Transformer Architecture Simplified

medium.com/@tech-gumptions/transformer-architecture-simplified-3fb501d461c8

Tech Gumptions October 16, 2023

Tech Gumptions

Transformer architecture was introduced by a team of researchers at Google's Brain division in a 2017 paper titled "**Attention Is All You Need**." This advent of the transformer marked a seismic shift in the landscape of NLP. It ushered in the era of large language models (LLMs) that have since demonstrated remarkable advancements in the realm of NLP, surpassing the capabilities of earlier-generation recurrent neural networks (RNNs). From machine translation and sentiment analysis to question answering and text summarization, LLMs based on transformers have set new benchmarks and opened up exciting possibilities for the future of AI-driven language tasks.

Before we delve into the intricate details of the transformer architecture, let's draw an analogy that simplifies the understanding of how transformers operate.

Imagine you are trying to comprehend a lengthy story. You can't simply read the story from beginning to end, as you will forget what happened earlier in the narrative. Instead, you need to be able to focus on the important parts of the story and ignore the unimportant ones. Transformers function in a similar way. They can concentrate on the crucial elements of a sequence and disregard the less significant ones, even if the important parts are not consecutive.

A transformer accomplishes the following:

1. Pays Attention: Just like you might pay extra attention to important parts of a story, the transformer pays attention to important words in a sentence.

2. Understands Context: The transformer looks at all the words in the sentence together, not one after the other. This helps it understand how words depend on each other.

3. Weighs Relationships: It figures out how words are related to each other. For example, if the sentence is about a cat and a mouse, it knows that these words are connected.

4. Combines Insights: The transformer combines all this knowledge to understand the whole story and how words fit together.

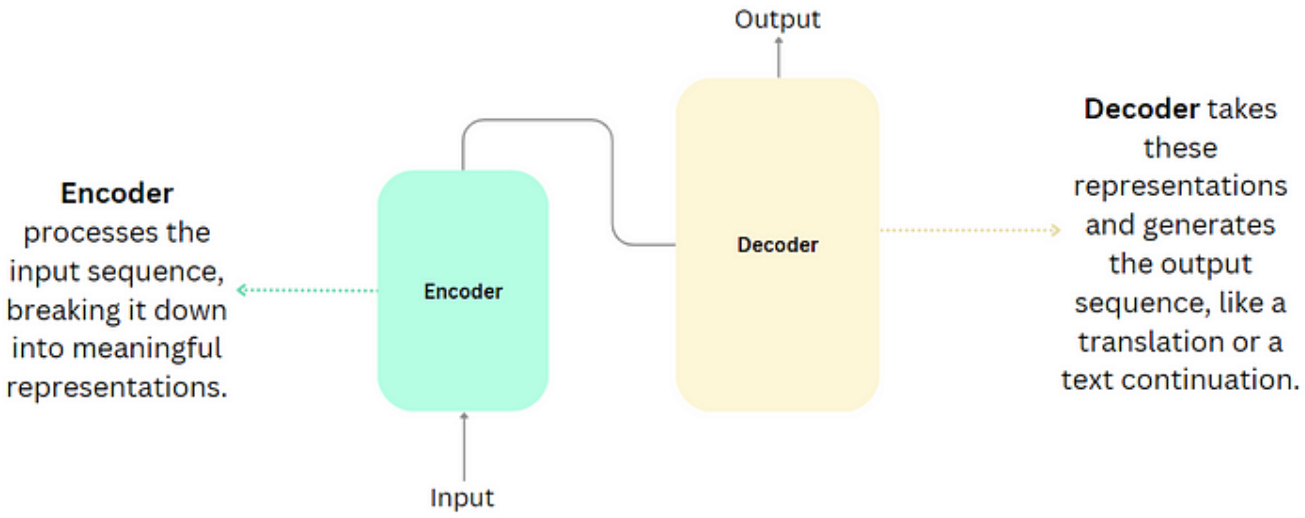
5. Predicts Next Steps: With its understanding, it can even guess what words might come next in the story.

At its core, the transformer architecture revolutionized the way neural networks process sequences by introducing the concept of “self-attention.” This self-attention mechanism lets each word in the sequence consider the entire context of the sentence, rather than just the words that came before it. This is akin to a person paying varying degrees of attention to different parts of a conversation.

