Transformer (deep learning architecture)

A **transformer** is a deep learning architecture developed by researchers at Google and based on the multi-head attention mechanism, proposed in a 2017 paper "Attention Is All You Need".[1] Text is converted to numerical representations called tokens, and each token is converted into a vector via looking up from a word embedding table.[1] At each layer, each token is then contextualized within the scope of the context window with other (unmasked) tokens via a parallel multi-head attention mechanism allowing the signal for key tokens to be amplified and less important tokens to be diminished.

A standard Transformer

Transformers have the advantage of having no recurrent units, and therefore require less training time than earlier recurrent neural architectures (RNNs) such as long short-term memory (LSTM).[2] Later variations have been widely adopted for training large language models (LLM) on large (language) datasets, such as the Wikipedia corpus and Common Crawl.[3]

Transformers were first developed as an improvement over previous architectures for translation,[4][5] but has found many applications since then. They are used in lar natural language processing, computer vision (vision transformers), reinfo learning,[6][7] audio,[8] multi-modal processing, robotics,[9] and even playing chess.[1 also led to the development of pre-trained systems, such as generative pre-trained trans (GPTs)[11] and BERT[12] (Bidirectional Encoder Representations from Transformers).

In computer science, **contextualization** is the process of identifying the data relevant to an entity based on the entity's contextual information.

History

Predecessors

For many years, sequence modelling and generation was done by using plain recurrent neural networks (RNNs). A well-cited early example was the Elman network (1990). In theory, the information from one token can propagate arbitrarily far down the sequence, but in practice the vanishing-gradient problem leaves the model's state at the end of a long sentence without precise, extractable information about preceding tokens.

A key breakthrough was LSTM (1995),[note 1] a RNN which used various innovations to overcome the vanishing gradient problem, allowing efficient learning of long-sequence modelling. One key innovation was the use of an attention mechanism which used neurons that multiply the outputs of other neurons, so-called *multiplicative units*.[13] Neural networks using multiplicative units were called *sigma-pi networks*[14] or *higher-order networks*,[15] but they faced high computational complexity.[2] LSTM became the standard architecture for long sequence modelling until the 2017 publication of Transformers.

However, LSTM still used sequential processing, like most other RNNs.[note 2] Specifically, RNNs operate one token at a time from first to last; they cannot operate in parallel over all tokens in a sequence. An early attempt to overcome this was the fast weight controller (1992) which computed the weight matrix for further processing depending on the input.[16] It used the fast weights architecture (1987),[17] where one neural network outputs the weights of another neural network. It was later shown to be equivalent to the linear Transformer without normalization.[18][19]

Attention with seq2seq

The idea of encoder-decoder sequence transduction had been developed in the early 2010s (see [20][21] for previous papers). The papers most commonly cited as the originators that produced seq2seq are two concurrently published papers from 2014.[20][21]

(Sutskever et al, 2014) [21] was a 380M-parameter model for machine translation using two long short-term memory (LSTM). The architecture consists of two parts. The *encoder* is an LSTM that takes in a sequence of tokens and turns it into a vector. The *decoder* is another LSTM that converts the vector into a sequence of tokens. Similarly, (Cho et al, 2014)[20] was 130M-parameter model that used gated recurrent units (GRU) instead of LSTM. Later research showed that GRUs are neither better nor worse than LSTMs for seq2seq.[22][23]

These early seq2seq models had no attention mechanism, and the state vector is accessible only after the *last* word of the source text was processed. Although in theory such a vector retains the information about the whole original sentence, in practice the information is poorly preserved, since the input is processed sequentially by one recurrent network into a *fixed*-size output vector, which was then processed by another recurrent network into an output. If the input is long, then the output vector would not be able to contain all relevant information, and the output quality degrades. As evidence, reversing the input sentence improved seq2seq translation.[24]

(Bahdanau et al, 2014)[4] introduced an attention mechanism to seq2seq for machine translation to solve the bottleneck problem, allowing the model to process long-distance dependencies more easily. They called their model *RNNsearch*, as it "emulates searching through a source sentence during decoding a translation".

(Luong et al, 2015)[25] compared the relative performance of global (that of (Bahdanau et al, 2014)) and local (sliding window) attention model architectures for machine translation, and found that a mixed attention architecture had higher quality than global attention, while the use of a local attention architecture reduced translation time.

In 2016, Google Translate was revamped to Google Neural Machine Translation, which replaced the previous model based on statistical machine translation. The new model was a seq2seq model where the encoder and the decoder were both 8 layers of bidirectional LSTM.[26] It took nine months to develop, and it achieved a higher level of performance than the statistical approach, which took ten years to develop.[27]

Attention

Seq2seq models with attention still suffered from the same issue with recurrent networks, which is that they are hard to parallelize, which prevented them to be accelerated on GPUs. In 2016, *decomposable attention* applied attention mechanism to the feedforward network, which are easy to parallelize.[28] One of its authors, Jakob Uszkoreit, suspected that attention *without* recurrence is sufficient for language translation, thus the title "attention is *all* you need".[29]

In 2017, the original (100M-sized) encoder-decoder transformer model was proposed in the "Attention is all you need" paper. At the time, the focus of the research was on improving seq2seq for machine translation, by removing its recurrence to process all tokens in parallel, but preserving its dot-product attention mechanism to keep its text processing performance.[1] Its parallelizability was an important factor to its widespread use in large neural networks.[30]

AI boom era

Transformers are used in many models that contribute to the ongoing AI boom.

In language modelling, ELMo (2018) was a bi-directional LSTM that produces contextualized word embeddings, improving upon the line of research from bag of words and word2vec. It was followed by BERT (2018), an encoder-only Transformer model.[31] In 2019 October, Google started using BERT to process search queries.[32] In 2020, Google Translate replaced the previous RNN-encoder–RNN-decoder model by a Transformer-encoder–RNN-decoder model.[33]

Starting in 2018, the OpenAI GPT series of decoder-only Transformers became state of the art in natural language generation. In 2022, a chatbot based on GPT-3, ChatGPT, became unexpectedly popular,[34] triggering a boom around large language models.[35][36]

Since 2020, Transformers have been applied in modalities beyond text, including the vision transformer,[37] speech recognition,[38] robotics,[6] and multimodal.[39] The vision transformer, in turn, stimulated new developments in convolutional neural networks.[40] Image and video generators like DALL-E (2021), Stable Diffusion 3 (2024),[41] and Sora (2024), are based on the Transformer architecture.

