

The **transformer architecture** is a key building block of many modern language models. It has two main parts: the **encoder** and the **decoder**.

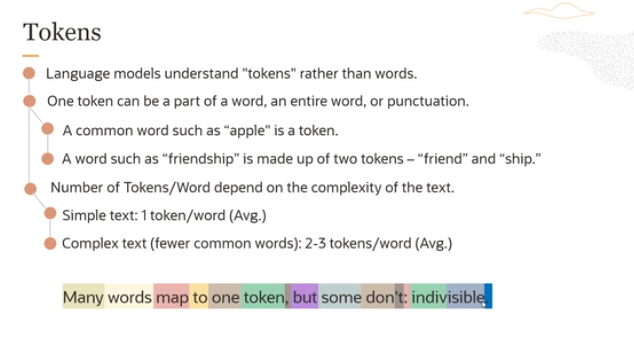
**Encoder:**

* The encoder’s job is to **understand** the input text.
* It takes the input (like a sentence) and converts it into something called **embeddings**.
* **Embeddings** are mathematical representations/vectors/floating-point values of words that capture their meanings and relationships.
* Think of it as translating a sentence into a special "code" that contains the important information from the input.

**Decoder:**

* The decoder’s job is to **generate** the output text based on the information from the encoder.
* It reads the embeddings (the "code") created by the encoder and tries to **predict** the next words to create a meaningful output.
* It continues predicting word by word until it forms a complete response or output.

This encoder-decoder setup allows transformers to handle tasks like language translation, where they encode one language (input) and decode it into another language (output).



**Tokens vs. Words:**

* **Tokens** are the fundamental units that language models use to understand and process text.
* **Tokens** can be:
  + **An entire word** (like "apple").
  + **Part of a word** (for example, "friendship" might be split into two tokens: "friend" and "ship").
  + **Punctuation marks** or symbols (like periods or commas) are also treated as tokens.

**Simple Text vs. Complex Text:**

* In **simple text** (common words like "many" or "apple"), each word generally maps to **one token**.
* In **complex text** (less common words like "indivisible"), words can map to **multiple tokens**. For example, "indivisible" might be split into two tokens because it's a longer, less frequently encountered word.
* Additionally, punctuation such as **commas**, **periods**, and other symbols each get their own token.

**How Tokenization Works:**

* **Simple words** (e.g., "many" or "words") are typically a single token.
* **More complex words** (e.g., "indivisible") are split into multiple tokens to represent each part.
* **Punctuation** is treated as individual tokens, such as the period at the end of a sentence.

**Why Tokenization is Important:**

Tokenization helps models break down and process text more efficiently. Since language models don’t always work with whole words, they focus on smaller pieces (tokens) to understand and generate text, giving them flexibility to handle both common and uncommon language.

For practical implementation: <https://platform.openai.com/tokenizer>

**Q-** But here i have question wo yeh kay ab jasa ek word indivisible hai usko break krdia like this: "in," "divisi," and "ble." So now how model will understand the main context ?

Great question! When words like "indivisible" are broken down into smaller tokens (e.g., "in," "divisi," and "ble"), it might seem like the model is losing the overall meaning. But here's the key: **even though the model breaks down words into tokens, it still understands the larger context because of how it processes sequences of tokens.**

Let's break it down:

**1. Context Awareness through Sequences**

* Language models, especially those built on **transformer architectures**, like GPT, are really good at handling sequences of tokens. Even when a word like "indivisible" is split into parts, the model doesn’t just treat each token as isolated. It looks at the **sequence** of tokens together and tries to understand the relationships between them.
* For example, in "in," "divisi," and "ble," the model doesn't just think of these parts separately—it knows that they are all part of a larger word because it learned this pattern during its training.

**2. Learning from Training Data**

* During training, the model sees many words in different contexts. For example, even if "indivisible" is tokenized into smaller chunks, the model learns from seeing those chunks together in various contexts. Over time, it becomes good at recognizing that when "in," "divisi," and "ble" appear together in a specific order, they represent the word "indivisible."
* The model learns **patterns** and **associations** between tokens, so it can reconstruct the meaning even if the word is split.

