UNIT 2: TRANSFORMER 9 Hrs

Data preprocessing: Tokenization, Embedding, Positional Encoding - Transformer Architecture: Attention mechanism, Types of Attention, Normalization - LayerNorm, RMSNorm, Encoder, Decoder, Feed Forward Network and Softmax - Model scalability - Parameters, Layers, and Performance - Application: Time Series Data, Sequence Based Data, Text and Vision.

Data Preprocessing

Data preprocessing is a crucial step in generative AI that involves transforming raw data into a format that can be used to train models: It helps to clean, format, and restructure data to make it suitable for AI systems. Generative AI automates preprocessing tasks like missing value imputation and feature scaling. It predicts missing values and applies standardization techniques for data consistency and quality.

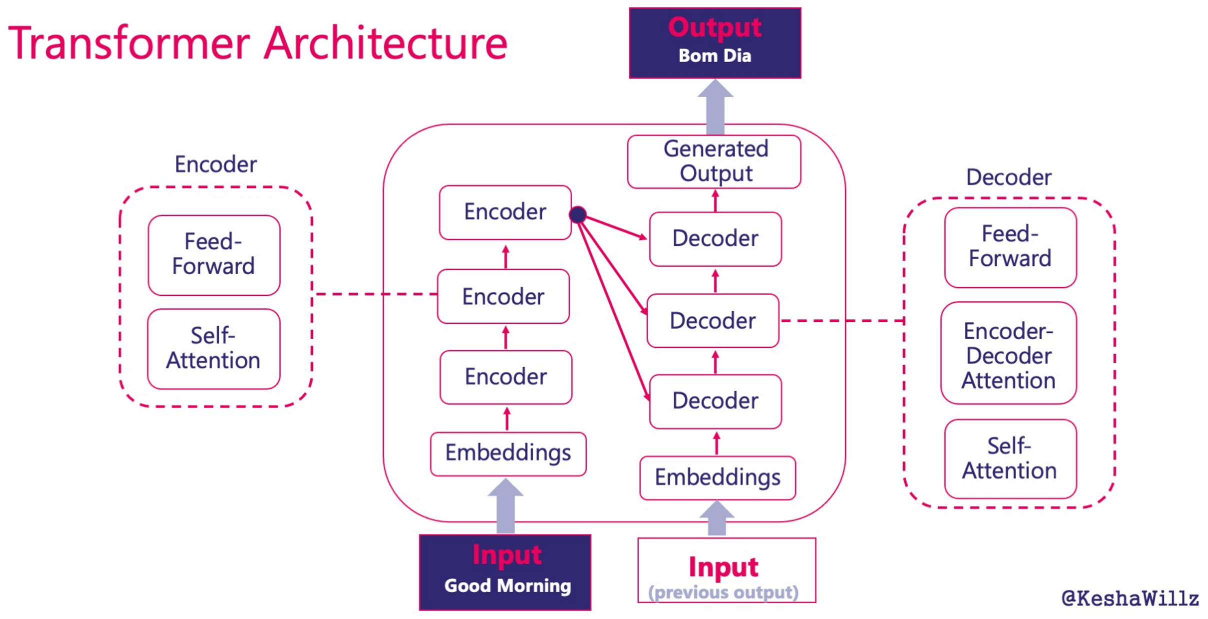
Data preprocessing tasks can include:

- Data cleaning
- Text-tokenization
- Resizing images
- Missing value imputation
- Feature scaling

A transformer is a deep learning architecture that adopts the mechanism of self attention which differentially weights the importance of each part of the input data. Like recurrent neural networks (RNNs), transformers are designed to process sequential input data, such as natural language, with applications for tasks such as translation and text summarization. However, unlike RNNs, transformers process the entire input all at once. The attention mechanism provides context for any position in the input sequence. Eventually, the output of a transformer (or an RNN in general) is a document embedding, which presents a lower-dimensional representation of text (or other input) sequences where similar texts are located in closer proximity which typically benefits downstream tasks as this allows to capture semantics and meaning.

How does the Transformer architecture work?

Transformer is an architecture of neural networks that takes a text sequence as input and produces another text sequence as output. For example, translating from English (“Good Morning”) to Portuguese (“Bom Dia”). Many popular language models are trained using this architectural approach.