Training

Methods for stabilizing training

The plain transformer architecture had difficulty converging. In the original paper[1] the authors recommended using learning rate warmup. That is, the learning rate should linearly scale up from 0 to maximal value for the first part of the training (usually recommended to be 2% of the total number of training steps), before decaying again.

A 2020 paper found that using layer normalization *before* (instead of after) multiheaded attention and feedforward layers stabilizes training, not requiring learning rate warmup.[42]

Pretrain-finetune

Transformers typically are first pretrained by self-supervised learning on a large generic dataset, followed by supervised fine-tuning on a small task-specific dataset. The pretrain dataset is typically an unlabeled large corpus, such as The Pile. Tasks for pretraining and fine-tuning commonly include:

- language modeling[12]
- next-sentence prediction[12]
- question answering[3]
- reading comprehension

- sentiment analysis[1]
- paraphrasing[1]

The T5 transformer report[43] documents a large number of natural language pretraining tasks. Some examples are:

- restoring or repairing incomplete or corrupted text. For example, the input, *"Thank you~~me to your party~~week",* might generate the output, *"Thank you **for inviting** me to your party **last** week".*
- translation between natural languages (machine translation)
- judging the pragmatic acceptability of natural language. For example, the following sentence might be judged "not acceptable",[44] because even though it is syntactically well-formed, it is improbable in ordinary human usage: *The course is jumping well.*

Note that while each of these tasks is trivial or obvious for human native speakers of the language (or languages), they have typically proved challenging for previous generations of machine learning architecture.

Tasks

In general, there are 3 classes of language modelling tasks: "masked",[45] "autoregressive",[46] and "prefixLM".[47] These classes are independent of a specific modeling architecture such as Transformer, but they are often discussed in the context of Transformer.

In a masked task,[45] one or more of the tokens is masked out, and the model would produce a probability distribution predicting what the masked-out tokens are based on the context. The loss function for the task is typically sum of log-perplexities for the masked-out tokens:

$$\text{Loss} = -\sum_{t \in \text{masked tokens}} \ln(\text{probability of } t \text{ conditional on its context})$$

and the model is trained to minimize this loss function. The BERT series of models are trained for masked token prediction and another task.

In an autoregressive task,[46] the entire sequence is masked at first, and the model produces a probability distribution for the first token. Then the first token is revealed and the model predicts the second token, and so on. The loss function for the task is still typically the same. The GPT series of models are trained by autoregressive tasks.

In a prefixLM task,[47] the sequence is divided into two parts. The first part is presented as context, and the model predicts the first token of the second part. Then that would be revealed, and the model predicts the second token, and so on. The loss function for the task is still typically the same. The T5 series of models are trained by prefixLM tasks.

Note that "masked" as in "masked language modelling" is not "masked" as in "masked attention", and "prefixLM" (prefix language modeling) is not "prefixLM" (prefix language model).

Architecture

All transformers have the same primary components:

- Tokenizers, which convert text into tokens.
- Embedding layer, which converts tokens and positions of the tokens into vector representations.
- Transformer layers, which carry out repeated transformations on the vector representations, extracting more and more linguistic information. These consist of alternating attention and feedforward layers. There are two major types of transformer layers: encoder layers and decoder layers, with further variants.
- Un-embedding layer, which converts the final vector representations back to a probability distribution over the tokens.

The following description follows exactly the Transformer as described in the original paper. There are variants, described in the following section.

By convention, we write all vectors as row vectors. This, for example, means that pushing a vector through a linear layer means multiplying it by a weight matrix on the right, as $\boldsymbol{xW}$.

Tokenization

As the Transformer architecture natively processes numerical data, not text, there must be a translation between text and tokens. A token is an integer that represents a character, or a short segment of characters. On the input side, the input text is parsed into a token sequence. Similarly, on the output side, the output tokens are parsed back to text. The module doing the conversion between token sequences and texts is a tokenizer.

The set of all tokens is the vocabulary of the tokenizer, and its size is the *vocabulary size* $n_{\text{vocabulary}}$. When faced with tokens outside the vocabulary, typically a special token is used, written as "[UNK]" for "unknown".

Some commonly used tokenizers are byte pair encoding, WordPiece, and SentencePiece.

Embedding

Each token is converted into an embedding vector via a lookup table. Equivalently stated, it multiplies a one-hot representation of the token by an embedding matrix M. For example, if the input token is 3, then the one-hot representation is $[0, 0, 0, 1, 0, 0, \ldots]$, and its embedding vector is

$$\text{Embed}(3) = [0, 0, 0, 1, 0, 0, \ldots]M$$

The token embedding vectors are added to their respective positional encoding vectors (see below), producing the sequence of input vectors.

The number of dimensions in an embedding vector is called *hidden size* or *embedding size* and written as d_{emb}.[48]

Un-embedding

An un-embedding layer is almost the reverse of an embedding layer. Whereas an embedding layer converts a token into a vector, an un-embedding layer converts a vector into a probability distribution over tokens.

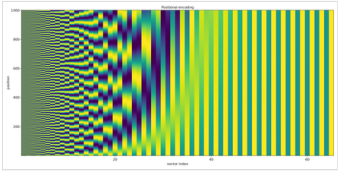
The un-embedding layer is a linear-softmax layer:

$$\text{UnEmbed}(x) = \text{softmax}(xW + b)$$

The matrix has shape $(d_{\text{emb}}, n_{\text{vocabulary}})$.

Positional encoding

A positional encoding is a fixed-size vector representation of the relative positions of tokens within a sequence: it provides the transformer model with information about *where* the words are in the input sequence. Without positional encoding, the model would be unable to process input sequence as more than a bag of words, as for example, both "man bites dog" and "dog bites man" would be processed exactly the same way.



A diagram of a sinusoidal positional encoding with parameters $N = 10000, d = 100$

The positional encoding is defined as a function of type $f : \mathbb{R} \to \mathbb{R}^d; d \in \mathbb{Z}, d > 0$, where d is a positive even integer. The full positional encoding defined in the original paper[1] is:

$$(f(t)_{2k}, f(t)_{2k+1}) = (\sin(\theta), \cos(\theta)) \quad \forall k \in \{0, 1, \ldots, d/2 - 1\}$$

where $\theta = \frac{t}{r^k}, r = N^{2/d}$.

