Notes on Transformers (DRAFT)

"A transformer is a deep learning model, common in large language models, with an encoder-decoder architecture that is based on the multi-head attention mechanism."

Paul F. Roysdon, Ph.D.

I. The Concept

(Note: This paper is incomplete, but it is posted as a starting point for the reader.)

A transformer is a deep learning model, common in large language models (LLMs), with an encoder-decoder architecture that is based on the multi-head attention mechanism wherein the calculation of the hidden representation of a token has equal access to any part of a sentence directly. Prior architectures, e.g., Recurrent Neural Networks and Long Short-Term Memory, used recurrent units that increased training time and favored recent information and attenuated distant information contained in a long sequence of words.

The transformer *encoder* architecture input is split into *n-grams* encoded as tokens in a *bag-of-words* model. Each token is converted into both a vector from a *word embedding* table and a *positional encoding* table, thus providing context within the scope of the context window. The parallel *attention mechanism* allows for the signal for key tokens to be amplified, while less important tokens are attenuated. Finally, each attention head uses a *softmax*, each are combined into a *vanilla neural network* with *ReLU* activations, and the result is normalized.

The *encoder* output is the input to the *decoder* architecture that generally follows the same process with the same functions as the encoder architecture.

The 'vanilla' transformer architecture is shown in Fig 1. In the next section we use a simple example to work through each step of the architecture.

II. Tutorial - Encoder

Let's first consider the encoder architecture; see Fig. 2.

A. Dataset Definition

While LLMs use very large datasets, e.g., the ChatGPT dataset is 570 GB, we will use as our dataset the first part of the first sentence from Charles Dickens "A Tale of Two Cities", namely *"It was the best of times, it was the worst of times, it was the age of wisdom,..."*

For the purpose of this simple example, we assume that each sentence fragment (separated by comma) is instead a full sentence. Thus, "It was the best of times, it was the worst of times, it was the age of wisdom,..." simplifies to "It was the best of times. It was the worst of times. It was the age of wisdom." Therefore our dataset (corpus) is:

It was the best of times.
It was the worst of times.

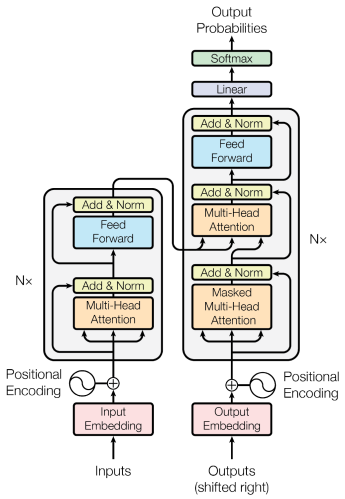

Fig. 1. Vanilla transformer architecture.

It was the age of wisdom.

Note, this is the entirety of our dataset, which is intentional for the following steps to be easy to follow and calculate by hand.

B. Calculate the Vocabulary Size

This step is often performed by a *bag-of-words* model. The vocabulary size is the number of unique words in our dataset. We can simplify the vocabulary by removing punctuation and capital letters to create a *set* (notice the *set notation* below).

{it, was, the, best, of, times}
{it, was, the, worst, of, times}
{it, was, the, age, of, wisdom}

Then, concatenate the sets and remove repeated words: Therefore, the entirety of our dataset (corpus) is:

{ it, was, the, best, of, times, worst, age, wisdom }

Finally, the *vocabulary size* is the count of the dataset words,

$$\text{vocab size} = \text{count}(\text{set}(N)).$$

There are 9 unique words in our dataset.

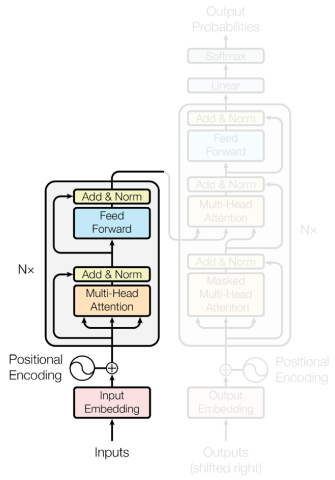

Fig. 2. Encoder.

C. Calculate the Input Encoding

To build the input, or *word* encoding (the first box in Fig. 2), we assign a unique *number* to each unique word.

1	2	3	4	5	6	7	8	9
it	was	the	best	of	times	worst	age	wisdom

Therefore a single token is a single word[1].

D. Calculate Embedding Vectors

Let's select the following sentence from our corpus as input to the transformer encoder:

1	2	3	7	5	6
it	was	the	worst	of	times

The word embedding vectors, $\mathbf{w}$, are dimension 6 because our input is 6 elements[2]. The values of the embedding vectors are between 0 and 1, initialized as a uniform random distribution, see Table I, and updated during training.

TABLE I
INITIAL EMBEDDING VECTORS.

