

# Advanced Methods in Text Analytics

## Exercise 5: Transformers - Part 1

Daniel Ruffinelli

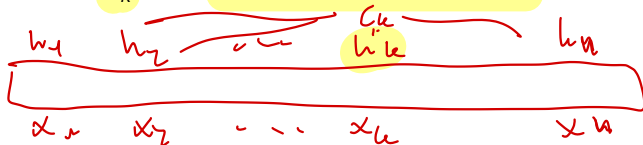
University of Mannheim

FSS 2024

# Transformer Basics

## Question a)

- ▶ Let  $x_1, x_2, \dots, x_n$  be a sequence of input tokens and  $\mathbf{H} = \{\mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_n\}$  where  $\mathbf{h}_i \in \mathbb{R}^D$  the corresponding sequence of representations given by some neural network's hidden layer (e.g. an RNN or transformer encoder).
- ▶ Give a formal expression for how to compute context vector  $\mathbf{c}_k$  using dot-product attention over  $\mathbf{H}$  with given query  $\mathbf{k} \in \mathbb{R}^K$  (e.g. the context vector corresponding to input token  $k$  in self-attention).
- ▶ Make sure you also provide a formal definition for every component in the expression you provide for  $\mathbf{c}_k$ .
- ▶ What are the names of these components? What is the size of  $\mathbf{c}_k$ ? And how does  $D$  relate to  $K$ ?



# Transformer Basics

Answer a)

- ▶ Using dot-product attention over hidden representations  $\mathbf{h}_i$  with query  $\mathbf{k}$ , we have: *sequence length*  
*↑*  
*power attend over H*

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

*Lo weights*

where  $\alpha_j = \text{softmax}(\mathbf{s})_j$  and  $\mathbf{s}_j = \mathbf{k}^T \mathbf{h}_j$ .

# Transformer Basics

Answer a)

- ▶ Using dot-product attention over hidden representations  $\mathbf{h}_j$  with query  $\mathbf{k}$ , we have:

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

where  $\alpha_j = \text{softmax}(\mathbf{s})_j$  and  $s_j = \mathbf{k}^T \mathbf{h}_j$ .

- ▶ Components  $s_j$  and  $\alpha_j$  are the *attention score* and *attention weight* for input  $j$ , respectively.

# Transformer Basics

Answer a)

- ▶ Using dot-product attention over hidden representations  $\mathbf{h}_j$  with query  $\mathbf{k}$ , we have:

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

where  $\alpha_j = \text{softmax}(\mathbf{s})_j$  and  $s_j = \mathbf{k}^T \mathbf{h}_j$ .

- ▶ Components  $s_j$  and  $\alpha_j$  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  $\mathbf{c}_k \in \mathbb{R}^D$ , which is our context vector over  $\mathbf{H}$  given query  $\mathbf{k}$ .

# Transformer Basics

Answer a)

- ▶ Using dot-product attention over hidden representations  $\mathbf{h}_j$  with query  $\mathbf{k}$ , we have:

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

v  
q k

where  $\alpha_j = \text{softmax}(\mathbf{s})_j$  and  $s_j = \mathbf{k}^T \mathbf{h}_j$ .

- ▶ Components  $s_j$  and  $\alpha_j$  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  $\mathbf{c}_k \in \mathbb{R}^D$ , which is our context vector over  $\mathbf{H}$  given query  $\mathbf{k}$ .
- ▶ **Transformers perspective:** representations  $\mathbf{h}_i$  used as (i) keys when computing  $s_j$  and (ii) values when computing  $\mathbf{c}_k$ .

# Transformer Basics

## Question b)

- ▶ What is the cost of a self-attention layer w.r.t. to the input size?
- ▶ Why? Answer this for both the settings where we attend to all tokens in the input sequence and when we attend only to tokens previously seen in the input sequence.
- ▶ Why are the computations in a self-attention layer parallelizable?

$$O(n^2 d) \rightarrow O(n^2)$$

$\downarrow$  constant

$\downarrow$  input length

$\rightarrow$  b, direction ✓  
 $\rightarrow$  causal ✓  
Infinite-attention

$$O(nh) \rightarrow \text{length of previously seen, range } [0, n-1]$$

# Transformer Basics

Answer b)

- ▶ The cost is  $O(n^2)$  where  $n$  is number of tokens in the input sequence for the case where we attend to all tokens, or the number of previously seen tokens in the case where we attend only to tokens to the left.



