Transformers

Abstract

This chapter presents the Transformer model in detail.

1. Introduction

Textual data is inherently sequential. Accordingly, *recurrent neural networks* represent a natural fit for handling most NLP tasks. Until recently, the state-of-the-art NLP models were mainly based on recurrent architectures, like LSTM or GRU neural networks [1, 2]. But recurrent architectures present major limitations. First, the processing of input sequences cannot be fully parallelized. Secondly, long-term dependencies between inputs cannot be optimally captured.

In 2017, in a seminal paper entitled "Attention Is All You Need", the *Transformer* model was proposed as a relevant solution to these issues [3]. The model consists of an encoder-decoder feedforward architecture augmented with a self-attention mechanism. According to the authors, their work "propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely" [3]. The feed-forward architecture alleviates the parallelization issue and the self-attention mechanism is capable of focusing on any part of the input, to the limit of the model's dimensionality. The main message of the paper is that the attention mechanism can on its own outperform complex architectures. The Transformer model achieved impressive improvements on machine translation tasks. Most of all, it opened the way for a new generation of language models that broke the barriers of NLP.

In particular, the *Generative Pre-Training (GPT)* models (GPT, GPT-2, GPT-3, GPT-4) are children of the Transformer [4–6]. Their architecture consists of several decoder blocks of the Transformer stacked upon one another. These models can leverage an impressive amount of linguistic information from unlabeled text data, which in turn, can be efficiently transferred to the learning of most common NLP tasks. The GPT model employs a 2-stage training process: an unsupervised pre-training followed by a supervised fine-tuning. During pre-training, a language model is learned using a large corpus of unlabeled text. During fine-tuning, the architecture of the pre-trained model is slightly modified to learn a specific downstream task, using labeled data [4]. In contrast to GPT, the GPT-2 and GPT-3 models are not fine-tuned, but pre-trained on larger and larger text corpora. By means of a simple task conditioning procedure, they can achieve multitask learning via unsupervised pre-training only. In fact, these models show remarkable performance in zero-shot, one-shot and few-shot learning on most common downstream NLP tasks [5, 6].

In 2019, another offspring of the Transformer, the *Bidirectional Encoder Representations from Transformers (BERT)* model has been released [7]. It consists of several encoder (rather than decoder) blocks stacked together. It also uses the 2-stage pre-training and fine-tuning procedure as well. But in contrast to GPT, the left-to-right pre-training process is replaced by a *bidirectional* pre-training consisting of two steps: (i) a masked language model (MLM) task, and (ii) a next sentence prediction task (NSP). This pre-training provides the language model with a representation of both left and right contexts. Fine-tuning is also achieved via minimal modifications of the language model's architecture. With these improvements, BERT achieves impressive performance on most NLP tasks.

The pre-trained BERT model can not only be used as a language model, but also as a word or sentence dynamic *embedding*. In this case, the input text is encoded into a vectorial representation at the character, word, or sentence level, before being fed to a subsequent model for the task at hand. The BERT embedding is *dynamic* is the sense that a same word will not yield the same embedding depending on its left and right contexts. This feature contrasts with the previous embeddings which are mostly *static*. In fact, the BERT embedding takes over most previous pre-trained embeddings like word2vec [8], GloVe [9], FastText [10], ELMo [11] and Flair [12], and appears nowadays as a best choice.

But like all Transformer-based models, BERT is highly resource consuming, containing from 110M to 340M parameters. And while the pre-trained model is available off-the-shelf, the fine-tuning process remains computationally expensive. In an effort to address these issues, lighter and faster versions of the model have been proposed [13, 14]. An enlightening presentation of several Transformer-based models can be found in Jay Alammar's blog [15].

2. Encoder-Decoder

Before the Transformer architecture is introduced, some considerations about the encoder-decoder architecture are recalled. An *encoder-decoder* architecture is composed of two models assembled together, an *encoder* and a *decoder* (see Figure 2). In summary, the encoder encodes the successive input words, step by step, and generates a *context vector* that represents the embedding of the whole input sequence. The decoder takes the context vector as a hidden state, and outputs a sequence of decoded words, step by step. A typical application for the encoder-decoder architecture is machine translation, where the input sequences are English sentences and the output ones are their corresponding French translations, for instance.

More specifically, consider the case where the encoder-decoder architecture is composed of two recurrent neural networks (RNNs). Recall that the dynamics of an RNN is given by the following equations

$$\begin{aligned} \mathbf{h^t} &= f(\mathbf{x^t}, \mathbf{h^{t-1}}; \mathbf{\Theta_f}) \\ \mathbf{y^t} &= g(\mathbf{h^t}; \mathbf{\Theta_g}) \end{aligned} \tag{1}$$

where $\mathbf{h^{t-1}}$ and $\mathbf{h^t}$ are the hidden states of the RNN at time steps $t-1$ and t, respectively, $\mathbf{x^t}$ and $\mathbf{y^t}$ are the input and output of the RNN at time step t, respectively, f is the equation of some recurrent layers, (e.g., LSTMs or GRUs) involving parameters $\mathbf{\Theta_f}$ (baises and weights), and g is the equation of some feedforward layers (e.g., fully-connected or softmax) involving parameters $\mathbf{\Theta_g}$ (baises and weights). According to this equation, the network computes its current state $\mathbf{h^t}$ as a function f of its previous state $\mathbf{h^{t-1}}$ and its current input $\mathbf{x^t}$, and it produces the current output $\mathbf{y^t}$ as a function g of its current state $\mathbf{h^t}$, as illustrated in Figure 1.

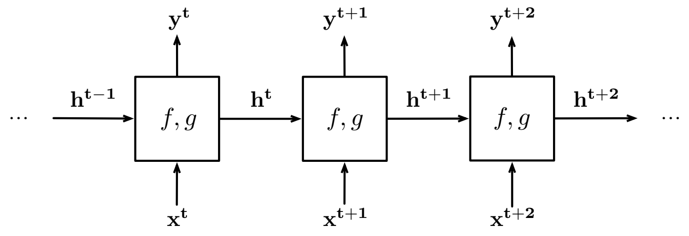

Figure 1: Dynamics of an RNN unfolded in time. The current state $\mathbf{h^t}$ of the network is computed as a function f of the previous state $\mathbf{h^{t-1}}$ and the current input $\mathbf{x^t}$, for each $t = 1, \ldots, N$. The current output $\mathbf{y^t}$ is computed as a function g of the current state $\mathbf{h^t}$, for each $t = 1, \ldots, N$.