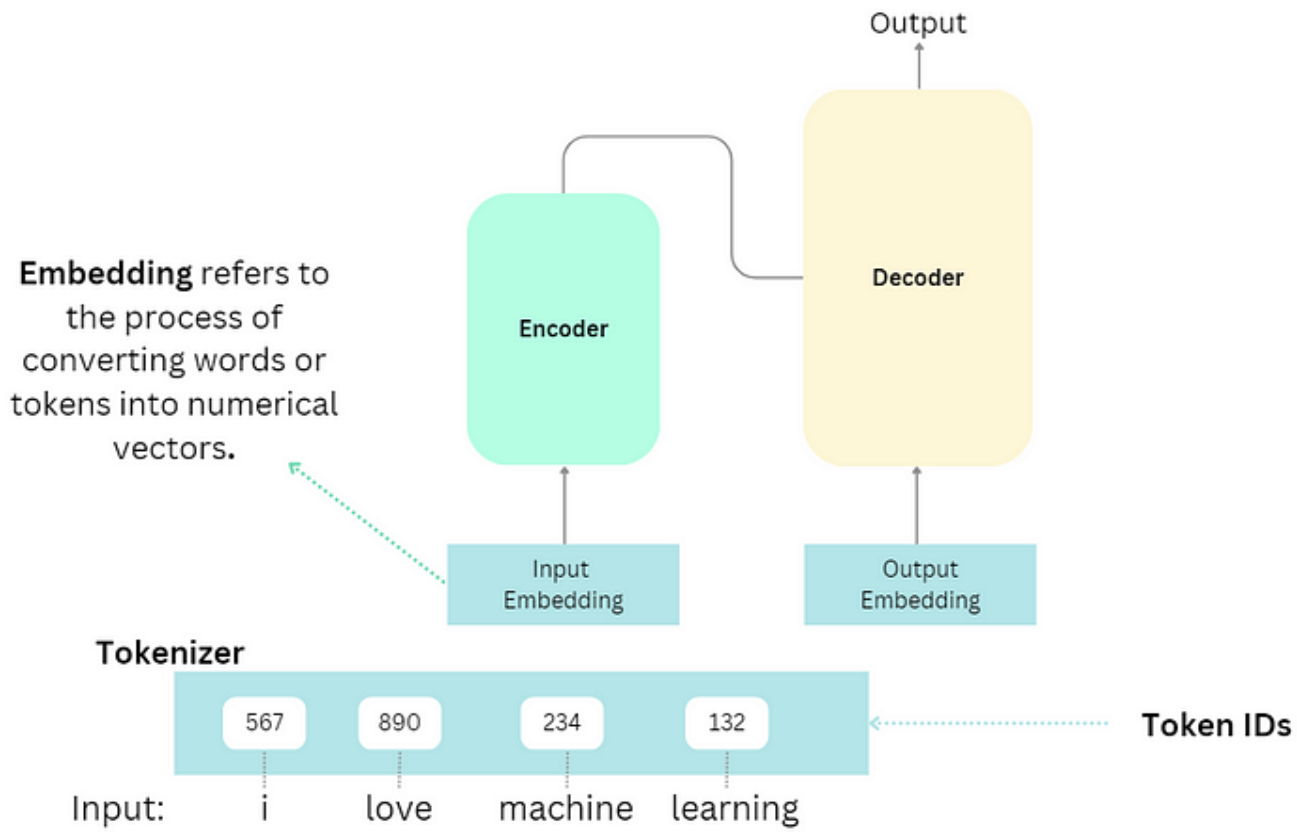
Now, let’s explore the details of the transformer architecture.

The transformer architecture consists of two main components: **the encoder and the decoder.**

The **Encoder** processes the input sequence, breaking it down into meaningful representations. On the other hand, a **Decoder** takes these representations and generates the output sequence, like a translation or a text continuation.



Machine learning models are essentially large statistical calculators, primarily designed to process numerical data rather than text. Therefore, before feeding text into the model for processing, we must first convert words into numerical tokens. This process is called **embedding**.



