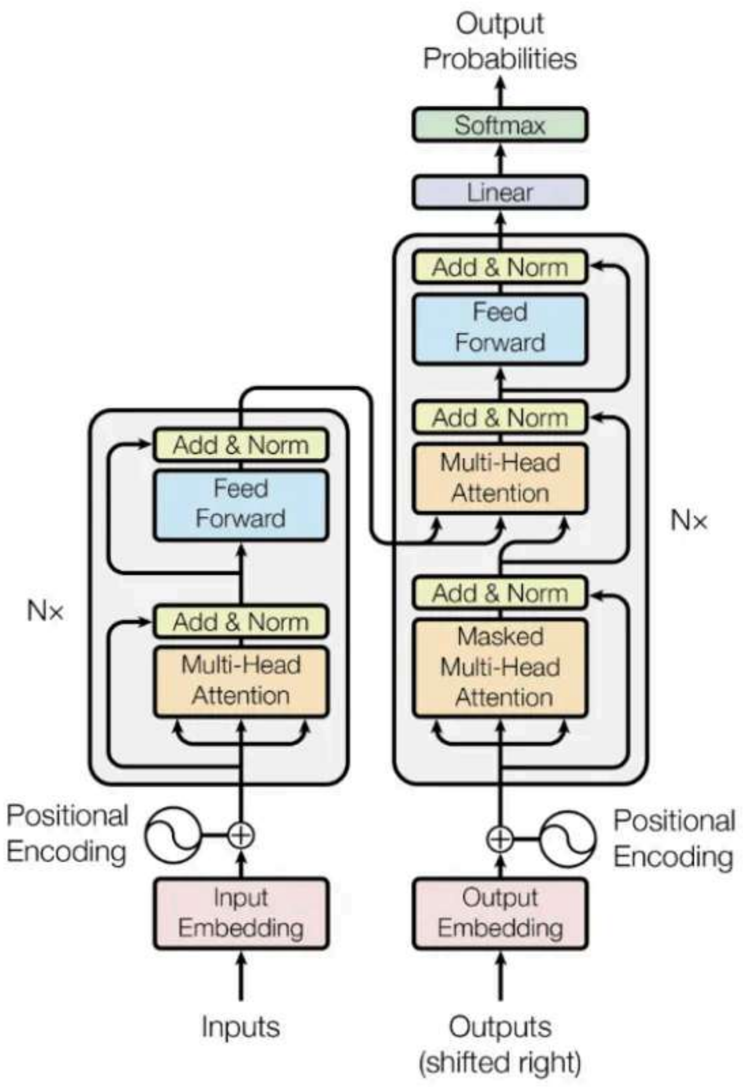
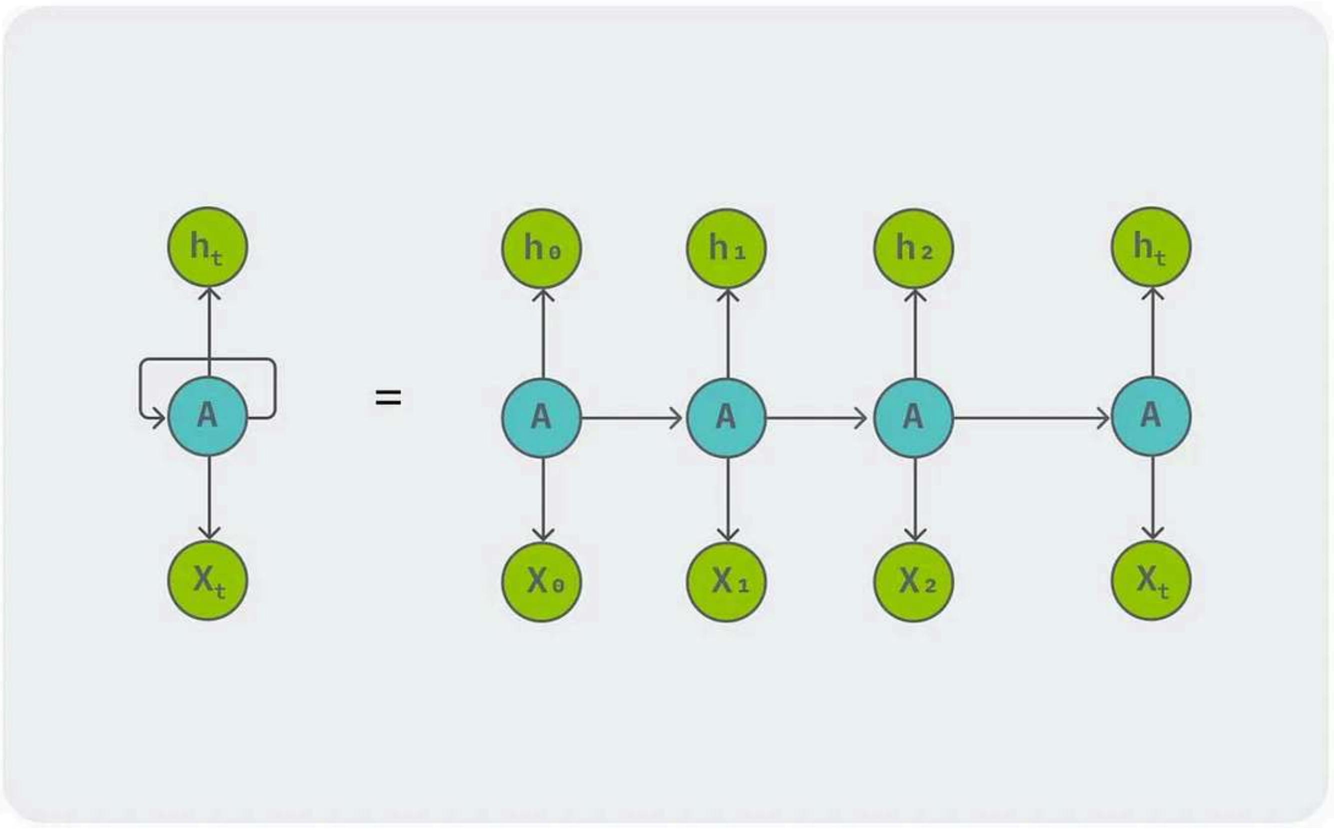
Mastering Transformer: Detailed Insights into Each Block

Ebad Sayed · Follow
10 min read · Jun 9, 2024



https://cdn.analyticsvidhya.com/wp-content/uploads/2019/06/Screenshot-from-2019-06-17-19-53-10.png

Before Transformers, Recurrent Neural Networks (RNNs) were used to handle sequence-to-sequence tasks. RNNs process sequences by maintaining a hidden state, allowing them to map an input sequence to an output sequence over several time steps. At each time step, an input token and the hidden state from the previous step are fed into the RNN to produce an output.



https://miro.medium.com/v2/resize:fit:1400/1*xs2EgGPGIpWrSW4zUANYXA.png

We split the sequence into single items (**X_0**) and pass it to RNN along with the initial state usually made up of only zeros and the model produces an output (**h_0**). This happens at the first time step. In the second time step we take the hidden input from the first and the next token **X_1** to get **h_1**. so if we have **n** tokens then we need **n** time steps to map a sequence input to sequence output. This works fine for a lot of tasks but has some problems.
While effective for many tasks, RNNs had limitations, particularly with capturing long-range dependencies due to vanishing and exploding gradients. Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs) improved on this but still struggled with efficiency and scalability, especially for long sequences.

Problems with RNNs

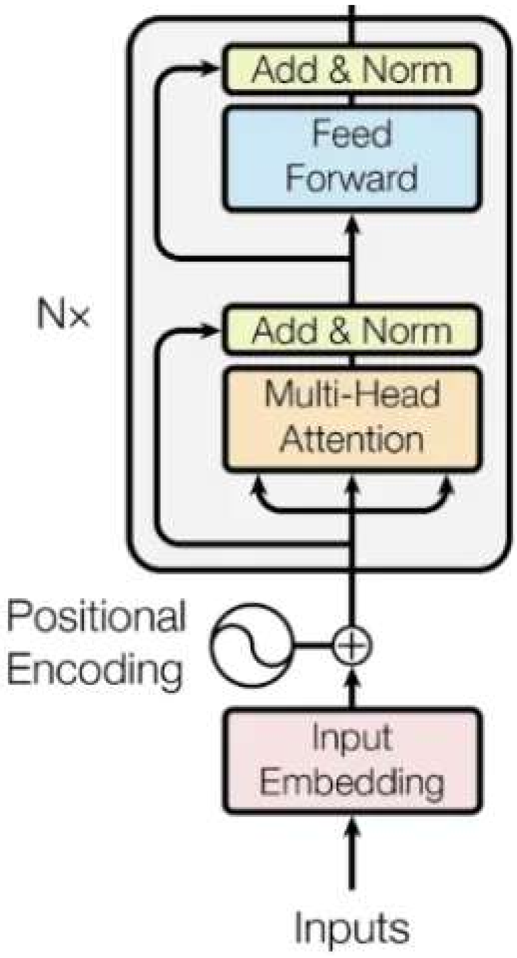
1. **Slow Computation for Long Sequences:** Processing each token sequentially involves a for-loop that runs for every token, making it a time-consuming process.

