Seq2seq, NMT, Transformer

Milan Straka

April 22, 2024

EUROPEAN UNION
European Structural and Investment Fund
Operational Programme Research,
Development and Education

Charles University in Prague
Faculty of Mathematics and Physics
Institute of Formal and Applied Linguistics

CC BY SA
unless otherwise stated

Sequence-to-Sequence Architecture

Sequence-to-Sequence Architecture

Sequence-to-Sequence is a name for an architecture allowing to produce an arbitrary output sequence $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_M$ from an input sequence $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$.

Unlike span labeling/CTC, no assumptions are necessary and we condition each output sequence element on all input sequence elements and all already generated output sequence elements:

$$P\big(\boldsymbol{y} \mid \boldsymbol{X}\big) = \prod_{i=1}^{M} P\big(y_i \mid \boldsymbol{x}_1, \ldots, \boldsymbol{x}_N, y_1, \ldots, y_{i-1}\big).$$

Sequence-to-Sequence Architecture

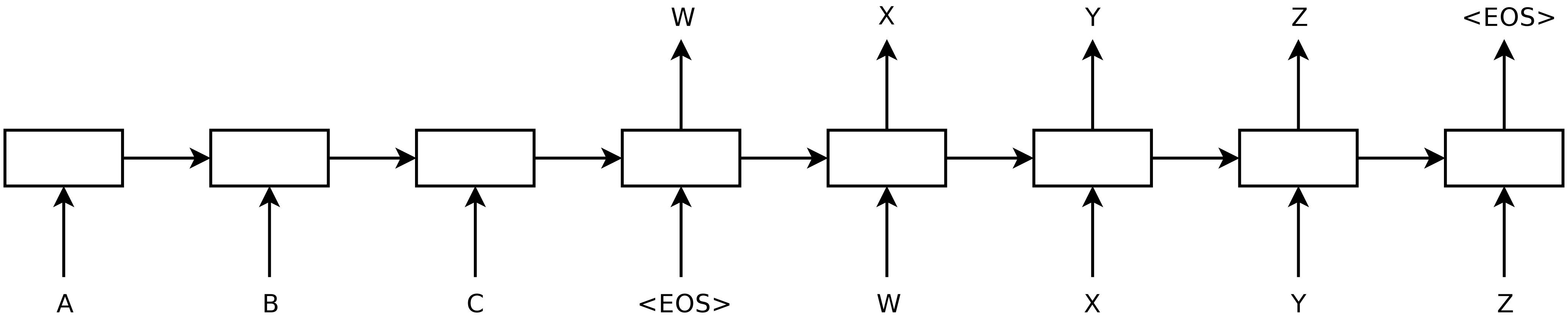


Figure 1 of "Sequence to Sequence Learning with Neural Networks", https://arxiv.org/abs/1409.0473

Sequence-to-Sequence Architecture

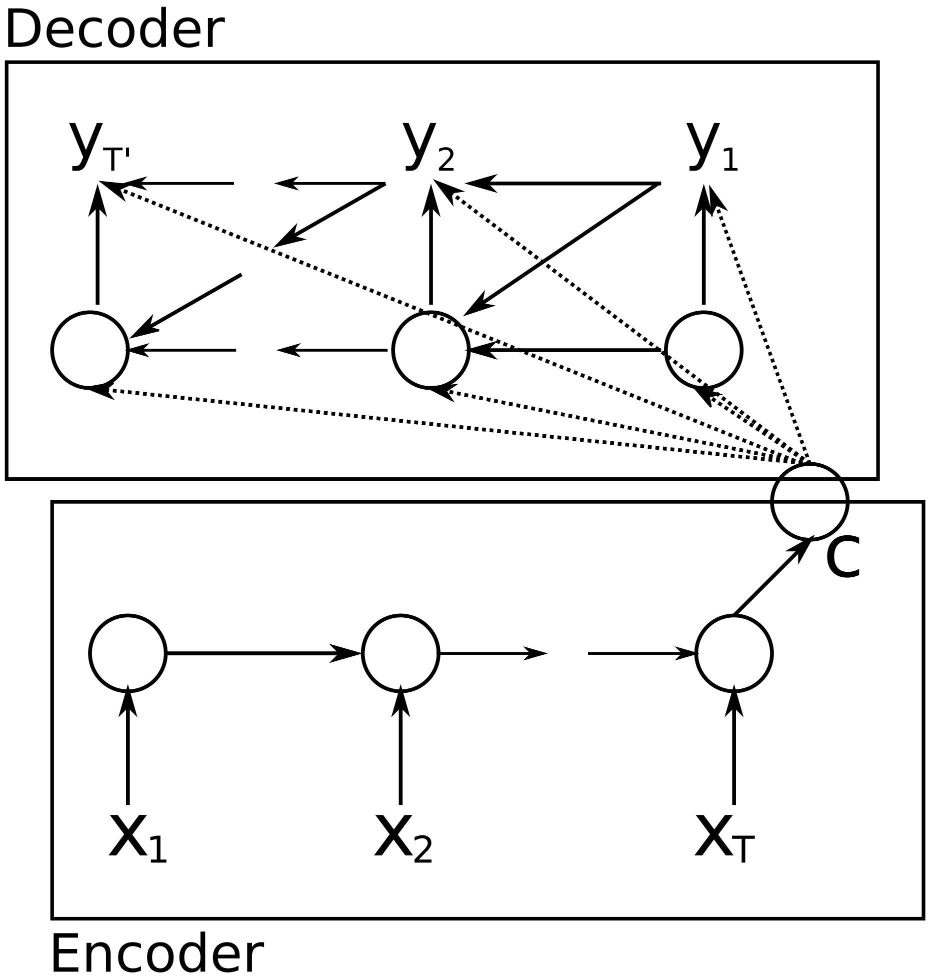
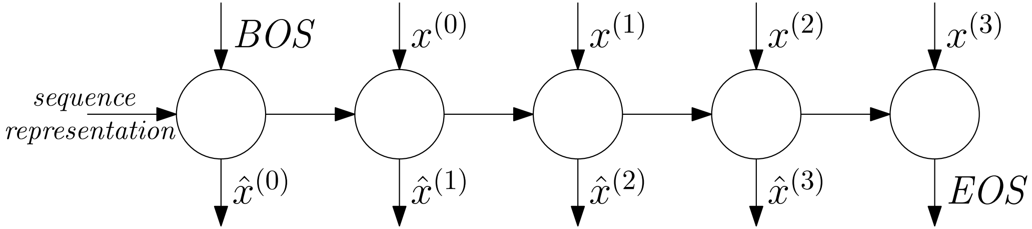

Decoder
$y_{T'}$
y_2
y_1
c
x_1
x_2
x_T
Encoder

Figure 1 of "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation", https://arxiv.org/abs/1406.1078

Sequence-to-Sequence Architecture

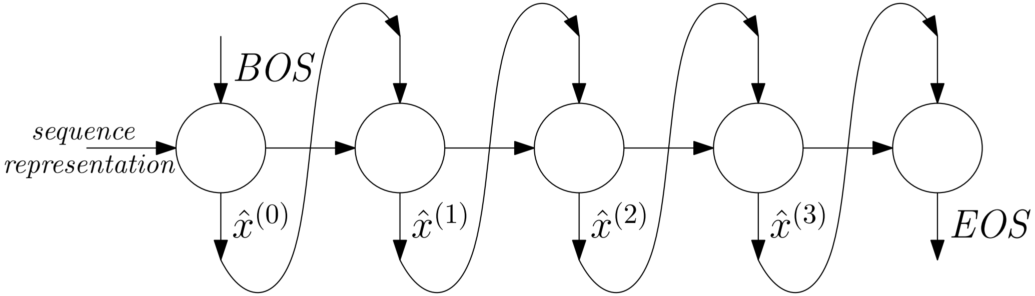
Training

The so-called **teacher forcing** is used during training – the gold outputs are used as inputs during training.


BOS
$x^{(0)}$
$x^{(1)}$
$x^{(2)}$
$x^{(3)}$
sequence representation
$\hat{x}^{(0)}$
$\hat{x}^{(1)}$
$\hat{x}^{(2)}$
$\hat{x}^{(3)}$
EOS

Inference

During inference, the network processes its own predictions – such an approach is called **autoregressive decoding**.


BOS
sequence representation
$\hat{x}^{(0)}$
$\hat{x}^{(1)}$
$\hat{x}^{(2)}$
$\hat{x}^{(3)}$
EOS

Usually, the generated logits are processed by an $\arg\max$, the chosen word embedded and used as next input.

Tying Word Embeddings

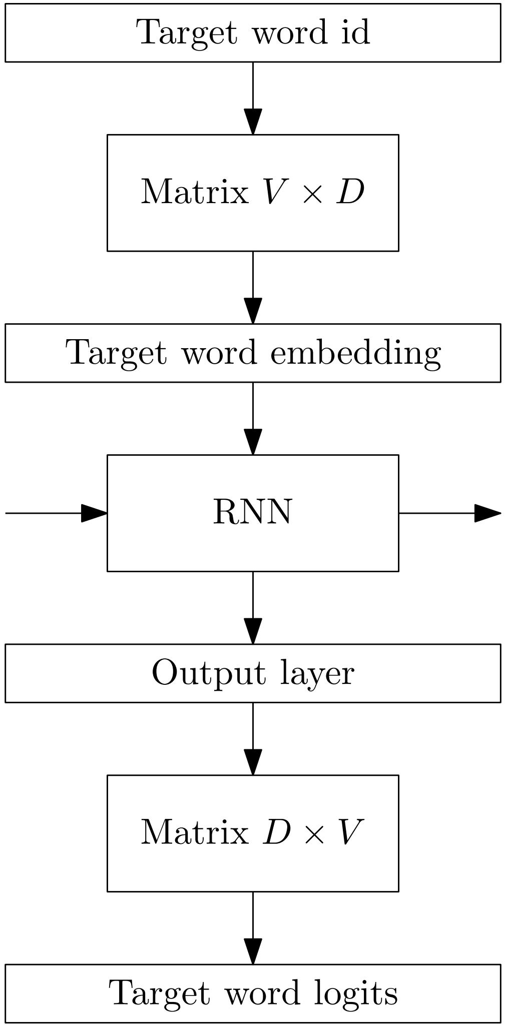
In the decoder, we both:

- embed the previous prediction, using a matrix of size $\mathbb{R}^{V \times D}$, where V is the vocabulary size and D is the embedding size;
- classify the hidden state into current prediction, using a matrix of size $\mathbb{R}^{D \times V}$.

Both these matrices have similar meaning – they represent words in the embedding space (the first explicitly represents words by the embeddings, the second produces logits by computing weighted cosine similarity of the inputs and columns of the weight matrix).

Therefore, it makes sense to **tie** these matrices, i.e., to represent one of them as a transposition of the other.

- However, while the embedding matrix should usually have constant variance per dimension, the output layer should keep the variance of the RNN output; therefore, the output layer matrix is usually the embedding matrix divided by $\sqrt{D}$.


Target word id

Matrix $V \times D$

Target word embedding

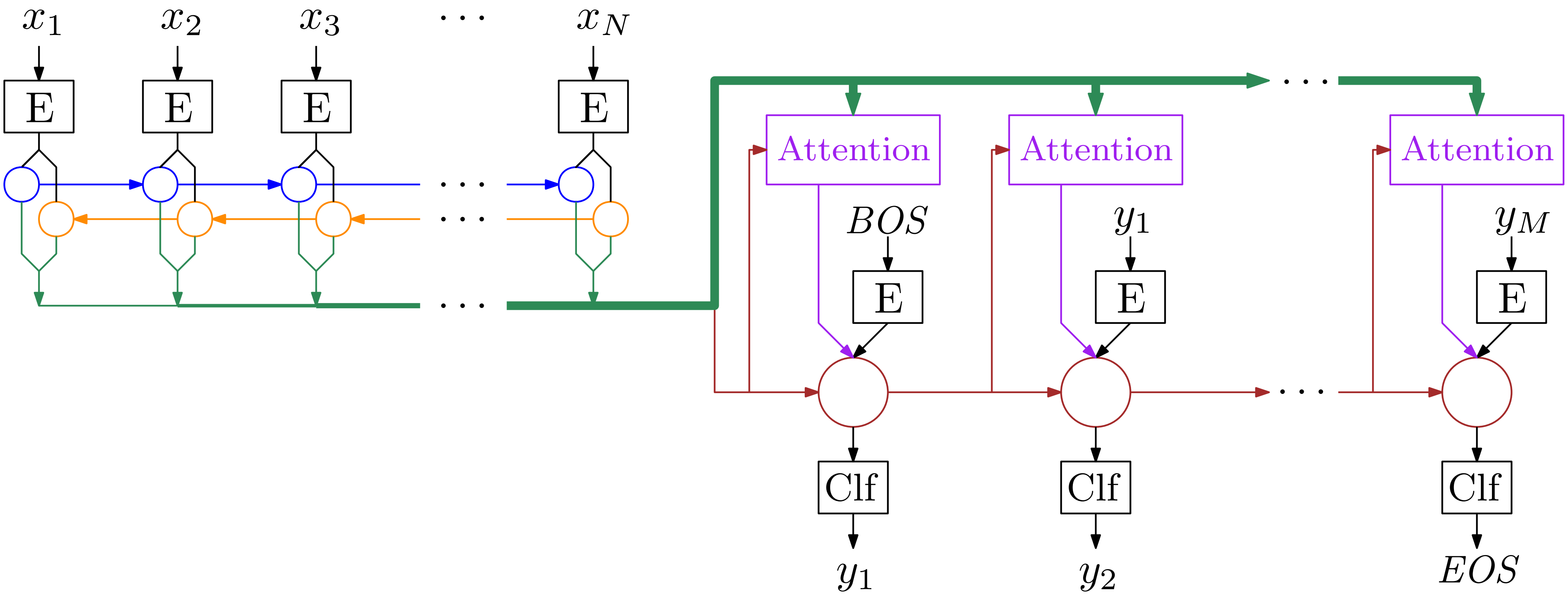
RNN

Output layer

Matrix $D \times V$

Target word logits

Attention


x_1
x_2
x_3
…
x_N
E
E
E
E
Attention
Attention
Attention
BOS
y_1
y_M
E
E
E
Clf
Clf
Clf
y_1
y_2
EOS

Bahdanau Attention

As another input during decoding, we add *context vector* c_i:

$$\boldsymbol{s}_i = f(\boldsymbol{s}_{i-1}, \boldsymbol{y}_{i-1}, \boldsymbol{c}_i).$$

We compute the context vector as a weighted combination of source sentence encoded outputs:

$$\boldsymbol{c}_i = \sum_j \alpha_{ij} \boldsymbol{h}_j$$

The weights α_{ij} are softmax of e_{ij} over j,

$$\boldsymbol{\alpha}_i = \operatorname{softmax}(\boldsymbol{e}_i),$$

with e_{ij} being

$$e_{ij} = \boldsymbol{v}^\top \tanh(\boldsymbol{V}\boldsymbol{h}_j + \boldsymbol{W}\boldsymbol{s}_{i-1} + \boldsymbol{b}).$$