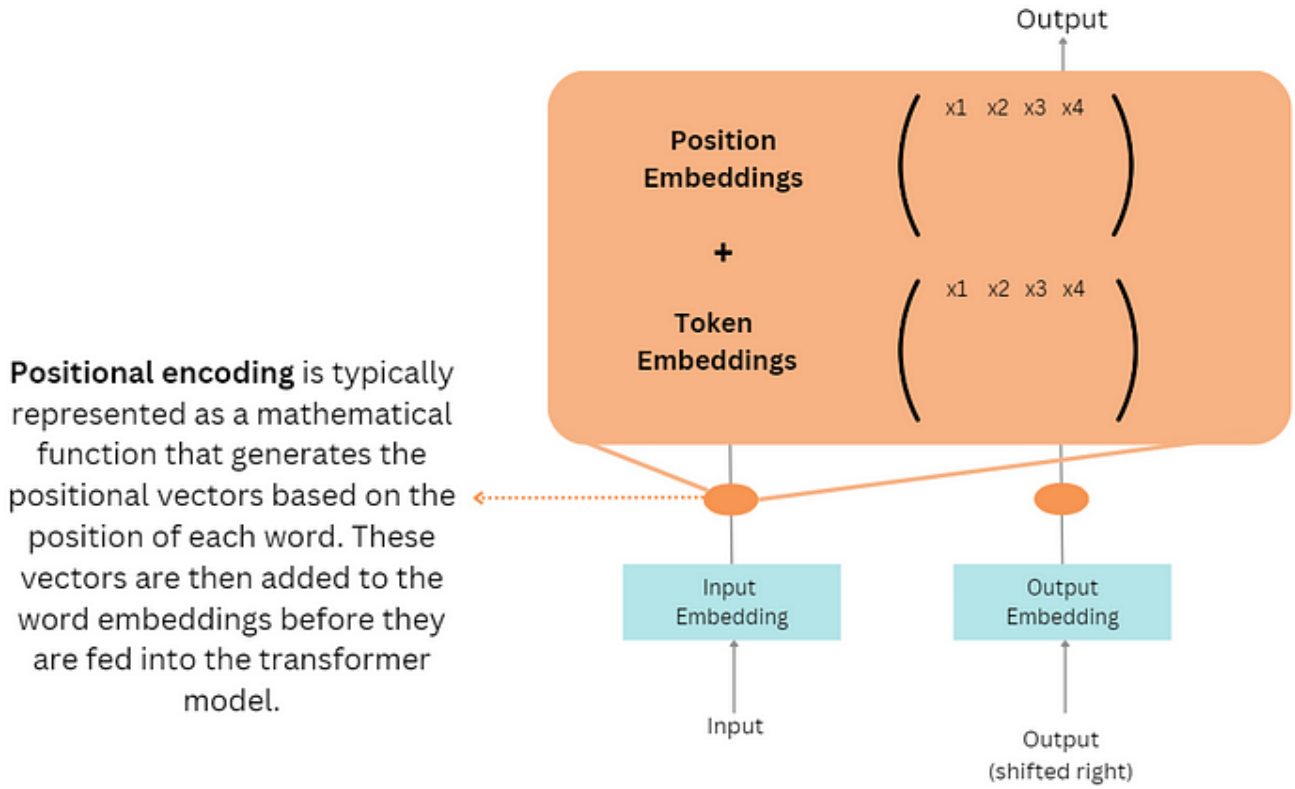
Embedding serves the following purposes:

1. Word to Vector Conversion: Each word or token in the input text is assigned a unique numerical vector. This vector represents the word's meaning and context within the given language. These word vectors are often pre-trained on vast text corpora and capture semantic relationships between words.

2. Embedding Layer: These word vectors pass through an "embedding layer" in the model. This layer acts as a lookup table, associating each word with its corresponding vector.

Next, we assign **Positional Encodings**.

Positional encoding is a technique used to provide the model with information about the position or order of words in a sequence. Since transformers process words in parallel rather than sequentially, they lack the inherent understanding of word order that other models, like recurrent neural networks (RNNs), have. Positional encoding addresses this limitation.



Here's how positional encoding works:

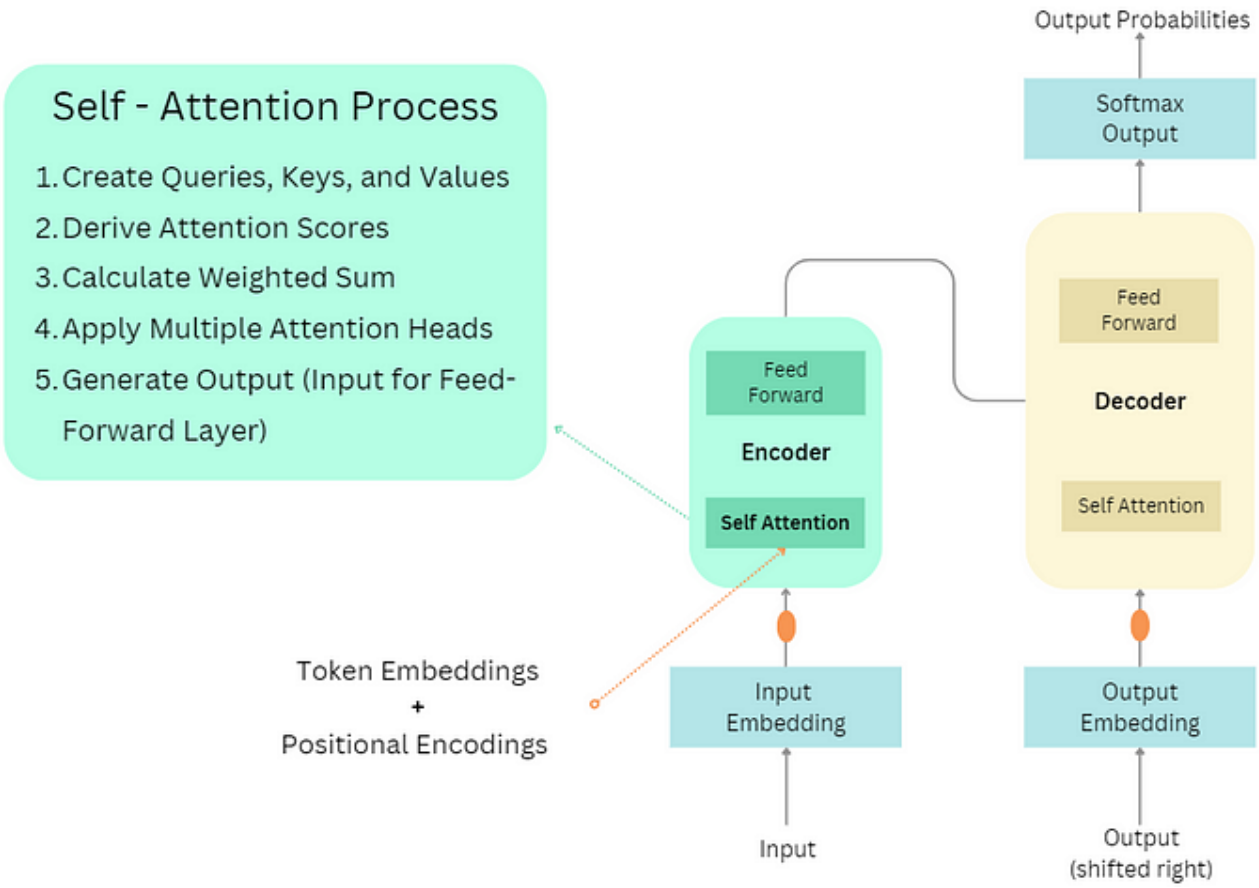
Assigning Positions and Preserving Order Information: Each word or token input sequence is assigned a unique positional encoding vector. These vectors represent the position of each word in the sequence.

Incorporating into Word Embeddings: These positional encoding vectors are added to the word embeddings of the input tokens. In essence, this adjustment augments each word's initial word embedding to encompass crucial positional information within the sequence. Additionally, this allows the model to differentiate between words with the same content but different positions in the sequence.

The token embeddings (input tokens) and position encodings are fed to the self-attention layer in the transformer model.

