Unknown Title

Ashish Vaswani* Google Brain avaswani@google.com
Noam Shazeer* Google Brain noam@google.com
Niki Parmar* Google Research nikip@google.com
Jakob Uszkoreit* Google Research usz@google.com
Llion Jones* Google Research llion@google.com
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu
Łukasz Kaiser* Google Brain lukaszkaiser@google.com
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

The Transformer has been on a lot of people's minds over the last ~~year~~ five years. This post presents an annotated version of the paper in the form of a line-by-line implementation. It reorders and deletes some sections from the original paper and adds comments throughout. This document itself is a working notebook, and should be a completely usable implementation. Code is available here.

The goal of reducing sequential computation also forms the foundation of the Extended Neural GPU, ByteNet and ConvS2S, all of which use convolutional neural networks as basic building block, computing hidden representations in parallel for all input and output positions. In these models, the number of operations required to relate signals from two arbitrary input or output positions grows in the distance between positions, linearly for ConvS2S and logarithmically for ByteNet. This makes it more difficult to learn dependencies between distant positions. In the Transformer this is reduced to a constant number of operations, albeit at the cost of reduced effective resolution due to averaging attention-weighted positions, an effect we counteract with Multi-Head Attention.

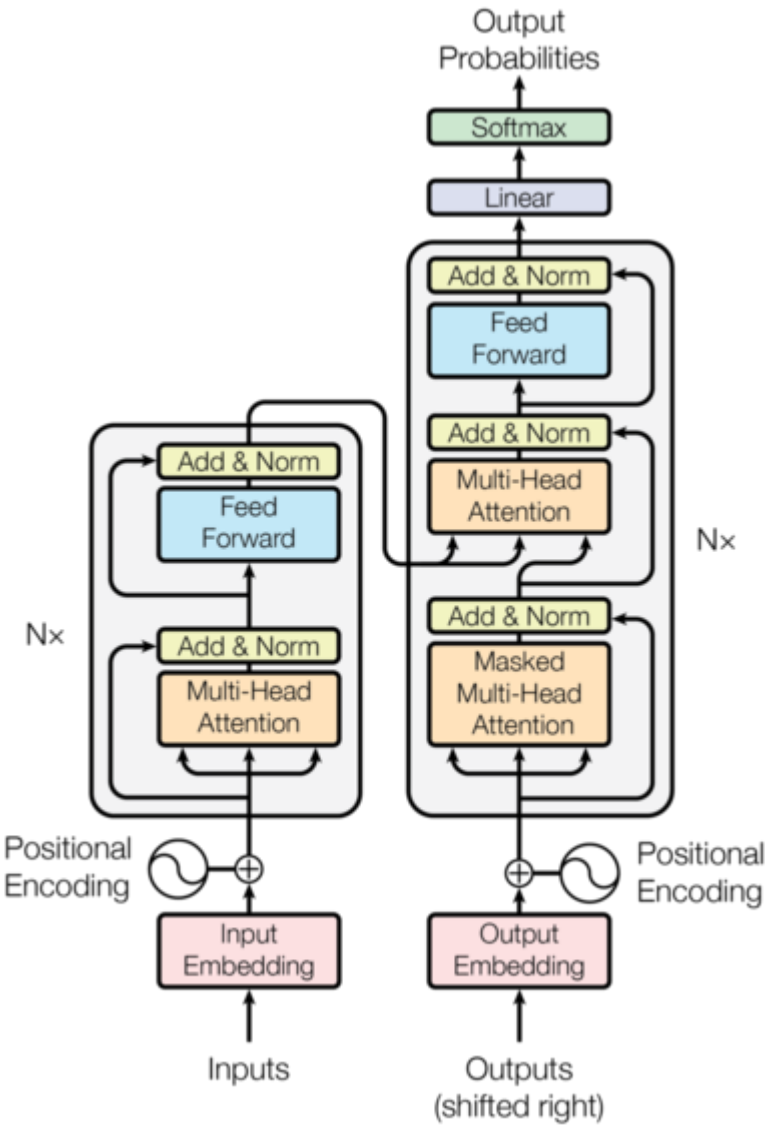
Self-attention, sometimes called intra-attention is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence. Self-attention has been used successfully in a variety of tasks including reading comprehension, abstractive summarization, textual entailment and learning task-independent sentence representations. End-to-end memory networks are based on a recurrent attention mechanism instead of sequencealigned recurrence and have been shown to perform well on simple-language question answering and language modeling tasks.

