TRANSFORMERS

THE MATHEMATICS OF TRANSFORMER ARCHITECTURE

ARAV AGARWAL

Abstract. This exposition provides a rigorous, and complete, mathematical walkthrough of the watershed neural network architecture introduced in the 2017 paper, *Attention Is All You Need* ([10]), now commonly known as the transformer. Every piece of the transformer architecture is covered, and no black box is left mathematically unopened. Further, we use this mathematical knowledge as an alternate path to build the intuition associated with transformers, which is commonly provided in naive visual format.

Contents

1. Preface

The original transformer paper [10] is terse, assuming a high level of familiarity with existing NLP literature. Naturally, there are a plethora of easily available resources explaining the intuition behind transformers. Some of these are pleasures to read, such as [1] and [6]. They are very accessible, and provide a naive overview of the attention mechanism using lovely illustrations, but also involve much hand-waving. The purpose of this exposition is to fix the lack of a resource explicating the transformer to a mathematically mature audience. It is not necessarily the difficulty of the mathematics involved (though it does get quite clever at times), but the sheer quantity of it: we open every black box from Figure 2 and uncover their deepest, darkest, mathematical secrets.

Online resources also do not provide a complete mathematical explanantion of the different modifications (see §5) of the vanilla attention mechanism (see §4) used in the transformer architecture. To obtain this, we manually carry out lots of linear algebra in §4 and §5 to discover what is actually going on.

As for structure, this paper begins with a section on pre-processing, covering tokenization and word embedding algorithms. We then start with a shallow dive into the transformer architecture of [10] in §3. Now the real fun starts, §4 serves as a (somewhat long) mathematical interlude, and can be read in isolation as a purely mathematical exposition of the attention mechanism. We then bring this back to our transformer model in §5 – this time for a deep dive into the original attention paper [10] – where we cover the three different ways transformer architecture utilizes the attention mechanism, and provide a mathematical explication of the same. At this point, all the attention black boxes of Figure 2 have been opened, but we are left with three more non-attention related black boxes. Namely, layer normalization (§6.1), FFNNs (§6.2), and positional encoding (§6.3). Each has their own interesting bit of mathematics, which is only to be found in the research papers introducing the concepts. We conduct a literature review, covering those in §6, and with that, end the exposition.

2. Pre-processing

2.1. **Tokenizination.**

Language models are provided with (unsurprisingly), language, in the form of text, as an input. How should the model preprocess this language input before running training or inference on it? This is where tokenization comes in, and is the primary non-trivial step in preprocessing text.

Definition 2.1. ***Tokenizination*** *of a text refers to splitting it into words or subwords. The components we split into are the* ***tokens****.*

Definition 2.2. *The set of all unique tokens used is referred to as the* ***vocabulary****.*

At the end of the day, models can only accept numbers as inputs, so tokens in our vocabulary are then simply converted to ids using a look-up table.

Now, splitting a text is no straightforward matter; after all, there are multiple ways of doing so. Broadly, however, we can think of three ways of doing this. The primary resource we use for this is [7].

Word-based tokenization is the obvious way of tokenizing text, whereby we split it by spaces, giving each word its own token. For example, consider the sentence

```
Isn't this project amazing?
```

which we would tokenize as

```
["Isn't", "this", "project", "amazing?"]
```

Sensible first step, but there are some problems. First, the punctuation symbols are attached to the words. This is suboptimal because the model would have to learn each word as a token, and associated tokens consisting of the word with all possible punctuations, drastically inflating vocabulary size. So, let's separate the puncuation into separate tokens.

```
["Isn","'","t", "this", "project", "amazing","?"]
```

The next step would be fix how the model tokenizes `Isn't`, which since it stands for `Is not`, is perhaps best tokenized as `"Is", "n't"`. This is a tokenization rule, and shows us why each model needs to have its own tokenizer type: depending on the rules, a different tokenized output would be generated for a given input. Naturally, a pretrained model would only work as intended if fed input tokenized with the same rules used for its tokenizing the data it was trained on.

```
["Is","n't", "this", "project", "amazing","?"]
```

Above we have an example of space and punctuation and rule based tokenization.

There is one glaring problem though: when we have a large text corpora, we would require a large vocabulary size. We could restrict our vocabulary and throw all unknown words into the `UNK` token, but if we do this for too many words, we would see poor performance; so word-based tokenizer models do in fact utilize very large voacabularies. For example, Transformer XL uses word-based tokenization, resulting in a vocabulary size of 267,735. The English language has an estimated vocabulary size of 170000, with the excess in Transformer XL appearing as punctuated words and conjunctions.

Large vocabulary sizes result in massive embedding matrices for both the first and last layer, which of course leads to higher time and space complexity.

How should we fix a large vocabulary size? Well, one obvious solution is to simply tokenize characters instead of words.

Character-based tokenization refers to tokenizing text by splitting it into individual characters. The vocabulary size is then limited by the number of alphabets, numbers, special characters etc. in our langauge. This is of course a drastic reduction from word-based tokenization. Further, any string can be split it into its characters, so unlike word-based tokenization nothing will be covered up by the `UNK` token.

The problem now is our model would struggle to learn a meaningful representation of a letter. That is, the letter `a` by itself carries no intrinsic meaning, and only lends itself to utility when used in a word; but `a` could be used in any number of words. It would be much easier to learn a context-independent representation of the word `apple` than of the letter `a`. Naturally, character-based tokenization suffers from significant loss in performance.

So, word- and character-based tokenization both have their pitfalls. What do we do? Well, we compromise: the secret to any good relationship.

Subword tokenization relies on the principle that "Frequently used words should not be split into smaller subtokens, but rare words should be decomposed into meaningful subtokens." ([7]). For example `tokenization` may be split into `token` and `ization`.

This allows for the model to have a moderate voacabulary size, but also learn meaningful representations of words and subwords. Further, new words which the model has never encountered before can be decomposed into subwords; if we force characters as part of our vocabulary, then in the worst case any word can be tokenized as its individual characters.

Following is an example where we use the pretrained BertTokenizer model:

```
>>> from transformers import BertTokenizer
>>> tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
>>> tokenizer.tokenize("This is not nonsense.")
["this", "is", "not", "non", "##sense", "."]
```

2.2. **Word-Embedding algorithms.** What happens after tokenization? As in most machine learning applications, we must first convert our tokens to vectors. This is carried out by means of an embedding algorithm; in particular, we mean building low-dimensional vector representations of words from a corpus of text, preserving the contextual similarity of words. We do not get into the details of this here, primarily because we do not have anything remotely original to add. [4], [8], [11], [14], and [12] are all excellent resources, and we are under no illusions that we could do a better job here. The algorithm is fairly simple and intuitive, involving the training of a simple classification neural network with one hidden layer. If one spends enough time staring at the following diagram, the dedicated reader can probably fully discern what the `Word2Vec` embedding algorithm does.