Self-Attention Layer

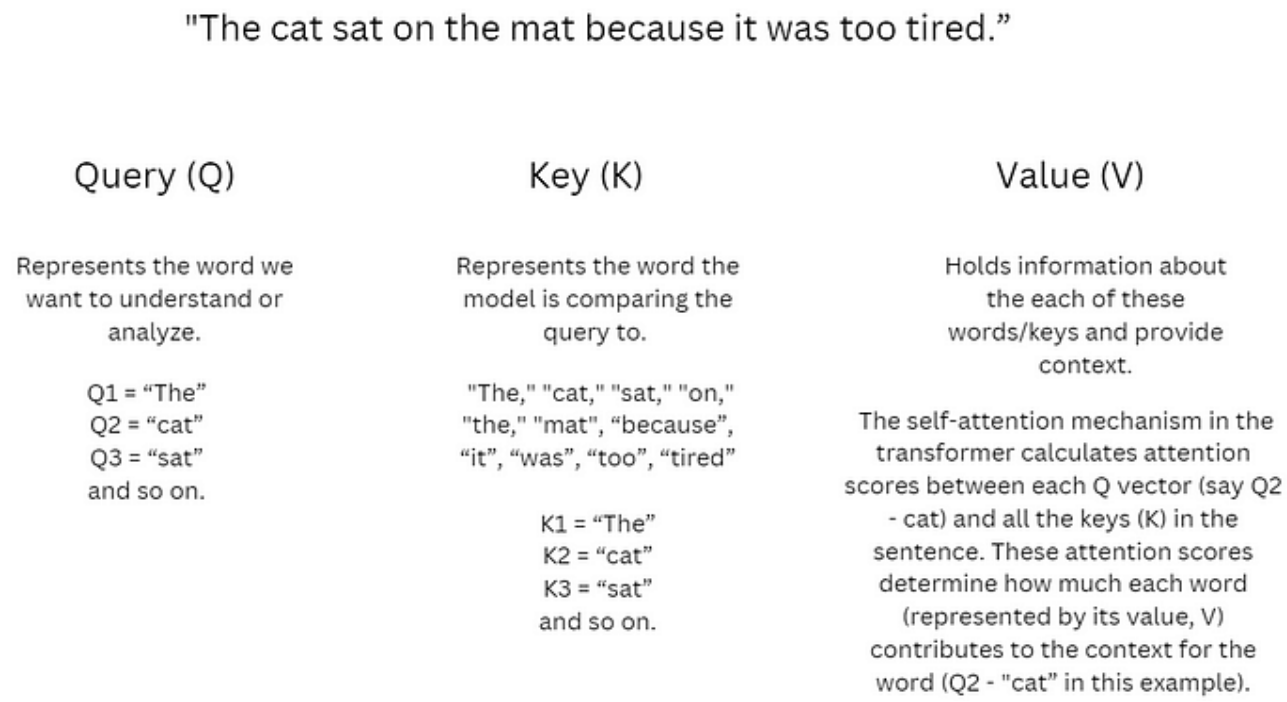
The **self-attention layer** is a pivotal component of the transformer architecture, and it is responsible for capturing relationships and dependencies between words in a sequence. Let's break down how the self-attention layer works step by step:



1. Create Queries, Keys, and Values: To understand how words relate to each other, the self-attention layer creates three sets of vectors for each word in the input sequence:

- : Represents the word we are currently focusing on. Each word has its corresponding Query vector.
- : Represents all words we want to pull information from to help determine the relevance of each word to the Query.
- : Contains the information of words that we'll extract when there's a match between Query and Key.

Let's outline the Q, K, and V values for the following sentence.



2. Derive Attention Scores: Next, the self-attention mechanism calculates attention scores between the Query vectors (Q) and the Key vectors (K). These scores indicate how much each word should pay attention to other words. Higher scores imply higher attention, suggesting that the word is more relevant to the Query. Lower scores suggest lower relevance or importance to the Query.

