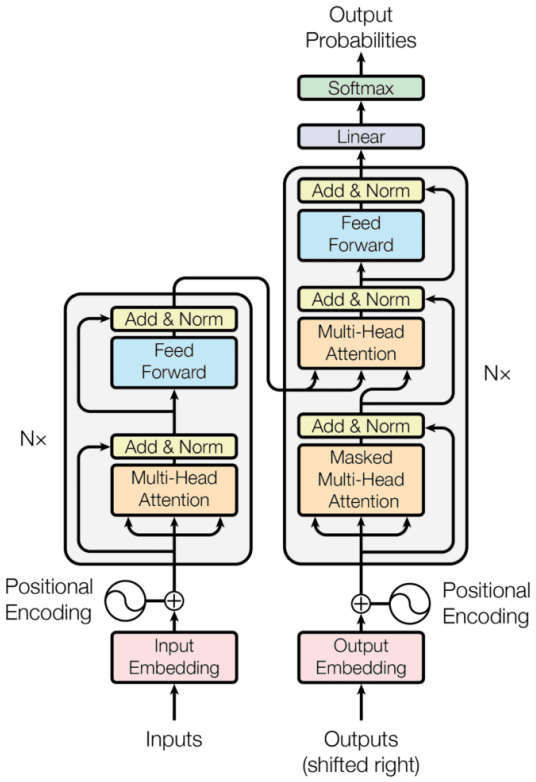
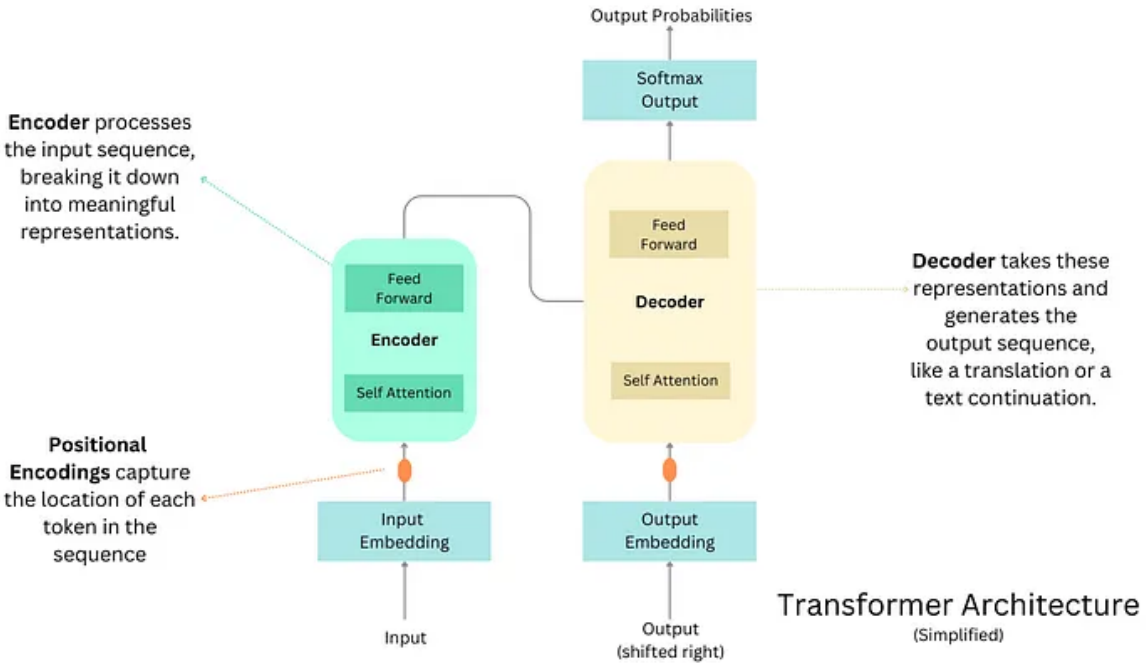
https://draw.chat/dc20ss9mbzt6aab3uuwptxjosiu8br

Transformers :

Understanding the Transformer Architecture:



The reason it is called the Transformer architecture is because the inputs go through a series of “transformations”.



A transformer accomplishes the following:

Transformers unlike other architecture like LSTM,RNN can remember whole context in sentence. For eg, it can remember grammatical, positional(how far) and relational context of a word with all other words in sentence.

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.

