The Transformer – Attention is all you need. (/articles/2017/Sep/12/Transformer-Attention-is-all-you-need/)

Date Tue, 12 Sep 2017 Modified Mon, 30 Oct 2017 By *Michał Chromiak (/author/michal-chromiak.html)* Category *Sequence Models (/category/sequence-models.html)* Tags *NMT (/tag/nmt.html)* / *transformer (/tag/transformer.html)* / *Sequence transduction (/tag/sequence-transduction.html)* / *Attention model (/tag/attention-model.html)* / *Machine translation (/tag/machine-translation.html)* / *seq2seq (/tag/seq2seq.html)* / *NLP (/tag/nlp.html)*

Recommended reading before approaching this post:

- RNN – Andrej Karpathy's blog *The Unreasonable Effectiveness of Recurrent Neural Networks (http://karpathy.github.io/2015/05/21/rnn-effectiveness/)*
- Seq2Seq - Nathan Lintz *Sequence Modeling With Neural Networks (Part 1): Language & Seq2Seq (https://indico.io/blog/sequence-modeling-neuralnets-part1/)*, Part2 *Sequence modeling with attention (https://indico.io/blog/sequence-modeling-neural-networks-part2-attention-models/)*
- LSTM – Christopher Olah's blog *Understanding LSTM Networks (http://colah.github.io/posts/2015-08-Understanding-LSTMs/)* and R2Rt.com *Written Memories: Understanding, Deriving and Extending the LSTM (https://r2rt.com/written-memories-understanding-deriving-and-extending-the-lstm.html)*.
- Attention – Christopher Olah *Attention and Augmented Recurrent Neural Networks (https://distill.pub/2016/augmented-rnns/#attentional-interfaces)*

Objective or goal for the algorithm

- **Parallelization of Seq2Seq:** RNN/CNN handle sequences word-by-word sequentially which is an obstacle to parallelize. Transformer achieve parallelization by replacing recurrence with attention and encoding the symbol position in sequence. This in turn leads to significantly shorter training time.

- **Reduce sequential computation:** Constant $O(1)$ number of operations to learn dependency between two symbols independently of their position distance in sequence.

TL;DR

RNN:

- **Advantages:** are popular and successful for variable-length representations such as sequences (e.g. languages), images, etc. RNN are considered core of seq2seq (with attention). The gating models such as LSTM or GRU are for long-range error propagation.
- **Problems:** The sequentiality prohibits parallelization within instances. Long-range dependencies still tricky, despite gating. Sequence-aligned states in RNN are wasteful. Hard to model hierarchical-alike domains such as languages.

CNN:

- **Advantages:** Trivial to parallelize (per layer) and fit intuition that most dependencies are local.
- **Problems:** Path length between positions can be logarithmic when using dilated convolutions, left-padding for text. (autoregressive CNNs WaveNet, ByteNET)

Solution: Multi-head self-attention mechanism. Why attention? Table 2 of the paper shows that such attention networks can save 2–3 orders of magnitude of operations!

Intro

In this post I will elaborate on lately presented paper introducing the **Transformer** architecture. Paper: *ArXiv (https://arxiv.org/abs/1706.03762)*.

As the paper assumes the in-depth prior knowledge of some topics, I will try to explain the ideas of the paper so that they can be understood by a DL beginner.

When RNN's (or CNN) takes a sequence as an input, it handles sentences word by word. This sequentiality is an obstacle toward parallelization of the process. What is more, in cases when such sequences are too long, the model is prone to forgetting the content of distant positions in sequence or mix it with following positions' content.

One solution to this was Convolution seq2seq. Convolution enables parallelization for GPU processing. Thus *Gehring et al, 2017 (https://arxiv.org/abs/1705.03122)* (Facebook AI) present a 100% convolutional architecture to represent hierarchical representation of input sequence. The crux is that close input elements interact at lower layers while distant interacts at higher layers. The stacking of convolution layers is being used to evaluate long range dependencies between words. This is all possible despite fixed width kernels of convolution filters. The authors has also included two tricks:

- Positional embeddings - that are added to the input embeddings, capturing a sense of order in a sequence
- Multi-step attention - attention is computed with current decoder state and embedding of previous target word token

