Transformers

Or as I like to call it Attention on Steroids. 💉💊

Ria Kulshrestha · Follow

Published in Towards Data Science · 10 min read · Jun 29, 2020

532 2

Optimus Prime here. It is also not about the electrical device that is used to transfer energy from one electrical circuit to another. What is this about then, you ask?

It is about the one in the most sci-fi fields of all time, Artificial Intelligence — Natural Language Processing in particular and it is pretty optimal at transferring information and primely used. (See what I did there. :P)

This post is based on the paper: Attention is All You Need. P.S. the authors were not kidding when they chose that title because you will need all the attention at your disposal for this one. But don't let that scare you, it is SO SO worth it!!

What is a Transformer?

The Transformer in NLP is a **novel architecture** that aims to solve sequence-to-sequence tasks while handling long-range dependencies with ease. It relies entirely on self-attention to compute representations of its input and output **WITHOUT** using sequence-aligned RNNs or convolution. 🤯

If you recall my previous post, Understanding Attention In Deep Learning, we discussed how and why many models fell short when it came to handling long-range dependencies. The concept of attention somewhat allowed us to overcome that problem and now in Transformers we will build on top of attention and unleash its full potential.

Understanding Attention In Deep Learning

How a little attention changed the AI game!

towardsdatascience.com

Few things to know before diving into Transformers

Self-Attention

Let us start with revisiting what attention is in the NLP universe? Understanding Attention In Deep Learning. (*I apologize for these blatant self-advertisements, but seriously give it a read. It will help you under Transformers much better. I promise.*)

Attention allowed us to focus on parts of our input sequence while we predicted our output sequence. If our model predicted the word "*rouge*" [French translation for the color red], we are very likely to find a high weight-age for the word "*red*" in our input sequence. So attention, in a way, allowed us to map some connection/correlation between the input word "*rouge*" and the output word "*red*".

> ***Self attention****, sometimes called intra-attention is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence.*

In simpler terms, ***self attention helps us create similar connections but within the same sentence.*** Look at the following example:

```
“I poured water from the bottle into the cup until it was full.”
it => cup
```

```
“I poured water from the bottle into the cup until it was empty.”
it=> bottle
```

By changing one word “*full*” — > “*empty*” the reference object for “*it*” changed. If we are translating such a sentence, we will want to know the word “*it*” refers to.

The three kinds of Attention possible in a model:

1. ***Encoder-Decoder Attention***: Attention between the input sequence and the output sequence.

2. ***Self attention in the input sequence***: Attends to all the words in the input sequence.

