Transformer's Self-Attention: Why Is Attention All You Need?

2021-11-15 08:50 Language Models, Machine Translation, Transformer

In 2017, Vaswani et al. published a paper titled "Attention Is All You Need" for the NeurIPS conference. The transformer architecture does not use any recurrence or convolution. It solely relies on attention mechanisms.

In this article, we discuss the attention mechanisms in the transformer:

- Dot-Product And Word Embedding
- Scaled Dot-Product Attention
- Multi-Head Attention

- Self-Attention

1. Dot-Product And Word Embedding

The dot-product takes two equal-length vectors and returns a single number.

$$\boldsymbol{a} = [a_1, a_2, \ldots, a_n]$$
$$\boldsymbol{b} = [b_1, b_2, \ldots, b_n]$$

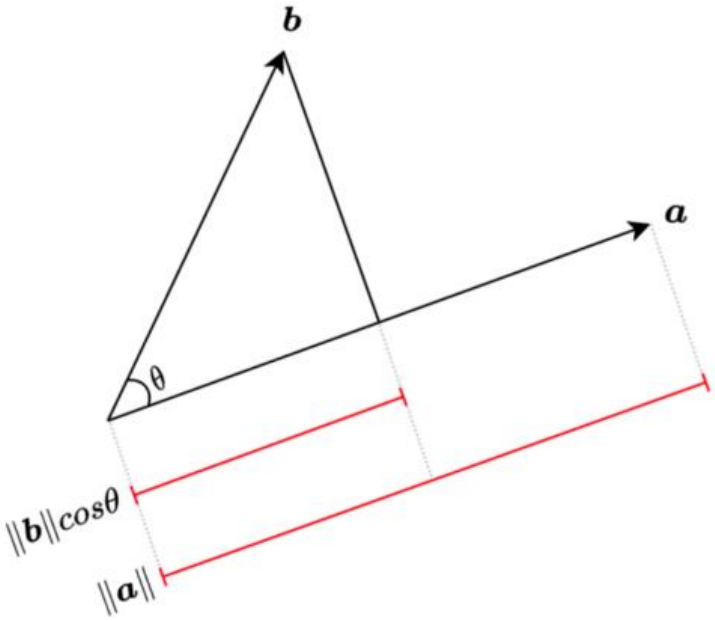
$$\boldsymbol{a} \cdot \boldsymbol{b} = [a_1 b_1 + a_2 b_2 + \ldots + a_n b_n]$$
$$= \sum_{i=1}^{n} a_i b_i$$

We use the dot operator to express the dot-product operation. We also call it the inner product as we calculate element-wise multiplication (that is, “inner product”) and sum those products together.

The geometric definition of the dot-product is as follows:

$$\boldsymbol{a} \cdot \boldsymbol{b} = \|\boldsymbol{a}\| \|\boldsymbol{b}\| cos\theta$$

$\|\boldsymbol{a}\|$ is the magnitude of vector $\boldsymbol{a}$, and $\|\boldsymbol{b}\|$ is the magnitude of vector $\boldsymbol{b}$. θ is the angle between vector $\boldsymbol{a}$ and vector $\boldsymbol{b}$.



We can also project vector ***a*** onto vector ***b*** and visualize the product of ‖***a***‖*cosθ* and ‖***b***‖. The calculated value is the same either way.

The dot-product ‖***a***‖‖***b***‖*cosθ* is at maximum when *θ* is 0 (*cos0* = 1), and at minimum when is θ is 2π (*cos*2π = -1). In other words, the dot-product is bigger when two vectors have similar directions than otherwise.

Before discussing the dot-product attention, we should talk about the word embeddings as it gives us some intuition on how the dot-product attention works.

Word embeddings use distributed representations of words (tokens), which is more efficient than one-hot vector representation. The well-known pre-trained word embeddings are the word2vec and GloVe (Global Vectors for Word Representation). Those word embedding vectors allow us to decompose a word into analogies.

The following diagram shows how we can decompose king and queen using word embedding vectors.

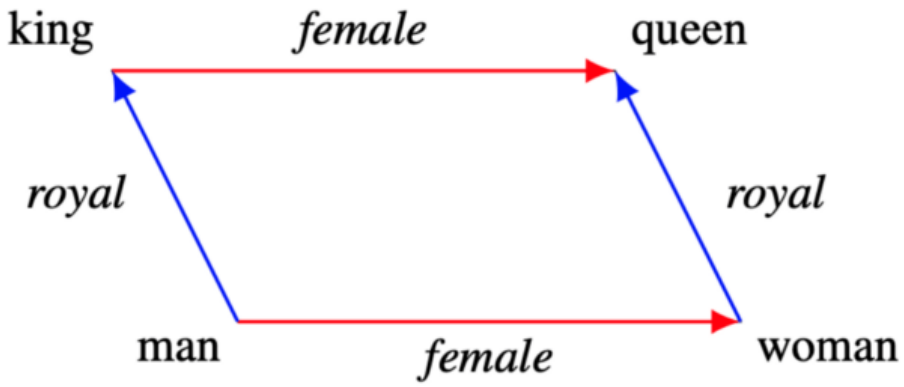


Figure 1 of Towards Understanding Linear Word Analogies

In other words, a word vector may contain multiple semantics. Hypothetically, we could query if a word vector has “royal” by projecting the vector with some matrix operation.

The question is can a language model automatically learn to query different meanings and functions of each word in a sentence?

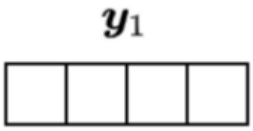
The transformer learns word embeddings from scratch, learns matrix weights for word vector projections, and calculates the dot-products for attention mechanisms. In the next section, we discuss how those vector mathematics work together.

2. Scaled Dot-Product Attention

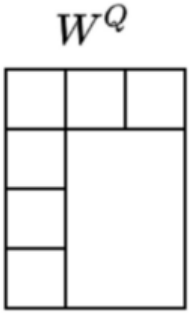
Let’s say we have a model that translates an English sentence *X* into a French sentence *Y*. In the output sentence, the first-word vector would be a BOS (beginning-of-sentence) marker which is just another word vector (albeit fixed), and we call it vector y1. The model needs to extract a context

from the input sentence X for the first-word vector ***y***1 in the output sentence Y.

We express a word vector as a row vector with four dimensions for simplicity.



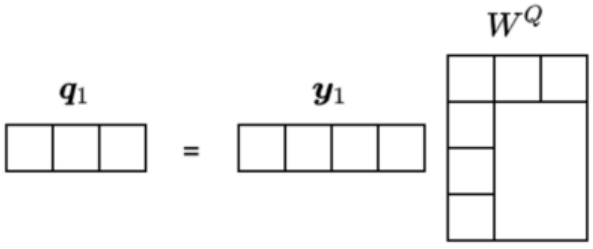
We use a 4×3 matrix *W*Q to project word vector ***y***1:



It’s a simple matrix multiplication on the word vector ***y***1.

$$\boldsymbol{q}_1 = \boldsymbol{y}_1 W^Q$$

As you can see, the dimension of ***q***1 does not have to be the same as the dimension of **y**1.

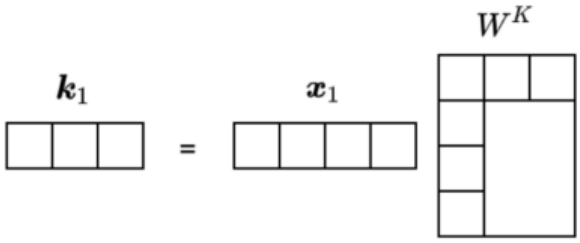


In this example, the dimension of the resulted vector became three.

We also project word vectors in *X* using another matrix *W*K. For example, we project word vector ***x***1 in X as follows:

$$\boldsymbol{k}_1 = \boldsymbol{x}_1 W^K$$

Again, the dimension of ***k***1 does not have to be the same as the dimension of ***x***1.



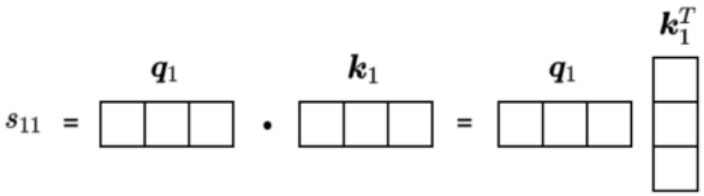
However, the dimension of ***k***1 must be the same as the dimension of ***q***1. Only then can we apply the dot-product to them.

$$\dim(\boldsymbol{q}_1) = \dim(\boldsymbol{k}_1)$$

We apply the dot-product to see how strong these projected features (we call them query ***q***1 and key ***k***1) relate to each other:

$$s_{11} = \boldsymbol{q}_1 \cdot \boldsymbol{k}_1$$

We can calculate the dot-product by transposing vector ***k***1 and performing matrix multiplication:



The result is a scalar value. Let's call it "score". As noted above, we also call vector $\boldsymbol{q}1$ "query" and vector $\boldsymbol{k}1$ "key". In other words, we extract a query from vector $\boldsymbol{y}1$, extract a key from vector $\boldsymbol{x}1$, and check how well the query matches with the key by the score $s11$. So, we are finding out how two-word vectors relate in terms of extracted features (the query and the key).

Since we are using a linear transformation, we can calculate scores between multiple word vectors by a simple matrix operation.

Sentence X consists of *n* word vectors (in reality, an input sentence has a pre-defined number of tokens and unused word positions are filled with a filler which are effectively ignored):

$$X = \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \vdots \\ \boldsymbol{x}_n \end{bmatrix}$$

We can apply the following matrix operation to extract keys from sentence *X*:

$$XW^K = \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \vdots \\ \boldsymbol{x}_n \end{bmatrix} W^K = \begin{bmatrix} \boldsymbol{x}_1 W^K \\ \boldsymbol{x}_2 W^K \\ \vdots \\ \boldsymbol{x}_n W^K \end{bmatrix} = \begin{bmatrix} \boldsymbol{k}_1 \\ \boldsymbol{k}_2 \\ \vdots \\ \boldsymbol{k}_n \end{bmatrix} = K$$

We can apply the dot-product between the query $\boldsymbol{q}1$ and all of the keys in *K*:

$$
\begin{aligned}
\boldsymbol{q}_1 K^\top &= \boldsymbol{q}_1 \begin{bmatrix} \boldsymbol{k}_1 \\ \boldsymbol{k}_2 \\ \vdots \\ \boldsymbol{k}_n \end{bmatrix}^\top \\
&= \boldsymbol{q}_1 \begin{bmatrix} \boldsymbol{k}_1^\top & \boldsymbol{k}_2^\top & \dots & \boldsymbol{k}_n^\top \end{bmatrix} \\
&= \begin{bmatrix} \boldsymbol{q}_1 \boldsymbol{k}_1^\top & \boldsymbol{q}_1 \boldsymbol{k}_2^\top & \dots & \boldsymbol{q}_1 \boldsymbol{k}_n^\top \end{bmatrix} \\
&= \begin{bmatrix} s_{11} & s_{12} & \dots & s_{1n} \end{bmatrix}
\end{aligned}
$$

We now have a list of scores that tells us how the query $\boldsymbol{q}1$ relates to each key in sentence *X*. We can apply the softmax function to convert the scores into weights:

$$
\begin{aligned}
\text{softmax}(\boldsymbol{q}_1 K^\top) &= \text{softmax}([s_{11} \; s_{12} \; \dots \; s_{1n}]) \\
&= [w_{11} \; w_{12} \; \dots \; w_{1n}]
\end{aligned}
$$

The weights indicate how much the model should pay attention to each word in sentence X regarding the query $\boldsymbol{q}1$. So, let's call the weights "attention weights".

We use the attention weights to extract a context from sentence X*, but we need to handle one more step.* Since we are dealing with a specific query, we should also extract particular values from sentence *X.* We use another matrix *WV* to project sentence *X* to extract values.

$$XW^V = \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \vdots \\ \boldsymbol{x}_n \end{bmatrix} W^V = \begin{bmatrix} \boldsymbol{x}_1 W^V \\ \boldsymbol{x}_2 W^V \\ \vdots \\ \boldsymbol{x}_n W^V \end{bmatrix} = \begin{bmatrix} \boldsymbol{v}_1 \\ \boldsymbol{v}_2 \\ \vdots \\ \boldsymbol{v}_n \end{bmatrix} = V$$

The dimension of vectors in *V* may be different from the dimension of vectors in *X*.

We calculate the weighted sum of value vectors using the attention weights:

$$\begin{aligned} \text{softmax}(\boldsymbol{q}_1^T K)V &= [w_{11}\ w_{12}\ \dots\ w_{1n}]\, V \\ &= [w_{11}\ w_{12}\ \dots\ w_{1n}] \begin{bmatrix} \boldsymbol{v}_1 \\ \boldsymbol{v}_2 \\ \vdots \\ \boldsymbol{v}_n \end{bmatrix} \\ &= \sum_{i=1}^{n} w_{1i}\, \boldsymbol{v}_i \end{aligned}$$

The weighted sum of value vectors is the vector representing a context from sentence *X* regarding the query ***q***1 from word vector ***y***1.