Here, N is a free parameter that should be significantly larger than the biggest k that would be input into the positional encoding function. The original paper uses $N = 10000$.

The function is in a simpler form when written as a complex function of type $f : \mathbb{R} \to \mathbb{C}^{d/2}$

$$f(t) = \left(e^{it/r^k}\right)_{k=0,1,\ldots,\frac{d}{2}-1}$$

where $r = N^{2/d}$.

The main reason for using this positional encoding function is that using it, shifts are linear transformations:

$$f(t + \Delta t) = \text{diag}(f(\Delta t))f(t)$$

where $\Delta t \in \mathbb{R}$ is the distance one wishes to shift. This allows the transformer to take any encoded position, and find the encoding of the position n-steps-ahead or n-steps-behind, by a matrix multiplication.

By taking a linear sum, any convolution can also be implemented as linear transformations:

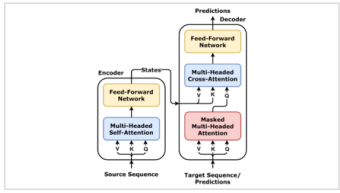
$$\sum_j c_j f(t + \Delta t_j) = \left(\sum_j c_j \,\text{diag}(f(\Delta t_j))\right) f(t)$$

for any constants c_j. This allows the transformer to take any encoded position and find a linear sum of the encoded locations of its neighbors. This sum of encoded positions, when fed into the attention mechanism, would create attention weights on its neighbors, much like what happens in a convolutional neural network language model. In the author's words, "we hypothesized it would allow the model to easily learn to attend by relative position."

In typical implementations, all operations are done over the real numbers, not the complex numbers, but since complex multiplication can be implemented as real 2-by-2 matrix multiplication, this is a mere notational difference.

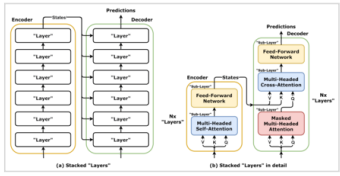
Encoder-decoder (overview)

Like earlier seq2seq models, the original transformer model used an **encoder-decoder** architecture. The encoder consists of encoding layers that process all the input tokens together one layer after another, while the decoder consists of decoding layers that iteratively process the encoder's output and the decoder's output tokens so far.



One encoder-decoder block.

The purpose of each encoder layer is to create contextualized representations of the tokens, where each representation corresponds to a token that "mixes" information from other input tokens via self-attention mechanism. Each decoder layer contains two attention sublayers: (1) cross-attention for incorporating the output of encoder (contextualized input token representations), and (2) self-attention for "mixing" information among the input tokens to the decoder (i.e. the tokens generated so far during inference time).[49][50]


A Transformer is composed of stacked encoder layers and decoder layers.

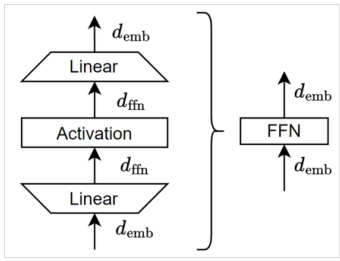
Both the encoder and decoder layers have a feed-forward neural network for additional processing of their outputs and contain residual connections and layer normalization steps.[50] These feed-forward layers contain most of the parameters in a Transformer model.

Feedforward network

The feedforward network (FFN) modules in a Transformer are 2-layered multilayer perceptrons:

$$\mathbf{FFN}(x) = \phi(xW^{(1)} + b^{(1)})W^{(2)} + b^{(2)}$$

where ϕ is its activation function. The original Transformer used ReLU activation.



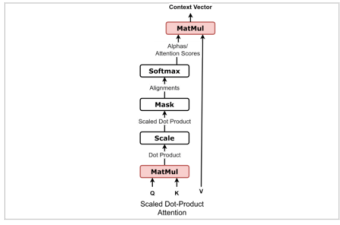
The feedforward network module. It is a two-layered network that maps d_{emb}-dimensional vectors into d_{emb}-dimensional vectors.

The number of neurons in the middle layer is called *intermediate size* (GPT),[51] *filter size* (BERT),[48] or *feedforward size* (BERT).[48] It is typically larger than the embedding size. For example, in both GPT-2 series and BERT series, the intermediate size of a model is 4 times its embedding size: $d_{\text{ffn}} = 4d_{\text{emb}}$.

Scaled dot-product attention

Attention head

The attention mechanism used in the Transformer architecture are scaled dot-product attention units. For each unit, the transformer model learns three weight matrices: the query weights W^Q, the key weights W^K, and the value weights W^V.



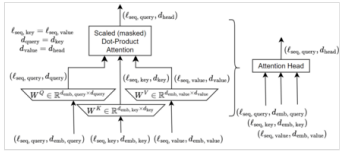
Scaled dot-product attention, block diagram.

The module takes three sequences, a query sequence, a key sequence, and a value sequence. The query sequence is a sequence of length $\ell_{\text{seq, query}}$, and each entry is a vector of dimension $d_{\text{emb, query}}$. Similarly for the key and value sequences.

For each vector $x_{i,\text{query}}$ in the query sequence, it is multiplied by a matrix W^Q to produce a query vector $q_i = x_{i,\text{query}}W^Q$. The matrix of all query vectors is the query matrix:

$$Q = X_{\text{query}}W^Q$$

Similarly, we construct the key matrix $K = X_{\text{key}}W^K$ and the value matrix $V = X_{\text{value}}W^V$.


Exact dimension counts within an attention head module.

It is usually the case that all W^Q, W^K, W^V are square matrices, meaning $d_{\text{emb, query}} = d_{\text{query}}$, etc.

Attention weights are calculated using the query and key vectors: the attention weight a_{ij} from token i to token j is the dot product between q_i and k_j. The attention weights are divided by the square root of the dimension of the key vectors, $\sqrt{d_k}$, which stabilizes gradients during training, and passed through a softmax which normalizes the weights. The fact that W^Q and W^K are different matrices allows attention to be non-symmetric: if token i attends to token j (i.e. $q_i \cdot k_j$ is large), this does not necessarily mean that token j will attend to token i (i.e. $q_j \cdot k_i$ could be small). The output of the attention unit for token i is the weighted sum of the value vectors of all tokens, weighted by a_{ij}, the attention from token i to each token.