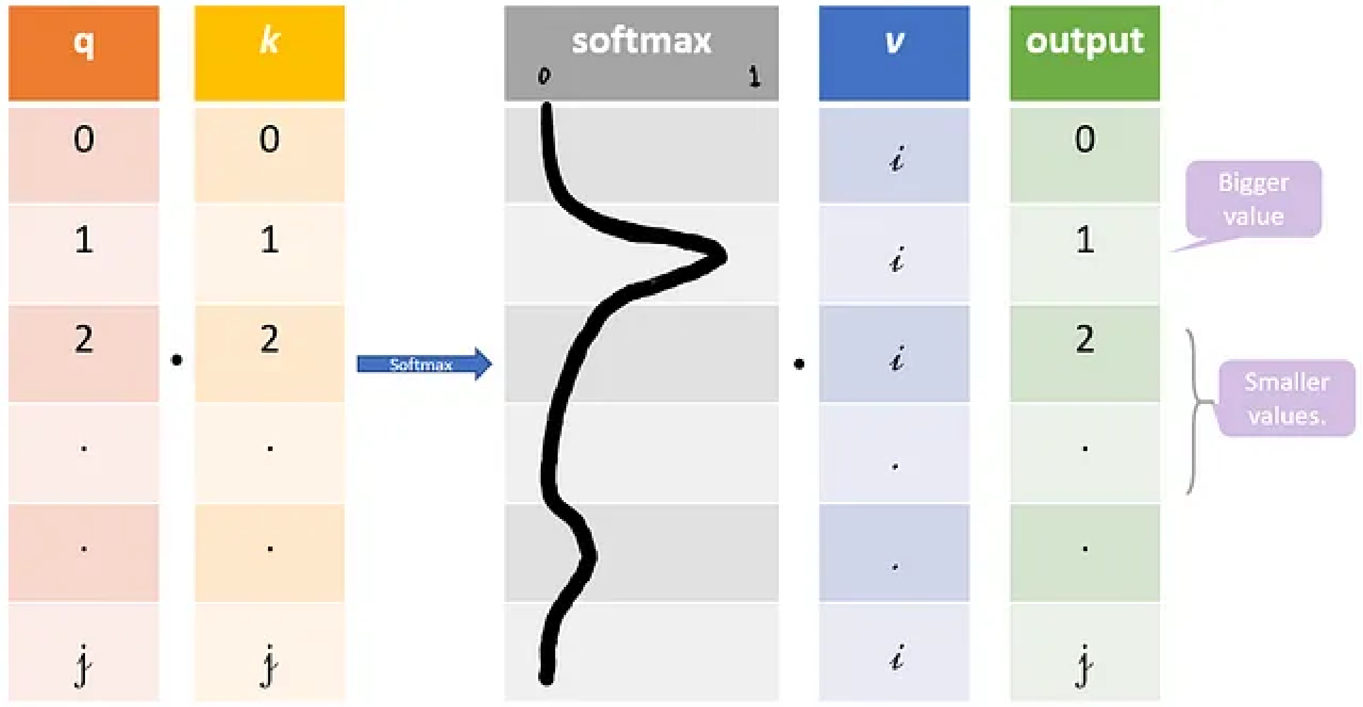
3. ***Self attention in the output sequence:*** One thing we should be wary of here is that the scope of self attention is limited to the words that occur before a given word. This prevents any information leaks during the training of the model. This is done by masking the words that occur after it for each step. So for step 1, only the first word of the output sequence is NOT masked, for step 2, the first two words are NOT masked and so on.

Keys, Values, and Queries:

The three random words I just threw at you in this heading are vectors created as abstractions are useful for calculating self attention, more details on each below. These are calculated by multiplying your input vector(*X*) with weight matrices that are learnt while training.

- ***Query Vector***: $q = X * Wq$. Think of this as the current word.
- ***Key Vector:*** $k = X * Wk$. Think of this as an indexing mechanism for Value vector. Similar to how we have key-value pairs in hash maps, where keys are used to uniquely index the values.
- ***Value Vector:*** $v = X * Wv$. Think of this as the information in the input word.

What we want to do is take query q and find the most similar key k, by doing a dot product for q and k. The closest query-key product will have the highest value, followed by a softmax that will drive the $q.k$ with smaller values close to 0 and $q.k$ with larger values towards 1. This softmax distribution is multiplied with v. The value vectors multiplied with ~1 will get more attention while the ones ~0 will get less. The sizes of these q, k and v vectors are referred to as "***hidden size***" by various implementations.



The values represent the index for q, k and i.

All these matrices *Wq, Wk* and *Wv* are learnt while being jointly trained during the model training.

Calculating Self attention from q, k and v:

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

Formula for self-attention. Source: paper.

If we are calculating self attention for *#i* input word,

- ***Step 1:*** Multiply q_i by the k_j key vector of word.
- ***Step 2:*** Then divide this product by the square root of the dimension of key vector.
  This step is done **for better gradient flow** which is specially important in cases when the value of the dot product in previous step is too big. As using them directly might push the softmax into regions with very little gradient flow.
- ***Step 3:*** Once we have scores for all *j*s, we pass these through a softmax. We get normalized value for each *j*.
- ***Step 4:*** Multiply softmax scores for each *j* with v_i vector.
  The idea/purpose here is, very similar attention, to keep preserve only the values *v* of the input word(s) we want to focus on by multiplying them with high probability scores from softmax ~1, and remove the rest by driving them towards 0, i.e. making them very small by multiplying them with the low probability scores ~0 from softmax.

$$Step\ 1\ =\ q_i\ \cdot\ k_j \qquad for\ all\ 0\ \leqq\ j\ \leqq n$$

$$Step\ 2\ =\ \frac{q_i\ \cdot\ k_j}{\sqrt{dim(k_j)}} \qquad for\ all\ 0\ \leqq\ j\ \leqq n$$

$$Step\ 3\ =\ softmax\left(\frac{q_i\ \cdot\ k_j}{\sqrt{dim(k_j)}}\right) \qquad for\ all\ 0\ \leqq\ j\ \leqq n$$