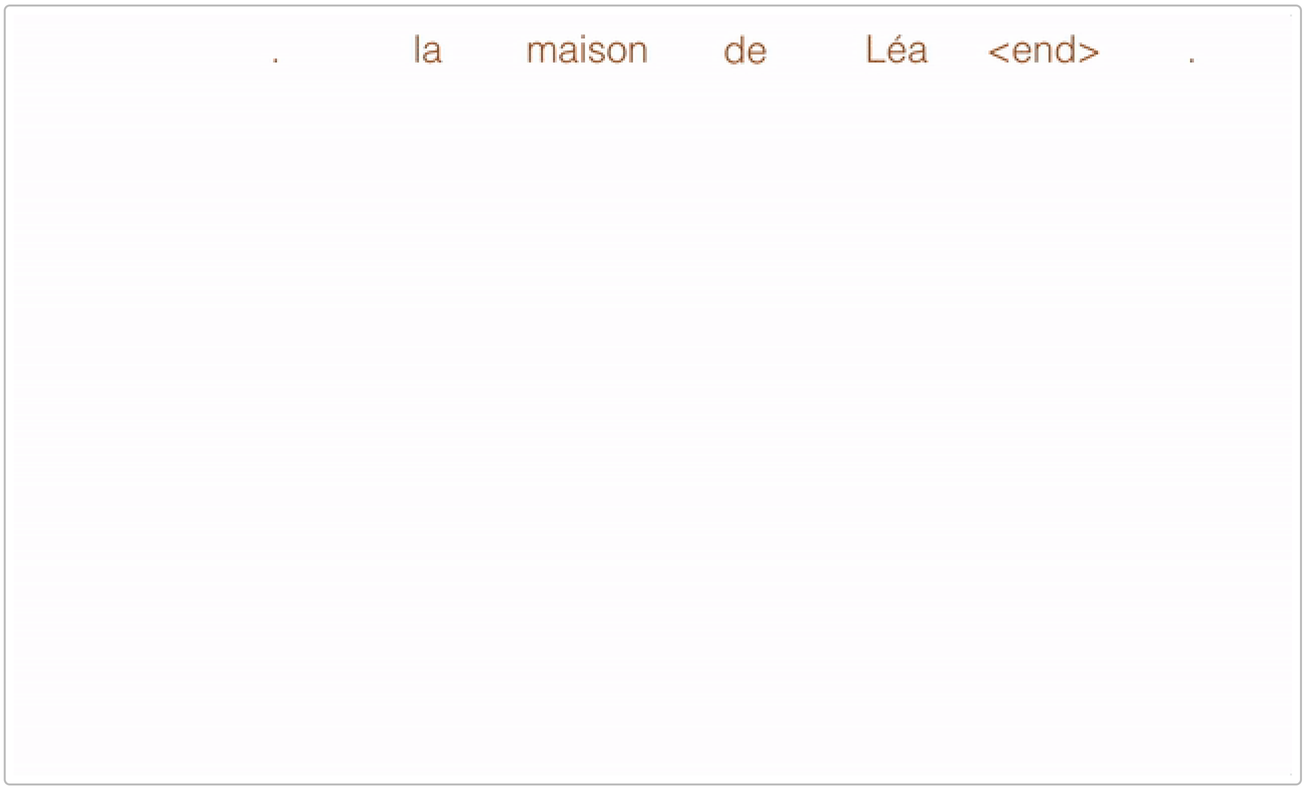

Figure 1. Multi-step attention form ConvS2S.

As an alternative to convolutions, a new approach is presented by the *Transformer*. It proposes to encode each position and applying the attention mechanism, to relate two distant words, which then can be parallelized thus, accelerating the training.

Currently, in NLP the SOTA (*state-of-the-art*) performance achieved by seq2seq models is focused around the idea of encoding the input sentence into a fixed-size vector representation. The vector size is fixed, regardless of the length of the input sentence. In obvious way this must loose some information. To face this issue *Transformer* employs an alternative approach based on attention.

Due to multiple referred work it is beneficial to also read the mentioned research

- *Denny Britz on Attention, 2017 (https://arxiv.org/abs/1703.03906)*
- *Self-attention (a.k.a Intra-attention), 2017 (https://arxiv.org/abs/1705.04304)*

Motivation

The Transformer architecture is aimed at the problem of *sequence transduction (by Alex Graves) (https://arxiv.org/abs/1211.3711)*, meaning any task where input sequences are transformed into output sequences. This includes speech recognition, text-to-speech transformation, machine translation, protein secondary structure prediction, Turing machines etc. Basically the goal is to design a single framework to handle as many sequences as possible.

Currently, complex RNN and CNN based on encoder-decoder scheme are dominating transduction models (language modeling and machine learning). Recurrent models due to sequential nature (computations focused on the position of symbol in input and output) are not allowing for parallelization along training, thus have a problem with learning long-term dependencies (*Hochreiter,, et al. (http://ieeexplore.ieee.org/xpl/articleDetails.jsp?reload=true&arnumber=5264952)*) from memory. The bigger the memory is, the better, but the memory eventually constrains batching across learning examples for long sequences, and this is why parallelization can not help.

Reducing this fundamental constraint of sequential computation has been target for numerous research like *Wavenet (https://arxiv.org/abs/1609.03499)* / *Bytenet (https://arxiv.org/abs/1610.10099)* or *ConvS2S (https://arxiv.org/abs/1705.03122)*. However, in those CNN-based approaches, the number of calculations in parallel computation of the hidden representation, for input→output position in sequence, grows with the distance between those positions. The complexity of $O(n)$ for ConvS2S and $O(nlogn)$ for ByteNet makes it harder to learn dependencies on distant positions.

Transformer reduces the number of sequential operations to relate two symbols from input/output sequences to a constant $O(1)$ number of operations. Transformer achieves this with the *multi-head attention* mechanism that allows to model dependencies regardless of their distance in input or output sentence.