To the best of our knowledge, however, the Transformer is the first transduction model relying entirely on self-attention to compute representations of its input and output without using sequence aligned RNNs or convolution.

Most competitive neural sequence transduction models have an encoder-decoder structure (cite). Here, the encoder maps an input sequence of symbol representations (x_1, ..., x_n)(x1,...,xn) to a sequence of continuous representations \mathbf{z} = (z_1, ..., z_n)z=(z1,...,zn). Given \mathbf{z}z, the decoder then generates an output sequence (y_1,...,y_m)(y1,...,ym) of symbols one element at a time. At each step the

model is auto-regressive (cite), consuming the previously generated symbols as additional input when generating the next.

The Transformer follows this overall architecture using stacked self-attention and point-wise, fully connected layers for both the encoder and decoder, shown in the left and right halves of Figure 1, respectively.



The encoder is composed of a stack of N=6N=6 identical layers.

We employ a residual connection (cite) around each of the two sub-layers, followed by layer normalization (cite).

That is, the output of each sub-layer is \mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))LayerNorm(x+Sublayer(x)), where \mathrm{Sublayer}(x)Sublayer(x) is the function implemented by the sub-layer itself. We apply dropout (cite) to the output of each sub-layer, before it is added to the sub-layer input and normalized.

To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension d_{\text{model}}=512dmodel=512.

Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network.

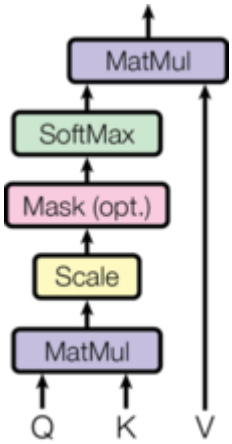
The decoder is also composed of a stack of N=6N=6 identical layers.

In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization.

We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position ii can depend only on the known outputs at positions less than ii.

An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

We call our particular attention “Scaled Dot-Product Attention”. The input consists of queries and keys of dimension d_kdk, and values of dimension d_vdv. We compute the dot products of the query with all keys, divide each by \sqrt{d_k}dk$\sqrt{}$, and apply a softmax function to obtain the weights on the values.