The attention calculation for all tokens can be expressed as one large matrix calculation using the softmax function, which is useful for training due to computational matrix operation optimizations that quickly compute matrix operations. The matrices $\boldsymbol{Q}$, $\boldsymbol{K}$ and $\boldsymbol{V}$ are defined as the matrices where the $\boldsymbol{i}$th rows are vectors $\boldsymbol{q_i}$, $\boldsymbol{k_i}$, and $\boldsymbol{v_i}$ respectively. Then we can represent the attention as

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

where the softmax is applied over each of the rows of the matrix.

The number of dimensions in a query vector is *query size* $\boldsymbol{d_{\text{query}}}$ and similarly for the *key size* $\boldsymbol{d_{\text{key}}}$ and *value size* $\boldsymbol{d_{\text{value}}}$. The output dimension of an attention head is its *head dimension* $\boldsymbol{d_{\text{head}}}$. The attention mechanism requires the following three equalities to hold:

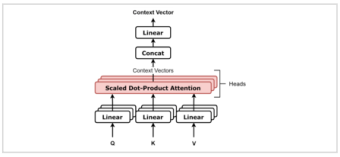
$$\boldsymbol{\ell_{\text{seq, key}} = \ell_{\text{seq, value}},\ d_{\text{query}} = d_{\text{key}},\ d_{\text{value}} = d_{\text{head}}}$$

but is otherwise unconstrained.

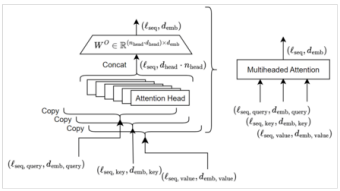
If the attention head is used in a self-attention fashion, then $\boldsymbol{X_{\text{query}} = X_{\text{key}} = X_{\text{value}}}$. If the attention head is used in a cross-attention fashion, then usually $\boldsymbol{X_{\text{query}} \neq X_{\text{key}} = X_{\text{value}}}$. It is theoretically for all three to be different, but it is rarely used in practice.

Multiheaded attention

One set of $\boldsymbol{(W^Q, W^K, W^V)}$ matrices is called an *attention head*, and each layer in a transformer model has multiple attention heads. While each attention head attends to the tokens that are relevant to each token, multiple attention heads allow the model to do this for different definitions of "relevance". In addition, the influence field representing relevance can become progressively dilated in successive layers. Many transformer attention heads encode relevance relations that are meaningful to humans. For example, some attention heads can attend mostly to the next word, while others mainly attend from verbs to their direct objects.[52] The computations for each attention head can be performed in parallel, which allows for fast processing. The outputs for the attention layer are concatenated to pass into the feed-forward neural network layers.



Multiheaded attention, block diagram.



Exact dimension counts within a multiheaded attention module.

Concretely, let the multiple attention heads be indexed by $\boldsymbol{i}$, then we have

$$\mathbf{MultiheadedAttention}(Q, K, V) = \mathbf{Concat}_{i \in [n_{\text{heads}}]}(\mathbf{Attention}(XW_i^Q, XW_i^K, XW_i^V))W^O$$

where the matrix $\boldsymbol{X}$ is the concatenation of word embeddings, and the matrices $\boldsymbol{W_i^Q, W_i^K, W_i^V}$ are "projection matrices" owned by individual attention head $\boldsymbol{i}$, and $\boldsymbol{W^O}$ is a final projection matrix owned by the whole multi-headed attention head.

It is theoretically possible for each attention head to have a different head dimension $\boldsymbol{d_{\text{head}}}$, but it is rarely used in practice.

As an example, in the smallest GPT-2 model, there are only self-attention mechanisms. It has the following dimensions:

$$\boldsymbol{d_{\text{emb}} = 768, n_{\text{head}} = 12, d_{\text{head}} = 64}$$

Since $\boldsymbol{12 \times 64 = 768}$, its projection matrix $\boldsymbol{W^O \in \mathbb{R}^{(64\times 12)\times 768}}$ is a square matrix.

Masked attention

It may be necessary to cut out attention links between some word-pairs. For example, the decoder, when decoding for the token position $\boldsymbol{t}$, should not have access to the token at position $\boldsymbol{t+1}$. This may be accomplished before the softmax stage by adding a mask matrix $\boldsymbol{M}$ that is $\boldsymbol{-\infty}$ at entries where the attention link must be cut, and $\boldsymbol{0}$ at other places:

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

A non-masked attention module can be thought of as a masked attention module where the mask has all entries zero.

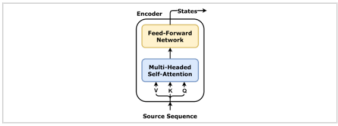
For example, the following matrix is commonly used in decoder self-attention modules, called "causal masking":

$$M_{\text{causal}} = \begin{bmatrix} 0 & -\infty & -\infty & \dots & -\infty \\ 0 & 0 & -\infty & \dots & -\infty \\ 0 & 0 & 0 & \dots & -\infty \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix}$$

In words, it means that each token can pay attention to itself, and every token before it, but not any after it. As an example of an uncommon use of mask matrix, the XLNet considers all masks of the form $PM_{\text{causal}}P^{-1}$, where P is a random permutation matrix.[53]

Encoder

An encoder consists of an embedding layer, followed by multiple encoder layers.



One encoder layer.

Each encoder layer consists of two major components: a self-attention mechanism and a feed-forward layer. It takes an input as a sequence of input vectors, applies the self-attention mechanism, to produce an intermediate sequence of vectors, then applies the feed-forward layer for each vector individually. Schematically, we have:

$$\begin{aligned} &\text{given input vectors } h_0, h_1, \dots \\ &\text{combine them into a matrix } H = \begin{bmatrix} h_0 \\ h_1 \\ \vdots \end{bmatrix} \\ &\mathbf{EncoderLayer}(H) = \begin{bmatrix} \mathbf{FFN}(\mathbf{MultiheadedAttention}(H, H, H)_0) \\ \mathbf{FFN}(\mathbf{MultiheadedAttention}(H, H, H)_1) \\ \vdots \end{bmatrix} \end{aligned}$$

where $\mathbf{FFN}$ stands for "feed-forward network". We can more succinctly write it as

$$\mathbf{EncoderLayer}(H) = \mathbf{FFN}(\mathbf{MultiheadedAttention}(H, H, H))$$

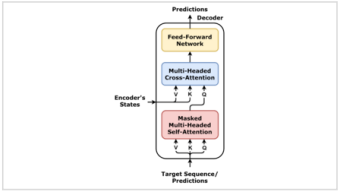
with the implicit convention that the $\mathbf{FFN}$ is applied to each row of the matrix individually.