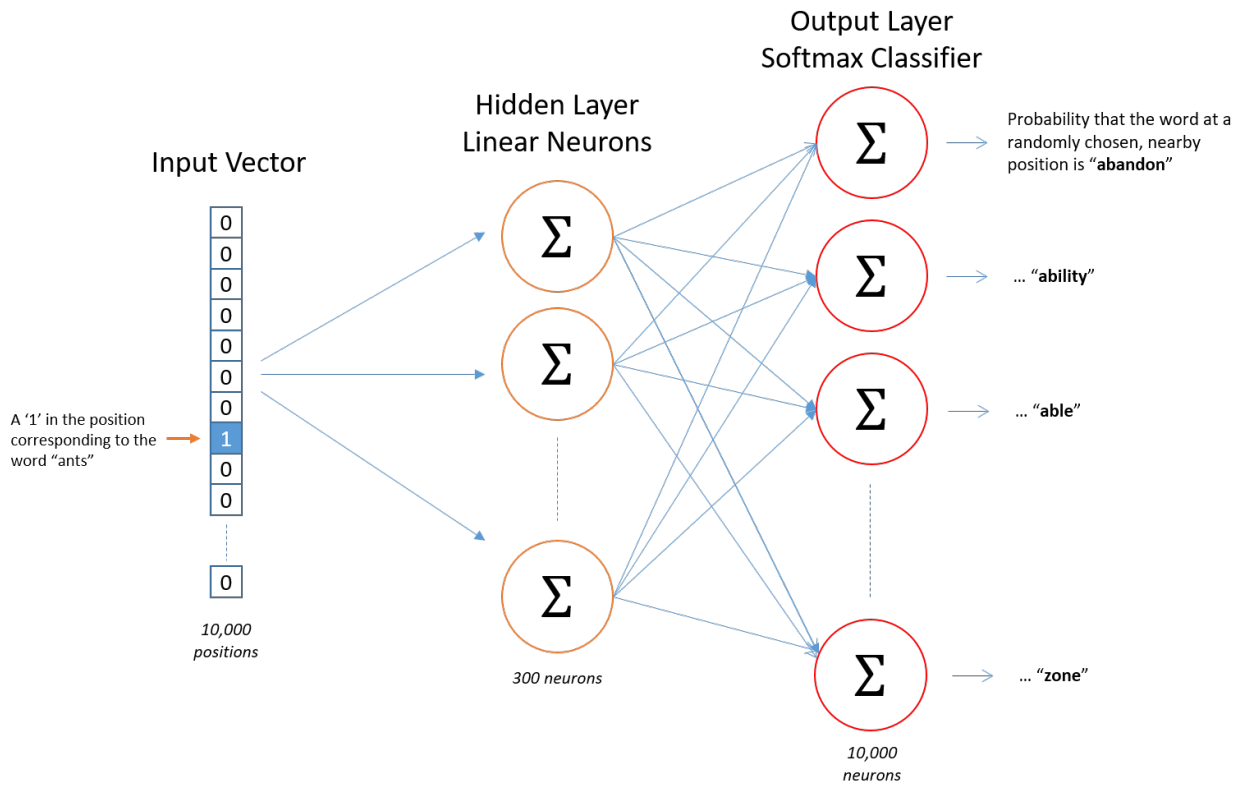

FIGURE 1. Word2Vec architecture, from [12]

We do however take this opportunity to introduce notation that will be used throughout this paper.

Remark 2.3 (Input Notation)**.** *In the pre-processing phase, we start with input text. This text is tokenized as in §2.1, and then passed through a word-embedding algorithm like* `Word2Vec`*. This finishes the pre-processing, resulting in our input sequence of vectors* $\vec{x}_1^{(0)}, \vec{x}_2^{(0)}, \ldots, \vec{x}_m^{(0)} \in \mathbb{R}^{d_{model}}$*, having dimensionality* d_{model}*. These are stacked row-wise resulting in a matrix* $X^{(0)} \in \mathbb{R}^{m \times d_{model}}$*:*

$$X^{(0)} = \begin{pmatrix} - & \vec{x}_1^{(0)} & - \\ - & \vec{x}_2^{(0)} & - \\ - & \vdots & - \\ - & \vec{x}_m^{(0)} & - \end{pmatrix}.$$

Note: Throughout this paper, a vector $\vec{x}$ will be a row vector, in accordance with the standard convention of machine learning literature.

3. Transformer Architecture

This section provides a high-level overview of transformer architecture, leaving the black boxes closed for now.

Let us begin with the architecture diagram used in the original paper [10].

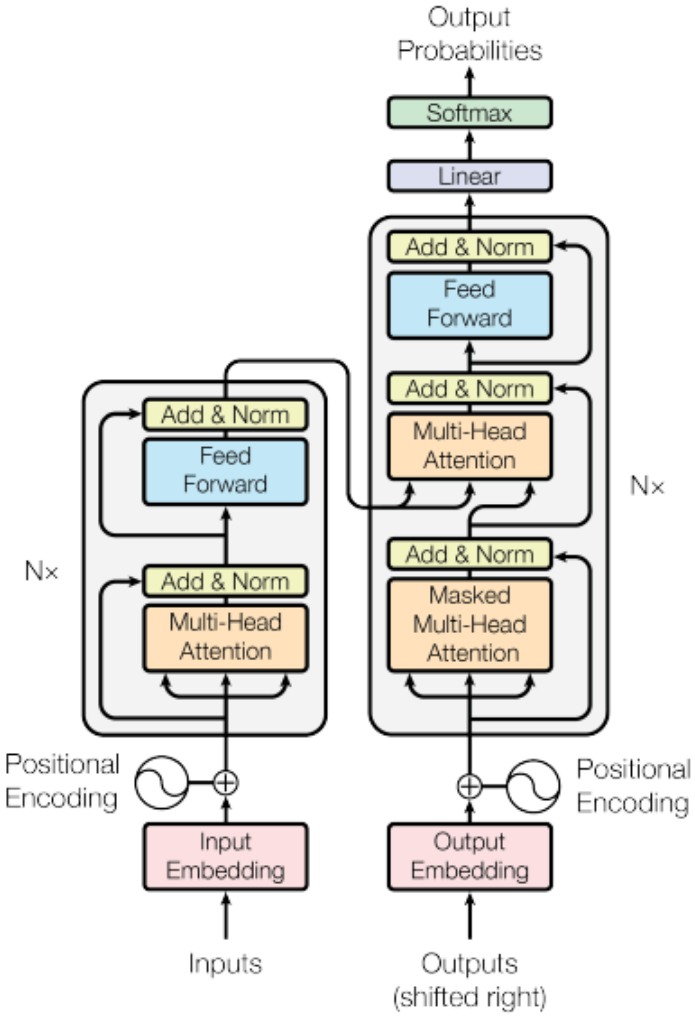

Figure 2. The Transformer Model Arhcitecture from [10]

The left block in the diagram is referred to as an **encoder**, and the right block is a **decoder**: this transformer is hence an encoder-decoder model. In general, the encoder composes of N blocks on the left, and the decoder an equal number of N blocks. In [10] the authors set $N = 6$.