1	2	3	7	5	6
it	was	the	worst	of	times
$\mathbf{w}_1$	$\mathbf{w}_2$	$\mathbf{w}_3$	$\mathbf{w}_4$	$\mathbf{w}_5$	$\mathbf{w}_6$
0.8147	0.27	0.95	0.79	0.67	0.70
0.9058	0.54	0.48	0.95	0.75	0.03
0.1270	0.95	0.80	0.65	0.74	0.27
0.9134	0.96	0.14	0.03	0.39	0.04
0.6324	0.15	0.42	0.84	0.65	0.09
0.0975	0.97	0.91	0.93	0.17	0.82

Therefore,

$$\mathbf{W} = \begin{bmatrix} 0.8147 & 0.27 & 0.95 & 0.79 & 0.67 & 0.70 \\ 0.9058 & 0.54 & 0.48 & 0.95 & 0.75 & 0.03 \\ 0.1270 & 0.95 & 0.80 & 0.65 & 0.74 & 0.27 \\ 0.9134 & 0.96 & 0.14 & 0.03 & 0.39 & 0.04 \\ 0.6324 & 0.15 & 0.42 & 0.84 & 0.65 & 0.09 \\ 0.0975 & 0.97 & 0.91 & 0.93 & 0.17 & 0.82 \end{bmatrix}.$$

[1]Note, ChatGPT uses the equation: 1 token = 0.75 word.
[2]Note, ChatGPT uses a 512-dimension embedding vector for each word.

E. Calculate the Positional Embedding

To build the positional embedding (the circle in Fig. 2), we assign a unique *vector* to each unique word. There are two equations for positional embedding depending on the position of the i-th value of that embedding vector for each word. Even positions use the equation

$$\mathbf{p}_{(\text{pos},2i)} = \sin\left(\frac{\text{pos}}{10000^{2i/d}}\right),$$

while odd positions use the equation

$$\mathbf{p}_{(\text{pos},2i+1)} = \cos\left(\frac{\text{pos}}{10000^{2i/d}}\right).$$

The first word of our input is 'it' corresponding to embedding vector $\mathbf{w}_1$ and a starting positional embedding index 0. Table II outlines the first positional embedding vector $\mathbf{p}_1$.

TABLE II
POSITIONAL EMBEDDING VECTOR FOR THE FIRST INPUT WORD *'it'*.

i	$\mathbf{w}_1$	Position	Equation	$\mathbf{p}_1$
0	0.81	even	$\sin(0/10000^{(2*0/6)})$	0
1	0.90	odd	$\cos(0/10000^{(2*1/6)})$	1
2	0.12	even	$\sin(0/10000^{(2*2/6)})$	0
3	0.91	odd	$\cos(0/10000^{(2*3/6)})$	1
4	0.63	even	$\sin(0/10000^{(2*4/6)})$	0
5	0.09	odd	$\cos(0/10000^{(2*5/6)})$	1

Use the same method for the remaining positional embedding vectors $\mathbf{p}_2, \ldots, \mathbf{p}_5$, that represent the columns of $\mathbf{P}$,

$$\mathbf{P} = \begin{bmatrix} 0 & 0.84 & 0.90 & 0.14 & -0.75 & -0.95 \\ 1 & 0.99 & 0.99 & 0.99 & 0.98 & 0.97 \\ 0 & 0.01 & 0.01 & 0.01 & 0.01 & 0.01 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}.$$

F. Concatenate the Positional & Word Embeddings

The next operation is a matrix add operation of the positional embedding matrix $\mathbf{P}$, and word embedding matrix, $\mathbf{W}$,

$$\mathbf{E} = \mathbf{W} + \mathbf{P}$$

the result is denoted as the embedding matrix $\mathbf{E}$,

$$\mathbf{E} = \begin{bmatrix} 0.8147 & 1.12 & 1.86 & 0.93 & -0.07 & -0.25 \\ 1.9058 & 1.54 & 1.48 & 1.94 & 1.74 & 1.00 \\ 0.1270 & 0.95 & 0.80 & 0.66 & 0.75 & 0.28 \\ 1.9134 & 1.96 & 1.14 & 1.03 & 1.39 & 1.04 \\ 0.6324 & 0.15 & 0.42 & 0.84 & 0.65 & 0.09 \\ 1.0975 & 1.97 & 1.91 & 1.93 & 1.17 & 1.82 \end{bmatrix}.$$

This step is represented by the $\oplus$ symbol in Fig. 2.

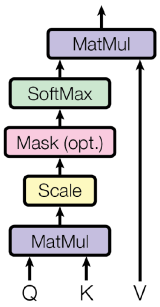

Fig. 3. Single-Head Attention.

G. Single-Head Attention

The *Multi-Head Attention* box in Fig. 2 contains the *single-head* attention operations illustrated in Fig. 3; the multi-head attention simply stacks several single-head attention blocks, as we will see later.