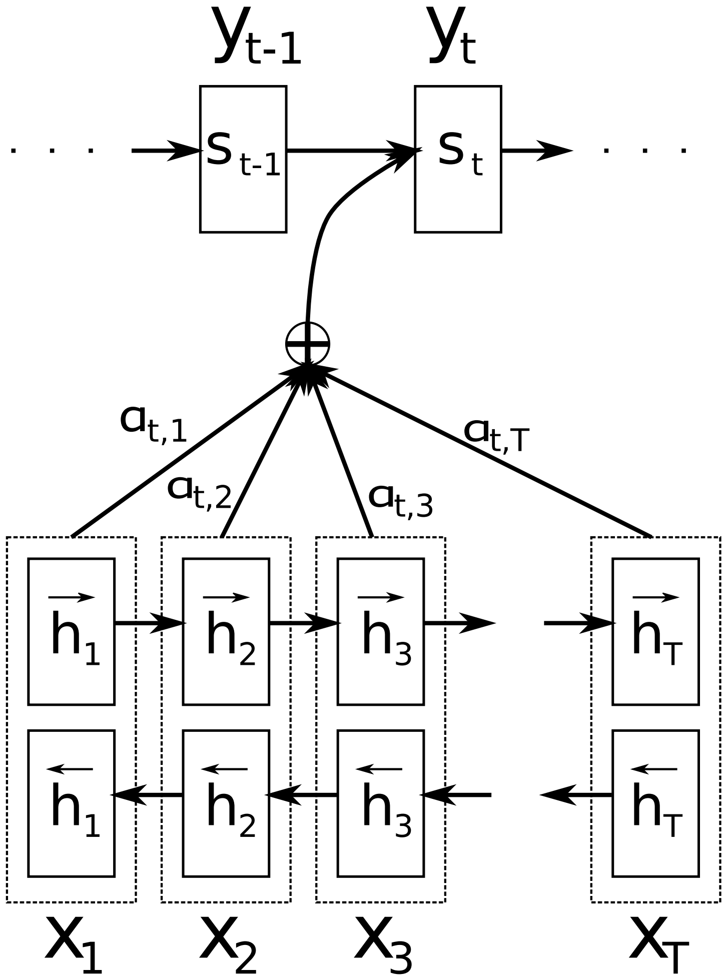
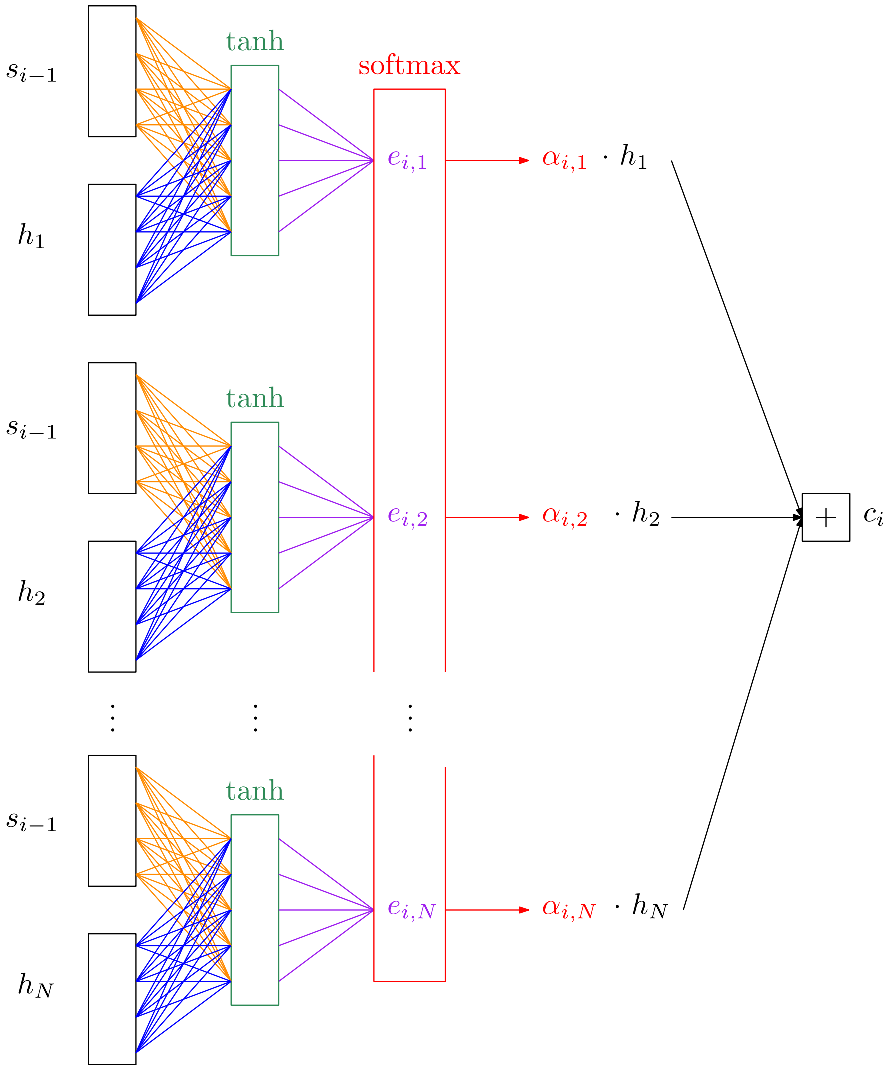

y_{t-1}
y_t
s_{t-1}
s_t
$\alpha_{t,1}$
$\alpha_{t,2}$
$\alpha_{t,3}$
$\alpha_{t,T}$
$\overrightarrow{h}_1$
$\overrightarrow{h}_2$
$\overrightarrow{h}_3$
$\overrightarrow{h}_T$
$\overleftarrow{h}_1$
$\overleftarrow{h}_2$
$\overleftarrow{h}_3$
$\overleftarrow{h}_T$
x_1
x_2
x_3
x_T

Figure 1 of "Neural Machine Translation by Jointly Learning to Align and Translate", https://arxiv.org/abs/1409.0473

Bahdanau Attention Implementation


s_{i-1}
h_1
tanh
softmax
$e_{i,1}$
$\alpha_{i,1} \cdot h_1$
s_{i-1}
h_2
tanh
$e_{i,2}$
$\alpha_{i,2} \cdot h_2$
$+$
c_i
s_{i-1}
h_N
tanh
$e_{i,N}$
$\alpha_{i,N} \cdot h_N$

Trained Attention Visualization

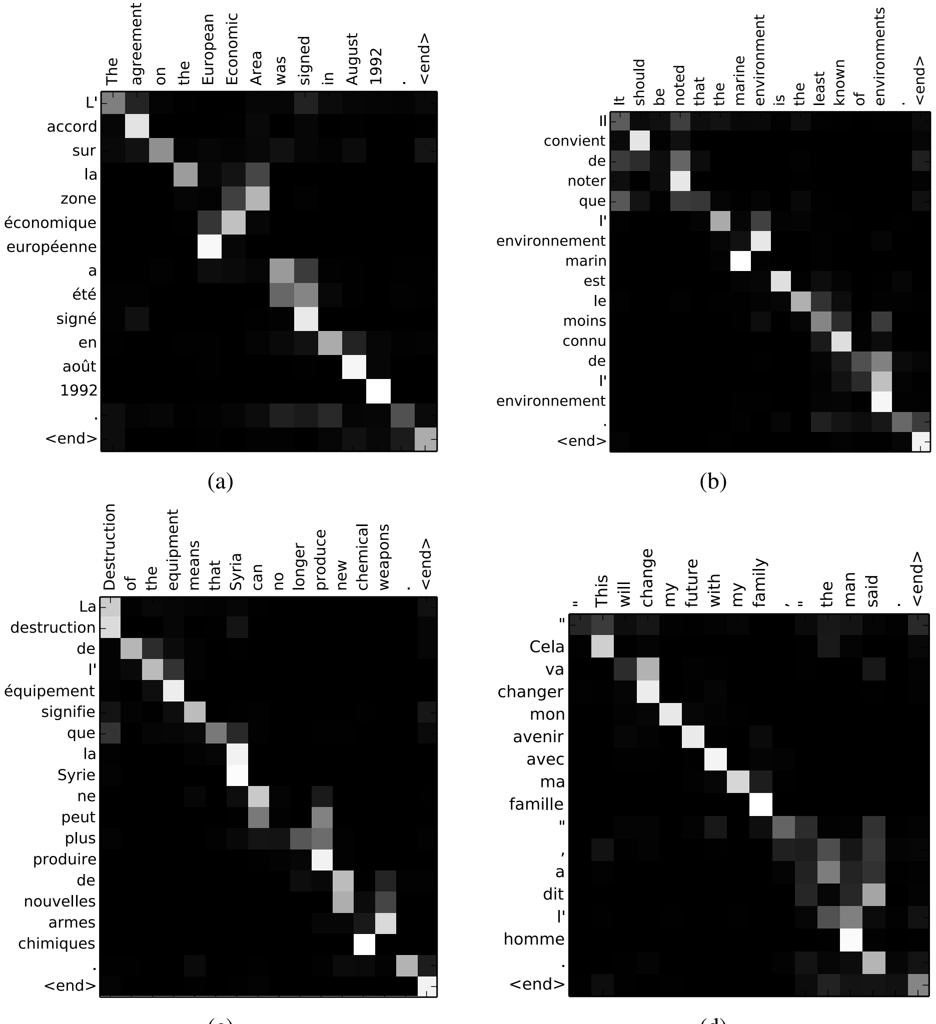


Figure 3 of "Neural Machine Translation by Jointly Learning to Align and Translate", https://arxiv.org/abs/1409.0473

Luong Attention

In the described *Bahdanau* (or *additive*) attention, we performed

$$e_{ij} = \boldsymbol{v}^\top \tanh(\boldsymbol{V}\boldsymbol{h}_j + \boldsymbol{W}\boldsymbol{s}_{i-1} + \boldsymbol{b}).$$

There are however other methods how $\boldsymbol{V}\boldsymbol{h}_j$ and $\boldsymbol{W}\boldsymbol{s}_{i-1}$ can be combined, most notably the *Luong* (or *dot-product*) attention, which uses just a dot product:

$$e_{ij} = \big(\boldsymbol{V}\boldsymbol{h}_j\big)^T \big(\boldsymbol{W}\boldsymbol{s}_{i-1}\big).$$

The latter is easier to implement, but may sometimes be more difficult to train (scaling helps a bit, wait for the Transformer self-attention description); both approaches are used in quite a few papers.

Subword Units

Translate **subword units** instead of words. The subword units can be generated in several ways, the most commonly used are:

- **BPE**: Using the *byte pair encoding* algorithm. Start with individual characters plus a special end-of-word symbol •. Then, merge the most occurring symbol pair A, B by a new symbol AB, with the symbol pair never crossing word boundary (so that the end-of-word symbol cannot be inside a subword).

  Considering text with words *low, lowest, newer, wider*, a possible sequence of merges:

$$\begin{aligned} r \;\; \bullet &\rightarrow r\bullet \\ l \;\; o &\rightarrow lo \\ lo \;\; w &\rightarrow low \\ e \;\; r\bullet &\rightarrow er\bullet \end{aligned}$$

  The BPE algorithm is executed on the training data, and it generates the resulting dictionary, merging rules, and training data encoded using this dictionary.

Subword Units

- **Wordpieces**: Given a text divided into subwords, we can compute unigram probability of every subword, and then get the likelihood of the text under a unigram language model by multiplying the probabilities of the subwords in the text.

  When we have only a text and a subword dictionary, we divide the text in a greedy fashion, iteratively choosing the longest existing subword.

  When constructing the subwords, we again start with individual characters (compared to BPE, we have a *start-of-word* character instead of an *end-of-word* character), and then repeatedly join such a pair of subwords that increases the unigram language model likelihood the most.
  - In the original implementation, the input data were once in a while "reparsed" (retokenized) in a greedy fashion with the up-to-date dictionary. However, the recent implementations do not seem to do it – but they retokenize the training data with the final dictionary, contrary to the BPE approach.

For both approaches, usually quite little subword units are used (32k-64k), often generated on the union of the two vocabularies of the source and target languages (the so-called *joint BPE* or *shared wordpieces*).

BPE and WordPieces Comparison

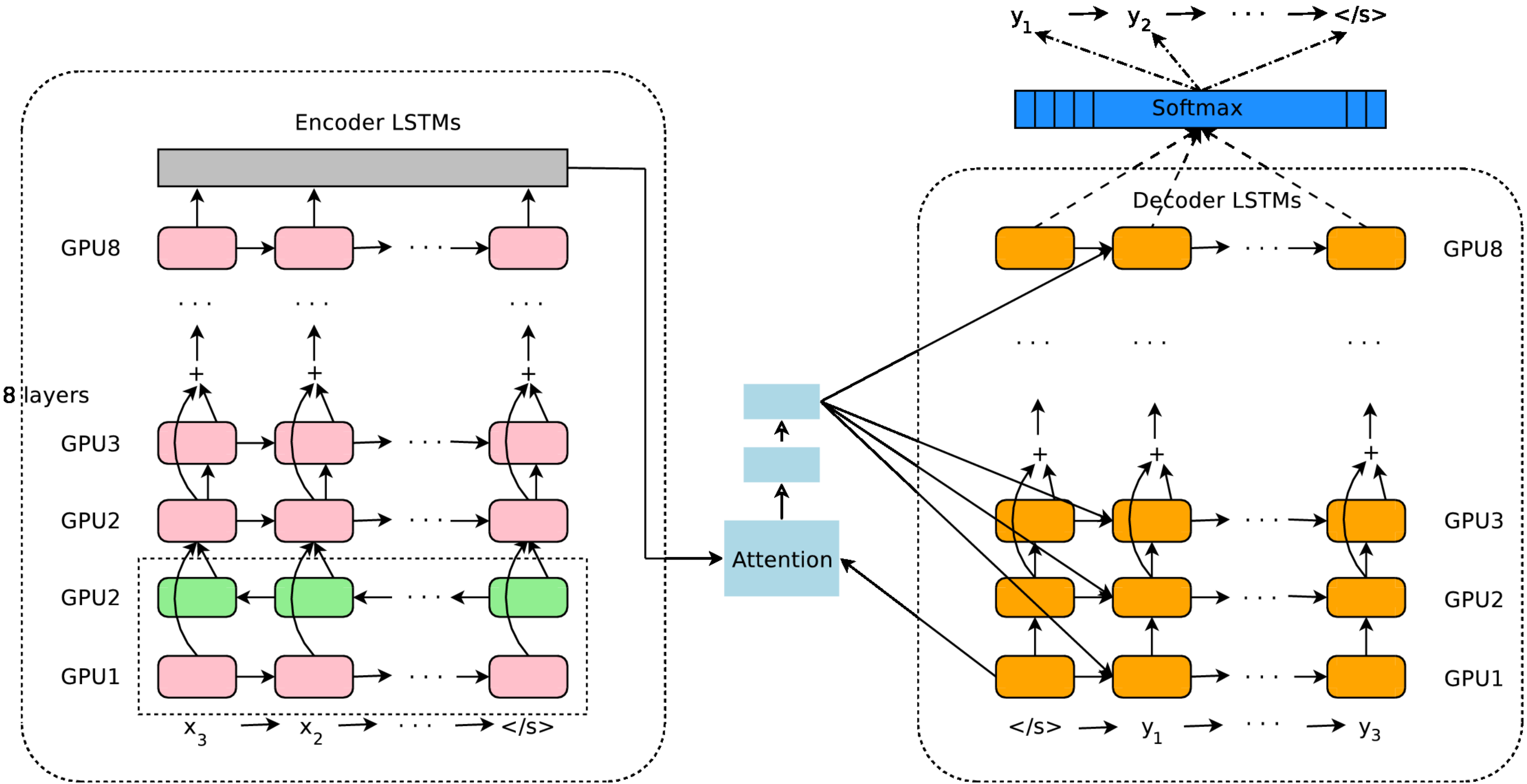
Both the BPE and the WordPieces give very similar results; the biggest difference is that during the inference:

- for BPE, the sequence of merges must be performed in the same order as during the construction of the BPE (because we use the output of BPE as training data),
- for Wordpieces, it is enough to find longest matches from the subword dictionary (because we reprocessed the training data with the final dictionary);
- note that the above difference is mostly artificial – if we reparsed the training data in the BPE approach, we could also perform "greedy tokenization".

Of course, the two algorithms also differ in the way how they choose the pair of subwords to merge.

Both algorithms are implemented in quite a few libraries, most notably the `sentencepiece` library and the Hugging Face `tokenizers` package.

Google NMT



y1 → y2 → · · · → </s>
Softmax
Encoder LSTMs
Decoder LSTMs
8 layers
GPU8
GPU3
GPU2
GPU2
GPU1
GPU8
GPU3
GPU2
GPU1
Attention
x3 → x2 → · · · → </s>
</s> → y1 → · · · → y3

Figure 1 of "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation", https://arxiv.org/abs/1609.08144

Google NMT

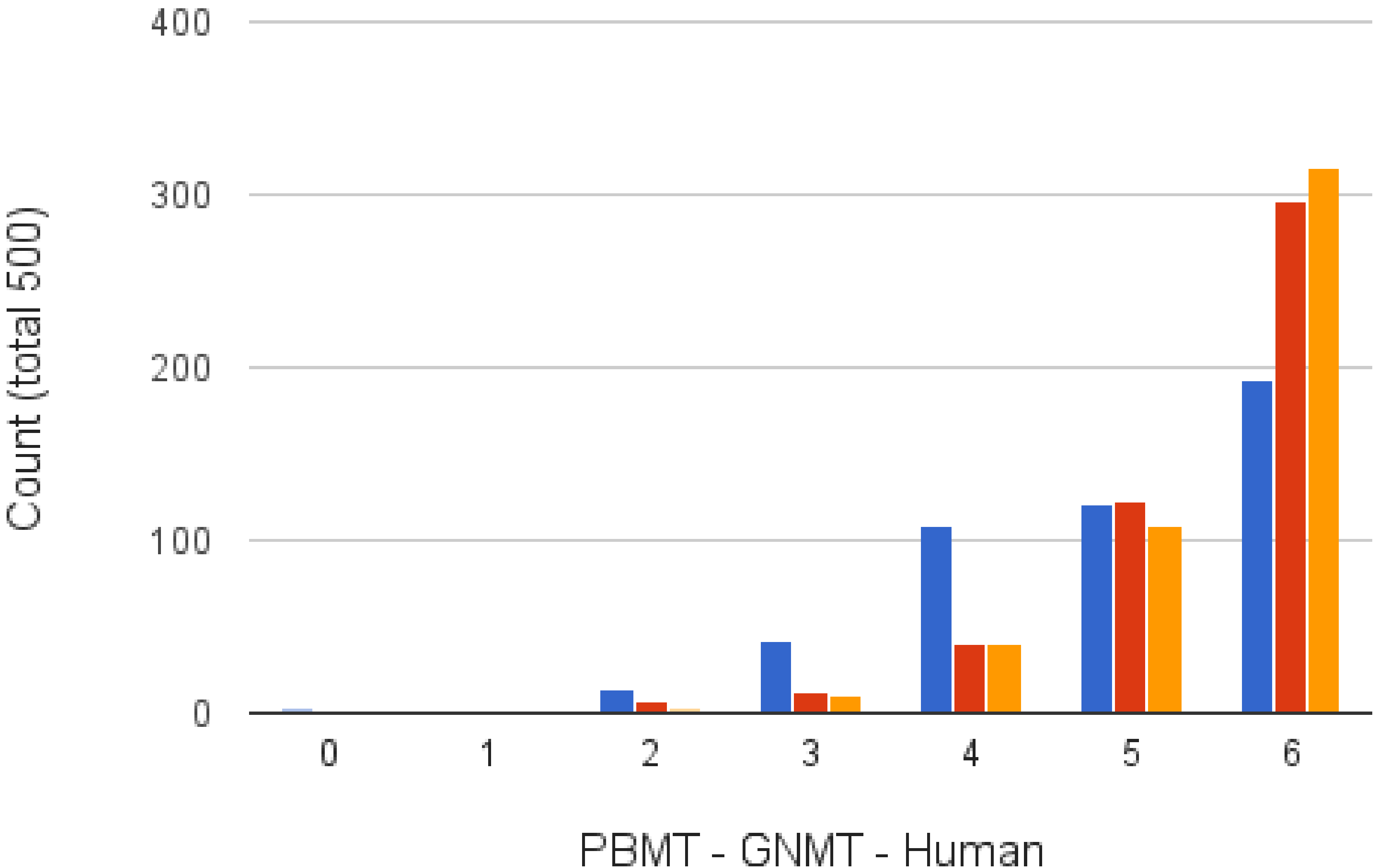


Figure 6 of "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation", https://arxiv.org/abs/1609.08144