Our input henceforth will be a sequence of vectors; recall this was obtained from our input text and involves tokenization followed by an embedding algorithm. So, the model accepts a sequence of input

vectors $\vec{x}_1^{(0)}, \vec{x}_2^{(0)}, \ldots, \vec{x}_m^{(0)}$, and these are passed into the first encoder block *all at once* in the form of the matrix $X^{(0)}$ from . Indeed, since the model accepts a matrix input and (as we will see) carries out symmetric operations on the whole matrix, the model lends itself to parallelization. Now, a given block processes the input, and passes its output on to the next encoder block. This repeats till we have passed through all N encoder blocks.

Each block by itself consists of two components. First, we have a Multi-Head Self-Attention layer. This is followed by a Feed-Forward Dense Neural Network (FFNN), with the network processing its input – received as the output from the self-attention layer – in parallel. After both the self-attention layer and the FFNN, we have an Add & Norm step; the "add" refers to a residual connection whereby we add the input of a given layer to its output, and the "norm" refers to Layer Normalization.

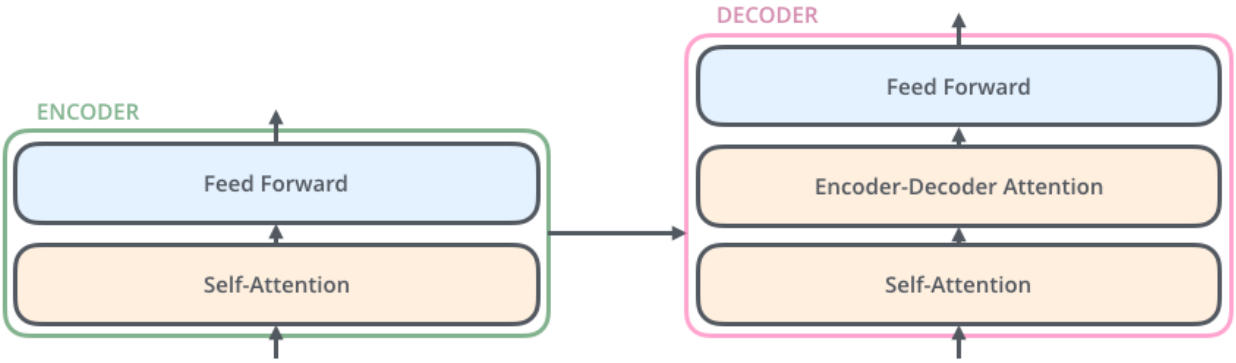

FIGURE 3. Encoder and Deocder blocks, from [1]

A decoder block is similar, but slightly different, to an encoder block. This time, each block consists of three elements: a Masked Multi-Head Self-Attention layer, followed by a Multi-Head Attention layer (or Encoder-Decoder Attention layer), and finally a feed-forward neural network. The similarity to the encoder occurs in the sense that the first and last layers are self-attention and FFNN respectively. The first primary difference is the presense of the middle Multi-Head Attention layer, often specifically referred to as the Encoder-Decoder Attention layer (as labeled in Figure 3). We would like to clarify explicity that this layer is *not* a self-attention layer; it is however an attention layer, and the difference will be clarified soon. The job of encoder-decoder attention is to connect the encoded source representation in $X^{(0)}$ with the target words $Y^{(0)}$. The second primary difference to an encoder block is that the first self-attention layer is masked, which roughly means any word in the target output can only attend to words that came before it.

4. Attention Please!

For now, this section may be read in isolation as a purely mathematical exposition, and we do not expect the reader to understand how precisely what we do here ties back to our transformer. With that disclaimer in §3, we made, over and over again, mysterious allusions to something called Attention. We now pull back the covers and clarify what we mean by the Attention mechanism. It is used in any and all attention layers of our transformer, but we will explain precisely how in §5. Our primary source for this section is [9], but much of the material is original.

4.1. **Scaled Dot-Product Attention.** Here, we focus on the purely mathematical study of a particular function called the **Scaled Dot-Product Attention**, defined below, as introduced in [10].

Definition 4.1 (Attention, Queries, Keys, and Values)**.** *Define a function which takes three matrices as input, and outputs one matrix, by* $\text{Attention} : \mathbb{R}^{m\times d_k} \times \mathbb{R}^{n\times d_k} \times \mathbb{R}^{n\times d_v} \to \mathbb{R}^{m\times d_v}$ *by*

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V.$$

Here, $Q \in \mathbb{R}^{m\times d_k}, K \in \mathbb{R}^{n\times d_k}$*, and* $V \in \mathbb{R}^{n\times d_v}$*. In accordance with [10] and preceeding literature (such as [3], [5], [13]) the rows of* Q *are called the "queries", those of* K *are the "keys", and those of* V *are called "values".*

In the above definition, and throughout this paper, note that applying softmax to a matrix refers to applying it to each row of the matrix. Further, observe that for the matrix products to work out, we require the dimensionality of queries and keys to match: so $\#\{\text{columns of } Q\} = \#\{\text{columns of } K\} = d_k$. Similarly, the number of keys and values must be equal: so $\#\{\text{rows of } K\} = \#\{\text{rows of } V\} = n$.

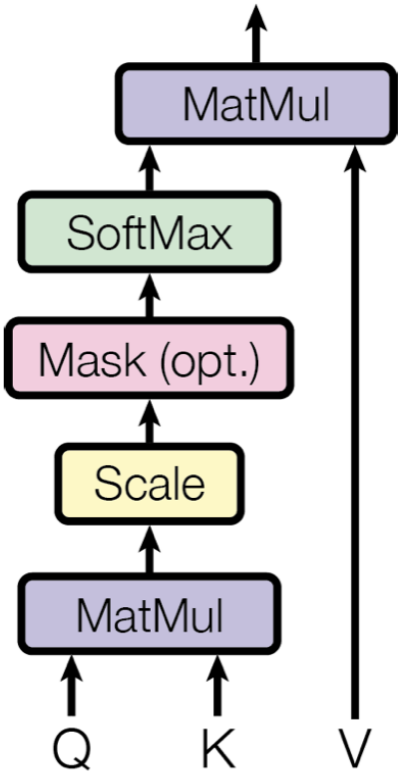

FIGURE 4. Scaled Dot-Product Attention, from [10]

Let us write down the matrices explicitly, compute the Attention, and see what the function actually does.

$$Q = \begin{pmatrix} - & \vec{q}_1 & - \\ - & \vec{q}_2 & - \\ - & \vdots & - \\ - & \vec{q}_m & - \end{pmatrix}, \quad K^T = \begin{pmatrix} | & | & & | \\ \vec{k}_1 & \vec{k}_2 & \cdots & \vec{k}_n \\ | & | & & | \end{pmatrix}.$$

$$QK^T = \begin{pmatrix} \vec{q}_1 \cdot \vec{k}_1 & \vec{q}_1 \cdot \vec{k}_2 & \cdots & \vec{q}_1 \cdot \vec{k}_n \\ \vec{q}_2 \cdot \vec{k}_1 & \vec{q}_2 \cdot \vec{k}_2 & \cdots & \vec{q}_2 \cdot \vec{k}_n \\ \vdots & \vdots & \ddots & \vdots \\ \vec{q}_m \cdot \vec{k}_1 & \vec{q}_m \cdot \vec{k}_2 & \cdots & \vec{q}_m \cdot \vec{k}_n \end{pmatrix}.$$

