Advanced Methods in Text Analytics
Exercise 5: Transformers - Part 1
Solutions

Daniel Ruffinelli

FSS 2025

1 Transformer Basics

(a) Using dot-product attention over hidden representations $\boldsymbol{h}_i$ with query $\boldsymbol{k}$, we have:

$$\boldsymbol{c}_k = \sum_j \alpha_j \boldsymbol{h}_j,$$

where α_j is defined as follows:

$$\alpha_j = \text{softmax}(\boldsymbol{s})_j,$$

and s_j is given by:

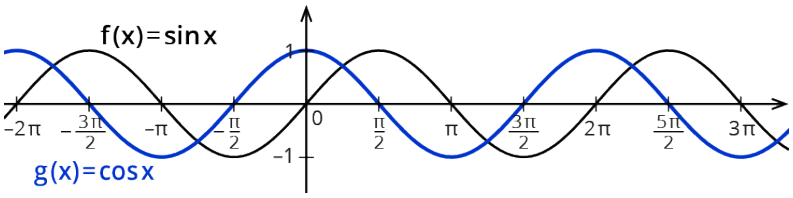
$$s_j = \boldsymbol{k}^T \boldsymbol{h}_j.$$

Components s_j and α_j are the *attention score* and *attention weight* for input j, respectively. Due to the dot product used to compute attention scores, we have that $D = K$ and $\boldsymbol{c}_k \in \mathbb{R}^D$, which is our context vector over $\boldsymbol{H}$ given query $\boldsymbol{k}$. Note that from the transformers perspective, hidden representations $\boldsymbol{h}_i$ are used as both (i) keys when computing s_j and (ii) as values when computing $\boldsymbol{c}_k$.

(b) The cost is $O(n^2)$ where n is number of tokens in the input sequence for the case where attend to all tokens, or the number of previously seen tokens in the case where we attend only to tokens to the left. The cost is explained by the fact that each token attends to every other token in the sequence (or only those to the left). This is highly parallelizable, since computing the contextualized representation of each input token is indepedent of that same computation for every other token. This is easy to see when representing self-attention as matrix products, an operation that is highly parallelizable.

Efforts exist to reduce this cost, e.g. using sparse attention this cost is reduced to $O(n\sqrt{n})$. Other effors exist, such as the performer models, which use sparse factorizations of the self-attention projection matrices, or FNet, which replaces self-attention altogether with linear transformations based on Fourier transforms.

(c) (i) Here's an image depicting how sine and cosine functions typically look like (source here).



They are examples of periodic functions, i.e. functions whose output values repeat in cycles. Such functions are often described in terms of properties such as frequency, period, etc. We will define these properties as needed in the following.

The construction of positional embeddings only depends on hyperparameters n and d, so the resulting embedding for position k is the same no matter the input sequence. This is something we want, as these embeddings should encode positional information independently of factors such as input sequence length. The i-th row of the following matrix thus contains the i-th positional embedding:

$$\begin{matrix} 0.000 & 1.000 & 0.000 & 1.000 \\ 0.841 & 0.540 & 0.032 & 1.000 \\ 0.909 & -0.416 & 0.063 & 0.998 \\ 0.141 & -0.990 & 0.095 & 0.996 \\ -0.757 & -0.654 & 0.126 & 0.992 \end{matrix}$$

The vectors are not large, as each element in this matrix is close to zero. This is so because the output of sin/cos functions is in the range $[-1, 1]$. Given that in the original architecture we *add* these positional embeddings to the token embeddings, we don't want them to be large. The translation that we apply to token embeddings via this sum should be large enough to distinguish the same token embeddings (which should encode semantics) when used in different positions, but not too large that the position information becomes a stronger signal to the model than the semantics in the input sequence. In addition, large positional vectors could introduce issues during training, such as large gradients that are mostly influenced by the position information, or similar issues that the model may deal with by making token embeddings small enough that they are too constrained to encode semantic information well. Note that such considerations may change if we use a different operation to combine token embeddings with position embeddings, e.g. concatenation.