Beyond one Language Pair

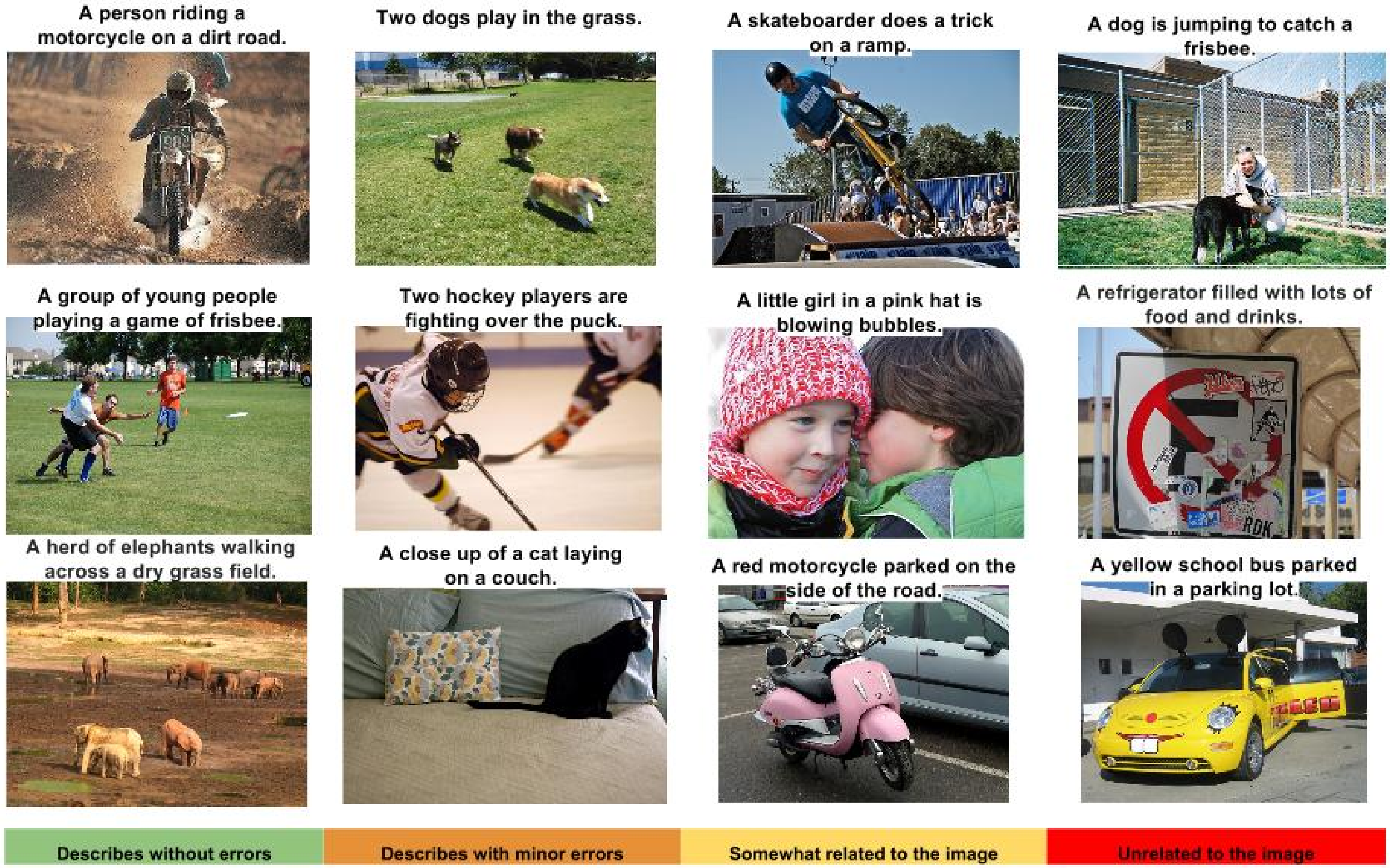
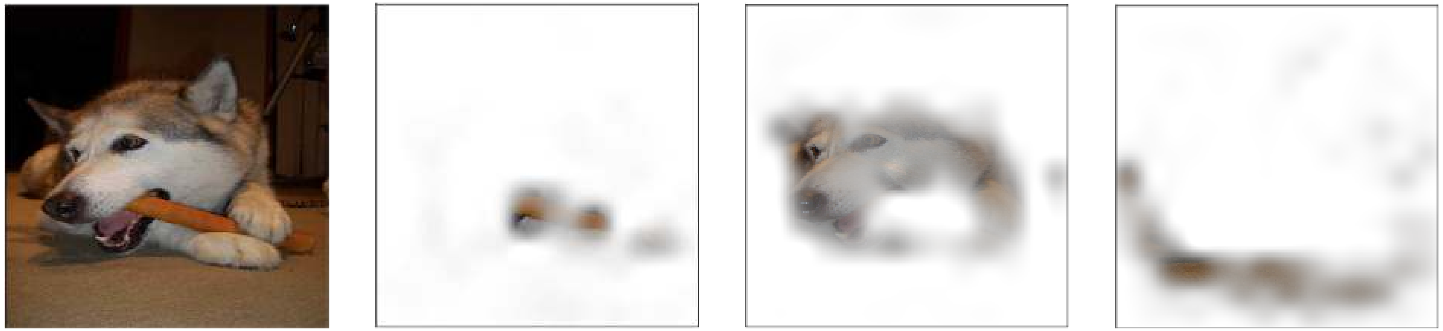

A person riding a motorcycle on a dirt road.
Two dogs play in the grass.
A skateboarder does a trick on a ramp.
A dog is jumping to catch a frisbee.
A group of young people playing a game of frisbee.
Two hockey players are fighting over the puck.
A little girl in a pink hat is blowing bubbles.
A refrigerator filled with lots of food and drinks.
A herd of elephants walking across a dry grass field.
A close up of a cat laying on a couch.
A red motorcycle parked on the side of the road.
A yellow school bus parked in a parking lot.
Describes without errors
Describes with minor errors
Somewhat related to the image
Unrelated to the image

Fig. 5. A selection of evaluation results, grouped by human rating.

Figure 5 of "Show and Tell: Lessons learned from the 2015 MSCOCO...", https://arxiv.org/abs/1609.06647

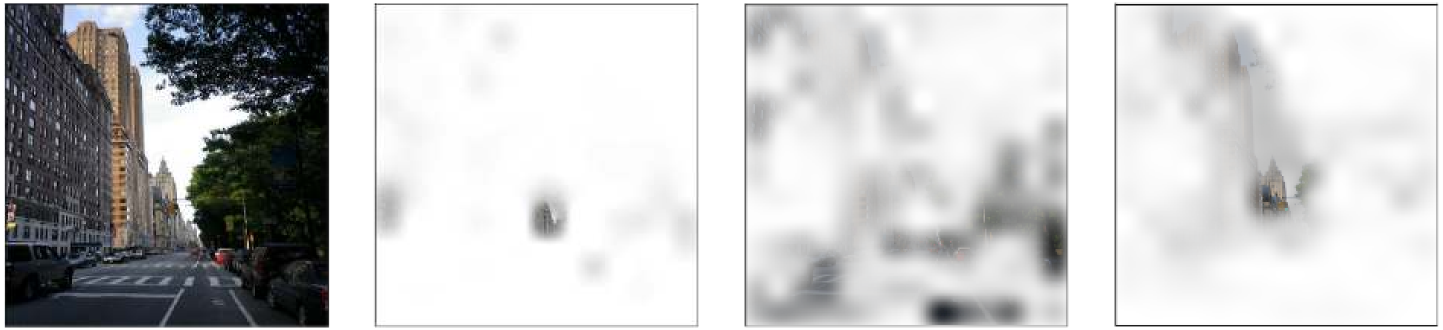
Beyond one Language Pair



What vegetable is the dog chewing on?
MCB: carrot
GT: carrot

What kind of dog is this?
MCB: husky
GT: husky

What kind of flooring does the room have?
MCB: carpet
GT: carpet



What color is the traffic light?
MCB: green
GT: green

Is this an urban area?
MCB: yes
GT: yes

Where are the buildings?
MCB: in background
GT: on left

Figure 6 of "Multimodal Compact Bilinear Pooling for VQA and Visual Grounding", https://arxiv.org/abs/1606.01847

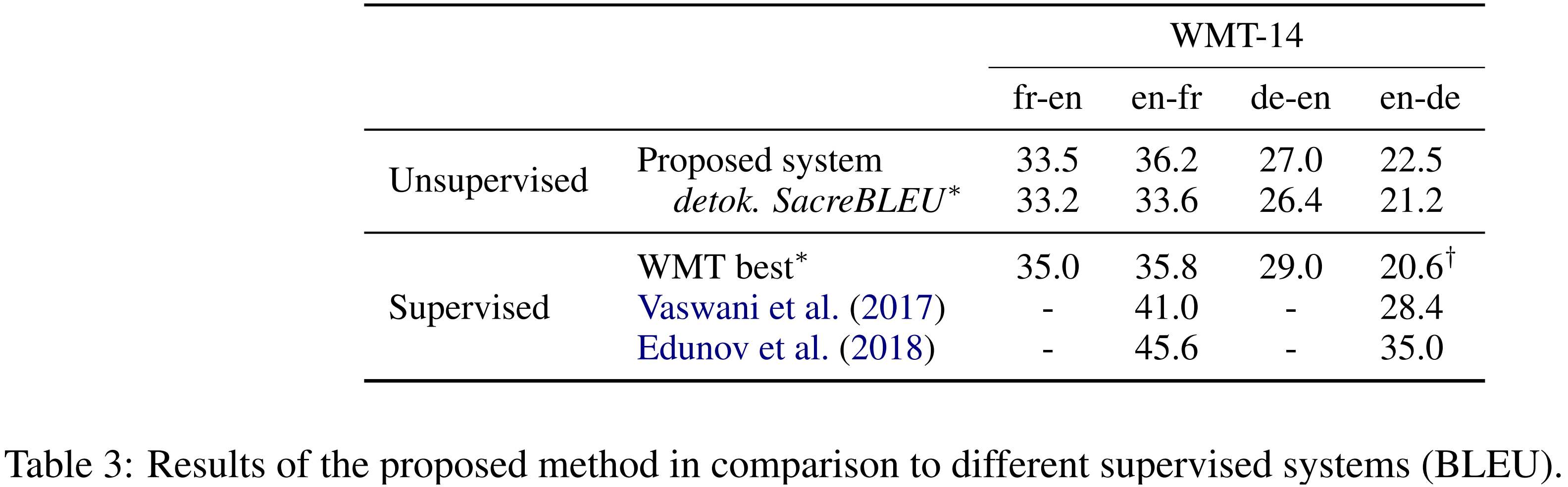
Multilingual and Unsupervised Translation

Many attempts at multilingual translation.

- Individual encoders and decoders, shared attention.
- Shared encoders and decoders.

Surprisingly, even unsupervised translation is attempted lately. By unsupervised we understand settings where we have access to large monolingual corpora, but no parallel data.

In 2019, the best unsupervised systems were on par with the best 2014 supervised systems.



		WMT-14			
		fr-en	en-fr	de-en	en-de
Unsupervised	Proposed system	33.5	36.2	27.0	22.5
	detok. SacreBLEU*	33.2	33.6	26.4	21.2
Supervised	WMT best*	35.0	35.8	29.0	20.6†
	Vaswani et al. (2017)	-	41.0	-	28.4
	Edunov et al. (2018)	-	45.6	-	35.0

Table 3: Results of the proposed method in comparison to different supervised systems (BLEU).

Table 3 of "An Effective Approach to Unsupervised Machine Translation", https://arxiv.org/abs/1902.01313

Attention is All You Need

For some sequence processing tasks, *sequential* processing (as performed by recurrent neural networks) of its elements might be too restrictive.

Instead, we may want to be able to combine sequence elements independently on their distance.

Such processing is allowed in the **Transformer** architecture, originally proposed for neural machine translation in 2017 in *Attention is All You Need* paper.

Transformer

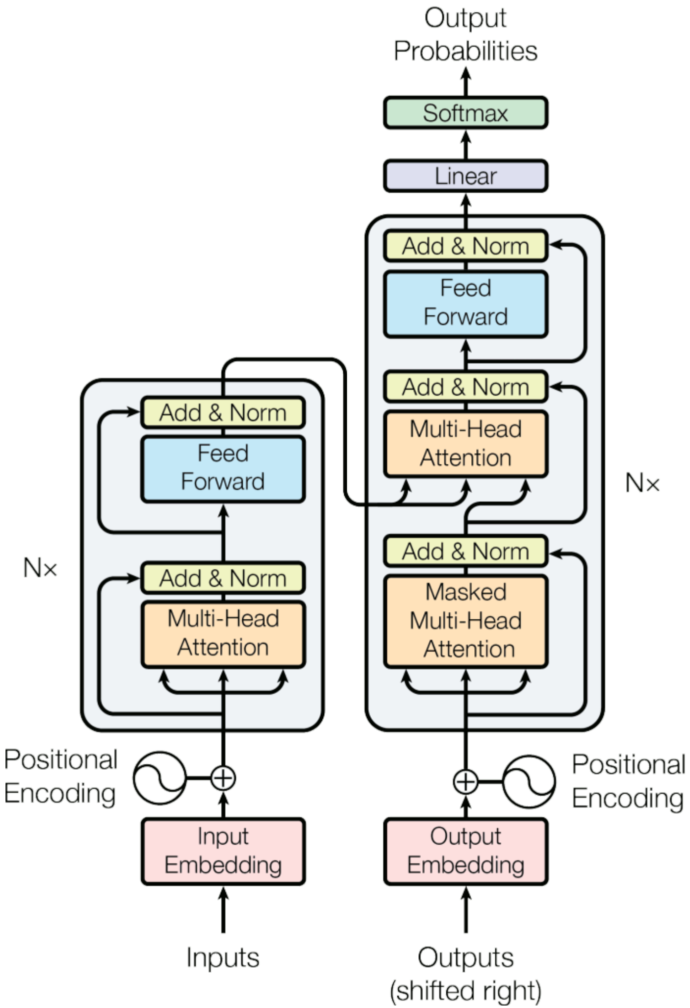
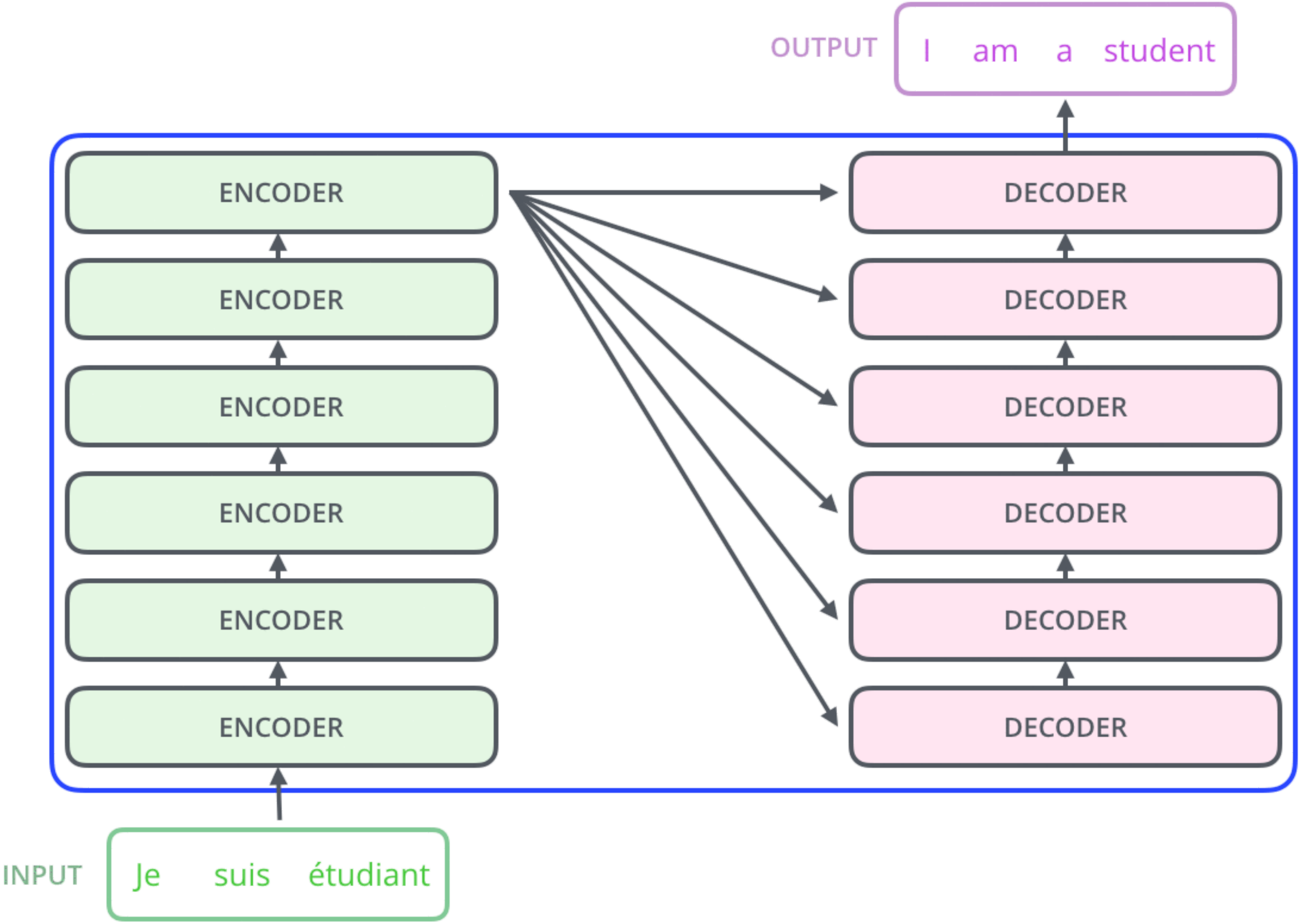