We can extend the logic to multiple words (***y***1, ***y***2, .., ***y***m) in sentence *Y* (in reality, an output sentence has a pre-defined number of tokens and unpredicted word positions are masked and won't affect the attention mechanism):

$$Y = \begin{bmatrix} \boldsymbol{y}_1 \\ \boldsymbol{y}_2 \\ \vdots \\ \boldsymbol{y}_m \end{bmatrix}$$

We represent all queries in a matrix:

$$Q = \begin{bmatrix} \boldsymbol{q}_1 \\ \boldsymbol{q}_2 \\ \vdots \\ \boldsymbol{q}_n \end{bmatrix} = \begin{bmatrix} \boldsymbol{y}_1 W^Q \\ \boldsymbol{y}_2 W^Q \\ \vdots \\ \boldsymbol{y}_n W^Q \end{bmatrix} = \begin{bmatrix} \boldsymbol{y}_1 \\ \boldsymbol{y}_2 \\ \vdots \\ \boldsymbol{y}_n \end{bmatrix} W^Q = YW^Q$$

So, we can calculate the attention vectors for all tokens in sentence Y:

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

$$\text{where} \quad Q = YW^Q,\ K = XW^K,\ V = XW^V$$

The scaling `1/square-root(dk)` deserves some explanation. “dk” is the dimension of query and key vectors. When dk is large, the dot-product tends to be large in magnitude.

The paper explains the reason, assuming a scenario where the elements of the query and key vectors are independent random variables with mean 0 and variance 1. The dot-product of vectors ***q*** and ***k*** has mean 0 and variance dk due to the distribution of the product of two random variables:

$$\boldsymbol{q} \cdot \boldsymbol{k} = \sum_{i=1}^{d_k} q_i k_i$$

$$E[\boldsymbol{q} \cdot \boldsymbol{k}] = E\left[\sum_{i=1}^{d_k} q_i k_i\right] = \sum_{i=1}^{d_k} E[q_i k_i] = \sum_{i=1}^{d_k} E[q_i]\,E[k_i] = 0$$

$$Var(\boldsymbol{q} \cdot \boldsymbol{k}) = Var\left(\sum_{i=1}^{d_k} q_i k_i\right) = \sum_{i=1}^{d_k} Var[q_i k_i] = \sum_{i=1}^{d_k} (\sigma_{q_i}^2 + \mu_{q_i}^2)(\sigma_{k_i}^2 + \mu_{k_i}^2) = \sum_{i=1}^{d_k} (1+0)(1+0) = d_k$$

In short, more dimensions tend to produce larger scores, which is problematic because the softmax function uses the exponential function, pushing large values even larger.

The following example Python script should clarify the reason:

```
import numpy as np



def softmax(x):

   return np.exp(x)/np.exp(x).sum()



s1 = np.array([1.0, 2.0, 3.0, 4.0])

s2 = s1 * 10



print(f's1: {softmax(s1)}')

print(f's2: {softmax(s2)}')
```

The output is as follows:

```
s1: [0.0320586 0.08714432 0.23688282 0.64391426]

s2: [9.35719813e-14 2.06106005e-09 4.53978686e-05 9.99954600e-01]
```

The weights in s2 except the last element are almost zero, which makes the gradients very small.

Let the softmax probability of class $i \in \{1,2,...,N\}$ $\blacksquare \in \{1,2,\ldots,\blacksquare\}$ be:
$p_i = \text{softmax}(z_i) = e^{z_i} \sum_k e^{z_k}$ $\blacksquare\blacksquare = \text{softmax}(\blacksquare\blacksquare) = \blacksquare\blacksquare\blacksquare \sum \blacksquare\blacksquare\blacksquare\blacksquare\blacksquare$
Then, the partial derivatives of the softmax with respect to the variable z_j $\blacksquare\blacksquare$ is as follows:

$\partial p_i \partial z_j = p_i(\delta_{ij} - p_j), \delta_{ij} = \{1 \text{if } i=j 0 \text{otherwise} \partial ▯▯ \partial ▯▯ = ▯▯(▯▯▯ - ▯▯), ▯▯▯ = \{1 \text{if } i=j 0 \text{otherwise}$

So, it's like p_i▯▯ and p_j▯▯ are close to 0 or 1. So, any of the partial derivatives will be almost 0.

As such, Vaswani et al. introduced the scaling factor, and they call the attention mechanism "scaled dot-product attention".

3. Multi-Head Attention

A word could have a different meaning or function depending on the context. So, we should use multiple queries per word rather than just one.

In the paper, the author mentioned that they used eight parallel attention calculations. They call each attention function "head". In other words, they used eight heads (h=8).

The base transformer uses 512-dimensional word vectors, which are projected into eight vectors of 64 (=512/8) dimensions—yielding eight representation subspaces. The scaled product-dot attention processes each of eight representations using a different set of projection matrices:

$$\text{head}_i = \text{Attention}(Q_i, K_i, V_i) \qquad (i = 1 \ldots h)$$

$$\text{where} \quad Q_i = YW_i^Q,\ K_i = XW_i^K,\ V_i = XW_i^V$$

Even though we have eight sets of matrix operations, we can perform them in parallel. So, it's very fast.

The next step of the multi-head attention is to concatenate all the eight heads and apply one more matrix operation *W*O.

$$\text{MultiHead}(Y, X) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

Note: I'm using a slightly different notation than the paper to keep similar usage of Q, K, and V from the scaled dot-product attention section.

Since we use the same number of dimensions (=64) for value vectors, the concatenated vectors restore the original dimension (=64×8=512). But they could also use a different number of dimensions for value vectors because *W*O can adjust the final vector dimension.

The transformer uses multi-head attention in multiple ways. One is for encoder-decoder (source-target) attention where Y and X are different language sentences. Another use of multi-head attention is for self-attention, where Y and X are the same sentence.

4. Self-Attention

A word can have a different meaning or function depending on the word itself as well as the words around the word.

For example, in the below two sentences, the word "second" has different meanings:

Give me a **second,** please.

I came **second** in the exam.

But there is only one-word embedding for the word “second”. However, the meaning of the word “second” depends on its context. So, we must treat the word “second” together with its context.

Let’s look at another example.

```
My dog chases after the thief.
```

In the above sentence, we see some functional relationship between “dog” and “chases”. We also see another kind of relationship between “after” and “thief”.

In summary, we need to extract word contexts and relationships within a sentence, where self-attention comes into play.

With self-attention, all queries, keys, and values originate from the same sentence. So, we use multi-head attention like MultiHead(X, X) to extract contexts for each word in a sentence.

Self-attention handles long-range dependences well compared with RNN and CNN. For RNN, we have diminishing gradients that even LSTMs can not completely eliminate. CNN kernels are limited by the kernel size. Self-attention has none of those issues. Moreover, self-attention operations run in parallel and much faster than sequential processing like RNN cells.

Also, we can visually inspect self-attention, making it more interpretable. The paper has a few visualizations of the attention mechanism. For example, the following is a self-attention visualization for the word “making” in layer 5 of the encoder.

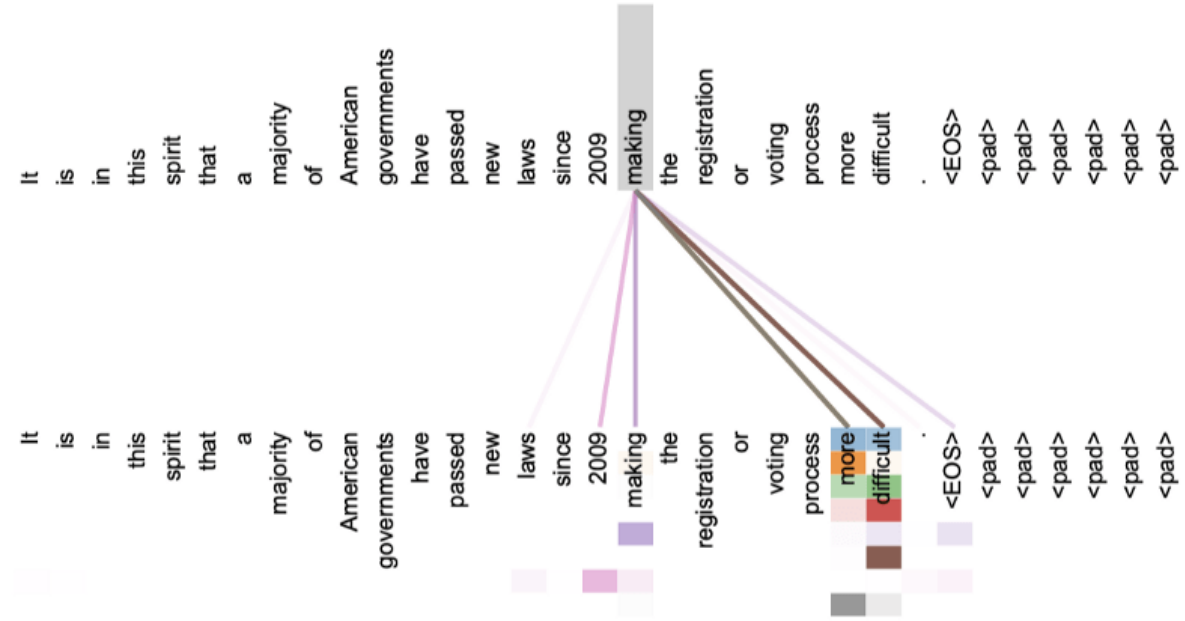


Figure 3 in Attention Is All You Need

There are eight different colors with various intensities, representing the eight attention heads.

It is clear there is a strong relationship between “making” and “more difficult”.

The below image shows the word “its” and the referred words in layer 5 of the encoder (isolating only attention head 5 and attention head 6).

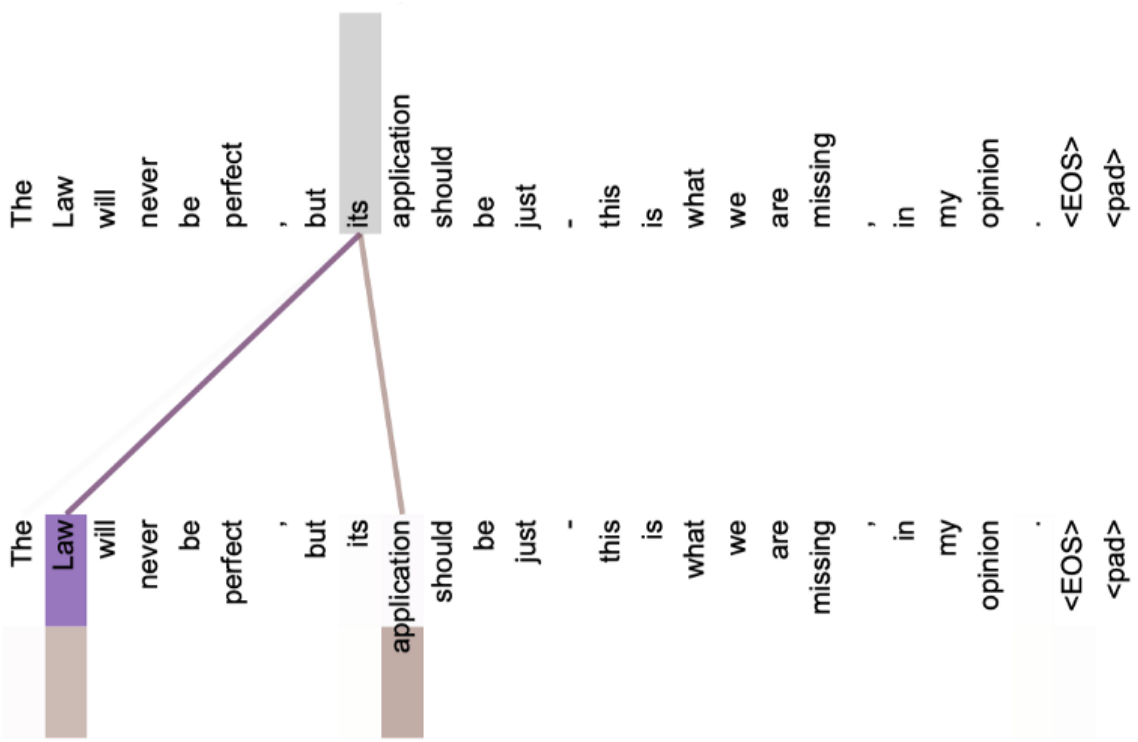


Figure 4 (bottom) in Attention Is All You Need

The word “its” has very strong attention to “Law” and “application”.

The below shows two heads separately (green and red). Each head seems to have learned to perform different tasks based on the structure of the sentence.