Next, we compute what are called the "attention weights" by dividing by $\sqrt{d_k}$ and applying softmax to each row. We have

$$\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) = \begin{pmatrix} \text{softmax}(\frac{1}{\sqrt{d_k}}\langle \vec{q}_1 \cdot \vec{k}_1, \vec{q}_1 \cdot \vec{k}_2, \ldots, \vec{q}_1 \cdot \vec{k}_n \rangle) \\ \text{softmax}(\frac{1}{\sqrt{d_k}}\langle \vec{q}_2 \cdot \vec{k}_1, \vec{q}_2 \cdot \vec{k}_2, \ldots, \vec{q}_2 \cdot \vec{k}_n \rangle) \\ \vdots \\ \text{softmax}(\frac{1}{\sqrt{d_k}}\langle \vec{q}_m \cdot \vec{k}_1, \vec{q}_m \cdot \vec{k}_2, \ldots, \vec{q}_m \cdot \vec{k}_n \rangle) \end{pmatrix} = \begin{pmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \cdots & w_{m,n} \end{pmatrix}.$$

Finally, we multiply by V to obtain the Attention.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V$$

$$= \begin{pmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \cdots & w_{m,n} \end{pmatrix} \begin{pmatrix} - & \vec{v}_1 & - \\ - & \vec{v}_2 & - \\ - & \vdots & - \\ - & \vec{v}_n & - \end{pmatrix}$$

$$= \begin{pmatrix} \sum_{j=1}^n w_{1,j} \vec{v}_j \\ \sum_{j=1}^n w_{2,j} \vec{v}_j \\ \vdots \\ \sum_{j=1}^n w_{m,j} \vec{v}_j \end{pmatrix} = \begin{pmatrix} \vec{a}_1 \\ \vec{a}_2 \\ \vdots \\ \vec{a}_m \end{pmatrix}$$

So, Attention gives us a sequence of weighted averages of the values; more precisely, exactly m distinct weighted averages (because Attention has m rows) of the n rows of V. The weights $w_{i,j}$ depend on Q and K, i.e., the queries and the keys. Above we have also called a given row of the Attention matrix, i.e., a given weighted average of values, an attention vector. So, we have m attention vectors $\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_m$, where $a_i \in \mathbb{R}^{d_v}$, and

$$\vec{a}_i = \sum_{j=1}^{n} w_{i,j}\, \vec{v}_j.$$

Let us take a closer look at the values of these weights. We can write

$$w_{i,j} = \frac{1}{S} \exp\left(\frac{\vec{q}_i \cdot \vec{k}_j}{\sqrt{d_k}}\right),$$

where the expression follows from by applying softmax to each row of $\frac{QK^T}{\sqrt{d_k}}$. The normalization constant S will be

$$S = \sum_{j=1}^{n} \exp\left(\frac{\vec{q}_i \cdot \vec{k}_j}{\sqrt{d_k}}\right).$$

Finally, we can substitute our expression for weights in the definition of an attention vector to write

$$\vec{a}_i = \sum_{j=1}^{n} w_{i,j}\, \vec{v}_j = \frac{1}{S} \sum_{j=1}^{n} \exp\left(\frac{\vec{q}_i \cdot \vec{k}_j}{\sqrt{d_k}}\right) \vec{v}_j.$$

We can now understand the terminology of queries, keys, and values. Analogous to a hashtable, Attention picks out values $\vec{v}_j$ via the corresponding one-to-one keys $\vec{k}_j$ (hence number of keys and values is equal). The keys we use to compute the an attention vector $\vec{a}_i$ are indicated, or modified, by the ith query $\vec{q}_i$. To understand the effect of the query $\vec{q}_i$ on the attention vector $\vec{a}_i$, let us consider the dot-product

$$\vec{q}_i \cdot \vec{k}_j = |\vec{q}_i||\vec{k}_j| \cos(\theta).$$

And so, on exponentiating, we have

$$w_{i,j} = \frac{1}{S} \exp\left(\frac{\vec{q}_i \cdot \vec{k}_j}{\sqrt{d_k}}\right) \propto \exp(\cos\theta).$$

Observe the effect of this exponentation in the following graph

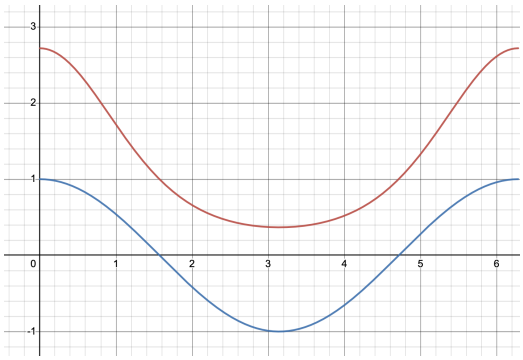

FIGURE 5. exp cos in red, and cos in blue, from 0 to 2π

where we note that larger values of cosine are amplified by exponentation and smaller values are diminished. That is to say,

$$\cos\theta > 0 \implies \exp\cos\theta > \cos\theta, \text{ and } \cos\theta < 0 \implies |\exp\cos\theta| < |\cos\theta|.$$

So, the closer in angle (i.e., the more aligned) the query $\vec{q}_i$ is with the key $\vec{k}_j$, the higher the representation of the value $\vec{v}_j$ in the attention vector $\vec{a}_i$.

4.2. **Multi-Head Attention.**
We now consider an extension of the Scaled Dot-Product Attention function we considered above. A number of h attention layers (attention heads) are connected in parallel to form what we call **Multi-Head Attention**. Note that the parameters for each head are independent.

In particular, linear transformations are applied to the queries, keys, and values yielding multiple sets of inputs to the Attention function. The function is then computed in parallel for each of these given sets of inputs. The outputs are concatenated on columns, that is to say the matrices are stacked side by side along their columns. This concatenated matrix is now brought back into the right dimensions by means of yet another linear transformation. Figure 6 below provides a helpful illustration.