There are three input matrices to each single-head attention module: query, $\mathbf{Q}$, key, $\mathbf{K}$, and value, $\mathbf{V}$; see Fig. 3. Each matrix $\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$ is multiplied by a weight matrix $\mathbf{W}_Q$, $\mathbf{W}_K$, $\mathbf{W}_V$ respectively. Each weight matrix is dimension 6×4, it is initialized from a uniform random distribution with values between 0 and 1, and updated during training. Therefore $\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$ are calculated to be

$$\mathbf{Q} = \mathbf{E} \times \mathbf{W}_Q$$
$$\mathbf{K} = \mathbf{E} \times \mathbf{W}_K$$
$$\mathbf{V} = \mathbf{E} \times \mathbf{W}_V.$$

Let the initial value of the weight matrices be defined as

$$\mathbf{W}_Q = \begin{bmatrix} 0.69 & 0.76 & 0.70 & 0.11 \\ 0.31 & 0.79 & 0.75 & 0.49 \\ 0.95 & 0.18 & 0.27 & 0.95 \\ 0.03 & 0.48 & 0.67 & 0.34 \\ 0.43 & 0.44 & 0.65 & 0.58 \\ 0.38 & 0.64 & 0.16 & 0.22 \end{bmatrix}$$

$$\mathbf{W}_K = \begin{bmatrix} 0.75 & 0.54 & 0.81 & 0.61 \\ 0.25 & 0.13 & 0.24 & 0.47 \\ 0.50 & 0.14 & 0.92 & 0.35 \\ 0.69 & 0.25 & 0.35 & 0.83 \\ 0.89 & 0.84 & 0.19 & 0.58 \\ 0.95 & 0.25 & 0.25 & 0.54 \end{bmatrix}$$

$$\mathbf{W}_V = \begin{bmatrix} 0.91 & 0.07 & 0.56 & 0.31 \\ 0.28 & 0.05 & 0.46 & 0.52 \\ 0.75 & 0.53 & 0.01 & 0.16 \\ 0.75 & 0.77 & 0.33 & 0.60 \\ 0.38 & 0.93 & 0.16 & 0.26 \\ 0.56 & 0.12 & 0.79 & 0.65 \end{bmatrix},$$

then calculate

$$\mathbf{Q} = \begin{bmatrix} 2.59 & 2.12 & 2.48 & 2.66 \\ 4.43 & 5.34 & 5.55 & 4.32 \\ 1.61 & 1.85 & 2.02 & 1.99 \\ 4.08 & 5.04 & 4.94 & 3.70 \\ 1.24 & 1.45 & 1.70 & 1.25 \\ 4.48 & 5.41 & 5.17 & 4.70 \end{bmatrix}$$

$$\mathbf{K} = \begin{bmatrix} 2.18 & 0.99 & 2.91 & 2.27 \\ 6.45 & 3.69 & 4.58 & 5.61 \\ 2.15 & 1.19 & 1.53 & 1.96 \\ 5.48 & 3.19 & 3.99 & 4.76 \\ 1.99 & 1.22 & 1.39 & 1.75 \\ 6.44 & 3.10 & 4.51 & 5.57 \end{bmatrix}$$

$$\mathbf{V} = \begin{bmatrix} 3.01 & 1.73 & 1.11 & 1.53 \\ 6.01 & 4.28 & 3.56 & 3.94 \\ 1.94 & 1.74 & 1.10 & 1.46 \\ 5.08 & 3.10 & 3.43 & 3.49 \\ 1.88 & 1.56 & 0.90 & 1.09 \\ 5.95 & 4.04 & 3.86 & 4.36 \end{bmatrix}.$$

The first step in Fig. 3 is the multiplication of the query and the key transposed, we denote this as attention step 1, or $\mathbf{A}^{(1)}$, where the superscript in parenthesis denotes the step number. Then,

$$\mathbf{A}^{(1)} = \mathbf{Q} \times \mathbf{K}^\intercal,$$

where

$$\mathbf{A}^{(1)} = \begin{bmatrix} 21.07 & 50.92 & 17.17 & 43.59 & 15.92 & 49.36 \\ 41.04 & 98.15 & 33.00 & 84.19 & 30.76 & 94.40 \\ 15.83 & 37.80 & 12.74 & 32.40 & 11.84 & 36.47 \\ 36.76 & 88.46 & 29.71 & 75.88 & 27.74 & 84.95 \\ 11.99 & 28.28 & 9.51 & 24.26 & 8.85 & 27.24 \\ 40.96 & 99.08 & 33.33 & 84.93 & 31.07 & 95.30 \end{bmatrix}.$$

The next step in Fig. 3 scales $\mathbf{A}^{(1)}$ by $\sqrt{d}$ (recall the dimension in this example is 6), thus

$$\mathbf{A}^{(2)} = \mathbf{A}^{(1)}/\sqrt{d},$$

where

$$\mathbf{A}^{(2)} = \begin{bmatrix} 8.60 & 20.78 & 7.01 & 17.79 & 6.50 & 20.15 \\ 16.75 & 40.07 & 13.47 & 34.37 & 12.56 & 38.54 \\ 6.46 & 15.43 & 5.20 & 13.22 & 4.83 & 14.89 \\ 15.01 & 36.11 & 12.13 & 30.98 & 11.32 & 34.68 \\ 4.89 & 11.54 & 3.88 & 9.90 & 3.61 & 11.12 \\ 16.72 & 40.44 & 13.61 & 34.67 & 12.68 & 38.90 \end{bmatrix}.$$

The next step in Fig. 3 is a mask, which we will skip in this example. The mask constrains the model to only consider information prior to a specified point, eliminating the possibility of 'looking' into the future, while determining the importance of each word in the sequence.