Suppose that the encoder and decoder are two RNNs given by the functions f_e, g_e and f_d, g_d, respectively. The encoding-decoding process, illustrated in Figure 2, can be described as follows. Consider some input sentence, $w_e^1, \ldots, w_e^N$, where each w_e^i is a token (word) of the input sentence. The input tokens $w_e^1, \ldots, w_e^N$ are first embedded into input vectors $\mathbf{x_e^1}, \ldots, \mathbf{x_e^N}$. These input vectors are then passed to the encoder RNN with some initial state $\mathbf{h_e^0}$, producing the sequence of states and outputs $\mathbf{h_e^1}, \ldots, \mathbf{h_e^N}$ and $\mathbf{y_e^1}, \ldots, \mathbf{y_e^N}$, respectively. The outputs of the encoder are generally discarded. It is worth noting that the last hidden state $\mathbf{h_e^N}$ contains some memory (i.e., some dependence) of all the input vectors $\mathbf{x_e^1}, \ldots, \mathbf{x_e^N}$, by construction. It is usually called the *context vector*, and represents an *embedding* of the whole input sequence. This context vector is then taken as the initial state of the decoder $\mathbf{h_d^0} = \mathbf{h_e^N}$, and the decoding process can begin. A first input token $w_d^1 =$ [START] is embedded into the input vector $\mathbf{x_d^1}$. When $\mathbf{h_d^0}$ and $\mathbf{x_d^1}$ are passed to the decoder, it produces a hidden state $\mathbf{h_d^1}$ and an output $\mathbf{y_d^1}$. The output $\mathbf{y_d^1}$ corresponds to some decoded word w_d^1. This word is embedded and passed to the decoder as a second input vector

$\mathbf{x_d^2}$. With $\mathbf{h_d^1}$ and $\mathbf{x_d^2}$, the decoder produces a hidden state $\mathbf{h_d^2}$ and an output $\mathbf{y_d^2}$. The output $\mathbf{y_d^2}$ corresponds to some decoded word w_d^2, which is embedded and passed to the decoder as a third input vector $\mathbf{x_d^3}$. The process continues until the output token w_d^M = [END] is produced. The sequence of tokens $w_d^1, \ldots, w_d^M$ obtained via this process is the decoding of the input sequence $w_e^1, \ldots, w_e^N$.

Note that, in the case where the encoder and the decoder are RNNs, the encoding-decoding process cannot be parallelized. More specifically, an RNN cannot process its successive inputs $\mathbf{x^0}, \mathbf{x^1}, \mathbf{x^2}, \ldots$ in parallel, since each current input $\mathbf{x^t}$ requires the preceding hidden state $\mathbf{h^{t-1}}$ to be computed before being processed, according to Eq. (1). And the state $\mathbf{h^{t-1}}$ precisely comes from the processing of the previous input $\mathbf{x^{t-1}}$.

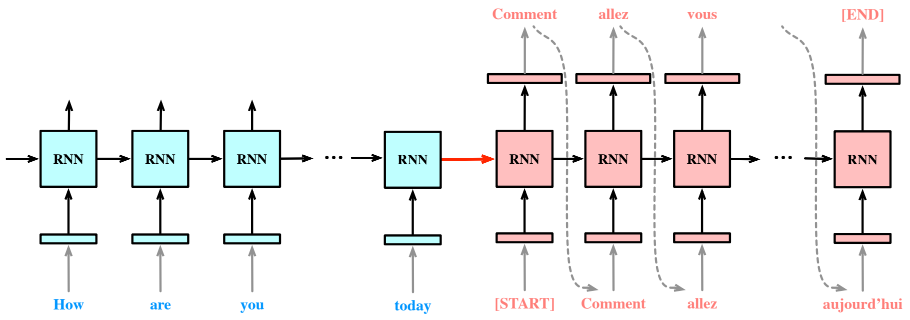

Figure 2: Encoder-decoder architecture. The input words (in blue) are embedded (blue vectors) and passed to the RNN encoder (blue boxes). At each step, the RNN encoder computes its current state from its previous state and its current embedded input. After all inputs have been processed, the final state reached by the RNN encoder (red arrow), the context vector, represents an embedding of the whole input sequence. The context vector (red arrow) is then plugged into the RNN decoder (red boxes). At each step, the RNN decoder computes its current state and current output (upper red vector) from its previous state and from the the embedding of the word that has been previously decoded (lower red vector).

3. Model

Transformer architecture. The *Transformer* model consists of an encoder-decoder architecture, where both the encoder and the decoder are feedforward instead of recurrent neural networks [3]. This architecture makes intensive use of the attention mechanism. The feedforward architecture enables paralleization, and the attention implements memory capabilities, to the limit of the model's dimensionality. This architecture is illustrated in Figure 3 and 4.

The *encoder* consists of a stacking of 6 identical blocks, each of which being composed of two sub-blocks: a multi-head self-attention mechanism, and a simple fully connected layer. Residual connections followed by layer normalization around each of the two sub-blocks are also added. An encoder block is illustrated in Figures 5 and 6.

The *decoder* also consists of a stacking of 6 identical blocks, each of which being composed of three sub-blocks: a multi-head self-attention mechanism similar to that of the encoder, a new multi-head masked attention mechanism operating over the output of the encoder stack, and a simple fully connected layer. Here again, residual connections followed by layer normalization around each of the three sub-blocks are added. In the decoder, the self-attention mechanism includes an additional masking functionality, which ensures that the attention is computed only on the basis of the backward context, since the subsequent tokens have, in theory, not been decoded yet. An decoder block is illustrated in Figures 5 and 6.