1. **Self-attention Mechanism**: The Transformer relies heavily on the self-attention mechanism, which allows it to capture relationships between different positions in the input sequence. This mechanism enables the

model to consider all positions simultaneously, unlike traditional architectures where information flow is sequential or localized.

2. **Parallelization**: Due to its attention mechanism and absence of recurrent connections, Transformers can be efficiently parallelized across different positions in the sequence. This makes training and inference faster compared to sequential architectures like RNNs, which are inherently sequential in nature.
3. **Long-range Dependencies**: Transformers excel at capturing long-range dependencies in sequences. Since attention is not restricted by distance, the model can theoretically capture dependencies between tokens that are far apart in the sequence, which can be challenging for RNNs and other sequential models.
4. **Positional Encoding**: Transformers incorporate positional encoding to provide information about the position of tokens in the sequence. This enables the model to understand the order of elements in the input sequence, which is crucial for tasks like language modeling and sequence generation.
5. **Encoder-Decoder Architecture**: The Transformer architecture naturally lends itself to encoder-decoder setups, where one Transformer stack serves as an encoder to process the input sequence, and another stack serves as a decoder to generate the output sequence. This design is particularly effective for sequence-to-sequence tasks like machine translation and text summarization.
6. **Transfer Learning**: Transformers have been pre-trained on large text corpora using unsupervised learning objectives like language modeling or masked language modeling. .

“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.

The **Encoder** processes the input sequence, breaking it down into meaningful representations. On the other hand,

Decoder takes these representations and generates the output sequence, like a translation or a text continuation.

Pre-processing (Tokens, Word Embedding and Positional Encoding)

(to get relation between words)

(what is posn of each word in sentence) to achieve parellelism

1. Each word is converted into a Token (numeric representation of the word) using a process called Tokenization.***— numeric representation of the word***
2.
3. For each word a Word Embedding is created which is a Tensor Matrix, that contain additional “attributes” about the word. ***represent the semantic meaning of the word and the relationships between the word and other words of the vocabulary in general***

1. An additional **“Positional Encoding”** is added to the Word Embedding, that signifies the relative position of each of the words in the sentence in relation to each other.***— — position of the word in the sentence***

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.

Word Embeddings are complex mathematical models that use vectors, tensors and weights to establish relationships between the tokens, thereby

creating context to the collection of tokens.

Tensors are basically multi dimensional arrays, that model some real words data (eg. Tokens).

`tensor1` is a 1-dimensional tensor with shape (3),

`tensor2` is a 2-dimensional tensor with shape (2, 3), and

`tensor3` is a 3-dimensional tensor with shape (2, 2, 3).

operations are fundamental to learning and creating context in machine learning and deep learning models.

1. **Element-wise operations:** These operations allow you to manipulate individual elements of tensors, which is essential for performing calculations and transformations in neural networks. For example, element-wise multiplication can highlight important features in the data, while addition can help combine information from different sources.
2. **Matrix operations:** Matrix operations are crucial for tasks like **linear transformations and projections**, which are fundamental in many machine learning algorithms. Matrix multiplication, for instance, is used extensively in neural networks for weight updates and activations.
3. **Reduction operations:** Reduction operations help summarize information across dimensions, such as computing the sum or mean of elements. These operations can be used to reduce the complexity of data or extract key features for learning.
4. **Concatenation operations:** Concatenation operations allow you to combine tensors along specific axes, which is useful for combining information from different parts of a model or dataset.
5. **Broadcasting operations:** Broadcasting operations help ensure that tensors have compatible shapes for element-wise operations, which simplifies computation and allows for more flexible model architectures.
6. **Element-wise comparison operations:** These operations are useful for tasks like classification, where you need to compare elements to make decisions or assign labels.

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.

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

Transformer Layers (Linear Transformation, Self Attention, Feed Forward Network)

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.