The next step in Fig. 3 calculates an element-wise and row-wise *softmax* on $\mathbf{A}^{(2)}$. The softmax function is defined as

$$s(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

where the elements $x_i = (\frac{\mathbf{Q} \times \mathbf{K}}{\sqrt{d}})_i$.

For each row i and column j, the i,j-th element of $\mathbf{A}^{(3)}$ is calculated as

$$\mathbf{A}^{(3)}_{i,j} = \frac{e^{\mathbf{A}^{(2)}_{i,j}}}{\sum_{j=1}^{n} e^{\mathbf{A}^{(2)}_{i,j}}},$$

Using the equation above, the $\mathbf{A}^{(3)}_{1,1}$ element is

$$\mathbf{A}^{(3)}_{1,1} = \frac{e^{8.60}}{e^{8.60} + e^{20.78} + e^{7.01} + e^{17.79} + e^{6.50} + e^{20.15}} \approx 0, \quad \rightarrow \text{see the bold value in } \mathbf{A}^{(3)} \text{ below.}$$

Repeating this calculation for each element

$$\mathbf{A}^{(3)} = \begin{bmatrix} \mathbf{0} & 0.63 & 0 & 0.03 & 0 & 0.33 \\ 0 & 0.81 & 0 & 0 & 0 & 0.17 \\ 1E{-}4 & 0.59 & 0 & 0.06 & 0 & 0.34 \\ 0 & 0.80 & 0 & 0 & 0 & 0.19 \\ 7E{-}4 & 0.54 & 3E{-}4 & 0.10 & 2E{-}4 & 0.35 \\ 0 & 0.82 & 0 & 0 & 0 & 0.17 \end{bmatrix}.$$

The final step in Fig. 3 is a matrix multiplication of $\mathbf{A}^{(3)}$ times the value matrix $\mathbf{V}$,

$$\mathbf{A}^{(4)} = \mathbf{A}^{(3)} \times \mathbf{V} = \begin{bmatrix} 5.96 & 4.16 & 3.66 & 4.07 \\ 6.00 & 4.24 & 3.61 & 4.01 \\ 5.93 & 4.12 & 3.65 & 4.05 \\ 5.99 & 4.23 & 3.62 & 4.02 \\ 5.89 & 4.07 & 3.65 & 4.04 \\ 6.00 & 4.24 & 3.61 & 4.01 \end{bmatrix}.$$

H. Multi-Head Attention

The *Multi-Head Attention* box in Fig. 2 contains the stacking h-many *single-head* attention modules as illustrated in Fig. 4.

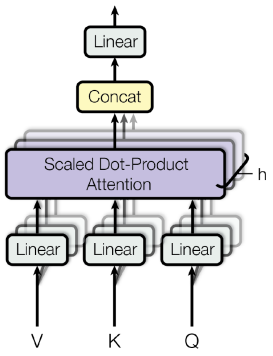

Fig. 4. Multi-Head Attention.

Each single-head attention layer has three inputs: query, key, and value, and each has a linear transformation (i.e., a matrix multiply) by multiplying each input by different and unique weight matrix that is updated during training. The resultant matrix from each attention layer $\mathbf{A}^{(4,1)}, \ldots, \mathbf{A}^{(4,h)}$ (where the second number in the superscript represents each single-head attention layer) is concatenated and another linear transformation is calculated with another weight matrix that is updated during training. Expanding our prior notation to account for h-many single-head attention modules, the unique weight matrices are represented as query weights $\mathbf{W}_{Q,1}, \ldots, \mathbf{W}_{Q,h}$, key weights $\mathbf{W}_{K,1}, \ldots, \mathbf{W}_{K,h}$, and value weights $\mathbf{W}_{V,1}, \ldots, \mathbf{W}_{V,h}$.

In our simple example we assume that we only have one single-head attention layer, i.e., $\mathbf{A}^{(4,1)}$ and therefore skip the concatenation.