The input

The input is a sequence of tokens, which can be words or sub words, extracted from the text provided. In our example, that's "Good Morning." Tokens are just chunks of text that hold meaning. In this case, "Good" and "Morning" are both tokens, and if you added an "!", that would be a token too.

The embeddings

Once the input is received, the sequence is converted into numerical vectors, known as embeddings, which capture the context of each token. These embeddings allow models to process textual data mathematically and understand the intricate details and relationships of language. Similar words or tokens will have similar embeddings.

For example, the word "Good" might be represented by a set of numbers that capture its positive sentiment and common use as an adjective. That means it would be positioned closely to other positive or similar-meaning words like "great" or "pleasant", allowing the model to understand how these words are related.

Positional embeddings are also included to help the model understand the position of a token within a sequence, ensuring the order and relative positions of tokens are understood and considered during processing. After all, "hot dog" means something entirely different from "dog hot" - position matters!

The encoder

Now that our tokens have been appropriately marked, they pass through the encoder. The encoder helps process and prepare the input data — words, in our case — by understanding its structure and nuances. The encoder contains two mechanisms: the **self-attention** and **feed-forward** mechanisms.

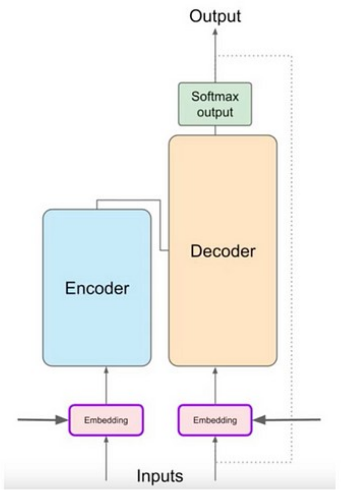
The **self-attention mechanism** relates every word in the input sequence to every other word, allowing the process to focus on the most important words. It's like giving each word a score that represents how much attention it should pay to every other word in the sentence.
The **feed-forward mechanism** is like your fine-tuner. It takes the scores from the self-attention process and further refines the understanding of each word, ensuring the subtle nuances are captured accurately. This helps optimize the learning process.

The decoder

At the culmination of every epic Transformers battle, there's usually a transformation, a change that turns the tide. The Transformation architecture is no different! After the encoder has done its part, the decoder takes the stage. It uses its own previous outputs — the output embeddings from the previous time step of the decoder — and the processed input from the encoder.
This dual input strategy ensures that the decoder takes into account both the original data and what it has produced thus far. The goal is to create a coherent and contextually appropriate final output sequence.

The output

At this stage, we've got the "Bom Dia" — a new sequence of tokens representing the translated text. It's just like the final roar of victory from Optimus Prime after a hard-fought battle! Hopefully, you've now got a bit more of an idea of how Transformer architecture works.



.

Components of Transformers Architecture

Tokenization

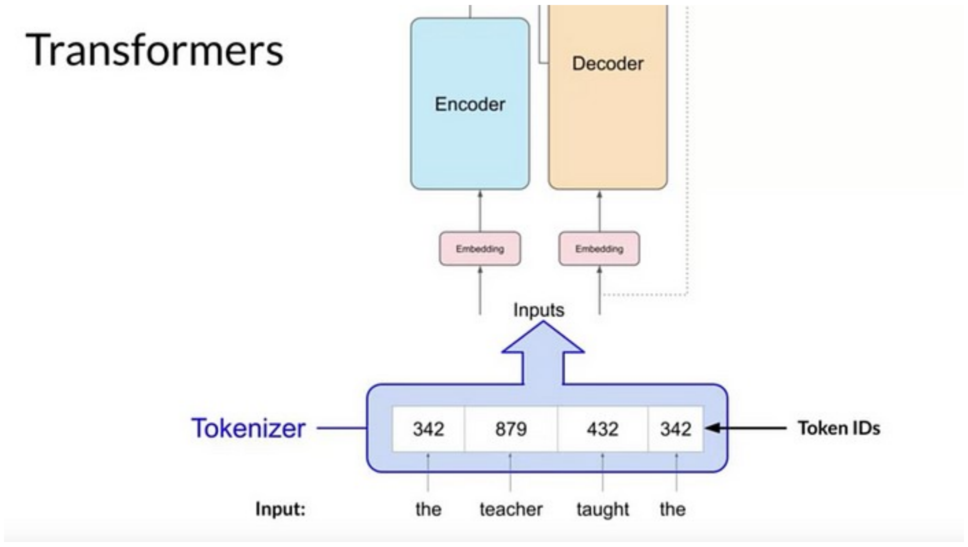
Tokenization is a preprocessing technique in natural language processing (NLP). It breaks down unstructured text data into smaller units called tokens. A single token can range from a single character or individual word to much larger textual units. Tokenization is one stage in text mining pipelines that converts raw text data into a structured format for machine processing