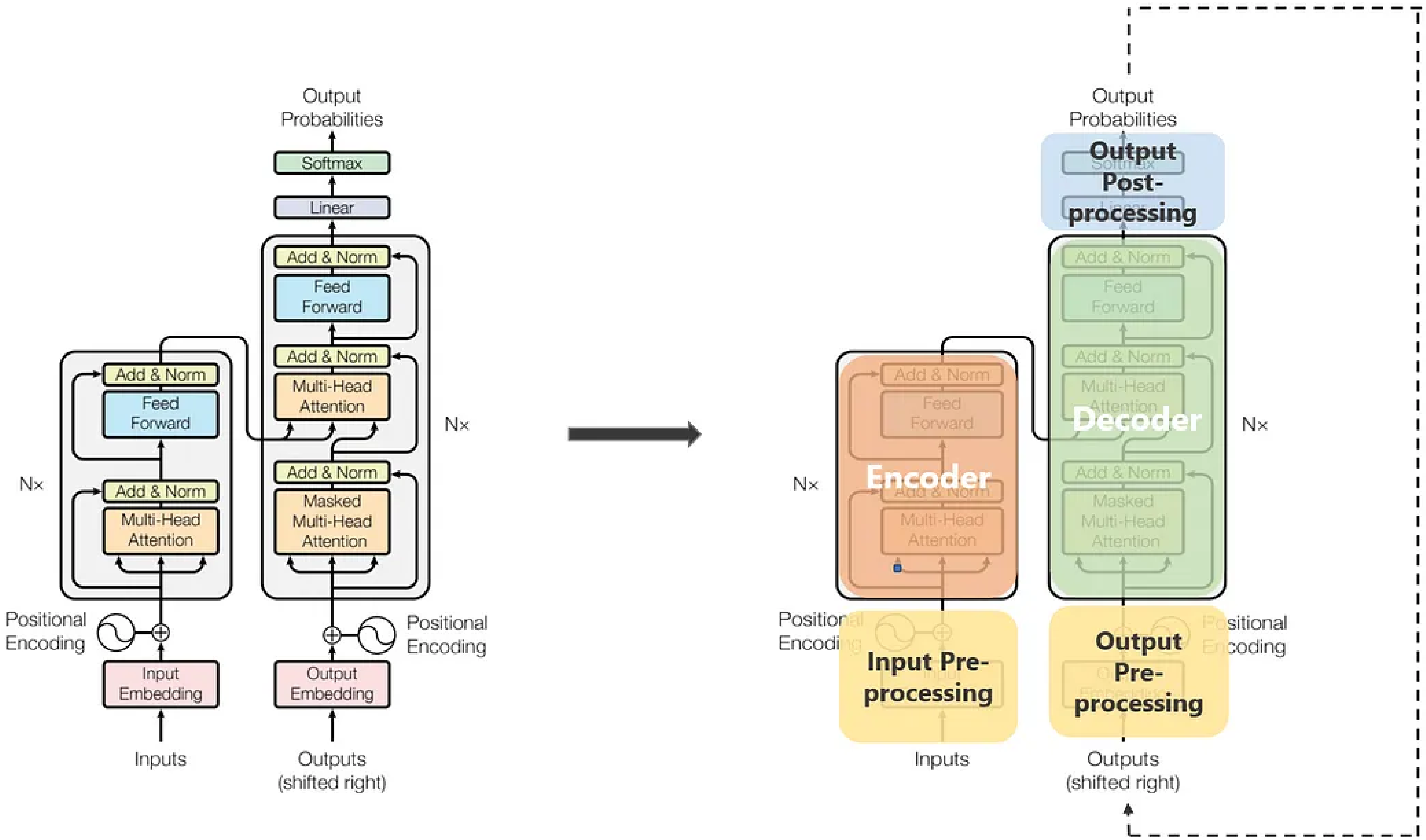
$$Step\ 4\ =\ \left\{softmax\left(\frac{q_i\ \cdot\ k_j}{\sqrt{dim(k_j)}}\right)\ \cdot\ v_i\ \right\} \qquad for\ all\ 0\ \leqq\ j\ \leqq n$$

$$Finally\ :\ z_i\ =\sum_{j=0}^{n}\left(softmax\left(\frac{q_i\ \cdot\ k_j}{\sqrt{dim(k_j)}}\right)\ \cdot\ v_i\right)$$

Calculating output of self attention for the ith input word. If you are looking for an analogy between self attention and attention, think of z serving the purpose of context vectors and not global alignment weights.