Up till now, most of the research including attention is used along with RNNs. The novel approach of Transformer is however, to eliminate recurrence completely and replace it with attention to handle the dependencies between input and output. The Transformer moves the sweet spot of current ideas toward attention entirely. It eliminates the not only recurrence but also convolution in favor of applying **self-attention** (a.k.a intra-attention). Additionally Transformer gives more space for parallelization (details present in paper).

The top performance in the paper is achieved while applying the **attention** mechanism connecting encoder and decoder.

Transformer is claimed by authors to be the first to rely entirely on self-attention to compute representations of input and output.

Information on processing strategy of the algorithm

Transformer is based on sequence-to-sequence model for *Statistical Machine Translation* (SMT) as introduced in *Cho et al., 2014 (https://arxiv.org/abs/1406.1078)*. It includes two RNNs, one for **encoder** to process the input and the other as a **decoder**, for generating the output.

In general, transformer's *encoder* maps input sequence to its continuous representation z which in turn is used by *decoder* to generate output, one symbol at a time.

The final state of the encoder is a fixed size vector z that must encode entire source sentence which includes the sentence meaning. This final state is therefore called *sentence embedding*[1].

The encoder-decoder model is designed at its each step to be **auto-regressive** - i.e. use previously generated symbols as extra input while generating next symbol. Thus, $x_i + y_{i-1} \rightarrow y_i$

Transformer is based on Encoder-Decoder

In Transformer (as in ByteNet or ConvS2S) the *decoder* is stacked directly on top of *encoder*. Encoder and decoder both are composed of stack of identical layers. Each of those stacked layers is composed out of two general types of sub-layers:

- multi-head self-attention mechanism, and
- position-wise fully connected FFN.

  In contrast to ConvS2S however, where input representation considered each input element combined with its absolute position number in sequence (providing sense of ordering; ByteNet have dilated convolutions and no position-wise FNN), transformer introduces two different NN for these two types of information.

The way the attention mechanism is applied and customized is what makes the Transformer novel.

One can find the reference Transformer model implementation from authors is present in *Tensor2Tensor (T2T) library (https://github.com/tensorflow/tensor2tensor/blob/master/tensor2tensor/models/transformer.py)*