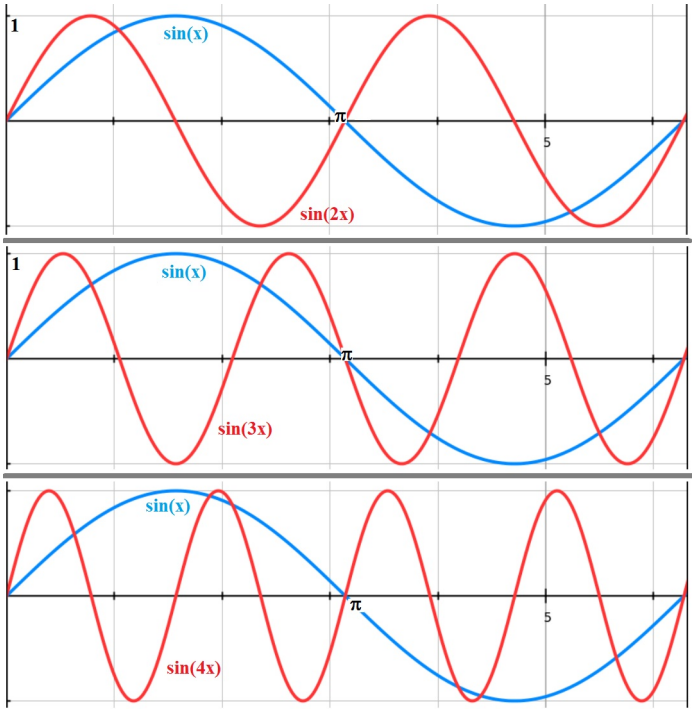
(ii) Since the frequency depends on n, d, i, each column of our PE matrix is a sin/cos function with a fixed frequency. If we look at each column as we increase values of k, i.e. across rows, we see that elements change values as we traverse the sin/cos (the values of the lasts columns also change, but this is less clear due to rounding).

$$\begin{matrix} 0.000 & 1.000 & 0.000 & 1.000 & 0.000 & 1.000 & 0.000 & 1.000 & 0.000 & 1.000 \\ 0.841 & 0.5409 & 0.158 & 0.987 & 0.025 & 1.000 & 0.004 & 1.000 & 0.001 & 1.000 \\ 0.909 & -0.416 & 0.312 & 0.95 & 0.05 & 0.999 & 0.008 & 1.000 & 0.001 & 1.000 \\ 0.141 & -0.990 & 0.458 & 0.889 & 0.075 & 0.997 & 0.012 & 1.000 & 0.002 & 1.000 \\ -0.757 & -0.654 & 0.592 & 0.806 & 0.100 & 0.995 & 0.016 & 1.000 & 0.003 & 1.000 \\ -0.959 & 0.284 & 0.712 & 0.702 & 0.125 & 0.992 & 0.020 & 1.000 & 0.003 & 1.000 \\ -0.279 & 0.960 & 0.814 & 0.581 & 0.150 & 0.989 & 0.024 & 1.000 & 0.004 & 1.000 \\ 0.657 & 0.754 & 0.895 & 0.445 & 0.175 & 0.985 & 0.028 & 1.000 & 0.004 & 1.000 \\ 0.989 & -0.146 & 0.954 & 0.298 & 0.200 & 0.980 & 0.032 & 0.999 & 0.005 & 1.000 \\ 0.412 & -0.911 & 0.99 & 0.144 & 0.224 & 0.975 & 0.036 & 0.999 & 0.006 & 1.000 \end{matrix}$$

In other words, the values in each column vary according to a sin/cos function. For uneven values of i, we see that values start at 1 and decrease as k increases (cosine).

For even values of k, we see values increase from 0 (sine). This allows us to create vectors with different values for contiguous features, i.e. vectors that are distinct from one another, which is a good thing, because we want *each* of them to encode a *different* position, and these vectors are not learned. We discuss the relation between i and k, i.e. rows and columns in the PE matrix, further in the next question.

(iii) An example of how a sine function changes with increasing frequencies is shown below (source here).



The columns, i.e. features, in our PE matrix change following such patterns. The higher the values of i, the lower the frequency and the higher the period of these functions. For example, for the values above, we have for $i = [0, 1, 2, 3]$ the corresponding periods $[6.28, 198.70, 6283.185, 198691.76]$. With higher periods, the values that the same feature (column) in our PE matrix takes across different rows (i.e. values of k) change more slowly. Thus, the rate of change for a given feature decreases with increasing values of i and we get more distinct features across different values of k (input positions) and across different values of i (positional embedding features). This allows this non-parametric approach to use vectors that are distinct from one another and each encode a different position from 1 to some maximum input length.