Character tokenization

One issue that can arise when using a word-level tokenizer is unknown word tokens. Out-of-vocabulary (OOV) terms (that is, words not recognized by a tokenizer with a pretrained vocabulary) might be returned as unknown tokens ([UNK]). OOV terms can arise if one uses a tokenizer with a pretrained vocabulary. Character tokenization is one method of solving for this. Because character tokenization tokenizes at the character level, the chances of meeting OOV terms is miniscule. But character tokens in and of themselves might not provide meaningful or helpful data for NLP tasks that focus on word-level units, such as word embedding models.

Sentence tokenization

Sentence tokenization has several use cases, such as sentiment analysis tasks or machine translation. For example, with regard to machine translation, a word's significance or meaning in another language cannot always be determined in isolation from its context. In this case, you might prefer a sentence tokenization algorithm as opposed to word-level tokenization.



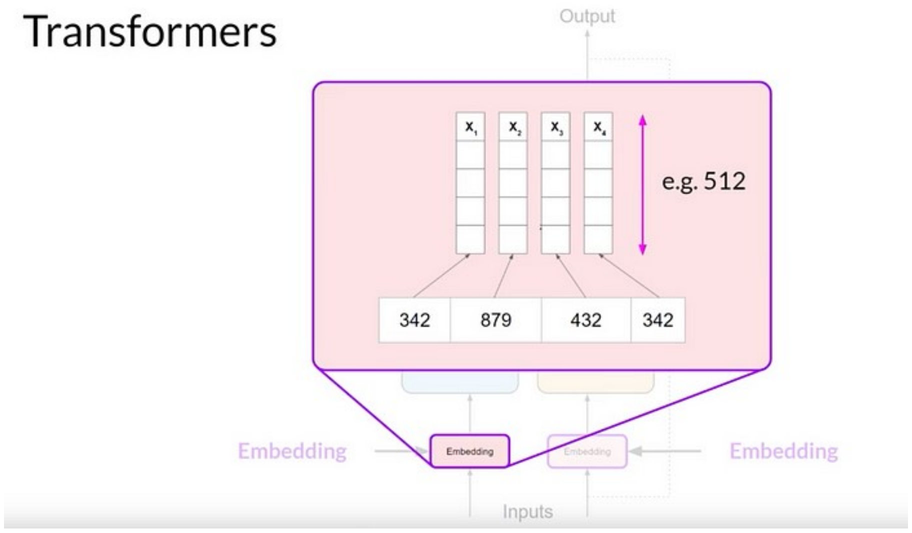
- Machine-learning models are just big statistical calculators that deal with numbers, not words. So, before feeding them text, we need to turn words into numbers. This is called tokenization. It's like giving each word a special number that the model understands from a big dictionary of words it knows. There are different ways to do this.

- Tokenization method Examples:

1. Token IDs matching two complete words.
2. Using token IDs to represent parts of words.

Embedding Layer

- The tokenized text is now passed to a high-dimensional space where each token is represented as a vector and occupies a unique location within that space. Each token ID in the vocabulary is matched to a multi-dimensional vector, and the intuition is that these vectors learn to encode the meaning and context of individual tokens in the input sequence.



Word Embeddings

Embeddings are numeric representations of words in a lower-dimensional space, capturing semantic and syntactic information. They play a vital role in **Natural Language Processing (NLP) tasks**. This explores traditional and neural approaches, such as TF-IDF, Word2Vec, and GloVe, offering insights into their advantages and disadvantages. Understanding the importance of pre-trained word embeddings, providing a comprehensive understanding of their applications in various NLP scenarios.

What is Word Embedding in NLP?