The encoder layers are stacked. The first encoder layer takes the sequence of input vectors from the embedding layer, producing a sequence of vectors. This sequence of vectors is processed by the second encoder, and so on. The output from the final encoder layer is then used by the decoder.

As the encoder processes the entire input all at once, every token can attend to every other token (all-to-all attention), so there is no need for causal masking.

Decoder

A decoder consists of an embedding layer, followed by multiple decoder layers, followed by an un-embedding layer.



One decoder layer.

Each decoder consists of three major components: a causally masked self-attention mechanism, a cross-attention mechanism, and a feed-forward neural network. The decoder functions in a similar fashion to the encoder, but an additional attention mechanism is inserted which instead draws relevant information from the encodings generated by the encoders. This mechanism can also be called the *encoder-decoder attention*.[1][50]

Like the first encoder, the first decoder takes positional information and embeddings of the output sequence as its input, rather than encodings. The transformer must not use the current or future output to predict an output, so the output sequence must be partially masked to prevent this reverse information flow.[1] This allows for autoregressive text generation. For decoding, all-to-all attention is inappropriate, because a token cannot attend to tokens not yet generated. Thus, the self-attention module in the decoder is causally masked.

In contrast, the cross-attention mechanism attends to the output vectors of the encoder, which is computed before the decoder starts decoding. Consequently, there is no need for masking in the cross-attention mechanism.

Schematically, we have:

$$\begin{aligned} H' &= \mathbf{MaskedMultiheadedAttention}(H, H, H) \\ \mathbf{DecoderLayer}(H) &= \mathbf{FFN}(\mathbf{MultiheadedAttention}(H', H^E, H^E)) \end{aligned}$$

where H^E is the matrix with rows being the output vectors from the encoder.

The last decoder is followed by a final un-embedding layer. to produce the output probabilities over the vocabulary. Then, one of the tokens is sampled according to the probability, and the decoder can be run again to produce the next token, etc, autoregressively generating output text.

Full transformer architecture

Sublayers

Each encoder layer contains 2 sublayers: the self-attention and the feedforward network. Each decoder layer contains 3 sublayers: the causally masked self-attention, the cross-attention, and the feedforward network.

The final points of detail are the residual connections and layer normalization (LayerNorm, or LN), which while conceptually unnecessary, are necessary for numerical stability and convergence. Similarly to how the feedforward network modules are applied individually to each vector, the LayerNorm is also applied individually to each vector.

There are two common conventions in use: the *post-LN* and the *pre-LN* convention. In the post-LN convention, the output of each sublayer is

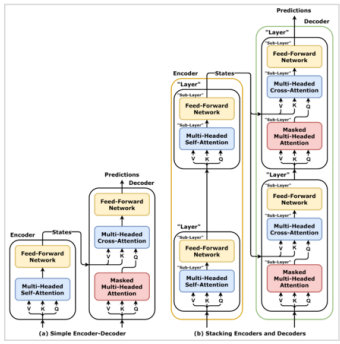
$$\mathbf{LayerNorm}(x + \mathbf{Sublayer}(x))$$

where $\mathbf{Sublayer}(x)$ is the function implemented by the sublayer itself.

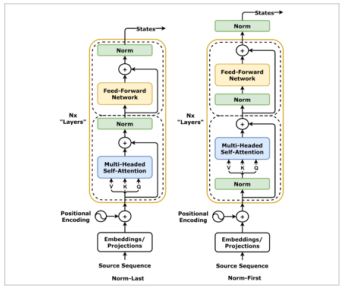
In the pre-LN convention, the output of each sublayer is

$$x + \mathbf{Sublayer}(\mathbf{LayerNorm}(x))$$

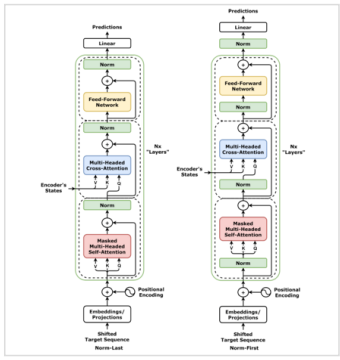
The original 2017 Transformer used the post-LN convention. It was difficult to train and required careful hyperparameter tuning and a "warm-up" in learning rate, where it starts small and gradually increases. The pre-LN convention was developed in 2020, which was found to be easier to train, requiring no warm-up, leading to faster convergence.[42]



(a) One encoder layer and one decoder layer. (b) Two encoder layers and two decoder layers. The sublayers are labelled as well.



Transformer encoder with norm-first and norm-last.



Transformer decoder with norm-first and norm-last.