Tokenization, embedding and positional encoding. The first stage of the encoder and decoder involves the tokenization, embedding and positional encoding of a given text, illustrated in Example 1 and Figure 7. *Tokenization* is the process of splitting a text into a list of tokens. These tokens are then replaced by integers, their token ids, which correspond to indices in a pre-defined vocabulary of about 37 K sub-words. The *embedding* process converts the input ids into dense vectors of dimension 512. Towards this purpose, a pre-defined lookup dense matrix of dimension 30K $\times$ 512 is employed. The k-th row of this matrix corresponds to the embedding of the sub-word with token id k (cf. Figure 7). Using this static embedding, the input text is converted into a sequence of embedded inputs. But this embedding process does actually not capture the positions of the tokens in

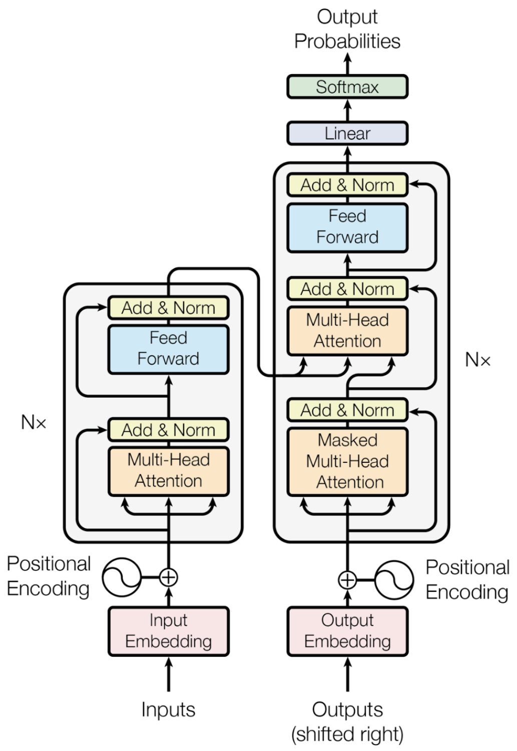

Figure 3: The Transformer model. The Transformer is a feedforward neural network arranged in an encoder-decoder architecture. It is composed of $N = 6$ encoding bocks stacked upon one another, followed by $N = 6$ decoding blocks also stacked together. The model makes intensive use of attention. Figure taken from the original paper [3].

145 the initial text, which is a crucial information. Hence, the relative positions of the token ids (i.e., $0, 1, 2, \dots$) are encoded into dense vectors of dimension 512 called *positional encodings*, using the following formula

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

where and pos is the position of the input token $(0, 1, 2, \dots)$, i is the encoding's dimension $(0 \leq i \leq 255)$, and $d = 512$ is the embedding's dimension. The
150 positional encodings are then added to the embeddings to form the *embedded inputs*, that can further be passed to the encoder or the decoder (cf. Figure 7).

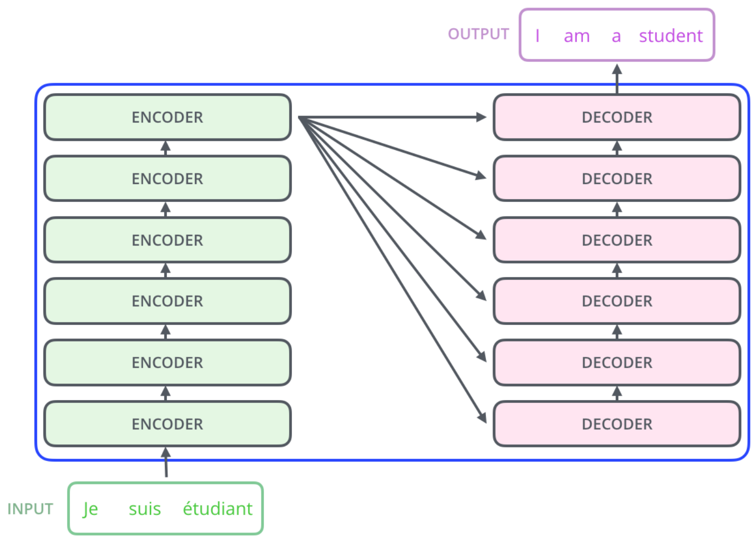

Figure 4: Encoder-decoder block architecture of the Transformer. Figure taken from Jay Alammar's blog [15].

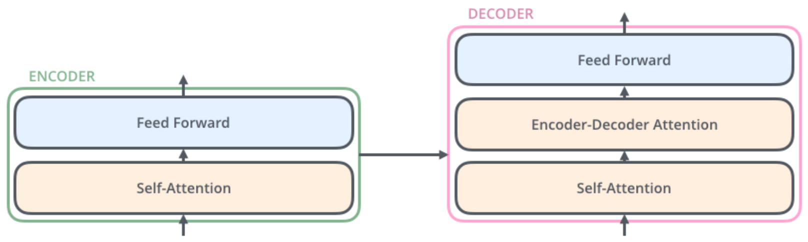

Figure 5: An encoder and a decoder block of the Transformer. Figure taken from Jay Alammar's blog [15].

Example 1. This example shows how an input text is tokenized, converted into token ids, embedded into input vectors, and added to positional encoding vectors. After this process, the inputs can be passed to the Transformer.

```
# Original Sentence
"Let’s learn deep learning!"
# Tokenized Sentence
```