The Transformer

⚠️ ***A word of caution***: the contents of this image may appear exponentially more complicated than they are. We will break this scary beast down into small baby beasts and it will all make sense. (I promise #2)



(left) The Transformer architecture. Source: paper. (right) An abstracted version of the same for better understanding.

Beast #1: Encoder-Decoder stacks

Encoder: The encoder maps an input sequence of symbol representations $(x_1, \ldots, x_n)$ to a sequence of representations $z = (z_1, \ldots, z_n)$. Think of them as the outputs from self attention with some post-processing.

Each encoder has two sub-layers.

1. A **multi-head self attention mechanism** on the input vectors (Think parallelized and efficient sibling of self attention).
2. A simple, position-wise **fully connected feed-forward network** (Think post-processing).

Check out this absolute bomb 3D diagram of the Encoder block used in BERT. **Seriously you can't miss this!!!** It is like a whole new level of understanding.

Decoder: Given z, the decoder then generates an output sequence $(y_1, \ldots, y_m)$ of symbols one element at a time.

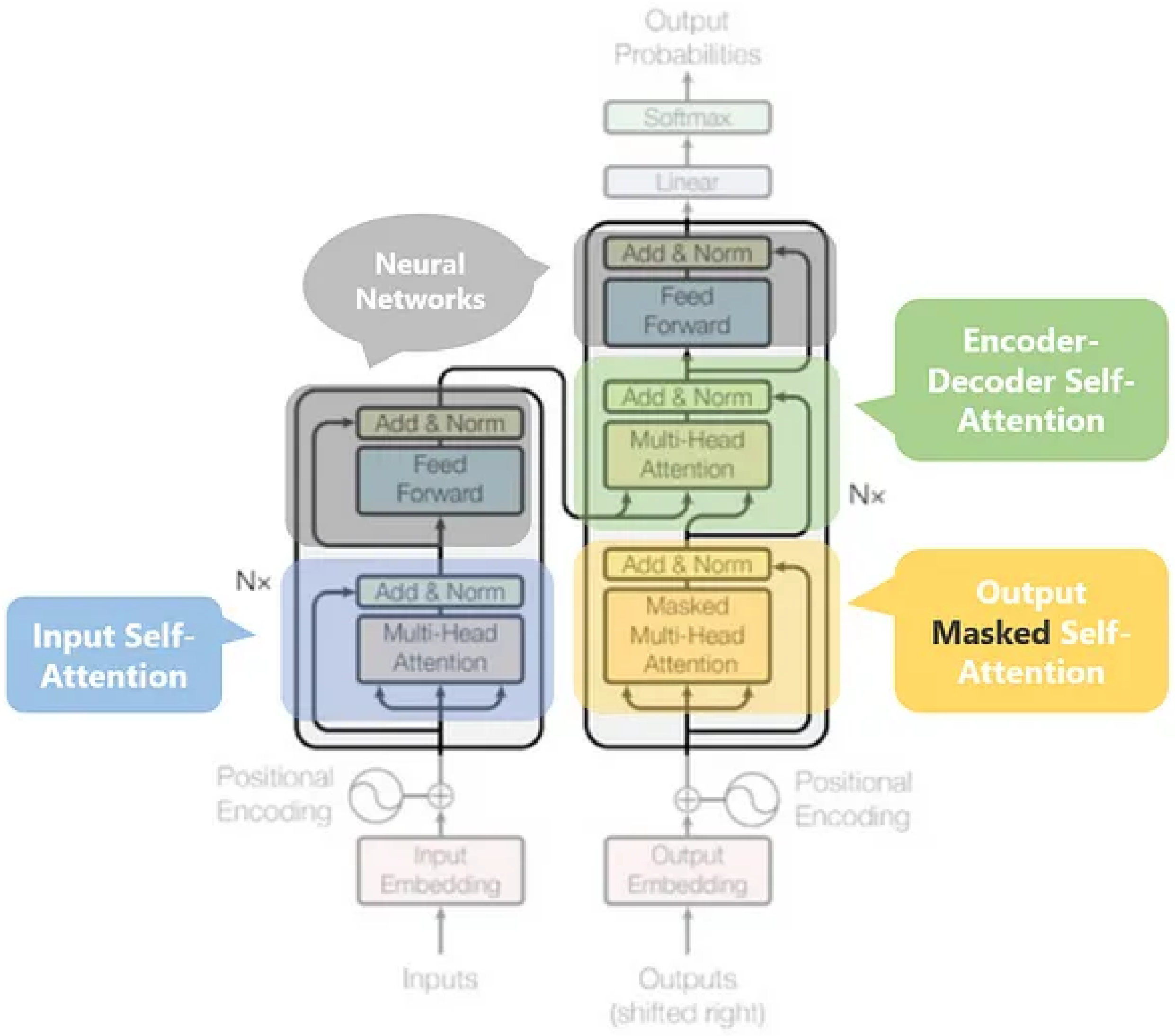
Each decoder has three sub-layers.

