Transformer in NYC (created from phtofunia)

Solving Transformer by Hand: A Step-by-Step Math Example

Fareed Khan · Follow
Published in Level Up Coding · 13 min read · Dec 18, 2023

I have already written a detailed blog on how transformers work using a very small sample of the dataset, which will be my best blog ever because it has elevated my profile and given me the motivation to write more. However, that blog is incomplete as it only covers **20% of the transformer architecture** and contains numerous calculation errors, as pointed out by readers. After a considerable amount of time has passed since that blog, I will be revisiting the topic in this new blog.

My previous blog on transformer architecture (**covers only 20%**):

I plan to explain the transformer again in the same manner as I did in my previous blog **(for both coders and non-coders)**, providing a complete guide with a step-by-step approach to understanding how they work.

Table of Contents

Step 1 — Defining our Dataset

The dataset used for creating ChatGPT is **570 GB.** On the other hand, for our purposes, we will be using a very small dataset to perform numerical calculations visually.

Dataset (corpus)

I drink and I know things.
When you play the game of thrones, you win or you die.
The true enemy won't wait out the storm, He brings the storm.

Our entire dataset containing only three sentences

Our entire dataset contains only three sentences, all of which are dialogues taken from a TV show. Although our dataset is cleaned, in real-world scenarios like ChatGPT creation, cleaning a 570 GB dataset requires a significant amount of effort.

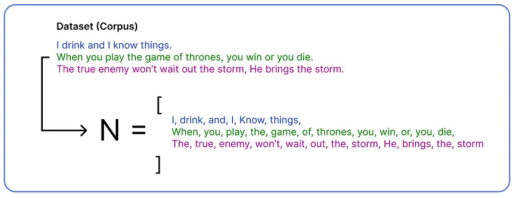
Step 2— Finding Vocab Size

The vocabulary size determines the total number of **unique words** in our dataset. It can be calculated using the below formula, where **N** is the total number of words in our dataset.

$$vocab\ size = count(set(N))$$

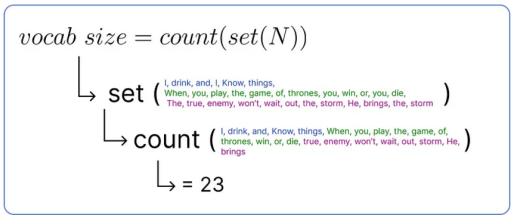
vocab_size formula where N is total number of words

In order to find N, we need to break our dataset into individual words.



calculating variable **N**

After obtaining N, we perform a set operation to remove duplicates, and then we can count the unique words to determine the vocabulary size.



finding vocab size

Therefore, the vocabulary size is **23**, as there are **23** unique words in our dataset.

Step 3 — Encoding

Now, we need to assign a unique number to each unique word.

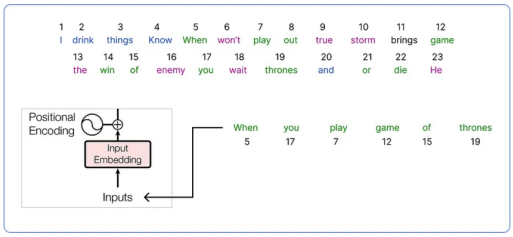
encoding our unique words

As we have considered a single token as a single word and assigned a number to it, ChatGPT has considered a portion of a word as a single token using this formula: `1 Token = 0.75 Word`

After encoding our entire dataset, it's time to select our input and start working with the transformer architecture.

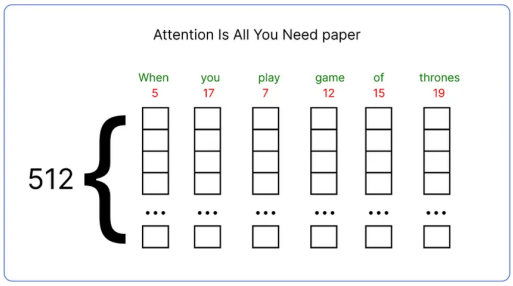
Step 4 — Calculating Embedding

Let's select a sentence from our corpus that will be processed in our transformer architecture.



Input sentence for transformer

We have selected our input, and we need to find an embedding vector for it. The original paper uses a **512-dimensional embedding vector** for each input word.



Original Paper uses 512 dimension vector

Since, for our case, we need to work with a smaller dimension of embedding vector to visualize how the calculation is taking place. So, we will be using a dimension of `6` for the embedding vector.