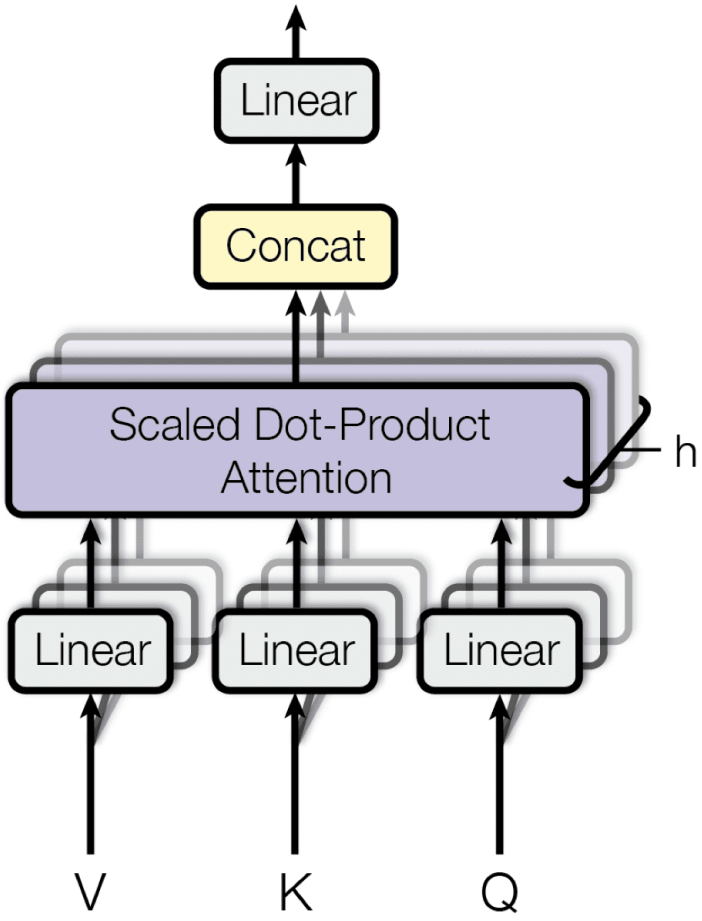

FIGURE 6. Multi-Head Attention, from [10]

We can formalize this process mathematically with the following definition.

Definition 4.2. *The function which is the h-parallelization of* Attention *from 4.1 is called* ***Multi-Head Attention****. The final output is obtained by concatenation followed by a linear transformation. We have* $\text{MultiHead} : \mathbb{R}^{m\times d_{model}} \times \mathbb{R}^{n\times d_{model}} \times \mathbb{R}^{n\times d_{model}} \to \mathbb{R}^{m\times d_{model}}$ *where*

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)\, W^O.$$

where head_i *is the Scaled Dot-Product Attention applied to the* i*th set of linearly transformed queries, keys, and values:*

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V).$$

We have $Q \in \mathbb{R}^{m \times d_{model}}, K \in \mathbb{R}^{n \times d_{model}}$*, and* $V \in \mathbb{R}^{n \times d_{model}}$*. The function has (trainable) parameter matrices* $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}, W_i^K \in \mathbb{R}^{d_{model} \times d_k}, W_i^V \in \mathbb{R}^{d_{model} \times d_v}$*, and* $W^O \in \mathbb{R}^{hd_v \times d_{model}}$*. Here,* d_{model} *is the dimensionality of the vectors in our model, and* h *is the number of attention heads.*

As we can see, all that is meant by a linear transformation is a right matrix product (on the right because here we stack vectors as rows, not columns). In [10], $d_{model} = 512$ and $h = 8$.

5. Multi-Head Attention in Transformers

With our solid mathematical understanding of the Attention mechanism, we can begin working through how this function is used in the transformer architecture of [10]. Recalling our discussion in §3 and in particular referring to Figures 2 and 3, we observe the use of Attention function in three different places in the architecture. First, as *self*-attention in the encoder blocks. Second, in computing *masked* self-attention in decoder blocks. Finally, as encoder-decoder attention in the decoder. Let us study each of them individually.

5.1. Self-Attention.

The notion of **Multi-Head Self-Attention** is just a modification of the Attention mechanism. That is to say, it is just a special case of attention, whereby the queries, keys, and values, are all the same. So, in the encoder block, we perform the operation defined below.

Definition 5.1. *Set* $Q = K = V = X \in \mathbb{R}^{m \times d_{model}}$*. Computing* MultiHead *with these inputs is called* ***Multi-Head Self-Attention****, so it is a special case of Multi-Head Attention from 4.2. Define the function by* $\text{MultiHeadSelf} : \mathbb{R}^{m \times d_{model}} \to \mathbb{R}^{m \times d_{model}}$ *with*

$$\text{MultiHeadSelf}(X) = \text{MultiHead}(X, X, X) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O,$$

where

$$\text{head}_i = \text{Attention}(XW_i^Q, XW_i^K, XW_i^V).$$

$W_i^Q, W_i^K, W_i^V, W^O, d_{model}$ *and* h *are the same as in 4.2.*

Let $X^{(0)} \in \mathbb{R}^{m \times d_{model}}$ be our first input matrix consisting of our input sequence $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_m \in \mathbb{R}^{d_{model}}$ stacked as rows. Recall the the $\vec{x}_i$s were obtained using an embedding algorithm from our tokens. Further, let $X^{(i)}$ be the output of the ith encoder block, and note the input to the $i+1$th encoder block is $X^{(i)}$. So, the first thing the ith encoder block does is execute the Multi-Head Self-Attention layer (before Layer normalization and the FFNN), which computes $\text{MultiHeadSelf}(X^{(i)})$, and passes the result onwards.

5.2. Masked Attention.

We now explain what we mean by *masked* attention, and how that leads further leads to masked multi-head self-attention. What are we masking? The purpose is to prevent queries from seeing keys that follow them in an input sequence. So, for a given query $\vec{q}_i$ we want to hide, or "mask", all $\vec{k}_j$ with $j > i$. Why would we want to do this? Consider the state of the model when doing inference. We have our input sequence $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_m$ (actually stored in $X^{(0)}$), and execute the model repeatedly to produce a target sequence $\vec{y}_1, \vec{y}_2, \ldots, \vec{y}_n$. In the first iteration, the model only has available to it the source sequence in $X^{(0)}$ to generate the first target $\vec{y}_1$. In the second iteration, the model has access to $X^{(0)}$ as well as $\vec{y}_1$. This continues, where in the ith iteration the model has access to $X^{(0)}$ and all $\vec{y}_j$ with $j < i$, and never has access to any future elements $\vec{y}_j$ with $j > i$.

The purpose of masking in training is to simulate these conditions – faced during inference – in training; for we would like the model to be capable of using only past elements of the output to further generate meaningful outputs, and not rely on information from future elements, allowing it to generalize well during inference.

How do we execute masking during training in practice? There are two steps.

The first step is to shift the the target sequence one over to the right. That is, if the original target sequence was $\vec{y'_1}, \vec{y'_2}, \ldots, \vec{y'_{n-1}}$ define a new sequence $\vec{y}_1, \vec{y}_2, \ldots, \vec{y}_n$, where $\vec{y}_1$ is the null or empty token (or rather the embedding corresponding to that token), and $\vec{y}_i = \vec{y}_{i-1}$ for $i > 2$.

$$Y' = \begin{pmatrix} \vec{y'_1} \\ \vec{y'_2} \\ \vdots \\ \vec{y'_{n-1}} \end{pmatrix} \longrightarrow Y = \begin{pmatrix} \vec{y}_1 \\ \vec{y}_2 \\ \vdots \\ \vec{y}_n \end{pmatrix}$$

So, in our new target sequence matrix Y, the first row is devoid of information, and it is in the the second row that we first see some actual information (the embedding of the first menaingful token in our input text). This simulates the fact that when conducting inference, in the first iteration, our model does not have access to any information from the target sequence, but rather is only able to utlize the source sequence in $X^{(0)}$: hence $\vec{y}_1$ corresponds to the empty token. In the second iteration, the model can use information from the actual first target $\vec{y}_2 = \vec{y'_1}$, and so on.