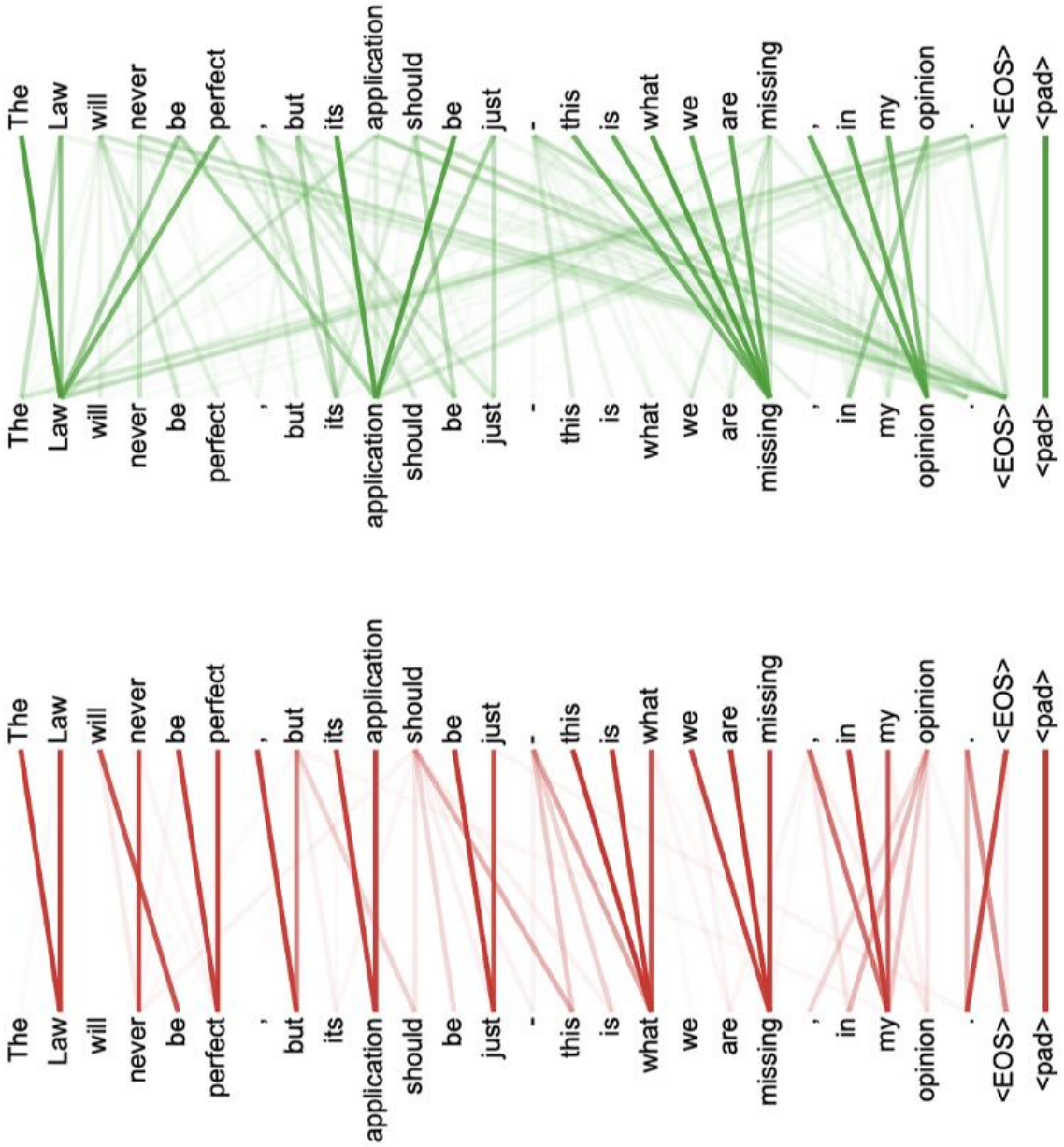
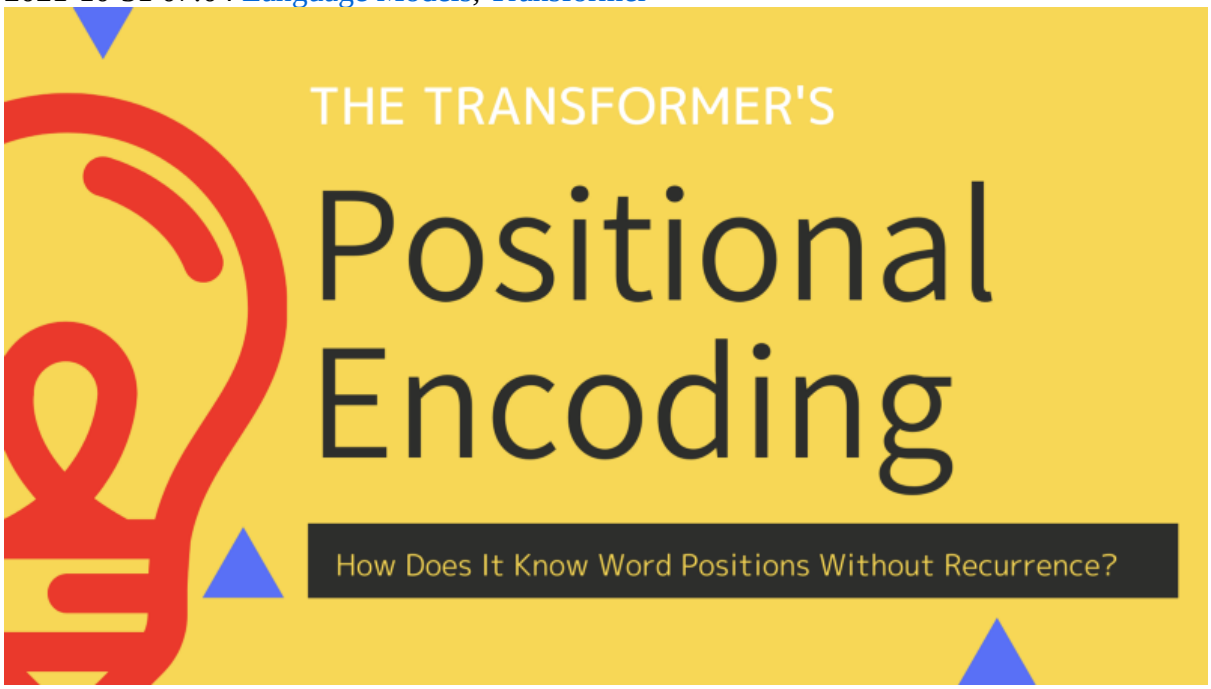


Figure 5 in Attention Is All You Need

Transformer's Positional Encoding: How Does It Know Word Positions Without Recurrence?

2021-10-31 07:04 Language Models, Transformer



In 2017, Vaswani et al. published a paper titled "Attention Is All You Need" for the NeurIPS conference. They introduced the original transformer architecture for machine translation, performing better and faster than RNN encoder-decoder models, which were mainstream.

The big transformer model achieved SOTA (state-of-the-art) performance on NLP tasks.

- On the WMT 2014 English-to-German translation task, it reached a BLEU score of 28.4.
- On the WMT 2014 English-to-French translation task, it achieved a BLEU score of 41.0.

The transformer is more performant and straightforward than other models that it superseded. The author says:

```
We propose a new simple network architecture, the
Transformer, based solely on attention mechanisms,
dispensing with recurrence and convolutions entirely.
```

However, the word “simple” is probably not how most readers feel when looking at the architecture diagram first.

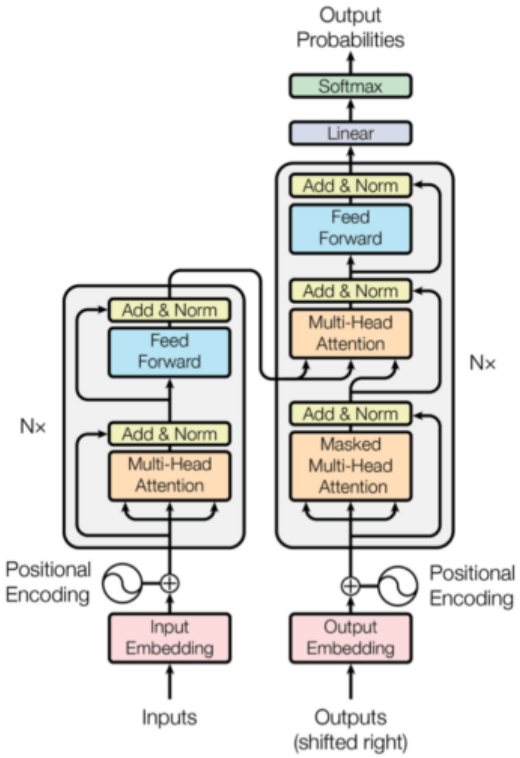


Figure 1: The Transformer—model architecture

Reading the paper is inspirational but provokes many questions, and searching for the answers does not always yield a satisfactory explanation.

Detail-oriented readers might have many doubts about **positional encoding**, which we discuss in this article with the following questions:

1. Why Positional Encoding?

In **3.5 Positional Encoding** of the paper, the author explains why they need to encode the position of each token (word, special character, or whatever distinct unit):

```
Since our model contains no recurrence and no convolution, in order for the model to make use of the order of the sequence, we must inject some information about the relative or absolute position of the tokens in the sequence.
```

Historically, researchers used RNNs (Recurrent Neural Networks) to keep track of ordinal dependencies in sequential data. Later, researchers incorporated attention mechanisms into RNN encoder-decoder architectures to better extract contextual information for each word.

On the contrary, the transformer's encoder-decoder architecture uses attention mechanisms without recurrence and convolution. That's what the paper title "Attention Is All You Need" means.

However, without positional information, an attention-only model might believe the following two sentences have the same semantics:

```
Tom bit a dog.
```

```
A dog bit Tom.
```

That'd be a bad thing for machine translation models. So, yes, we need to encode word positions (note: I'm using 'token' and 'word' interchangeably).

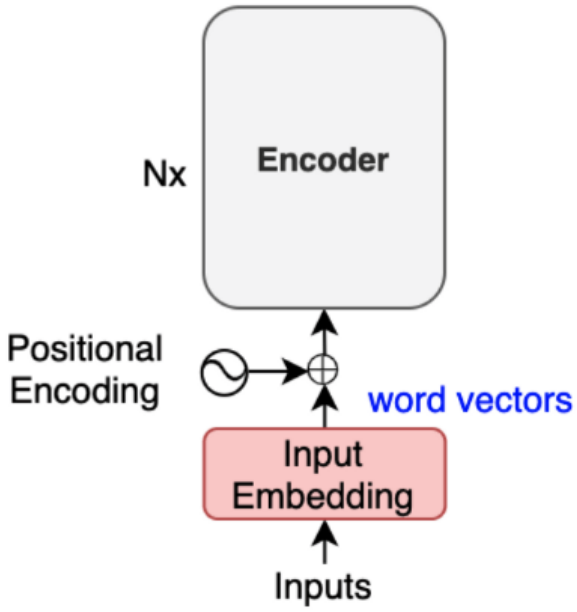
The question is: how?

2. Why Add Positional Encoding To Word Embeddings?

Vaswani et al. explain how they put positional encoding (PE):

```
we add "positional encodings" to the input embeddings
at the bottoms of the encoder and decoder stacks.
```

Let's look at where inputs feed to the encoder.

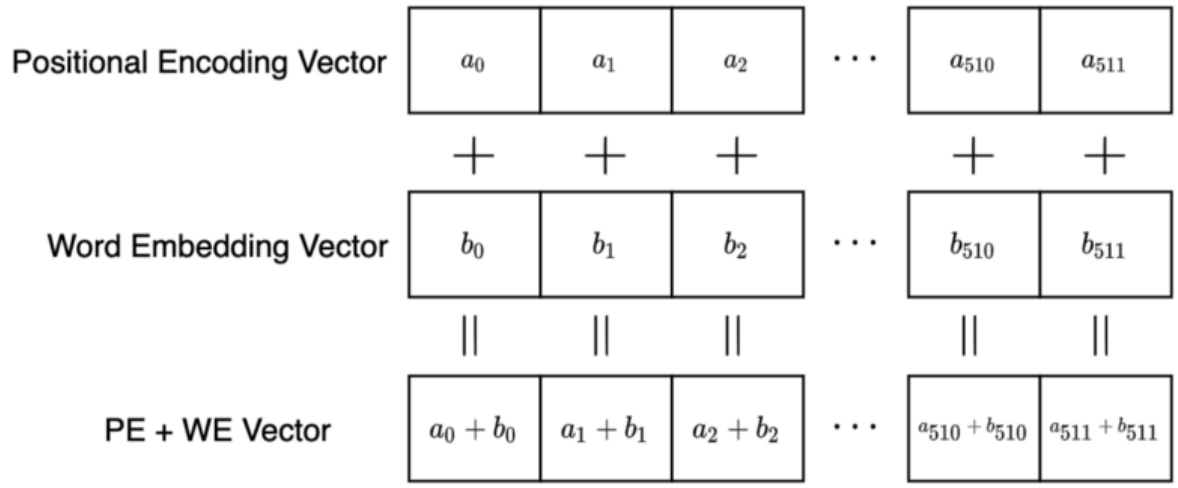


The positional encoding happens after input word embedding and before the encoder.