1. A *masked* **multi-head self attention mechanism** on the output vectors of the previous iteration.
2. A **multi-head attention mechanism** on the output from encoder and masked multi-headed attention in decoder.
3. A simple, position-wise **fully connected feed-forward network** (think post-processing).

A few additional points:

- In the original paper, 6 layers were present in the encoder stack (2 sub-layer version) and 6 in the decoder stack (3 sub-layer version).
- All sub-layers in the model, as well as the embedding layers, produce outputs of the same dimension. This is done to facilitate the residual connections.

Beast #2 Inside Encoder-Decoder stacks — Multi-Head Attention:



The three kinds of attention in encoder and decoder stacks along with feed forward neural networks.

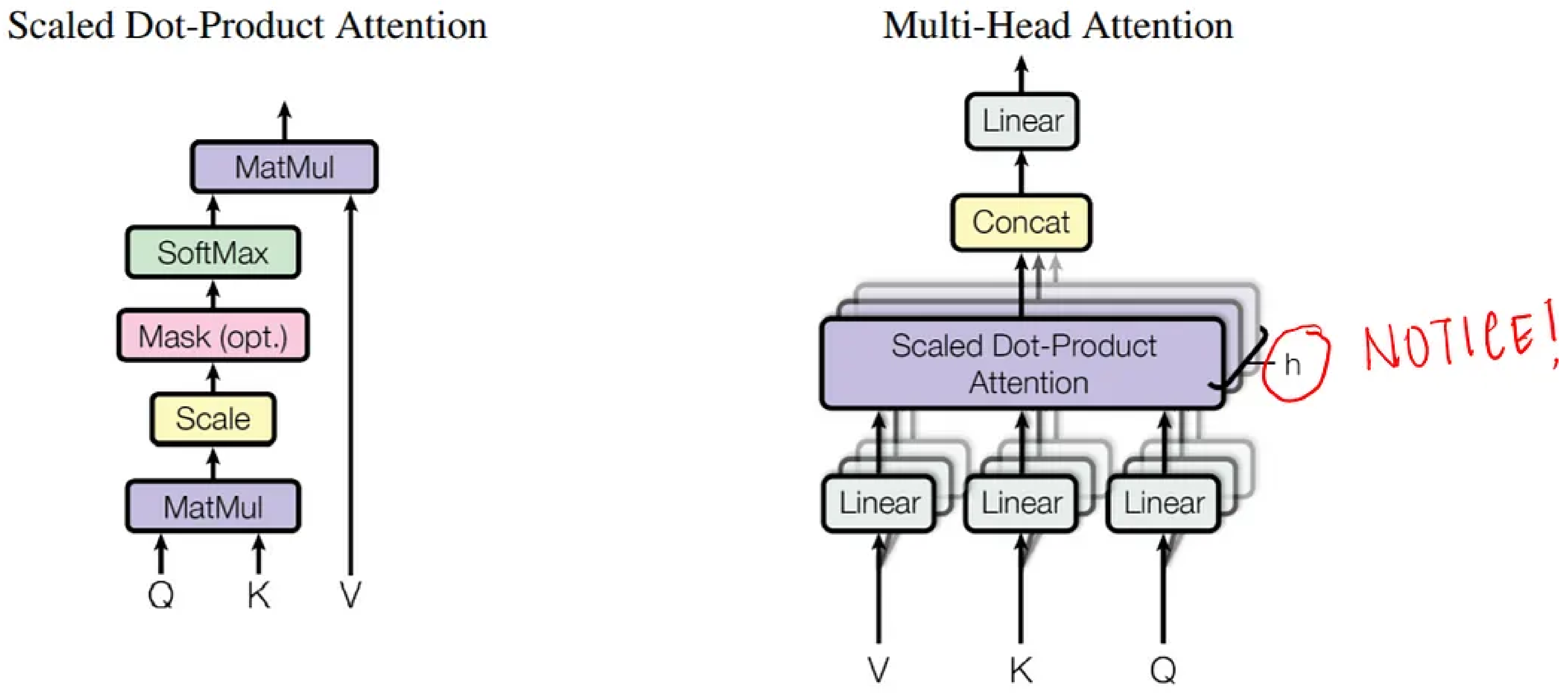
We just noted that the output of each sub-layer needs to be of the same dimension which is 512 in our paper.