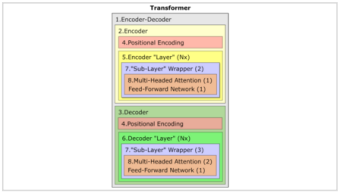
Pseudocode

The following is the pseudocode for a standard pre-LN encoder-decoder Transformer, adapted from[54]

```
input: Encoder input t_e
       Decoder input t_d
output: Array of probability distributions, with shape (decoder vocabulary size x length(decoder
output sequence))

/* encoder */
z_e ← encoder.tokenizer(t_e)

for each t in 1:length(z_e) do
    z_e[t] ← encoder.embedding(z_e[t]) + encoder.positional_embedding(t)

for each l in 1:length(encoder.layers) do
    layer ← encoder.layers[l]

    /* first sublayer */
    z_e_copy ← copy(z_e)
    for each t in 1:length(z_e) do
        z_e[t] ← layer.layer_norm(z_e[t])
    z_e ← layer.multiheaded_attention(z_e, z_e, z_e)
    for each t in 1:length(z_e) do
        z_e[t] ← z_e[t] + z_e_copy[t]

    /* second sublayer */
    z_e_copy ← copy(z_e)
    for each t in 1:length(z_e) do
        z_e[t] ← layer.layer_norm(z_e[t])
    z_e ← layer.feedforward(z_e)
    for each t in 1:length(z_e) do
        z_e[t] ← z_e[t] + z_e_copy[t]

for each t in 1:length(z_e) do
    z_e[t] ← encoder.final_layer_norm(z_e[t])

/* decoder */
z_d ← decoder.tokenizer(t_d)

for each t in 1:length(z_d) do
    z_d[t] ← decoder.embedding(z_d[t]) + decoder.positional_embedding(t)

for each l in 1:length(decoder.layers) do
        layer ← decoder.layers[l]

        /* first sublayer */
        z_d_copy ← copy(z_d)
        for each t in 1:length(z_d) do
            z_d[t] ← layer.layer_norm(z_d[t])
```

```
        z_d ← layer.masked_multiheaded_attention(z_d, z_d, z_d)
        for each t in 1:length(z_d) do
            z_d[t] ← z_d[t] + z_d_copy[t]

        /* second sublayer */
        z_d_copy ← copy(z_d)
        for each t in 1:length(z_d) do
            z_d[t] ← layer.layer_norm(z_d[t])
        z_d ← layer.multiheaded_attention(z_d, z_e, z_e)
        for each i in 1:length(z_d) do
            z_d[t] ← z_d[t] + z_d_copy[t]

        /* third sublayer */
        z_d_copy ← copy(z_d)
        for each t in 1:length(z_d) do
            z_d[t] ← layer.layer_norm(z_d[t])
        z_d ← layer.feedforward(z_d)
        for each t in 1:length(z_d) do
            z_d[t] ← z_d[t] + z_d_copy[t]

z_d ← decoder.final_layer_norm(z_d)

output_distributions ← []
for each t in 1:length(z_d) do
    output_distributions.append(decoder.unembed(z_d[t]))

return output_distributions
```

Block diagram for the full Transformer architecture.



Schematic object hierarchy for the full Transformer architecture, in object-oriented programming style.

Terminology

The Transformer architecture, being modular, allows variations. Several common variations are described here.[55]

An "encoder-only" Transformer applies the encoder to map an input text into a sequence of vectors that represent the input text. This is usually used for text embedding and representation learning for downstream applications. BERT is encoder-only. They are less often used currently, as they were found to be not significantly better than training an encoder-decoder Transformer, then taking just the encoder.[47]

A "decoder-only" Transformer is not literally decoder-only, since without an encoder, the cross-attention mechanism has nothing to attend to. Thus, the decoder layers in a decoder-only Transformer is composed of just two sublayers: the causally masked self-attention, and the feedforward network. This is usually used for text generation and instruction following. The models in the GPT series and Chinchilla series are decoder-only.

An "encoder-decoder" Transformer is generally the same as the original Transformer, with 2 sublayers per encoder layer and 3 sublayers per decoder layer, etc. They might have minor architectural improvements, such as alternative activation functions, changing the location of normalization, etc. This is also usually used for text generation and instruction following. The models in the T5 series are encoder-decoder.[55]

A "prefixLM" (prefix language model) is a decoder-only architecture, but with prefix masking, which is different from causal masking. Specifically, it has mask of the form[55]

$$M_{\text{prefixLM}} = \begin{bmatrix} \mathbf{0} & \mathbf{0}, -\infty \\ \mathbf{0} & M_{\text{causal}} \end{bmatrix}$$

where the first columns correspond to the "prefix", and the subsequent columns correspond to the autoregressively generated text based on the prefix. They resemble encoder-decoder models, but has less "sparsity". Such models are rarely used, though they are cited as theoretical possibilities and benchmarked comparisons.[47]

There are also mixed seq2seq models. For example, in 2020, Google Translate replaced the previous RNN-encoder–RNN-decoder model by a Transformer-encoder–RNN-decoder model, on the argument that an RNN-decoder runs much faster than Transformer-decoder when run autoregressively.[56]

Subsequent work

Alternative activation functions

The original transformer uses ReLU activation function. Other activation functions were developed. The Llama series used SwiGLU;[57] both GPT-1 and BERT[31] used GELU.[58]

Alternative normalizations

The normalization used in the Transformer can be different from LayerNorm. One example is RMSNorm[59] which is used in the Llama series. Other examples include ScaleNorm,[60] or FixNorm.[60]

Alternative positional encodings

Transformers may use other positional encoding methods than sinusoidal.[61]

The original Transformer paper reported using a learned positional encoding,[62] but finding it not superior to the sinusoidal one.[1] Later, [63] found that causal masking itself provides enough signal to a Transformer decoder that it can learn to implicitly perform absolute positional encoding without the positional encoding module.

RoPE

RoPE (rotary positional embedding),[64] is best explained by considering a list of 2-dimensional vectors $[(x_1^{(1)}, x_1^{(2)}), (x_2^{(1)}, x_2^{(2)}), (x_3^{(1)}, x_3^{(2)}), \dots]$. Now pick some angle θ. Then RoPE encoding is

$$\mathrm{RoPE}\left(x_m^{(1)}, x_m^{(2)}, m\right) = \begin{pmatrix} \cos m\theta & -\sin m\theta \\ \sin m\theta & \cos m\theta \end{pmatrix} \begin{pmatrix} x_m^{(1)} \\ x_m^{(2)} \end{pmatrix} = \begin{pmatrix} x_m^{(1)} \cos m\theta - x_m^{(2)} \sin m\theta \\ x_m^{(2)} \cos m\theta + x_m^{(1)} \sin m\theta \end{pmatrix}$$

Equivalently, if we write the 2-dimensional vectors as complex numbers $z_m := x_m^{(1)} + i x_m^{(2)}$, then RoPE encoding is just multiplication by an angle:

$$\mathrm{RoPE}\left(z_m, m\right) = e^{im\theta} z_m$$

For a list of $2n$-dimensional vectors, a RoPE encoder is defined by a sequence of angles $\theta^{(1)}, \dots, \theta^{(n)}$. Then the RoPE encoding is applied to each pair of coordinates.

The benefit of RoPE is that the dot-product between two vectors depends on their relative location only:

$$\mathrm{RoPE}\left(x, m\right)^T \mathrm{RoPE}\left(y, n\right) = \mathrm{RoPE}\left(x, m+k\right)^T \mathrm{RoPE}\left(y, n+k\right)$$

for any integer k.