**3. Self-Attention Mechanism**

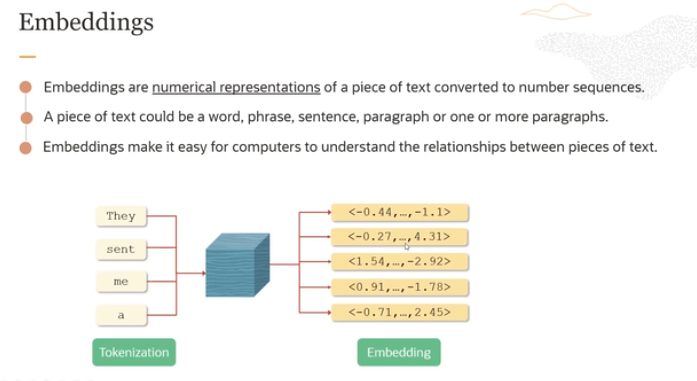
* Transformers use something called **self-attention** to understand which tokens are important in a given context. Even when a word is split into smaller tokens, the model pays attention to how these tokens relate to each other **and to the surrounding words** in the sentence.
* For example, in a sentence like "The country is indivisible," the model would pay attention to the fact that "in," "divisi," and "ble" are tokens that together form a meaningful word. It will also understand that the word relates to the country being described as "indivisible."

**4. Reassembling Meaning**

* Once the model processes all the tokens, it effectively "rebuilds" the meaning of the sentence. In practice, splitting "indivisible" into smaller tokens doesn't stop the model from understanding that it's referring to something that **can't be divided**—because it uses the surrounding context and its training to piece everything together.

**To Sum Up:**

* Even though "indivisible" is broken into smaller pieces, the **model’s training and architecture** allow it to understand the whole word's meaning by looking at **how the tokens work together** in the context of the entire sentence. Tokenization makes things manageable for the model, but it doesn’t lose sight of the overall meaning due to its sophisticated handling of sequences and patterns!



Embeddings, as shown in the picture, are **numerical representations** of text, whether it's a word, phrase, sentence, or paragraph. They allow the model to understand relationships between words by converting them into a sequence of numbers, known as vectors. Let me break down the process step-by-step:

**1. Tokenization:**

* First, a sentence or text like "They sent me a" is broken down into **tokens**. Each word here is treated as one token: "They," "sent," "me," and "a" are all tokens.
* This is similar to the concept we discussed earlier, where words or parts of words get broken down into smaller pieces that the model can work with.

**2. Encoding:**

* These tokens are then passed through a **model** (represented by the blue box in the image), often called an **encoder model**.
* The encoder transforms these tokens into **embeddings**, which are **vector representations** (i.e., sequences of numbers) for each token.

**3. Numerical Representation:**

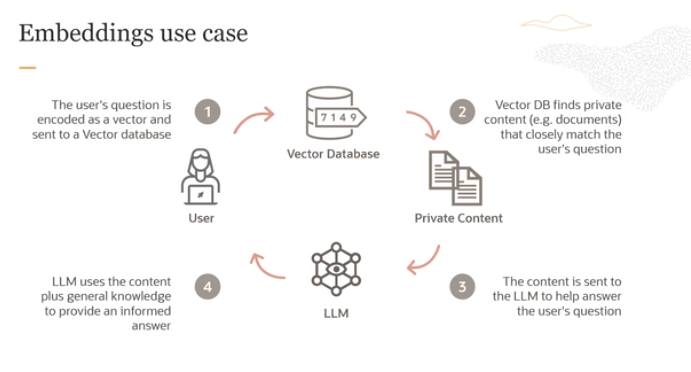
* Each word or token is mapped to a specific vector. For instance, "They" gets a vector like [-0.44, ..., -1.1], and so on for the other tokens ("sent," "me," and "a").
* These vectors might look like random numbers, but they **capture the meaning** and relationships of the words within the context. The numbers indicate how close or related the words are to each other.