# Transformer Basics

Answer b)

- ▶ The cost is  $O(n^2)$  where  $n$  is number of tokens in the input sequence for the case where we 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).

# Transformer Basics

Answer b)

- ▶ The cost is  $O(n^2)$  where  $n$  is number of tokens in the input sequence for the case where we 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 independent of that same computation for every other token.

# Transformer Basics

Answer b)

- ▶ The cost is  $O(n^2)$  where  $n$  is number of tokens in the input sequence for the case where we 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 independent 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. ~~✗~~ ~~✗~~

# Transformer Basics

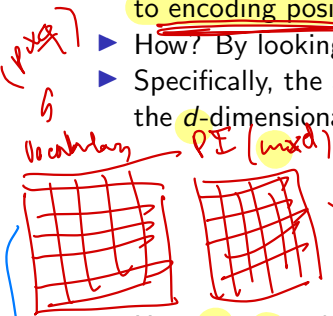
Answer b)

- ▶ The cost is  $O(n^2)$  where  $n$  is number of tokens in the input sequence for the case where we 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 independent 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.
  - ▶ Sparse attention reduces this cost to  $O(n\sqrt{n})$ .
  - ▶ Performer models use sparse factorizations of self-attention projection matrices
  - ▶ FNet replaces self-attention altogether with linear transformations based on Fourier transforms.

# Transformer Basics

## Question c) (Context)

- ▶ **Goal:** to develop an intuition for how non-learned approaches to encoding positions work
- ▶ How? By looking at approach used in original [transformer](#).
- ▶ Specifically, the authors used the following encoding to create the  $d$ -dimensional positional embedding (PE) for position  $k$ :



$$PE_{(\underline{k}, \underline{2i})} = \sin\left(\frac{1}{n^{2i/d}} k\right),$$

$$PE_{(\underline{k}, \underline{2i+1})} = \cos\left(\frac{1}{n^{2i/d}} k\right).$$

$k = \text{rows}$   
 $i = \text{columns}$


- ▶ Here,  $0 \leq i \leq d$ ,  $n$  is hyperparameter originally set to 10000.
- ▶ Note that the first equation is used to assign values to the *even* elements of the PE, and the second equation to the *uneven* elements.
- ▶ In other words, each element in positional embeddings is determined by either a sine or cosine function, each with different arguments.


# Transformer Basics

## Question c) (i)

- ▶ Given a maximum input length  $L$ , we can use these positional encodings to construct a PE matrix of size  $L \times d$ . Compute that matrix for the following sequences using  $d = 4$  and  $n = 1000$ :

*My cat is fine* 

*My dog is well* 

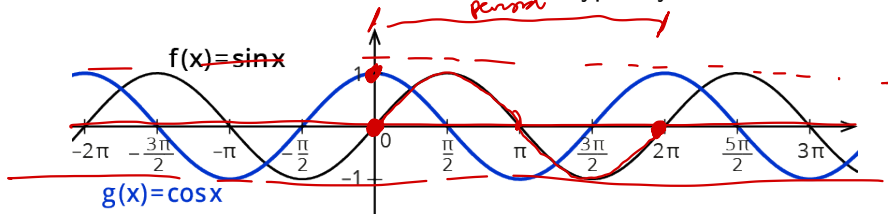
*My pets are very well* 

- ▶ How do the resulting matrices compare to one another?
- ▶ What about the size of the vectors? Are they large in terms of magnitude? Is that what we want?
- ▶ Discuss this w.r.t. the information that we expect that positional encodings provide to our transformer model.