ALiBi

ALiBi (Attention with Linear Biases)[65] is not a *replacement* for the positional encoder on the original transformer. Instead, it is an *additional* positional encoder that is directly plugged into the attention mechanism. Specifically, the ALiBi attention mechanism is

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^\mathrm{T}}{\sqrt{d_k}} + sB\right)V$$

Here, s is a real number ("scalar"), and B is the *linear bias* matrix defined by

$$B = \begin{pmatrix} 0 & 1 & 2 & 3 & \cdots \\ -1 & 0 & 1 & 2 & \cdots \\ -2 & -1 & 0 & 1 & \cdots \\ -3 & -2 & -1 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

in other words, $B_{i,j} = j - i$. The idea being that the linear bias matrix is a softened mask. Just as 0 represent full attention paid, and $-\infty$ represents no attention paid, the linear bias matrix increases attention paid in one direction and decreases attention paid in the other direction.

ALiBi allows pretraining on short context windows, then finetuning on longer context windows. Since it is directly plugged into the attention mechanism, it can be combined with any positional encoder that is plugged into the "bottom" of the entire network (which is where the sinusoidal encoder on the original transformer, as well as RoPE and many others, are located).

Relative Position Encodings

Relative Position Encodings[66] is similar to ALiBi, but more generic:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^{\mathrm{T}}}{\sqrt{d_k}} + B\right)V$$

where B is a Toeplitz matrix, that is, $B_{i,j} = B_{i',j'}$ whenever $i - j = i' - j'$. This is contrasted with the original sinusoidal positional encoding, which is an "absolute positional encoding".[67]

Efficient implementation

The transformer model has been implemented in standard deep learning frameworks such as TensorFlow and PyTorch. *Transformers* is a library produced by Hugging Face that supplies transformer-based architectures and pretrained models.[11]

FlashAttention

FlashAttention[68] is an algorithm that implements the transformer attention mechanism efficiently on a GPU. It performs matrix multiplications in blocks, such that each block fits within the cache of a GPU, and by careful management of the blocks it minimizes data copying between GPU caches (as data movement is slow).

An improved version, FlashAttention-2,[69][70][71] was developed to cater to the rising demand for language models capable of handling longer context lengths. It offers enhancements in work partitioning and parallelism, enabling it to achieve up to 230 TFLOPs/s on A100 GPUs (FP16/BF16), a 2x speed increase over the original FlashAttention.

Key advancements in FlashAttention-2 include the reduction of non-matmul FLOPs, improved parallelism over the sequence length dimension, better work partitioning between GPU warps, and added support for head dimensions up to 256 and multi-query attention (MQA) and grouped-query attention (GQA).

Benchmarks revealed FlashAttention-2 to be up to 2x faster than FlashAttention and up to 9x faster than a standard attention implementation in PyTorch. Future developments include optimization for new hardware like H100 GPUs and new data types like FP8.

Multi-Query Attention

Multi-Query Attention changes the multiheaded attention mechanism.[72] Whereas normally,

$$\text{MultiheadedAttention}(Q, K, V) = \text{Concat}_{i \in [n_{\text{heads}}]}\left(\text{Attention}(XW_i^Q, XW_i^K, XW_i^V)\right)W^O$$

with Multi-Query Attention, there is just one W^K, W^V, thus:

$$\text{MultiQueryAttention}(Q, K, V) = \text{Concat}_{i \in [n_{\text{heads}}]}\left(\text{Attention}(XW_i^Q, XW^K, XW^V)\right)W^O$$

This has a neutral effect on model quality and training speed, but increases inference speed.

Caching

When an autoregressive transformer is used for inference, such as generating text, the query vector is different at each step, but the already-computed key and value vectors are always the same. The **KV caching** method saves the computed key and value vectors at each attention block, so that they are not recomputed at each new token. **PagedAttention** applies memory paging to KV caching.[73][74][75]

If a transformer is used with a baked-in prompt, such as ["You are a customer support agent..."], then the key and value vectors can be computed for the prompt, and saved on disk. The saving in compute is significant when the model is used for many short interactions, such as in online chatbots.

Speculative decoding

Transformers are used in large language models for autoregressive sequence generation: generating a stream of text, one token at a time. However, in most settings, decoding from language models is memory-bound, meaning that we have spare compute power available. Speculative decoding[76][77] uses this spare compute power by computing several tokens in parallel. Similarly to speculative execution in CPUs, future tokens are computed concurrently, by speculating on the value of previous tokens, and are later discarded if it turns out the speculation was incorrect.

Specifically, consider a transformer model like GPT-3 with a context window size of 512. To generate an entire context window autoregressively with greedy decoding, it must be run for 512 times, each time generating a token $x_1, x_2, \ldots, x_{512}$. However, if we had some educated guess for the values of these tokens, we could verify all of them in parallel, in one run of the model, by checking that each x_t is indeed the token with the largest log-likelihood in the t-th output.

In speculative decoding, a smaller model or some other simple heuristic is used to generate a few speculative tokens that are subsequently verified by the larger model. For example, suppose a small model generated four speculative tokens: $\tilde{x_1}, \tilde{x_2}, \tilde{x_3}, \tilde{x_4}$. These tokens are run through the larger model, and only $\tilde{x_1}$ and $\tilde{x_2}$ are accepted. The same run of the large model already generated a new token x_3 to replace $\tilde{x_3}$, and $\tilde{x_4}$ is completely discarded. The process then repeats (starting from the 4th token) until all tokens are generated.

For non-greedy decoding, similar ideas apply, except the speculative tokens are accepted or rejected stochastically, in a way that guarantees the final output distribution is the same as if speculative decoding was not used.[76][78]

Sub-quadratic transformers

Training transformer-based architectures can be expensive, especially for long inputs.[79] Many methods have been developed to attempt to address the issue. *Long Range Arena* (2020)[80] is a standard benchmark for comparing the behavior of transformer architectures over long inputs.

Alternative attention graphs

The standard attention graph is either all-to-all or causal, both of which scales as $O(N^2)$ where N is the number of tokens in a sequence.

Reformer (2020)[79][81] reduces the computational load from $O(N^2)$ to $O(N \ln N)$ by using locality-sensitive hashing and reversible layers.[82]

Sparse attention[83] uses attention graphs that grows slower than $O(N^2)$. For example, BigBird (2020)[84] uses random small-world networks which grows as $O(N)$.

Ordinary transformers require a memory size that is quadratic in the size of the context window. Attention-free transformers[85] reduce this to a linear dependence while still retaining the advantages of a transformer by linking the key to the value.