2. **Vanishing or Exploding Gradients:** During training, PyTorch calculates the derivative of the loss function with respect to the weights using the chain rule. As the number of hidden layers increases, the chain becomes longer, leading to derivatives that can either be extremely large or extremely small. This is problematic because CPUs and GPUs have limited precision for representing numbers. Consequently, the gradients can either vanish or explode, causing very small or very large updates to the model parameters, which is undesirable.

3. **Difficulty in Accessing Information from Earlier in the Sequence:** Due to the long chain of dependencies, the influence of the hidden state from the initial tokens diminishes as the sequence progresses. This means the effect of the first token on the last token becomes negligible, making it hard for the model to retain long-range dependencies.

Transformer solves all the above problems. The structure of the transformer can be divided into two macro blocks, the left part is called **encoder** and the right one is called **decoder**.
Paper :- Attention is all you need

Encoder



We will look at each blocks on the encoder.

Input Embeddings

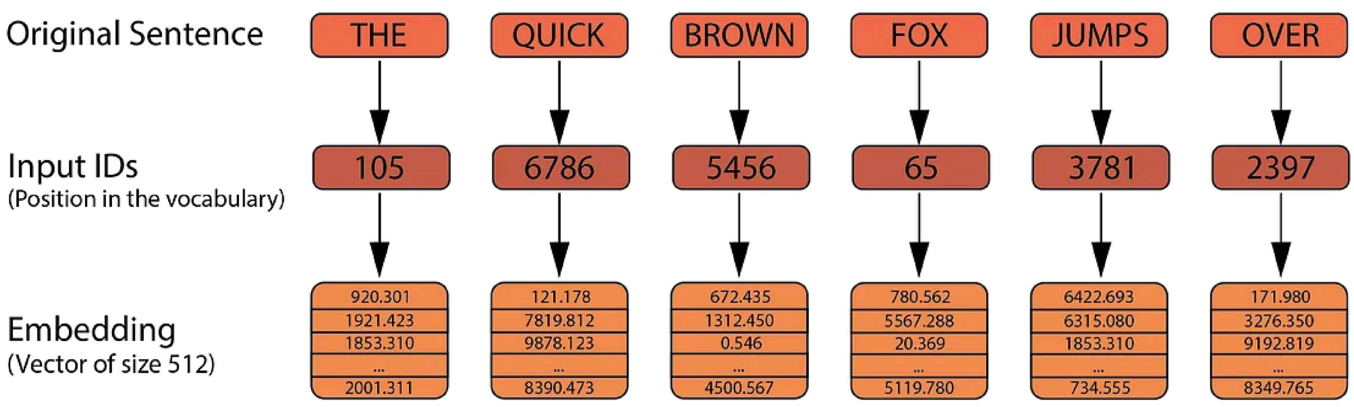


Image by Author

The encoder begins with the input embeddings. Given a sentence of six words, “The quick brown fox jumps over” we first transform the sentence into tokens. These tokens are then mapped to numbers that represent their positions in the vocabulary, known as **Input IDs**. These Input IDs are converted into vectors of size **512**, referred to as embeddings. These embeddings are **not fixed**; they are parameters that the model learns and adjusts to capture the meaning of each word.
Note that while the Input IDs remain constant, the embeddings will change throughout the training process.

Positional Encoding

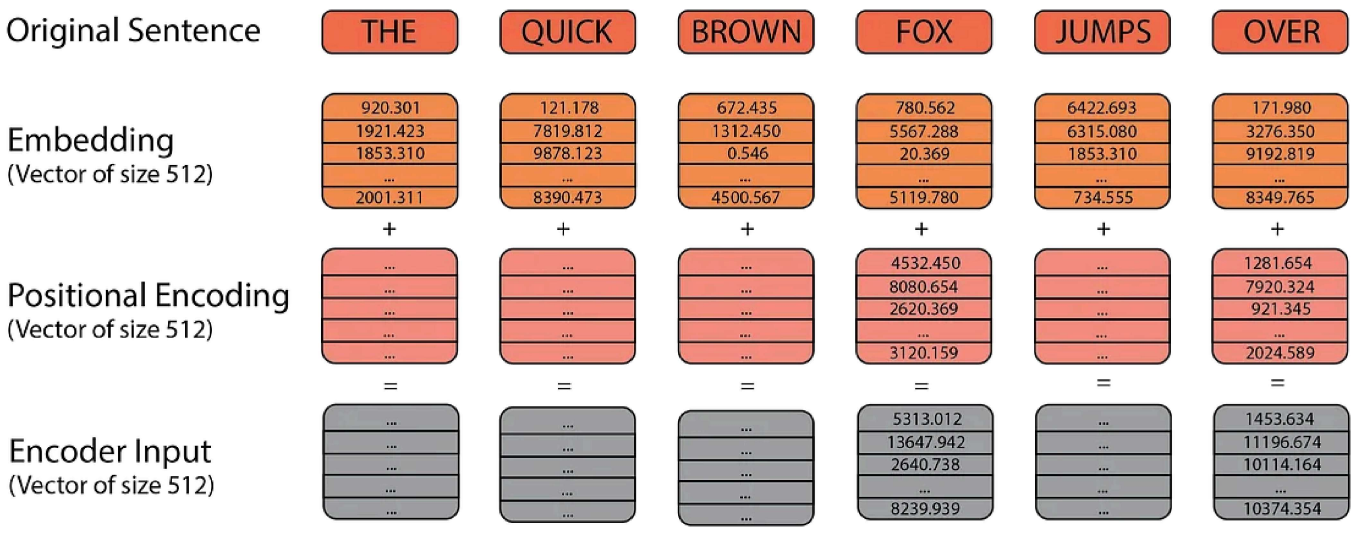


Image by Author

Next, we have the positional encoding. While our word embeddings capture the meaning of the words, they don’t convey any information about the position of each word in the sentence. We want the model to understand that words appearing close together in a sentence are related, and those far apart are less so. Positional encoding provides **spatial information** about each word's position, helping the model recognize patterns in the sentence structure.