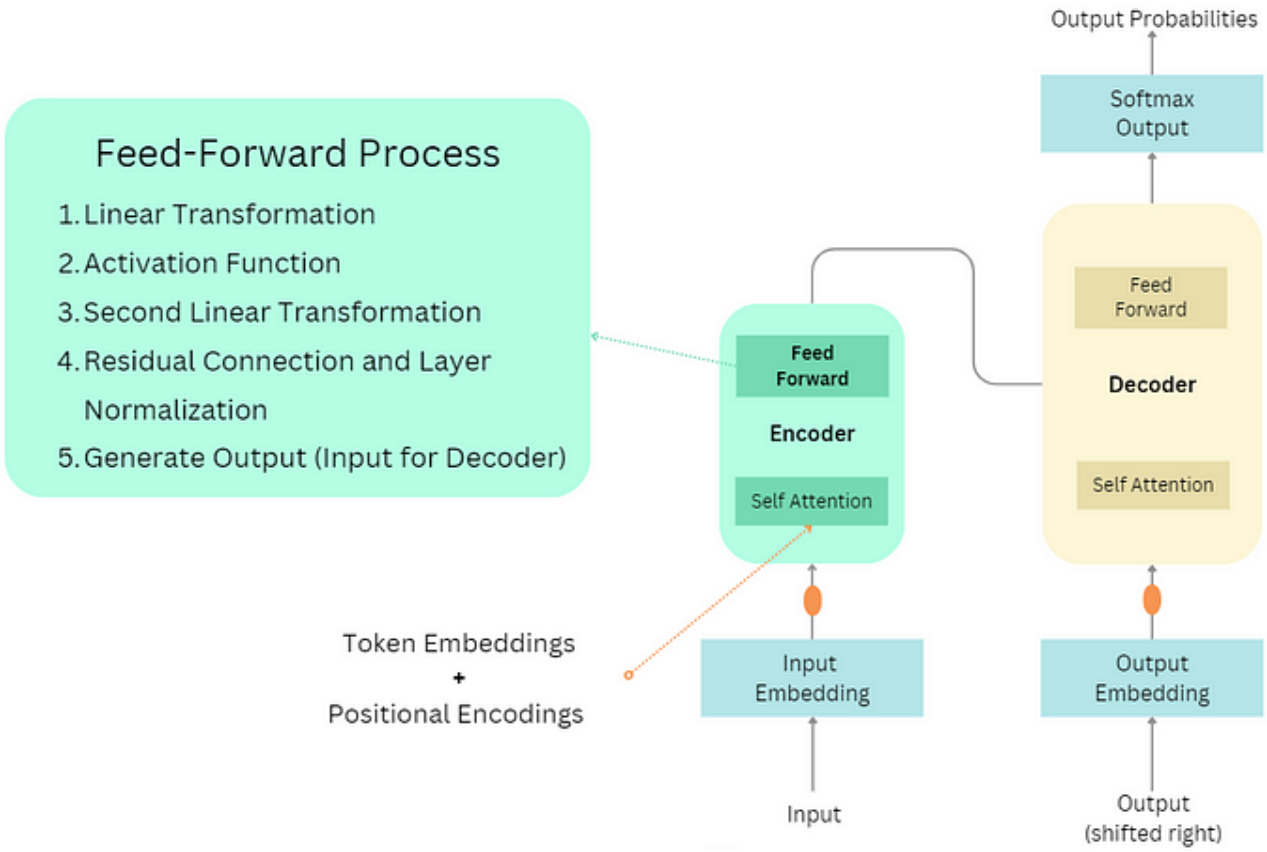
3. Calculate Weighted Sum: Using the calculated attention scores, the self-attention layer computes a weighted sum of the value vectors (V). Words with higher attention scores contribute more to this weighted sum. This step effectively combines the context of all words in the sequence for the Query word.

4. Apply Multiple Attention Heads: To enhance its understanding of context, the transformer often employs multiple sets of Queries, Keys, and Values, known as "attention heads." These heads allow the model to focus on different aspects of the sequence simultaneously. It's like having multiple people in a group discussion, each paying attention to different parts of the conversation.

5. Generate Output: The output contains contextualized representations of the words in the input sequence, taking into account their relationships with other words. This output now becomes the input for the Feed-Forward layer.

Feed-Forward Layer

The feed-forward layer in the transformer architecture is a neural network layer that is applied to the output of the self-attention layer. It is a position-wise transformation, meaning that it is applied to each position in the sequence independently of the other positions.



The feed-forward layer plays a multifaceted role in the transformer architecture, facilitating the modeling of long-range dependencies, introducing non-linearity for complex relationships, and boosting the model's capacity to process and extract meaningful information from input sequences.

Let's Simplify!

Imagine you have a big box of Lego pieces. Each Lego represents a word or a part of a sentence. Now, when you're building something with these Legos, sometimes you want to change the color or shape of a piece to make it fit better in your creation.

The feed-forward layer in the transformer is like a magical Lego workshop. You send your Lego pieces (words or parts of sentences) into this workshop one by one. Inside, there's a magical machine that can reshape and repaint each Lego based on how it should fit best in the bigger picture of your creation.

Once the Lego piece (word) has gone through the machine and has been reshaped and repainted, it comes out, ready to be a part of your amazing Lego world (sentence). And the best part? Even though every Lego piece goes through the same machine, each one might come out looking a bit different based on its unique role in your creation.

That's what the feed-forward layer does! It tweaks each word a little, so everything fits together better.

Let’s now delve into the details of feed-forward layer.

Linear Transformation: The first step in the feed-forward layer is to apply a linear transformation to the input. This involves multiplying the input by a learnable weight and adding a bias. This linear transformation projects the input into a different (often higher-dimensional) space, setting the stage for intricate interactions and transformations.

Activation Function: After the linear transformation, a non-linear activation function, such as Rectified Linear Unit (ReLU), is applied element-wise to the transformed input. The activation function introduces non-linearity, enabling the model to capture complex patterns in the data.

Second Linear Transformation: Following the activation function, another linear transformation is applied to the results. This transformation employs a different set of learnable weights and a bias term to further adapt the data, potentially altering its dimensionality, though not necessarily reverting to the original size. This step refines the representations further, preparing them for subsequent layers or final outputs.