Word Embedding is an approach for representing words and documents. Word Embedding or Word Vector is a numeric vector input that represents a word in a lower-dimensional space. It allows words with similar meanings to have a similar representation.

Word Embeddings are a method of extracting features out of text so that we can input those features into a machine learning model to work with text data. They try to preserve syntactical and semantic information. The methods such as Bag of Words (BOW), CountVectorizer and TFIDF rely on the word count in a sentence but do not save any syntactical or semantic information. In these algorithms, the size of the vector is the number of elements in the vocabulary. We can get a sparse matrix if most of the elements are zero. Large input vectors will mean a huge number of weights which will result in high computation required for training. Word Embeddings give a solution to these problems.

Need for Word Embedding?

- To reduce dimensionality
- To use a word to predict the words around it.
- Inter-word semantics must be captured.

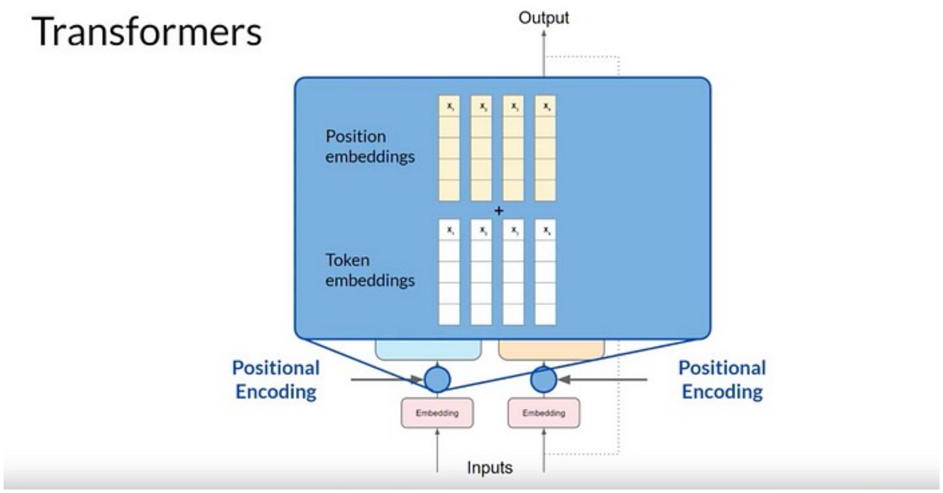
How are Word Embeddings used?

- They are used as input to machine learning models. Take the words —-> Give their numeric representation —-> Use in training or inference.

- To represent or visualize any underlying patterns of usage in the corpus that was used to train them.

Positional Encoding

- To maintain word order relevance, positional encoding is added to token vectors in both the encoder and decoder. This preserves information on word position, ensuring effective parallel processing of input tokens.

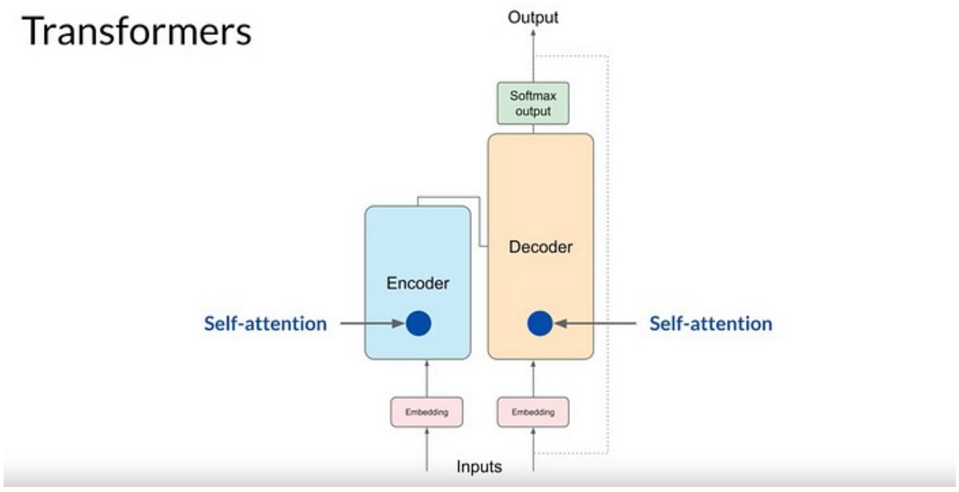


Attention Layer

- Once the input tokens are summed and the positional encodings, pass the resulting vectors to the self-attention layer. Here, the model analyzes the relationships between the tokens and input sequence.