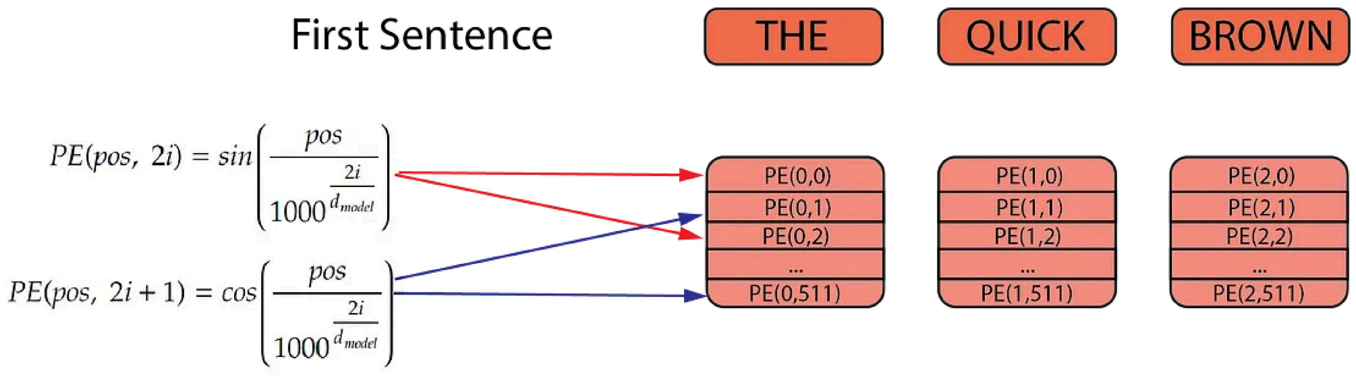
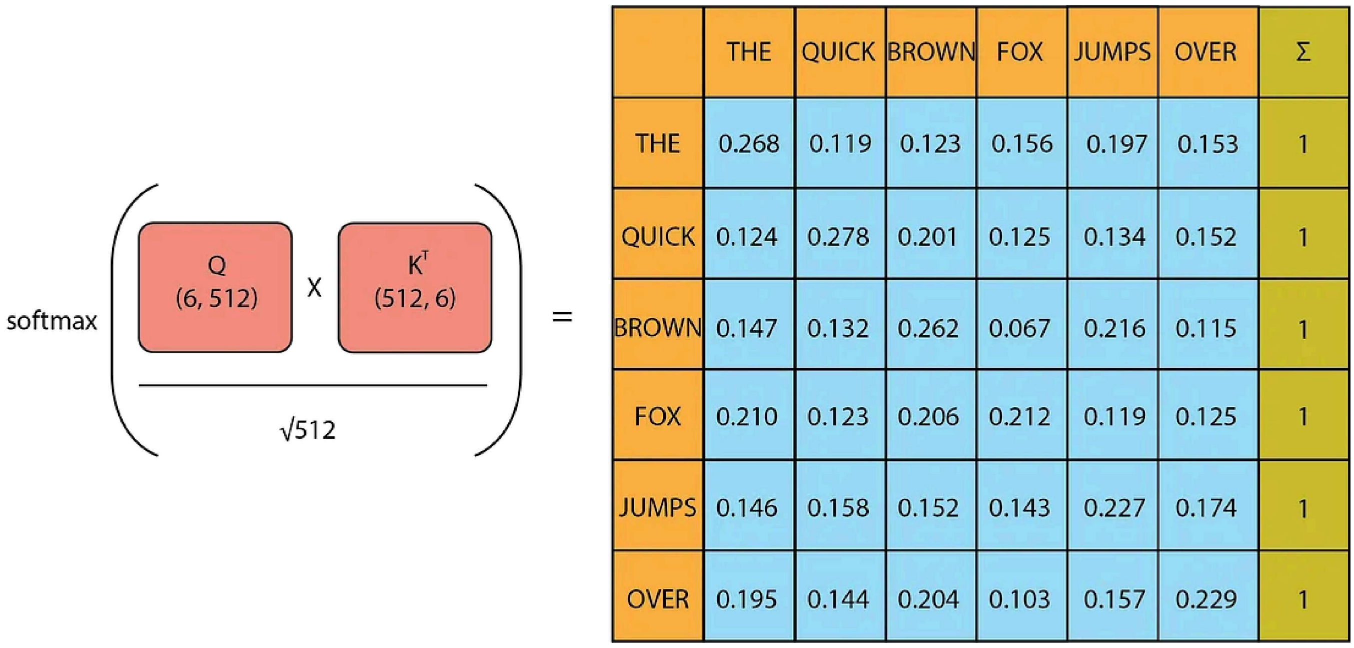

First Sentence
THE
QUICK
BROWN
$PE(pos, 2i) = sin\left(\frac{pos}{1000^{\frac{2i}{d_{model}}}}\right)$
$PE(pos, 2i+1) = cos\left(\frac{pos}{1000^{\frac{2i}{d_{model}}}}\right)$
PE(0,0)
PE(0,1)
PE(0,2)
...
PE(0,511)
PE(1,0)
PE(1,1)
PE(1,2)
...
PE(1,511)
PE(2,0)
PE(2,1)
PE(2,2)
...
PE(2,511)

Image by Author

To calculate the positional encodings, we use two formulas as described in the original Transformer paper. The first formula is applied to even positions and the second to odd positions. These positional encodings are computed once and then reused during both training and inference for every sentence.

Multi-Head Attention

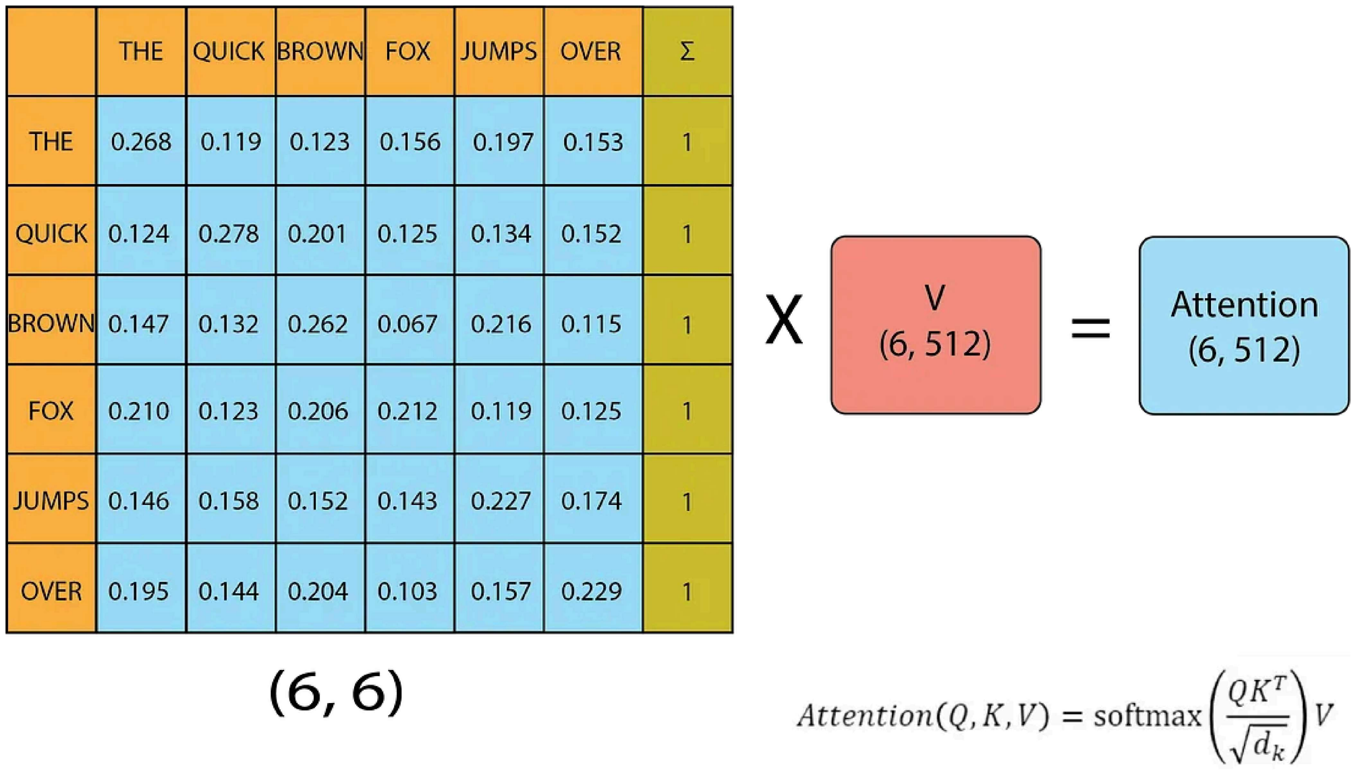
First, let's understand what **self-attention** is. This mechanism existed before the Transformer, but the creators of the Transformer adapted it into multi-head attention. Self-attention allows the model to relate words to one another. The input embeddings capture the meanings of the words, while the positional encoding provides information about the position of each word in the sentence. The input matrix (6,512) is used **three times** here, for **Q(query), K(key)** and **V(values)**. Using the formula mentioned in the paper, we will multiply **Q (6,512)** with **K^T (512,6)** then divide it with **$\sqrt{d_k}$** and then apply **softmax**.



	THE	QUICK	BROWN	FOX	JUMPS	OVER	Σ
THE	0.268	0.119	0.123	0.156	0.197	0.153	1
QUICK	0.124	0.278	0.201	0.125	0.134	0.152	1
BROWN	0.147	0.132	0.262	0.067	0.216	0.115	1
FOX	0.210	0.123	0.206	0.212	0.119	0.125	1
JUMPS	0.146	0.158	0.152	0.143	0.227	0.174	1
OVER	0.195	0.144	0.204	0.103	0.157	0.229	1

Image by Author

These values are a **score**, representing how intense the relationship between one word and another is.



