Transformer Language Models

Jonathan May

September 12, 2023

In 2017 some researchers at Google considered whether the recurrent part of RNNs/LSTMs was really that important at all in neural MT. In the paper 'Attention is all you need', they described their model, Transformer[1], which outperformed the state of the art at the time at a variety of data points and at lower cost to train.

Within a year or so Transformer models took over most of NLP as they were shown to be useful as language models and as feature sets for classification and structured prediction models. All the images in these notes come from others' papers, lectures, blog posts, etc. Apart from the original transformer paper I recommend the illustrated transformer [2] or the annotated transformer[3].

BTW, Transformer is usually presented in the context of machine translation, as it was used there first, but we're going to describe it in the context of a language model, so it will look a little different from other presentations.

1 Base Model

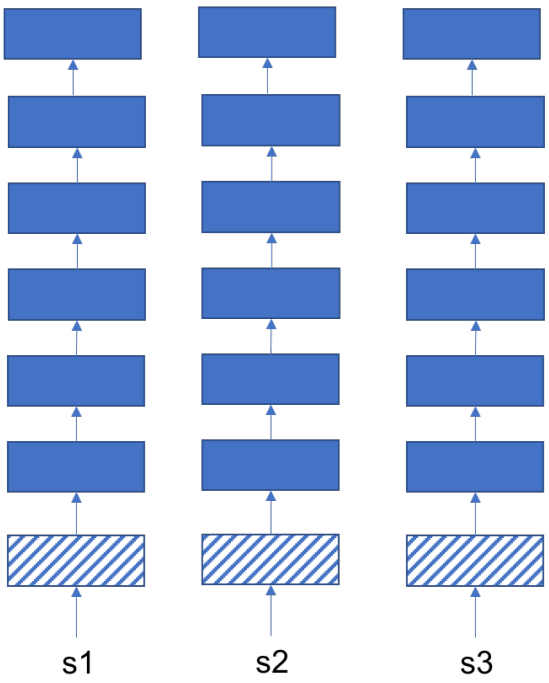
We're going to cover the details in Transformer in various order, sort of from the outside in. To begin with, the overall shape is stacks of representations, conventionally of size 6, with one representation stack per word. To begin with let's imagine each block is just a feed-forward network. Each word is embedded, then at each stage, it's passed through nonlinear transformation via ReLU. In fact there are two linear transformations and one ReLU at each level. So if x is the embedding (or input from last layer), the output is $\max(0, \max(0, xW_1 + b_1)W_2 + b_2)$.[4]

[1] `https://arxiv.org/abs/1706.03762`

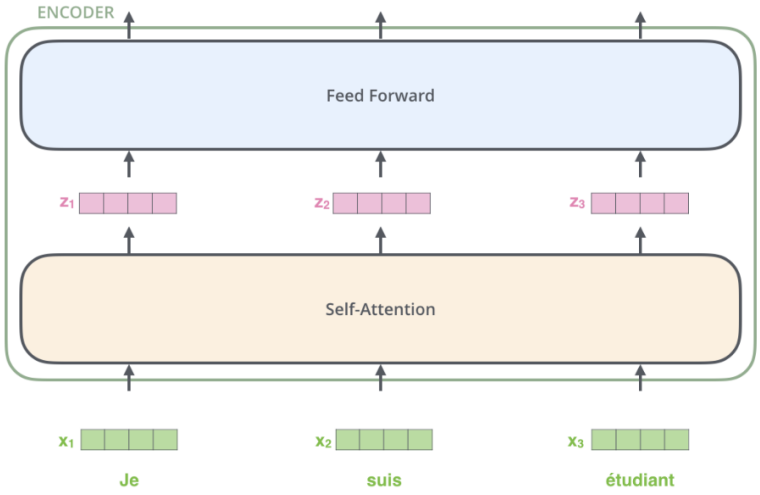
[2] `http://jalammar.github.io/illustrated-transformer/`

[3] `https://nlp.seas.harvard.edu/2018/04/03/attention.html`

[4] W_1 is (512×2048) and W_2 is (2048×512).