Output
Probabilities
Softmax
Linear
Add & Norm
Feed
Forward
Add & Norm
Multi-Head
Attention
Add & Norm
Masked
Multi-Head
Attention
Nx
Add & Norm
Feed
Forward
Add & Norm
Multi-Head
Attention
Nx
Positional
Encoding
Positional
Encoding
Input
Embedding
Output
Embedding
Inputs
Outputs
(shifted right)

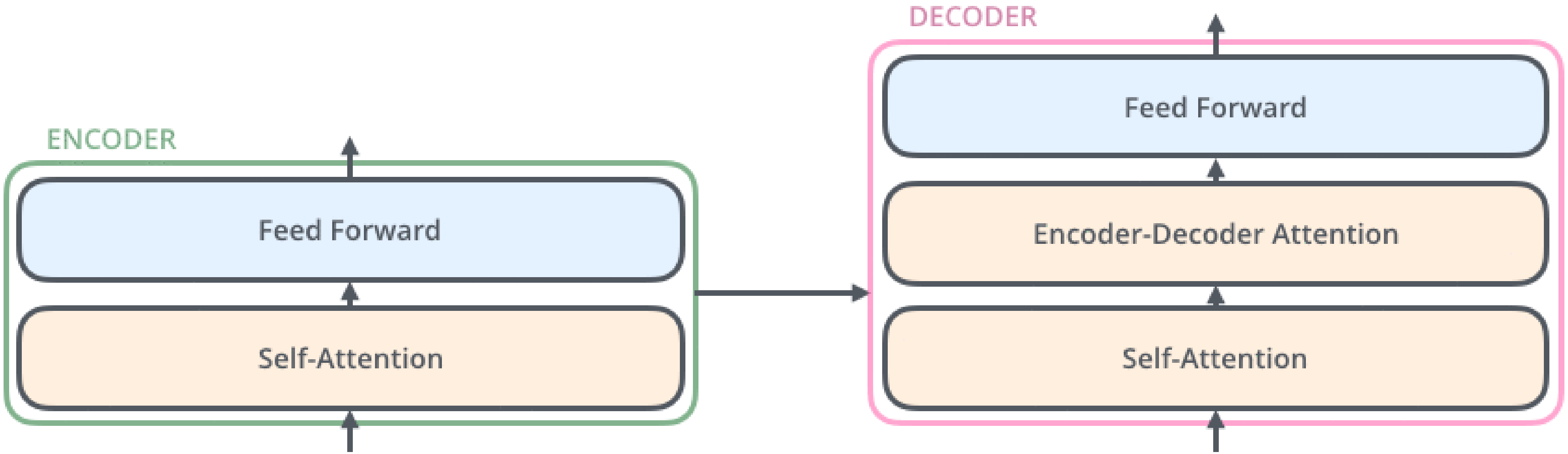
Figure 1 of "Attention Is All You Need", https://arxiv.org/abs/1706.03762

Transformer


OUTPUT
I am a student
ENCODER
ENCODER
ENCODER
ENCODER
ENCODER
ENCODER
DECODER
DECODER
DECODER
DECODER
DECODER
DECODER
INPUT
Je suis étudiant

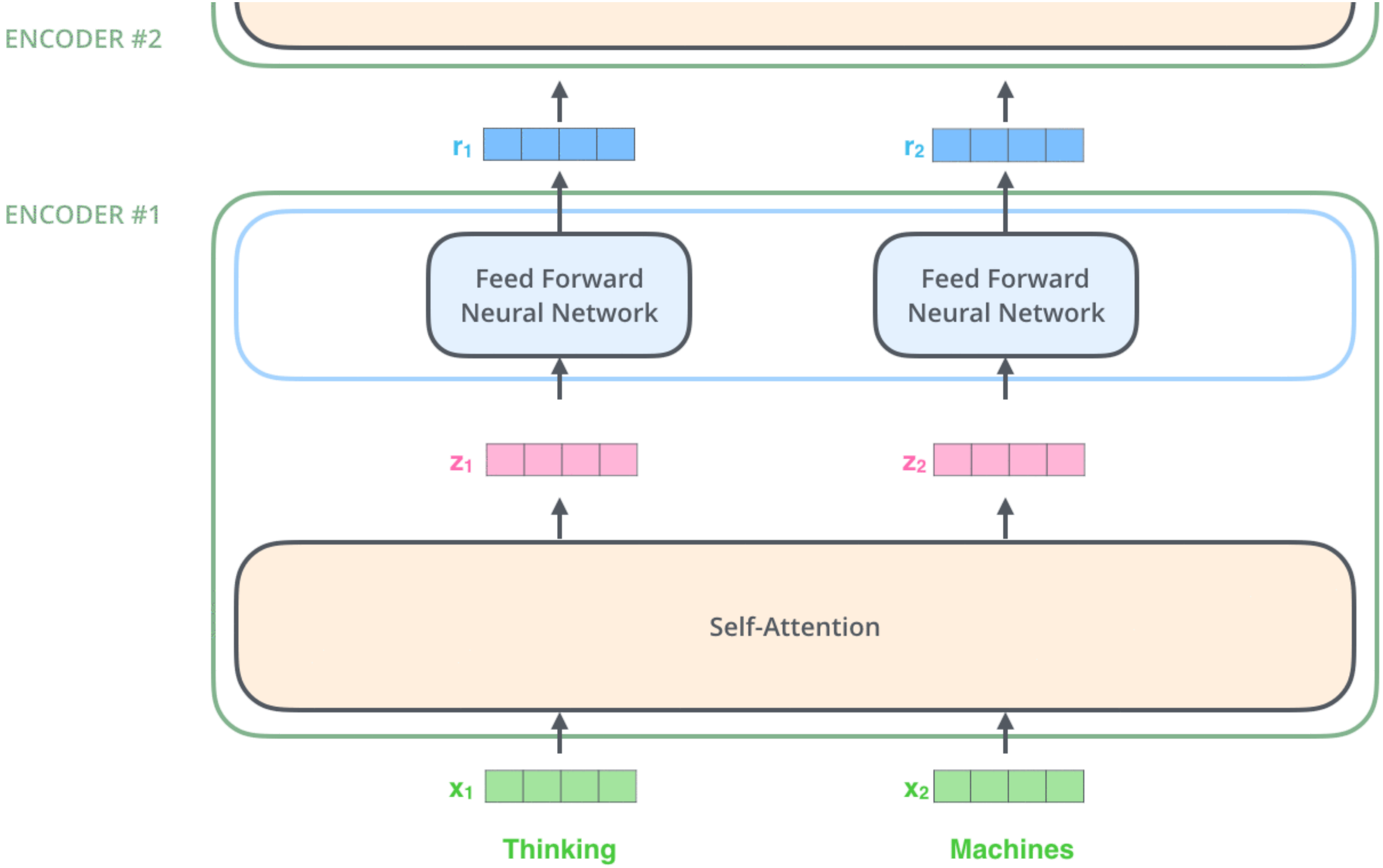
http://jalammar.github.io/images/t/The_transformer_encoder_decoder_stack.png

Transformer


DECODER
Feed Forward
Encoder-Decoder Attention
Self-Attention
ENCODER
Feed Forward
Self-Attention

http://jalammar.github.io/images/t/Transformer_decoder.png

Transformer


ENCODER #2
ENCODER #1
r_1
r_2
Feed Forward Neural Network
Feed Forward Neural Network
z_1
z_2
Self-Attention
x_1
x_2
Thinking
Machines

http://jalammar.github.io/images/t/encoder_with_tensors_2.png

Transformer – Self-Attention

Assume that we have a sequence of n words represented using a matrix $\boldsymbol{X} \in \mathbb{R}^{n\times d}$.

The attention module for queries $\boldsymbol{Q} \in \mathbb{R}^{n\times d_k}$, keys $\boldsymbol{K} \in \mathbb{R}^{n\times d_k}$ and values $\boldsymbol{V} \in \mathbb{R}^{n\times d_v}$ is defined as:

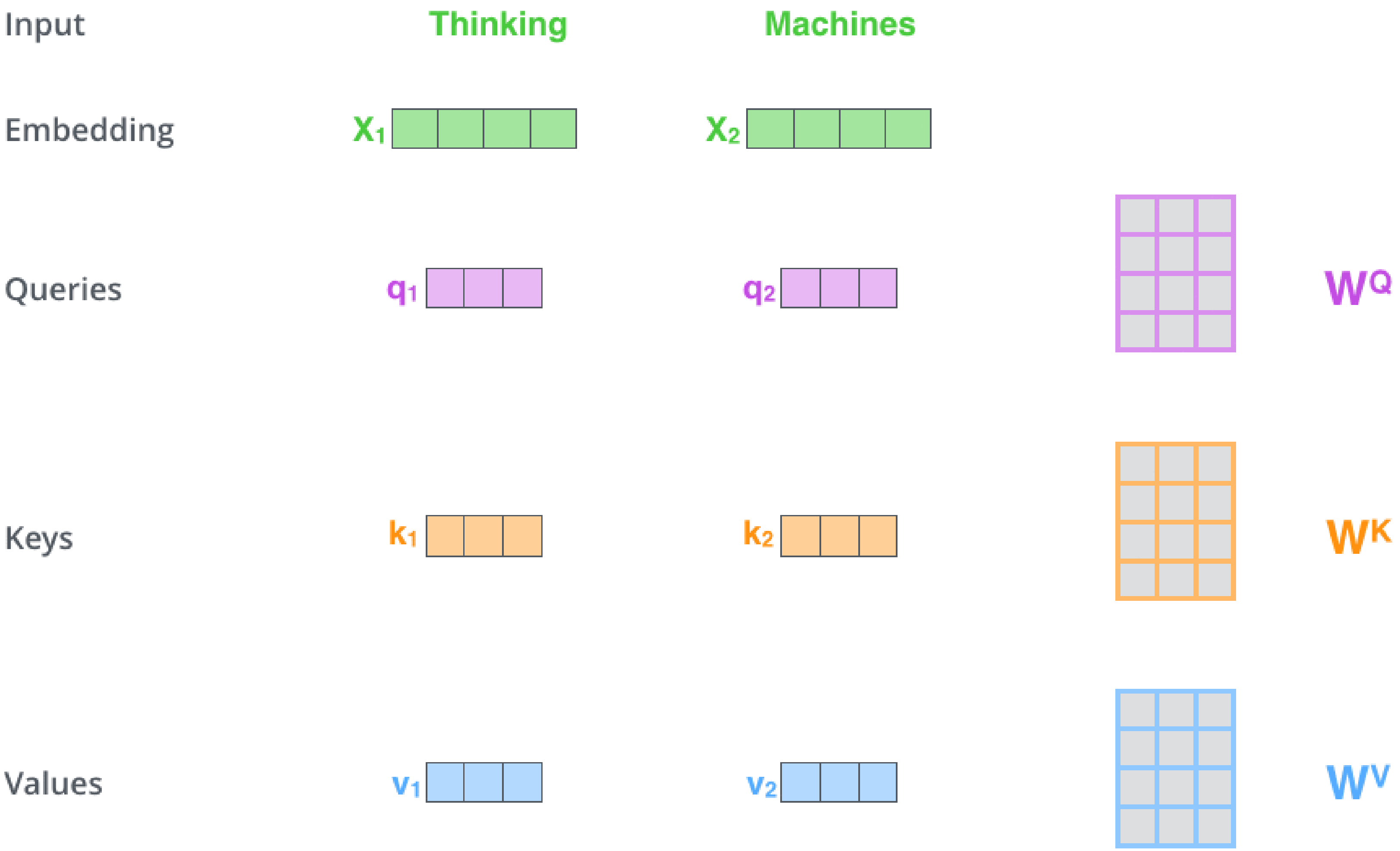
$$\operatorname{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \operatorname{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d_k}}\right)\boldsymbol{V}.$$

The queries, keys and values are computed from the input word representations $\boldsymbol{X}$ using a linear transformation as

$$\begin{aligned}\boldsymbol{Q} &= \boldsymbol{X}\boldsymbol{W}^Q\\ \boldsymbol{K} &= \boldsymbol{X}\boldsymbol{W}^K\\ \boldsymbol{V} &= \boldsymbol{X}\boldsymbol{W}^V\end{aligned}$$

for trainable weight matrices $\boldsymbol{W}^Q, \boldsymbol{W}^K \in \mathbb{R}^{d\times d_k}$ and $\boldsymbol{W}^V \in \mathbb{R}^{d\times d_v}$.

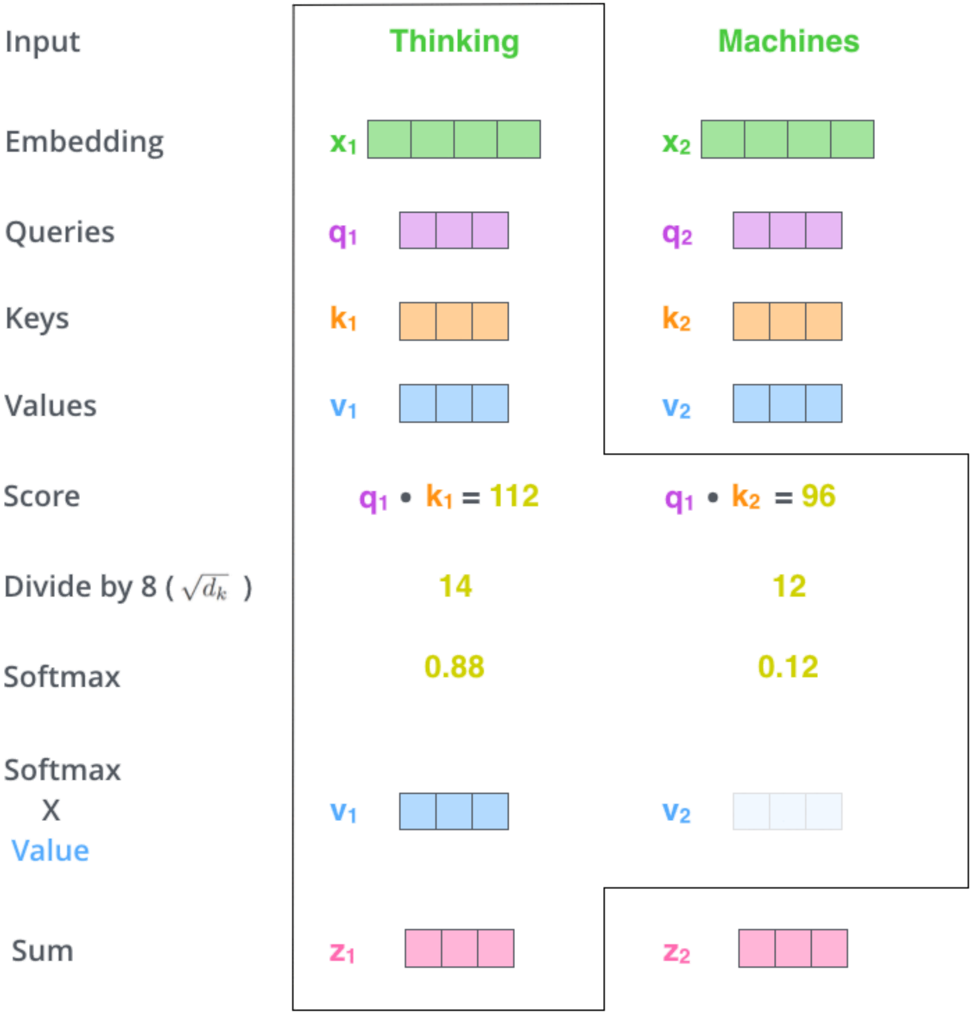
Transformer – Self-Attention



Input
Thinking
Machines
Embedding
x_1
x_2
Queries
q_1
q_2
W^Q
Keys
k_1
k_2
W^K
Values
v_1
v_2
W^V