The second step in masking is to actually emulate the condition of the model not having access to future elements. In implementation terms, this means the model should never query a future key. That is, the dot product $\vec{q}_i \cdot \vec{k}_j$ with $i < j$ should not be allowed. So, recalling our matrix QK^T from §4.1:

$$QK^T = \begin{pmatrix} \vec{q}_1 \cdot \vec{k}_1 & \vec{q}_1 \cdot \vec{k}_2 & \cdots & \vec{q}_1 \cdot \vec{k}_n \\ \vec{q}_2 \cdot \vec{k}_1 & \vec{q}_2 \cdot \vec{k}_2 & \cdots & \vec{q}_2 \cdot \vec{k}_n \\ \vdots & \vdots & \ddots & \vdots \\ \vec{q}_m \cdot \vec{k}_1 & \vec{q}_m \cdot \vec{k}_2 & \cdots & \vec{q}_m \cdot \vec{k}_n \end{pmatrix}$$

we observe that the terms in the lower triangle and the diagonal are fine, but the terms in the upper triangle should be masked. In practice, we add to $\frac{1}{\sqrt{d_k}} QK^T$ a masking matrix M, where M is 0s in

the lower triangle and $-\infty$s elsewhere, effectuating the following:

$$\frac{1}{\sqrt{d_k}}QK^T + M = \frac{1}{\sqrt{d_k}}\begin{pmatrix} \vec{q}_1\cdot\vec{k}_1 & \vec{q}_1\cdot\vec{k}_2 & \cdots & \vec{q}_1\cdot\vec{k}_n \\ \vec{q}_2\cdot\vec{k}_1 & \vec{q}_2\cdot\vec{k}_2 & \cdots & \vec{q}_2\cdot\vec{k}_n \\ \vdots & \vdots & \ddots & \vdots \\ \vec{q}_m\cdot\vec{k}_1 & \vec{q}_m\cdot\vec{k}_2 & \cdots & \vec{q}_m\cdot\vec{k}_n \end{pmatrix} + \begin{pmatrix} 0 & -\infty & -\infty & \cdots & -\infty & -\infty \\ 0 & 0 & -\infty & \cdots & -\infty & -\infty \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & -\infty \\ 0 & 0 & 0 & \cdots & 0 & 0 \end{pmatrix}$$

$$= \frac{1}{\sqrt{d_k}}\begin{pmatrix} \vec{q}_1\cdot\vec{k}_1 & -\infty & \cdots & -\infty \\ \vec{q}_2\cdot\vec{k}_1 & \vec{q}_2\cdot\vec{k}_2 & \cdots & -\infty \\ \vdots & \vdots & \ddots & \vdots \\ \vec{q}_m\cdot\vec{k}_1 & \vec{q}_m\cdot\vec{k}_2 & \cdots & \vec{q}_m\cdot\vec{k}_n \end{pmatrix}$$

With this, we observe that on applying softmax to each row we send the entries with $-\infty$ to 0 since $\exp(-\infty) = 0$. We can now formalize the notion of masking with the following definition.

Definition 5.2. *The **Masked Attention** function is a simple modification of the Attention function from 4.1. Define* $\text{MaskAttention} : \mathbb{R}^{m\times d_k} \times \mathbb{R}^{n\times d_k} \times \mathbb{R}^{n\times d_v} \to \mathbb{R}^{m\times d_v}$ *by*

$$\text{MaskAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V,$$

where the masking matrix M is given by

$$M = \begin{pmatrix} 0 & -\infty & -\infty & \cdots & -\infty & -\infty \\ 0 & 0 & -\infty & \cdots & -\infty & -\infty \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & -\infty \\ 0 & 0 & 0 & \cdots & 0 & 0 \end{pmatrix}.$$

Here, $Q \in \mathbb{R}^{m\times d_k}, K \in \mathbb{R}^{n\times d_k}, M \in \mathbb{R}^{m\times n}$ and $V \in \mathbb{R}^{n\times d_v}$.

We refer again to Figure 4, and note that the difference in 4.1 and 5.2 is precisely the implementation of the optional `Mask` block.

Finally, we can add on the "Multi-Head" and "Self" modifiers, which have the expected effect, and define masked multi-head self-attention.

Definition 5.3. *The **Masked Multi-Head Self-Attention** function is a simple modification of Multi-Head Self Attention from 5.1, where instead of using* Attention *from 4.1 in each* head_i*, we use* MaskAttention *from 5.2. Define* $\text{MaskMultiHeadSelf} : \mathbb{R}^{m\times d_{model}} \to \mathbb{R}^{m\times d_{model}}$ *with*

$$\text{MaskMultiHeadSelf}(X) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O,$$

where

$$\text{head}_i = \text{MaskAttention}(XW_i^Q, XW_i^K, XW_i^V).$$

$W_i^Q, W_i^K, W_i^V, W^O, d_{model}$ and h are the same as in 4.2.

5.3. **Encoder-Decoder Attention.** The third and final use of multi-head attention in transformers is in the Encoder-Decoder Attention layer, which lies on the decoder side of the architecture. First, note that unlike the other two uses explicated in the previous two subsectoins 5.1 and 5.2, and also shown in Figures 2 and 3, the attention of this subsection is not self-attention. By now, we know that all this means is that the input matrices to MultiHead from 4.2 are not all equal.

Observe from Figure 2 that Encoder-Decoder follows directly after the Masked Multi-Head Self-Attention layer. Its purpose is to relate the source and target sequences to one another, that is, have them "attend" to one another. Indeed, Encoder-Decoder Attention is just Multi-Head Attention from 4.2 with a specified set of inputs. The key and value matrices K, V are both taken from the encoder output, while the queries Q come from the output of the Masked Multi-Head Self-Attention layer below.

Let $X^{(encoded)}$ be the final encoded representation of our source sequence $X^{(0)}$, i.e., the encoded representation of $X^{(0)}$ resulting from being processed by all N encoder blocks. Next, in the ith decoder block, let $Y^{(i)}$ be the output of ith decoder block for $i > 0$; for $i = 0$ it will be our target sequence as defined in §5.2 (or rather the embedding of the same). Finally, let

$$\overline{Y^{(i)}} = \text{LayerNorm}\Big(\text{MaskMultiHeadSelf}\Big(Y^{(i)}\Big) + Y^{(i)}\,;\,\cdot,\cdot\Big) \tag{5.4}$$

(see §6.1 for LayerNorm). Then, $\overline{Y^{(i)}}$ is precisely the query matrix for the $i+1$th Encoder-Decoder Attention layer, which computes the quantity

$$\text{MultiHead}\Big(\overline{Y^{(i)}}, X^{(encoded)}, X^{(encoded)}\Big),$$

and comes with its own set of trainable parameter matrices. The output of this is as usual passed through LayerNorm onto an FFNN; executing the FFNN followed by LayerNorm results in $Y^{(i+1)}$, and the process repeats, onward to the next decoder block.