But it's not that simple. As you might imagine, attention is heavily involved.[5] In fact each of those cells except the first (which is the embedding) is attention across the entire input followed by the feed-forward component:



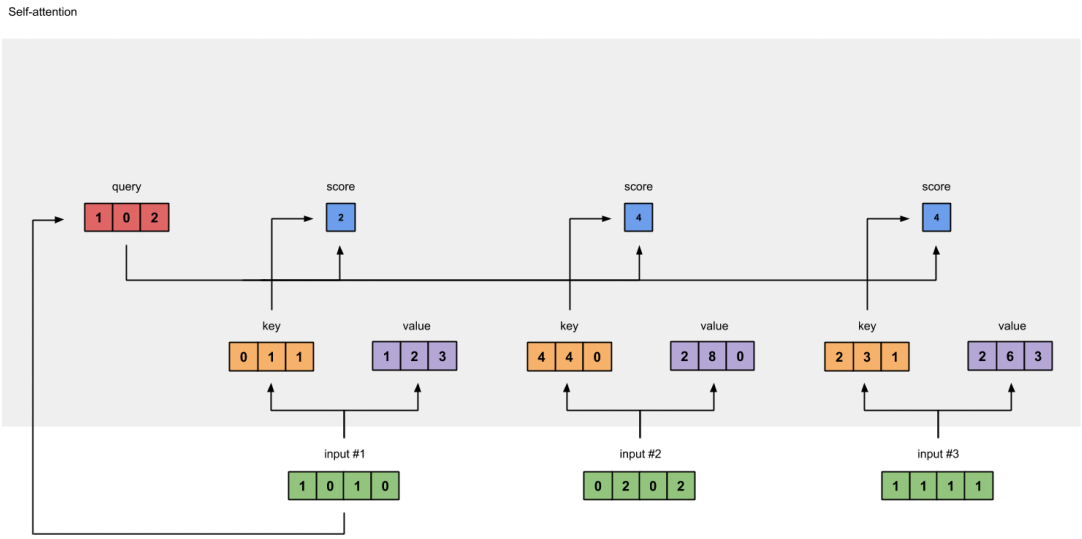
Remember before, we took every previous word's hidden state, multiplied them each by a matrix to turn them into 'comparison' vectors, then compared them before taking softmax to get a distribution? It's the same thing here but a little more complex and, in a way, principled:

[5]It's called *self-attention*, to distinguish from the attention between two different languages (*cross-attention*) we'll get to later.

1.1 Key, Query, Value Attention

As before, if we want to know the attention to some state t, we first transform each state i by a matrix. We call it the 'Key' matrix (K). Different from before, we also transform t by a "query" matrix (Q) Then $tQ(iK)^T$ is the raw score of the influence of i on t; this is done for every i and turned into a distribution α_t by softmax.[6]

Now, instead of using α_t to linearly combine each i, the i (which includes t) are transformed again by a "value" matrix V. These are then linearly combined. That is then fed to the feed-forward unit. Attention, followed by feed forward, is one layer, and there are usually six but sometimes there are more and sometimes there are fewer.

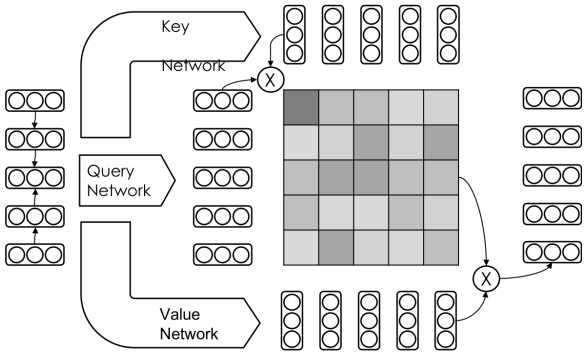


$$K = \begin{bmatrix} 0 & 0.5 & 0.5 \\ 1 & 1 & 0 \\ 0 & 0.5 & 0.5 \\ 1 & 1 & 0 \end{bmatrix} \quad Q = \begin{bmatrix} 0.5 & 0 & 1 \\ 0 & 0 & 0 \\ 0.5 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \quad V = \begin{bmatrix} 0.5 & 1 & 1.5 \\ 0.5 & 2 & 0 \\ 0.5 & 1 & 1.5 \\ 0.5 & 2 & 0 \end{bmatrix}$$

Then $\alpha_1 = \text{softmax}([2, 4, 4]) = [.06, .47, .47]$ and the result would be $[1.9, 6.7, 1.6]$.