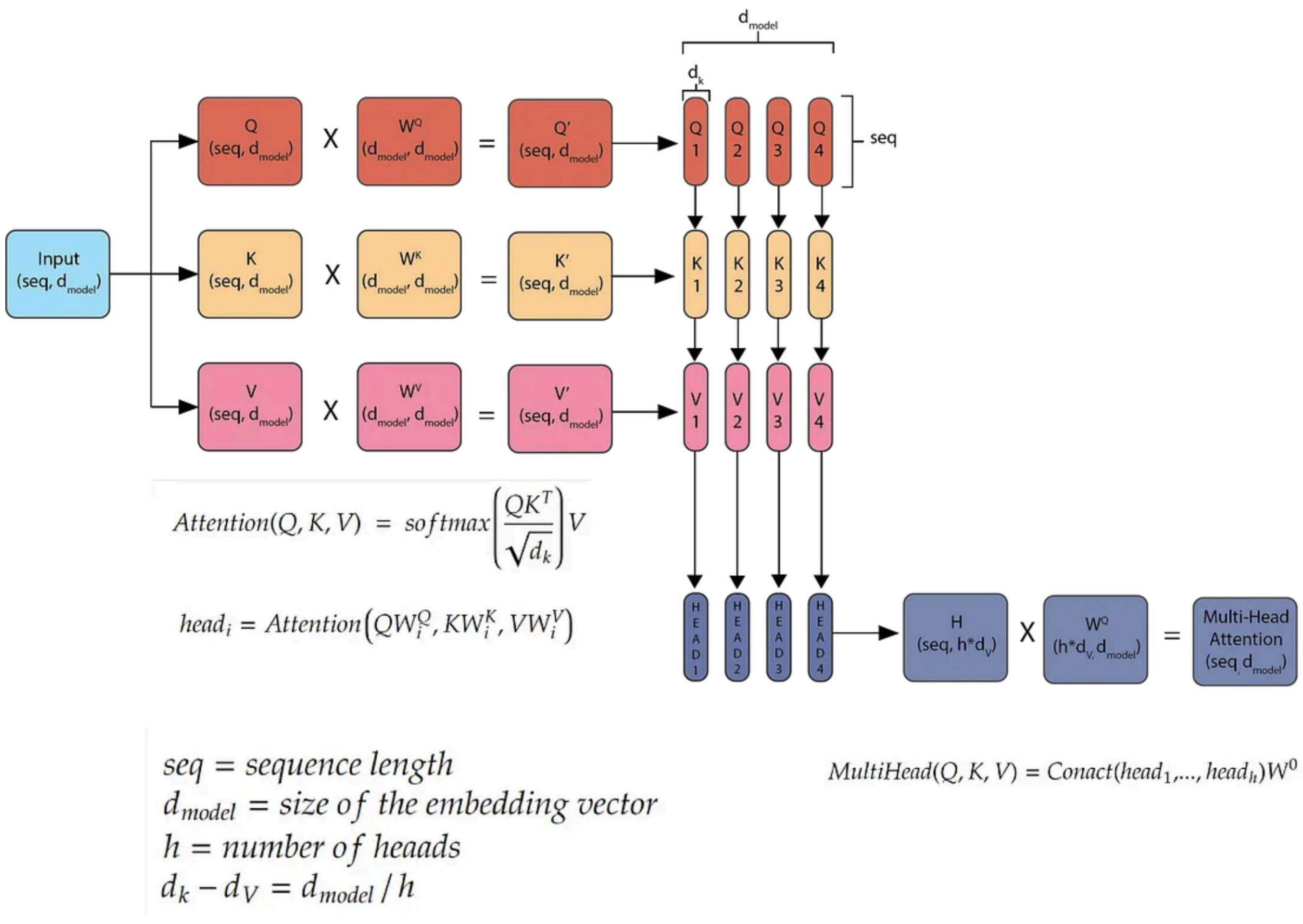
	THE	QUICK	BROWN	FOX	JUMPS	OVER	Σ
THE	0.268	0.119	0.123	0.156	0.197	0.153	1
QUICK	0.124	0.278	0.201	0.125	0.134	0.152	1
BROWN	0.147	0.132	0.262	0.067	0.216	0.115	1
FOX	0.210	0.123	0.206	0.212	0.119	0.125	1
JUMPS	0.146	0.158	0.152	0.143	0.227	0.174	1
OVER	0.195	0.144	0.204	0.103	0.157	0.229	1

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

Image by Author

Finally we multiply this matrix with **V** to get the attention matrix. The dimension of the attention matrix is the same as the dimension of the initial matrix. After this the matrix embedding represents not only meaning and position of the word but also its relation with other words.

- Self-attention is permutation invariant
- We expect values along the diagonal to be the highest
- If we don't want some positions to interact, we can always set their values to -∞ before applying the softmax in this matrix and the model will not learn those interactions.



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

$$head_i = Attention\left(QW_i^Q, KW_i^K, VW_i^V\right)$$

$$MultiHead(Q,K,V) = Conact(head_1,\ldots,head_h)W^0$$

$$\begin{aligned} seq &= sequence\ length \\ d_{model} &= size\ of\ the\ embedding\ vector \\ h &= number\ of\ heaads \\ d_k - d_V &= d_{model}/h \end{aligned}$$

Image by Author

In multi-head attention, the input embeddings is duplicated to make three copies of it **(Q, K, V)** with dimension **(seq, d_model)** and each multiplied by parameter matrices **(W^Q, W^K, W^V)** with dimension **(d_model, d_model)**. We get the resultant matrices (Q', K', V') with dimension **(seq, d_model)**. Next we will split these matrices by the d_model dimension. **d_k = d_model/h** where h is the number of heads here equal to **4**. We can then find the attention using the formula given in the paper. And we get small matrices head1, head2, head3 and head4 with dimension **(seq, d_v)** here **d_v= d_k**. Then we will concatenate these matrices along the d_v dimension. Then we multiply this new matrix with weight matrix **(h*d_v, d_model)** to get the final output of multi-head attention **(seq, d_model)**.

So instead of calculating the attentions on **(Q', K', V')** we split them into multiple heads and calculate the attentions between these smaller matrices. Each head is watching a different aspect of the same word. For example in context a word can be a noun, a verb or an adverb. So head1 learns to relate the word as a noun, head2 learns to relate it as a verb and so on.

ADD & Norm (Layer Normalization)

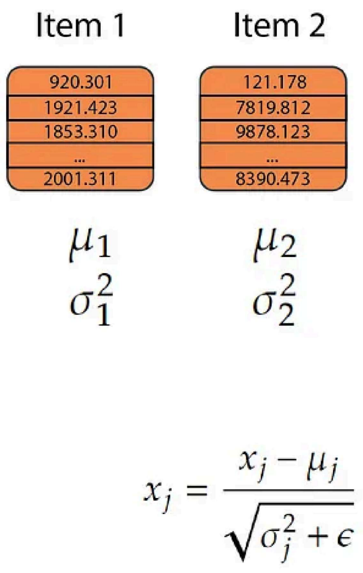
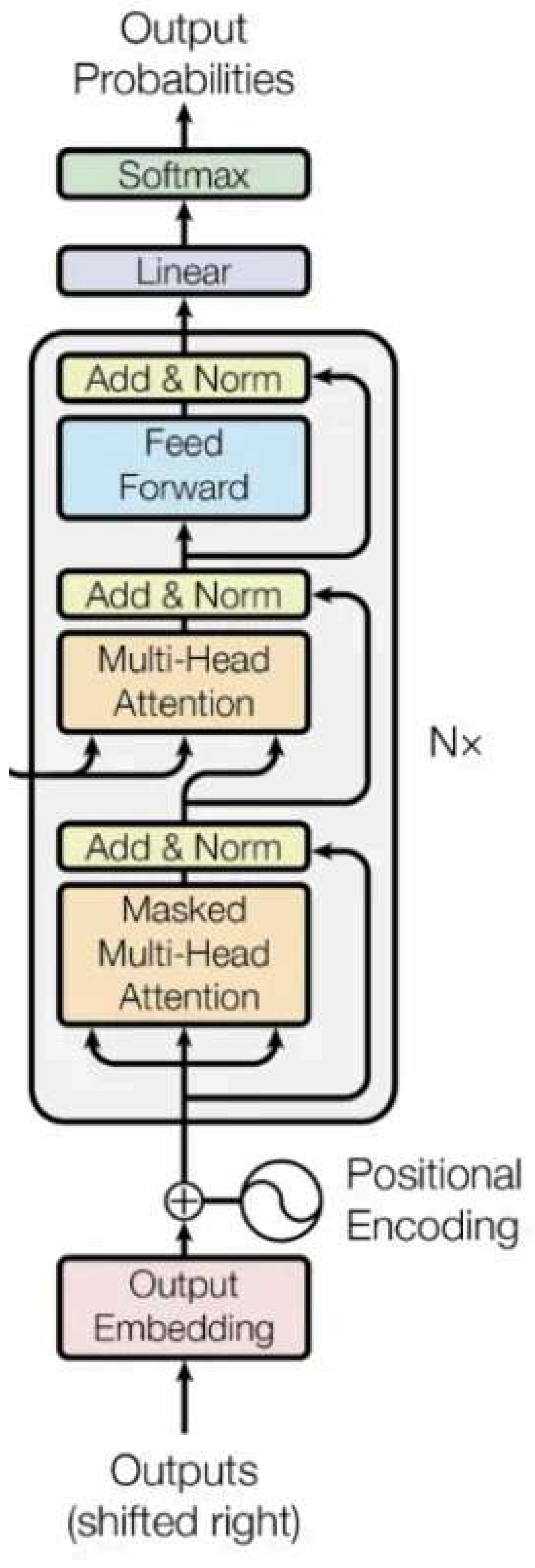


