TRANSFORMERS

The transformer model is a type of neural network architecture that excels at processing sequential data, most prominently associated with large language models (LLMs). Transformer models have also achieved elite performance in other fields of artificial intelligence (AI), such as computer vision, speech recognition and time series forecasting.

The transformer architecture was first described in the seminal 2017 paper "Attention is All You Need" by Vaswani and others, which is now considered a watershed moment in deep learning.

Originally introduced as an evolution of the recurrent neural network (RNN)-based sequence-to-sequence models used for machine translation, transformer-based models have since attained cutting-edge advancements across nearly every machine learning (ML) discipline.

Despite their versatility, transformer models are still most commonly discussed in the context of natural language processing (NLP) use cases, such as chatbots, text generation, summarization, question answering and sentiment analysis.

BERT (or Bidirectional Encoder Representations from Transformers), an encoder-only model introduced by Google in 2019, was a major landmark in the establishment of transformers and remains the basis of most modern word embedding applications, from modern vector databases to Google search.

Autoregressive decoder-only LLMs, such as the GPT-3 (short for Generative Pre-trained Transformer) model that powered the launch of OpenAI's ChatGPT, catalyzed the modern era of generative AI (gen AI).

The ability of transformer models to intricately discern how each part of a data sequence influences and correlates with the others also lends them many multimodal uses.

For instance, vision transformers (ViTs) often exceed the performance of convolutional neural networks (CNNs) on image segmentation, object detection and related tasks. The transformer architecture also powers many diffusion models used for image generation, multimodal text-to-speech (TTS) and vision language models (VLMs).

Why are transformer models important?

The central feature of transformer models is their self-attention mechanism, from which transformer models derive their impressive ability to detect the relationships (or dependencies) between each part of an input sequence. Unlike the RNN and CNN architectures that preceded it, the transformer architecture uses only attention layers and standard feedforward layers.

The benefits of self-attention, and specifically the multi-head attention technique that transformer models employ to compute it, are what enable transformers to exceed the performance of the RNNs and CNNs that had previously been state-of-the-art.

Before the introduction of transformer models, most NLP tasks relied on recurrent neural networks (RNNs). The way RNNs process sequential data is inherently *serialized*: they ingest the elements of an input sequence one at a time and in a specific order.

This hinders the ability of RNNs to capture long-range dependencies, meaning RNNs can only process short text sequences effectively.

This deficiency was somewhat addressed by the introduction of long short term memory networks (LSTMs), but remains a fundamental shortcoming of RNNs.

Attention mechanisms, conversely, can examine an entire sequence simultaneously and make decisions about how and when to focus on specific time steps of that sequence.

In addition to significantly improving the ability to understand long-range dependencies, this quality of transformers also allows for *parallelization*: the ability to perform many computational steps at once, rather than in a serialized manner.

Being well-suited to parallelism enables transformer models to take full advantage of the power and speed offered by GPUs during both training and inference. This possibility, in turn, unlocked the opportunity to train transformer models on unprecedentedly massive datasets through self-supervised learning.

Especially for visual data, transformers also offer some advantages over convolutional neural networks. CNNs are inherently local, using convolutions to process smaller subsets of input data one piece at a time.

Therefore, CNNs also struggle to discern long-range dependencies, such as correlations between words (in text) or pixels (in images) that aren't neighboring one another. Attention mechanisms don't have this limitation.

What is self-attention?

Understanding the mathematical concept of attention, and more specifically self-attention, is essential to understanding the success of transformer models in so many fields. Attention mechanisms are, in essence, algorithms designed to determine which parts of a data sequence an AI model should "pay attention to" at any particular moment.

Consider a language model interpreting the English text " *on Friday, the judge issued a sentence*. "

- The preceding word " the " suggests that " judge " is acting as a noun—as in, a *person presiding over a legal trial*—rather than a verb meaning *to appraise or form an opinion*.
- That context for the word " judge " suggests that " sentence " probably refers to a *legal penalty*, rather than a grammatical "sentence."
- The word " issued " further implies that " sentence " refers to the legal concept, not the grammatical concept.
- Therefore, when interpreting the word " sentence ," the model should *pay close attention to* " judge " and " issued. " It should also pay some attention to the word " the ." It can more or less ignore the other words.

How does self-attention work?

Broadly speaking, a transformer model's attention layers assess and use the specific context of each part of a data sequence in 4 steps:

1. The model "reads" raw data sequences and converts them into vector embeddings, in which each element in the sequence is represented by its own feature vector(s) that numerically reflect qualities such as semantic meaning.