I think the figure below by Alireza Zareian nicely expresses the calculation for self-attention; remember, although we did it for one position, in training it can be done for all positions at once:

[6] Not quite. Actually it's $\text{softmax}(\frac{tQ(iK)^T}{\sqrt{|K|}})$, i.e. divide by the dimension of K. This keeps gradients from getting too small, per notes in the paper.



If X is a 5-word input, XQ is the 5 query transforms and XK is the 5 key transforms. Then $XQ(XK^T)$ are the 25 un-normalized weights, each row corresponding to the weights for a position. Taking row-wise softmax gives the 5x5 table of α. Then $V\alpha$ would give the result of self-attention at each position

However, if we're treating this like a next-word-predicting language model[7] you can't *actually* attend to the future because it doesn't exist yet! This isn't a problem when decoding but is in practice, when training, so a mask to block the future would be element-wise multiplied with α:

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

1.2 Multi-Head

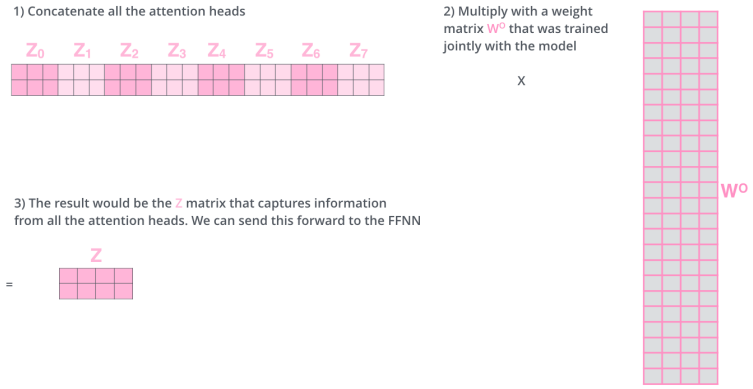
Self attention can be viewed as a generalization of convolving kernels used in convolutional neural networks (CNNs). CNN filters, however, have dimension tied to the relative offset of adjacent inputs (words, pixels, hidden units) while the same Q, K, and V are applied to each input on a layer (different set for source, self-encoder, and self-decoder). Also, CNN filters do fixed combination, not a distributional interpolation. But the information sharing paradigm is very similar.

What are we actually doing when we do self-attention? We are probably combining some semantic and syntactic coordination.

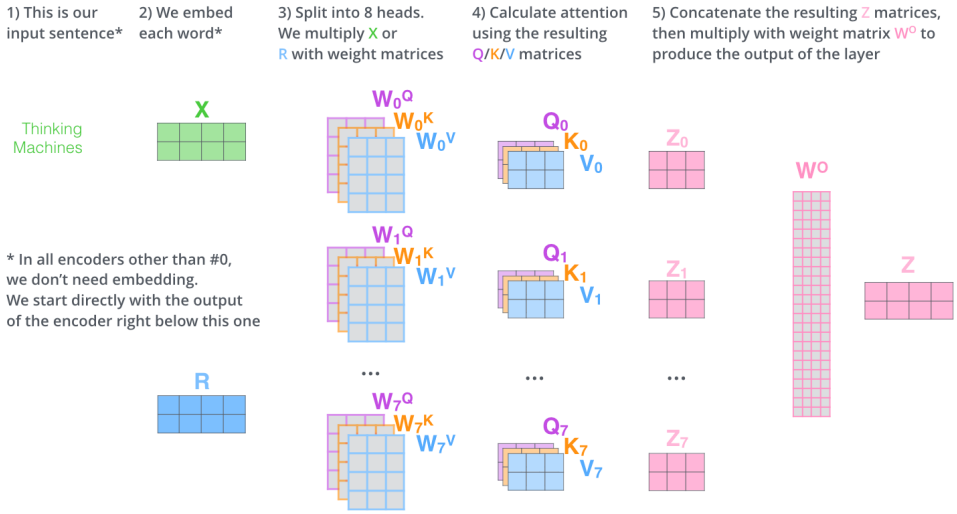
But there are different aspects of information we might want to attend. It seems odd to distill them down into a single (Q, K, V) triple. And since we noticed the similarity to CNNs we can use a technique used in CNNs: multiple filters! Indeed, we actually do attention in one place many[8] times with a different learned (Q, K, V) set for each time; each attention that is learned is called a 'head'. Rather than use e.g. max-pooling or mean-pooling as is often done in CNN, Transformer instead does a linear projection of the heads (Alammar):