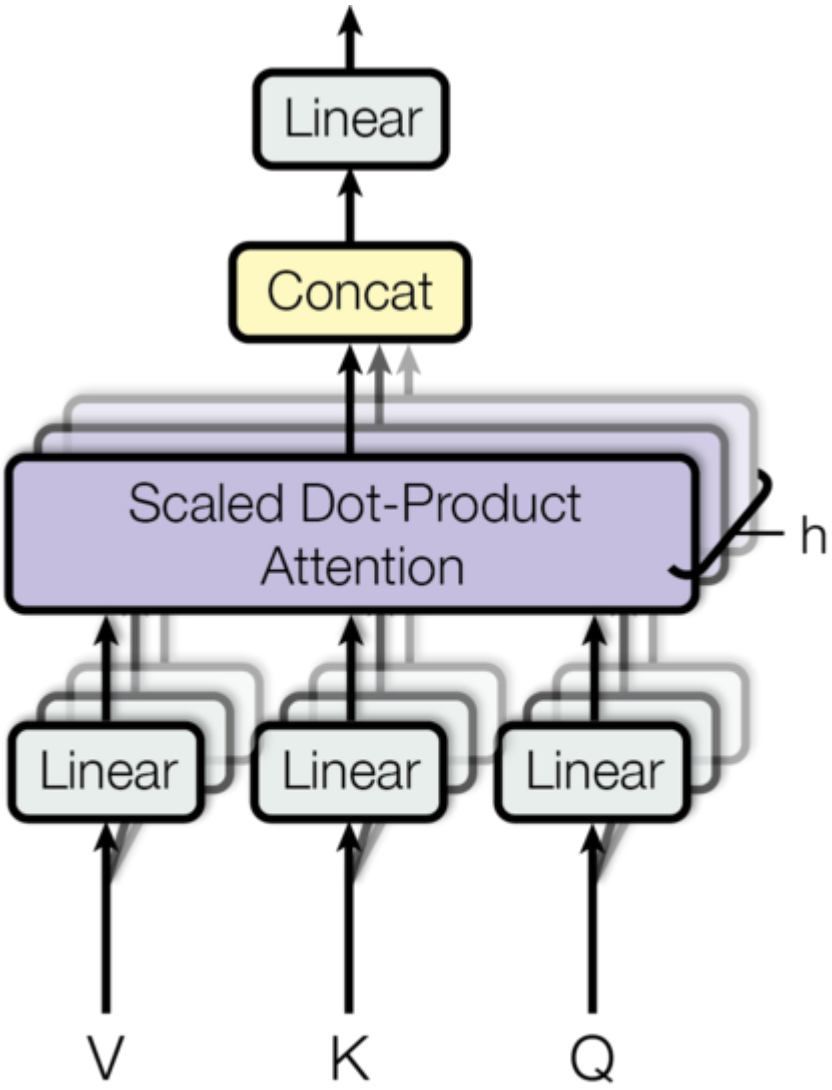


In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix QQ. The keys and values are also packed together into matrices KK and VV. We compute the matrix of outputs as:

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

The two most commonly used attention functions are additive attention (cite), and dot-product (multiplicative) attention. Dot-product attention is identical to our algorithm, except for the scaling factor of \frac{1}{\sqrt{d_k}}dk$\sqrt{}$1. Additive attention computes the compatibility function using a feed-forward network with a single hidden layer. While the two are similar in theoretical complexity, dot-product attention is much faster and more space-efficient in practice, since it can be implemented using highly optimized matrix multiplication code.

While for small values of d_kdk the two mechanisms perform similarly, additive attention outperforms dot product attention without scaling for larger values of d_kdk (cite). We suspect that for large values of d_kdk, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients (To illustrate why the dot products get large, assume that the components of qq and kk are independent random variables with mean 00 and variance 11. Then their dot product, q \cdot k = \sum_{i=1}^{d_k} q_ik_iq·k=∑i=1dkqiki, has mean 00 and variance d_kdk.). To counteract this effect, we scale the dot products by \frac{1}{\sqrt{d_k}}dk√1.



Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. With a single attention head, averaging inhibits this.

\mathrm{MultiHead}(Q, K, V) = \mathrm{Concat}(\mathrm{head_1}, ..., \mathrm{head_h})W^O \\ \text{where}~\mathrm{head_i} = \mathrm{Attention}(QW^Q_i, KW^K_i, VW^V_i) MultiHead(Q,K,V)=Concat(head1,...,headh)WOwhere headi=Attention(QWiQ,KWiK,VWiV)

Where the projections are parameter matrices W^Q_i \in \mathbb{R}^{d_{\text{model}} \times d_k}WiQ ∈Rdmodel×dk, W^K_i \in \mathbb{R}^{d_{\text{model}} \times d_k}WiK∈Rdmodel×dk, W^V_i \in \mathbb{R}^{d_{\text{model}} \times d_v}WiV∈Rdmodel×dv and W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}}WO∈Rhdv×dmodel.

In this work we employ h=8h=8 parallel attention layers, or heads. For each of these we use d_k=d_v=d_{\text{model}}/h=64dk=dv=dmodel/h=64. Due to the reduced dimension of each head, the total computational cost is similar to that of single-head attention with full dimensionality.

The Transformer uses multi-head attention in three different ways: 1) In “encoder-decoder attention” layers, the queries come from the previous decoder layer, and the memory keys and values come from the output of the encoder. This allows every position in the decoder to attend over all positions in the input sequence. This mimics the typical encoder-decoder attention mechanisms in sequence-to-sequence models such as (cite).

-

The encoder contains self-attention layers. In a self-attention layer all of the keys, values and queries come from the same place, in this case, the output of the previous layer in the encoder. Each position in the encoder can attend to all positions in the previous layer of the encoder.

-

Similarly, self-attention layers in the decoder allow each position in the decoder to attend to all positions in the decoder up to and including that position. We need to prevent leftward information flow in the decoder to preserve the auto-regressive property. We implement this inside of scaled dot-product attention by masking out (setting to -\infty−∞) all values in the input of the softmax which correspond to illegal connections.