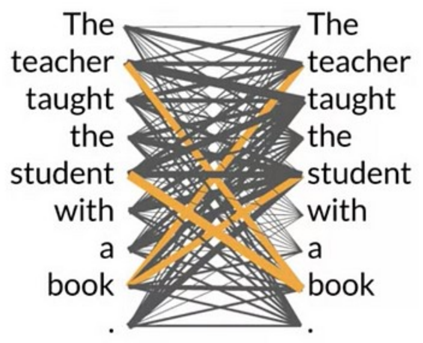
- In the transformer architecture, multiple sets of self-attention weights, called multi-headed self-attention, are learned independently in parallel. The number of attention heads varies between models but typically falls within the range of 12 to 100.

Transformers

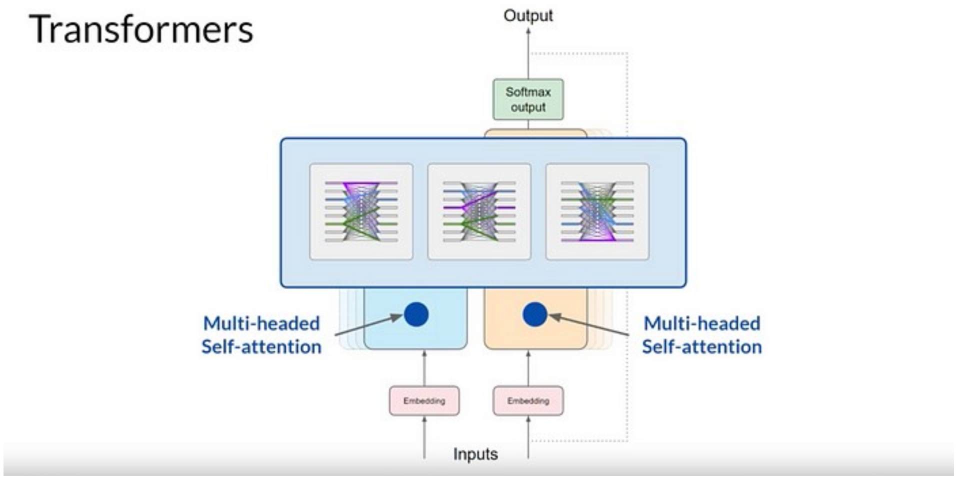


- Each self-attention head learns distinct language aspects, such as relationships between entities, sentence activities, or rhyming words. It's crucial to emphasize that we don't predefine what these heads focus on. Instead, they autonomously discover language nuances.

Self-attention



Transformers



Importance of Attention Mechanisms

The advent of attention mechanisms has been nothing short of revolutionary in the realm of deep learning. Attention allows models to dynamically focus on pertinent parts of the input data, akin to the way humans pay attention to certain aspects of a visual scene or conversation. This selective focus is particularly crucial in tasks where context is key, such as language understanding or image recognition.
In the context of transformers, attention mechanisms serve to weigh the influence of different input tokens when producing an output. This is not merely a replication of human attention but an enhancement, enabling machines to surpass human performance in certain tasks. Consider the following points that underscore the importance of attention mechanisms:

- They provide a means to handle variable-sized inputs by focusing on the most relevant parts.

- Attention-based models can capture long-range dependencies that earlier models like RNNs struggled with.

- They facilitate parallel processing of input data, leading to significant improvements in computational efficiency.

Types of Attention

Here are some types of attention mechanisms used in Transformer architecture:

1. Scaled Dot-Product Attention

- The Scaled Dot-Product Attention is the fundamental building block of the Transformer's attention mechanism. It involves three main components: queries (Q), keys (K), and values (V). The attention score is computed as the dot product of the query and key vectors, scaled by the square root of the dimension of the key vectors. This score is then passed through a softmax function to obtain the attention weights, which are used to compute a weighted sum of the value vectors.

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

where d_k is the dimension of the key vectors.