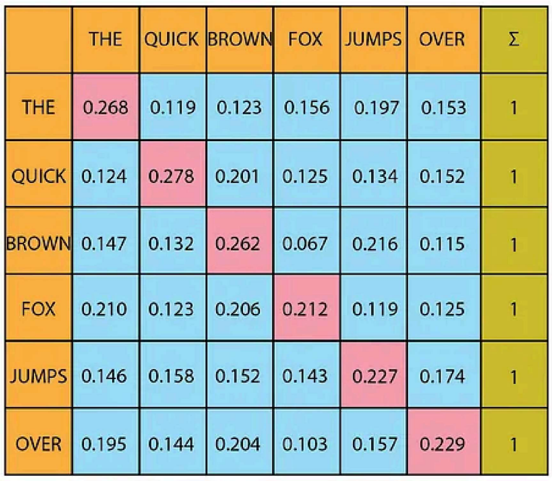
Image by Author

Suppose we have a batch of **n** items and each of them will have some features it could be an embedding. We will calculate the **mean** and **variance** of each of these items independently from each other and we replace each value with another value given by the expression. We multiply this with parameters $\boldsymbol{\alpha}$ or γ**(multiplicative)** and β**(additive)**. We also add ϵ for numerical stability so that the denominator value doesn't approach zero.

Decoder



Masked Multi-Head Attention



	THE	QUICK	BROWN	FOX	JUMPS	OVER	Σ
THE	0.268	0.119	0.123	0.156	0.197	0.153	1
QUICK	0.124	0.278	0.201	0.125	0.134	0.152	1
BROWN	0.147	0.132	0.262	0.067	0.216	0.115	1
FOX	0.210	0.123	0.206	0.212	0.119	0.125	1
JUMPS	0.146	0.158	0.152	0.143	0.227	0.174	1
OVER	0.195	0.144	0.204	0.103	0.157	0.229	1

Image by Author

To ensure the model is causal, meaning that the output at a given position depends only on the words in the preceding positions, we need to prevent it from seeing future words. In multi-head attention, we achieve this by masking. Specifically, we replace all values above the main diagonal with negative infinity before applying the softmax function.
Output of the encoder are **keys** and **values** which go inside the decoder's multi-head attention. This is **cross self-attention** as two parameters are the output of the encoder and the **query** matrix comes from the decoder's input after processing through masked multi-head attention.

Feed Forward

Linear Transformation: The input to the feedforward block is a tensor with shape **(batch_size, seq, d_model).** This tensor is passed through a fully connected linear layer, which transforms it to another tensor with shape **(batch_size, seq, d_ff)**, where d_ff is the dimensionality of the feedforward layer, typically larger than d_model.
ReLU Activation: The output of the first linear layer is then passed through a rectified linear unit (ReLU) activation function element-wise. This introduces non-linearity into the model.
Second Linear Transformation: The result of the ReLU activation is passed through another linear layer, which projects the tensor back to the original dimensionality, **(batch_size, seq, d_model)**.
The feedforward block allows the model to learn complex, non-linear transformations of the input. It is applied independently to each position in the sequence, making it highly parallelizable and efficient for processing sequences.

Training

Suppose we are performing a language translation from English to French. We begin by taking the English sentence and sending it to the encoder. We add two special tokens: one at the beginning to mark the start of the sentence and another at the end to mark its conclusion. These tokens, taken from the vocabulary, indicate the boundaries of the sentence to the model.
Next, we convert the sentence into input embeddings, add positional encodings, and pass it through the encoder. The encoder produces an output matrix with the shape (seq, d_model). This matrix contains embeddings that capture the meaning of each word, its position, and its relationship with every other word in the sentence.

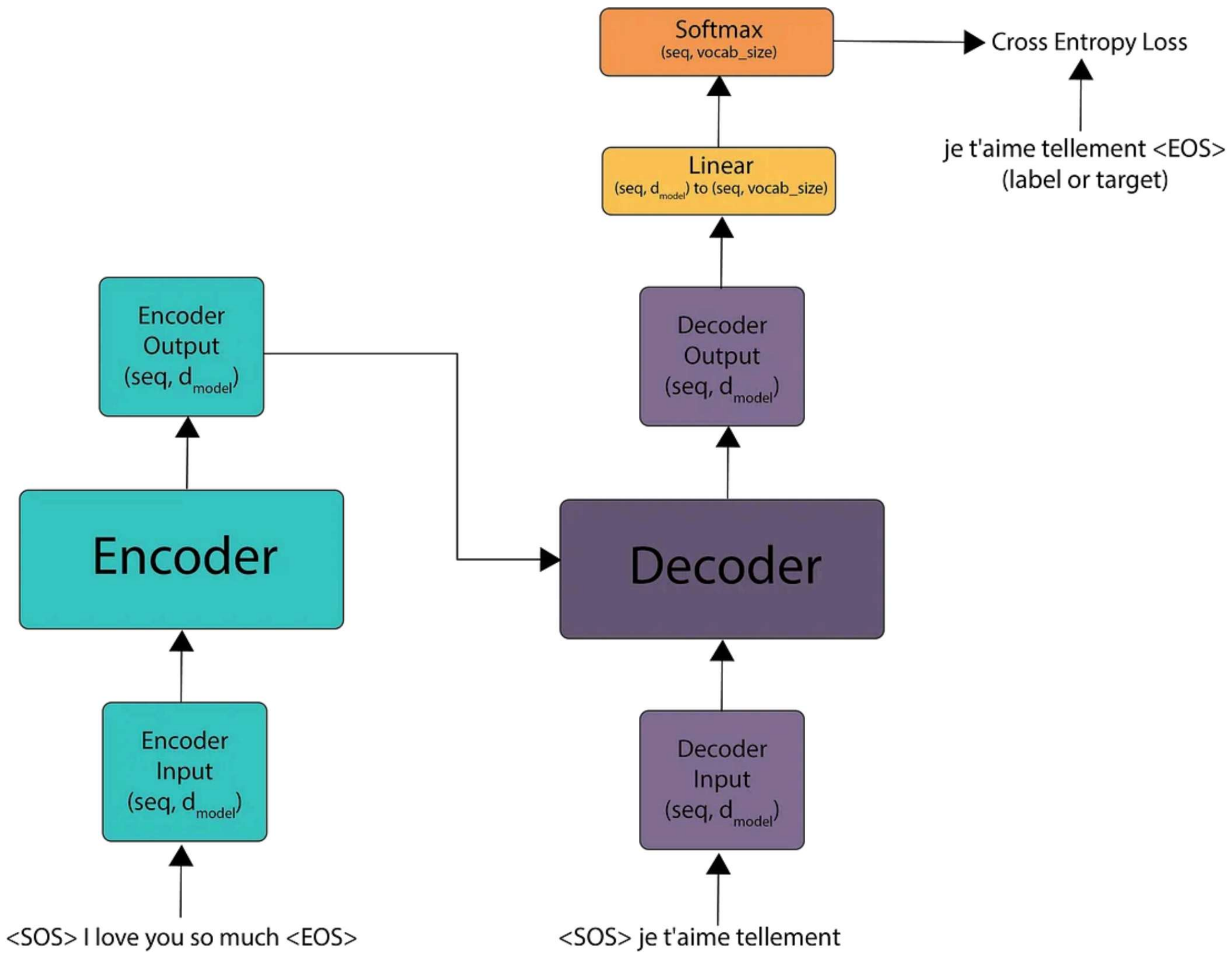

Softmax
(seq, vocab_size)
Cross Entropy Loss
Linear
(seq, d_{model}) to (seq, vocab_size)
je t'aime tellement <EOS>
(label or target)
Encoder Output (seq, d_{model})
Decoder Output (seq, d_{model})
Encoder
Decoder
Encoder Input (seq, d_{model})
Decoder Input (seq, d_{model})
<SOS> I love you so much <EOS>
<SOS> je t'aime tellement

Image by Author

To convert an English sentence ("I love you so much") into French ("je t'aime tellement") using the Transformer model, we start by passing the start token (SOS) to the decoder. The output (shifted right) is then processed by converting it into embeddings, adding positional encoding, and feeding it into the decoder's masked multi-head attention. Next, we take the output from the encoder as the query and keys, and the output from the masked multi-head attention as the values. These three components are then passed as input to the decoder's multi-head attention.