2. The model determines similarities, correlations and other dependencies (or lack thereof) between each vector and each other vector. In most transformer models, the relative importance of one vector to another is determined by computing the *dot product* between each vector. If the vectors are well aligned, multiplying them together will yield a large value. If they're not aligned, their dot product will be small or negative.

3. These "alignment scores" are converted into *attention weights.* This is achieved by using alignment scores as inputs to a *softmax* activation function, which normalizes all values to a range between 0–1 such that they all add up to a total of 1. So for instance, assigning an attention weight of 0 between "Vector A" and "Vector B" means that Vector B should be ignored when making predictions about Vector A. Assigning Vector B an attention weight of 1 means that it should receive 100% of the model's attention when making decisions about Vector A.

4. These attention weights are used to emphasize or deemphasize the influence of specific input elements at specific times. In other words, attention weights help transformer models focus on or ignore specific information at a specific moment.

Before training, a transformer model doesn't yet "know" how to generate optimal vector embeddings and alignment scores. During training, the model makes predictions across millions of examples drawn from its training data, and a loss function quantifies the error of each prediction.

Through an iterative cycle of making predictions and then updating model weights through backpropagation and gradient descent, the model "learns" to generate vector embeddings, alignment scores and attention weights that lead to accurate outputs.

How do transformer models work?

Transformer models such as relational databases generate *query, key* and *value vectors* for each part of a data sequence, and use them to compute attention weights through a series of matrix multiplications.

Relational databases are designed to simplify the storage and retrieval of relevant data: they assign a unique identifier ("key") to each piece of data, and each *key* is associated with a corresponding *value.* The "Attention is All You Need" paper applied that conceptual framework to processing the relationships between each token in a sequence of text.

- The *query vector* represents the information a specific token is "seeking." In other words, a token's query vector is used to compute how other tokens might influence its meaning, conjugation or connotations in context.
- The *key vectors* represent the information that each token contains. Alignment between query and key is used to compute attention weights that reflect how relevant they are in the context of that text sequence.
- The *value* (or *value vector*) "returns" the information from each key vector, scaled by its respective attention weight. Contributions from keys that are strongly aligned with a query are weighted more heavily; contributions from keys that are not relevant to a query will be weighted closer to zero.

For an LLM, the model's "database" is the vocabulary of tokens it has learned from the text samples in its training data. Its attention mechanism uses information from this "database" to understand the context of language.

Tokenization and input embeddings

Whereas *characters*—letters, numbers or punctuation marks—are the base unit we humans use to represent language, the smallest unit of language that AI models use is a *token*. Each token is assigned an ID number, and these ID numbers (rather than the words or even the tokens themselves) are the way LLMs navigate their vocabulary "database." This *tokenization* of language significantly reduces the computational power needed to process text.

To generate query and key vectors to feed into the transformer's attention layers, the model needs an initial, contextless vector embedding for each token. These initial token embeddings can be either learned during training or taken from a pretrained word embedding model.

Positional encoding

The order and position of words can significantly impact their semantic meanings. Whereas the serialized nature of RNNs inherently preserves information about the position of each token, transformer models must explicitly *add* positional information for the attention mechanism to consider.

With *positional encoding,* the model adds a vector of values to each token's embedding, derived from its relative position, before the input enters the attention mechanism. The nearer the 2 tokens are, the more similar their positional vectors will be and therefore, the more their alignment score will increase from adding positional information. The model thereby learns to pay greater attention to nearby tokens.

Generating query, key and value vectors

When positional information has been added, each updated token embedding is used to generate three new vectors. These *query, key* and *value* vectors are generated by passing the original token embeddings through each of three parallel feedforward neural network layers that precede the first attention layer. Each parallel subset of that linear layer has a unique matrix of weights, learned through self-supervised pretraining on a massive dataset of text.

- The embeddings are multiplied by the weight matrix W_Q to yield the *query vectors* (Q), which have d_k dimensions
- The embeddings are multiplied by the weight matrix W_K to yield the key vector (K), also with dimensions d_k
- The embeddings are multiplied by the weight matrix W_V to yield the value vectors (V), with dimensions d_v

Computing self-attention

The transformer's attention mechanism's primary function is to assign accurate attention weights to the pairings of each token's query vector with the key vectors of all the other tokens in the sequence. When achieved, you can think of each token x as now having a corresponding vector of attention weights, in which each element of that vector represents the extent to which some other token should influence it.