When	you	play	game	of	thrones
5	17	7	12	15	19

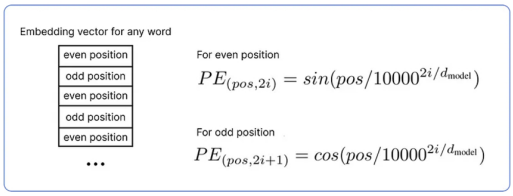
e1	e2	e3	e4	e5	e6
0.79	0.38	0.01	0.12	0.88	0.6
0.6	0.12	0.51	0.6	0.41	0.33
0.96	0.06	0.27	0.65	0.79	0.75
0.64	0.79	0.31	0.22	0.62	0.48
0.97	0.9	0.56	0.07	0.5	0.94
0.2	0.74	0.59	0.37	0.7	0.21

Embedding vectors of our input

These values of the embedding vector are between 0 and 1 and are filled randomly in the beginning. They will later be updated as our transformer starts understanding the meanings among the words.

Step 5 — Calculating Positional Embedding

Now we need to find positional embeddings for our input. There are two formulas for positional embedding depending on the position of the ith value of that embedding vector for each word.



Positional Embedding formula

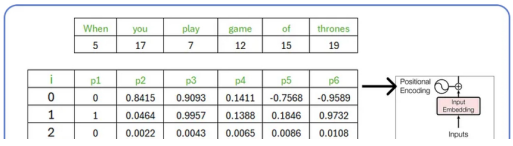
As you do know, our input sentence is **"when you play the game of thrones"** and the starting word is **"when"** with a starting index (POS) value is `0`, having a dimension (d) of `6`. For `i` from `0 to 5`, we calculate the positional embedding for our first word of the input sentence.

When
5

i	e1	Position	Formula	p1
0	0.79	Even	$sin(0/10000^{(2*0/6)})$	0
1	0.6	Odd	$cos(0/10000^{(2*1/6)})$	1
2	0.96	Even	$sin(0/10000^{(2*2/6)})$	0
3	0.64	Odd	$cos(0/10000^{(2*3/6)})$	1
4	0.97	Even	$sin(0/10000^{(2*4/6)})$	0
5	0.2	Odd	$cos(0/10000^{(2*5/6)})$	1
d (dim)	6			
POS	0			

Positional Embedding for word: **When**

Similarly, we can calculate positional embedding for all the words in our input sentence.



When	you	play	game	of	thrones
5	17	7	12	15	19

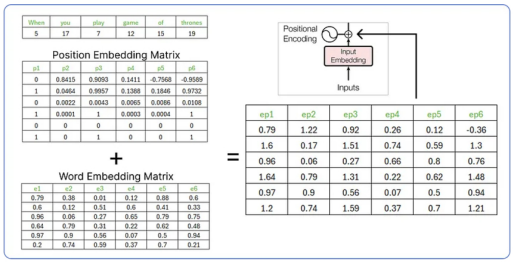
i	p1	p2	p3	p4	p5	p6
0	0	0.8415	0.9093	0.1411	-0.7568	-0.9589
1	1	0.0464	0.9957	0.1388	0.1846	0.9732
2	0	0.0022	0.0043	0.0065	0.0086	0.0108

3	1	0.0001	1	0.0003	0.0004	1
4	0	0	0	0	0	0
5	1	0	1	0	0	1
d (dim)	6	6	6	6	6	6
POS	0	1	2	3	4	5

Calculating Positional Embeddings of our input **(The calculated values are rounded)**

Step 6 — Concatenating Positional and Word Embeddings

After calculating positional embedding, we need to add word embeddings and positional embeddings.

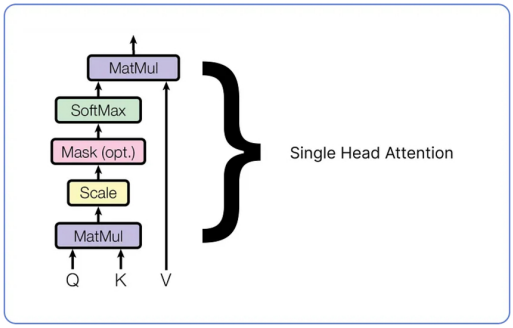


concatenation step

This resultant matrix from combining both matrices (**Word embedding matrix** and **positional embedding matrix**) will be considered as an input to the encoder part.

Step 7 — Multi Head Attention

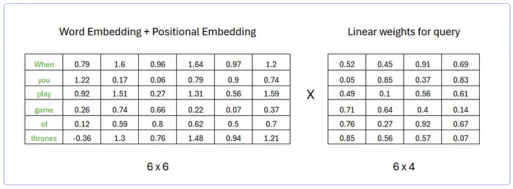
A multi-head attention is comprised of many single-head attentions. It is up to us how many single heads we need to combine. For example, LLaMA LLM from Meta has used 32 single heads in the encoder architecture. Below is the illustrated diagram of how a single-head attention looks like.



Single Head attention in Transformer

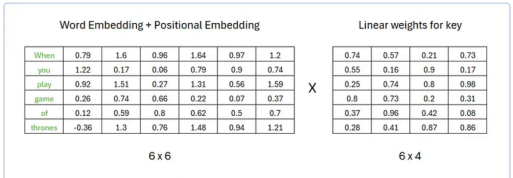
There are three inputs: **query**, **key**, and **value**. Each of these matrices is obtained by multiplying a different set of weights matrix from the **Transpose** of same matrix that we computed earlier by adding the word embedding and positional embedding matrix.