Next, we calculate the linear transformation with a new weight matrix for the single-head attention layer, $\mathbf{W}_A$, that is initialized by a uniform random distribution,

$$\mathbf{W}_A = \begin{bmatrix} 0.68 & 0.22 & 0.53 & 0.10 & 0.81 & 0.25 \\ 0.74 & 0.91 & 0.99 & 0.96 & 0.86 & 0.80 \\ 0.45 & 0.15 & 0.07 & 0.01 & 0.08 & 0.43 \\ 0.08 & 0.82 & 0.44 & 0.77 & 0.39 & 0.91 \end{bmatrix}.$$

Then

$$\mathbf{A}^{(5)} = \mathbf{A}^{(4,1)} \times \mathbf{W}_A = \begin{bmatrix} 9.22 & 9.09 & 9.45 & 7.81 & 10.43 & 10.17 \\ 9.27 & 9.11 & 9.51 & 7.85 & 10.50 & 10.17 \\ 9.16 & 9.03 & 9.38 & 7.76 & 10.36 & 10.11 \\ 9.27 & 9.11 & 9.51 & 7.84 & 10.49 & 10.17 \\ 9.09 & 8.96 & 9.30 & 7.69 & 10.28 & 10.05 \\ 9.27 & 9.11 & 9.51 & 7.85 & 10.50 & 10.17 \end{bmatrix}.$$

I. Add & Normalize

The next step in Fig. 2 requires addition of the prior step with the embedding matrix

$$\mathbf{A}^{(6)} = \mathbf{E} + \mathbf{A}^{(5)} = \begin{bmatrix} \mathbf{10.03} & \mathbf{10.21} & \mathbf{11.31} & \mathbf{8.75} & \mathbf{10.35} & \mathbf{9.92} \\ 11.18 & 10.66 & 10.99 & 9.80 & 12.24 & 11.17 \\ 9.29 & 9.99 & 10.19 & 8.42 & 11.11 & 10.40 \\ 11.18 & 11.08 & 10.65 & 8.88 & 11.88 & 11.22 \\ 9.72 & 9.12 & 9.72 & 8.54 & 10.93 & 10.14 \\ 10.37 & 11.08 & 11.43 & 9.78 & 11.67 & 11.99 \end{bmatrix},$$

and normalization of the output of multi-head attention, such that

$$\mathbf{A}^{(7)} = \eta(\mathbf{A}^{(6)}).$$

The normalization function $\eta(\cdot)$ requires a few steps.

First, using $\mathbf{A}^{(6)}$, calculate the mean of the i-th row for columns $j = 1, \ldots, N$,

$$\mu_i = \frac{\sum_{j=1}^{N} x_j}{N}.$$

Calculate the standard deviation of the i-th row for columns $j = 1, \ldots, N$,

$$\sigma_i = \sqrt{\frac{\sum_{j=1}^{N} (x_j - \mu_i)}{N}}.$$

Therefore, given $\mathbf{A}^{(6)}$, the row-wise mean and standard deviation are:

i	μ_i	σ_i
1	**10.09**	**0.82**
2	11.01	0.79
3	9.90	0.93
4	10.82	1.02
5	9.70	0.82
6	11.05	0.83

Notice, the bold row in $\mathbf{A}^{(6)}$ is used to calculate the bold μ and σ values above, and this is repeated for each row.

Now we can define the function $\eta(\cdot)$, such that

$$\eta(\mathbf{A}^{(6)}) = \frac{\mathbf{A}^{(6)}_{i,j} - \mu_i}{\sigma_i + \nu},$$

where the error $\nu = 0.0001$ is a small, non-zero value to eliminate the possibility of a division by zero. Using $\eta(\cdot)$, μ_i, and σ_i, we can calculate each element of $\mathbf{A}^{(7)}$,

$$\mathbf{A}^{(7)} = \eta(\mathbf{A}^{(6)}) = \begin{bmatrix} \mathbf{-0.07} & 0.13 & 1.47 & -1.62 & 0.31 & -0.21 \\ 0.21 & -0.43 & -0.01 & -1.52 & 1.54 & 0.21 \\ -0.65 & 0.09 & 0.30 & -1.58 & 1.29 & 0.53 \\ 0.35 & 0.25 & -0.15 & -1.88 & 1.04 & 0.39 \\ 0.03 & -0.70 & 0.03 & -1.40 & 1.49 & 0.54 \\ -0.81 & 0.03 & 0.44 & -1.52 & 0.73 & 1.12 \end{bmatrix}.$$

Notice, the bold row in $\mathbf{A}^{(6)}$ and the bold values in the table for μ and σ are used to calculate the bold value in $\mathbf{A}^{(7)}$. This is repeated for each element of $\mathbf{A}^{(7)}$.

J. Feed Forward Network

The next step in Fig. 2 requires a feed forward vanilla neural network. While larger models will use multiple linear layers proceeded by a ReLU layer, for our simple example we will use one linear layer $\mathbf{L}$, and one ReLU layer, $\mathbf{R}$.