- Each other token's *value vector* is now multiplied by its respective attention weight.
- These attention-weighted value vectors are all summed together. The resulting vector represents the aggregated contextual information being provided to token x by all of the other tokens in the sequence.
- Finally, the resulting vector of attention-weighted changes from each token is added to the token x's original, post-positional encoding vector embedding.

In essence, x's vector embedding has been updated to better reflect the context provided by the other tokens in the sequence.

Multi-head attention

To capture the many multifaceted ways tokens might relate to one another, transformer models implement multi-head attention across multiple attention blocks.

Before being fed into the first feedforward layer, each original input token embedding is split into *h* evenly sized subsets. Each piece of the embedding is fed into one of *h* parallel matrices of *Q, K* and *V* weights, each of which are called a *query* head, *key head* or *value head*. The vectors output by each of these parallel triplets of query, key and value heads are then fed into a corresponding subset of the next attention layer, called an *attention head.*

In the final layers of each attention block, the outputs of these *h* parallel circuits are eventually concatenated back together before being sent to the next feedforward layer. In practice, model training results in each circuit learning different weights that capture a separate aspect of semantic meanings.

Residual connections and layer normalization

In some situations, passing along the contextually-updated embedding output by the attention block might result in an unacceptable loss of information from the original sequence.

To address this, transformer models often balance the contextual information provided by the attention mechanism with the original semantic meaning of each token. After the attention-updated subsets of the token embedding have all been concatenated back together, the updated vector is then added to the token's original (position-encoded) vector embedding. The original token embedding is supplied by a *residual connection* between that layer and an earlier layer of the network.

The resulting vector is fed into another linear feedforward layer, where it's normalized back to a constant size before being passed along to the next attention block. Together, these measures help preserve stability in training and help ensure that the text's original meaning is not lost as the data moves deeper into the neural network.

Generating outputs

Eventually, the model has enough contextual information to inform its final outputs. The nature and function of the output layer will depend on the specific task the transformer model has been designed for.

In autoregressive LLMs, the final layer uses a softmax function to determine the probability that the next word will match each token in its vocabulary "database." Depending on the specific sampling

hyperparameters, the model uses those probabilities to determine the next token of the output sequence.

Transformer models in natural language processing (NLP)

Transformer models are most commonly associated with NLP, having originally been developed for machine translation use cases. Most notably, the transformer architecture gave rise to the large language models (LLMs) that catalyzed the advent of generative AI.

Most of the LLMs that the public is most familiar with, from closed source models such as OpenAI's GPT series and Anthropic's Claude models to open source models including Meta Llama or IBM® Granite®, are autoregressive decoder-only *LLMs.*

Autoregressive LLMs are designed for text generation, which also extends naturally to adjacent tasks such as summarization and question answering. They're trained through self-supervised learning, in which the model is provided the first word of a text passage and tasked with iteratively predicting the next word until the end of the sequence.

Information provided by the self-attention mechanism enables the model to extract context from the input sequence and maintain the coherence and continuity of its output.

Encoder-decoder masked language models (MLMs), such as BERT and its many derivatives, represent the other main evolutionary branch of transformer-based LLMs. In training, an MLM is provided a text sample with some tokens *masked*—hidden—and tasked with completing the missing information.

While this training methodology is less effective for text generation, it helps MLMs excel at tasks requiring robust contextual information, such as translation, text classification and learning embeddings.

Transformer models in other fields

Though transformer models were originally designed for, and continue to be most prominently associated with natural language use cases, they can be used in nearly any situation involving sequential data. This has led to the development of transformer-based models in other fields, from fine-tuning LLMs into multimodal systems to dedicated time series forecasting models and ViTs for computer vision.

Some data modalities are more naturally suited to transformer-friendly sequential representation than others. Time series, audio and video data are inherently sequential, whereas image data is not. Despite this, ViTs and other attention-based models have achieved state-of-the-art results for many computer vision tasks, including image captioning, object detection, image segmentation and visual question answering.

To use transformer models for data not conventionally thought of as "sequential" requires a conceptual workaround to represent that data as a sequence. For instance, to use attention mechanisms to understand visual data, ViTs use *patch embeddings* to make image data interpretable as sequences.

- First, an image is split into an array of patches. For instance, a 224x224 pixel image can be subdivided into 256 14x14 pixel patches, dramatically reducing the number of computational steps required to process the image.

- Next, a linear projection layer maps each patch to a vector embedding.
- Positional information is added to each of these patch embeddings, akin to the positional encoding described earlier in this article.
- These patch embeddings can now essentially function as a sequence of token embeddings, allowing the image to be interpreted by an attention mechanism.