2. Multi-Head Attention

- Multi-Head Attention enhances the model's ability to focus on different parts of the input sequence simultaneously. It involves multiple attention heads, each with its own set of query, key, and value matrices. The outputs of these heads are concatenated and linearly transformed to produce the final output. This allows the model to capture different features and dependencies in the input sequence.

$$MultiHead(Q,K,V) = Concat(head_1, head_2, \ldots, head_h)W^O$$

where each $Attention(QW_i^Q, KW_i^K, VW_i^V)$ and W^O is the output projection matrix.

3. Self-Attention

- Self-Attention, also known as intra-attention, allows the model to consider different positions of the same sequence when computing the representation of a word. In the context of the Transformer, self-attention is applied in both the encoder and decoder layers. It enables the model to capture long-range dependencies and relationships within the input sequence.

4. Encoder-Decoder Attention

- Encoder-Decoder Attention, also known as cross-attention, is used in the decoder layers of the Transformer. It allows the decoder to focus on relevant parts of the input sequence (encoded by the encoder) when generating each word of the output sequence. This type of attention ensures that the decoder has access to the entire input sequence, helping it produce more accurate and contextually appropriate translations.

5. Causal or Masked Self-Attention

- Causal or Masked Self-Attention is used in the decoder to ensure that the prediction for a given position only depends on the known outputs at positions before it. This is crucial for tasks like language modeling, where future tokens should not be visible during training. The attention scores for future tokens are masked out, ensuring that the model cannot look ahead.

  $$MaskedAttention(Q, K, V) = softmax(\frac{QK^T + M}{\sqrt{d_k}})V$$

  where M is the mask matrix with $-\infty$ in positions that should be masked.

Feed forward Network

The Transformer model revolutionizes language processing with its unique architecture, which includes a crucial component known as the Feedforward Network (FFN). Positioned within both the encoder and decoder modules of the Transformer, the FFN plays a vital role in refining the data processed by the attention mechanisms.

The FFN within both the encoder and decoder of the Transformer is constructed as a fully connected, position-wise network. This design means that each position in the input sequence is processed separately but in the same manner, which is crucial for maintaining the positional integrity of the input data.

Key Characteristics of the FFN

1**. Fully Connected Layers**:

The FFN comprises two linear (fully connected) layers that transform the input data. The first layer expands the input dimension from model=512 to a larger dimension dff=2048, and the second layer projects it back to model.

2. **Activation Function**:

A Rectified Linear Unit (ReLU) activation function is applied between these two linear layers. This function is defined as ReLU(x)=max(0,x) and is used to introduce non-linearity into the model, helping it to learn more complex patterns.

3. **Position-wise Processing**:

Despite the sequential nature of the input data, each position (i.e., each word's representation in a sentence) is processed independently with the same FFN. This is akin to applying the same transformation across all positions, ensuring uniformity in extracting features from different parts of the input sequence.

Mathematical Representation

The operations within the FFN can be mathematically described by the following equations:

$\mathrm{FFN}(x)=\max(0,xW1+b1)W2+b2$

Where:

W1 and W2 are the weight matrices for the first and second linear layers, respectively.

b1 and b2 are the biases for these layers.

The ReLU activation is applied element-wise after the first linear transformation.

Example of FFN Processing

Consider a simplified example where the input x is a vector representing a single word's output from the post-LN stage:

$x=[0.5,-0.2,0.1,\ldots]x=[0.5,-0.2,0.1,\ldots]$ (512-dimensional)

The first layer of the FFN transforms this vector into a higher 2048-dimensional space, adds a bias, and applies the ReLU activation:

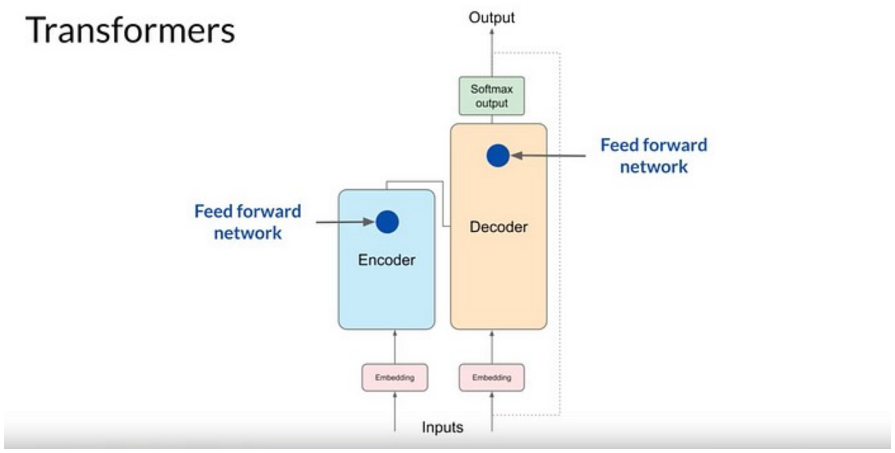
$x'=\max(0,xW1+b1)$

Assuming non-negative outputs from ReLU for simplicity, the second layer then projects this vector back down to the original 512-dimensional space:

FFN output$=x'W2+b2$

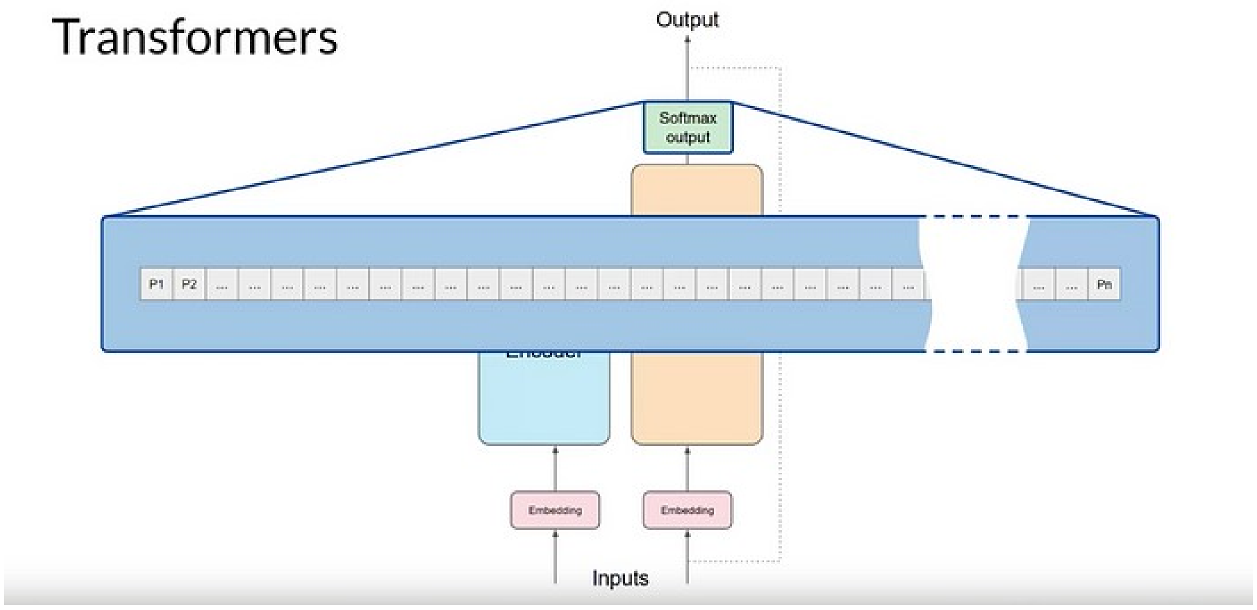
This output is then normalized by a subsequent post-LN step and either fed into the next layer of the encoder or used as part of the input to the multi-head attention layer in the decoder.\

- This layer produces logits, which represent the probability scores for all tokens in the tokenized dictionary.



Softmax layer

- To generate text, the model calculates probabilities for each word in the vocabulary using a softmax layer. This results in thousands of scores, with one token having the highest score, indicating the most likely prediction. Various methods can be applied to select the final output from this probability vector.
- Softmax is typically used only as the last layer in these networks in order to generate the final probabilities used in classification tasks, and thus represents only a small fraction of computation time and energy. However, this is no longer true for Transformer networks, which use softmax as a key component of the attention mechanism. For these networks, softmax can become a significant bottleneck, as shown in Figure 1. The softmax operation is inefficient in current hardware for two main reasons. First, softmax requires the use of the exponential function. Exponential functions tend to require large look-up table (LUTs) to compute the result through the use of Taylor expansions. This is particularly true for general-purpose hardware such as CPUs and GPGPUs, which cater to exponential computations with high accuracy requirements due to their use in various scientific computing applications. This large area and power overhead makes it difficult to instantiate a large number of these units. Second, in order to improve training stability, deep neural networks typically use a numerically stable softmax, which subtracts the max of the vector on which softmax is being performed in order to ensure that the result does not blow up to infinity. However, this stability comes at a cost, as calculating the max introduces an additional pass through the vector, incurring latency and memory overheads.