The linear layer weight matrix is $\mathbf{W}_L$ and the bias vector $\mathbf{b}_L$ are both initialized by a uniform random distribution such that

$$\mathbf{W}_L = \begin{bmatrix} 0.18 & 0.54 & 0.40 & 0.41 & 0.33 & 0.24 \\ 0.26 & 0.14 & 0.07 & 0.04 & 0.90 & 0.40 \\ 0.14 & 0.85 & 0.23 & 0.90 & 0.36 & 0.09 \\ 0.13 & 0.62 & 0.12 & 0.94 & 0.11 & 0.13 \\ 0.86 & 0.35 & 0.18 & 0.49 & 0.78 & 0.94 \\ 0.57 & 0.51 & 0.24 & 0.48 & 0.38 & 0.95 \end{bmatrix},$$

and

$$\mathbf{b}_L^\intercal = \begin{bmatrix} 0.57 & 0.05 & 0.23 & 0.35 & 0.82 & 0.01 \end{bmatrix}.$$

The linear layer is calculated by

$$\mathbf{L} = \mathbf{A}^{(7)} \times \mathbf{W}_L + \mathbf{b}_L = \begin{bmatrix} \mathbf{0.73} & 0.27 & 0.37 & 0.16 & 1.44 & 0.06 \\ 1.75 & -0.19 & 0.43 & -0.16 & 1.61 & 1.34 \\ 1.74 & -0.27 & 0.22 & -0.23 & 1.84 & 1.45 \\ 1.55 & -0.44 & 0.41 & -0.70 & 1.86 & 1.29 \\ 1.82 & -0.06 & 0.43 & 0.03 & 1.43 & 1.48 \\ 1.58 & -0.11 & 0.23 & -0.11 & 1.58 & 1.44 \end{bmatrix},$$

The last step in the feed-forward block is the calculation of the matrix $\mathbf{F}$ wherein the ReLU activation function is used to calculate each entry of $\mathbf{F}$, such that

$$\mathbf{F}_{i,j} = \text{ReLU}(\mathbf{L}_{i,j}) = \max(0, \mathbf{L}_{i,j}).$$

Therefore,

$$\mathbf{F} = \begin{bmatrix} \mathbf{0.73} & 0.27 & 0.37 & 0.16 & 1.44 & 0.06 \\ 1.75 & 0 & 0.43 & 0 & 1.61 & 1.34 \\ 1.74 & 0 & 0.22 & 0 & 1.84 & 1.45 \\ 1.55 & 0 & 0.41 & 0 & 1.86 & 1.29 \\ 1.82 & 0 & 0.43 & 0.03 & 1.43 & 1.48 \\ 1.58 & 0 & 0.23 & 0 & 1.58 & 1.44 \end{bmatrix}.$$

Notice the first element of $\mathbf{L}$ (in bold) is used to calculate the first element of $\mathbf{F}$ (also bold).

K. Add & Normalize

The last step of the encoder architecture in Fig. 2 is, again, an add and normalize operation. The same steps above are repeated. This time

$$\mathbf{A}^{(8)} = \mathbf{A}^{(7)} + \mathbf{F},$$

then

$$\mathbf{A}^{(9)} = \eta(\mathbf{A}^{(8)}) = \begin{bmatrix} 0.11 & -0.07 & 1.07 & -1.59 & 1.00 & -0.53 \\ 0.65 & -0.75 & -0.25 & -1.39 & 1.34 & 0.40 \\ 0.13 & -0.48 & -0.21 & -1.51 & 1.39 & 0.68 \\ 0.62 & -0.35 & -0.35 & -1.62 & 1.21 & 0.49 \\ 0.58 & -0.93 & -0.23 & -1.32 & 1.22 & 0.68 \\ -0.02 & -0.51 & -0.08 & -1.54 & 1.00 & 1.16 \end{bmatrix},$$

where $\mathbf{A}^{(9)}$ is the input to the *decoder* architecture; see Fig. 5.

III. Tutorial - Decoder

Now that we have completed the *encoder* architecture, let's consider the *decoder* architecture; see Fig. 5. We will skip most of the decoder blocks because the functions are the same as the encoder, with a few exceptions that we cover in this section.

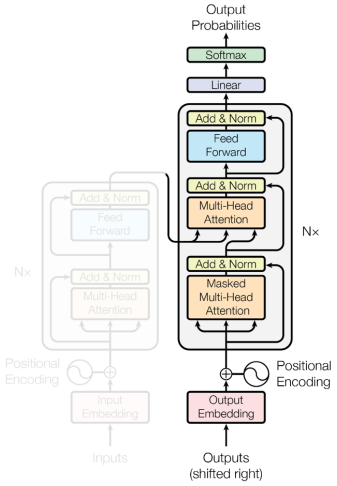

Fig. 5. Decoder.

A. Encoder Input to Decoder

The output of the encoder architecture, $\mathbf{A}^{(9)}$, serve as both the query $\mathbf{Q}$ and key $\mathbf{K}$ matrices for the second multi-head attention layer in the decoder architecture, while the value matrix $\mathbf{V}$ is the result of the add and normalization step in the first part of the decoder; see Fig. 5.

