathbf{X}, \mathcal{A}\Big) \times \mathbf{Q}. \tag{33}$$

I.e., the transformer outputs are the same except that $[\Phi(\mathbf{X}, \mathcal{A})]_t$ and $[\Phi(\mathbf{X}, \mathcal{A})]_u$ exchange places. The relationship in (35) is true not only for a exchange of places between entries but for any permutation matrix $\mathbf{Q}$.

Positional encoding is a strategy to break permutation symmetry so that words can have different effects depending on their positions. To that end we define the positional encoding time series $\mathbf{P} \in \mathbb{R}^{n \times T}$ and it to the input time series,

$$\tilde{\mathbf{X}} = \mathbf{X} + \mathbf{P}. \tag{34}$$

The entries $\mathbf{p}_t$ of the positional encoding series $\mathbf{P}$ are independent of the input times series $\mathbf{X}$ and depend solely on their time index t. Thus, the time series $\tilde{\mathbf{X}}$ modifies the input data $\mathbf{X}$ to codify the position of the different entries of the time series.

This breaks the permutation equivariance of the transformer. If we exchange the positions of $\mathbf{x}_t$ and $\mathbf{x}_u$ the outputs of the transformer are *not* necessarily the same because the positional encodings $\mathbf{p}_t$ and $\mathbf{p}_u$ do not exchange places. I.e., in general,

$$\Phi\Big(\mathbf{X} \times \mathbf{Q} + \mathbf{P}, \mathcal{A}\Big) \;\not\equiv\; \Phi\Big(\mathbf{X} \times \mathbf{Q}, \mathcal{A}\Big) \times \mathbf{Q}. \tag{35}$$

The symmetry breaks the input time series $\mathbf{X}$ is permuted but the positional enccoding series $\mathbf{P}$ is not.

6.1 Learnable and Designed Positional Encodings

There are two standard approaches to positional encoding in language models. The first approach is to make $\mathbf{P}$ a learnable parameter.

Alternatively, we can use oscillations to design positional encodings. For a time series made up of vectors $\mathbf{x}_t$ with n entries we define $n/2$ frequencies α_i. For each of these frequencies we define a times series $\mathbf{P}$ in which the values for time t and index i are given by

$$\begin{aligned} p_{ti} &= \cos\Big(2\pi\,\alpha_{(i+1)/2}\,(t/T)\Big), \qquad i \quad \text{odd}, \\ p_{ti} &= \sin\Big(2\pi\,\alpha_{i/2} \qquad (t/T)\Big), \qquad i \ \text{ even}. \end{aligned} \tag{36}$$

As per (36), positional encoding includes sines and cosines of different frequencies in different rows of the positional encoding time series. Odd rows of $\mathbf{P}$ are cosines of frequency $\alpha_{(i+1)/2}$. Even rows of $\mathbf{P}$ are sines of frequency $\alpha_{i/2}$. The use of sines and cosines in (36) is motivated by the Fourier basis, which has intimate connections with convolution. This is a story for another day.

The positional encoding in (36) is the most common in language models. Learned positional encodings are simpler and not innefective.

Task 10 Modify the transformer of Task 6 to incorporate positional encoding. Implement the positional encoding $\mathbf{P}$ as a learnable parameter.

7 Practical Considerations

The transformer in Task 10 contains all of the fundamental aspects of a language model. It includes a word embedding to translate words into meaningful numerical representations (Section 2), a transformer to process the embedded sequence (Section 3), and a readout layer that takes the output of the transformer and produces a probabilistic policy (Section 4). To these three components we add positional encoding to break permutation symmetry and allow words to have different effects on the output depending on their position in the prompt (Section 6). This transformer

can be trained and then used to generate language sequences using the method in Section 5. The results are middling.

We describe here two small modifications that improve performance further. They are layer normalization (Section 7.1) and future masking (Section 7.2).