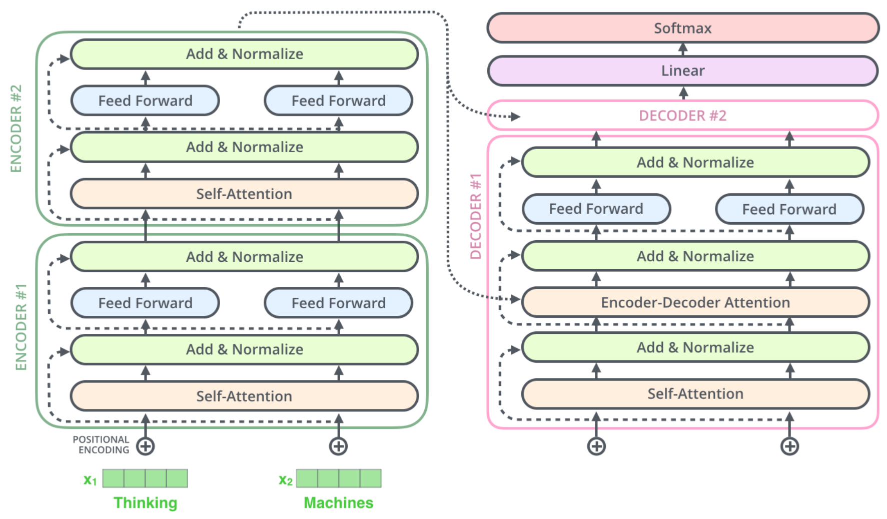

Figure 6: Details of two encoder and one decoder blocks of the Transformer. Figure taken from Jay Alammar's blog [15].

```
["Let", "'", "s", "learn", "deep", "learning", "!"]
# Adding [CLS] and [SEP] Tokens
["[CLS]", "Let", "'", "s", "learn", "deep", "learning", "!", "[SEP]"]
# Padding
["[CLS]", "Let", "'", "s", "learn", "deep", "learning", "!", "[SEP]",
"[PAD]"]
# Converting to IDs
[101, 2421, 112, 188, 3858, 1996, 3776, 106, 102, 0]
```

Attention, self-attention, and multi-head attention. We now introduce the *multi-head attention* involved in the Transformer. We describe the self-attention present in each encoder block, as well as the masked self-attention and the attention that happen at the first and second stage of each decoder block, respectively.

Intuitively, an attention mechanism is a process which, for any input sequence $(\mathbf{x^1}, \mathbf{x^2}, \ldots, \mathbf{x^N})$ and any element $\mathbf{y}$, computes a sequence of weights $(w_1, w_2, \ldots, w_N)$, called *attention weights*. Each attention weight w_i represents

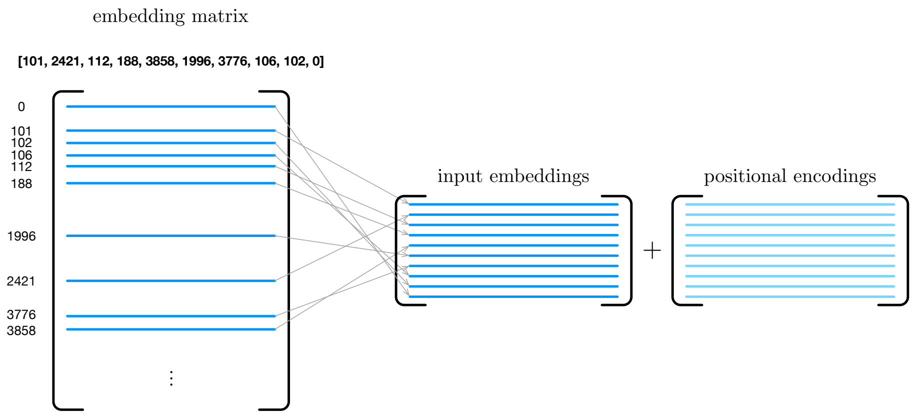

Figure 7: The sequence of token ids are embedded into dense vectors of dimension 512, using a static embedding lookup matrix of dimension 37 K $\times$ 512 (large matrix). The k-th row of this matrix corresponds to the dense embedding of the sub-word with token id k. For each embedded token id (dark blue vector), its positional encoding of dimension 512 is computed (light blue vector) and added to it.

the importance that element $\mathbf{y}$ attaches to – or equivalently, the *attention* that $\mathbf{y}$ puts on – the element $\mathbf{x^i}$ of the sequence. The computation ensures that the attention weights sum to 1, i.e., $\sum_{i=1}^{N} w_i = 1$, like probabilities. The larger the attention weight w_i, the more attention is put on $\mathbf{x^i}$ by element $\mathbf{y}$. Whenever $\mathbf{y}$ is an element of the sequence $(\mathbf{x^1}, \mathbf{x^2}, \ldots, \mathbf{x^N})$ itself (i.e., $\mathbf{y} = \mathbf{x^k}$ for some $1 \leq k \leq N$), we encounter a situation where the attention of $\mathbf{y} = \mathbf{x^k}$ is put on every element of the sequence $(\mathbf{x^1}, \mathbf{x^2}, \ldots, \mathbf{x^N})$, including itself. This process is referred to as *self-attention*.

The attention mechanism used in the original paper is referred to as *scaled dot-product attention* [3]. This mechanism has been conceived by analogy with a *retrieval system* based on *queries (Q)*, *keys (K)* and *values (V)*. Suppose that a set of objects is stored in a key-value database. The *keys* are the objects identifiers and the *values* are the objects themselves. For a given database *query*, the retrieval system will first return a list of importance scores, the *attention weights*, which describe how much do the successive *keys* match with the *query*. Then, using these attention weights, the answer to the query will be computed as the weighted combination of the *values* associated to the *keys*. In practice, the *queries*, *keys* and *values* are all vectors, and multiple *queries* are performed in parallel. Most importantly, the generation of the *queries*, *keys* and *values* is learned by the model in a precise sense described below.

Example 2. Consider the following movie review:

"Not a bad **movie**, typical action movie with a reasonable storyline."

Suppose that this review is represented as sequence of embedded words (or embedded tokens), and stored in a *key-value* database of the following form:

keys	0	1	2	3	4	5	6	7	8	9	10	11	12
values	Not	a	bad	movie	,	typical	action	movie	with	a	reasonable	storyline	.

Now, suppose also that the embedded word "movie" (in bold) makes the following *query* to the database: "Which are the adjectives associated to myself in this review?" As an answer, the retrieval system will first provide a list of *attention weights* such that the *keys* 2, 5 and 10, corresponding to the values "x_2 = bad", "x_5 = typical" and "x_{10} = reasonable" , obtain high attention weights w_2, w_5 and w_{10}, respectively, while the other *keys* will obtain low or zero weights. Then, the answer given to this *query* by the retrieval system will be the weighted combination of *values* given by these *attention weights*, namely, $\sum_{i=0}^{12} w_i x_i$. This corresponds to some weighted average of the adjectives associated with the word "movie" in this review. This process describes the attention mechanism associated to a single *query*, but in practice, every element of the sequence makes its own *query*, and all the *queries* are performed in parallel.