=> z_i needs to be of 512 dimensions.
=> v_i needs to be of 512 dimensions as z_i are just sort of weighted sums of v_is.

Additionally, we want to allow the model to focus on different positions is by **calculating self attention multiple times with different sets** of *q, k* and *v* vectors, then take an average of all those outputs to get our final *z*.

So instead of dealing with these humongous vectors and averaging multiple outputs, we reduce the size of our *k,q* and *v* vectors to some smaller dimension — reduces size of *Wq, Wk*, and *Wv* matrices as well. We keep the multiple sets (*h*) of *k,q* and *v and* refer to each set as an "*attention head",* hence the name *multi-headed* attention. And lastly, instead of averaging to get final *z,* we concatenate them.

The size of the concatenated vector will be too large to be fed to the next sub-layer, so we scale it down by multiplying it with another learnt matrix *Wo*.



(left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel. Source: paper.

Multiple attention heads allowed the model to jointly attend to information from different representation sub-spaces at different positions which was inhibited by averaging in a single attention head.

Beast #3— Input and Output Pre-processing:

The input words are represented using some form of embedding. This is done for both encoder and decoder.

Word embedding on their own lack any positional information which is achieved in RNNs by virtue of their sequential nature. Meanwhile in self-attention, due to softmax, any such positional information is lost.

To preserve the positional information, the transformer injects a vector to individual input embeddings (could be using word embeddings for corresponding to the input words). These vectors follow a specific periodic function (Example: combination of various sines/cosines having different frequency, in short not in sync with each other) that the model learns and is able to **determine the position of individual word wrt each other** based on the values **.**

This injected vector is called "*positional encoding*" and are added to the input embeddings at the bottoms of both encoder and decoder stacks.

Beast #4 — Decoder stack: Revisited

The output of the decoder stack at each step is fed back to the decoder in the next time step — pretty similar to how outputs from previous steps in RNNs were used as next hidden states. And just as we did with the encoder inputs, we embed and add positional encoding to those decoder inputs to preserve the position of each word. This positional encoding + word embedding combo is then fed into a masked multi-headed self attention.

This self-attention sub-layer in the decoder stack is modified to prevent positions from attending to subsequent positions — you can't look at future words. This masking ensures that the predictions for position *i* can depend only on the known outputs at positions less than *i*.

The outputs from the encoder stack are then used as multiple sets of key vectors ***k*** and value vectors ***v***, for the "encoder decoder attention" — shown in green in the diagram — layer. It helps the decoder focus on the contextually relevant parts in the input sequence for that step. (The part similar to global attention vectors.) The ***q*** vector comes from the "output self attention" layer.

Once we get the output from the decoder, we do a softmax again to select the final probabilities of words.

Conclusion

Let’s finish with a quick wrap-up revision.

- We started with understanding what self attention is and how to calculate self-attention from these *v, k ,q* vectors.
- Multi-headed attention is an efficient modification of self attention that uses multiple smaller sets of *v, k ,q* and concatenates the outputs from each set to get the final *z*.
- Then we saw how and where the three kinds of self attention are used in the model.
- Followed by the pre-processing done on the inputs for the encoder and decoder stacks.

References + Recommended Reads