The author explains further:

> The positional encodings have the same dimension d_model as the embeddings, so that the two can be summed.

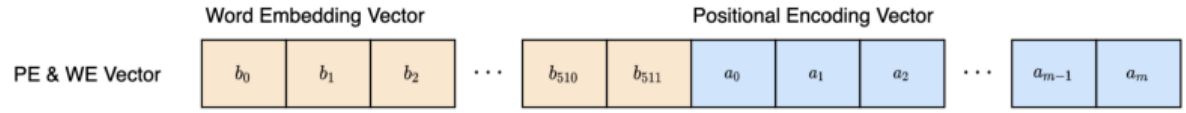
The base transformer uses word embeddings of 512 dimensions (elements). Therefore, the positional encoding also has 512 elements, so we can sum a word embedding vector and a positional encoding vector by element-wise addition.



But why element-wise addition?

Why do we mix two different concepts into the same multi-dimensional space? How can a model distinguish between word embeddings and positional encodings?

Why don’t we use vector concatenation like below?



Using a concatenation of two vectors, the model sees positional encoding vector independent from word embedding, which might make learning (optimization) easier.

However, there are some drawbacks. It increases the memory footprint, and training will likely take longer as the optimization process needs to adjust more parameters due to the extra dimensions.

How about reducing the number of elements in positional encoding? Concatenation does not require two vectors to have equal dimensions. So, we can adjust the size of the encoding vector just big enough to support positional encoding, can’t we? One problem with this approach is that the

number of dimensions in the positional encoding will become another hyper-parameter to tune.

Granted that we can think of various alternative approaches to positional encoding, the truth is they pushed positional encoding vectors into the same space as the word embedding vectors, and it worked.

The model can learn to use the positional information without getting confused with the embedding (semantic) information. It's hard to imagine what's happening inside the network, yet it works.

Moreover, it benefits from requiring less memory and potentially less training time than the hypothetical concatenation approach. Also, we don't need an additional hyper-parameter. So, let's accept this as a viable method and continue reading the paper.

3. How To Encode Word Positions?

How do we calculate positional encoding? The author explains it as follows:

`In this work, we use sine and cosine functions of different frequencies:`

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

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

3.5 Positional Encoding

`where pos is the position and i is the dimension. That is, each dimension of the positional encoding`

corresponds to a sinusoid. The wavelengths form a geometric progression from 2π to 10000 · 2π.

The above formulas look complex. So, let's expand it, assuming the model uses 512 dimensions for positional encoding (and word embedding).

$$d_{model} = 512$$

We represent each token as a vector of 512 dimensions (from element 0 to element 511).

A position (pos) is an integer from 0 to a pre-defined maximum number of tokens in a sentence (a.k.a. max length) minus 1. For example, if the max length is 128, positions are 0 to 127.

For every pair of elements, we calculate the angle (in radian) as follows:

$$\frac{\text{pos}}{10000^{\frac{2i}{512}}}$$

Since we have 512 dimensions, we have 256 pairs of sine and cosine values. As such, the value of i goes from 0 to 255.

Considering everything, we calculate each element of the positional encoding vector for a position (pos) as follows:

$$\text{PE}(\text{pos}, 0) = \sin\left(\frac{\text{pos}}{10000^{\frac{0}{512}}}\right)$$
$$\text{PE}(\text{pos}, 1) = \cos\left(\frac{\text{pos}}{10000^{\frac{0}{512}}}\right)$$
$$\text{PE}(\text{pos}, 2) = \sin\left(\frac{\text{pos}}{10000^{\frac{2}{512}}}\right)$$
$$\text{PE}(\text{pos}, 3) = \cos\left(\frac{\text{pos}}{10000^{\frac{2}{512}}}\right)$$
$$\text{PE}(\text{pos}, 4) = \sin\left(\frac{\text{pos}}{10000^{\frac{4}{512}}}\right)$$
$$\text{PE}(\text{pos}, 5) = \cos\left(\frac{\text{pos}}{10000^{\frac{4}{512}}}\right)$$
$$\vdots$$
$$\text{PE}(\text{pos}, 510) = \sin\left(\frac{\text{pos}}{10000^{\frac{510}{512}}}\right)$$
$$\text{PE}(\text{pos}, 511) = \cos\left(\frac{\text{pos}}{10000^{\frac{510}{512}}}\right)$$

So, the encoding for word position 0 is:

$$\begin{aligned} PE(0) &= \left(\sin\left(\frac{0}{10000^{\frac{0}{512}}}\right), \cos\left(\frac{0}{10000^{\frac{0}{512}}}\right), \sin\left(\frac{0}{10000^{\frac{2}{512}}}\right), \cos\left(\frac{0}{10000^{\frac{2}{512}}}\right), \ldots, \sin\left(\frac{0}{10000^{\frac{510}{512}}}\right), \cos\left(\frac{0}{10000^{\frac{510}{512}}}\right)\right) \\ &= (\sin(0), \quad \cos(0), \quad \sin(0), \quad \cos(0), \quad \ldots, \quad \sin(0), \quad \cos(0)) \\ &= (0, \quad 1, \quad 0, \quad 1, \quad \ldots, \quad 0, \quad 1) \end{aligned}$$

Position 0 is a vector with alternatively repeating 0s and 1s.

Position 1 is as follows:

$$\begin{aligned} PE(1) &= \left(\sin\left(\frac{1}{10000^{\frac{0}{512}}}\right), \cos\left(\frac{1}{10000^{\frac{0}{512}}}\right), \sin\left(\frac{1}{10000^{\frac{2}{512}}}\right), \cos\left(\frac{1}{10000^{\frac{2}{512}}}\right), \ldots, \sin\left(\frac{1}{10000^{\frac{510}{512}}}\right), \cos\left(\frac{1}{10000^{\frac{510}{512}}}\right)\right) \\ &= (0.8414, \quad 0.5403, \quad 0.8218, \quad 0.5696, \quad \ldots, \quad 0.0001, \quad 0.9999) \end{aligned}$$

Note: I truncated element values to the fourth decimal place.

We can see that the sine and cosine pair at the end is close to 0 and 1, which makes sense since the denominator inside the sine and cosine functions becomes very big, that the angle is pretty much zero.

Looking at these numbers makes not much sense. Let's visualize them and see if we can find any patterns.

4. How To Visualize Positional Encoding?

We can write a short Python script to generate all the positional encoding values:

```
import math
import numpy as np

MAX_SEQ_LEN = 128 # maximum length of a sentence
d_model = 512 # word embedding (and positional encoding) dimensions

# pre-allocates vectors with zeros
PE = np.zeros((MAX_SEQ_LEN, d_model))

# for each position, and for each dimension
for pos in range(MAX_SEQ_LEN):
   for i in range(d_model//2):
      theta = pos / (10000 ** ((2*i)/d_model))
      PE[pos, 2*i ] = math.sin(theta)
      PE[pos, 2*i + 1] = math.cos(theta)
```

Instead of using for-loop, we can take advantage of NumPy's parallelizable operations (inspired by the PyTorch tutorial):

```
# replace the above for loop
pos = np.arange(MAX_SEQ_LEN)[:, np.newaxis]
div_term = np.exp(np.arange(0, d_model, 2) * (-math.log(10000.0) / d_model))

PE[:, 0::2] = np.sin(pos * div_term)
PE[:, 1::2] = np.cos(pos * div_term)
```

`np.arange(0, d_model, 2)` generates integers from `0` to `d_model - 2`. Below is an example where `d_model = 10`.

```
>>> np.arange(0, 10, 2)
```

```
array([0, 2, 4, 6, 8])
```

So, we are calculating the below for each `2i` from the `np.arange` term:

$$\exp\left(\frac{2i \times (-\log 10000)}{d_{model}}\right) = \exp\left(\log 10000^{-\frac{2i}{d_{model}}}\right)$$

$$= 10000^{-\frac{2i}{d_{model}}}$$

$$= \frac{1}{10000^{\frac{2i}{d_{model}}}}$$

We use this as the division term in the positional encoding formula:

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

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

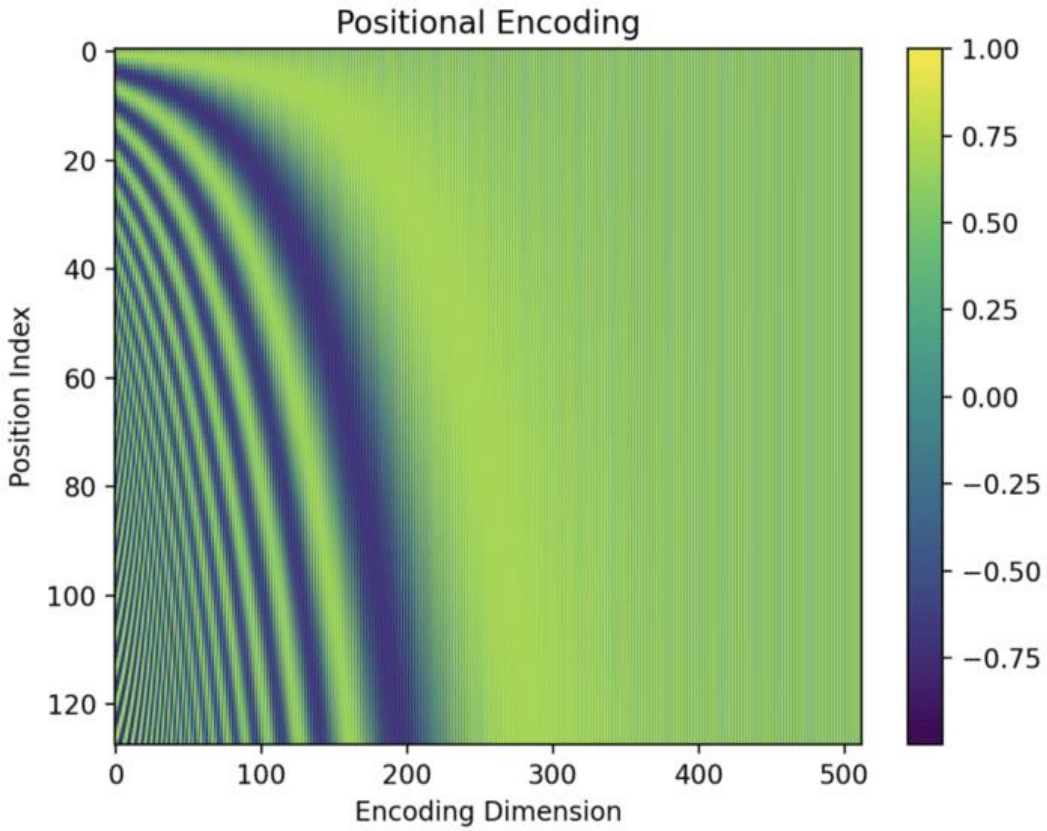
3.5 Positional Encoding

Both codes produce the same values. I like the one with for-loops better because it's easy to understand, and it runs fast enough to experiment with a small MAX_SEQ_LEN. For actual implementations, it'd be better to use the faster version. In either case, we only need to generate positional encoding values once.

Let's draw the positional encoding values as a 2D image:

```
import matplotlib.pyplot as plt

im = plt.imshow(PE, aspect='auto')

plt.title("Positional Encoding")
```

```
plt.xlabel("Encoding Dimension")
plt.ylabel("Position Index")
plt.colorbar(im)
plt.show()
```

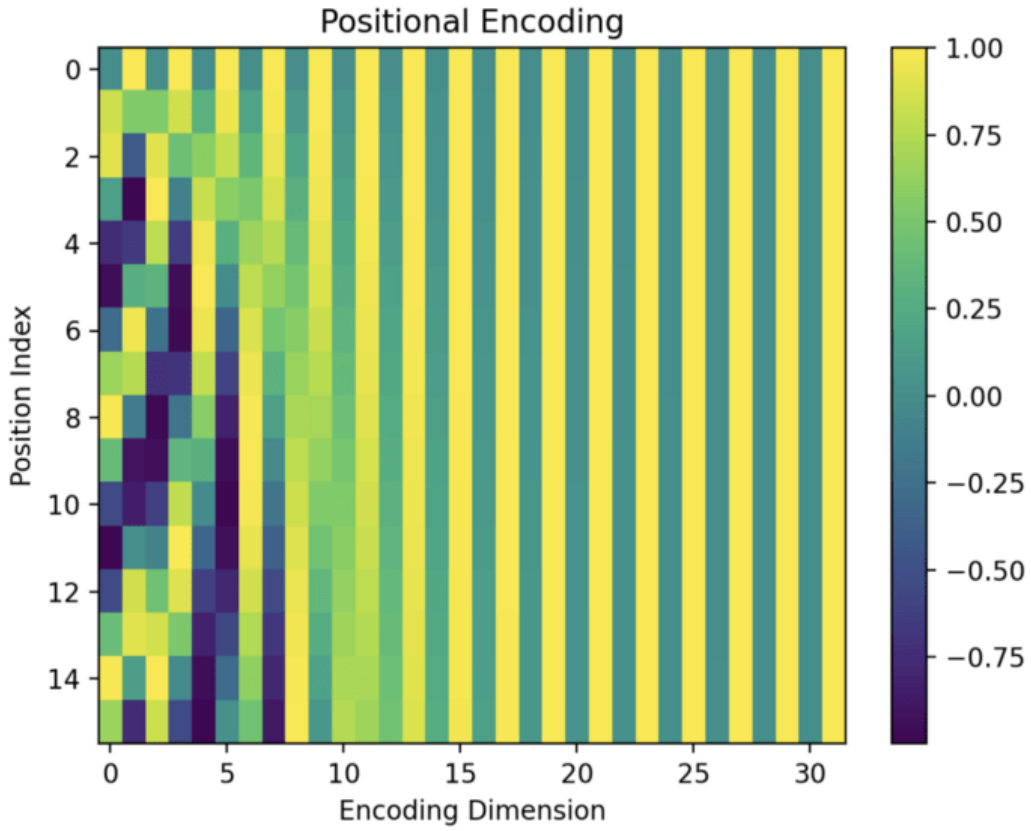


The top row is position 0, and the bottom is position 127 as I used the max length of a sentence = 128. The x-axis is for vector dimension index from 0 to 511.