Formally, the *scaled dot-product attention* is composed of three fully-connected layers, a softmax layer, a scaling and a matrix multiplication operations, and potentially, an additional masking operation in the case of *masked attention*. This attention mechanism is illustrated in Figures 8, 9, 10 and 11 and given by the following formula explained below:

$$\text{Attention}(Q, K, T) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V. \tag{2}$$

Before entering into the detailed description of the mechanism, note that a sequence of vectors is represented and implemented as a matrix, where the successive rows of the matrix are the successive vectors of the sequence. To begin with, three sequences of vectors, the matrices Q_0, K_0 and V_0, associated with queries (Figure 9, bottom orange matrix), keys and values (Figure 9, bottom blue matrix), are given as input. In this context, the input matrices associated with keys and values are the same, i.e., $K_0 = V_0$. The three matrices are passed through three fully connected layers with weights W^Q, W^K and W^K, respectively, which transforms them into three corresponding matrices Q, K and V called *queries* (Figure 9, middle orange matrix), *keys* and *values* (Figure 9, middle blue matrix), respectively, i.e.

$$Q = W^Q Q_0 \quad K = W^K K_0 \quad V = W^V V_0. \tag{3}$$

Note that the weights W^Q, W^K and W^K are learned during training, meaning that the model learns how the queries, keys and values should be generated from the inputs in order to achieve good performance, in the sense of reducing the loss function. At this stage, each row $(Q)_i$ of Q, $(K)_j$ of K and $(V)_k$ of V represents a single *query*, *key* and *value*, respectively, and each query $(Q)_i$ operates on all the keys and values, i.e., on the full matrices K and V. The matrix representation ensure that all queries are performed in parallel.

As the next step, the *attention scores* are computed as the matrix product QK^T (Figure 9, first red matrix). Accordingly, the attention score that the i-th query $(Q)_i$ puts on the j-th key $(K)_j$ is computed as the *dot-product* $(Q)_i \cdot (K)_j$. The attention scores that the i-th query $(Q)_i$ puts on all the keys K is given by the i-th row of QK^T. The attention that all the queries Q puts on all the keys K is given by the matrix product QK^T.

Afterwards, the attention scores are *rescaled* for stability purposes by dividing QK^T by $\sqrt{d_k}$, where $d_k = 512$ is the dimension of the model. The reason for this rescaling is empirical and justified in original paper as follows [3]: "We suspect that for large values of d_k, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients. To counteract this effect, we scale the dot products by $\frac{1}{\sqrt{d_k}}$." Then, the *attention weights* are obtained from the attention scores by applying a *softmax* operation row-wise to the matrix $\frac{1}{\sqrt{d_k}}QK^T$ (Figure 9, second red matrix). We recall that the softmax operation is defined by

$$\text{softmax}(x_0, \ldots, x_n) = \left(\frac{e^{x_0}}{\sum_{i=0}^{n} e^{x_i}}, \ldots, \frac{e^{x_n}}{\sum_{i=0}^{n} e^{x_i}} \right).$$

In this way, the attention weights associated to each query $(Q)_i$, which correspond to the i-th row of the matrix softmax $\left(\frac{1}{\sqrt{d_k}}QK^T\right)$, sum to 1.

Finally, the answer $(A)_i$ to each query $(Q)_i$ is given by a weighted sum of all the values V, where the weights are precisely the attention weights that have just been computed (Figure 9, third red matrix). Formally,

$$(A)_i = \text{softmax}\left(\frac{1}{\sqrt{d_k}}QK^T\right)_i (V)_i.$$

When all the queries are performed in parallel, the matrix of answers A is given by

$$A = \text{softmax}\left(\frac{1}{\sqrt{d_k}}QK^T\right)V. \tag{4}$$

To summarize, the attention mechanism applied to the input sequences Q_0, K_0 and V_0 are given by

$$\text{Attention}\left(Q_0, K_0, V_0\right) = \text{softmax}\left(\frac{1}{\sqrt{d_k}}(W^Q Q_0)(W^K K_0)^T\right)(W^V V_0), \tag{5}$$

according to Equations 4 and 3. In this way, the input sequence Q_0 associated with the queries is transformed into another sequence A, whose every element $(A)_i$ puts attention on all elements of the sequence associated with the values K_0.

The mechanism of attention is called *self-attention* when the sequence Q_0 associated with queries is the same as those associated with keys and values K_0 and V_0, respectively. Self-attention is illustrated in Figure 10. Intuitively, this means that every element of the input sequence puts attention on itself as well as all other elements of the sequence to which it belongs. Self-attention is employed in the encoder, to compute attention on the input sequence, and

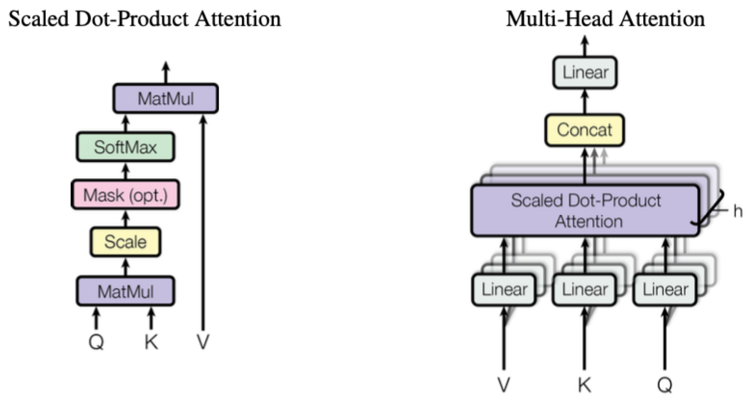

Figure 8: Scaled dot-product attention and its multi-headed version. Figure taken from the original paper [3].

in the decoder to compute attention on the sequence of words that have been decoder so far. By contrast, the attention computed in the second part of the decoder is not self-attention. It builds its queries on the basis of the words that have been decoded so far, but builds its keys and values based on the sequence computed by the encoder.