B. Output Embedding

Recall, the input to the *encoder* is *"it was the **best** of times"*. The input to the *decoder* is the predicted text, *"it was the **worst** of times"*

Unlike the encoder input, the decoder needs 'start' and 'end' tokens to learn the sentence completion logic of beginning and termination, we denoted this with tokens in red:

<start> it was the worst of times <end>

The corpus is appended with the new tokens, and the bag-of-words model is recalculated[3]. Similar to the word embedding vector, we now calculate the embedding vectors for the input to the decoder, including the new 'start' and 'end' tokens.

{ it, was, the, . . . , wisdom, <start>, <end> }

10
<start>
$\mathbf{w}_1$
0.8147
0.9058
0.1270
0.9134
0.6324
0.0975

C. Masked Multi-Head Attention

Assume that we have two single-head attention modules, following the same steps listed above for single-head attention, with outputs

$$\mathbf{A}^{(\cdot,1)} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \quad \mathbf{A}^{(\cdot,2)} = \begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \\ 2 & 2 & 2 \end{bmatrix}.$$

The mask constrains the transformer to focus on certain portions of the data, for example

$$\mathbf{A}^{(\cdot,1)}_{masked} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix}, \quad \mathbf{A}^{(\cdot,2)}_{masked} = \begin{bmatrix} 0 & 2 & 2 \\ 0 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix}.$$

Recall from Fig. 4, the second to last step in multi-head attention is a concatenation. While each attention module provide unique information about the input sentence, the concatenation of $\mathbf{A}^{(\cdot,1)}_{masked}$ and $\mathbf{A}^{(\cdot,2)}_{masked}$ combine the information, providing more complete information based on different aspects of the input data. For example, if you read two books, one book on ensemble methods (e.g., decision trees and random forests) and the other book on regression (e.g., linear and logistic regression), you will have a more complete understanding of machine learning than you would if you only consider one source of information. In our example, concatenation produces

$$\mathbf{A}^{(\cdot,1)}_{concat.} = \begin{bmatrix} 0 & 2 & 2 \\ 1 & 0 & 2 \\ 1 & 1 & 0 \end{bmatrix}.$$

Notice the combination of information.

[3]Typically, we include these tokens in the original corpus to avoid redundant calculations

D. Calculating the Predicted Word

In Fig. 5, three steps remain: add and normalize, linear layer, and softmax. Let $\mathbf{A}^{(10)}$ denote the result of the add and normalize step, where

$$\mathbf{A}^{(10)} = \begin{bmatrix} \mathbf{0.18} & \mathbf{0.54} & \mathbf{0.40} & \mathbf{0.41} & 0.33 & 0.24 \\ 0.26 & 0.14 & 0.07 & 0.04 & 0.90 & 0.40 \\ 0.14 & 0.85 & 0.23 & 0.90 & 0.36 & 0.09 \\ 0.13 & 0.62 & 0.12 & 0.94 & 0.11 & 0.13 \\ 0.86 & 0.35 & 0.18 & 0.49 & 0.78 & 0.94 \\ 0.57 & 0.51 & \mathbf{0.24} & \mathbf{0.48} & \mathbf{0.38} & \mathbf{0.95} \end{bmatrix}.$$

The linear layer requires a vector input, so we 'flatten' $\mathbf{A}^{(10)}$ by serially assembling the transpose of each row, such that

$$\mathbf{A}^{(10)}_{flat} = \begin{bmatrix} \mathbf{A}^{(10)\intercal}_{row1} \\ \mathbf{A}^{(10)\intercal}_{row2} \\ \vdots \\ \mathbf{A}^{(10)\intercal}_{row5} \\ \mathbf{A}^{(10)\intercal}_{row6} \end{bmatrix} = \begin{bmatrix} 0.18 \\ 0.54 \\ 0.40 \\ 0.41 \\ \vdots \\ 0.24 \\ 0.48 \\ 0.38 \\ 0.95 \end{bmatrix}.$$

Notice the bold values in $\mathbf{A}^{(10)}$ are serialized in $\mathbf{A}^{(10)}_{flat}$ with dimension 1×36.

The linear layer is a simple multiplication

$$\boldsymbol{\lambda} = \mathbf{A}^{(10)\intercal}_{flat} \times \mathbf{W}_v,$$

where $\mathbf{W}_v$ is the weight matrix representing the vocabulary size with dimensions $n \times m$. In our example $n = 6$ the input size, and $m = 9$ the vocabulary size. Also notice the transpose, i.e., $\mathbf{A}^{(10)\intercal}_{flat}$, in the above equation; a reminder that the rows are transposed to columns for the 'flattening' operation. The result $\boldsymbol{\lambda}$ is a vector of *logits* that correspond to the vocabulary, such that

index	1	2	· · ·	7	5	6
words	it	was	· · ·	worst	age	wisdom
logits	1.20	1.27	· · ·	5.67	0.70	1.95

The next step calculates the softmax, as we saw earlier, resulting in the *Output Probabilities* shown below; see the last step in Fig. 5.

index	1	2	· · ·	7	5	6
words	it	was	· · ·	**worst**	age	wisdom
probabilities	0.20	0.05	· · ·	**0.98**	0.10	0.01

Given the input sentence "it was the best of times, it was the", the decoder should output the next word "worst" based on the probability 0.98, shown in the table above.