7.1 Layer Normalization

In order to improve training stability and convergence, one common implementation trick is to normalize the output vectors of a layer to have zero mean and unit variance. This is commonly referred to as *layer normalization*. To simplify notation let $\mathbf{X}$ stand in for the output of Layer ℓ. The normalized output $\hat{\mathbf{X}}$ of layer ℓ is computed as:

$$\hat{\mathbf{X}}_{ti} = \gamma_t \cdot \frac{\mathbf{X}_{ti} - \mu_t}{\sqrt{\sigma_t^2 + \epsilon}} + \beta_t \tag{37}$$

Here, $\epsilon > 0$ is a small number to avoid dividing by zero, μ and σ^2 are the row-wise mean and variance of the elements $\mathbf{X}_{ij}$ at layer ℓ:

$$\mu_t = \frac{1}{n} \sum_{i=1}^{n} X_{ti} \tag{38}$$

$$\sigma_t^2 = \frac{1}{n} \sum_{i=1}^{n} (\mathbf{X}_{ti} - \mu_t)^2 \tag{39}$$

The learnable parameters γ_i and β_i play the role of recovering the mean and the variance. This might seem like we didn't do anything, but now the learnable parameters do not depend on the computation of X. This results in more stable training. By normalizing each hidden vector, layer normalization helps to mitigate internal covariate shift and ensures more stable gradients during training.

Task 11 Modify the transformer of Task 10 to incorporate layer normalization at the output of each transformer layer. Use the PyTorch function nn.LayerNorm.

7.2 Future Masking

In Equation (13), we have attention coefficients for each pair of words in a sequence. This means that our model can learn to have $\mathbf{A}_{tu}$ with nonzero attention even if the word $\mathbf{w}_t$ is ahead of the word $\mathbf{w}_u$. This is undesirable for word generation as it is reasonable that attention coefficients should focus on past words only. This is a better match to the goal of predicting the next word in a sequence. We use future masking to ensure this.

Recall then the definition of the attention matrix in (13) and write it as $\mathbf{A}$ to simplify notation. The tth rows of the attention matrix $\mathbf{a}_t$ contains the attention coefficients for time index t. We want to modify attention so that the weight is zero for all the words beyond t,

$$\mathbf{a}_t = [a_{t0}, a_{t1}, \ldots, a_{tt}, 0, \ldots, 0]. \tag{40}$$

To accomplish this recall the definition of the linear attention matrix

$$\mathbf{B} = (\mathbf{QX})^T(\mathbf{KX}) \tag{41}$$

The softmax of $\mathbf{b}_t$, Row t of the matrix $\mathbf{B}$, is our standard definition of the attention coefficient $\mathbf{a}_t$. To obtain an attention coefficient with the form in (40) we compute the softmax over the first t entries of $\mathbf{b}_t$ only,

$$\mathbf{a}_t = \Big[\mathrm{sm}\Big(\big[b_{t0}, b_{t1}, \ldots, b_{tt} \big] \Big), 0, \ldots, 0 \Big]. \tag{42}$$

The attention coefficients in (42) add up to one and have zero mass on times $u > t$.

Task 12 Modify the transformer of Task 11 to incorporate future masking in the attention layers.

Task 13 Repeat the training in Task 7 using the transformer in Task 12.

Task 14 Repeat the generative excersise in Task 9 using the transformer trained in Task 13.

8 Report

Do not take much time to prepare a lab report. We do not want you to report your code and we don't want you to report your work. Just give us answers to questions we ask. Specifically give us the following:

Question	**Report deliverable**
Task 1	Word histogram
Task 2	Do not report
Task 3	Do not report
Task 4	Do not report
Task 5	Train and test MSE. Paragraph explaining why word predictions are not good
Task 6	Do not report
Task 7	Train and test crossentropy losses
Task 8	Three examples of your favorite pairs of sequences and predictions. You will get 0 points in the lab if you report our pairs.
Task 9	One example of your favorite generated sequence. You will get 0 points in the lab if you report our sequence.
Task 10	Do not report
Task 11	Do not report
Task 12	Do not report
Task 13	Do not report
Task 14	One example of your favorite generated sequence. You will get 0 points in the lab if you report our sequence.

We will check that your answers are correct. If they are not, we will get back to you and ask you to correct them. As long as you submit responses, you get an A for the assignment. It counts for 8 points total.