Model scalability - Parameters, Layers, and Performance

Scaling up the number of parameters in a transformer model can improve its performance and make it more capable of understanding and generating complex text. For example, GPT-3 has 175 billion parameters, while GPT-4 has over 1 trillion.

Using larger datasets

Using larger datasets to train a transformer model can improve its performance.

Allocating more computational resources

Using more computational resources to train a transformer model can improve its performance.

Using distillation

Knowledge from a larger model can be transferred to a smaller model using distillation. This can be useful because larger models are more expensive and slower to use.

Using Token Former

Token Former is an architecture that treats model parameters as tokens, allowing for efficient scaling without retraining from scratch.

The scale of a transformer model, which is determined by the number of parameters, the size of the dataset, and the computational resources used for training, has a greater impact on model loss than the model's architectural structure

Application: Time Series Data, Sequence Based Data, Text and Vision.

Transformers treat time series data as a sequence of values, with each value representing a time step. The model uses an encoder-decoder architecture, where the encoder takes in the time series history and the decoder predicts future values. The decoder uses an attention mechanism to learn which parts of the history are most useful for making predictions.

Benefits

Transformers can capture temporal patterns across different time scales, which can help provide a more comprehensive understanding of the data.

Examples

Some examples of transformer-based time series models include:

Autoformer: Uses a decomposition layer to capture seasonality and trend-cycle components, and an auto-correlation mechanism to improve performance.

Informer: Uses distillation to extract active data points and pass them to the next encoder layer, which can help reduce memory usage.

Applications

Transformers have been used in various aspects of time series analysis, such as time series forecasting

This allows transformers to capture complex temporal patterns and dependencies, potentially outperforming traditional methods that struggle with long-term dependencies and sequential processing.

Temporal Convolution vs. Self-Attention for Time Series Forecasting

In time series forecasting, two main approaches have been prevalent: temporal convolutional networks (TCNs) and recurrent neural networks (RNNs), such as LSTM and GRU. TCNs leverage convolutional layers to capture local patterns within the data, while RNNs process sequences recursively, retaining memory of past states.

Transformers, on the other hand, employ self-attention mechanisms to weigh the importance of each element in the sequence concerning all other elements. This allows transformers to capture both local and global dependencies simultaneously, making them particularly well-suited for time series forecasting tasks where long-range dependencies are crucial.

The self-attention mechanism in transformers enables them to capture temporal patterns across varying time scales, providing a more comprehensive understanding of the underlying data dynamics. Additionally, transformers are inherently parallelizable, leading to faster training times compared to sequential models like RNNs.

How Transformers Can Improve Time Series?

Using multi-head attention enabled by transformers could help improve the way time series models handle long-term dependencies, offering benefits over current approaches. To give you an idea of how well transformers work for long dependencies, think of the long and detailed responses that ChatGPT can generate in language-based models. Applying multi-head attention to time series could produce similar benefits by allowing one head to focus on long-term dependencies while another head focuses on short-term dependencies. We believe transformers could make it possible for time series models to predict as many as 1,000 data points into the future, if not more.

The Quadratic Complexity Issue

The way transformers calculate multi-head self-attention is problematic for time series. Because data points in a series must be multiplied by every other data point in the series, each data point you add to your input exponentially increases the time it takes to calculate attention. This is called quadratic complexity, and it creates a computational bottleneck when dealing with long sequences.

The Space time former Architecture

Space time former proposes a new way to represent inputs. Temporal attention models like Informer represent the value of multiple variables per time step in a single input token, which fails to consider spatial relationships between features. Graph attention models allow you to manually represent relationships between features but rely on hardcoded graphs that cannot change over time. Space time former combines both temporal and spatial attention methods, creating an input token to represent the value of a single feature at a given time. This helps the model understand more about the relationship between space, time, and value information.