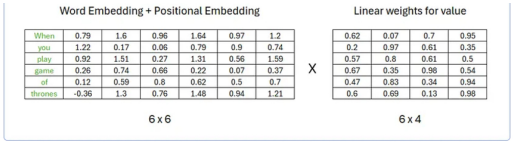
Let's say, for computing the query matrix, the set of weights matrix must have the number of rows the same as the number of columns of the transpose matrix, while the columns of the weights matrix can be any; for example, we suppose `4` columns in our weights matrix. The values in the weights matrix are between `0 and 1` randomly, which will later be updated when our transformer starts learning the meaning of these words.



calculating Query matrix

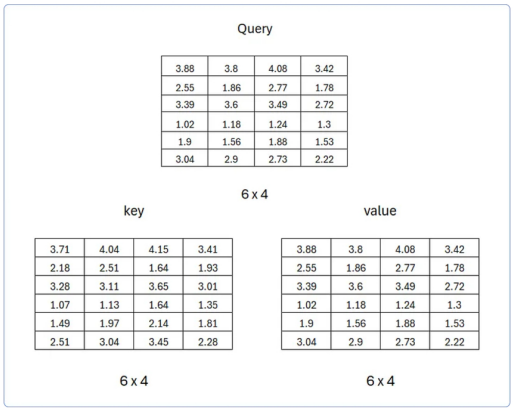
Similarly, we can compute the **key** and **value** matrices using the same procedure, but the values in the weights matrix must be different for both.





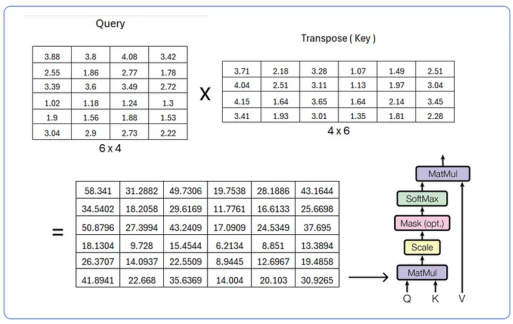
Calculating Key and Value Matrices

So, after multiplying matrices, the resultant **query**, **key**, and **values** are obtained:



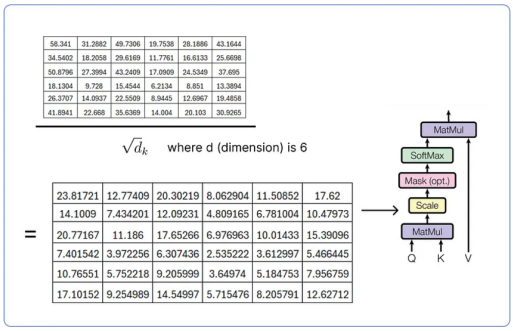
Query, Key, Value matrices

Now that we have all three matrices, let's start calculating single-head attention step by step.



matrix multiplication between Query and Key

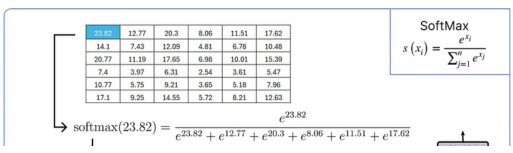
For scaling the resultant matrix, we have to reuse the dimension of our embedding vector, which is `6`.

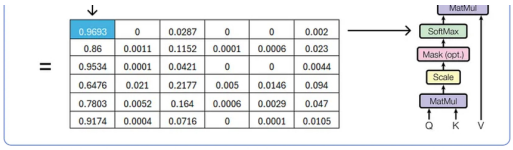


scaling the resultant matrix with dimension 5

The next step of **masking is optional**, and we won't be calculating it. Masking is like telling the model to focus only on what's happened before a certain point and not peek into the future while figuring out the importance of different words in a sentence. It helps the model understand things in a step-by-step manner, without cheating by looking ahead.

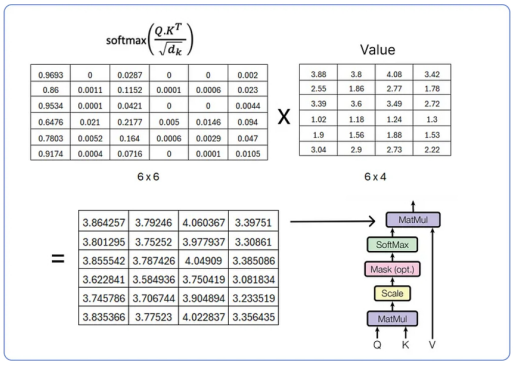
So now we will be applying the **softmax** operation on our scaled resultant matrix.