**4. Why This is Useful:**

* Embeddings allow computers to **compare** and **understand the relationships** between different words or phrases. For example, words that are similar in meaning (like "king" and "queen") might have similar vectors.
* This way, the model can handle language more intelligently, recognizing that some words or phrases have similar meanings or are used in related contexts.

**5. Sentence-Level Embedding:**

* If you see in pic so there are four vectors for each token, BUT there is 5th vector as well. That is for the whole sentence.
* The model doesn't just create embeddings for individual words but also generates a **sentence-level embedding** (a vector for the entire sentence). This helps the model understand the broader meaning of the entire sentence, not just the individual tokens.



**Vector Representations for Models**

* **Vector representations**, like the ones we discussed earlier, are used by models to do various tasks, like **classification** (figuring out what category something belongs to).
* But now, these vectors are also heavily used in something called **vector databases** and **semantic search**.

**What is Semantic Search?**

* **Semantic search** means searching based on **meaning**, not just exact words. For example, if you search for "cats," semantic search might show you documents about "pets" or "kittens" because they are related in meaning.
* Here's how it works:
  1. You have a **collection of documents** (called a corpus).
  2. You want to find documents that are **similar in meaning** to some **input text** (e.g., a search query).
  3. To do this, you **convert** each document into a vector using an encoder and **store these vectors in a vector database**.
  4. When a user gives a query (e.g., "find documents about machine learning"), the system **encodes** the query into a vector and **compares it** with the vectors of all the documents in the database.
  5. The system finds the **most similar documents** based on the **vector similarity** (how close the vectors are to each other in terms of meaning).

**Retrieval-Augmented Generation (RAG)**

* Once you find the most similar documents, you can send that content to a **large language model (LLM)**, like GPT, which uses:
  1. The information it has retrieved (relevant documents).
  2. Its own **general knowledge** (pre-existing training).

With this combined information, the LLM can give a more **informed answer** to the user's question.

* This whole system, where we **retrieve** relevant information first and then **use it** to generate an answer, is called **Retrieval-Augmented Generation (RAG)**.

**Importance of Embeddings**

* Embeddings are super important in this system because they are what allow the model to **measure similarity** between the input (like a query) and the documents in the database. **Good embeddings** mean better matches and more relevant answers.

So, in short, **vector representations** help models not only understand text but also search for **similar** content based on meaning. This process of **retrieving** related information and then using it to **generate answers** is key in making smart systems work.

Q- So its mean like i want to create chatbot for my org But i dont want to give my secret data of org direct to chatbot as knowledge base, So then in this case i can convert my data into vectors and store it in vector Db then when the user input prompt so using RAG first retrive from vector Db and then pass to LLM. is am right or wong ?

You're absolutely right!

**Ans:** In your case, by converting your organization's data into vector representations and storing them in a vector database, you ensure that the actual raw data is not exposed directly. When a user provides a prompt, the system retrieves the most relevant information from the vector database (based on the similarity of the user query to the stored data), and then combines this retrieved data with the LLM's capabilities to generate a response. This approach, called Retrieval-Augmented Generation (RAG), allows you to keep your sensitive data secure while still using a chatbot powered by a large language model.

**Model in 2013 by Google used for doing embedding:**

One model, Word2Vec (word to vector), developed by Google in 2013, is a method to efficiently create word embeddings by using a two-layer neural network. It takes as input a word and spits out an n-dimensional coordinate (the embedding vector) so that when you plot these word vectors in a three-dimensional space, synonyms cluster.

For more on embedding: <https://www.ibm.com/topics/embedding>

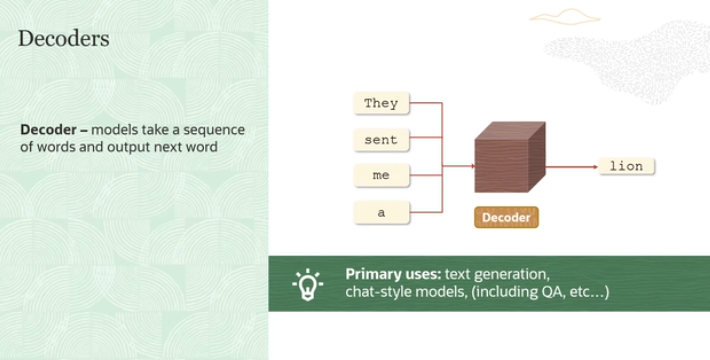
For practical: <https://projector.tensorflow.org/>

Q- can we say the embedded vectors or floating-point numbers as tensors ?