The word "worst" is then used as the input word for the decoder to predict the next word, until the <end> token is reached.

IV. Conclusion

While the transformer architecture may appear to be complicated, we have shown that the operations are actually quite simple. A –from scratch– code implementation is provided at the end of this chapter to emphasize the true simplicity of this approach. The challenge is obtaining enough data and computational power to provide meaningful results with LLMs.

V. Advantages & Limitations

- ✓ Relatively efficient.
- ✓ Best result when datasets are very large, e.g., >1 GB.
- ✓ Unlike RNNs and LSTMs, transformers do not have memory fade.
- ✓ Able to handle noisy data and outliers.
- ✗ Slow to train on large datasets.
- ✗ Difficult architecture to understand and implement.
- ✗ Sensitive to bias in the data.

VI. Simple Code Example

Now that we have completed the tutorial, we can show that this transformer is easily implemented –**from scratch**– in 46 lines of code, and roughly 50 lines of code for the supporting functions. Please note, this is not a complete transformer, just the steps we covered in the chapter.

```
% Main run file

% create the bag-of-words
str = "It was the best of times,
       it was the worst of times,
       it was the age of wisdom,";
newStr = tokenizedDocument(str);
newStr = erasePunctuation(newStr);
newStr = lower(newStr);
bag = bagOfWords(newStr);
% create the input for the encoder
str2 = "It was the worst of times,";
newStr2 = tokenizedDocument(str2);
newStr2 = erasePunctuation(newStr2);
newStr2 = lower(newStr2);
% calculate the length of the input string
d_k = length(newStr2.Vocabulary);
% initialize the word embedding matrix
W = rand(d_k,d_k);
% calculate the position encoding and embedding matrix
P = posenc(d_k);
E = P + W;
% initialze the weight matrices
W_Q = rand(d_k,4);
W_K = rand(d_k,4);
W_V = rand(d_k,4);
% calculate the query, key, and value matrices
Q = E * W_Q;
K = E * W_K;
V = E * W_V;
% initialize the weight matrix and error
W_A = rand(4,d_k);
nu = 0.0001; % error
% calculate the attention steps
A_1 = Q * K'; % linear transformation
A_2 = A_1 .* (1/sqrt(d_k)); % scaling
A_3 = softmax(A_2,d_k); % softmax
A_4 = A_3 * V; % linear transformation
A_5 = A_4 * W_A; % linear transformation
A_7 = addnorm(A_5,E,nu,d_k); %  add and normaliaztion
% initialize the weight and bias for the linear layer
W_L = rand(d_k,d_k);
b_L = rand(1,d_k);
% calculate the linear layer
L = linearlayer(A_7,W_L,b_L,d_k);
% calculate the feed forward network
F = feedforward(L,d_k);
% calculate the final add and normalization
A_9 = addnorm(F,A_7,nu,d_k);
```

```
function x = isodd(number)
% isodd(number)
%
% returns 1 if the number is Odd, 0 if it is even.
% Divide input by 2:
a = number/2;
whole = floor(a);
part = a-whole;
if part > 0
    x = 1;
else
    x = 0;
end
end
```

```
function B = posenc(d)
B = zeros(d,d);
for jj=0:d-1
    for ii = 0:d-1
        if ~isodd(ii)
            B(ii+1,jj+1) = sin(jj/10000^(2*ii/d));
        else
            B(ii+1,jj+1) = cos(jj/10000^(2*ii/d));
        end
    end
end
end
```

```
function B = softmax(A,d)
B = zeros(d,d);
for ii = 1:d
    for jj = 1:d
        den = 0;
        for kk = 1:d
            den = den + exp(A(ii,kk));
        end
        B(ii,jj) = exp(A(ii,jj)) / den;
    end
```

```
end
```

```
function B = feedforward(A,d)
B = zeros(d,d);
for ii = 1:d
    for jj = 1:d
        B(ii,jj) = relu(A(ii,jj));
    end
end
end
```

```
function C = addnorm(A,B,nu,d)
AB = A + B;
C = zeros(d,d);
for ii = 1:d
    mu = mean(AB(ii,:));
    sigma = std(AB(ii,:));
    for jj = 1:d
        C(ii,jj) = (AB(ii,jj) - mu) / (sigma + nu);
    end
end
end
```

```
function out = relu(in)
if in > 0
    out = in;
else % x <= 0
    out = 0;
end
end
```

```
function B = linearlayer(X,W,b,d)
B = X * W;
for ii = 1:d
    B(ii,:) = B(ii,:) + b;
end
end
```

```
end
```