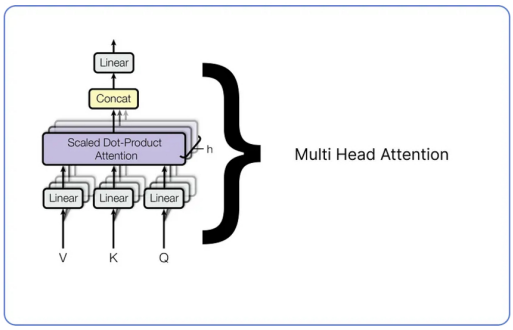
Applying softmax on resultant matrix

Doing the final multiplication step to obtain the resultant matrix from single-head attention.



calculating the final matrix of single head attention

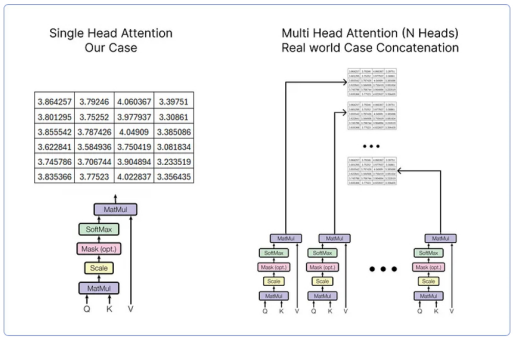
We have calculated single-head attention, while multi-head attention comprises many single-head attentions, as I stated earlier. Below is a visual of how it looks like:



Multi Head attention in Transformer

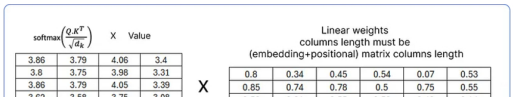
Each single-head attention has three inputs: **query**, **key**, and **value**, and each three have a different set of weights. Once all single-head attentions output their resultant matrices, they will all be concatenated, and the final concatenated matrix is once again transformed linearly by multiplying it with a set of weights matrix initialized with random values, which will later get updated when the transformer starts training.

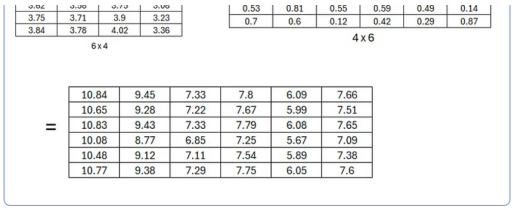
Since, in our case, we are considering a single-head attention, but this is how it looks if we are working with multi-head attention.



Single Head attention vs Multi Head attention

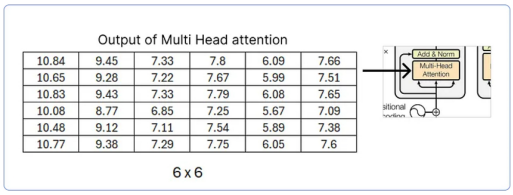
In either case, whether it's single-head or multi-head attention, the resultant matrix needs to be once again transformed linearly by multiplying a set of weights matrix.





normalizing single head attention matrix

Make sure the linear set of weights matrix number of columns must be equal to the matrix that we computed earlier (**word embedding + positional embedding**) matrix number of columns, because the next step, we will be adding the resultant normalized matrix with (**word embedding + positional embedding**) matrix.

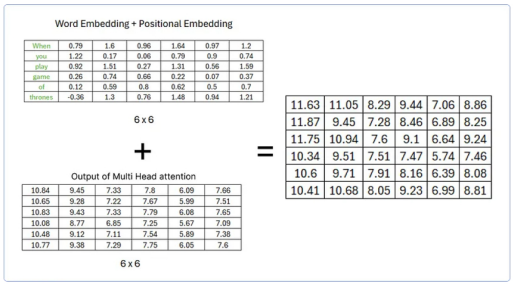


Output matrix of multi head attention

As we have computed the resultant matrix for multi-head attention, next, we will be working on adding and normalizing step.

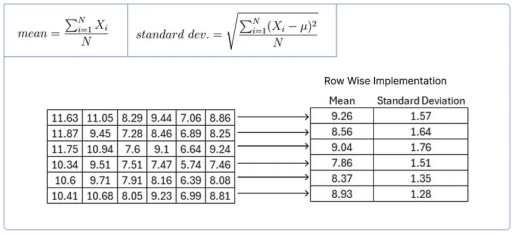
Step 8 — Adding and Normalizing

Once we obtain the resultant matrix from multi-head attention, we have to add it to our original matrix. Let's do it first.



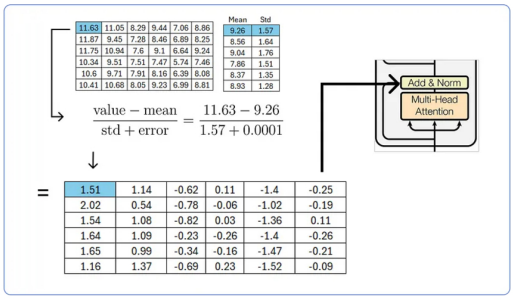
Adding matrices to perform add and norm step

To normalize the above matrix, we need to compute the mean and standard deviation row-wise for each row.



calculating meand and std.

we subtract each value of the matrix by the corresponding row mean and divide it by the corresponding standard deviation.