The values are between -1 and 1 since they are from sine and cosine functions. Darker colors indicate values closer to -1, and brighter (yellow) colors are closer to 1. Green colors indicate values in between. For example, dark green colors indicate values around 0.

As expected, the values towards the right (the end of vector elements) seem to have alternating 0s (dark green) and 1s (yellow). But it's hard to see due to fine pixels.

So, I created a smaller version of the encoding matrix with the max length = 16 and the d_model = 32.

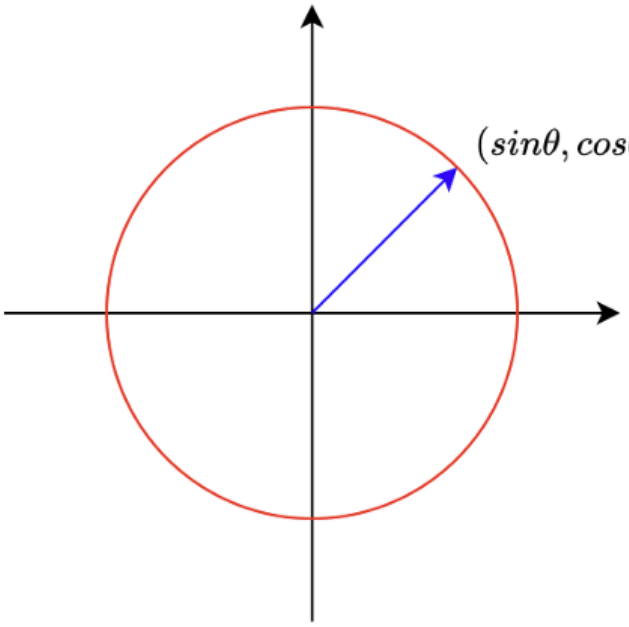


Encoding vectors with bigger position indices have less number of alternating 0s and 1s. Each row appears to have a unique pattern, so I would imagine that the model could somehow distinguish between them. We'll see more details of positional encoding calculation later on.

Next, let's interpret what positional encoding represents.

5. How To Interpret Positional Encoding?

To have more intuition about positional encoding, let's look at it from a different perspective. As we know, positional encoding has pairs of sine and cosine functions. If we take any pair and plot positions on a 2D plane, they are all on a unit circle.



When we increase the value pos, the point moves clockwise. It's like a clock hand that moves as word position increases. In a positional encoding vector with 512 dimensions, we have 256 hands.

Looking at the first two pairs in positional encoding, the second-hand moves slower than the first because the denominator increases as the dimensional index increases.

$$PE(pos, 0) = \sin\left(\frac{pos}{10000^{\frac{0}{512}}}\right)$$

$$PE(pos, 1) = \cos\left(\frac{pos}{10000^{\frac{0}{512}}}\right)$$

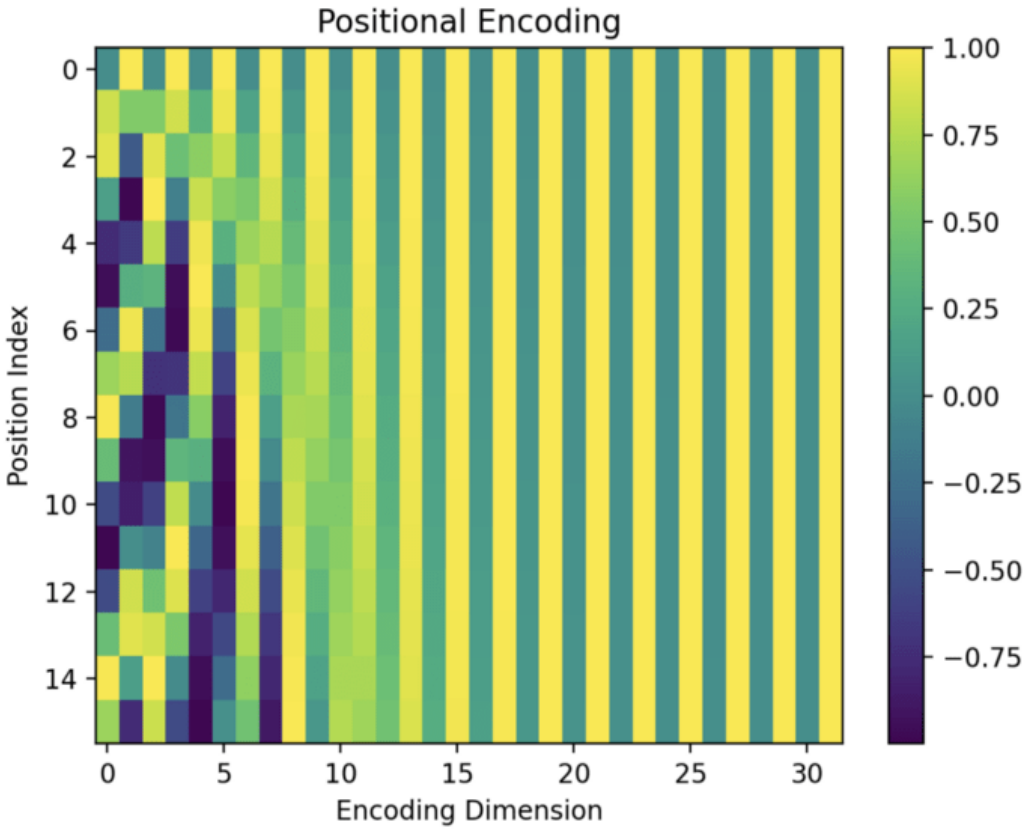
$$PE(pos, 2) = \sin\left(\frac{pos}{10000^{\frac{2}{512}}}\right)$$

$$PE(pos, 3) = \cos\left(\frac{pos}{10000^{\frac{2}{512}}}\right)$$

So, we can interpret positional encoding as a clock with many hands at different speeds. Towards the end of a positional encoding vector, hands move slower and slower when increasing position index (pos).

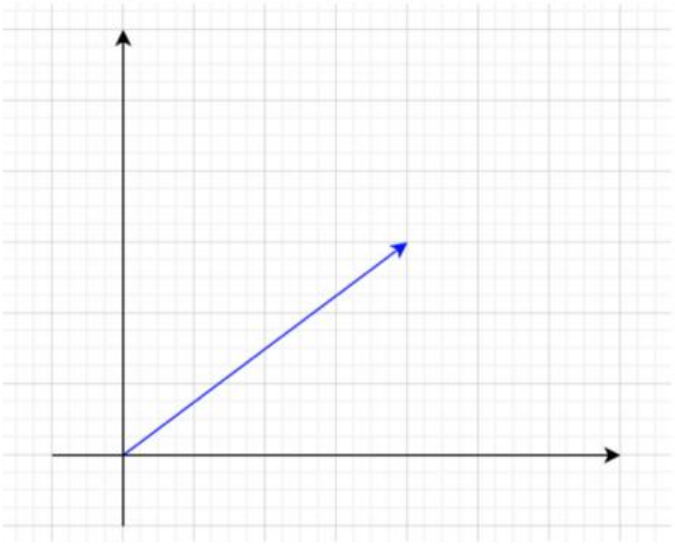
The hands near the end of the dimensions are slow because the denominator is large. The angles are approximately zeros there unless the position index is significant enough.

We can confirm that by looking at the positional encoding image again. Below is the smaller version of the positional encoding image.

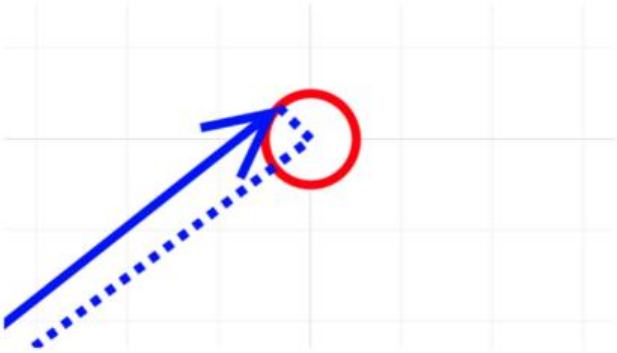


So, we can think of each position having a clock with many hands pointing to a unique time.

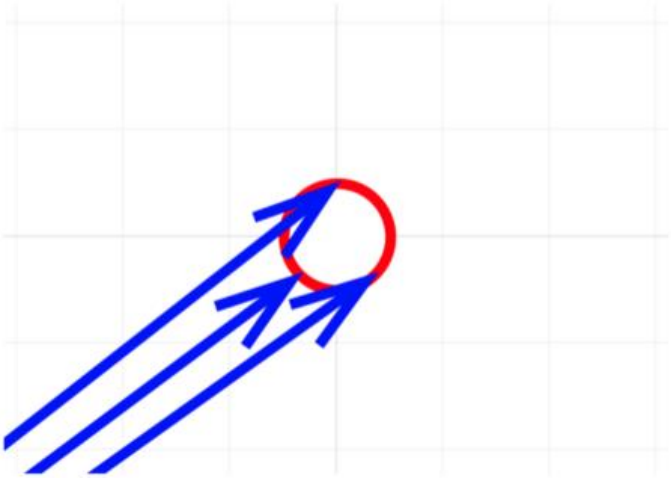
Now, let’s try interpreting the combination of positional encoding and word embedding. If we take the first two dimensions of a word embedding vector, we can draw it in a 2D plane.



Adding the first two dimensions of a positional encoding to this will point somewhere on a unit circle around the word embedding.



So, points on the unit circle represent the same word with different positions.



For higher dimensions, it’s hard to imagine visually. As such, we won’t go into that.

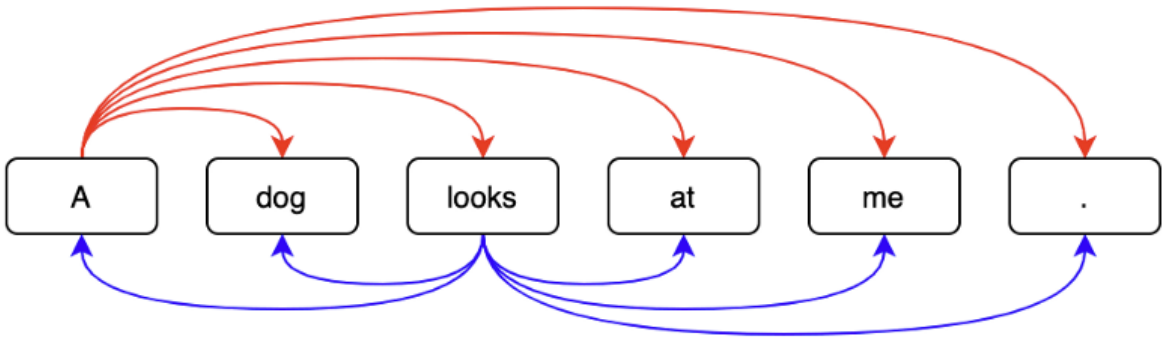
However, there is one question: if word embedding vectors and positional encoding vectors are in similar magnitudes, the model could get confused about which are which. So, how does the model distinguish between word embeddings and positional encodings?

The word embedding layer has learnable parameters, so the optimization process can adjust the magnitudes of word embedding vectors if necessary. I’m not saying that is indeed happening, but it is capable of that. It is amazing that the model somehow learns to use the word embeddings and the positional encodings without mixing them up.

The power of optimization is incredible.

6. Why Sine and Cosine Functions?

We can identify each word position uniquely using positional encoding. But how does the model know the relative positions of words in a sentence?