[7] an *autoregressive* language model, because the prediction of a word gives you a new state with which to predict the next word

[8] eight



Here is attention all together (Alammar):



1.3 Residual Connections and Layer Norm

In the story we've told so far, data enters a layer, is combined with information from all the other words in the sentence with self attention, then is projected through a feed-forward layer. So if we call the input to a layer x and the self-attention and feed-forward sublayers functions $self$ and ff, the output is $ff(self(x))$. This seems like a good opportunity for the information at that position to get lost; self-attention could decide not to attend to the self! As with RNNs, we can use *residual connections*; in each case we simply add the input back again after each sublayer. This is only done per-sublayer.

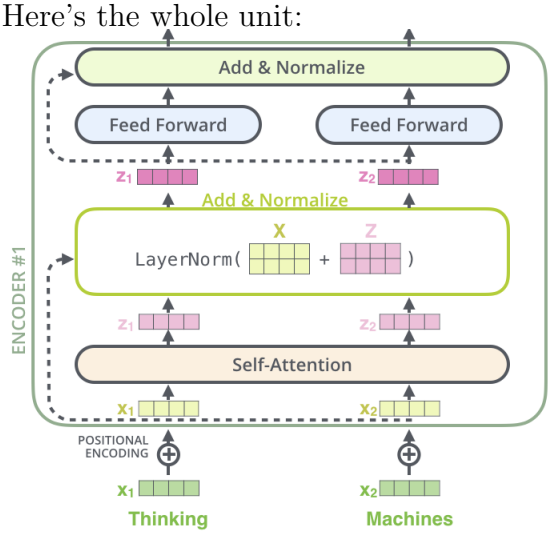
We introduce some sub-results: we calculate $x' = self(x) + x$. Then the output is $ff(x') + x'$.

OK but we don't actually even use the original x or the other intermediates without modification either! Instead we use a technique called 'layer norm' which essentially modifies each item by subtracting the mean and divides by the standard deviation over the vector.[9] So in fact $x' = self(norm(x)) + x$, and the output is $ff(norm(x')) + x'$. This is not well-

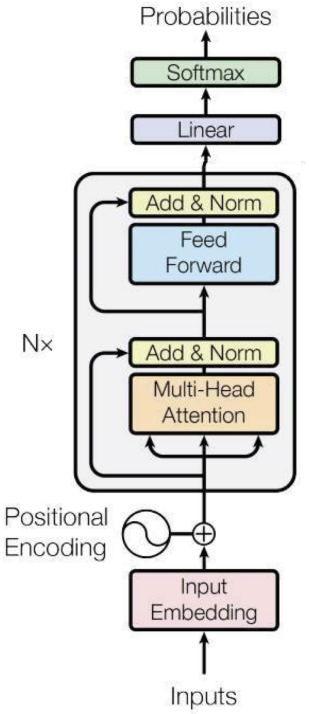
[9] it's a little more complicated than that but this is already rather in the weeds.

described in the original paper but is what has been uncovered post-facto.[10]

Here's the whole unit:



We now know almost everything in this handy diagram hacked together from the original paper:



1.4 Positional Embeddings

We haven't put any explicit notion of the ordering of the words in yet. So a specific sinusoidal function is added element-wise to each embedding. Specifically, let

[10] My former student Thamme Gowda clued me into it; I'm not sure it's written up anywhere.

$$w_k = \frac{1}{10000^{2k/d}}$$

where d is the embedding dimension. Then for the nth word (0-based), the position embedding p_n at position i in the embedding is:

$$p_n^{(i)} = \begin{cases} n \times sin(w_k) & \text{if } i = 2\text{k} \\ n \times cos(w_k) & \text{if } i = 2\text{k+2} \end{cases}$$