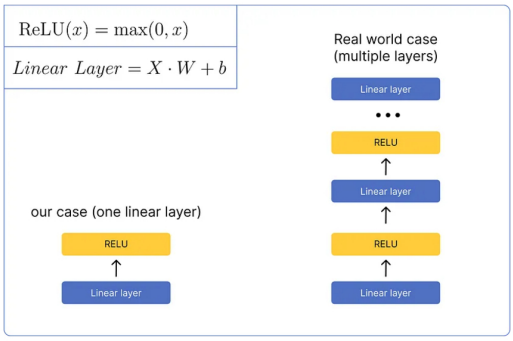


normalizing the resultant matrix

Adding a small value of error prevents the denominator from being zero and avoids making the entire term infinity.

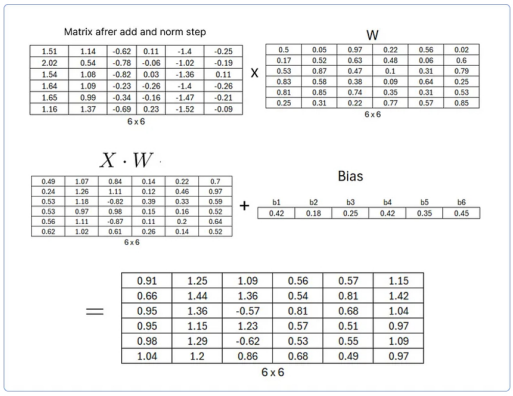
Step 9 — Feed Forward Network

After normalizing the matrix, it will be processed through a feedforward network. We will be using a very basic network that contains only one linear layer and one ReLU activation function layer. This is how it looks like visually:



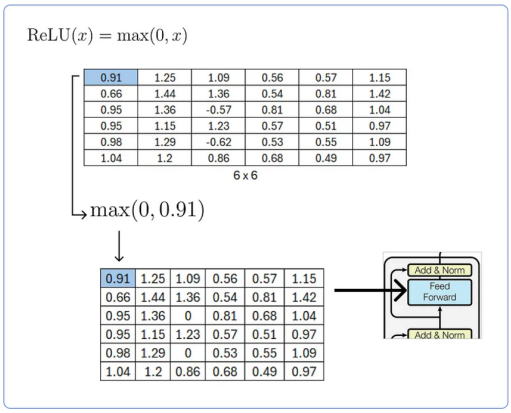
Feed Forward network comparison

First, we need to calculate the linear layer by multiplying our last calculated matrix with a random set of weights matrix, which will be updated when the transformer starts learning, and adding the resultant matrix to a bias matrix that also contains random values.



Calculating Linear Layer

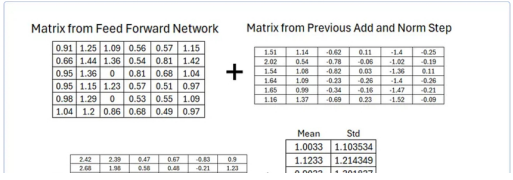
After calculating the linear layer, we need to pass it through the ReLU layer and use its formula.

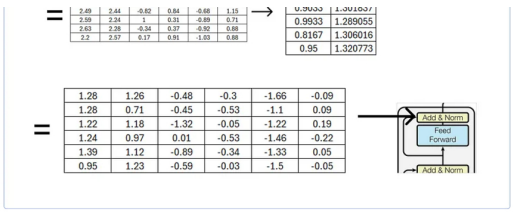


Calculating ReLU Layer

Step 10 — Adding and Normalizing Again

Once we obtain the resultant matrix from feed forward network, we have to add it to the matrix that is obtained from previous add and norm step, and then normalizing it using the row wise mean and standard deviation.





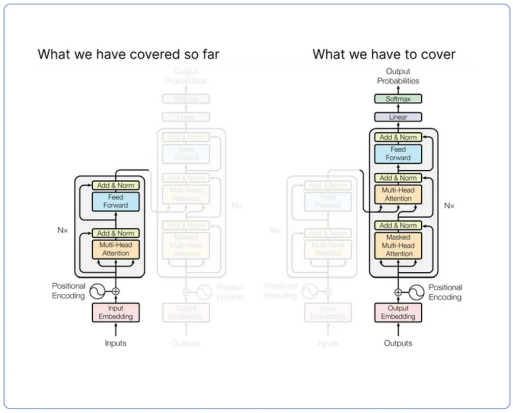
Add and Norm after Feed Forward Network

The output matrix of this add and norm step will serve as the query and key matrix in one of the multi-head attention mechanisms present in the decoder part, which you can easily understand by tracing outward from the add and norm to the decoder section.