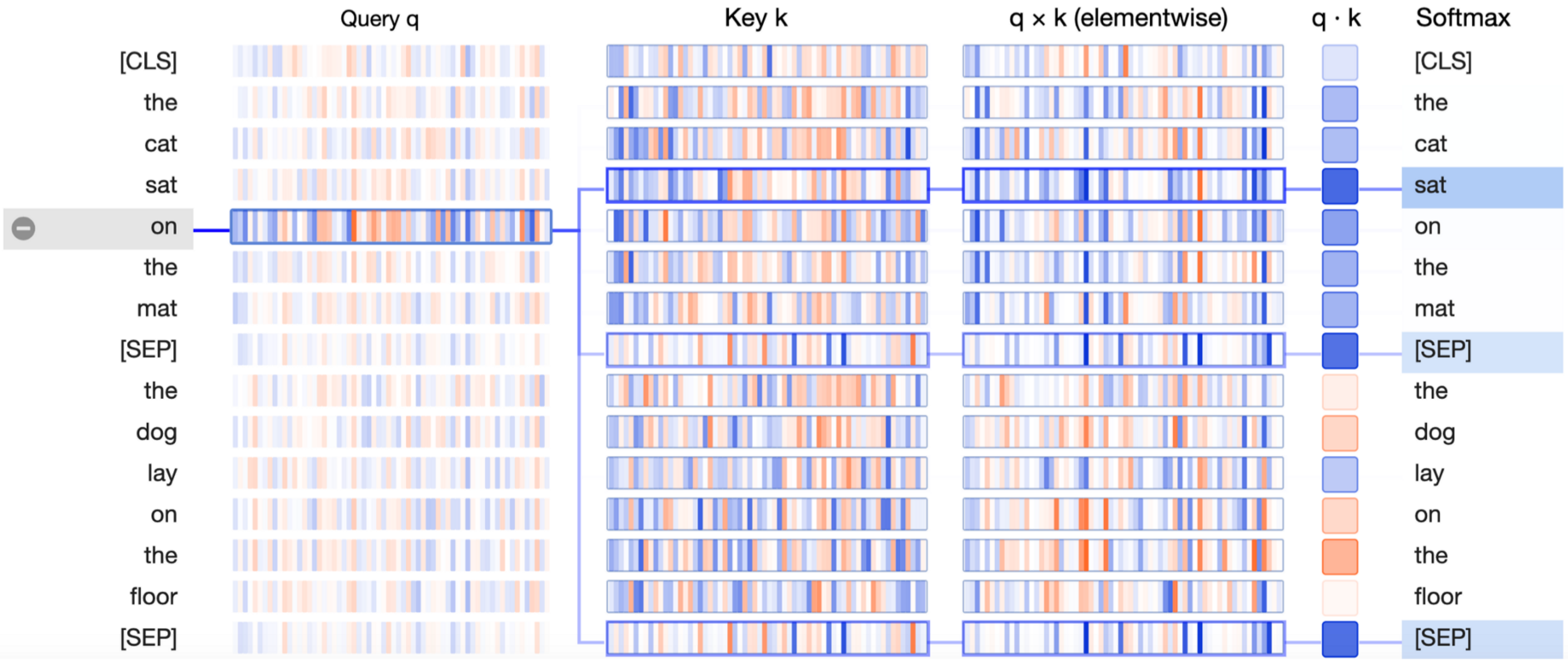
http://jalammar.github.io/images/t/transformer_self_attention_vectors.png

Transformer – Self-Attention


Input
Thinking
Machines
Embedding
x_1
x_2
Queries
q_1
q_2
Keys
k_1
k_2
Values
v_1
v_2
Score
$q_1 \bullet k_1 = 112$
$q_1 \bullet k_2 = 96$
Divide by 8 ($\sqrt{d_k}$)
14
12
Softmax
0.88
0.12
Softmax X Value
v_1
v_2
Sum
z_1
z_2

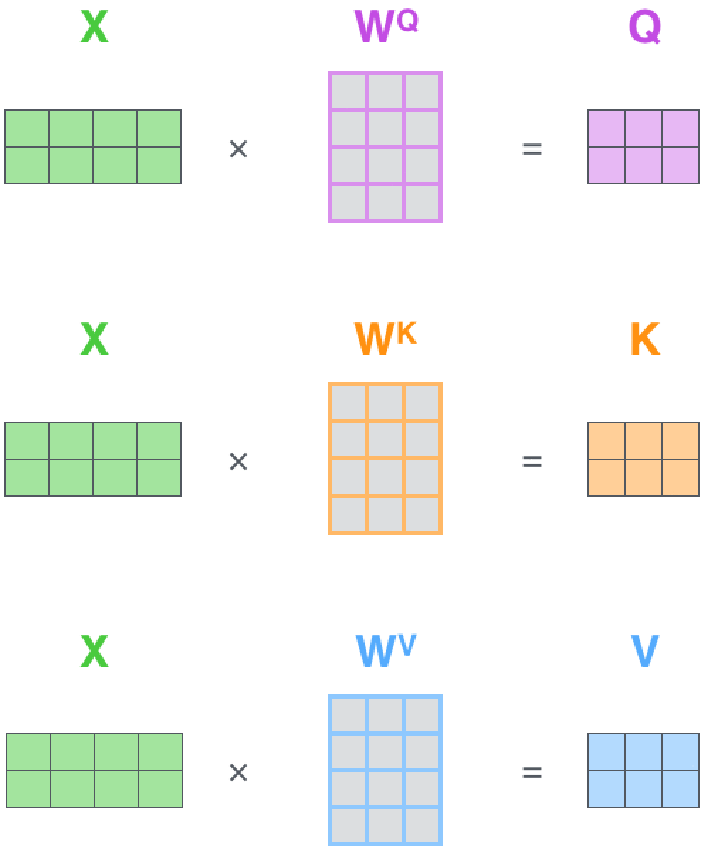
http://jalammar.github.io/images/t/self-attention-output.png

Transformer – Self-Attention

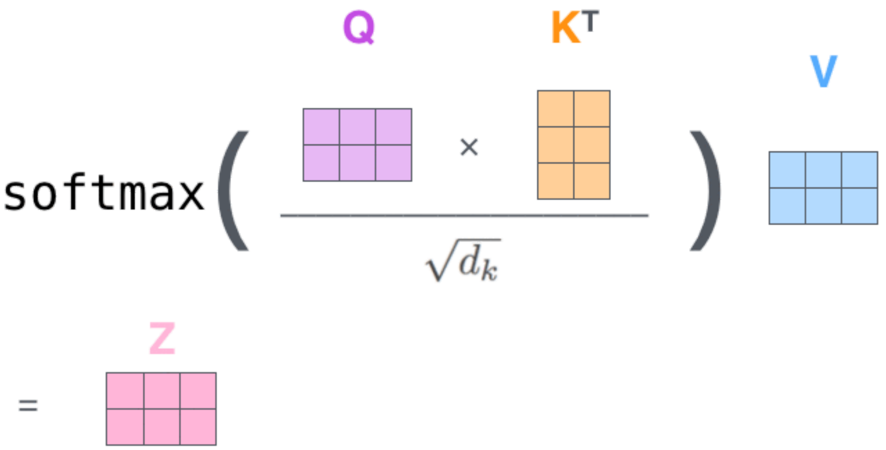

Query q
Key k
q × k (elementwise)
q · k
Softmax
[CLS]
the
cat
sat
on
the
mat
[SEP]
the
dog
lay
on
the
floor
[SEP]

*https://miro.medium.com/max/2000/1*jBsfVNOOcJ-I3tsLVgni_w.png*

Transformer – Self-Attention


X
WQ
Q
X
WK
K
X
WV
V
×
=

http://jalammar.github.io/images/t/self-attention-matrix-calculation.png


softmax
Q
×
KT
$\sqrt{d_k}$
V
=
Z

http://jalammar.github.io/images/t/self-attention-matrix-calculation-2.png

Transformer – Multihead Attention

Multihead attention is used in practice. Instead of using one huge attention, we split queries, keys and values to several groups (similar to how ResNeXt works), compute the attention in each of the groups separately, concatenate the results and multiply them by a matrix $\boldsymbol{W}^O$.

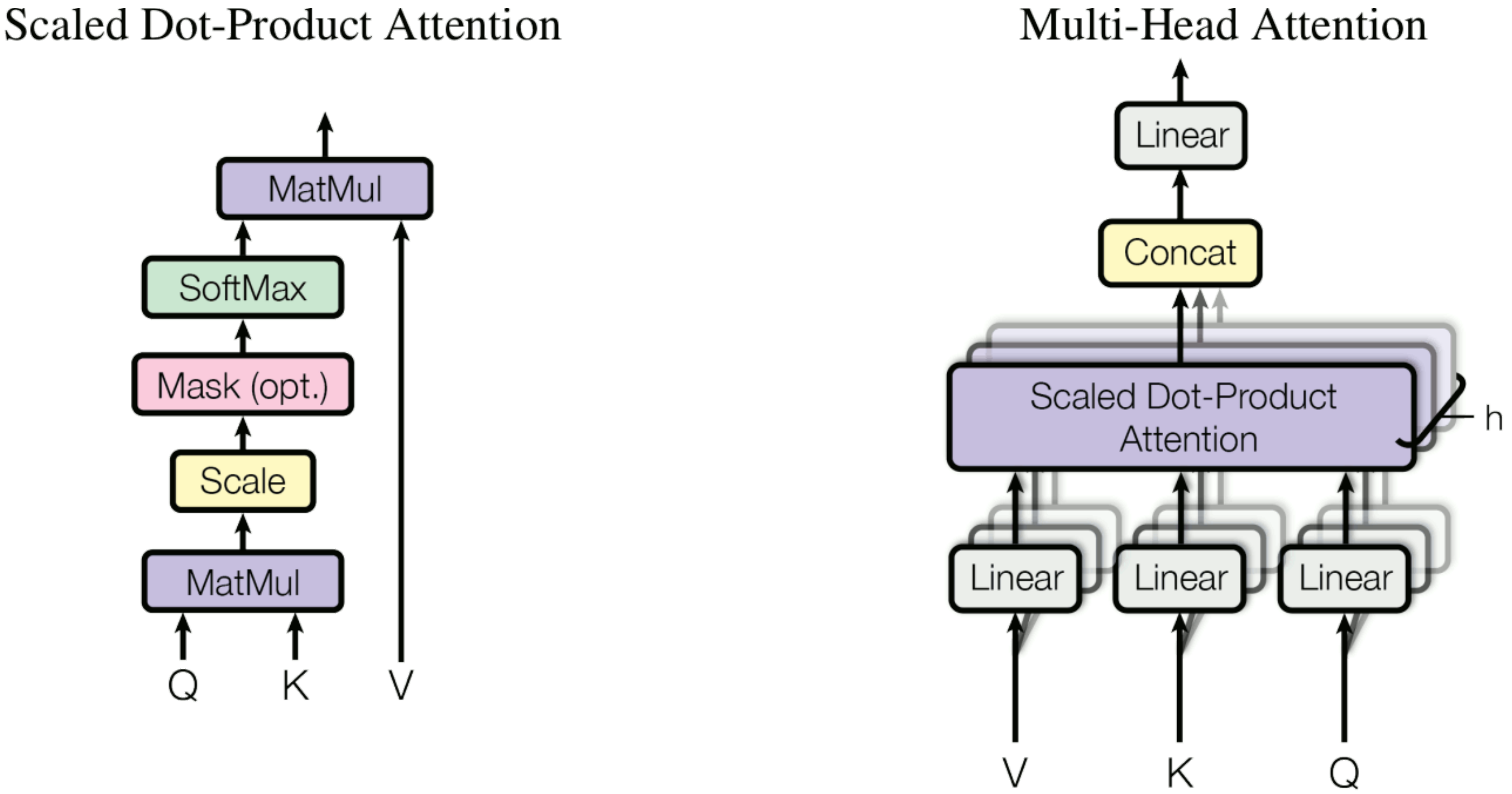
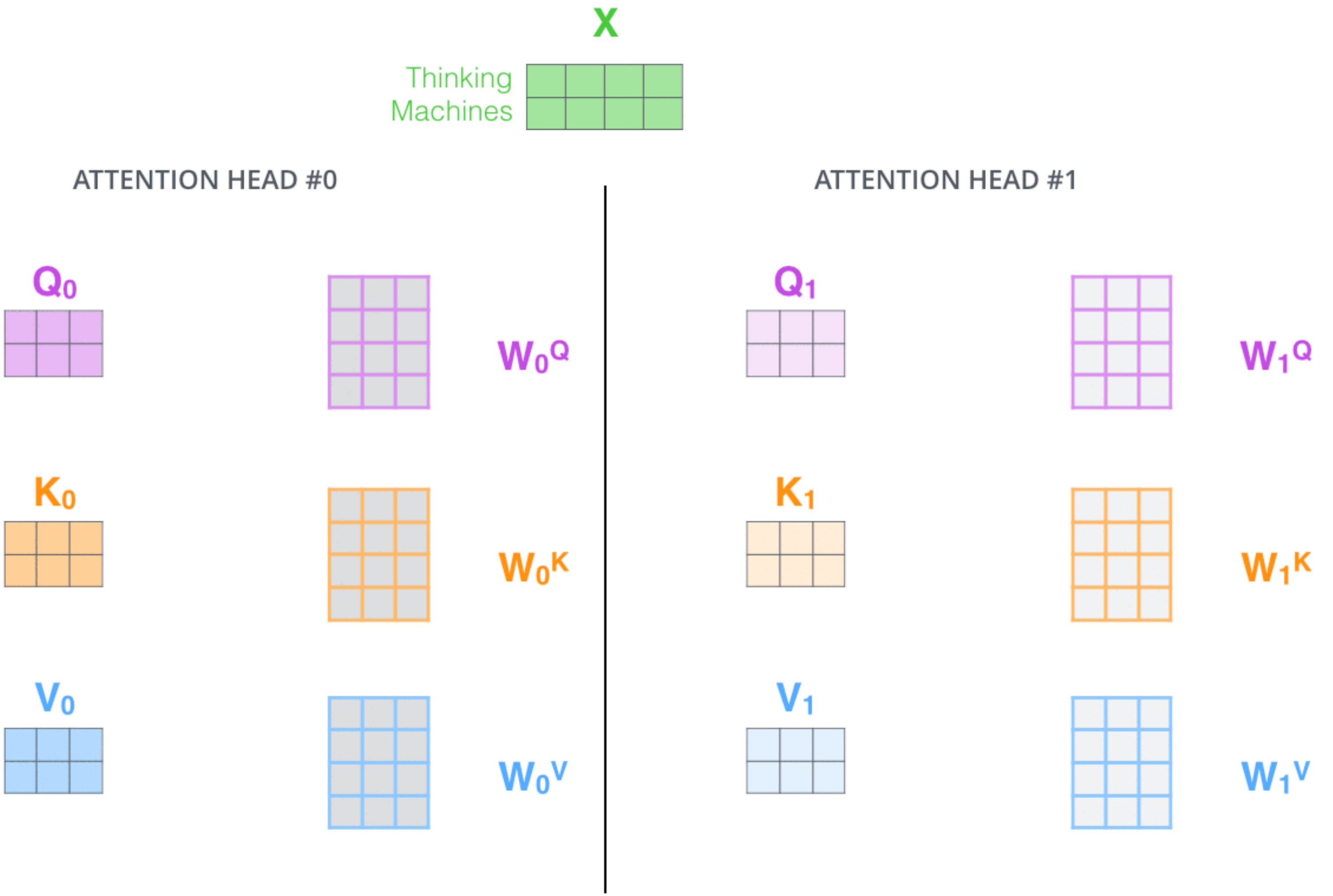

Scaled Dot-Product Attention
MatMul
SoftMax
Mask (opt.)
Scale
MatMul
Q
K
V
Multi-Head Attention
Linear
Concat
Scaled Dot-Product Attention
h
Linear
Linear
Linear
V
K
Q

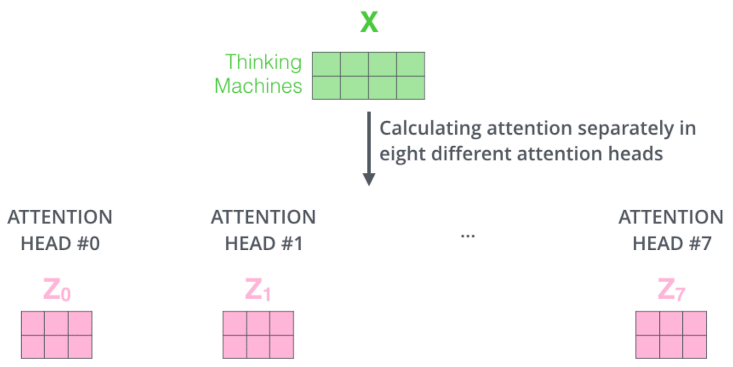
Figure 2 of "Attention Is All You Need", https://arxiv.org/abs/1706.03762

Transformer – Multihead Attention


X
Thinking
Machines
ATTENTION HEAD #0
ATTENTION HEAD #1
Q0
K0
V0
W0Q
W0K
W0V
Q1
K1
V1
W1Q
W1K
W1V

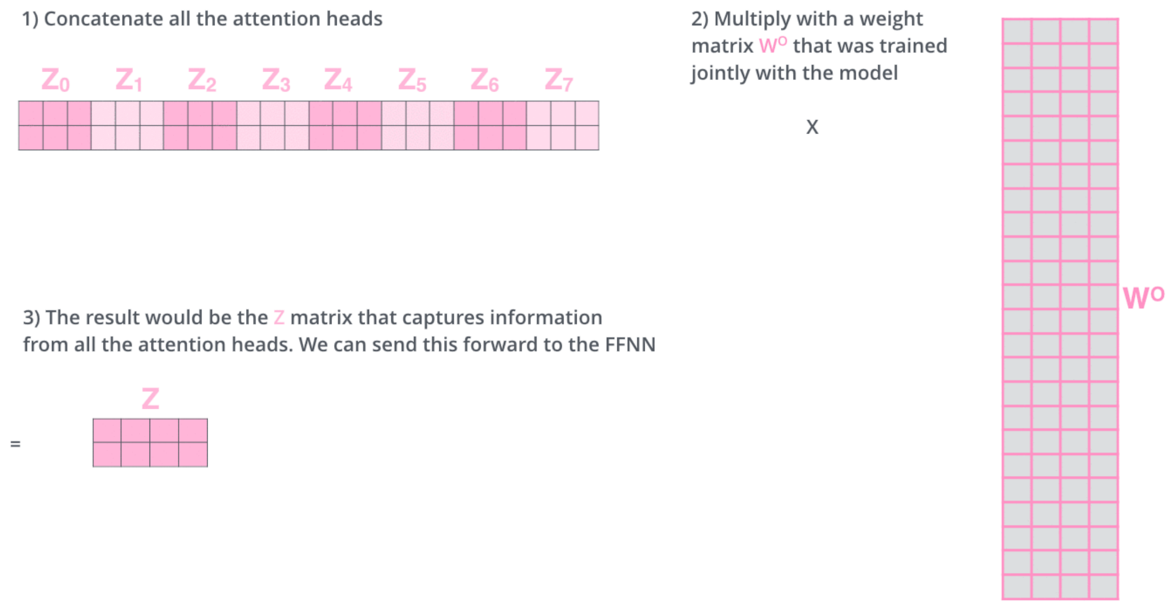
http://jalammar.github.io/images/t/transformer_attention_heads_qkv.png