Overall, this non-parametric approach for obtaining positional embeddings is still useful for learning absolute position embeddings (APE), but other variants exist. BERT and GPT-3 use learned embeddings that encode APEs. The T5 model learns relative positional embeddings with an approach known as *relative bias*, and more recent models like PaLM and LLaMa use an approach called *rotary*, where they apply rotations to the query and key embeddings that are proportional to the absolute positions they want to encode. There are studies that look into the differences of such approaches, and some works have suggested that causal language models do not require positional embeddings, but still learn positional information, which makes sense given the causal approach to training and predicting that these models rely on.

2 Thinking about Perplexity

(a) We have $p(X = i) = \frac{1}{6}$ so that:

$$H(x) = -\sum_{x=1}^{6} \frac{1}{6} \log_2 \left(\frac{1}{6}\right) \approx 2.58$$

Entropy is a concept from information theory. We use this definition to define a bit: the entropy of a *binary* random variable that takes values 0 or 1 with equal probability. We say we have gained 1 bit of information when the value of such a variable becomes known. When we define entropy using the natural logarithm (we can define it using any logarithm), we call such a unit of information a *nat*.

A more general intuition about entropy, and one commonly used in NLP, is what it tells us about the distribution represented by some random variable. Specifically, the higher the entropy, the more uniform the distribution. This is because the probability mass is distributed more evenly across the entire distribution, similar to how the physical entropy of a gas always increases to fill up the space it is contained in.

(b) We have:

$$ppl(x) = 2^{\left(-\sum_{x=1}^{6} \frac{1}{6} \log_2\left(\frac{1}{6}\right)\right)} = 6$$

This is the number of possible values our random variable can take. This is why, in the context of evaluating language models, perplexity can be interpreted as the number of possible next words given a word, often referred to as *branching factor* (see Jurafsky Section 3.2.1).

In general, we can use any base for the logarithm when computing entropy, but we need to use that same base to compute perplexity. In code, we usually use ln, which is why we use the exponential function to compute perplexity.

(c) We have:

$$H(w_1, w_2, \ldots, w_n) = -\sum_{w_{1:n} \in L} p(w_{1:n}) \log p(w_{1:n})$$

The average entropy per word in the sequence, also known as *entropy rate*, is given by:

$$H(w_1, w_2, \ldots, w_n) = -\frac{1}{n} \sum_{w_{1:n} \in L} p(w_{1:n}) \log p(w_{1:n})$$

Note that we overload $H(x)$ to also represent entropy rate.

(d) The cross-entropy of some model p of m is defined as:

$$H(w_1, w_2, \ldots, w_n) = -\frac{1}{n} \sum_{w_{1:n} \in L} m(w_{1:n}) \log p(w_{1:n})$$

The intuition here is that we draw sequences from m, but we weigh them by their probability according to model p. In other words, this expectation (average information, i.e. entropy) is according to p. From a training perspective, m is typically the target distribution that we want to learn with model p. In the case of entropy of sequences of text, you can think of p as the softmax distribution of a language model you are training, and m the target distribution set by self-supervision (all the mass on a single word in the vocabulary).

(e) We proceed as follows:

$$\begin{aligned} ppl(W) &= 2^{H(W)} \\ &= 2^{\left(-\frac{1}{N}\log p(W)\right)} && \text{(apply log properties)} \\ &= p(W)^{-\frac{1}{N}}. \end{aligned}$$

The approximation given in this question shows us not only how perplexity is related to entropy, but it also explains what we are doing when we compute the perplexity of a held-out validation corpus: We are estimating the cross-entropy of the entire language, where p is our language model and m the true distribution of the language. Recall from machine learning courses, that cross-entropy is equivalent to log-loss, which explains why we typically apply the exponential function to our loss value to get perplexity in code (exp because we use ln).

To summarize, the takeaways from this task about perplexity are:

1. Since we typically train language models with the cross-entropy loss, a.k.a. log loss, and we use ln to compute logarithms in code, applying the exp function to our loss gives us perplexity, because $ppl = e^{H(W)}$ according to Eq. 1 in the task description.
2. The definition of perplexity from the lecture, i.e. $p(W)^{\frac{1}{N}}$, is related to the definition of perplexity from information theory, i.e. $2^{H(W)}$ assuming log base 2, via the way to approximate the entropy of a language described in Eq. 2 in the task description.
3. The point above suggests that when computing the perplexity of a language model on a held-out corpus, we are estimating the cross-entropy of the entire language used in the validation corpus. Since the larger the N in Eq. 2, the better the estimate, we want to use a large enough validation corpus.