The decoder produces an output matrix of shape (seq, d_model). To map this output to the vocabulary, we use a linear layer that transforms the matrix from (seq, d_model) to (seq, vocab_size). This linear layer helps determine the position of each word in the vocabulary, allowing us to understand the actual token output by the model. Finally, we apply a softmax function to generate probabilities for each token, producing the predicted French sentence.

The model's output is compared to the target French sentence to calculate the loss. This loss is then used to update the model's weights through backpropagation. This entire process occurs in a single time step.

Inference

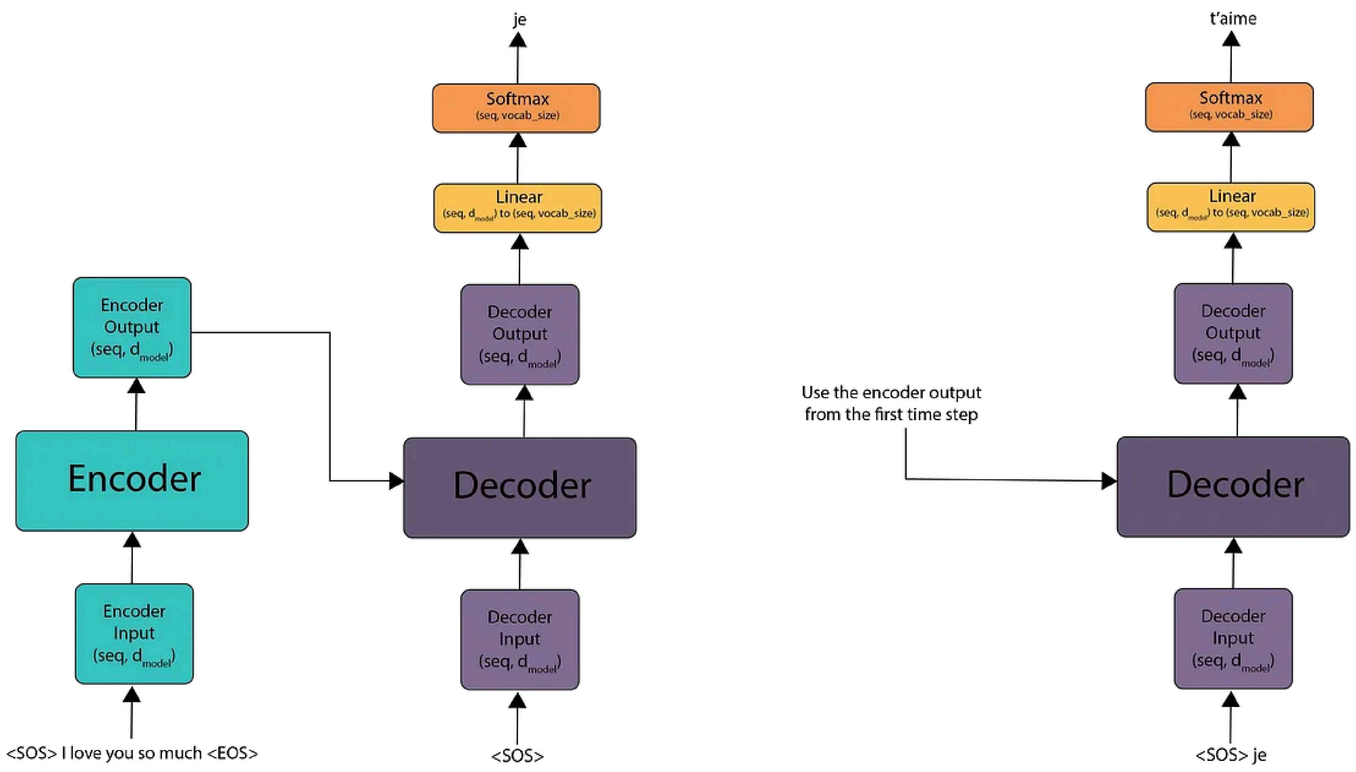

je
Softmax (seq, vocab_size)
Linear (seq, d_model) to (seq, vocab_size)
Encoder Output (seq, d_model)
Decoder Output (seq, d_model)
Encoder
Decoder
Encoder Input (seq, d_model)
Decoder Input (seq, d_model)
<SOS> I love you so much <EOS>
<SOS>
t'aime
Softmax (seq, vocab_size)
Linear (seq, d_model) to (seq, vocab_size)
Decoder Output (seq, d_model)
Use the encoder output from the first time step
Decoder
Decoder Input (seq, d_model)
<SOS> je

Image by Author

We start by passing the **start-of-sentence (SOS)** and **end-of-sentence (EOS)** tokens through the encoder, which produces an output matrix. For the decoder, we initially pass only the SOS token. The decoder's output is then fed into a linear layer, producing logits. After applying the softmax function, we select the token from the vocabulary that corresponds to the highest value. This gives us the first token of the translated sentence, all occurring at time step 1.
At time step 2, we do not need to recompute the encoder's output because the

English sentence remains unchanged. Instead, we take the first French word predicted in the previous time step, append it to the decoder's input, and feed it back into the decoder. This process is repeated to generate the second token. We continue this process until we encounter the EOS token.

In the decoding process, we traditionally selected the word with the highest softmax probability at each step. However, this **greedy** approach often does not yield optimal results.
A more effective strategy is **Beam Search**, where, at each decoding step, we consider the top B words based on their probabilities. We then evaluate all possible next words for each of these B words, keeping track of the top B most probable sequences. This strategy allows us to explore multiple potential sequences simultaneously, often leading to better overall performance in generating the output sequence.

Summary

The Transformer model has transformed natural language processing, especially in machine translation, with its self-attention mechanisms. It efficiently captures dependencies in input sequences, enabling accurate translations. In decoding, it selects the highest probability word at each step, though Beam Search, considering the top B words, often performs better. The Transformer's ability to handle long-range dependencies and parallel processing makes it a significant advancement in NLP, with future developments promising further enhancements.

To gain a better intuition into the math behind each block :- Understanding Transformer with Math Examples

Next article :- Build the Transformer Architecture from Scratch

Transformer Model Large Language Models Machine Learning

Follow

Written by Ebad Sayed

172 Followers · 9 Following

I am currently a final year undergraduate at IIT Dhanbad, looking to help out aspiring AI/ML enthusiasts with easy AI/ML guides.

No responses yet

What are your thoughts?

Respond

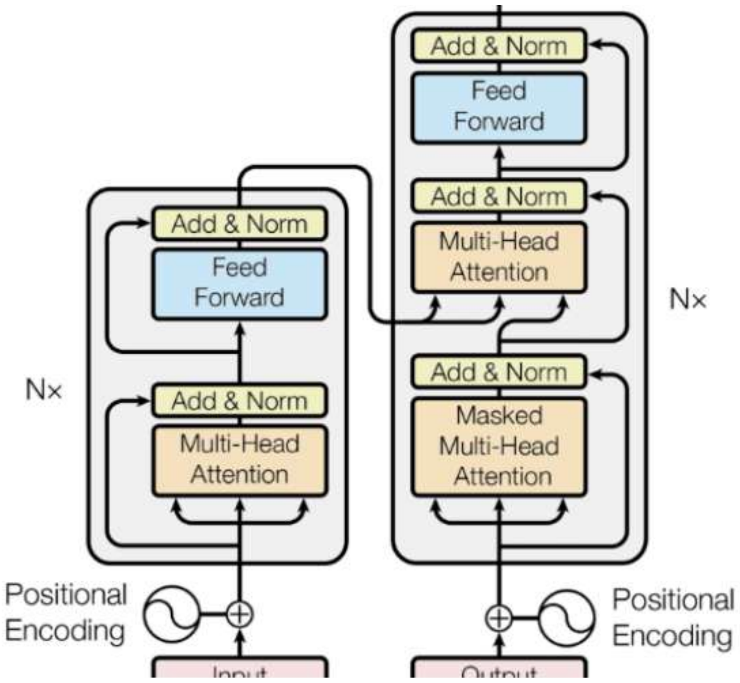
More from Ebad Sayed

Ebad Sayed

Scikit-LLM: Scikit-Learn Meets Large Language Models

As a beginner in Python and ML, I frequently relied on scikit-learn for almost all of my projects. Its simplicity and versatility made...