Residual Connection and Layer Normalization: The transformer architecture addresses challenges like the vanishing gradient problem by using a residual connection. This means that the output of a layer is added to the initial input, allowing the model to learn to only make small changes to the input. Layer normalization is applied before adding the residual connection, promoting stability in training and ensuring consistent gradients throughout the layers.

Let’s Simplify!

1. Residual Connection (or Skip Connection): Imagine you’re trying to solve a math problem, and your friend suggests a intricate method to solve it. Instead of completely discarding your initial approach, you synergize both: your method and your friend’s, to derive the answer.

Similarly, in deep learning, as we stack more layers, the learning process might become challenging. Residual connections come to the rescue by letting the input of a layer bypass some intermediate layers and directly merge with the output. In essence, the network can blend the original information (akin to your method) with the more refined information (similar to your friend’s approach).

Mathematically, it looks like this: **Output = Input + Transformed Input**

2. Layer Normalization: Let’s say you’re baking cookies by following a specific recipe. But every time you bake, your oven’s temperature fluctuates a lot. This makes it hard to consistently bake good cookies. To solve this, you decide to use an oven thermometer to ensure the temperature is consistent every time.

Layer normalization functions much like that oven thermometer. In deep learning, the values (like the oven's temperature) can become too big or too small as they pass through layers. This can cause problems and make the network harder to train. Layer normalization, thus, standardizes these values across layers, ensuring consistency. Specifically, it adjusts inputs to a layer such that their average is 0 and their variability (standard deviation) is 1. This equilibration not only accelerates the learning process but also fortifies the model's adaptability to unfamiliar data.

In the intricate framework of the transformer architecture, both residual connections and layer normalization are indispensable. They bolster the learning process, particularly given the model's depth and inherent complexity.

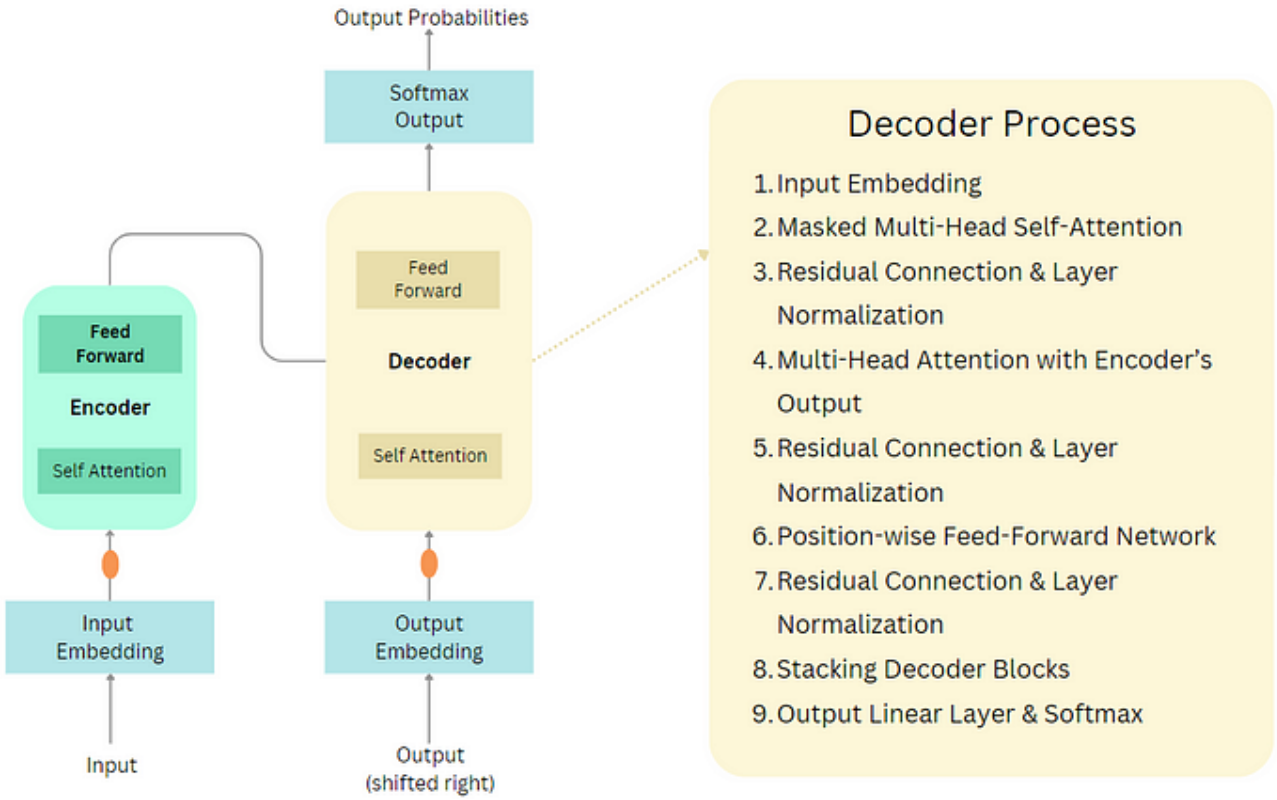
Now back to the Encoder process.