The purpose of the Self Attention Layer is to establish the *meaning of the individual word in relation to all the other words in the sentence*

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:

```
      name
     /
My => is
     \
      Upesh
```

sentence:" My name is Upesh"
So in this eg, my is query, {name,is,upesh} are keys and arrow representing relation is value
After this step, softmax is used to calculate prob to predict next word

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

Linear Transformation: A (learned) Linear Transformation process is applied to the (Word Embedding + Positional Encoding) for each word, to "project" the high dimensionality matrix to a lower dimensionality matrix. And for each word in the sentence the Transformer creates the Q, K, V vector.**-**

represent an individual word (Q) in relation to the other words in the sentence (K, V).As an example for a sentence "Hi how are you?" , following Q, K, V vectors may be created. eg.Q1 = "Hi" and K,V = ("how", "are","you")

Side note:Once the query vector Q1 = "Hi" is processed by the encoder side, the Query vector moves to the next word Q2 = "How" which is then provided as input to the Encoder, and so on, in a process called "Shifting Focus"Q2 = "How" and K,V = ("Hi", "are","you")Q3 = "are" and K,V = ("Hi", "how","you")Q4 = "you" and K,V = ("Hi", "how","are")

Linear Projections

Linear projections refer to specific types of linear transformations used to reduce matrix dimensionality and potentially extract relevant features from the data (word embeddings)

Word embeddings are dense tensors (high dimensional matrix) representing the meaning of each word. Before calculating **attention scores** the Transformer Architecture uses linear projections to map the original word embeddings (potentially high-dimensional) into lower-dimensional **query (Q), key (K), and value (V) vectors**

“Linear” projection indicates that a higher dimensional matrix is “project” onto a single dimensional matrix or in essence a vector (a vector is a single dimensional matrix).

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.

The Softmax function

Softmax is a function to convert the output of a model into probabilities, and are used to calculate the “attention weights” in the AI models

1. **Scaled Dot Product:** The Q, K, V values are fed to the Scaled Dot Product attention layer, which does a series of Matrix math *(dot product multiplication of the matrices, scaling of the result of the dot product multiplication, and softmax to provide a probability distribution between 0–1)*, and creates a context vector for that individual word.***— This provides a score of how relevant are the other words in that sentence are to that individual word. This in turn determines the “attention weight” that the individual word must give to the other words in that sentence.***

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.

1. **Multi-Head Attention:** There are several Scaled Dot Product attention layers that work in parallel comparing the individual word (Q) in parallel to all the other words (K, V) in that sentence at the same time. ***— -So in a sentence (“Hi how are you?”) If Q=”Hi” and K, V= (“how”, “are”,”you”) then the AI model compares the relevance of Q to each of the K, V in parallel (and not one at a time) i.e.“Hi” is compared to “how”,“Hi” is compared to “are”,“Hi” is compared to “you” etc. in parallel.***
2. ***This greatly speeds ups the computation, and also why transformers are considered “parallel” vs. the previous “sequence to sequence” models of LSTMs etc.***The results of all of the different Scaled Dot Products are then “combined” (via Concatenation and a Final Linear Transformation**), into a single output vector called the Context Vector (C1).**Also since different Scaled Dot Product attention layers work in parallel and the results are then combined, this part of the AI model is called the **Multi-Head Attention**.

The output of the Multi-Head Attention (Self Attention Layer) is the “context” or meaning of the individual word in relation to all the other words in the sentence

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.

The goal of the FFN is to extract additional features for each element potentially uncovering deeper meanings or subtle connections. It additionally learns non-linear patterns within the data of each word, which self-attention layer alone might miss.

It might uncover subtle nuances in word meaning, grammatical roles, or even stylistic elements

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.

Feed Forward Networks

The purpose of a FFN is to extract additional features for each element provided to it and potentially uncovering deeper meanings or subtle connections, and also to learn non-linear patterns within the data of each word, which self-attention layer alone might miss.

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.