In addition to attention sub-layers, each of the layers in our encoder and decoder contains a fully connected feed-forward network, which is applied to each position separately and identically. This consists of two linear transformations with a ReLU activation in between.

\mathrm{FFN}(x)=\max(0, xW_1 + b_1) W_2 + b_2FFN(x)=max(0,xW1+b1)W2+b2

While the linear transformations are the same across different positions, they use different parameters from layer to layer. Another way of describing this is as two convolutions with kernel size 1. The dimensionality of input and output is d_{\text{model}}=512dmodel=512, and the inner-layer has dimensionality d_{ff}=2048dff=2048.

Similarly to other sequence transduction models, we use learned embeddings to convert the input tokens and output tokens to vectors of dimension d_{\text{model}}dmodel. We also use the usual learned linear transformation and softmax function to convert the decoder output to predicted next-token probabilities. In our model, we share the same weight matrix between the two embedding layers and the pre-softmax linear transformation, similar to (cite). In the embedding layers, we multiply those weights by \sqrt{d_{\text{model}}}dmodel√ .

Since our model contains no recurrence and no convolution, in order for the model to make use of the order of the sequence, we must inject some information about the relative or absolute position of the tokens in the sequence. To this end, we add “positional encodings” to the input embeddings at the bottoms of the encoder and decoder stacks. The positional encodings have the same dimension d_{\text{model}}dmodel as the embeddings, so that the two can be summed. There are many choices of positional encodings, learned and fixed (cite).

PE_{(pos,2i)} = \sin(pos / 10000^{2i/d_{\text{model}}})PE(pos,2i)=sin(pos/100002i/dmodel)

PE_{(pos,2i+1)} = \cos(pos / 10000^{2i/d_{\text{model}}})PE(pos,2i+1)=cos(pos/100002i/dmodel)

where pospos is the position and ii is the dimension. That is, each dimension of the positional encoding corresponds to a sinusoid. The wavelengths form a geometric progression from 2\pi2π to 10000 \cdot 2\pi10000·2π. We chose this function because we hypothesized it would allow the model to easily learn to attend by relative positions, since for any fixed offset kk, PE_{pos+k}PEpos+k can be represented as a linear function of PE_{pos}PEpos.

In addition, we apply dropout to the sums of the embeddings and the positional encodings in both the encoder and decoder stacks. For the base model, we use a rate of P_{drop}=0.1Pdrop=0.1.

We also experimented with using learned positional embeddings (cite) instead, and found that the two versions produced nearly identical results. We chose the sinusoidal version because it may allow the model to extrapolate to sequence lengths longer than the ones encountered during training.

> Here we make a forward step to generate a prediction of the model. We try to use our transformer to memorize the input. As you will see the output is randomly generated due to the fact that the model is not trained yet. In the next tutorial we will build the training function and try to train our model to memorize the numbers from 1 to 10.
>
> We stop for a quick interlude to introduce some of the tools needed to train a standard encoder decoder model. First we define a batch object that holds the src and target sentences for training, as well as constructing the masks.
>
> We implement label smoothing using the KL div loss. Instead of using a one-hot target distribution, we create a distribution that has `confidence` of the correct word and the rest of the `smoothing` mass distributed throughout the vocabulary.
>
> Now we consider a real-world example using the Multi30k German-English Translation task. This task is much smaller than the WMT task considered in the paper, but it illustrates the whole system. We also show how to use multi-gpu processing to make it really fast.
>
> Batching matters a ton for speed. We want to have very evenly divided batches, with absolutely minimal padding. To do this we have to hack a bit around the default torchtext batching. This code patches their default batching to make sure we search over enough sentences to find tight batches.
>
> We can begin by trying out a simple copy-task. Given a random set of input symbols from a small vocabulary, the goal is to generate back those same symbols.

Once trained we can decode the model to produce a set of translations. Here we simply translate the first sentence in the validation set. This dataset is pretty small so the translations with greedy search are reasonably accurate.

With the addtional extensions in the last section, the OpenNMT-py replication gets to 26.9 on EN-DE WMT. Here I have loaded in those parameters to our reimplemenation.

2. Shared Embeddings: When using BPE with shared vocabulary we can share the same weight vectors between the source / target / generator. See the (cite) for details. To add this to the model simply do this:

Below the attention mask shows the position each tgt word (row) is allowed to look at (column). Words are blocked for attending to future words during training.