Regarding the self-attention used in the decoder, a *masking* mechanism needs to be introduced in order to prevent the words that have been decoded so far to put attention on the words that will be decoded in the next steps. Masking is illustrated in Figure 11. Towards this purpose, the upper right diagonal of the attention scores is masked with “$-\infty$” values, so that when the softmax is applied, the associated attention weights become equal to 0. Hence, each row i of the attention weight matrix puts zero attention on the next positions $i+1, i+2, \ldots, N$. This means that each query $(Q)_i$ puts attention only on the keys and values $(K)_j$ and (V_j), for $j \leq i$.

Finally, multiple attention processes are applied in parallel, a mechanism referred to as *multi-head attention* (see Figure 8). In this case, the results of all attention mechanisms – or *attention heads* – are concatenated row-wise before being passed into a fully-connected layer with weights W^O. Therefore, the full

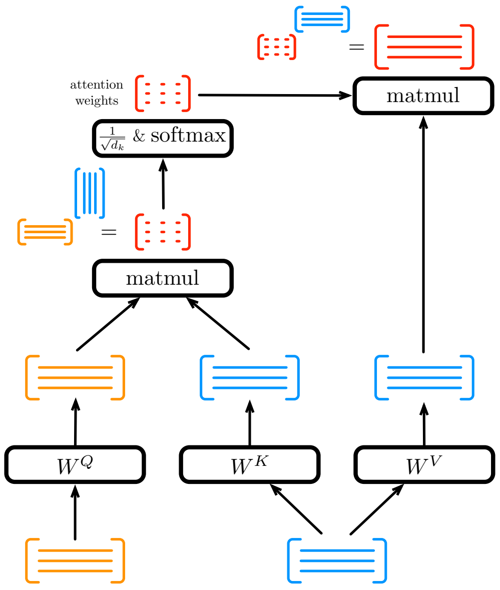

Figure 9: Scaled dot-product attention. Sequences of input vectors associated with queries (bottom orange matrix), keys and values (bottom blue matrices) are provided to the system in the form of matrices. These input matrices are passed through fully connected layers to obtain queries (middle orange matrix), keys and values (middle blue matrices). The queries and keys are multiplied to obtain attention scores (first red matrix). The attention scores are rescaled and passed through a softmax to obtain attention weights (second red matrix). The attention weight are multiplied by the values to obtain the answers to the queries (third red matrix). In this way, the input sequence (bottom orange matrix) is transformed into another attention-based sequence (top red matrix).

attention mechanism is given by the following equations:

$$\text{MultiHead}\left(Q_0, K_0, V_0\right) = \left[\text{Concat}_{i=1}^{h}\left(\text{Attention}_\text{i}\left(Q_0, K_0, V_0\right)\right)\right] W^O \quad (6)$$

$$\text{Attention}_\text{i}\left(Q_0, K_0, V_0\right) = \text{softmax}\left(\frac{1}{\sqrt{d_k}}(W_i^Q Q_0)(W_i^K K_0)^T\right)(W_i^V V_0) \quad (7)$$

275 where W_i^Q, W_i^K and W_i^V are the weights associated with attention head i.

Fully connected layers, residual connections and layer normalization. In each encoder and decoder block, a fully connected layer is added on top of the the attention mechanism. These layers transform the vectors obtain via multi-head

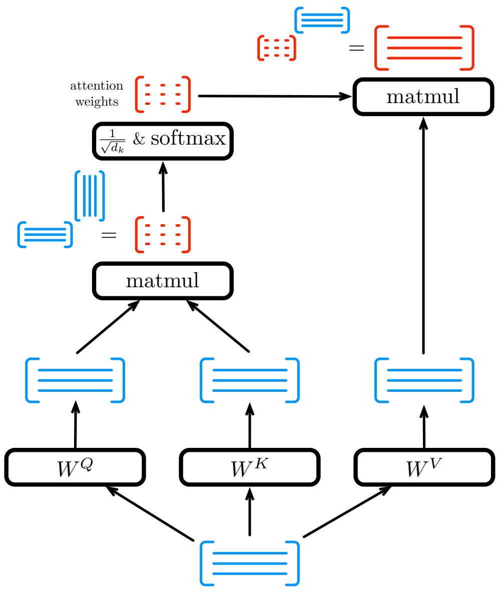

Figure 10: Scaled dot-product self-attention. In this case, the queries, keys and values all come from the same input sequence. The input sequence (bottom blue matrix) is transformed into another attention-based sequence (top red matrix).

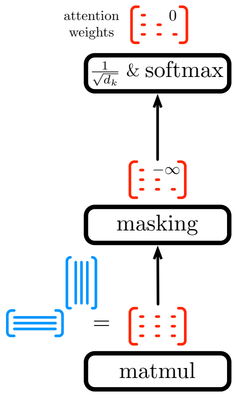

Figure 11: Masking mechanism involved in self-attention. A masking operation is inserted between the matmul and the scaling & softmax operations. The upper right diagonal of the attention scores is masked with $-\infty$, so that when the softmax is computed, the associated attention weights become 0.

attention into vectors of the same dimension, namely $d_k = 512$ in this case. In addition, *residual connections* (or *skip connections*) are employed around each attention and fully connected sub-block [16]. We recall a residual connection around a block consists in summing up the input of the block with its output in order to bypass it. Residual connections have been empirically shown to help convergence in training. This process is followed by *layer normalization* [17]. This technique aims at maintaining the mean and standard deviation of the previous layer activations, within each example, close to 0 and 1, respectively. It has been empirically shown to stabilize the hidden state dynamics, and in turn to reduce the training time of recurrent networks. It also has benefits for feedforward networks, like transformers.

Finally, on top of the last decoder block is a fully connected layer followed by a softmax operation, which transform the output vectors of dimension $d_k = 512$ to probability vectors of dimension equal to the vocabulary size, namely around 37 K. Accordingly, the index of the largest value in each output probability vector corresponds to the token id of the word output by the decoder.