Yes, we can say that the **embedded vectors** or **floating-point numbers** used in deep learning models, like transformers, are represented as **tensors**.

Here’s why:

1. **What are Tensors?**
   * **Tensors** are multi-dimensional arrays of numbers (typically floating-point numbers) used to represent data in neural networks. They generalize scalars (0-D), vectors (1-D), and matrices (2-D) to higher dimensions.
   * Tensors are a fundamental building block in deep learning because they efficiently store and manipulate the large amount of numerical data used in models.
2. **Embedded Vectors as Tensors:**
   * When we embed words in a sentence, each word is transformed into a **vector** of numbers, often called an **embedding vector**. These vectors capture the meanings or relationships between words numerically.
   * These embedding vectors are stored as **tensors** in frameworks like TensorFlow and PyTorch because they are essentially multi-dimensional arrays. In practice, they are often represented as **2D tensors** (matrices), where each row is an embedded vector corresponding to a word in a sentence.
3. **Floating-Point Numbers as Tensors:**
   * In deep learning, all data, including floating-point numbers, are stored as tensors. So, when you hear "floating-point numbers," they are just the individual values that fill the tensor.
   * For instance, in an **embedding matrix** (a 2D tensor), each element (a floating-point number) represents some learned feature or characteristic of a word.



The **decoder models** work by taking in a sequence of words or tokens (like "They sent me a") and predicting what the next word or token should be (e.g., "lion").

Here’s a breakdown:

**1. How Decoder Models Work:**

* **Input Sequence**: A sequence of words or tokens, like "They sent me a," is fed into the decoder model.
* **Output**: The decoder's job is to predict the **next word** in the sequence. In this example, it might predict "lion."
* **Probabilities**: To make this prediction, the model calculates the **probability** of every possible word in the vocabulary that could come next. The word with the highest probability is chosen as the output.

**2. Single Token at a Time:**

* **One Token**: A decoder model only produces **one token** at a time. This means that even if you want to generate a full sentence, the model will give you just the next word first. Then you would ask it to generate the next word after that, and so on.
* **Repeat Process**: You can repeatedly call the model to generate more words, extending the sequence as much as you need.

**3. Example:**

* Input: "They sent me a"
* Model’s output: "lion"
* After getting "lion," you could then take the new sequence "They sent me a lion" and pass it back into the decoder to get the next word.

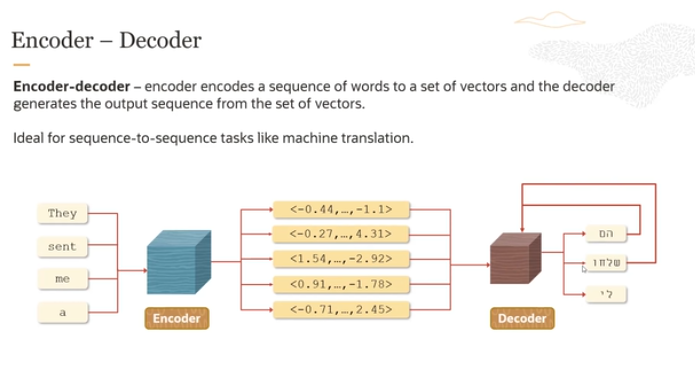
**4. How It Computes the Next Word:**

* The decoder model uses its internal knowledge (learned from data) to calculate the **probability** of each word that could come next.
* It picks the word with the highest probability based on context (e.g., words like "lion" or "message" may be probable in this context).