# Transformer Basics

Answer c) (i) (1)

- ▶ Here's how sine and cosine functions typically look like:



- ▶ Known as **periodic functions**: output values repeat in cycles
  - ▶ **Period**: how long it takes to complete a full cycle
  - ▶ **Frequency**: inverse of period (i.e. portion of period per unit in x-axis)
  - ▶ **Amplitude**: highest value from reference value, e.g. mean, origin, etc. (in sin/cos measured from origin: 1)

# Transformer Basics

Answer c) (i) (2)

The cat sat on the bed  
+1 +5

- ▶ 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:

75 for	1	→	0.000	1.000	0.000	1.000
11	2	→	0.841	0.540	0.032	1.000
11	3	→	0.909	-0.416	0.063	0.998
n	4	→	0.141	-0.990	0.095	0.996
n	5	→	-0.757	-0.654	0.126	0.992

- ▶ The vectors are not large, as each element in this matrix is close to zero, because the output of sin/cos functions is in the range  $[-1, 1]$ .



# Transformer Basics

Answer c) (i) (3)

- ▶ Given that in the original transformer we *add* these PEs to input embeddings, we don't want PEs 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.
- ▶ All of this depends on operation used to combine token embeddings with PEs, e.g. different with concatenation.

# Transformer Basics

## Question c) (ii)

- ▶ The multiplying factor to  $k$  in the argument of the sin/cos functions above is known as its frequency. *points to  $k$  and  $\sin/\cos$  values*
- ▶ Given a fixed frequency, i.e. using a fixed value of  $d, n, i$ , what happens to the value given by the sin/cos functions as we increase the values of  $k$ ? *cosine of PE matrix*
- ▶ What will the features of such vectors look like as a result?

# Transformer Basics

Answer c) (ii) (1)

- ▶ 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 last columns also change, but this is less clear due to rounding).

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

- ▶ I.e., values in each column vary according to  $\sin/\cos$ .

# Transformer Basics

Answer c) (ii) (2)

- ▶ For uneven values of  $i$ , we see that values start at 1 and decrease as  $k$  increases (cosine)
- ▶ For even values of  $i$ , 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.

# Transformer Basics

## Question c) (iii)

- ▶ The frequency discussed above is defined as the inverse to the period of the function as follows:

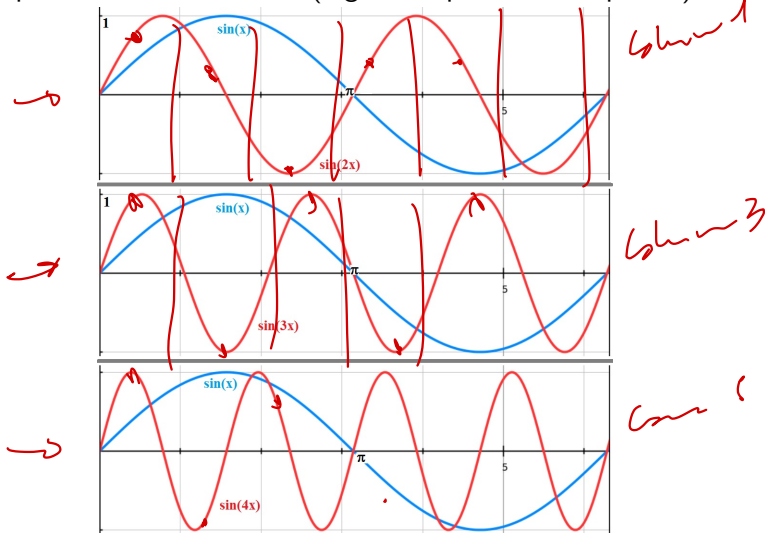
$$period = \frac{2\pi}{freq}$$

- ▶ The period of a function refers to how long it takes to complete a full cycle (after which they start to repeat themselves).
- ▶ In other words, its horizontal stretch.
- ▶ Using the values for  $d$  and  $n$  given above, compute the periods for different values of  $i$ .
- ▶ How do these periods relate to each other?
- ▶ And what happens to the value of a given element of  $i$  for different values of  $k$ ?

# Transformer Basics

Answer c) (iii) (1)

- An example of how a sine function changes with increasing frequencies is shown below (higher freq.  $\rightarrow$  shorter period).