Random Feature Attention

Random Feature Attention (2021)[86] uses Fourier random features:

$$\varphi(x) = \frac{1}{\sqrt{D}}[\cos\langle w_1, x\rangle, \sin\langle w_1, x\rangle, \cdots \cos\langle w_D, x\rangle, \sin\langle w_D, x\rangle]^T$$

where $w_1, \dots, w_D$ are independent samples from the normal distribution $N(0, \sigma^2 I)$. This choice of parameters satisfy $\mathbb{E}[\langle \varphi(x), \varphi(y)\rangle] = e^{-\frac{\|x-y\|^2}{2\sigma^2}}$, or

$$e^{\langle x,y\rangle/\sigma^2} = \mathbb{E}[\langle e^{\|x\|^2/2\sigma^2}\varphi(x), e^{\|y\|^2/2\sigma^2}\varphi(y)\rangle] \approx \langle e^{\|x\|^2/2\sigma^2}\varphi(x), e^{\|y\|^2/2\sigma^2}\varphi(y)\rangle$$

Consequently, the one-headed attention, with one query, can be written as

$$\text{Attention}(q, K, V) = \text{softmax}\left(\frac{qK^{\mathrm{T}}}{\sqrt{d_k}}\right)V \approx \frac{\varphi(q)^T \sum_i e^{\|k_i\|^2/2\sigma^2}\varphi(k_i)v_i^T}{\varphi(q)^T \sum_i e^{\|k_i\|^2/2\sigma^2}\varphi(k_i)}$$

where $\sigma = d_K^{1/4}$. Similarly for multiple queries, and for multiheaded attention.

This approximation can be computed in linear time, as we can compute the matrix $\varphi(k_i)v_i^T$ first, then multiply it with the query. In essence, we have managed to obtain a more precise version of

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^{\mathrm{T}}}{\sqrt{d_k}}\right)V \approx Q(K^TV/\sqrt{d_k})$$

Performer (2022)[87] uses the same Random Feature Attention, but $w_1, \dots, w_D$ are first independently sampled from the normal distribution $N(0, \sigma^2 I)$, then they are Gram-Schmidt processed.

Multimodality

Transformers can also be used/adapted for modalities (input or output) beyond just text, usually by finding a way to "tokenize" the modality.

Multimodal models can either be trained from scratch, or by finetuning. A 2022 study found that Transformers pretrained only on natural language can be finetuned on only 0.03% of parameters and become competitive with LSTMs on a variety of logical and visual tasks, demonstrating transfer learning.[88] The LLaVA was a vision-language model composed of a language model (Vicuna-13B)[89] and a vision model (ViT-L/14), connected by a linear layer. Only the linear layer is finetuned.[90]

Vision transformers[37] adapt the transformer to computer vision by breaking down input images as a series of patches, turning them into vectors, and treating them like tokens in a standard transformer.

Conformer[38] and later Whisper[91] follow the same pattern for speech recognition, first turning the speech signal into a spectrogram, which is then treated like an image, i.e. broken down into a series of patches, turned into vectors and treated like tokens in a standard transformer.

Perceivers[92][93] are a variant of Transformers designed for multimodality.

For image generation, two notable architectures are DALL-E 1 (2021) and Parti (2022).[94] Unlike later models, DALL-E is not a diffusion model. Instead, it uses a decoder-only Transformer that autoregressively generates a text, followed by the token representation of an image, which is then converted by a variational autoencoder to an image.[95] Parti is an encoder-decoder Transformer, where the encoder processes a text prompt, and the decoder generates a token representation of an image.[96]

Applications

The transformer has had great success in natural language processing (NLP). Many large language models such as GPT-2, GPT-3, GPT-4, Claude, BERT, XLNet, RoBERTa and ChatGPT demonstrate the ability of transformers to perform a wide variety of NLP-related subtasks and their related real-world or practical applications, including:

- machine translation
- time series prediction
- document summarization
- document generation
- named entity recognition (NER)[97]
- writing computer code based on requirements expressed in natural language.
- speech-to-text

Beyond traditional NLP, the transformer architecture has had success in other applications, such as:

- biological sequence analysis
- video understanding
- protein folding (such as AlphaFold)
- evaluating chess board positions. Using static evaluation alone (that is, with no Minimax search) transformer achieved an Elo of 2895, putting it at grandmaster level.[10]

See also

- seq2seq – Family of machine learning approaches
- Perceiver – Variant of Transformer designed for multimodal data
- Vision transformer – Variant of Transformer designed for vision processing
- Large language model – Type of artificial neural network
- BERT (language model) – Series of language models developed by Google AI
- Generative pre-trained transformer – Type of large language model
- T5 (language model) – Series of large language models developed by Google AI

Notes

1. Gated recurrent units (2014) further reduced its complexity.
2. Some architectures, such as RWKV or state space models, avoid the issue.

References

1. Vaswani, Ashish; Shazeer, Noam; Parmar, Niki; Uszkoreit, Jakob; Jones, Llion; Gomez, Aidan N; Kaiser, Łukasz; Polosukhin, Illia (2017). "Attention is All you Need" (https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf) (PDF). *Advances in Neural Information Processing Systems*. **30**. Curran Associates, Inc.
2. Hochreiter, Sepp; Schmidhuber, Jürgen (1 November 1997). "Long Short-Term Memory". *Neural Computation*. **9** (8): 1735–1780. doi:10.1162/neco.1997.9.8.1735 (https://doi.org/10.1162%2Fneco.1997.9.8.1735). ISSN 0899-7667 (https://search.worldcat.org/issn/0899-7667). PMID 9377276 (https://pubmed.ncbi.nlm.nih.gov/9377276). S2CID 1915014 (https://api.semanticscholar.org/CorpusID:1915014).
3. "Better Language Models and Their Implications" (https://openai.com/blog/better-language-models/). *OpenAI*. 2019-02-14. Archived (https://web.archive.org/web/20201219132206/https://openai.com/blog/better-language-models/) from the original on 2020-12-19. Retrieved 2019-08-25.
4. Bahdanau; Cho, Kyunghyun; Bengio, Yoshua (September 1, 2014). "Neural Machine Translation by Jointly Learning to Align and Translate". arXiv:1409.0473 (https://arxiv.org/abs/1409.0473) [cs.CL (https://arxiv.org/archive/cs.CL)].