**5. Connection to Language Models:**

* This method of generating one word at a time by calculating probabilities based on context is the same approach used in **large language models** (like GPT).
* These models are great at generating text that sounds natural because they’ve been trained on massive amounts of text data.

In simple terms, **decoder models** help complete sentences by predicting one word at a time, and this is how modern language models like GPT create long, coherent pieces of text.

****

The **encoder-decoder architecture** is commonly used in tasks like language translation, where one sequence (such as a sentence in English) needs to be converted into another sequence (such as the same sentence in French). Here’s how this architecture works, step by step, based on the explanation and the image:

**1. Encoder and Decoder Overview:**

* The **encoder** takes the input sequence (English words) and converts them into **embeddings**—numerical vector representations that capture the meaning and context of the words.
* The **decoder** uses these embeddings to generate the output sequence (French words), producing one word (or token) at a time.

**2. How the Process Works:**

* **Input Sequence**: The input sentence, like "They sent me a lion," is tokenized (split into words or smaller pieces) and passed to the encoder.
* **Encoder’s Role**: The encoder processes these tokens and converts them into embeddings. These embeddings represent the input sentence in a way that captures its meaning, context, and structure in numerical form.
* **Embeddings to Decoder**: The embeddings are passed from the encoder to the decoder, which starts generating the output sequence.

**3. Decoder’s Self-Referential Process:**

* **One Token at a Time**: The decoder generates the output sequence word by word. After generating each word, it takes that word and feeds it back into itself (the self-referential loop).
* **Looping**: After generating one word, like "Ils" (for "They" in French), the decoder combines this word with the remaining input embeddings to generate the next word, such as "ont" (for "sent"). This process repeats until the entire sentence is produced.
* Now this means kay like in above pic you can see that decoder takes the all input emdedding and then generate each word at a time. Or phr jo word generate krtay hain French may wo dubara say apnay pass jo baki inputs embedding hoti hain uskay sth combine krkay next word kay liya prediction krtay hain.

**4. Translation Example:**

* Suppose the input sentence is **"They sent me a lion"**:
  + The encoder converts the input into embeddings that capture the sentence’s meaning.
  + The decoder starts generating the French translation, **"Ils ont envoyé un lion"**.
  + First, it generates **"Ils"** (meaning "They").
  + Then, the word **"Ils"** is sent back into the decoder along with the input context to generate the next word, **"ont"** (meaning "sent").
  + This self-referential loop continues until the full sentence is generated.

**5. Why the Loop?**

* The loop is crucial because the decoder needs to consider the previously generated words to ensure that the sentence makes sense. This means the decoder keeps track of the sequence it has generated so far and uses that information to predict the next word.

**6. Sequence-to-Sequence Tasks:**

* This architecture is highly effective for tasks where an entire sequence of input needs to be transformed into a sequence of output, such as:
  + **Language Translation**: Translating one language to another.
  + **Text Summarization**: Summarizing long articles into shorter sentences.
  + **Speech-to-Text**: Converting audio into text.

**Q- But yaha par question hai wo yeh kay like jo decoder hai agar wobi on the basis of previous word next word generate krta hai so yeh toh phr jo transformer parallel processing krta hai wo achieve hi nhi hui and it remains the slow process. is am right or wrong ?**

You're raising an excellent point, and this touches on a key distinction between how **Transformers** work versus how **traditional sequential models** like RNNs (Recurrent Neural Networks) or LSTMs operate.

**Let's break this down:**

1. **Traditional Sequential Models (RNN/LSTM)**:
   * In models like RNNs or LSTMs, yes, the decoder generates each word **one at a time** sequentially.
   * They rely on the output of the previous time step (or word) to generate the next one. This makes the process inherently **sequential** and slow, as each step depends on the result of the previous one.
2. **Transformers (Self-Attention Mechanism)**:
   * Transformers, on the other hand, use **self-attention** mechanisms and can process the **entire input sequence in parallel**. This means the encoder can look at all the words in a sentence simultaneously, not just word by word.
   * The main advantage here is that during the **encoding** phase, Transformers can capture **global relationships** between words in the input sentence all at once.