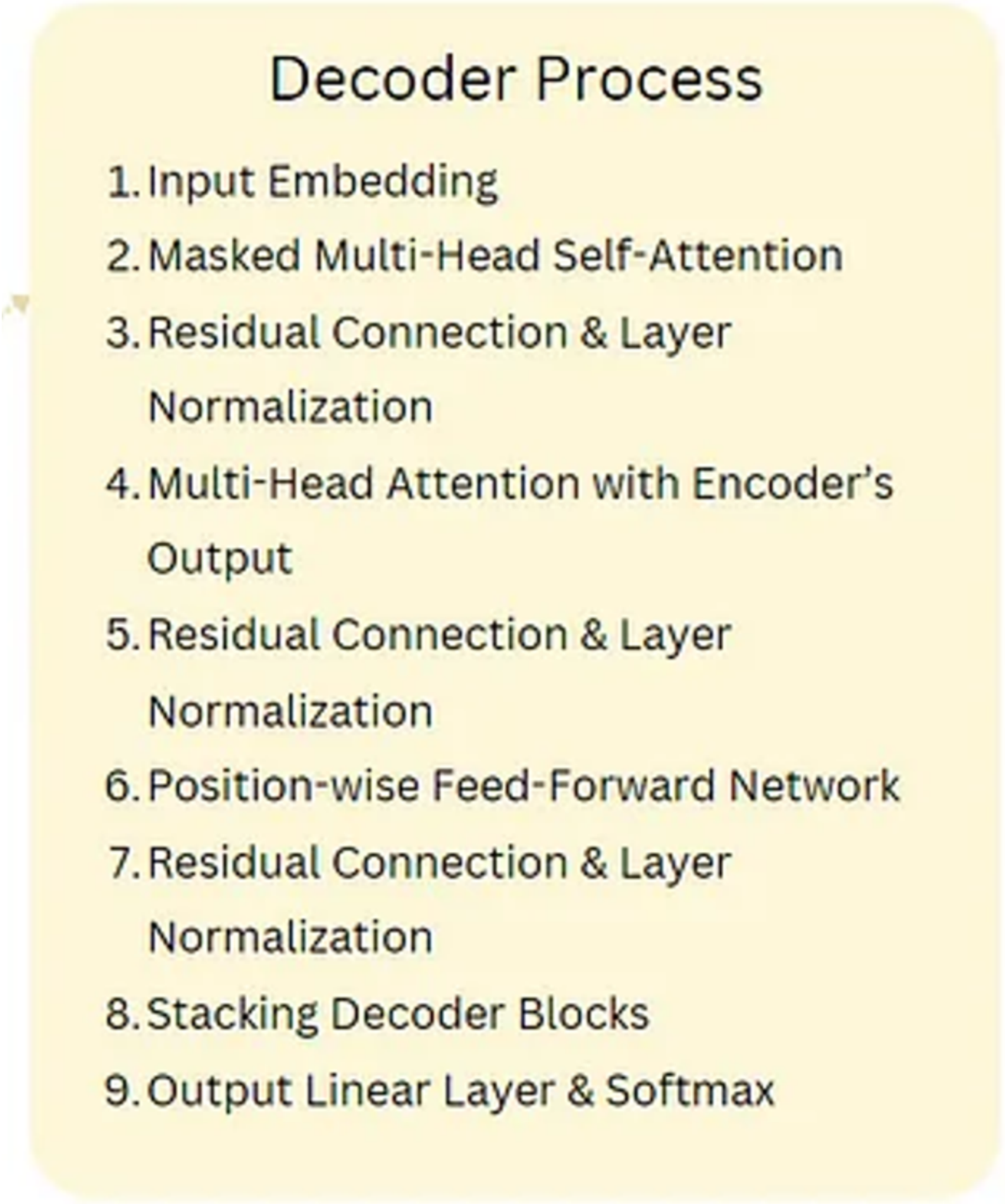
Residual Connection (or Skip Connection):

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).

Normalisation :

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.



1. **Input Embedding:** 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. **Masked Multi-Head Self-Attention:** 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. **Residual Connection & Layer Normalization (post Self-Attention):** 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. **Multi-Head Attention over Encoder's Output:** 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. **Residual Connection & Layer Normalization:** The output from the Multi-Head attention layer is combined with its input using a residual connection. This is followed by layer normalization.
6. **Position-wise Feed-Forward Network:** 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. **Residual Connection & Layer Normalization**: The output of the Feed-Forward network is combined with its input via a residual connection. Another layer normalization is applied.
8. **Stacking of Decoder Blocks:** 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. **Output Linear Layer & Softmax:** 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.

Summary

- Input (**Words**) →
- Tokens (**Numeric representation of the word**) →
- Word Embeddings + Positional Encoding (**semantic meaning and relative position of the word**) →
- Linear Transformations, **prepare the word to be compared to each other word in the sentence**(this was done using learned projections of the matrix to reduce the dimensionality of the matrix to simplify calculations and create Q, K, V vectors) →
- Self Attention Layer, **provide meaning and context of the word in relation to all the other words in the sentence** →(steps were, Dot Product → Scaling of Dot Product → Softmax (to normalize into 0–1, and generate context vector)
- Feed Forward Networks, **understand the deeper meaning of the individual word**