Transformer – Multihead Attention


Calculating attention separately in eight different attention heads

X
Thinking
Machines
ATTENTION HEAD #0
ATTENTION HEAD #1
...
ATTENTION HEAD #7
Z_0
Z_1
Z_7

http://jalammar.github.io/images/t/transformer_attention_heads_z.png


1) Concatenate all the attention heads

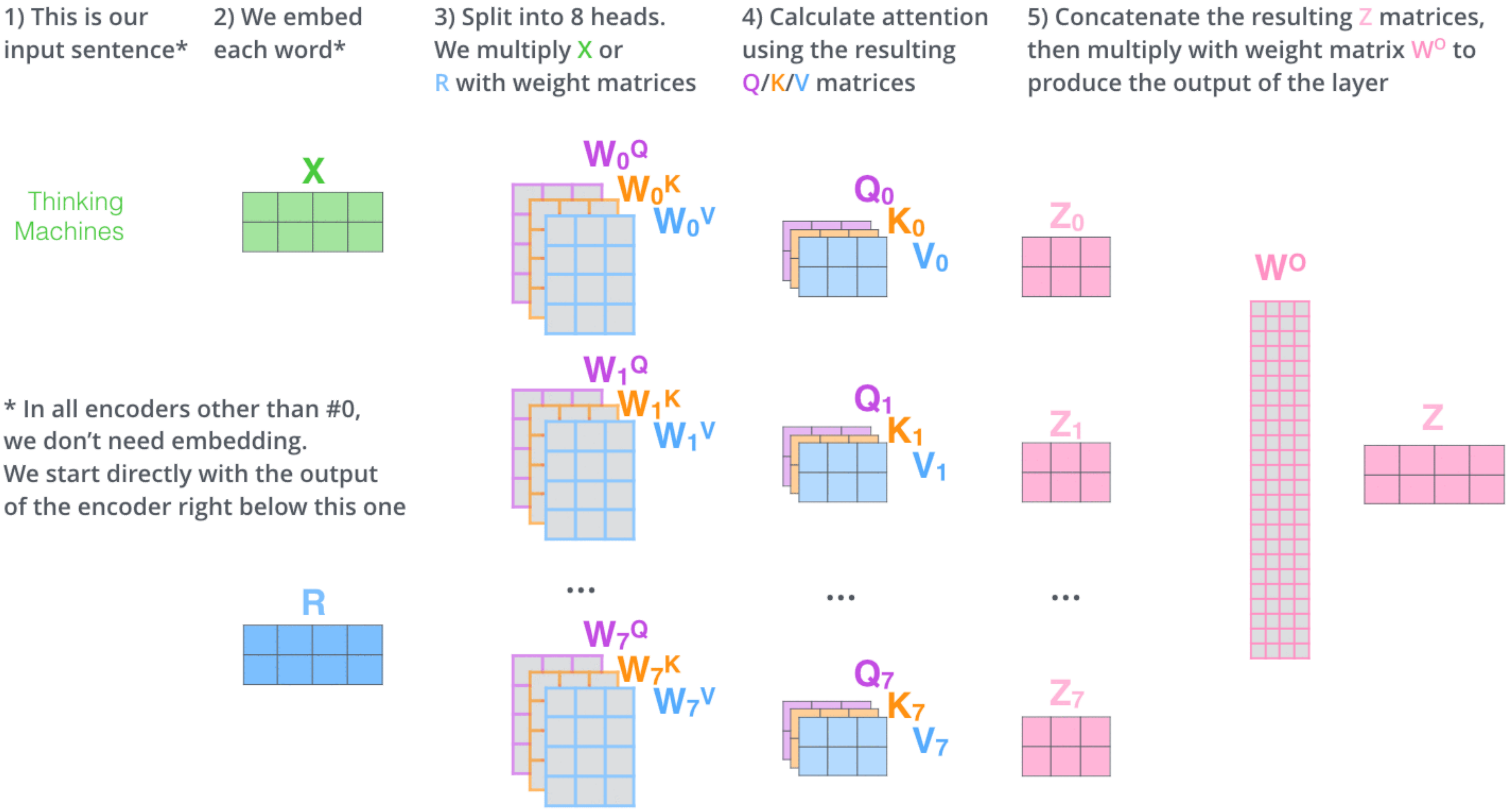
2) Multiply with a weight matrix W^O that was trained jointly with the model

3) The result would be the Z matrix that captures information from all the attention heads. We can send this forward to the FFNN

Z_0
Z_1
Z_2
Z_3
Z_4
Z_5
Z_6
Z_7
x
W^O
=
Z

http://jalammar.github.io/images/t/transformer_attention_heads_weight_matrix_o.png

Transformer – Multihead Attention



1) This is our input sentence*

2) We embed each word*

3) Split into 8 heads. We multiply X or R with weight matrices

4) Calculate attention using the resulting Q/K/V matrices

5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer

* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one

Thinking
Machines
X
R
W_0^Q
W_0^K
W_0^V
W_1^Q
W_1^K
W_1^V
...
W_7^Q
W_7^K
W_7^V
Q_0
K_0
V_0
Q_1
K_1
V_1
...
Q_7
K_7
V_7
Z_0
Z_1
...
Z_7
W^O
Z

http://jalammar.github.io/images/t/transformer_multi-headed_self-attention-recap.png

Why Attention

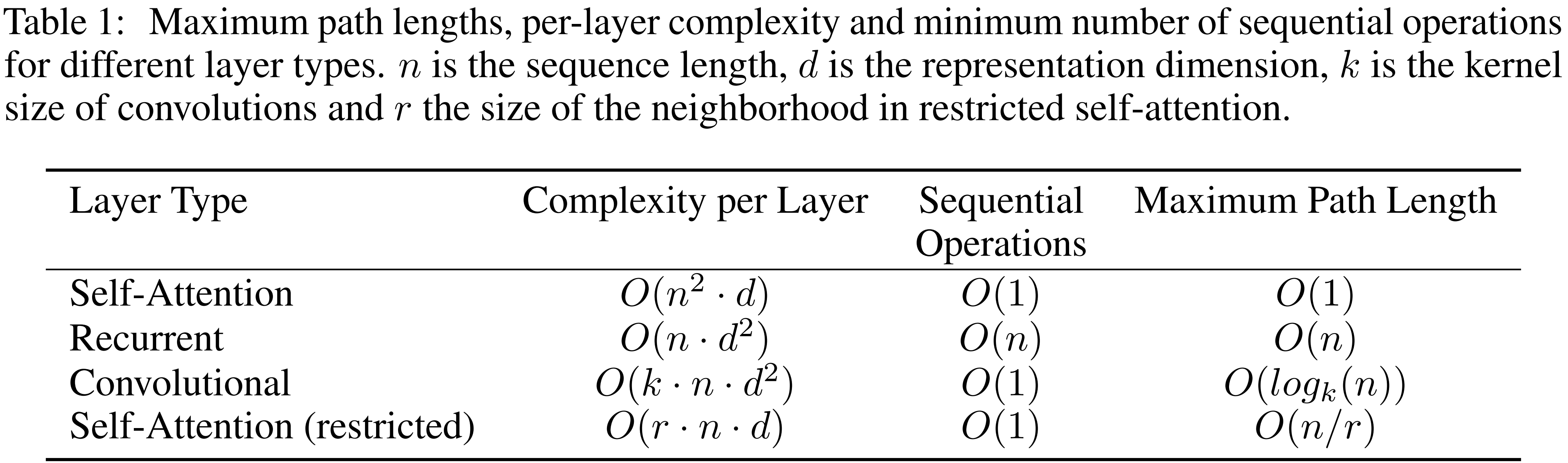


Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

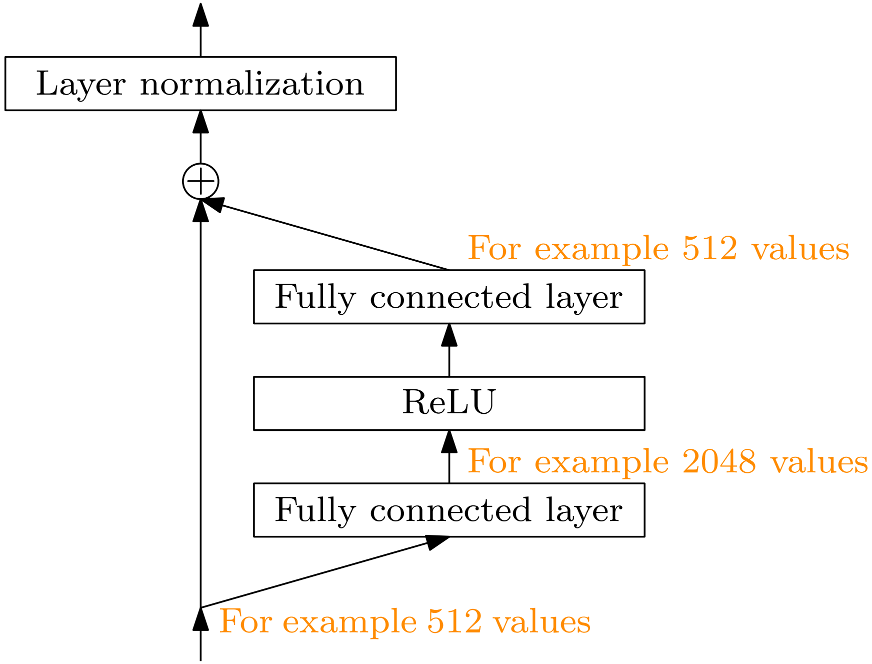
Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Table 1 of "Attention Is All You Need", https://arxiv.org/abs/1706.03762

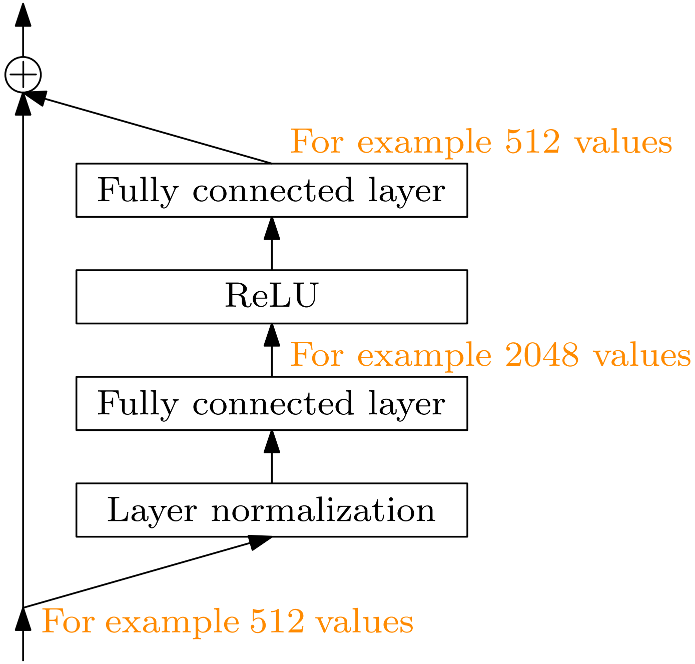
Feed Forward Networks

The self-attention is complemented with FFN layers, which is a fully connected ReLU layer with four times as many hidden units as inputs, followed by another fully connected layer without activation.

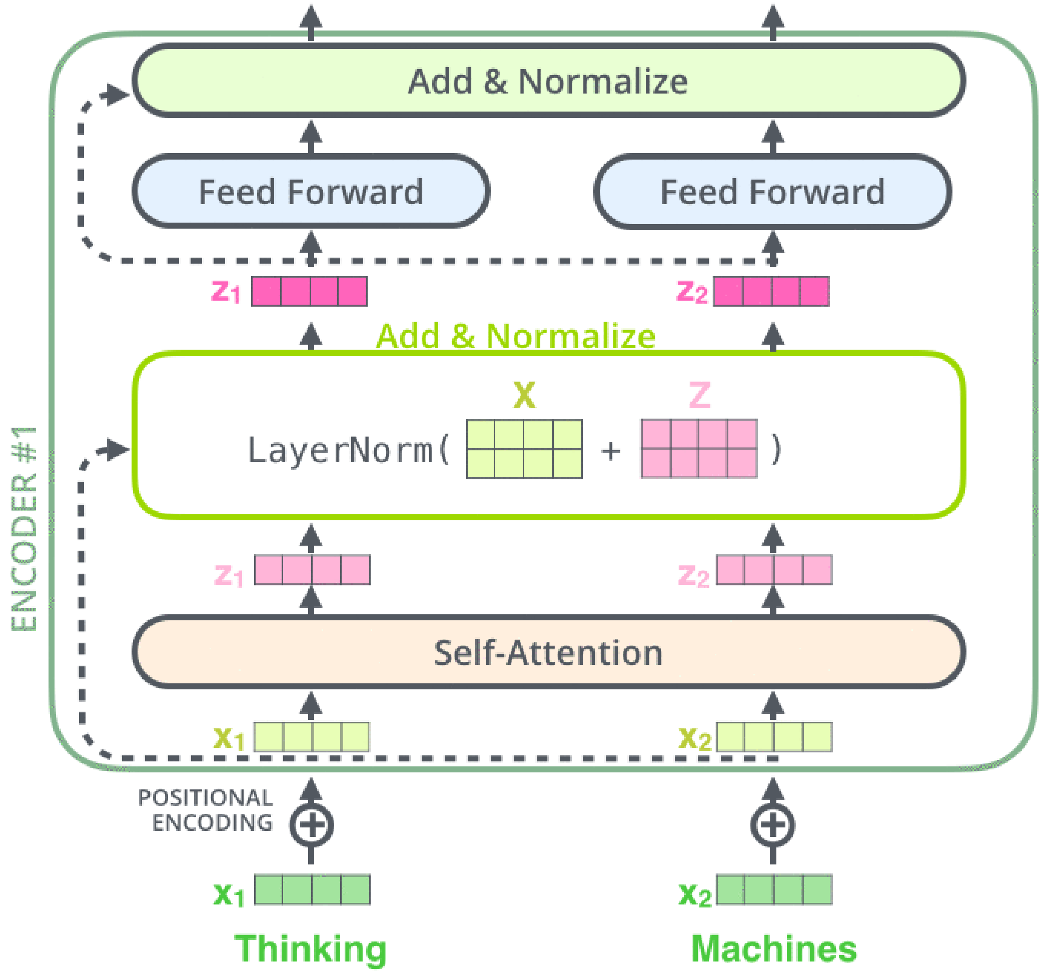
Original “Post-LN” configuration


Layer normalization
For example 512 values
Fully connected layer
ReLU
For example 2048 values
Fully connected layer
For example 512 values

Improved “Pre-LN“ configuration since 2020


For example 512 values
Fully connected layer
ReLU
For example 2048 values
Fully connected layer
Layer normalization
For example 512 values

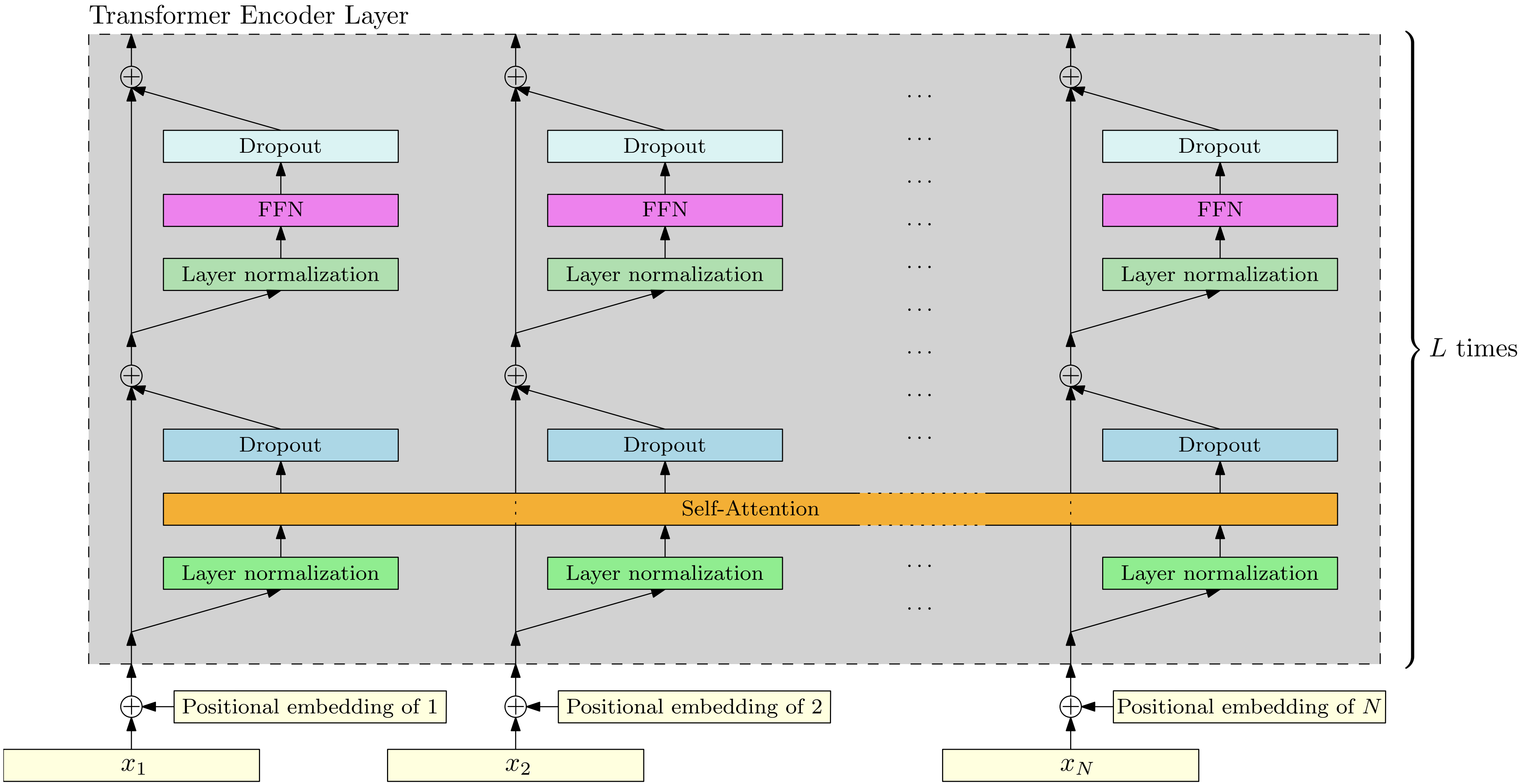
Transformer – Post-LN Configuration including Residuals