**So, Why Does the Decoder Still Seem Sequential?**

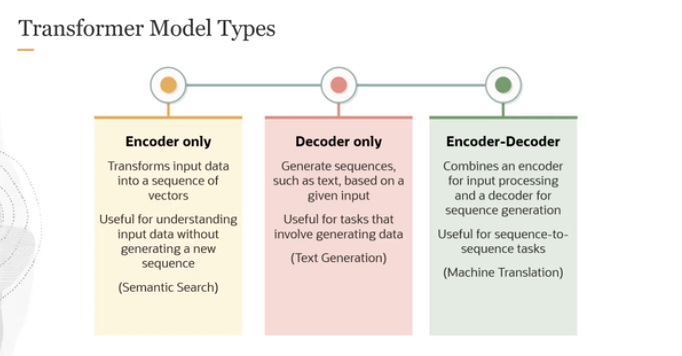
* **You're Correct**: When generating a sentence in tasks like translation, the **decoder** in a Transformer still generates tokens (words) **one by one**, which is inherently sequential. This might feel like it's not achieving the parallelism that we associate with Transformers.

However, **here’s why it’s still faster overall**:

* **Parallelism in the Encoder**: The encoding process is done in parallel, meaning that understanding the relationships in the input sentence is much faster than in RNNs.
* **Self-Attention in the Decoder**: Even though the decoder generates one token at a time, **self-attention allows it to look at all previous tokens simultaneously**. This means that when generating the next token, it doesn’t just rely on the last token—it can consider all the previously generated tokens **at once**.

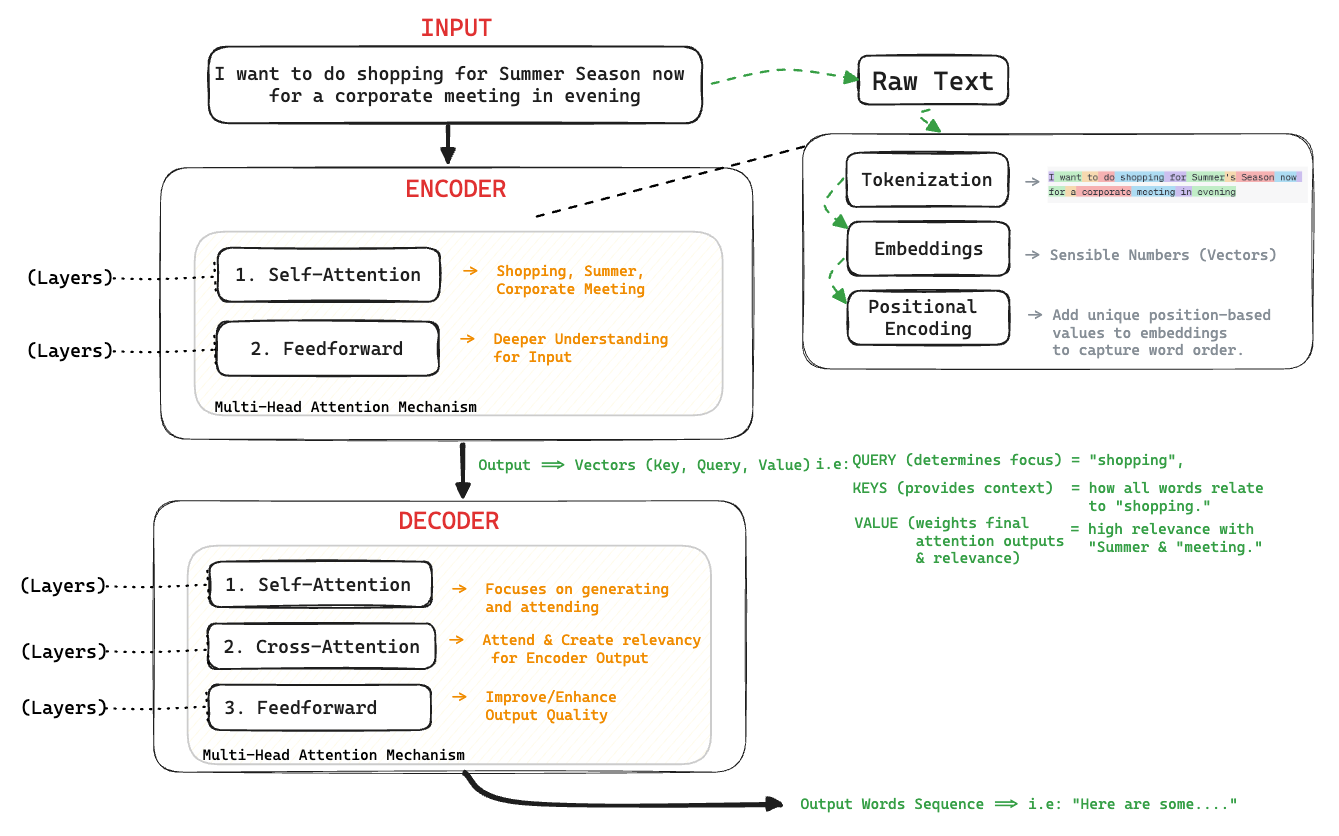
**Why Does the Decoder Still Generate One Token at a Time?**