Step 11 — Decoder Part

The good news is that up until now, we have calculated **Encoder part**, all the steps that we have performed, from encoding our dataset to passing our matrix through the feedforward network, are unique. It means we haven't calculated them before. But from now on, all the upcoming steps that is the remaining architecture of the transformer (**Decoder part**) are going to involve similar kinds of matrix multiplications.

Take a look at our transformer architecture. What we have covered so far and what we have to cover yet:

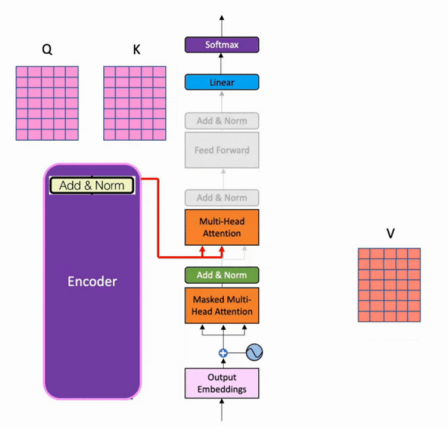


Upcoming steps illustration

We won't be calculating the entire decoder because most of its portion contains similar calculations to what we have already done in the encoder. Calculating the decoder in detail would only make the blog lengthy due to repetitive steps. Instead, we only need to focus on the calculations of the input and output of the decoder.

When training, there are two inputs to the decoder. One is from the encoder, where the output matrix of the last add and norm layer serves as the **query** and **key** for the second multi-head attention layer in the decoder part. Below is the visualization of it (from **batool haider**):

Top highlight



Visualization is from **Batool Haider**

While the value matrix comes from the decoder after the first **add and norm** step.

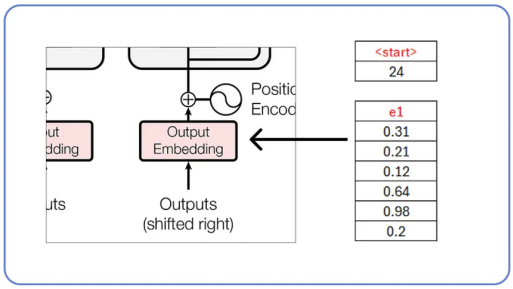
The second input to the decoder is the predicted text. If you remember, our input to the encoder is `when you play game of thrones` so the input to the

decoder is the predicted text, which in our case is `you win or you die` .

But the predicted input text needs to follow a standard wrapping of tokens that make the transformer aware of where to start and where to end.

input comparison of encoder and decoder

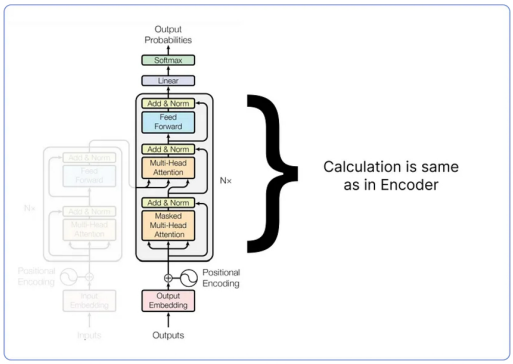
Where `<start>` and `<end>` are two new tokens being introduced. Moreover, the decoder takes one token as an input at a time. It means that `<start>` will be served as an input, and `you` must be the predicted text for it.



Decoder input **<start>** word

As we already know, these embeddings are filled with random values, which will later be updated during the training process.

Compute rest of the blocks in the same way that we computed earlier in the encoder part.



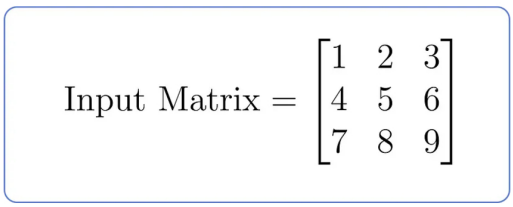
Calculating Decoder

Before diving into any further details, we need to understand what masked multi-head attention is, using a simple mathematical example.

Step 12 — Understanding Mask Multi Head Attention

In a Transformer, the masked multi-head attention is like a spotlight that a model uses to focus on different parts of a sentence. It's special because it doesn't let the model cheat by looking at words that come later in the sentence. This helps the model understand and generate sentences step by step, which is important in tasks like talking or translating words into another language.