# Transformer Basics

Answer c) (iii) (2)

- ▶ 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.
- ▶ So, the rate of change for a given feature decreases with increasing values of  $i$  and we get more distinct features across different values of  $k$  (i.e. rows, input positions) and across different values of  $i$  (i.e. columns, positional embedding features).
- ▶ Thus, this (non-parametric) approach uses PE vectors that are distinct from one another, each encoding a different position from 1 to MAX INPUT LENGTH.

# Transformer Basics

Answer c) (iii) (3)



- ▶ 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.



# Thinking about Perplexity

## Question a)

- ▶ For some random variable  $X$ , entropy  $H(X)$  is defined as follows:

$$H(x) = - \sum_{x \in X} p(x) \log p(x)$$

*Handwritten red annotations: A bracket above the sum is labeled 'w'. A bracket above the log term is labeled 'v'. A bracket below the entire term  $p(x) \log p(x)$  is labeled 'u'. The sum itself is underlined.*

- ▶ Say  $X$  is the number shown when you toss a fair die.
- ▶ What is the entropy of  $X$ ?
- ▶ Use log base 2 throughout.

# Thinking about Perplexity

Answer a) (1)

- ▶ 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*.

# Thinking about Perplexity

Answer a) (2)

- ▶ 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.

# Thinking about Perplexity

## Question b)

- ▶ Following a), perplexity is defined as  $2^{H(x)}$ .
- ▶ Compute the perplexity of the result you obtained in a).
- ▶ Can you interpret the result?
- ▶ How does this expression relate to how we compute perplexity in code?

$\log \text{ loss} \rightarrow \text{cross entropy} \sim \text{nn. loss entropy/loss}$

$\text{ppl} = \exp(\text{loss-value}) = e^{H(x)} \rightarrow$

# Thinking about Perplexity

Answer 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 sequence, 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.

# Thinking about Perplexity

## Question c)

- ▶ In NLP, we are interested in the entropy of sequences. Say  $X$  is now a random variable over all possible sequences of length  $n$  over some language  $L$ .
- ▶ Write the expression to compute this entropy.
- ▶ How would you compute the average entropy per word in such a sequence?

# Thinking about Perplexity

Answer c)

- ▶ We have:

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

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

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

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

# Thinking about Perplexity

## Question d)

- ▶ In the context of language models,  $p$  is the distribution over some natural language  $L$ , and we don't have access to this distribution.
- ▶ Instead, we learn an estimate  $m$  of this distribution from data.
- ▶ In such a context, the concept of *cross-entropy*  $H(p, m)$  is suitable.
- ▶ It is defined as the sum of  $p(x)$  for all  $x \in X$  weighted by their corresponding log probabilities according to  $m$ .
- ▶ Write the *cross-entropy rate* for a sequence of length  $n$ .



# Thinking about Perplexity

Answer d)

- ▶ The cross-entropy of some model  $m$  of  $p$  is defined as:

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

*target distribution* (above  $p(w_{1:n})$ )  
*estimate of  $p$*  (below  $m(w_{1:n})$ )

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

# Thinking about Perplexity

## Question e)

- ▶ It can be shown that the cross-entropy of a language can be approximated by the following expression:

$$H(W) = -\frac{1}{N} \log p(w_1, w_2, \dots, w_N)$$

where  $N$  is a sufficiently large number and  $w_i$  are words in that language.

- ▶ As seen in question (b), perplexity is formally defined as  $ppl(W) = 2^{H(W)}$ , where  $W$  is  $w_1, w_2, \dots, w_N$ , and we define entropy using log base 2.
- ▶ Use this approximation of cross-entropy given above to compute the expression for perplexity given in the lecture.
- ▶ What does this expression say about how we evaluate a language model by computing perplexity on a held-out corpus?

# Thinking about Perplexity

Answer e)

- ▶ We proceed as follows:

$$\begin{aligned} ppl(W) &= \underline{2^{H(W)}} \\ &= 2^{(-\frac{1}{N} \log p(W))} \quad (\text{apply log properties}) \\ &= p(W)^{-\frac{1}{N}}. \quad \text{def. from lecture} \end{aligned}$$

- ▶ 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  $m$  is our language model and  $p$  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  $\ln$ ).

held out  
ppl