The author explains their thought process:

We chose this function because we hypothesized it would allow the model to easily learn to attend by relative positions, since for any fixed offset k, PE pos+k can be represented as a linear function of PE pos.

We can rotate each hand in a clock to represent another time. In other words, we can manipulate one-word position to another using a rotation matrix which is a linear operation. Hence, a neural network can learn to understand relative word positions. As the author mentions, it was a hypothesis but sounds quite reasonable.

The author further explains why they chose sine and cosine functions:

We chose the sinusoidal version because it may allow the model to extrapolate to sequence lengths longer than the ones encountered during training.

If a model can identify relative positions of words by rotations, it should be able to detect any relative positions.

So, these are the reasons why they chose sine and cosine functions. They also tested the transformer with positional **embedding**, which encodes positional information using a network layer with learnable parameters.

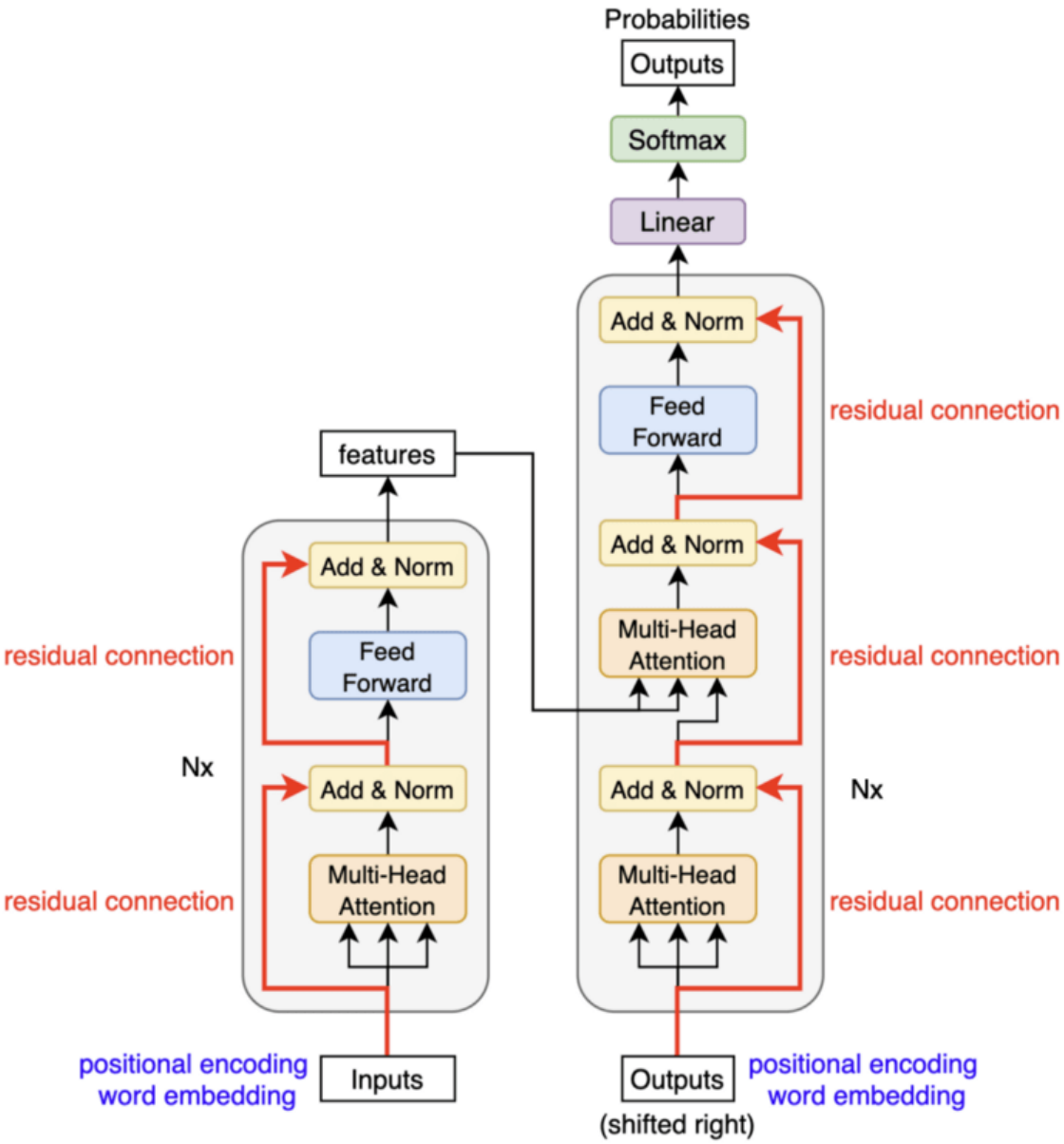
We also experimented with using learned positional embeddings [9] instead, and found that the two versions produced nearly identical results (see Table 3 row (E)).

So, they went for the simpler (fixed) encoding method.

7. How Positions Survive through Multiple Layers?

The encoder in the transformer takes input word embeddings containing positional encodings and applies further operations. If so, why doesn't the model lose the word positions in the process?

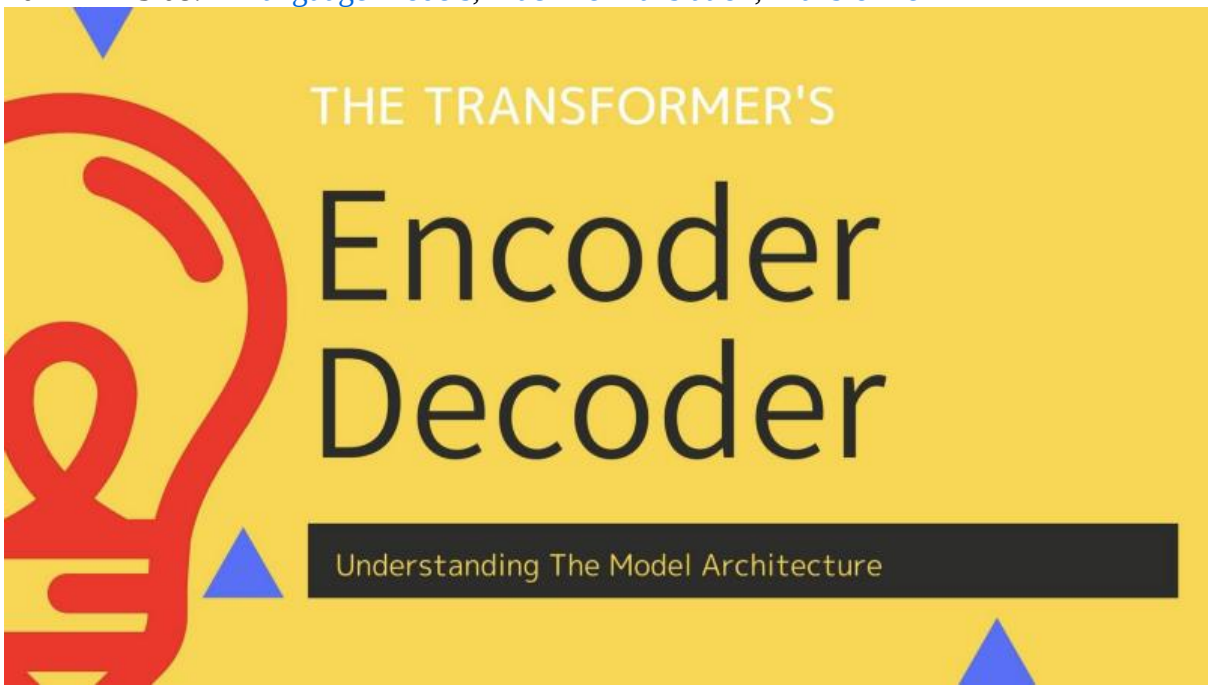
The transformer circumvents this problem by residual connections:



The positional encodings carry over through multiple layers in the encoder and decoder.

Transformer's Encoder-Decoder: Let's Understand The Model Architecture

2021-12-13 08:21 Language Models, Machine Translation, Transformer



In 2017, Vaswani et al. published a paper titled "Attention Is All You Need" for the NeurIPS conference. They introduced the original transformer architecture for machine translation, performing better and faster than RNN encoder-decoder models, which were mainstream.

The transformer architecture is the basis for recent well-known models like BERT and GPT-3. Researchers have already applied the transformer architecture in computer vision and reinforcement learning. So, understanding the transformer architecture is crucial if you want to know where machine learning is making headway.

However, the transformer architecture may look complicated to those without much background.

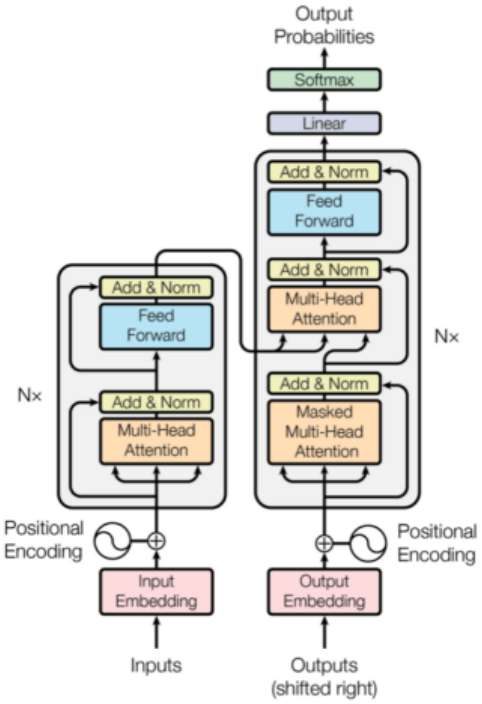


Figure 1 of the paper

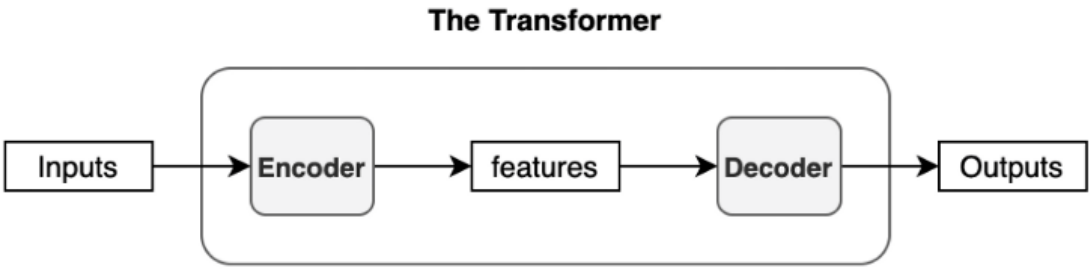
The paper's author says the architecture is simple because it has no recurrence and convolutions. In other words, it uses other common concepts like an encoder-decoder architecture, word embeddings, attention mechanisms, softmax, and so on without the complication introduced by recurrent neural networks or convolutional neural networks.

The transformer is an encoder-decoder network at a high level, which is very easy to understand. So, this article starts with the bird-view of the architecture and aims to introduce essential components and give an overview of the entire model architecture.

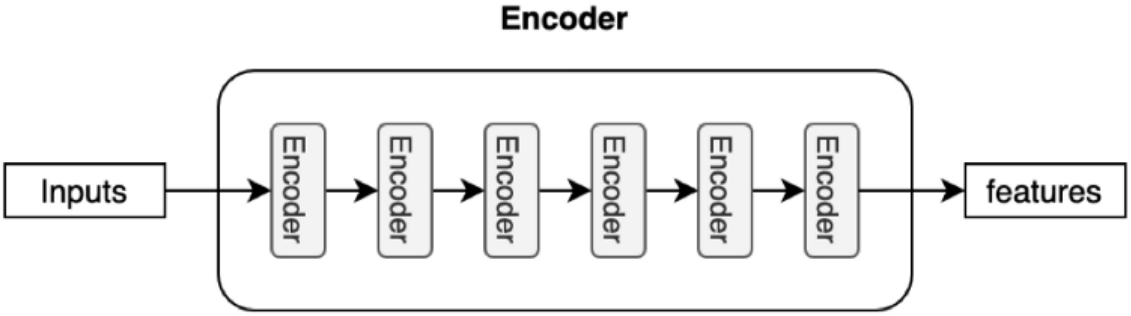
1. Encoder-Decoder Architecture

The original transformer published in the paper is a neural machine translation model. For example, we can train it to translate an English sentence into a French sentence.

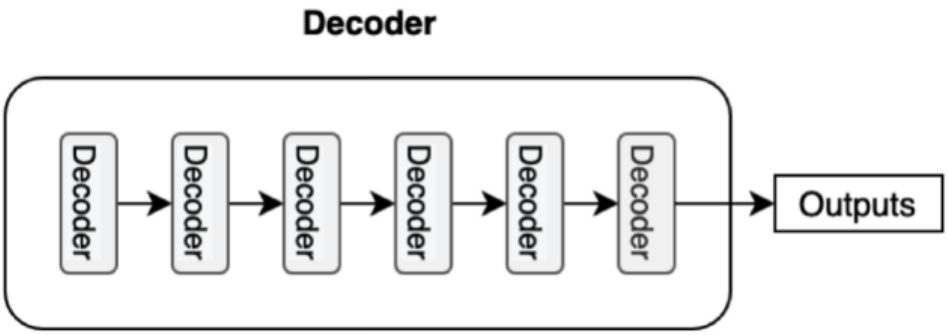
The transformer uses an encoder-decoder architecture. The encoder extracts features from an input sentence, and the decoder uses the features to produce an output sentence (translation).