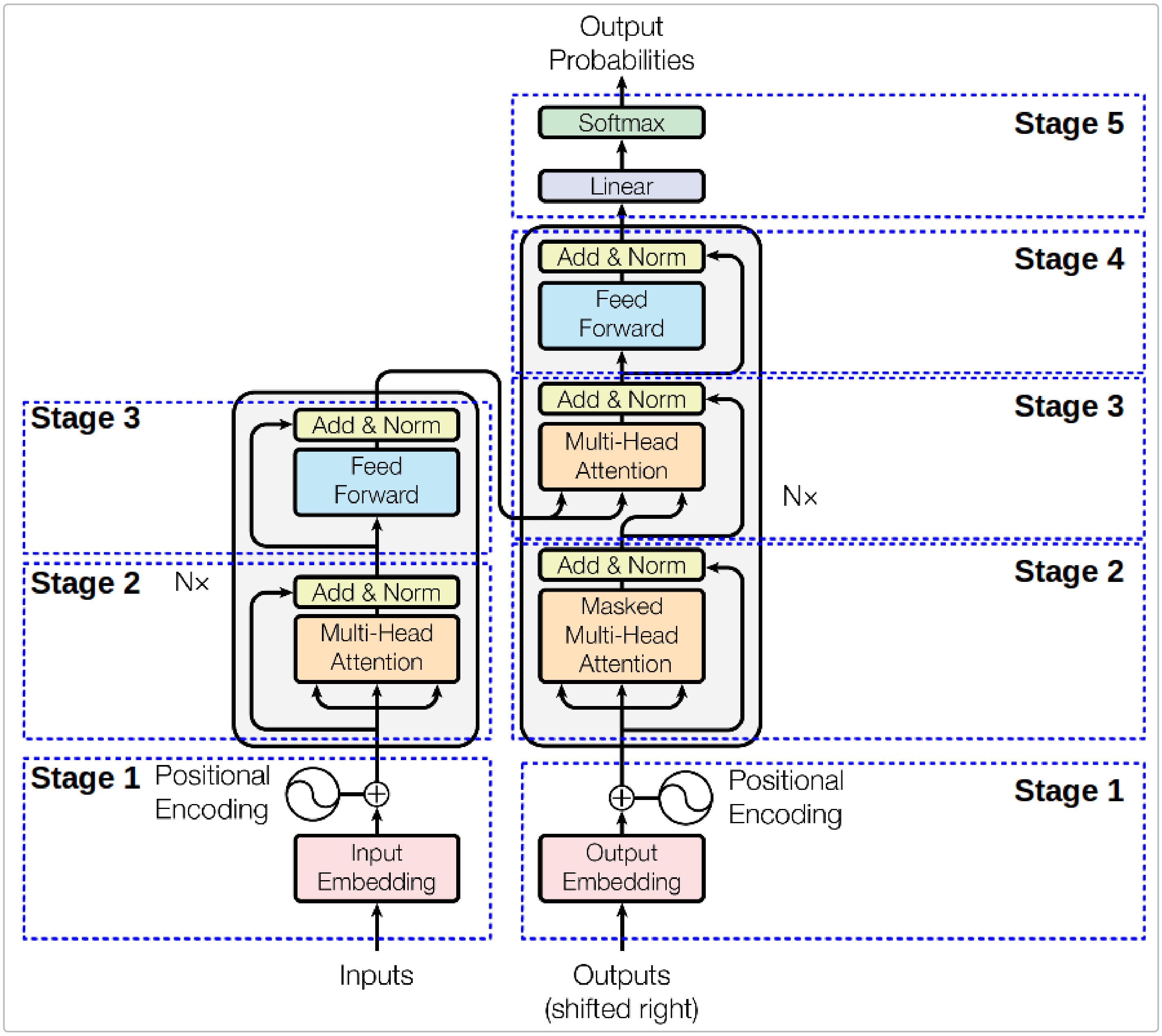

Figure 2. Single layer of Encoder (left) and Decoder (right) that is build out of $N = 6$ identical layers.

Encoder

- **Stage 1 – Encoder Input** Information on sequence ordering is very important. As there is no recurrence, nor convolution, this information on absolute (or relative) position of each token in a sequence is represented with use of "*positional encodings*" (*Read more (../../../../2017/Sep/12/Transformer-Attention-is-all-you-need/#positional-encoding-pe)*). The input for the encoder is therefore, represented as:
  - *positional encodings* added $\oplus$ to
  - *embedded inputs*

- $N = 6$ layers. In practice the $N = 6$ means more than 6 layers. Each of those "layers" are actually composed of two layers: position-wise FNN and one (encoder), or two (decoder), attention-based sublayers. Each of those additionally contains 4 linear projections and the attention logic. Thus, providing effectively deeper than 6 layer architecture.

  - **Stage 2 – Multi-head attention**
  - **Stage 3 – position-wise FFN**

  Stages 2 and 3 use the residual connection (thus, all employ $d_{model} = 512$) followed by normalization layer at its output.

Thus, encoder works like this:

```
Stage1_out = Embedding512 + TokenPositionEncoding512
Stage2_out = layer_normalization(multihead_attention(Stage1_out) + Stage1_out)
Stage3_out = layer_normalization(FFN(Stage2_out) + Stage2_out)

out_enc = Stage3_out
```

Decoder

Decoder's architecture is similar however, it employs additional layer in *Stage 3* with mask multi-head attention over encoder output.

- **Stage 1 – Decoder input** The input is the *output embedding*, offset by one position to ensure that the prediction for position i is only dependent on positions previous to/less than i.

- **Stage 2 Masked Multi-head attention** Modified to prevent positions to attend to subsequent positions.

Stages 2, 3 and 4 also use the residual connection followed by normalization layer at its output.

The details of each mechanism applied in the mentioned layers is more elaborated in following section.

Put together decoder works as follows:

```
Stage1_out = OutputEmbedding512 + TokenPositionEncoding512

Stage2_Mask = masked_multihead_attention(Stage1_out)
Stage2_Norm1 = layer_normalization(Stage2_Mask) + Stage1_out
Stage2_Multi = multihead_attention(Stage2_Norm1 + out_enc) +  Stage2_Norm1
Stage2_Norm2 = layer_normalization(Stage2_Multi) + Stage2_Multi

Stage3_FNN = FNN(Stage2_Norm2)
Stage3_Norm = layer_normalization(Stage3_FNN) + Stage2_Norm2

out_dec = Stage3_Norm
```

Mechanisms used to compose Transformer architecture

There are couple types of layers that transformer consists of. Their details are depict in following sections.

Positional Encoding – PE

In RNN (LSTM), the notion of time step is encoded in the sequence as inputs/outputs flow one at a time. In FNN, the *positional encoding* must be used to represent the time in some way. In case of the Transformer authors propose to encode time as *sine* wave, as an added extra input. Such signal is added to inputs and outputs to represent time passing[2].

In general, adding positional encodings to the input embeddings is a quite interesting topic. One way is to embed the absolute position of input elements (as in ConvS2S). However, authors use "sine and cosine functions of different frequencies". The "sinusoidal" version is quite complicated, while giving similar performance to the absolute position version. The crux is however, that it may allow the model to produce better translation on longer sentences at test time (at least longer than the sentences in the training data). This way sinusoidal method allows the model to extrapolate to longer sequence lengths[3].

This encoding gives the model a sense of which portion of the sequence of the input (or output) it is currently dealing with. The positional encoding can be learned, or fixed. Authors made tests (PPL, BLEU) showing that both: learned and fixed positional encodings perform similarly.

In paper authors have decided on fixed variant using sin and cos functions to enable the network to learn information about tokens relative positions to the sequence.

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}}) \tag{1}$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}}) \tag{2}$$

Of course authors motivate the use of sinusoidal functions due to enabling model to generalize to sequences longer than ones encountered during training.