Below the positional encoding will add in a sine wave based on position. The frequency and offset of the wave is different for each dimension.

Next we create a generic training and scoring function to keep track of loss. We pass in a generic loss compute function that also handles parameter updates.

Label smoothing actually starts to penalize the model if it gets very confident about a given choice.

4. Model Averaging: The paper averages the last k checkpoints to create an ensembling effect. We can do this after the fact if we have a bunch of models:

Even with a greedy decoder the translation looks pretty good. We can further visualize it to see what is happening at each layer of the attention

1. BPE/ Word-piece: We can use a library to first preprocess the data into subword units. See Rico Sennrich's subword-nmt implementation. These models will transform the training data to look like this:

So this mostly covers the transformer model itself. There are four aspects that we didn't cover explicitly. We also have all these additional features implemented in OpenNMT-py.

3. Beam Search: This is a bit too complicated to cover here. See the OpenNMT-py for a pytorch implementation.

My comments are blockquoted. The main text is all from the paper itself.

Here we define a function from hyperparameters to a full model.

Note: This part is very important. Need to train with this setup of the model.

Example of the curves of this model for different model sizes and for optimization hyperparameters.

Here we can see an example of how the mass is distributed to the words based on confidence.

This code predicts a translation using greedy decoding for simplicity.

We will load the dataset using torchtext and spacy for tokenization.

This section describes the training regime for our models.

We trained on the standard WMT 2014 English-German dataset consisting of about 4.5 million sentence pairs. Sentences were encoded using byte-pair encoding, which has a shared source-target vocabulary of about 37000 tokens. For English-French, we used the significantly larger WMT 2014 English-French dataset consisting of 36M sentences and split tokens into a 32000 word-piece vocabulary.

Sentence pairs were batched together by approximate sequence length. Each training batch contained a set of sentence pairs containing approximately 25000 source tokens and 25000 target tokens.

We trained our models on one machine with 8 NVIDIA P100 GPUs. For our base models using the hyperparameters described throughout the paper, each training step took about 0.4 seconds. We trained the base models for a total of 100,000 steps or 12 hours. For our big models, step time was 1.0 seconds. The big models were trained for 300,000 steps (3.5 days).

We used the Adam optimizer (cite) with \beta_1=0.9β1=0.9, \beta_2=0.98β2=0.98 and \epsilon=10^{-9} ϵ=10−9. We varied the learning rate over the course of training, according to the formula:

lrate = d_{\text{model}}^{-0.5} \cdot \min({step_num}^{-0.5}, {step_num} \cdot {warmup_steps}^{-1.5}) lrate=dmodel−0.5·min(step_num−0.5,step_num·warmup_steps−1.5)

This corresponds to increasing the learning rate linearly for the first warmup_stepswarmup_steps training steps, and decreasing it thereafter proportionally to the inverse square root of the step number. We used warmup_steps=4000warmup_steps=4000.

During training, we employed label smoothing of value \epsilon_{ls}=0.1ϵls=0.1 (cite). This hurts perplexity, as the model learns to be more unsure, but improves accuracy and BLEU score.

__Die __Protokoll datei __kann __ heimlich __per __E - Mail __oder __FTP __an __einen __bestimmte n __Empfänger __gesendet __werden .

On the WMT 2014 English-to-German translation task, the big transformer model (Transformer (big) in Table 2) outperforms the best previously reported models (including ensembles) by more than 2.0 BLEU, establishing a new state-of-the-art BLEU score of 28.4. The configuration of this model is listed in the bottom line of Table 3. Training took 3.5 days on 8 P100 GPUs. Even our base model surpasses all previously published models and ensembles, at a fraction of the training cost of any of the competitive models.

On the WMT 2014 English-to-French translation task, our big model achieves a BLEU score of 41.0, outperforming all of the previously published single models, at less than 1/4 the training cost of the previous state-of-the-art model. The Transformer (big) model trained for English-to-French used dropout rate Pdrop = 0.1, instead of 0.3.

Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	**$3.3 \cdot 10^{18}$**	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

Hopefully this code is useful for future research. Please reach out if you have any issues.

Cheers, Sasha Rush, Austin Huang, Suraj Subramanian, Jonathan Sum, Khalid Almubarak, Stella Biderman