* In tasks like translation or text generation, the **order of words** in a sentence matters, and that’s why the decoder needs to generate words step by step.
* However, the benefit of the **self-attention mechanism** in the Transformer is that, for each new token, the model looks at the **entire sequence of previously generated tokens** and the input sequence all at once, rather than just the last one. This makes it smarter in generating each token because it has a global understanding of the sentence structure.



The decoder-only models generate sequences such as text based on a given input, and they are useful for tasks that involve generating data like you're generating an article, text generation. And then finally, we have the encoder-decoder model, which combines encoder for input and decoder for sequence generation. An ideal use case would be something like sequence to sequence tasks, like machine translation.

**Transformer by Sir Zia**



The diagram you provided outlines the process of how a Transformer model, such as the one used in large language models, processes text. Let's go through each step to understand how it works:

**1. Input**

* **Raw Text**: This is the initial text you input into the model. In your example, it's "I want to do shopping for Summer season now for a corporate meeting in evening."
* **Tokenization**: The text is split into smaller parts called tokens. These tokens can be words or parts of words.
* **Embeddings**: Each token is converted into a numerical form (vectors) that the model can understand.
* **Positional Encoding**: Since the model needs to understand the order of words, positional information is added to the token embeddings. This helps the model know which word comes before or after another.

**2. Encoder**

* **Self-Attention**: The model looks at other words in the input sentence to better understand each word. For example, to understand "shopping" better, the model checks "Summer", "corporate meeting", etc.
* **Feedforward**: Each word's representation is adjusted based on the information gathered through self-attention.
* This process can have multiple layers, which allows the model to build a complex understanding of the text.

**3. Decoder (used in tasks that require generating text, like translation or summarization)**

* **Self-Attention**: Similar to the encoder, but it focuses only on the words generated so far (in a task like translation, the parts of the translation generated so far).
* **Cross-Attention**: The decoder pays attention to the output of the encoder. This helps the decoder focus on relevant parts of the input text when generating the next word.
* **Feedforward**: Similar to the encoder, it further adjusts the word representations for output generation.