Remark 5.5. *We emphasize that all decoder blocks receive the same data from the encoder side of the transformer. In particular, from the* 1*st to the* N*th decoder block, the keys and values for the Encoder-Decoder Attention layer are all given by* $X^{(encoded)}$*. That is,* $K = V = X^{(encoded)}$ *for all Encoder-Decoder Attention layers.*

With this, we have completed our unwrapping of the black boxes in Figure 1 involving attention, and understand their roles. There are three black boxes left to unpack, and we do so in the subsequent section.

6. The Final Three

6.1. **Layer Normalization.** Observe in Figure 1 every attention layer and FFNN, is followed by an "Add & Norm" operation, which acts as "Residual Connection". Let us tackle each of these terms one-by-one.

The "Norm" part of "Add & Norm" refers to performing layer normalization, a technique first introduced in [2].

Definition 6.1 (LayerNorm)**.** *Define the mathematical function* $\text{LayerNorm} : \mathbb{R}^k \to \mathbb{R}^k$ *with*

$$\text{LayerNorm}(\vec{a}; \vec{g}, \vec{b}) = \vec{g} \odot \frac{\vec{a} - \mu \cdot \vec{1}}{\sigma} + \vec{b},$$

where we re-center and re-scale by the mean and standard deviation of the input vector

$$\mu = \frac{1}{k}\sum_{i=1}^{k} a_i, \quad \sigma = \sqrt{\frac{1}{k}\sum_{i=1}^{k}(a_i - \mu)^2}.$$

Here, $\vec{1} = (1, 1, \ldots, 1)$ *is the vector of all* 1*s, and* $\odot$ *is element-wise vector multiplication. The bias* $\vec{b}$ *and gain* $\vec{g}$ *are (trainable) parameters in* $\mathbb{R}^k$*.*

But LayerNorm takes one vector to another of the same dimension. How do transformers use this?

Remark 6.2. *In the context of transformer architecture,* ***Layer Normalization*** *refers to the row-wise extension of* LayerNorm *to a matrix. So, given a matrix* $X \in \mathbb{R}^{m \times d_{model}}$ *with*

$$X = \begin{pmatrix} - & \vec{x}_1 & - \\ - & \vec{x}_2 & - \\ & \vdots & \\ - & \vec{x}_m & - \end{pmatrix},$$

$$\text{LayerNorm}(X; \vec{g}, \vec{b}) = \begin{pmatrix} - & \text{LayerNorm}(\vec{x}_1; \vec{g}, \vec{b}) & - \\ - & \text{LayerNorm}(\vec{x}_2; \vec{g}, \vec{b}) & - \\ & \vdots & \\ - & \text{LayerNorm}(\vec{x}_m; \vec{g}, \vec{b}) & - \end{pmatrix}.$$

Note that each separate Add & Norm layer has independent trainable parameters: the bias $\vec{b}$ and gain $\vec{g}$; but during Layer Normalization on a matrix the rows share the same bias and gain parameters.

Now for the "Add" part of "Add & Norm". This is straightforward. A given layer L, attention or FFNN, has an input matrix, say $X_L^{(in)}$, and an output matrix $X_L^{(out)}$. "Add" simply refers to computing $X_L^{(in)} + X_L^{(out)}$. We can now make the following complete definition for the Add & Norm black box.

Definition 6.3. *Following a given attention or FFNN layer* L*, we carry out the* ***Add & Norm*** *operation, which computes*

$$\text{LayerNorm}(X_L^{(in)} + X_L^{(out)}; \vec{g}, \vec{b}),$$

where $X_L^{(in)}, X_L^{(out)}$ *are the input and output matrices for the layer* L*.*

Remark 6.4. *Note that* $X_L^{(in)}$*, the input matrix, is well-defined for our FFNNs and (perhaps masked) Self-Attention layers* L*, both being functions of only matrix. For Encoder-Decoder Attention layers, recall from §5.3 that they accept two input matrices:* $X^{(encoded)}$ *and* $\overline{Y^{(i)}}$*. Which one to choose as the input matrix? We define* $X^{(in)}$ *to be the query matrix for any layer. So, for an Encoder-Decoder layer, we have* $X_L^{(in)} = \overline{Y^{(i)}}$*.*

6.2. **Feed-Forward Neural Network (FFNN).** Observe the penultimate step in any encoder or decoder block is to execute an FFNN. The network consists of two linear transformations with a ReLU activation in between.

Definition 6.5 (FFNN)**.** *We define a function* $\text{FFNN} : \mathbb{R}^{d_{model}} \to \mathbb{R}^{d_{model}}$ *with*

$$\text{FFNN}(\vec{x}) = \max(0, \vec{x}W_1 + \vec{b}_1)W_2 + \vec{b}_2.$$

In the transformer, this is applied to each position separately and identically. That is, given an input matrix

$$X = \begin{pmatrix} - & \vec{x}_1 & - \\ - & \vec{x}_2 & - \\ - & \vdots & - \\ - & \vec{x}_m & - \end{pmatrix} \in \mathbb{R}^{m \times d_{model}},$$

we have

$$\mathrm{FFNN}(X; \vec{g}, \vec{b}) = \begin{pmatrix} - & \mathrm{FFNN}(\vec{x}_1) & - \\ - & \mathrm{FFNN}(\vec{x}_2) & - \\ - & \vdots & - \\ - & \mathrm{FFNN}(\vec{x}_m) & - \end{pmatrix},$$

where each row identical parameters of weights and biases.

6.3. **Positional Encoding.** The astute reader would have noticed something missing from the transformer architecture we have covered so far. Namely, the transformer model is oblivious to relational, or ordering, structure between elements of its input sequence $\vec{x}_1^{(0)}, \vec{x}_2^{(0)}, \ldots, \vec{x}_m^{(0)}$. Yes, we stack them in order row-wise (and in the right order) in our matrix $X^{(0)}$ and off we go with attention, layer normalization, FFNNs, and so on. But at no point does the model use, or learn, the information that in fact $\vec{x}_1^{(0)}$ comes before $\vec{x}_2^{(0)}$, which is before $\vec{x}_3^{(0)}$ etc.

Indeed, certainly FFNN and layer normalization are oblivious to position, with them executing the same action on each row of an input matrix. But further, even self-attention is blind to the ordering: yes, each one of the input vectors "attends" to one another by querying each other's keys, but this takes place in a symmetric fashion between each element of the input vectors. That is, $\vec{x}_1^{(0)}$ attends to $\vec{x}_2^{(0)}$ just as much as it does to $\vec{x}_m^{(0)}$, even though $\vec{x}_2^{(0)}$ comes directly after, and $\vec{x}_m^{(0)}$ right at the end.