Attention

Attention between encoder and decoder is crucial in NMT. Authors formulate the definition of *attention* that has already been elaborated in *Attention primer (../../../../2017/Sep/01/Primer-NN/#attention-basis)*. Attention is a function that maps the 2-element input (*query*, *key-value* pairs) to an output. The output given by the mapping function is a weighted sum of the *values*. Where weights for each *value* measures how much each input *key* interacts with (or answers) the *query*. While the attention is a goal for many research, the novelty about transformer attention is that it is **multi-head self-attention**.

Scaled Dot-Product Attention

In terms of encoder-decoder, the **query** is usually the hidden state of the *decoder*. Whereas **key**, is the hidden state of the *encoder*, and the corresponding **value** is normalized weight, representing how much attention a *key* gets. Output is calculated as a wighted sum – here the dot product of *query* and *key* is used to get a *value*.

It is assumed that *queries* and *keys* are of d_k dimension and *values* are of d_v dimension. Those dimensions are imposed by the linear projection discussed in the multi-head attention section. The input is represented by three matrices: queries' matrix Q, keys' matrix K and values' matrix V.

The *compatibility function* (see *Attention primer (../../../../2017/Sep/01/Primer-NN/#attention-basis)*) is considered in terms of two, *additive* and *multiplicative* (dot-product) variants *Bahdanau et al. 2014 (https://arxiv.org/abs/1409.0473)* with similar theoretical complexity. However, the dot-product ($q \cdot k = \sum_{i=1}^{d_k} q_i k_i$) with scaling factor $1/\sqrt{d_k}$ is chosen due to being much faster and space-efficient, as it uses optimized matrix multiplication code[4].

$$Attention(Q, K, V) = softmax\Big(\frac{QK^T}{\sqrt{d_k}}\Big)V \tag{3}$$

Using NumPy:

```
def attention(Q, K, V):
        num = np.dot(Q, K.T)
        denum = np.sqrt(K.shape[0])
        return np.dot(softmax(num / denum), V)
```

Multi-head attention

Transformer reduces the number of operations required to relate (especially distant) positions in input and output sequence to a $O(1)$. However, this comes at cost of reduced effective resolution because of averaging attention-weighted positions.

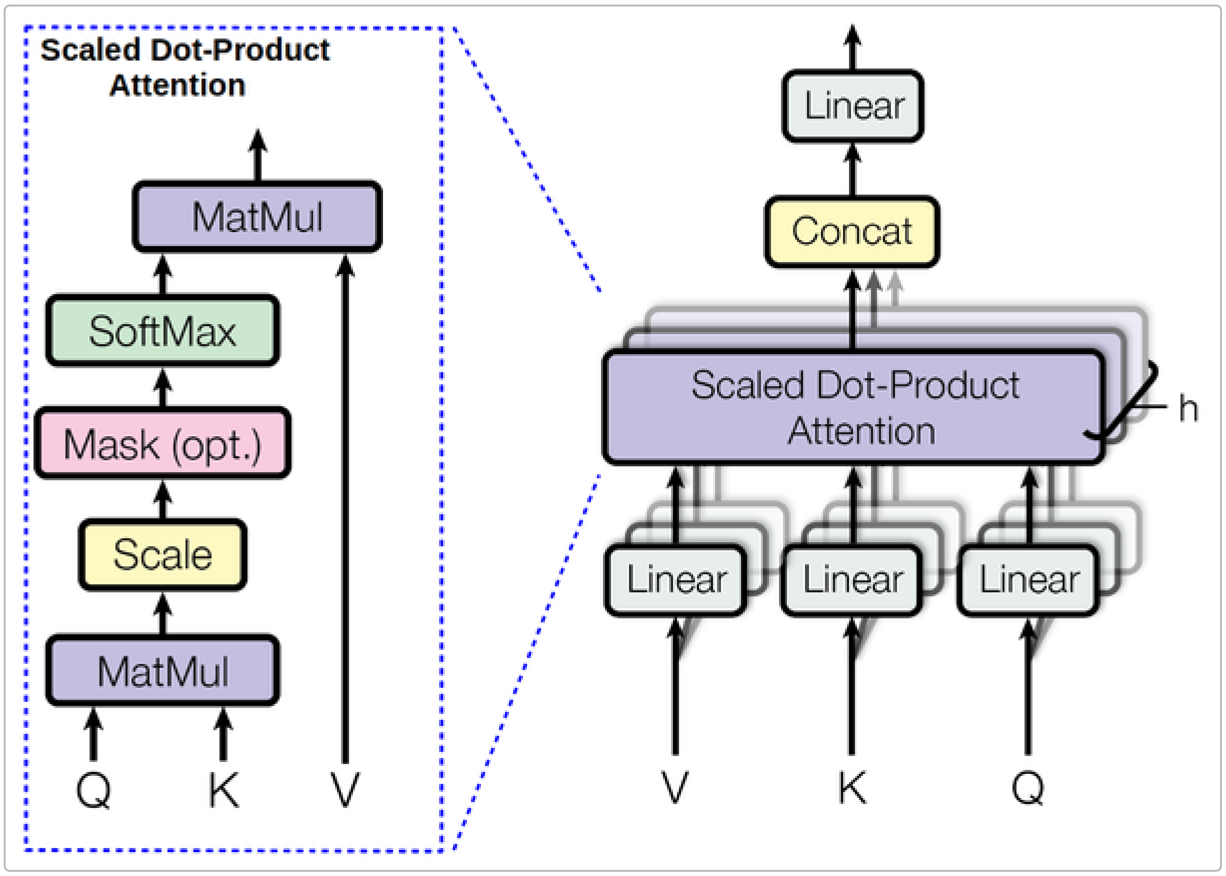

Figure 3. Multi-Head Attention consists of h attention layers running in parallel.