Suppose we have the following input matrix, where each row represents a position in the sequence, and each column represents a feature:



inpur matrix for masked multi head attentions

Now, let's understand the masked multi-head attention components having two heads:

1. **Linear Projections (Query, Key, Value):** Assume the linear projections for each head: **Head 1:** $Wq1, Wk1, Wv1$ and **Head 2:** $Wq2, Wk2, Wv2$
2. **Calculate Attention Scores:** For each head, calculate attention scores using the dot product of Query and Key, and apply the mask to prevent attending to future positions.
3. **Apply Softmax:** Apply the softmax function to obtain attention weights.
4. **Weighted Summation (Value):** Multiply the attention weights by the Value to get the weighted sum for each head.
5. **Concatenate and Linear Transformation:** Concatenate the outputs from both heads and apply a linear transformation.

Let's do a simplified calculation:

Assuming two conditions

- $Wq1 = Wk1 = Wv1 = Wq2 = Wk2 = Wv2 = I$, the identity matrix.
- Q=K=V=Input Matrix

Head 1:

$$Q_1 = K_1 = V_1 = \text{Input Matrix}$$
$$A_1 = Q_1 \cdot K_1^T$$
$$A_1 = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$
$$A_1 = \begin{bmatrix} 1 & 0 & 0 \\ 4 & 5 & 0 \\ 7 & 8 & 9 \end{bmatrix} \quad \text{(Masked)}$$
$$W_1 = \text{softmax}(A_1)$$
$$O_1 = W_1 \cdot V_1$$

Head 2:

$$Q_2 = K_2 = V_2 = \text{Input Matrix}$$
$$A_2 = Q_2 \cdot K_2^T$$
$$A_2 = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$
$$A_2 = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 5 & 6 \\ 0 & 0 & 9 \end{bmatrix} \quad \text{(Masked)}$$
$$W_2 = \text{softmax}(A_2)$$
$$O_2 = W_2 \cdot V_2$$

Concatenate and Linear Transformation:

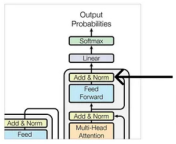
$$\text{Concatenate}([O_1, O_2])$$
(Apply Learnable Linear Transformation)

Mask Multi Head Attention (**Two Heads**)

The concatenation step combines the outputs from the two attention heads into a single set of information. Imagine you have two friends who each give you advice on a problem. Concatenating their advice means putting both pieces of advice together so that you have a more complete view of what they suggest. In the context of the transformer model, this step helps capture different aspects of the input data from multiple perspectives, contributing to a richer representation that the model can use for further processing.

Step 13 — Calculating the Predicted Word

The output matrix of the last add and norm block of the decoder must contain the same number of rows as the input matrix, while the number of columns can be any. Here, we work with `6`.

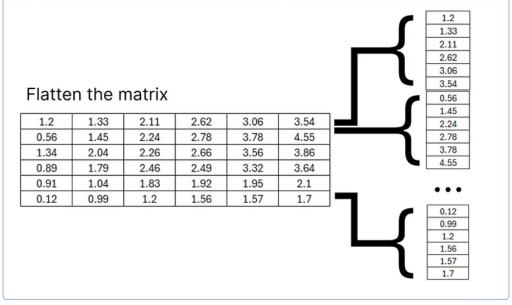


Rows must be 6 while columns can be any length

1.2	1.33	2.11	2.62	3.06	3.54
0.56	1.45	2.24	2.78	3.78	4.55
1.34	2.04	2.26	2.66	3.56	3.86
0.89	1.79	2.46	2.49	3.32	3.64
0.91	1.04	1.83	1.92	1.95	2.1
0.12	0.99	1.2	1.56	1.57	1.7

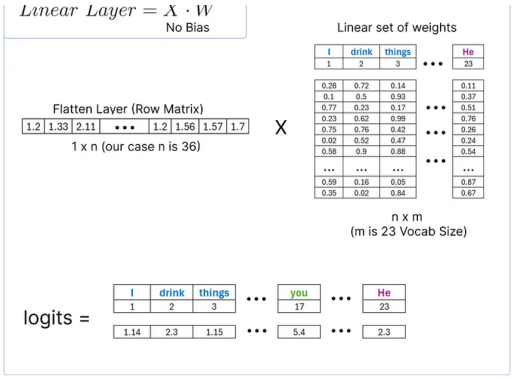
Add and Norm output of decoder

The last **add and norm block** resultant matrix of the decoder must be flattened in order to match it with a linear layer to find the predicted probability of each unique word in our dataset (corpus).



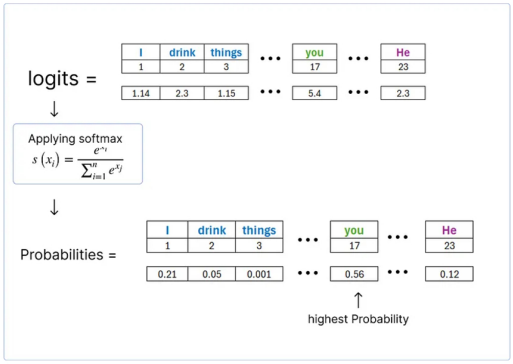
flattened the last add and norm block matrix

This flattened layer will be passed through a linear layer to compute the **logits** (scores) of each unique word in our dataset.