The model contains no recurrence or convolution, so has no way to learn information about the order of the sequence. To fix this, we need to express these positional relationships as data, and inject this information into our input sequence itself. This is the explanation for the Positional Enoding symbols following our starting embeddings in Figure 2, highlighted below.

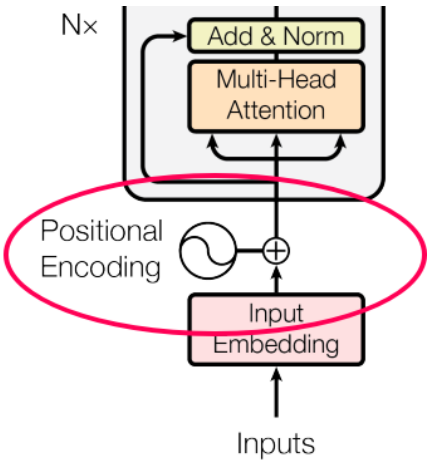

FIGURE 7.

The easiest way to capture positional information is to encode positions as 1-hot features. To do this end, let $X \in \mathbb{R}^{m \times d_{model}}$ be sequentially ordered data along the m-dimensional axis. (Note this is exactly the case for $X^{(0)}$, where the input sequence rows $\vec{x}_1^{(0)}, \vec{x}_2^{(0)}, \ldots, \vec{x}_m^{(0)}$ are sequentially ordered.)

Now, let $\vec{e}_n$ be the nth standard basis vector in $\mathbb{R}^m$; and $\vec{x}_n \in \mathbb{R}^{d_{model}}$ the nth element in the sequence X, i.e., the nth row vector of X. Extend each $\vec{x}_n^{(0)}$ to a new feature vector $\left(\vec{x}_n^{(0)}, \vec{e}_n\right)$ by concatenation. This results in an extension of the sequence X to a new sequence $X' \in \mathbb{R}^{m \times (d_{model}+m)}$. There any many methods of proceeding from here to encode positional data, we touch on a couple.

One method proposed in [5] is to learn a combined representation $Z \in \mathbb{R}^{m \times d_{model}}$ of the input and its position, using a neural architecture defined by

$$\vec{z}_n = \mathrm{ReLU}(\vec{x}_n W_x + \vec{e}_n W_e) W_z \in \mathbb{R}^{d_{model}},$$

with

$$W_x \in \mathbb{R}^{d_{model} \times k}, W_e \in \mathbb{R}^{m \times k}, W_z \in \mathbb{R}^{k \times d_{model}},$$

where $\vec{z}_n$ are the rows of Z.

A second approach proposed in [5] is to learn distinct representations of the input and output, and then simply sum them.

$$\vec{z}_n = \underbrace{\mathrm{ReLU}(\vec{x}_n W_x) W_{zx}}_{\text{input embedding}} + \underbrace{\mathrm{ReLU}(\vec{e}_n W_e) W_{ze}}_{\text{positional embedding}} \in \mathbb{R}^{d_{model}},$$

with

$$W_x \in \mathbb{R}^{d_{model} \times k}, W_e \in \mathbb{R}^{m \times k}, \text{ and } W_{zx}, W_{ze} \in \mathbb{R}^{k \times d_{model}}.$$

In [10], a somewhat more obscure approach is used to construct the positional embedding part of the representation, utilizing trignometric position embeddings $P \in \mathbb{R}^{m \times d_{model}}$ with row vectors $\vec{p}_n \in \mathbb{R}^{d_{model}}$, each having entries as follows

$$p_{n,2i} = \sin\left(\frac{n}{10000^{\frac{2i}{d}}}\right), \qquad p_{n,2i+1} = \cos\left(\frac{n}{10000^{\frac{2i}{d}}}\right).$$

With this in hand, they use the second approach of [5], fixing the positional embedding to be the P as described above:

$$\vec{z}_n = \underbrace{\mathrm{ReLU}(\vec{x}_n W_x) W_{zx}}_{\text{input embedding}} + \vec{p}_n \in \mathbb{R}^{d_{model}}.$$

The authors of [10] posit that the empirical efficacy of the sinusodial method of positional encoding is equivalent to a learned one. Further, they claim the sinusodial representation generalizes better to sequences longer than the length of the training sequences.

References

[1] Jay Alammar. *The Illustrated Transformer*. URL: `https://jalammar.github.io/illustrated-transformer/` (visited on 05/21/2023) (cit. on pp. 1, 6).

[2] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. "Layer normalization". In: *arXiv preprint arXiv:1607.06450* (2016) (cit. on p. 14).

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate". In: *arXiv preprint arXiv:1409.0473* (2014) (cit. on p. 7).

[4] Jaron Collis. *Glossary of Deep Learning: Word Embedding*. Deeper Learning. Apr. 5, 2023. URL: `https://medium.com/deeper-learning/glossary-of-deep-learning-word-embedding-f90c3cec34ca` (visited on 05/21/2023) (cit. on p. 4).

[5] Jonas Gehring et al. "Convolutional sequence to sequence learning". In: *International conference on machine learning*. PMLR. 2017, pp. 1243–1252 (cit. on pp. 7, 17).
[6] Maxime. *What is a Transformer?* Inside Machine learning. Mar. 5, 2020. URL: https://medium.com/inside-machine-learning/what-is-a-transformer-d07dd1fbec04 (visited on 05/21/2023) (cit. on p. 1).
[7] *Summary of the tokenizers*. URL: https://huggingface.co/docs/transformers/tokenizer_summary (visited on 05/21/2023) (cit. on pp. 2, 3).
[8] *The amazing power of word vectors — the morning paper*. Apr. 21, 2016. URL: https://blog.acolyer.org/2016/04/21/the-amazing-power-of-word-vectors/ (visited on 05/21/2023) (cit. on p. 4).
[9] *Transformers: a Primer*. URL: http://www.columbia.edu/~jsl2239/transformers.html (visited on 05/21/2023) (cit. on p. 6).
[10] Ashish Vaswani et al. "Attention is All you Need". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf (cit. on pp. 1, 2, 5, 7, 10, 11, 17).
[11] *Word Embedding Visual Inspector*. URL: https://ronxin.github.io/wevi/ (visited on 05/21/2023) (cit. on p. 4).
[12] *Word2Vec Tutorial - The Skip-Gram Model · Chris McCormick*. URL: http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/ (visited on 05/21/2023) (cit. on p. 4).
[13] Yonghui Wu et al. "Google's neural machine translation system: Bridging the gap between human and machine translation". In: *arXiv preprint arXiv:1609.08144* (2016) (cit. on p. 7).
[14] Xin Rong. *Word Embedding Explained and Visualized - word2vec and wevi*. Mar. 21, 2016. URL: https://www.youtube.com/watch?v=D-ekE-Wlcds (visited on 05/21/2023) (cit. on p. 4).

(Arav Agarwal) Bowdoin College, Department of Mathematics, Brunswick, ME 04011 USA
Email address: aagarwal@bowdoin.edu