4. Training and Inference

Training. The training and inference modes of the transformer are different. The training process is performed via *teacher forcing*, as described below and illustrated in Figure 12. Suppose that the Transformer is being trained on an English-to-French translation task. Consider some sentence pair (e.g., “Hello, how are you doing?” and “Bonjour, comment allez-vous?”). The English sentence (“Hello, how are you doing?”) is first passed to the encoder. Next, the output of the encoder and the target sentence (“Bonjour, comment allez-vous?”) are passed to the decoder. The decoder thus produces a sequence of output probabilities, which corresponds to the translated sentence. Afterwards, the *cross entropy loss* between the sequence of probabilities output by the decoder and the sequence of probabilities associated with the target sequence is computed, the gradients are computed, and the model is trained via backpropagation. This process is performed by batch.

The approach of feeding the target sequence to the decoder during training is referred to as *teacher forcing*, due to the analogy of a teacher that would provide us with the answer to some problem in order to learn from it. This approach has two advantages over a more straightforward step-by-step inference-like mode. First and foremost, the model can be trained in parallel, which significantly speeds up the training process. Indeed, with the help of the target sequence, the model is able to output all the decoded words in parallel, before applying backpropagation. Secondly, the model learns to predict each word based on the correct preceding words, instead of on potentially erroneous previous predictions. This feature prevents the errors made by the model from getting accumulated along the decoding process.

In the original paper, the Transformer model is trained on two datasets. First, the WMT 2014 English-German dataset which consist of about 4.5 M sentence pairs. In this case, the sentences are encoded using *byte-pair encoding* with a shared source-target vocabulary of about 37 K tokens. Secondly, the significantly larger WMT 2014 English-French dataset consisting of 36 M sentences and a 32 K word-piece vocabulary. Sentence pairs are batched together by approximate sequence length. The chosen optimizer is Adam with parameters $\beta_1 = 0.9$, $\beta_2 = 0.98$ and $\varepsilon = 10^{-9}$ and a variable learning rate. The loss function is the cross entropy (or a variant of it). The larger models were trained for $300K$ steps, which took approximately 3.5 days. For more details, the reader is invited to refer to the original paper [3].

Inference. The inference mode of the Transformer is achieved in a step-by-step mode, as in a Seq2Seq model. This process is illustrated in Figure 13. At the beginning, the whole input sequence is passed to the encoder, producing a corresponding encoded sequence. This encoded sequence as well as a [START] token is then given to the decoder. The decoder generates an output probability vector corresponding to the first decoded word. At the next time step, this first decoded word is appended to the [START] token and fed back to the decoder,

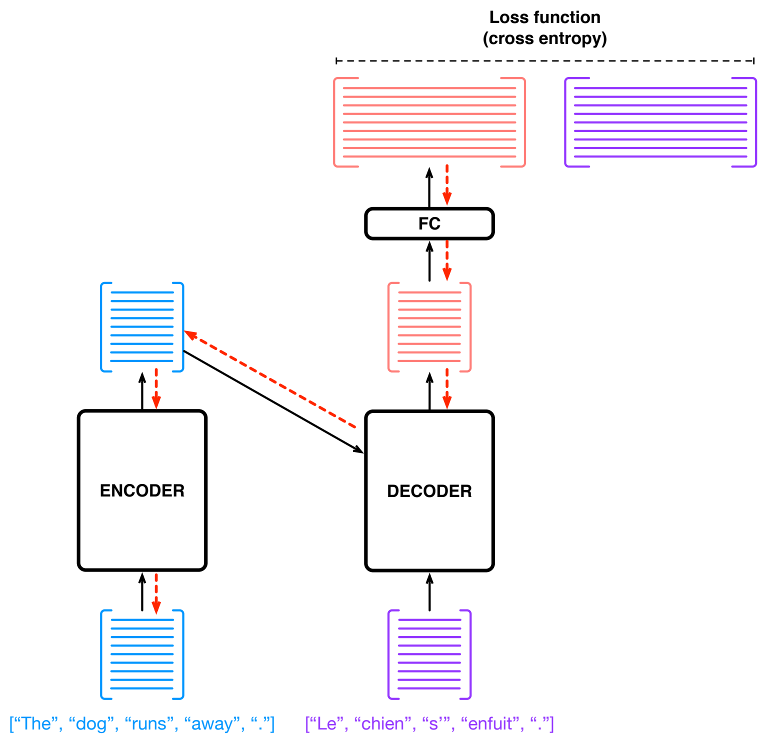
Loss function
(cross entropy)
FC
ENCODER
DECODER
["The", "dog", "runs", "away", "."]
["Le", "chien", "s'", "enfuit", "."]

Figure 12: Training process of a Transformer. The input sequence ("The dog runs away.") is passed to the encoder, and the target sequence ("Le chien s'enfuit.") is provided to the decoder in a teacher forcing way. With this information, the network outputs a decoded sequence. The probability sequence output by the model (upper red matrix) is then compared to that associated with the target sequence (upper purple matrix). The cross entropy loss between the two sequences is then computed and the network is trained via backpropagation (red dashed backward arrows).

together with the encoded sequence. The decoder then generates a sequence output probability vectors whose last element corresponds to the second decoded word. At the following time step, this second decoded word is appended to the sequence of tokens that have been decoded so far and fed back to the decoder, together with the encoded sequence. The decoder then produces the third decoded word. And so on and so forth. The decoding process continues until the end-of-sentence token [END] is reached.