Sep 10, 2024 473 6

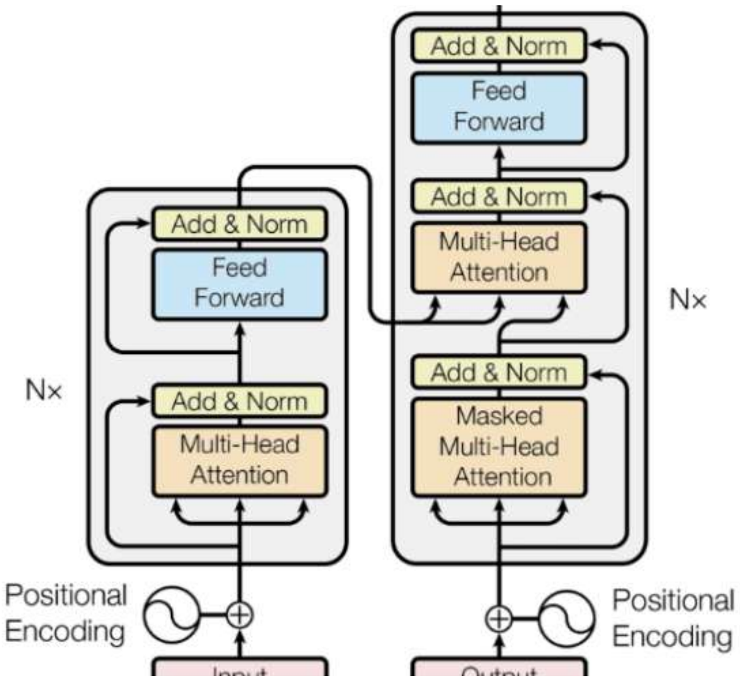


Ebad Sayed

Building a Transformer from Scratch: A Step-by-Step Guide

Introduction

Jun 9, 2024 62

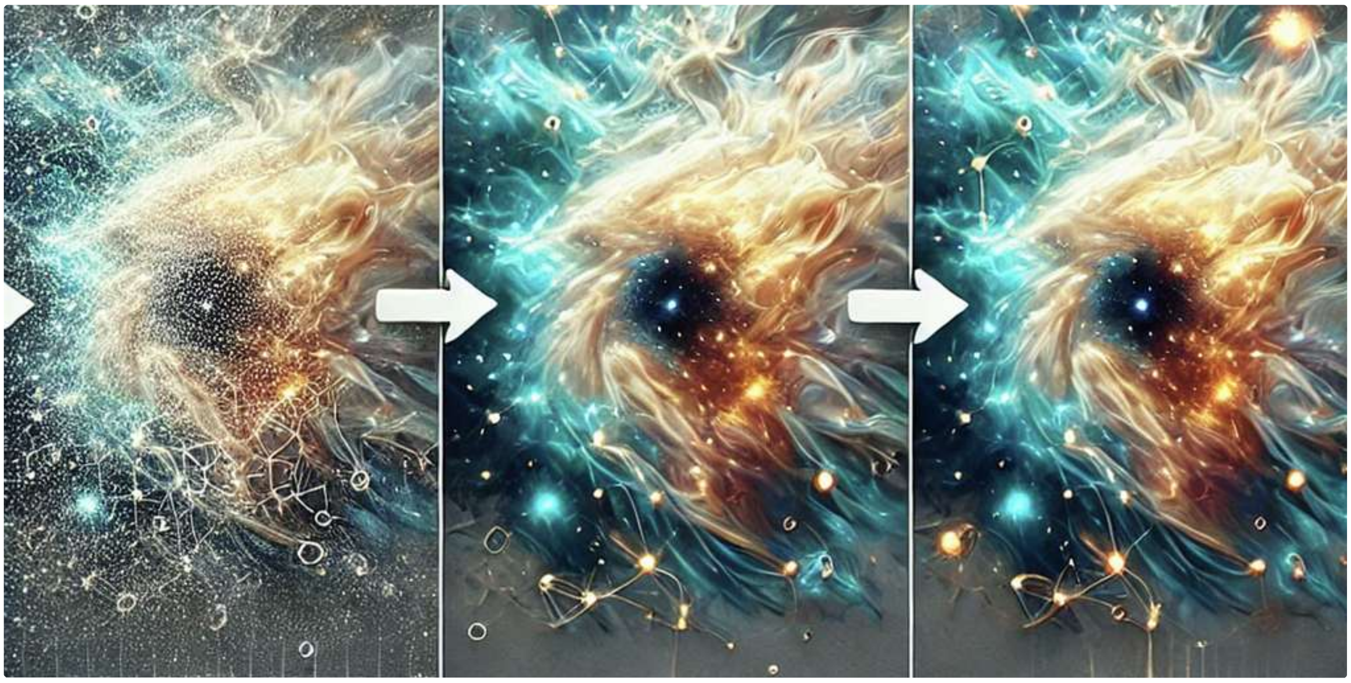


Ebad Sayed

Training a Transformer Model from Scratch

In this article, we will see how to train the Transformer model that we built in the previous article on various tasks such as...

Jun 11, 2024 72



Ebad Sayed

DDPM PyTorch Implementation from Scratch

Previous Article: Mastering Diffusion Probabilistic Models from Scratch

Dec 6, 2024 20 1

See all from Ebad Sayed

Recommended from Medium

In GoPenAI by Ashutosh

Exploring the Transformer Model

Transformer model has revolutionized how machines understand and generate human language. What started as a breakthrough in natural...