To reduce this cost authors propose the multi-head attention:

- $h = 8$ attention layers (aka "heads"): that represent linear projection (for the purpose of dimension reduction) of key K and query Q into d_k-dimension and value V into d_v-dimension:

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V), i = 1, \dots, h$$

  where projections are parameter matrices $W_i^Q, W_i^K \in \mathbb{R}^{d_{model} \times d_k}, W_i^V \in \mathbb{R}^{d_{model} \times d_v}$, for $d_k = d_v = d_{model}/h = 64$

- scaled-dot attention applied in parallel on each layer (different linear projections of k, q, v) results in d_v-dimensional output.

- concatenate outputs of each layer (different linear projection; also referred as *"head"*): $Concat(head_1, \ldots, head_h)$
- linearly project the concatenation result form the previous step:

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

where $W^0 \in \mathbb{R}^{d_{hd_v} \times d_{model}}$

Transformer use multi-head (d_{model}/h parallel attention functions) attention instead of single (d_{model}-dimensional) attention function (i.e. q, k, v all d_{model}-dimensional). It is at similar computational cost as in the case of single-head attention due to reduced dimensions of each head.

Transformer imitates the classical attention mechanism (known e.g. from *Bahdanau et al., 2014 (https://arxiv.org/abs/1409.0473)* or Conv2S2) where in encoder-decoder attention layers *queries* are form previous decoder layer, and the (memory) *keys* and *values* are from output of the encoder. Therefore, each position in decoder can attend over all positions in the input sequence.

Self-Attention (SA)

See *Attention Primer (../../../../2017/Sep/01/Primer-NN/#attention-basis)* for basics on attention.

In *encoder*, self-attention layers process input $queries$, $keys$ and $values$ that comes form same place i.e. the output of previous layer in encoder. Each position in encoder can attend to all positions from previous layer of the encoder

In *decoder*, self-attention layer enable each position to attend to all previous positions in the decoder, including the current position. To preserve auto-regressive property, the leftward information flow is presented inside the dot-product attention by masking out (set to $-\infty$) all $values$ that are input for softmax which correspond to this illegal connections.

Authors motivates the use of self-attention layers instead of recurrent or convolutional layers with three desiderata:

1. Minimize total computational complexity per layer

    - **Pros:** self-attention layers connects all positions with $O(1)$ number of sequentially executed operations (eg. vs $O(n)$ in RNN)

2. Maximize amount of parallelizable computations, measured by minimum number of sequential operations required

- **Pros:** for sequence length n < representation dimensionality d (true for SOTA sequence representation models like *word-piece, byte-pair*). For very long sequences $n > d$ self-attention can consider only neighborhood of some size r in the input sequence centered around the respective output position, thus increasing the max path length to $O(n/r)$

3. Minimize maximum path length between any two input and output positions in network composed of the different layer types . The shorter the path between any combination of positions in the input and output sequences, the easier to learn long-range dependencies. (See why *Hochreiter et al, 2001 (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.24.7321)*)

Position-wise FFN

In encoder and decoder the attention sublayers is being processed by a fully connected FNN. It is applied to each position separately and identically meaning two linear transformations and a ReLU

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$$

Linear transformations are the same for each position, but use different parameters from layer to layer. It works similarly to two convolutions of kernel size 1. The input/output dimension is $d_{model} = 512$ while inner0layer is $d_{ff} = 2048$

Pseudocode or flowchart description of the algorithm.

Figure 4. Transformer step-by-step sequence transduction in form of English-to-French translation. Adopted from *Google Blog (https://research.googleblog.com/2017/08/transformer-novel-neural-network.html)*

In encoder phase (shown in the Figure 1.), transformer first generates initial representation/embedding for each word in input sentence (empty circle). Next, for each word, self-attention aggregates information form all other words in context of sentence, and creates new representation (filled circles). The process is repeated for each word in sentence. Successively building new representations, based on previous ones is repeated multiple times and in parallel for each word (next layers of filled circles).

Decoder acts similarly generating one word at a time in a left-to-right-pattern. It attends to previously generated words of decoder and final representation of encoder.

It is worth noting that this self-attention strategy allows to face the issue of **coreference resolution** where e.g. word "*it*" in a sentence can refer to different noun of the sentence depending on context.