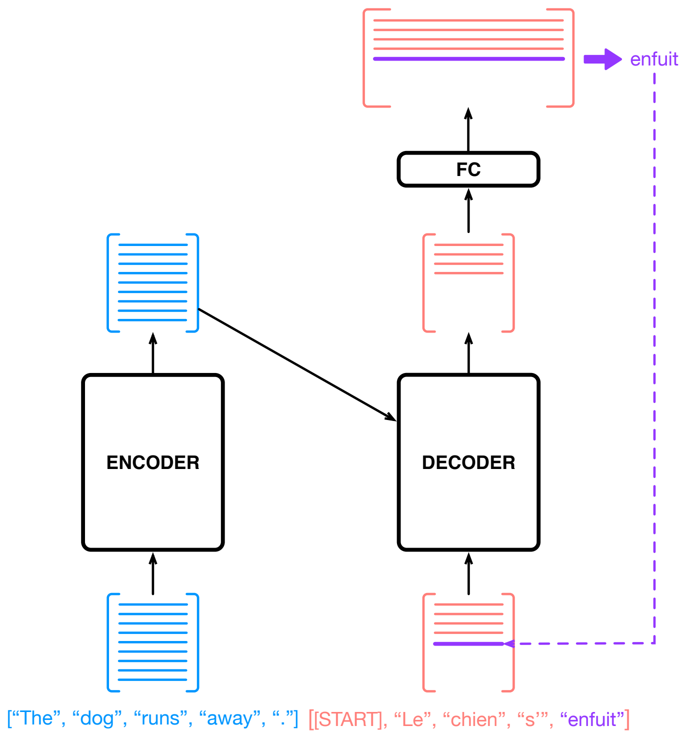
enfuit
FC
ENCODER
DECODER
["The", "dog", "runs", "away", "."]
[[START], "Le", "chien", "s'", "enfuit"]

Figure 13: Inference process of a Transformer. The input sequence ("The dog runs away.") is passed to the encoder, producing an encoded sequence (upper blue matrix). At each time step, this encoded sequence together with the sequence of words that have been decoded so far ("[START] Le chien s'") are passed to the decoder. The decoder generates a corresponding sequence of probability outputs, whose last element (purple vector) corresponds to the last decoded word ("enfuit"). Afterwards, this last decoder word is appended to the previous decoded words, and this new sequence of decoded words is fed back to the decoder (purple arrow). The decoding process continues until the end-of-sentence token is reached.

References

[1] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural Computation 9 (8) (1997) 1735–1780.

[2] K. Cho, B. van Merrienboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, Y. Bengio, Learning phrase representations using RNN encoder-decoder for statistical machine translation, in: A. Moschitti,

B. Pang, W. Daelemans (Eds.), Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL, ACL, 2014, pp. 1724–1734. `doi:10.3115/v1/d14-1179`. URL `https://doi.org/10.3115/v1/d14-1179`

[3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, I. Polosukhin, Attention is all you need, in: I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, R. Garnett (Eds.), Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA, 2017, pp. 5998–6008.
URL `https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html`

[4] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, Improving language understanding by generative pre-training, OpenAI.

[5] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, Language models are unsupervised multitask learners, OpenAI.

[6] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, D. Amodei, Language models are few-shot learners, in: H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, H. Lin (Eds.), Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual, 2020.
URL `https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html`

[7] J. Devlin, M. Chang, K. Lee, K. Toutanova, BERT: pre-training of deep bidirectional transformers for language understanding, in: J. Burstein, C. Doran, T. Solorio (Eds.), Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers), Association for Computational Linguistics, 2019, pp. 4171–4186. `doi:10.18653/v1/n19-1423`.
URL `https://doi.org/10.18653/v1/n19-1423`

[8] T. Mikolov, K. Chen, G. Corrado, J. Dean, Efficient estimation of word representations in vector space, in: Y. Bengio, Y. LeCun (Eds.), 1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings, 2013.
URL `http://arxiv.org/abs/1301.3781`

[9] J. Pennington, R. Socher, C. D. Manning, Glove: Global vectors for word representation, in: A. Moschitti, B. Pang, W. Daelemans (Eds.), Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL, ACL, 2014, pp. 1532–1543. `doi:10.3115/v1/d14-1162`.
URL `https://doi.org/10.3115/v1/d14-1162`

[10] P. Bojanowski, E. Grave, A. Joulin, T. Mikolov, Enriching word vectors with subword information, Trans. Assoc. Comput. Linguistics 5 (2017) 135–146.
URL `https://transacl.org/ojs/index.php/tacl/article/view/999`

[11] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, L. Zettlemoyer, Deep contextualized word representations, in: M. A. Walker, H. Ji, A. Stent (Eds.), Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguis-

tics: Human Language Technologies, NAACL-HLT 2018, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 1 (Long Papers), Association for Computational Linguistics, 2018, pp. 2227–2237. `doi:10.18653/v1/n18-1202`.
URL `https://doi.org/10.18653/v1/n18-1202`

[12] A. Akbik, D. Blythe, R. Vollgraf, Contextual string embeddings for sequence labeling, in: E. M. Bender, L. Derczynski, P. Isabelle (Eds.), Proceedings of the 27th International Conference on Computational Linguistics, COLING 2018, Santa Fe, New Mexico, USA, August 20-26, 2018, Association for Computational Linguistics, 2018, pp. 1638–1649.
URL `https://www.aclweb.org/anthology/C18-1139/`

[13] V. Sanh, L. Debut, J. Chaumond, T. Wolf, Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter, CoRR abs/1910.01108. `arXiv:1910.01108`.
URL `http://arxiv.org/abs/1910.01108`

[14] Z. Sun, H. Yu, X. Song, R. Liu, Y. Yang, D. Zhou, Mobilebert: a compact task-agnostic BERT for resource-limited devices, in: D. Jurafsky, J. Chai, N. Schluter, J. R. Tetreault (Eds.), Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020, Association for Computational Linguistics, 2020, pp. 2158–2170. `doi:10.18653/v1/2020.acl-main.195`.
URL `https://doi.org/10.18653/v1/2020.acl-main.195`

[15] J. Alammar, Visualizing machine learning one concept at a time.
URL `https://jalammar.github.io/`

[16] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, CoRR abs/1512.03385. `arXiv:1512.03385`.
URL `http://arxiv.org/abs/1512.03385`

[17] L. J. Ba, J. R. Kiros, G. E. Hinton, Layer normalization, CoRR

abs/1607.06450. `arXiv:1607.06450`.
URL `http://arxiv.org/abs/1607.06450`