This allows the position embedding for any $j+k$ to be represented as a linear function of the position embedding for j, so the other elements in the Transformer can take advantage of this, if they need to, and then there is theoretically no limit to the number of tokens that can be read. Why would this work? Consider representing integers in binary:[11]

And now compare the periodicity of those columns with the graphical representation of the vector:

0 :	0	0	0	0	8 :	1	0	0	0
1 :	0	0	0	1	9 :	1	0	0	1
2 :	0	0	1	0	10 :	1	0	1	0
3 :	0	0	1	1	11 :	1	0	1	1
4 :	0	1	0	0	12 :	1	1	0	0
5 :	0	1	0	1	13 :	1	1	0	1
6 :	0	1	1	0	14 :	1	1	1	0
7 :	0	1	1	1	15 :	1	1	1	1

Here is what the position embedding look like:

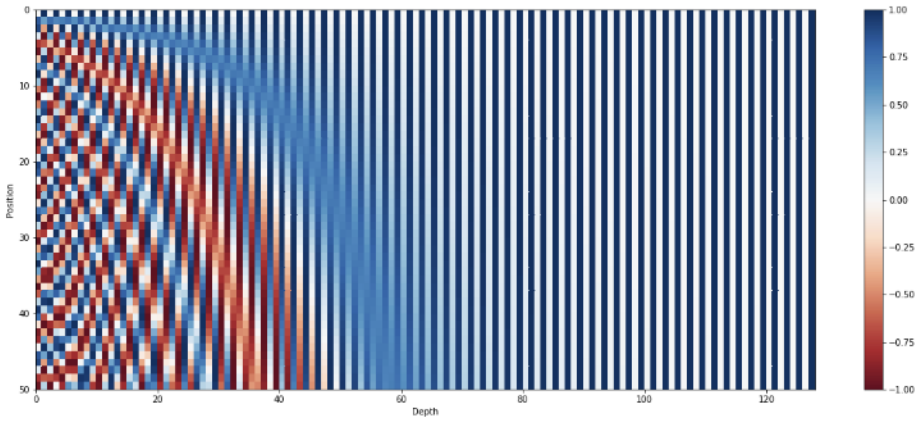


Figure 2 - The 128-dimensional positonal encoding for a sentence with the maximum lenght of 50. Each row represents the embedding vector $\overrightarrow{p_t}$

In practice there are other ways to do this, and having an upper bound on length is not that big a deal, so people have also used a fixed table of learned position embeddings and other similar approaches.

[11] Source for this section: `https://kazemnejad.com/blog/transformer_architecture_positional_encoding/`

1.5 Shared embeddings and BPE

One last part of the model: the word embeddings! Or should I say, the word *piece* embeddings. We covered this way in the beginning but it emerged in wide use contemporaneous with and by Transformer. Specifically, byte-pair encodings (BPE):[12]. This is a kind of unsupervised word segmentation algorithm that works as follows:

```
def bpe(merges, vocab):
  for i in range(merges):
    # count all adjacent bytes, e.g.
    #  "four" = "f o u r" = "f/o, o/u, u/r"
    # multiply by count of word in corpus
    pairs = get_stats(vocab)
    # find the most frequent pair. let's say it's "o/u"
    best = argmax(pairs)
    # now consider "ou" to be merged. So next time
    # "four" = "f ou r" and instead you count "f/ou, ou/r"
    vocab = merge_Vocab(vocab, best)
  return vocab
```

You run it as long as you want. Instead of `merges` you can consider the size of the vocabulary you want. For latin languages the minimum is around 60 (e.g. most of ASCII) but this means the vocabulary blow up of before is no longer relevant. Furthermore, you can now more easily understand why it's useful to combine the embedding tables.

What about different character sets? Well, most people don't seem to worry about that because of hegemony. But we (at ISI) do; we have romanization tools that convert all langauge data into a common space (latin letters...still pretty hegemonic).

1.6 Model optimization tips

- Batches had 25k tokens (in practice similar length sentences are grouped together for maximum parallelism with minimal lost work).
- Adam optimizer (SGD with fancy learning rate) plus a fancy learning rate on top of that
- Dropout! Everywhere in the model, with probability 0.1, treat a parameter as if it was 0. don't contribute to the loss, don't update on backprop.

[12] https://arxiv.org/abs/1508.07909