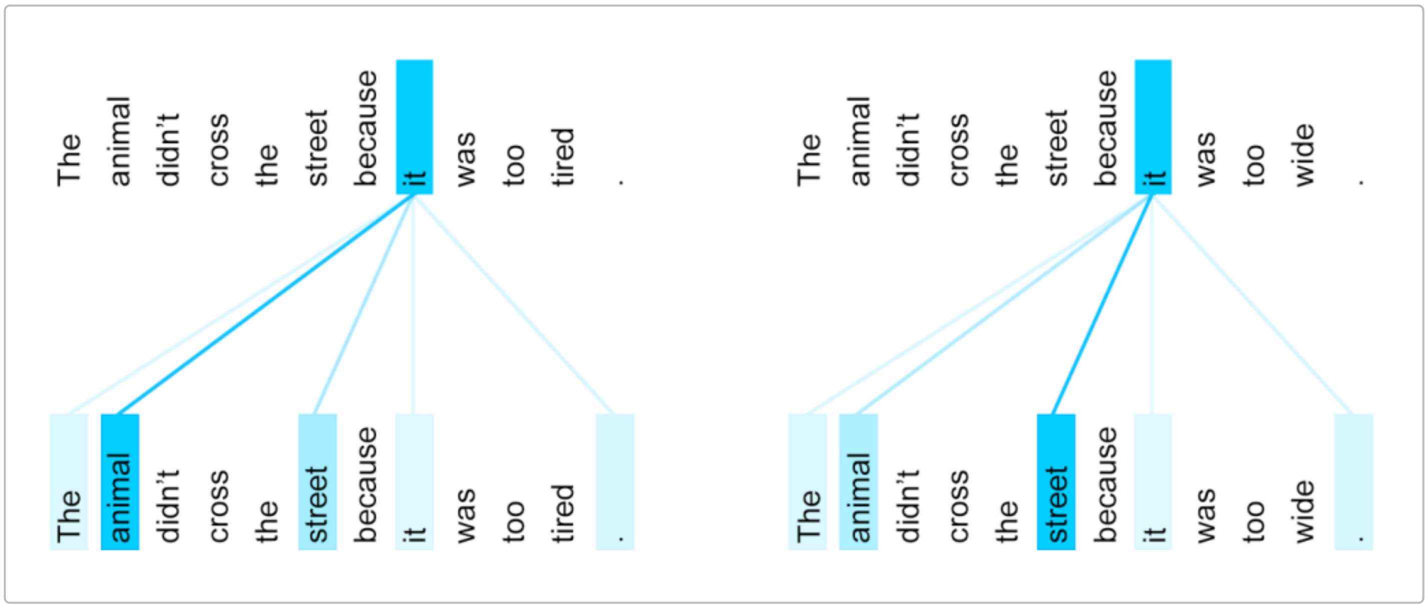
The animal didn't cross the street because it was too tired .
The animal didn't cross the street because it was too tired .
The animal didn't cross the street because it was too wide .
The animal didn't cross the street because it was too wide .

Figure 5. Co-reference resolution. The *it* in both cases relates to different token. Adopted from *Google Blog (https://research.googleblog.com/2017/08/transformer-novel-neural-network.html)*

Heuristics or rules of thumb.

Authors have conducted a series of tests (Table 3 of the paper) where they discuss recommendation of $N = 6$ layers with model size 512 based on $h = 8$ heads with key, values dimensions of 64 using 100K steps.

It is also stated that dot-product compatibility function might be further optimized due to model quality is decreased with smaller d_k (row B).

The proposed, fixed sinusoidal positional encodings are claimed to produce nearly equal score comparing to learned positional encodings.

What classes of problem is the algorithm well suited?

- sequence transduction (language translation)
- classic language analysis task of syntactic constituency parsing
- different inputs and outputs modalities, such as images and video
- coreference resolution

Common benchmark or example datasets used to demonstrate the algorithm.

- Perplexity (PPL) BLEU
- English-to-German translation development set WMT 2014 English-to-German and WMT 2014 English-to-French translation tasks
- newstest2013
- English constituency parsing

Useful resources for learning more about the algorithm.

- *Google blog post1 (https://research.googleblog.com/2017/08/transformer-novel-neural-network.html),*
- *Google blog post2 (https://research.googleblog.com/2017/06/accelerating-deep-learning-research.html)*

There are also some more general learning open-source software frameworks for NMT: *Tensorflow: Neural Machine Translation (seq2seq) Tutorial (https://github.com/tensorflow/nmt) Ground-up explaining NMT concepts tutorial with code, based on Tensorflow by Google Research. Based on Google Research blog. (https://research.googleblog.com/2017/07/building-your-own-neural-machine.html)*
Torch: FirSeq (https://github.com/facebookresearch/fairseq) seq2seq and ConvS2S from Facebook AI. * *Tensorflow: Tensor2Tensor (https://github.com/tensorflow/tensor2tensor)* From used by Transformer authors.

Thoughts on the idea.