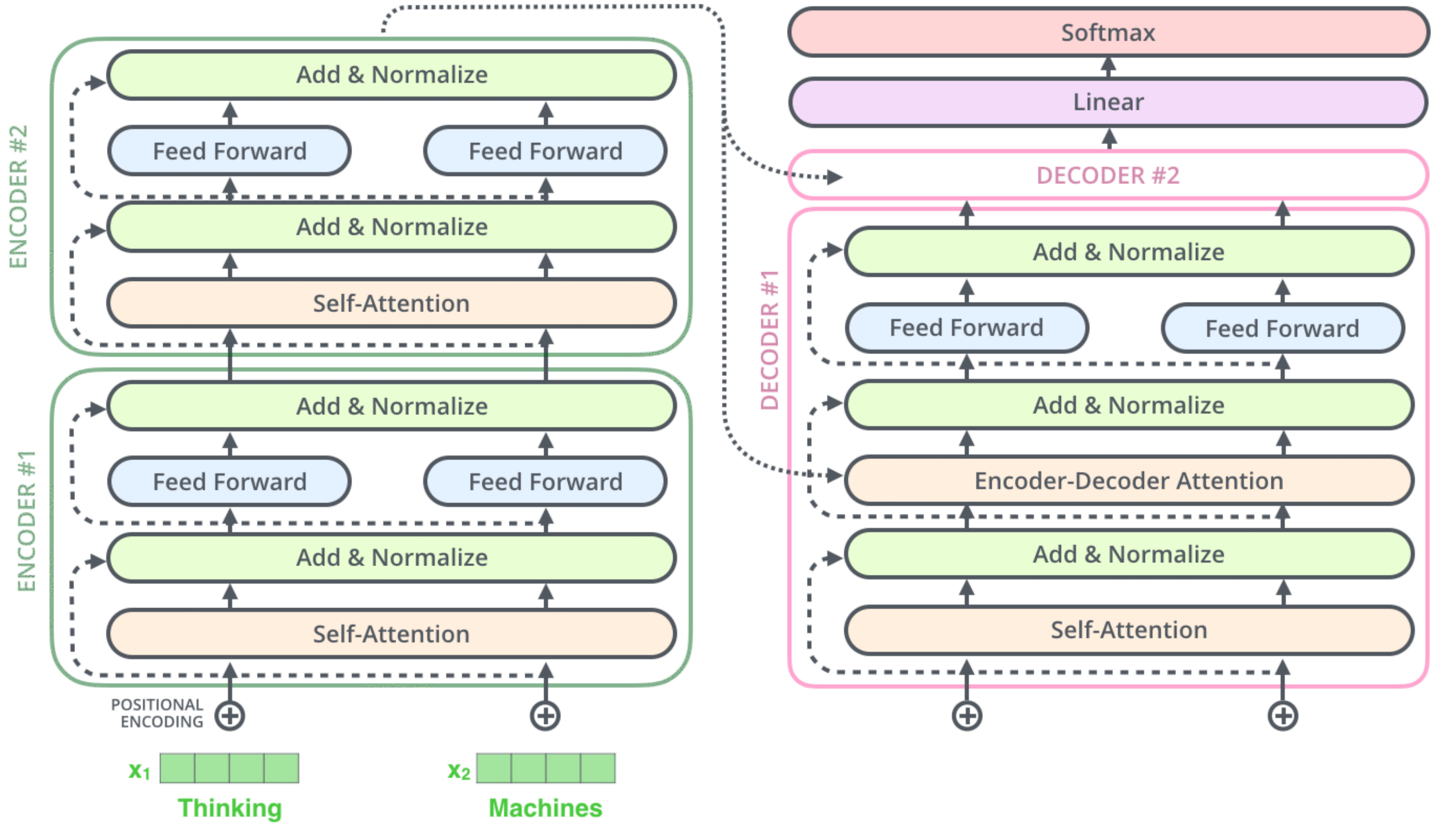
Add & Normalize
Feed Forward
Feed Forward
z_1
z_2
Add & Normalize
X
Z
LayerNorm(+)
z_1
z_2
Self-Attention
x_1
x_2
ENCODER #1
POSITIONAL ENCODING
x_1
x_2
Thinking
Machines

http://jalammar.github.io/images/t/transformer_resideual_layer_norm_2.png

Transformer – Pre-LN Configuration


Transformer Encoder Layer
Dropout
FFN
Layer normalization
Dropout
Self-Attention
Layer normalization
. . .
L times
Positional embedding of 1
Positional embedding of 2
Positional embedding of N
x_1
x_2
x_N

Transformer – Decoder


ENCODER #2
Add & Normalize
Feed Forward
Feed Forward
Add & Normalize
Self-Attention
ENCODER #1
Add & Normalize
Feed Forward
Feed Forward
Add & Normalize
Self-Attention
POSITIONAL ENCODING
x_1
Thinking
x_2
Machines
Softmax
Linear
DECODER #2
DECODER #1
Add & Normalize
Feed Forward
Feed Forward
Add & Normalize
Encoder-Decoder Attention
Add & Normalize
Self-Attention

http://jalammar.github.io/images/t/transformer_resideual_layer_norm_3.png

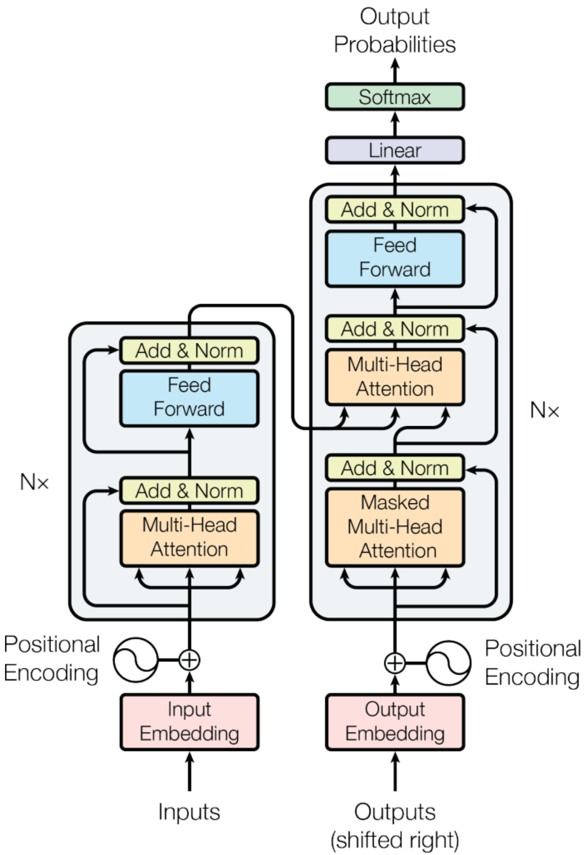

Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
N×
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
N×
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

Figure 1 of "Attention Is All You Need", https://arxiv.org/abs/1706.03762

Transformer – Decoder

Masked Self-Attention

During decoding, the self-attention must attend only to earlier positions in the output sequence.

This is achieved by **masking** future positions, i.e., zeroing their weights out, which is usually implemented by setting them to $-\infty$ before the $\operatorname{softmax}$ calculation.

Encoder-Decoder Attention

In the encoder-decoder attentions, the *queries* comes from the decoder, while the *keys* and the *values* originate from the encoder.

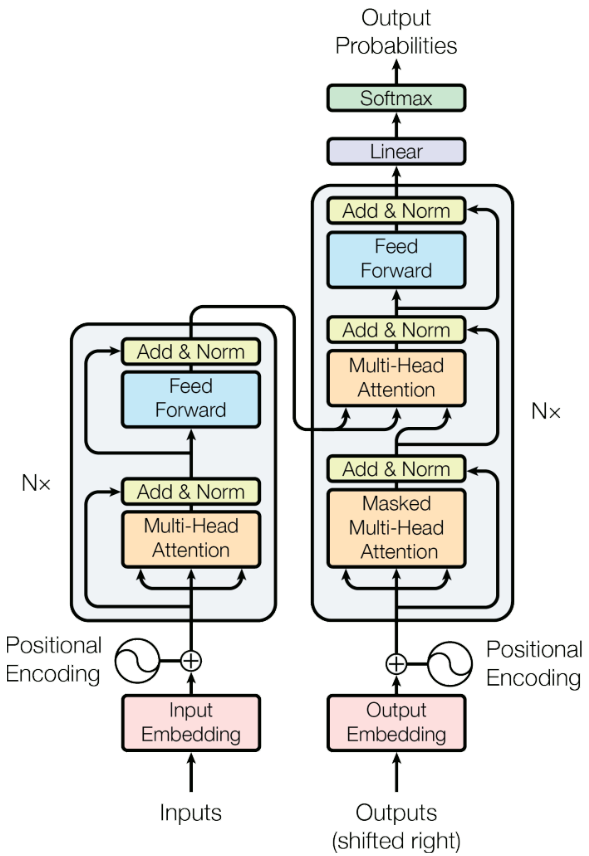
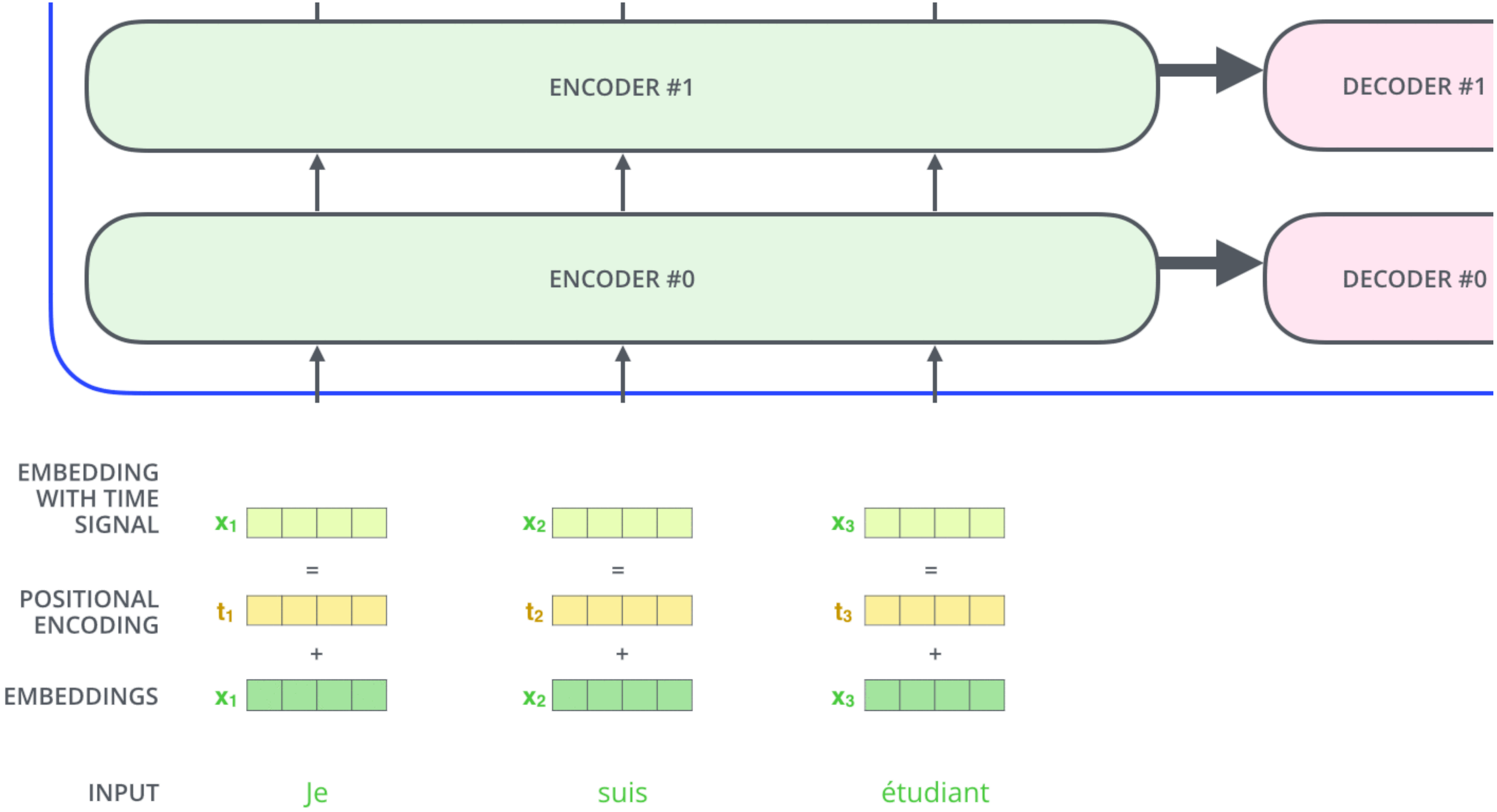

Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
N×
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
N×
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

Figure 1 of "Attention Is All You Need", https://arxiv.org/abs/1706.03762

Transformer – Positional Embedding


ENCODER #1
DECODER #1
ENCODER #0
DECODER #0
EMBEDDING WITH TIME SIGNAL
x1
x2
x3
=
POSITIONAL ENCODING
t1
t2
t3
+
EMBEDDINGS
x1
x2
x3
INPUT
Je
suis
étudiant

http://jalammar.github.io/images/t/transformer_positional_encoding_vectors.png

Positional Embeddings

We need to encode positional information (which was implicit in RNNs).

- Learned embeddings for every position.
- Sinusoids of different frequencies:

$$\begin{aligned}\mathrm{PE}_{(pos,2i)} &= \sin\left(pos/10000^{2i/d}\right)\\ \mathrm{PE}_{(pos,2i+1)} &= \cos\left(pos/10000^{2i/d}\right)\end{aligned}$$

This choice of functions should allow the model to attend to relative positions, since for any fixed k, PE_{pos+k} is a linear function of PE_{pos}, because

$$\begin{aligned}\mathrm{PE}_{(pos+k,2i)} &= \sin\left((pos+k)/10000^{2i/d}\right)\\ &= \sin\left(pos/10000^{2i/d}\right)\cdot\cos\left(k/10000^{2i/d}\right)+\cos\left(pos/10000^{2i/d}\right)\cdot\sin\left(k/10000^{2i/d}\right)\\ &= \mathit{offset}_{(k,2i)}\cdot\mathrm{PE}_{(pos,2i)}+\mathit{offset}_{(k,2i+1)}\cdot\mathrm{PE}_{(pos,2i+1)}.\end{aligned}$$

Positional Embeddings

Sinusoids of different frequencies

In the original description of positional embeddings (the one used on the previous slide), the sines and cosines are interleaved, so for $d = 6$, the positional embeddings would look like:

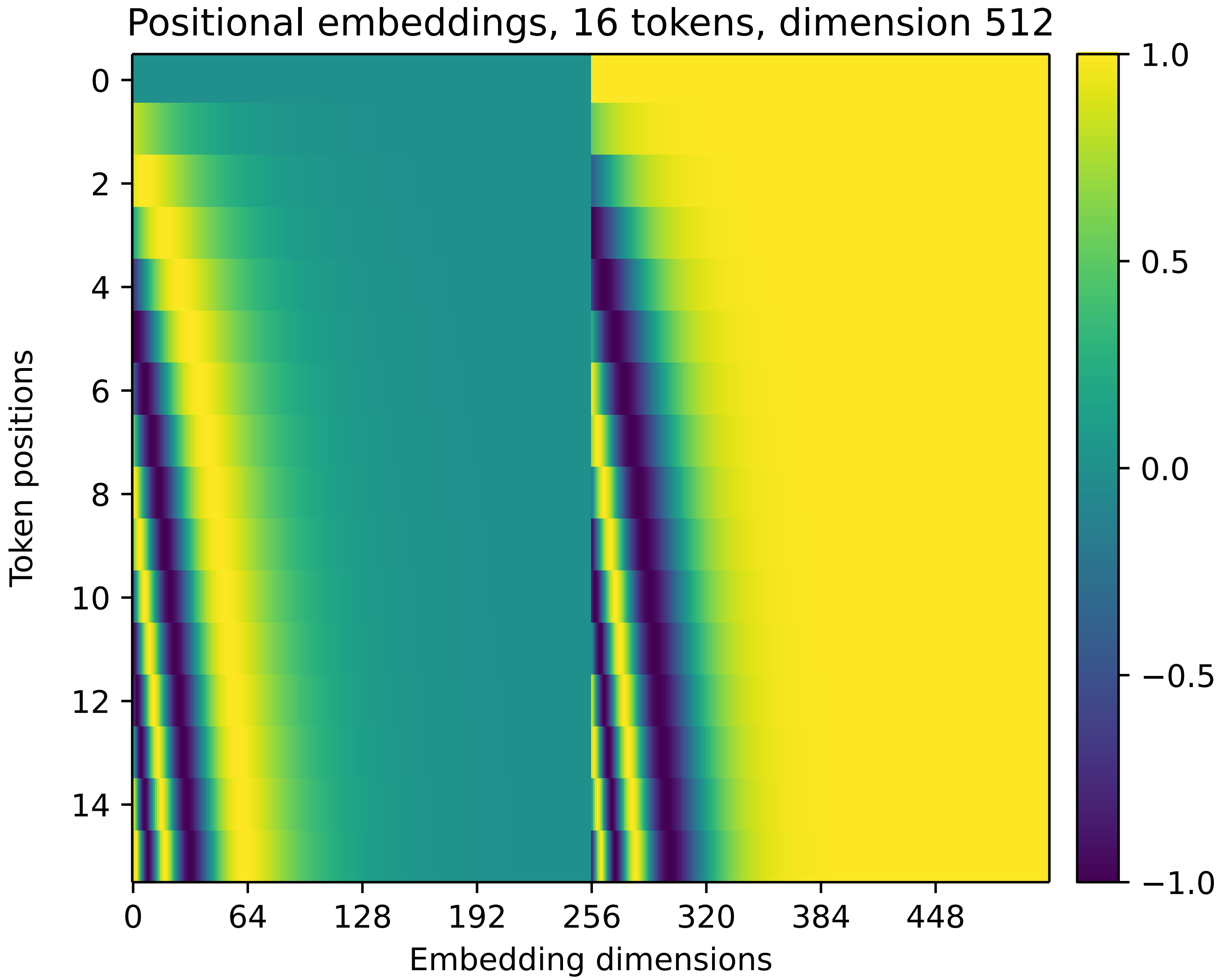
$$\mathrm{PE}_{pos} = \left(\sin\left(\frac{pos}{10000^0}\right), \cos\left(\frac{pos}{10000^0}\right), \sin\left(\frac{pos}{10000^{1/3}}\right), \cos\left(\frac{pos}{10000^{1/3}}\right), \sin\left(\frac{pos}{10000^{2/3}}\right), \cos\left(\frac{pos}{10000^{2/3}}\right)\right).$$

However, in practice, most implementations concatenate first all the sines and only then all the cosines:

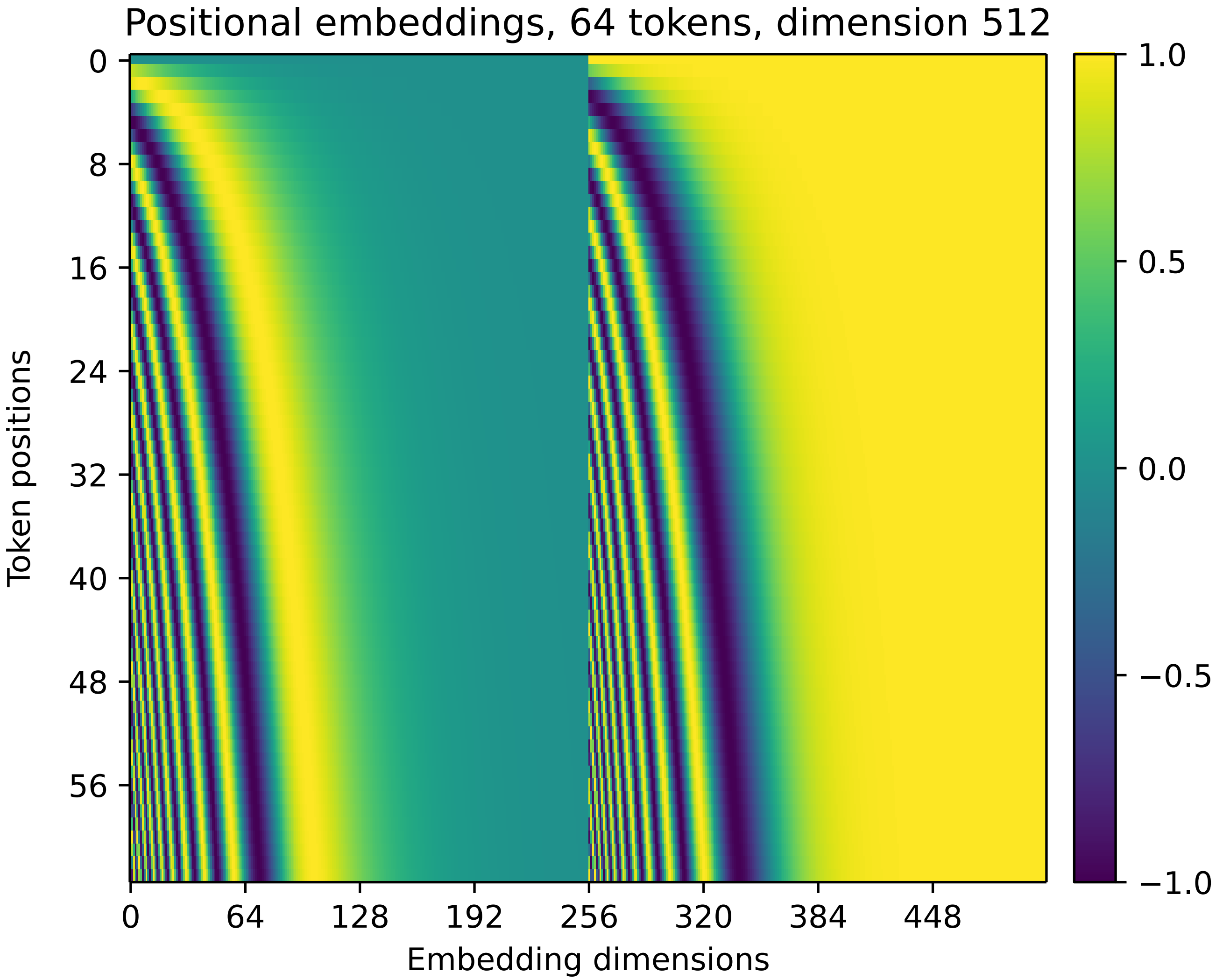
$$\widehat{\mathrm{PE}}_{pos} = \left(\sin\left(\frac{pos}{10000^0}\right), \sin\left(\frac{pos}{10000^{1/3}}\right), \sin\left(\frac{pos}{10000^{2/3}}\right), \cos\left(\frac{pos}{10000^0}\right), \cos\left(\frac{pos}{10000^{1/3}}\right), \cos\left(\frac{pos}{10000^{2/3}}\right)\right).$$

This is also how we visualize the positional embeddings on the following slides.

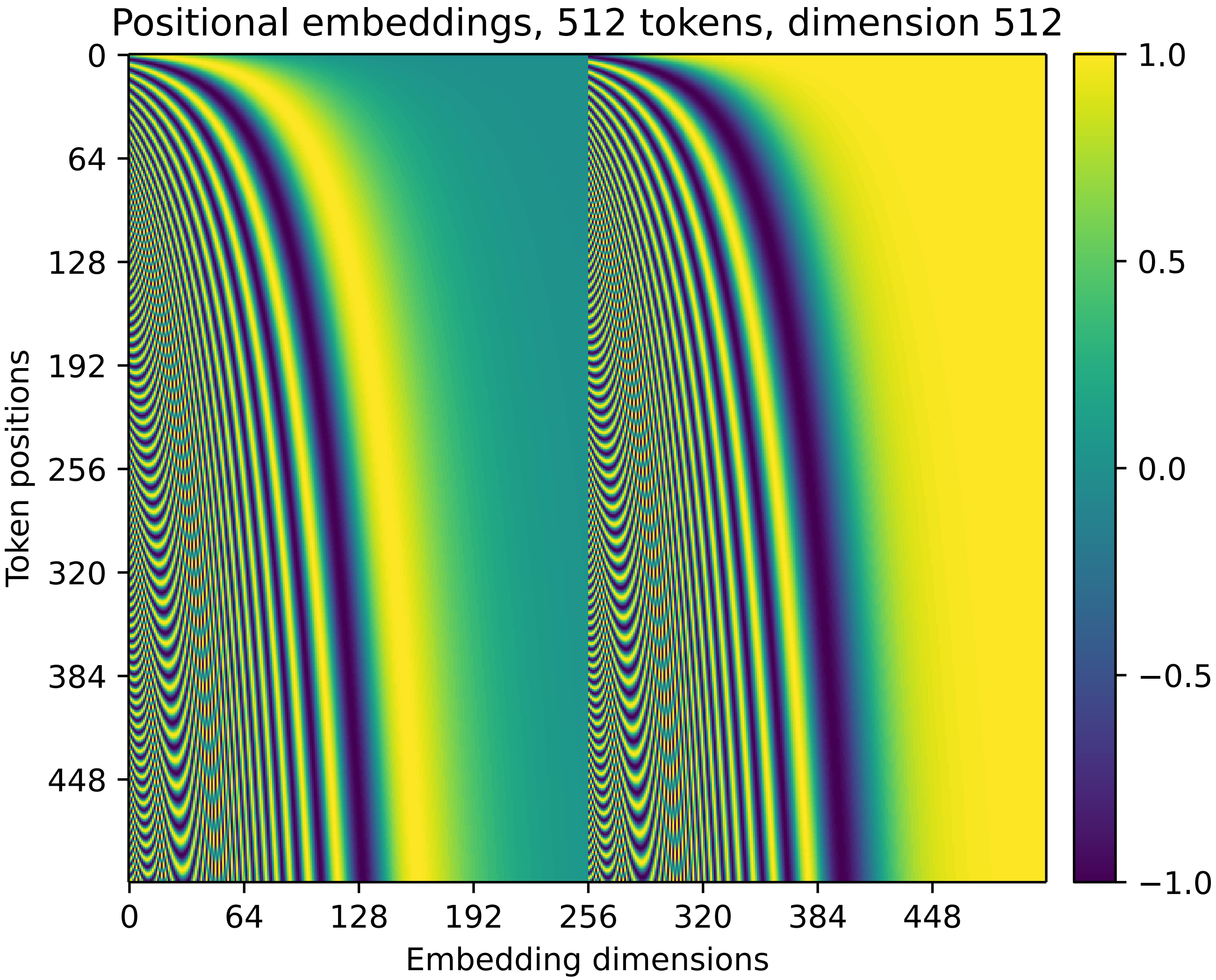
Transformer – Positional Embeddings



Transformer – Positional Embeddings



Transformer – Positional Embeddings



Transformer – Training

Regularization

The network is regularized by:

- dropout of input embeddings,
- dropout of each sub-layer, just before it is added to the residual connection (and then normalized),
- label smoothing.

Default dropout rate and also label smoothing weight is 0.1.

Parallel Execution

Because of the *masked attention*, training can be performed in parallel.

However, inference is still sequential.

Transformer – Training

Optimizer

Adam optimizer (with $\beta_2 = 0.98$, smaller than the default value of 0.999) is used during training, with the learning rate decreasing proportionally to inverse square root of the step number.

Warmup

Furthermore, during the first $\mathit{warmup\_steps}$ updates, the learning rate is increased linearly from zero to its target value.

$$\mathit{learning\_rate} = \frac{1}{\sqrt{d_\textrm{model}}} \min\left(\frac{1}{\sqrt{\mathit{step\_num}}}, \frac{\mathit{step\_num}}{\mathit{warmup\_steps}} \cdot \frac{1}{\sqrt{\mathit{warmup\_steps}}}\right).$$

In the original paper, 4000 warmup steps were proposed.

Note that the goal of warmup is mostly to prevent divergence early in training; the Pre-LN configuration usually trains well even without warmup.

Transformers Results

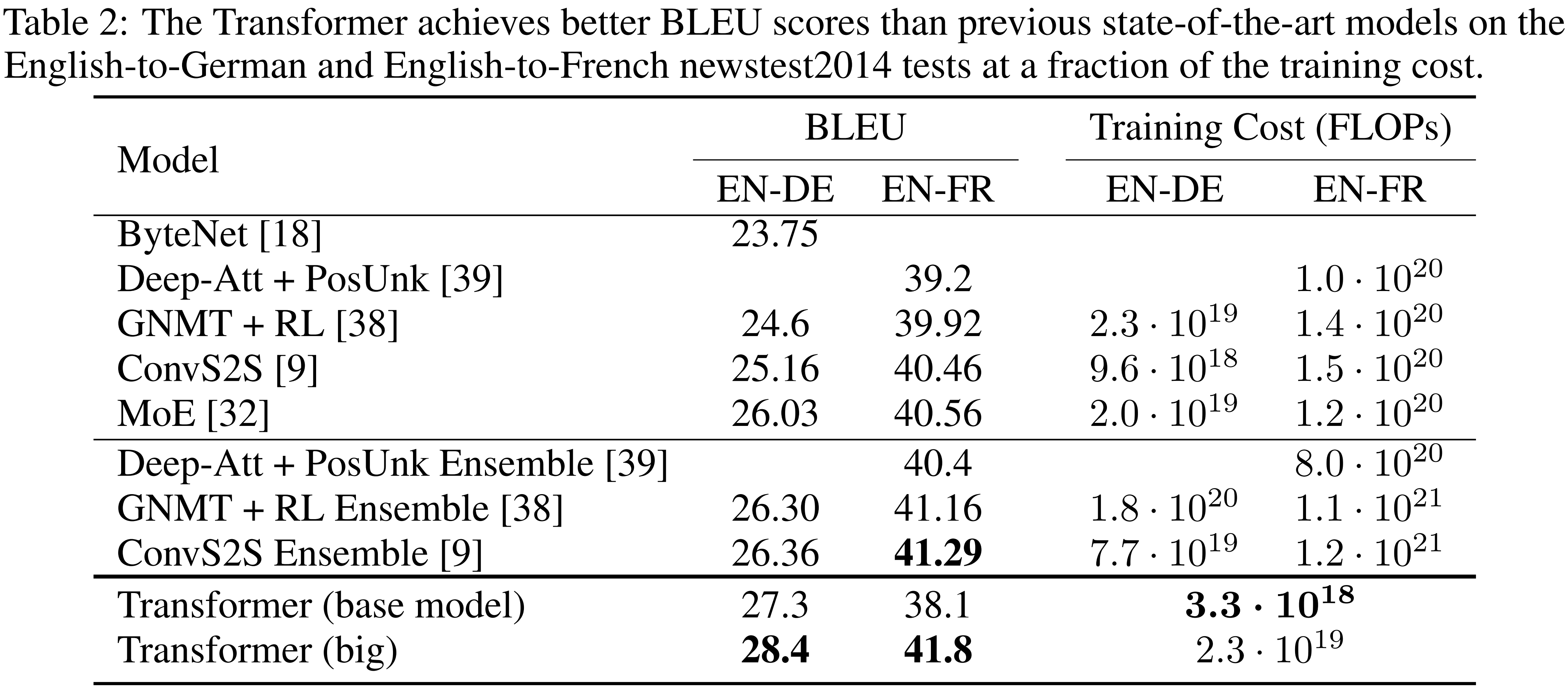


Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

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	$\mathbf{3.3 \cdot 10^{18}}$	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

Table 2 of "Attention Is All You Need", https://arxiv.org/abs/1706.03762

Wordpieces were constructed using BPE with a shared vocabulary of about 37k tokens.

Transformers Ablations on En→De newtest2014 Dev

	N	d_{model}	d_{ff}	h	d_k	d_v	P_{drop}	ϵ_{ls}	train steps	PPL (dev)	BLEU (dev)	params $\times 10^6$
base	6	512	2048	8	64	64	0.1	0.1	100K	4.92	25.8	65
(A)				1	512	512				5.29	24.9	
				4	128	128				5.00	25.5	
				16	32	32				4.91	25.8	
				32	16	16				5.01	25.4	
(B)					16					5.16	25.1	58
					32					5.01	25.4	60
(C)	2									6.11	23.7	36
	4									5.19	25.3	50
	8									4.88	25.5	80
		256			32	32				5.75	24.5	28
		1024			128	128				4.66	26.0	168
			1024							5.12	25.4	53
			4096							4.75	26.2	90
(D)							0.0			5.77	24.6	
							0.2			4.95	25.5	
								0.0		4.67	25.3	
								0.2		5.47	25.7	
(E)	positional embedding instead of sinusoids									4.92	25.7	
big	6	1024	4096	16			0.3		300K	**4.33**	**26.4**	213

Table 4 of "Attention Is All You Need", https://arxiv.org/abs/1706.03762

The PPL is *perplexity per wordpiece*, where perplexity is $e^{H(P)}$, i.e., e^{loss} in our case.

Main Takeaway

Generally, Transformer provides more powerful sequence-to-sequence architecture and also sequence element representation architecture than RNNs, but requires **substantially more** data.

3D Visualization of a Decoder-only Model

On https://bbycroft.net/llm you can find a 3D visualization with the description of the Transformer computation steps of several GPT models. The GPT models are language models (they estimate conditional probability of a word given its previous context), and therefore consist purely of the decoder part of a Transformer (so they do not contain neither an encoder nor encoder-decoder attention; consequently, all their self-attentions are masked).