Output to Next Encoder Block or Decoder

If additional Encoder blocks are stacked above the current one, the output from this Encoder block (after layer normalization) becomes the input for the subsequent Encoder block.

However, if this is the last Encoder block, its output is forwarded to the Decoder. This output acts as the key and value for the Decoder's Multi-Head Attention mechanism, allowing it to attend to the Encoder's output.

Assuming the output has been passed to the Decoder block, the diagram illustrates the steps encompassed within the Decoder process.



Let’s break down the decoder process in the transformer architecture using a simple analogy.

Imagine you’re trying to translate a sentence from one language (say, French) to another (say, English). The decoder’s job is to produce the English sentence based on both the original French sentence and the bits of the English sentence it has generated so far.

Here’s a simple step-by-step explanation:

1. : The decoder takes in a part of the English sentence that has already been generated (in the beginning, this is just a start token).
2. : Just like people often think about how words in a sentence relate to each other when translating, the decoder checks how each word in the English input relates to the other words. This helps in making sure the translation flows naturally and maintains context. This step is similar to the self-attention mechanism in the encoder but is focused on the target sentence.
3. : Next, the decoder pays “attention” to the French sentence (the encoded version). It’s like when you glance back at the original French text to make sure you’re translating correctly. The decoder checks which parts of the French sentence are most relevant to the word it’s currently trying to produce in English.
4. : The decoder then processes the information it has gathered from the above steps, decides on the next English word (or token), and adds it to the output sequence.
5. : The decoder keeps doing this, generating one word at a time, until it decides the translation is complete and produces an end token.

Throughout these steps, the transformer utilizes layers of decoders (often multiple stacked on top of each other) to refine the translation, ensuring accuracy and context.

So, in a nutshell, the decoder in the transformer architecture is like a smart translator that refers back to both the original sentence and what it has translated so far to produce a coherent and accurate translation.

Now let’s dive into the details of the decoder process. The decoder in the transformer architecture is a multi-step process.

1. Just as with the Encoder, the input to the Decoder (which is the target sequence during training) is first embedded into continuous vectors. This embedded input is then added to the positional encoding to incorporate the sequence’s order information.
2. The Decoder employs a masked version of the self-attention mechanism, ensuring each position can only attend to positions before it (or to itself) in the sequence. This masking is essential during training to prevent a word from “seeing” future words, maintaining the autoregressive property of the Decoder. It’s important to note that this masking is only applied during training. During inference, the decoder can attend to all words in the target sequence, including future words.

3. A residual connection adds the output of the self-attention layer to its input. Layer normalization is then applied to stabilize and scale the activations.
4. This multi-head attention mechanism uses the Encoder's output as its keys and values and the output from the Decoder's self-attention as its queries. It allows the Decoder to focus on relevant parts of the source sequence while generating the target sequence.
5. The output from the Multi-Head attention layer is combined with its input using a residual connection. This is followed by layer normalization.
6. The Decoder also has a position-wise Feed-Forward layer, similar to the one in the Encoder. This network consists of two linear (dense) layers with a ReLU activation function in between. It operates on each position independently, adding more expressive power to the Decoder.
7. : The output of the Feed-Forward network is combined with its input via a residual connection. Another layer normalization is applied.
8. Just as multiple Encoder blocks are stacked in the Transformer, multiple Decoder blocks are stacked as well. The output from one block serves as the input to the subsequent block, passing through the above operations repeatedly.
9. Once the data passes through all the Decoder blocks, it goes through a final linear layer which maps it to the desired output vocabulary size. A softmax function is then applied to produce the probability distribution over the target vocabulary, generating the final output sequence.

To summarize, the Decoder in the Transformer architecture processes its input through self-attention, cross-attention with the Encoder's output, and position-wise Feed-Forward networks, repeatedly for each stacked block, culminating in a final output sequence after the softmax operation.