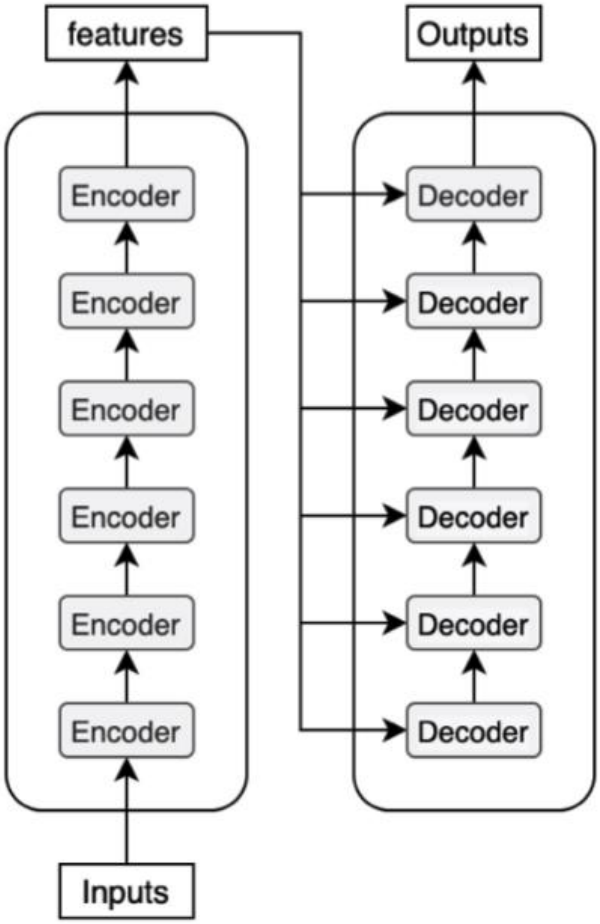
The encoder in the transformer consists of multiple encoder blocks. An input sentence goes through the encoder blocks, and the output of the last encoder block becomes the input features to the decoder.



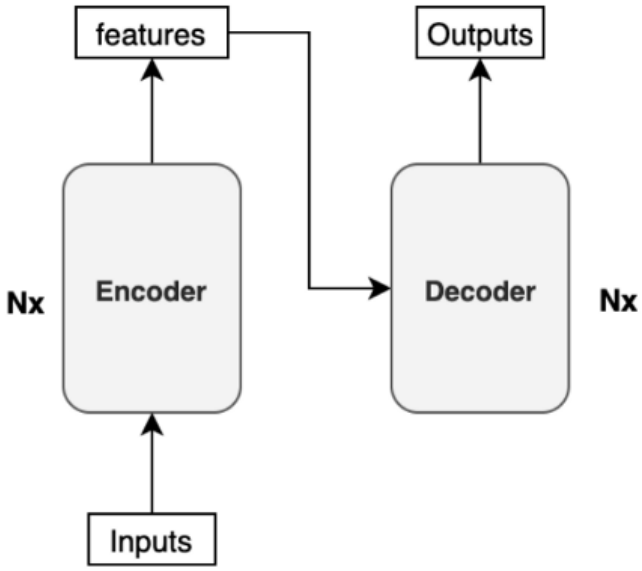
The decoder also consists of multiple decoder blocks.



Each decoder block receives the features from the encoder. If we draw the encoder and the decoder vertically, the whole picture looks like the diagram from the paper.



The paper uses “**Nx**” (N-times) to indicate multiple blocks. So, we can draw the same diagram in a concise format.



Before discussing the encoder/decoder block internals, let’s discuss the inputs and outputs of the transformer.

2. Input Embedding and Positional Encoding

We tokenize an input sentence into distinct elements (tokens) as we often do for any neural translation model. A tokenized sentence is a fixed-length sequence. For instance, if the maximum length is 200, every sentence will have 200 tokens with trailing paddings, which, if intuitively denoted, would look like below:

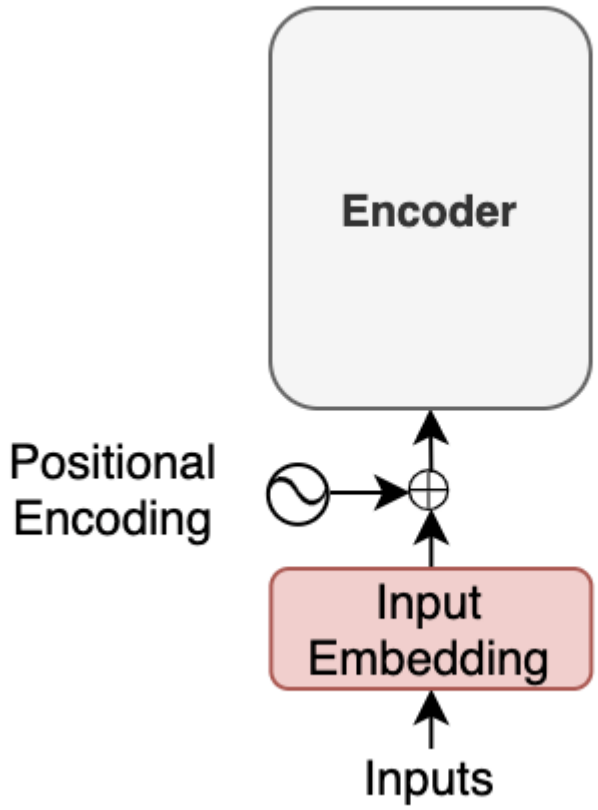
```
(‘Hello’, ‘world’, ‘!’, <pad>, <pad>, …, <pad>)
```

These tokens are typically integer indices in a vocabulary dataset. So, it may be a sequence of numbers like the below:

```
(8667, 1362, 106, 0, 0, …, 0)
```

The number 8667 corresponds to the token `Hello` in this example. It may also contain special characters like `<EOS>` (end-of-sentence marker). It depends on your tokenizer and vocabulary dataset. If you use a model from Hugging Face, there is a specific tokenizer for the model handling such details. If you build a model from scratch, you'd need to decide how to tokenize a sentence, set up a vocabulary dataset, and assign an index to each token.

To feed those tokens into the neural network, we convert each token into an embedding vector, a common practice in neural machine translation and other natural language models. In the paper, they use a 512-dimensional vector for such embedding. So, if the maximum length of a sentence is 200, the shape of every sentence will be (200, 512). The transformer learns those embeddings from scratch during training. If you are unfamiliar with word embeddings, please look at this article.

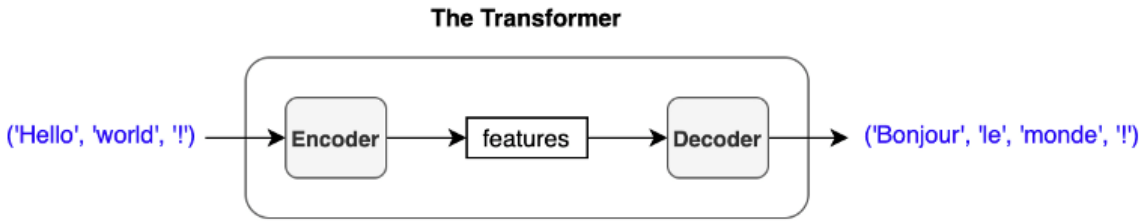


Moreover, with the transformer, we inject **positional encoding** into each embedding so that the model can know word positions without recurrence. For the details of the positional encoding, please take a look at this article.

The point here is that an input to the transformer is not the characters of the input text but a sequence of embedding vectors. Each vector represents the semantics and position of a token. The encoder performs linear algebra operations on those vectors to extract the contexts for each token from the entire sentence and enrich the embedding vectors with helpful information for the target task through the multiple encoder blocks.

3. Softmax and Output Probabilities

The decoder uses input features from the encoder to generate an output sentence. The input features are nothing but enriched embedding vectors.

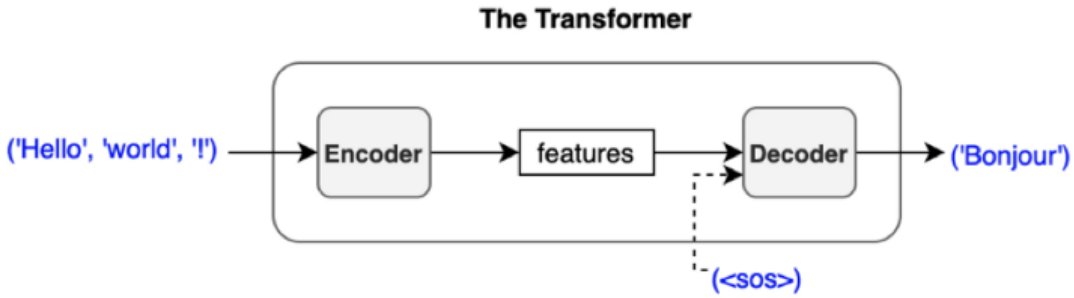


For simplicity, I express a sentence like `('Hello', 'world', '!')`, but the actual inputs to the encoder are input embeddings with positional encodings.

The decoder outputs one token at a time. An output token becomes the subsequent input to the decoder. In other words, a previous output from the decoder becomes the last part of the next input to the decoder. This kind of processing is called "**auto-regressive**"—a typical pattern for generating

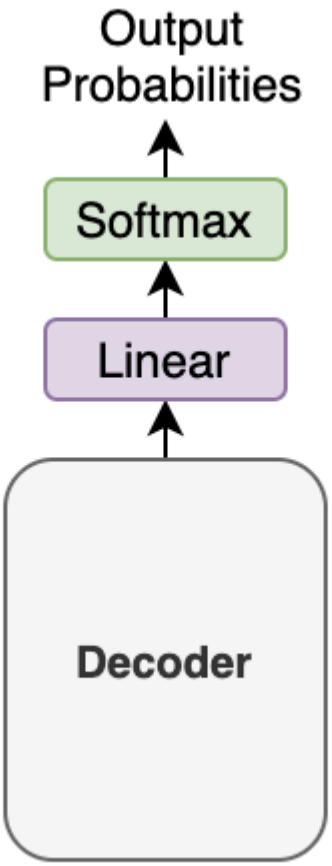
sequential outputs and not specific to the transformer. It allows a model to generate an output sentence of different lengths than the input.

However, we have no previous output at the beginning of a translation. So, we pass the start-of-sentence marker `<SOS>` (as known as the beginning-of-sentence marker `<BOS>`) to the decoder to initiate the translation.



The decoder uses the multiple decoder blocks to enrich `<SOS>` with the contextual information from the input features. In other words, the decoder transforms the embedding vector `<SOS>` into a vector containing information helpful to generating the first translated word (token).

Then, the output vector from the decoder goes through a linear transformation that changes the dimension of the vector from the embedding vector size (512) into the size of vocabulary (say, 10,000). The softmax layer further converts the vector into 10,000 probabilities.

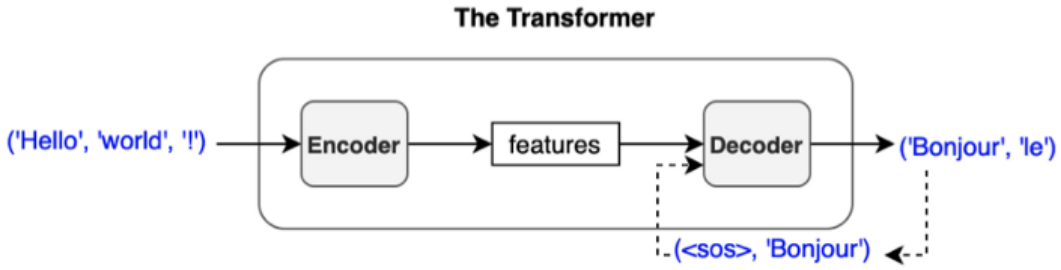


Let’s use a simple example with a tiny vocabulary dataset. It only has three words: `'like','cat', 'I'`. The model predicts the probabilities for the first output token as [0.01, 0.01, 0.98]. In this case, the word `'I'` is most probable. So, we should choose the word `'I'`. Or should we?

It depends. Granted, we must choose one word (token) from the calculated probabilities. This article assumes the greedy method, which selects the most probable word (i.e., the word with the highest probability number).