Jan 8 1

In Level Up Coding by Fareed Khan

Building a Million-Parameter LLM from Scratch Using Python

A Step-by-Step Guide to Replicating LLaMA Architecture

Dec 7, 2023 2.9K 34

Lists

Predictive Modeling w/ Python

20 stories · 1793 saves

Practical Guides to Machine Learning

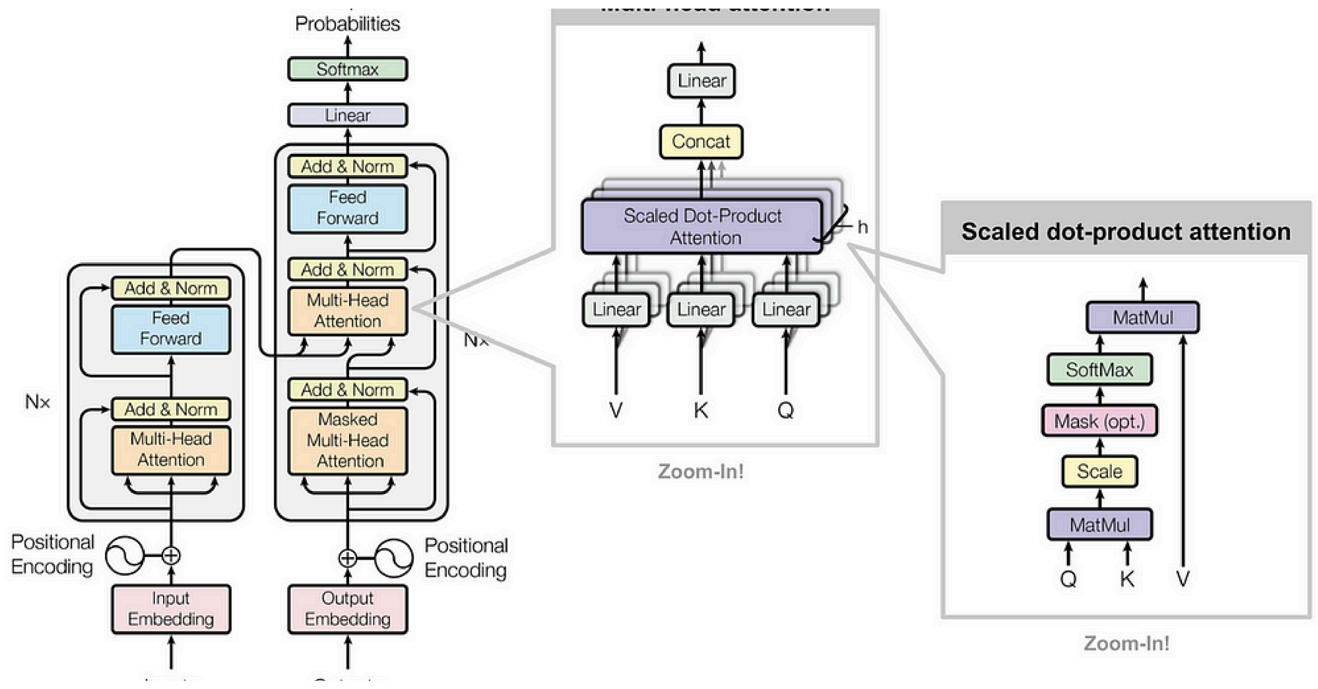
10 stories · 2173 saves

Natural Language Processing

1894 stories · 1555 saves

The New Chatbots: ChatGPT, Bard, and Beyond

12 stories · 545 saves

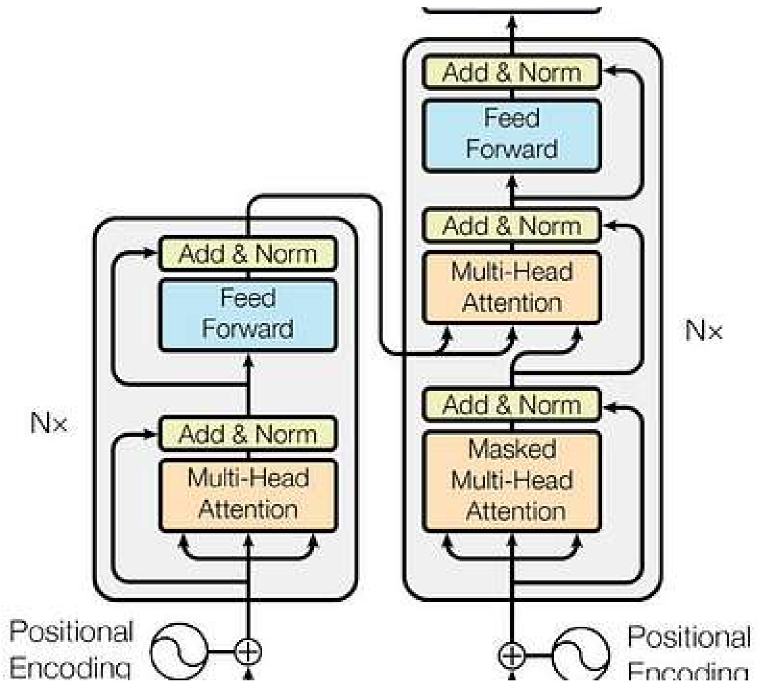


LM Po

Self-Attention and Transformer Network Architecture

The introduction of Transformer models in 2017 marked a significant turning point in the fields of Natural Language Processing (NLP) and...

Oct 18, 2024 53 1

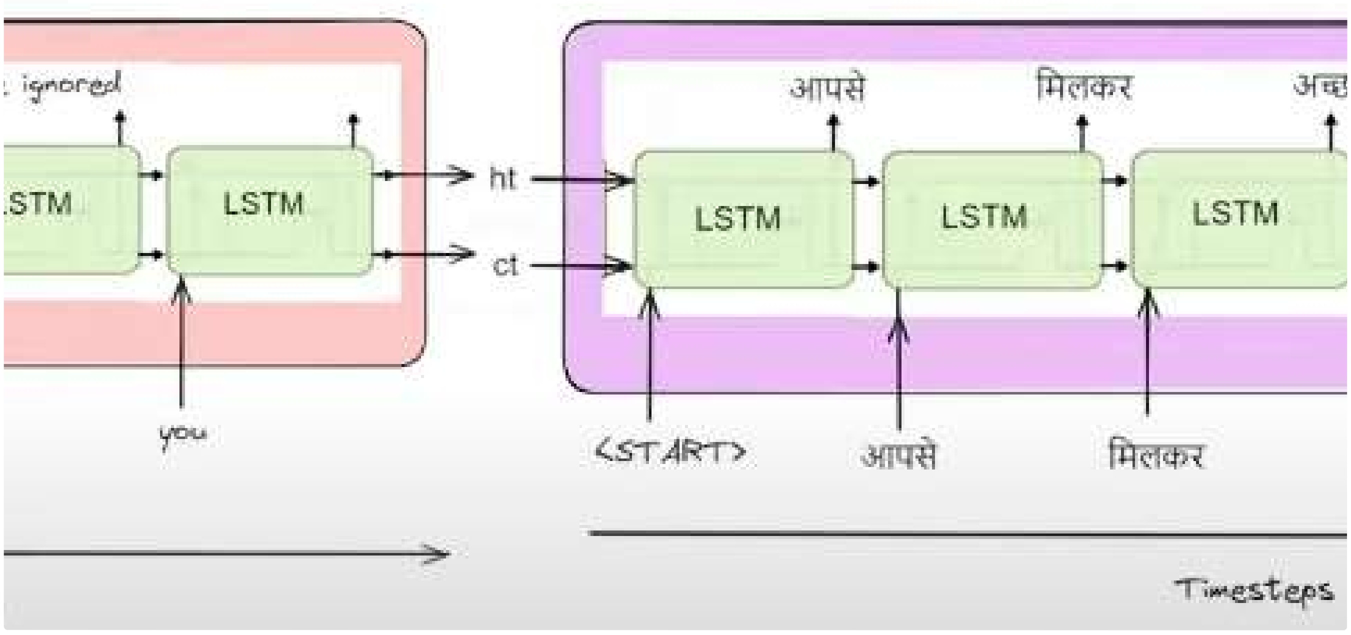


Vipra Singh

LLM Architectures Explained: Transformers (Part 6)

Deep Dive into the architecture & building real-world applications leveraging NLP Models starting from RNN to Transformer.

Nov 10, 2024 477 4

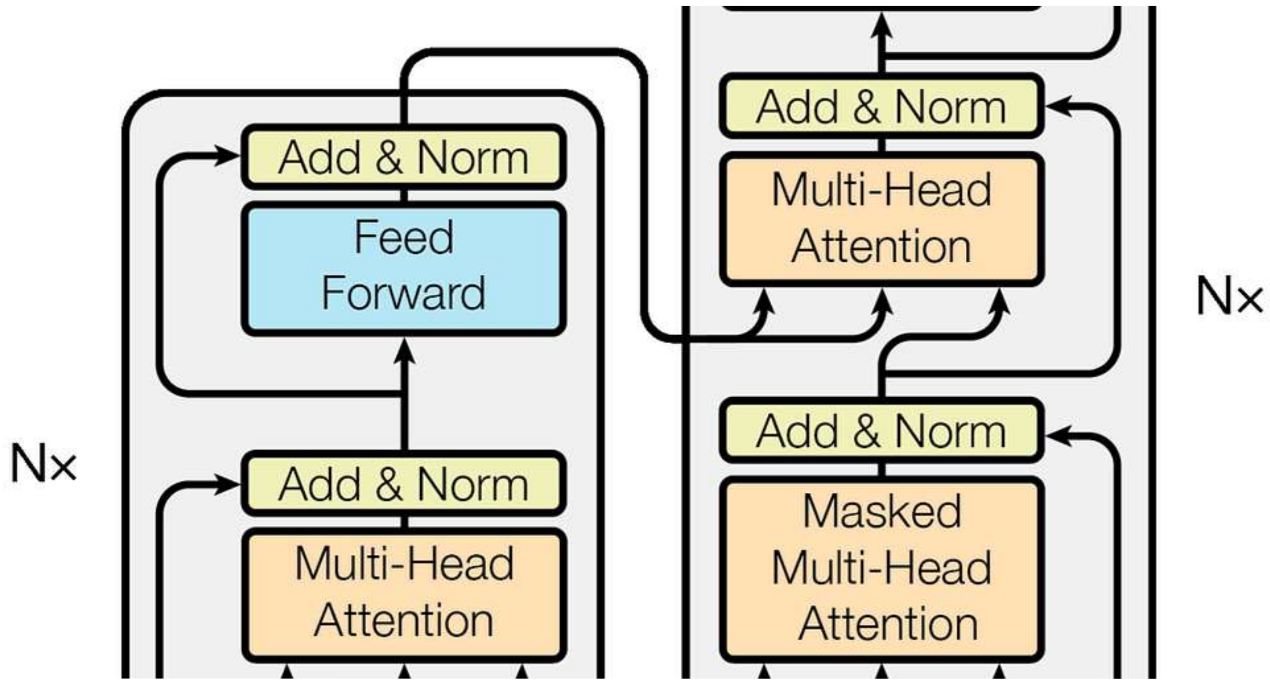


Abhishek Jain

Masked Multi Head Attention in Transformers

The transformer decoder is autoregressive at inference time and non auto regressive at training time

Dec 29, 2024



Code Thulo

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

Large Language Models (LLMs) are deep learning models that are trained on huge amounts of text data. They use a transformer architecture...

Oct 17, 2024 20

See more recommendations