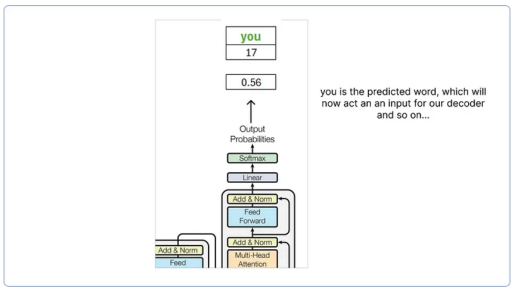
Calculating Logits

Once we obtain the logits, we can use the **softmax** function to normalize them and find the word that contains the highest probability.



Finding the Predicted word

So based on our calculations, the predicted word from the decoder is `you`.



Final output of decoder

This predicted word `you`, will be treated as the input word for the decoder, and this process continues until the `<end>` token is predicted.

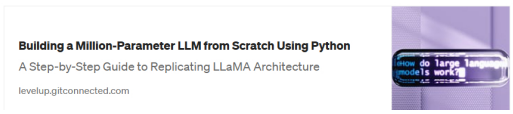
Important Points

1. The above example is very simple, as it does not involve epochs or any other important parameters that can only be visualized using a programming language like Python.
2. It has shown the process only until training, while evaluation or testing cannot be visually seen using this matrix approach.
3. Masked multi-head attentions can be used to prevent the transformer from looking at the future, helping to avoid overfitting your model.

Conclusion

In this blog, I have shown you a very basic way of how transformers mathematically work using matrix approaches. We have applied positional encoding, softmax, feedforward network, and most importantly, multi-head attention.

In the future, I will be posting more blogs on transformers and LLM as my core focus is on NLP. More importantly, if you want to build your own million-parameter LLM from scratch using Python, I have written a blog on it which has received a lot of appreciation on Medium. You can read it here:



Have a great time reading!

Data Science | Artificial Intelligence | Machine Learning | Python | Deep Learning

2.8K 38

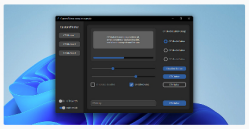
Written by Fareed Khan

Follow

31K Followers · Writer for Level Up Coding

AI Engineer, I write on AI https://www.linkedin.com/in/fareed-khan-dev/

More from Fareed Khan and Level Up Coding



Fareed Khan

Modern GUI using Tkinter

There are two things to remember:

Sep 4, 2022 1K 8

Bret Cameron in Level Up Coding

Automate Your Release Notes with AI

How to save time every week using GitLab, OpenAI, and Node.js

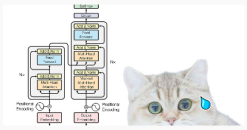
1d ago 15

Jennifer Fu in Level Up Coding

Ace Your Microservices Interviews

Essential concepts and insights of microservices

1d ago 359 4



Fareed Khan

Understanding Transformers: A Step-by-Step Math Example—Pa...

I understand that the transformer architecture may seem scary, and you might...

Jun 5, 2023 2.8K 60

See all from Fareed Khan | See all from Level Up Coding

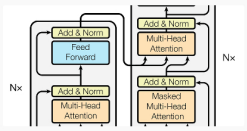
Recommended from Medium

Abdur Rahman in Stackademic

Python is No More The King of Data Science

5 Reasons Why Python is Losing Its Crown

Oct 23 7K 29



Code Thulo

Top Large Language Model (LLM) Interview Question | Basic LLM...

Large Language Models (LLMs) are deep learning models that are trained on huge...

Oct 17 5

Lists

Predictive Modeling w/ Python

20 stories · 1676 saves

Practical Guides to Machine Learning

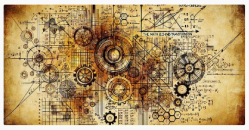
10 stories · 2031 saves

Natural Language Processing

1815 stories · 1428 saves

ChatGPT

21 stories · 879 saves

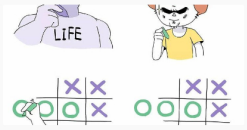


Cristian Leo

The Math Behind Transformers

Deep Dive into the Transformer Architecture, the key element of LLMs. Let's explore its...

Jul 25 884 6

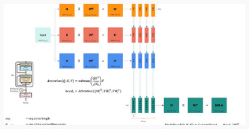


Amit Yadav

Computer Vision Project Ideas With Code

Not a Medium member? Read the full story by clicking here.

Aug 7 114



Vipra Singh

LLM Architectures Explained: Coding a Transformer (Part 7)

Deep Dive into the architecture & building real-world applications leveraging NLP...

Nov 10 285 3

Fareed Khan in Level Up Coding

Building a Million-Parameter LLM from Scratch Using Python

A Step-by-Step Guide to Replicating LLaMA Architecture

Dec 7, 2023 2.7K 33

See more recommendations