To my limited knowledge there are some statements that might benefit form more explanation:

1. How the scaling factor (Equation 3) makes an impact?
2. How actually the **positional encoding** work? Why they have chosen the sin/cos functions and why the position and dimension are in this relation? Finally how sinusoidal helps translate long sentences?
3. Does having separate position-wise FFNs help? (comparing to ConvS2S).
4. The *"cost of reduced effective resolution due to averaging attention-weighted position"* is claimed to be a motivation for multi-head attention. How to understand better what is the issue and how multi-head attention helps?
5. The Transformer brings a significantly improvement over ConvS2S, but where does the improvement come from? It is not clear from the work. ConvS2S lacks the self-attention, is it what brings the advantage?
6. Masked Attention. The problem of using same parts of input on different decoding step is claimed to be solved by penalizing (mask-out to $-\infty$) input tokens that have obtained high attention scores in the past decoding steps – a bit vague. How does it work? Maybe explicitly having a position-wise FFN automatically fixes that problem?
7. Applying multi-head attention might improve performance due to better parallelization. However, Table 3 also show increasing $h = 1to8$ improves accuracy. Why? Moving h to 16 or 32 is not that beneficial. How to interpret this correctly?

8. How important the autoregression is in context of this architecture?

Please leave a comment if you have any other question, or would like to get more explanation on any of the paper's particularities.

Primary references or resources in which the algorithm was first described.

- Paper: *ArXiv (https://arxiv.org/abs/1706.03762)*

Some interesting future research

- Devising more sophisticated compatibility function
- Increase maximum path length to $O(n/r)$, where r would be only a neighborhood size of positions to be considered by self-attention instead of all positions in sequence

References:

1. While applying dimensionality reduction techniques (e.g. PCA, t-SNE) on embeddings, the outcome plot gathers the semantically close sentences together *Sutskever et al., 2014 (https://arxiv.org/abs/1409.3215)*. ↩
2. It is interesting how this resembles the brain waves and *neural oscillations (https://en.wikipedia.org/wiki/Neural_oscillation)*. ↩
3. It reminds a bit the Pointer Networks that address similar problem ↩
4. The additional scaling factor is advised for large d_k where dot product grow large in magnitude, as softmax is suspected to be pushed to vanishing gradient area, thus making the additive attention perform better. ↩

Comments

About Michał Chromiak
Completed his PhD in 2016 by Polish Academy of Sciences (PAS) in Computer Science. Focus research on understanding chaos of data. Deeply understanding the phenomena makes it easy, but first you need to learn. Holds two MScs, in Mathematics and in Computer Science.

Social

- LinkedIn (https://www.linkedin.com/in/michal-chromiak)
- GitHub (https://github.com/MichalChromiak)

Twitter (https://twitter.com/drChromiak)

ResearchGate (https://www.researchgate.net/profile/Michal_Chromiak)

Google Scholar (https://scholar.google.pl/citations?user=UeOad3YAAAAJ&hl=en)

RSS (localhost:8000/feeds/all.rss)

Recent Posts

ERNIE 2.0: A continual pre-training framework for language understanding (/articles/2019/Jul/30/ernie-2-0/)

NLP: Explaining Neural Language Modeling (/articles/2017/Nov/30/Explaining-Neural-Language-Modeling/)

The Transformer – Attention is all you need. (/articles/2017/Sep/12/Transformer-Attention-is-all-you-need/)

Categories

Applications (/category/applications.html)

ML Dojo (/category/ml-dojo.html)

Sequence Models (/category/sequence-models.html)

Tags

(/)

application (/tag/application.html) Attention model (/tag/attention-model.html) basics (/tag/basics.html) BERT (/tag/bert.html) ELMo (/tag/elmo.html) ERNIE 1.0 (/tag/ernie-10.html) language model (/tag/language-model.html) Machine translation (/tag/machine-translation.html) ngram (/tag/ngram.html) NLP (/tag/nlp.html) NMT (/tag/nmt.html) OpenAI GPT (/tag/openai-gpt.html) PatternRecognition (/tag/patternrecognition.html) perplexity (/tag/perplexity.html) seq2seq (/tag/seq2seq.html) Sequence transduction (/tag/sequence-transduction.html) smoothing (/tag/smoothing.html) transformer (/tag/transformer.html) XLNet (/tag/xlnet.html)

Links

ICLR Conf (http://www.iclr.cc)

ICML Conf (http://icml.cc)

NIPS Conf (https://nips.cc/)

AI Frontiers (http://aifrontiers.com/)

ML Glossary (https://developers.google.com/machine-learning/glossary/)

Deep Dream Generator (https://deepdreamgenerator.com/)

DeepArt Generator (https://deepart.io/)

Stanford ML Group Andrew Ng (https://stanfordmlgroup.github.io/)

AI•ON open ML collaboration (https://ai-on.org/)

My old blog on Java an SE (http://java-hive.blogspot.com/)

PhD (http://karpathy.github.io/2016/09/07/phd/)

SemEval2017 (http://alt.qcri.org/semeval2017/)

Free CS courses (https://medium.freecodecamp.org/450-free-online-programming-computer-science-courses-you-can-start-in-september-59712e77635c)

Archive

July 2019 (1) (/archive/2019/Jul/index.html)

November 2017 (1) (/archive/2017/Nov/index.html)

September 2017 (2) (/archive/2017/Sep/index.html)

August 2017 (1) (/archive/2017/Aug/index.html)

Back to top