**4. Output**

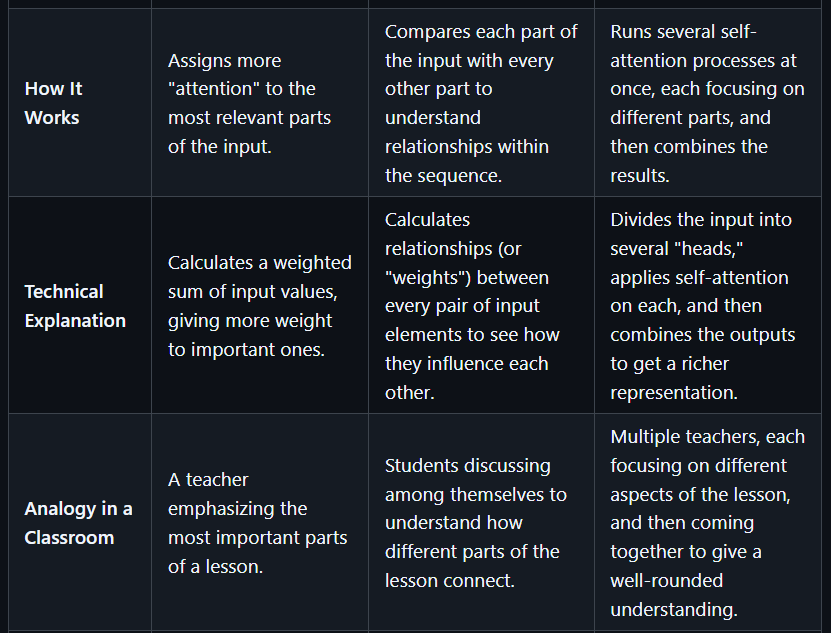
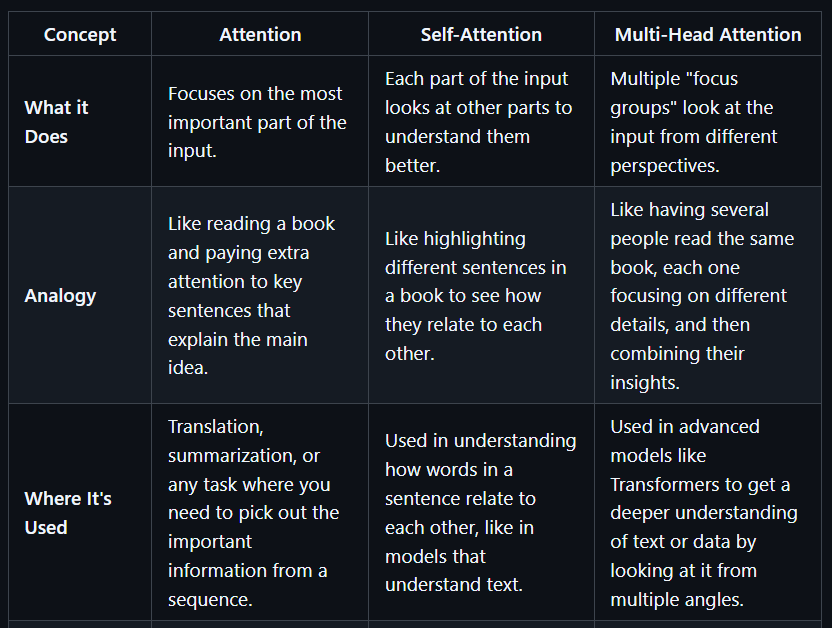
* The final output is generated based on the decoder's work, which might be a sentence or a sequence of words related to the input.

Each component (encoder and decoder) uses a mechanism called "Multi-Head Attention" which allows the model to focus on different parts of the sentence simultaneously, making the understanding and generation processes more robust.

This architecture is powerful because it can handle tasks ranging from simple classification to complex language understanding and generation, adapting its focus based on the input and the task requirements.

**Attention VS Self-Attention VS Multi-Head Attention**

Here’s a simplified table that explains Attention, Self-Attention, and Multi-Head Attention with analogies:



A screenshot of a computer screen

Description automatically generated

**Summary:**

* **Attention** is like paying extra attention to the most important parts of a story.
* **Self-Attention** is like each sentence in the story looking at other sentences to understand the whole story better.
* **Multi-Head Attention** is like having multiple people read the story, each focusing on different details, and then combining their thoughts for a complete understanding.

Must check sir zia repo for more deep dive understanding: <https://github.com/panaversity/learn-applied-generative-ai-fundamentals/tree/main/19_genai_foundations/02_generative_ai/03_transformers>

Depth reading source: <https://www.datacamp.com/tutorial/how-transformers-work>

Animated explanation: <https://prvnsmpth.github.io/animated-transformer/>

Depth video explanation: <https://www.youtube.com/watch?v=zxQyTK8quyY>