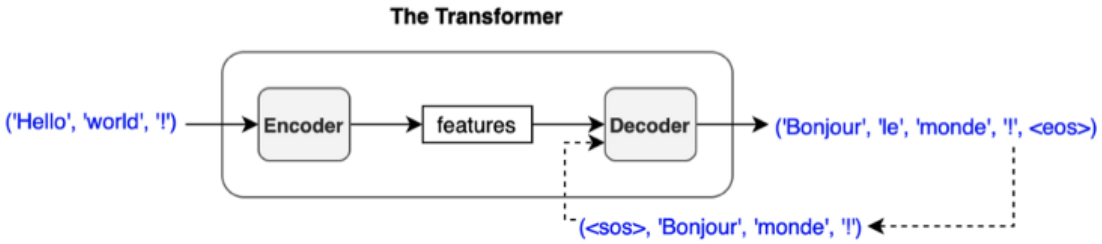
However, another approach, called **beam search**, may produce better performance in the BLUE score. In essence, the beam search looks for the best combination of the tokens rather than selecting the most probable token at a time. The paper also uses the beam search with a beam size of 4.

In any case, we feed the predicted word back to the decoder to produce the next token. As shown below, we provide the chosen token to the decoder as the last part of the next decoder input.



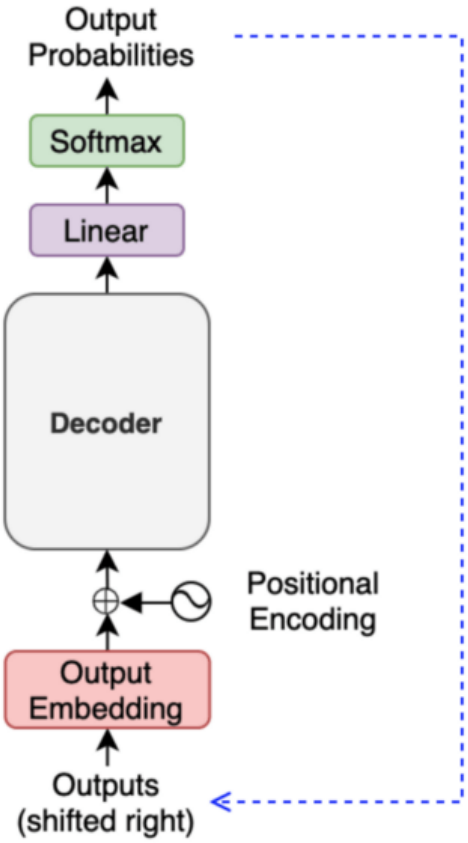
If we choose ‘Bonjour’ as the first output token, the next input to the decoder will be `(<SOS>, 'Bonjour')`. Then, we may get `('Bonjour', 'le')` as output.

The process repeats until the model predicts the end-of-sentence `<EOS>` as the most probable output.

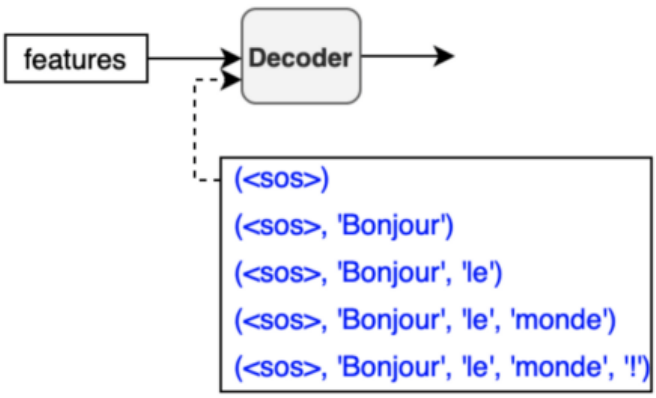


I write sequences like `(<SOS>, 'Bonjour'), (<SOS>, 'Bonjour', 'le')`, and so on but they are not in variable length. Like the encoder input, an input to the decoder is a sequence of tokens in a fixed length. Also, we convert each token to an embedding vector through the embedding layer. So, we are not passing characters or integer indices to the decoder (again, the same as the encoder).

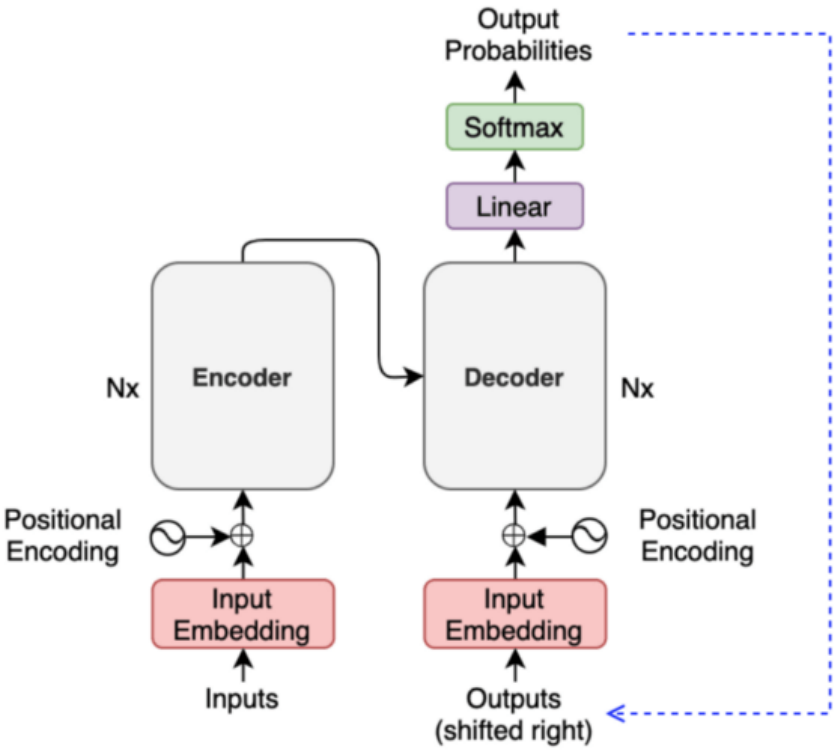
We add an output token to the next input to the decoder. So, the first output token becomes the second input token because the first input is `<SOS>`. Hence, the diagram says ‘**Outputs (shifted right)**’.



It may look like a slow process as we have to generate one output at a time, especially for training. However, during training, we typically use the **teacher-forcing** method, which feeds label tokens (rather than predicted ones), making learning more stable. It also makes the training run faster as we can prepare an attention mask, allowing parallel processing.



The below diagram shows what we have discussed so far:

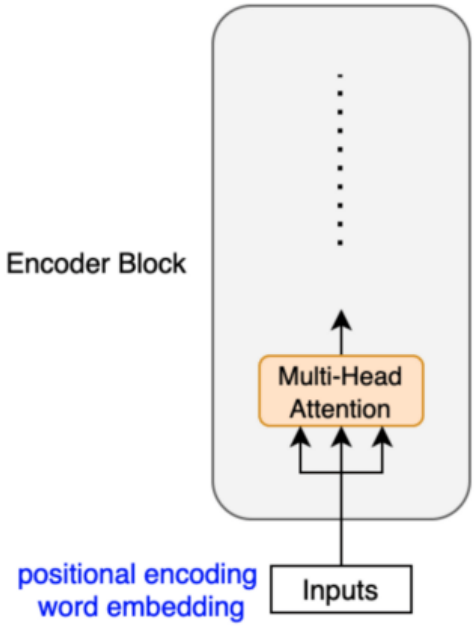


So, we only need to discuss the internals of the encoder and the decoder to understand the overall architecture.

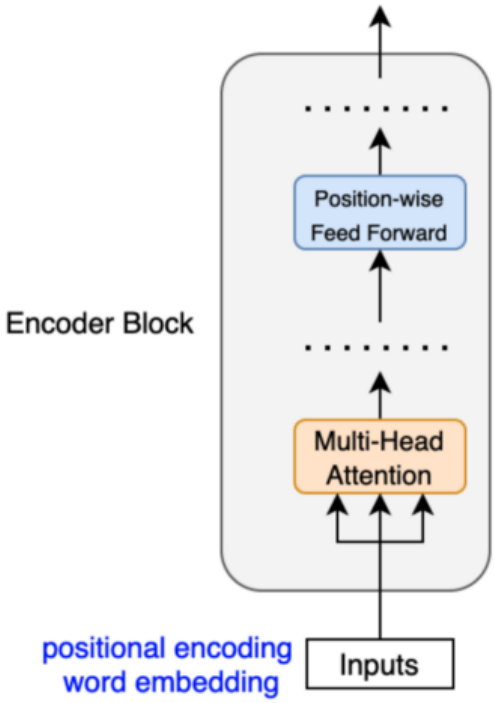
4. Encoder Block Internals

The encoder block uses the **self-attention mechanism** to enrich each token (embedding vector) with contextual information from the whole sentence.

Depending on the surrounding tokens, each token may have more than one semantic and/or function. Hence, the self-attention mechanism employs multiple heads (eight parallel attention calculations) so that the model can tap into different embedding subspaces. For details of the self-attention mechanism, please refer to this article.



The **position-wise feed-forward network** (FFN) has a linear layer, ReLU, and another linear layer, which process each embedding vector independently with identical weights. So, each embedding vector (with contextual information from the multi-head attention) goes through the position-wise feed-forward layer for further transformation.



Let ***x*** be an embedding vector. The position-wise feed-forward network is:

$$\mathrm{FFN}(\boldsymbol{x}) = \mathrm{ReLU}(\boldsymbol{x}W_1 + b_1)W_2 + b_2$$

The dimension of ***x*** increases from 512 to 2048 by *W1* and reduces from 2048 to 512 by *W2*. The weights in FFN are the same across all positions within the same layer. The paper says:

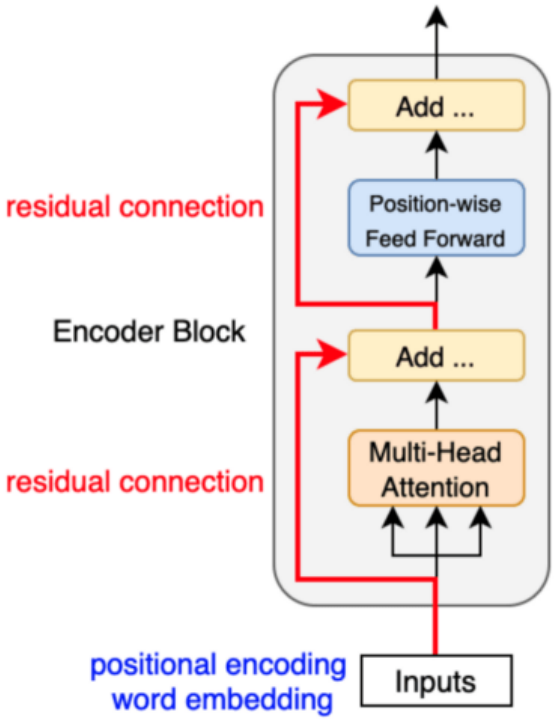
> While the linear transformations are the same across different positions, they use different parameters from layer to layer. Another way of describing this is as two convolutions with kernel size 1.

The paper does not explicitly explain the reason for dimensionality expansion. But it reminds me of the inverted residual block in MobileNetV2, which uses 1×1 convolution to expand dimensionality. Since ReLU discards negative values, it is inevitable to lose information. However, if we increase the dimensionality and then apply ReLU, it is more likely to preserve information better. Therefore, we can introduce non-linearity (ReLU) without losing much information, thanks to the intermediate dimensionality expansion.

Another point is that the encoder block uses **residual connections**, which is simply an element-wise addition:

$$\boldsymbol{x} + \mathrm{Sublayer}(\boldsymbol{x})$$

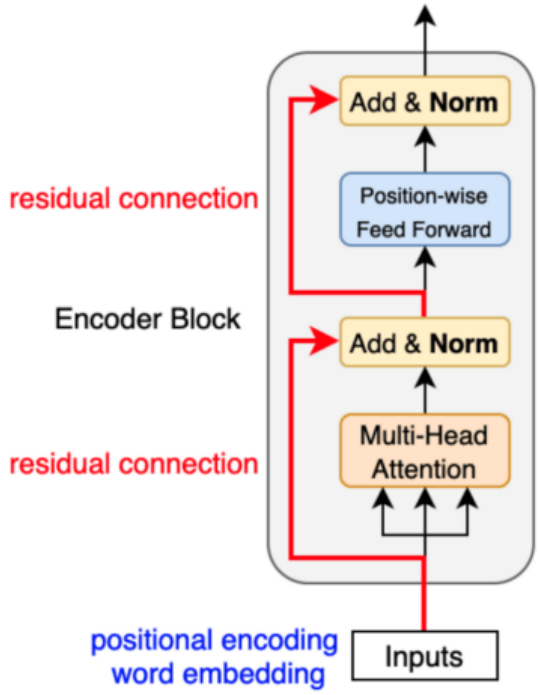
Note: Sublayer is either multi-head attention or point-wise feed-forward network.



Residual connections carry over the previous embeddings to the subsequent layers. As such, the encoder blocks enrich the embedding vectors with additional information obtained from the multi-head self-attention calculations and position-wise feed-forward networks.

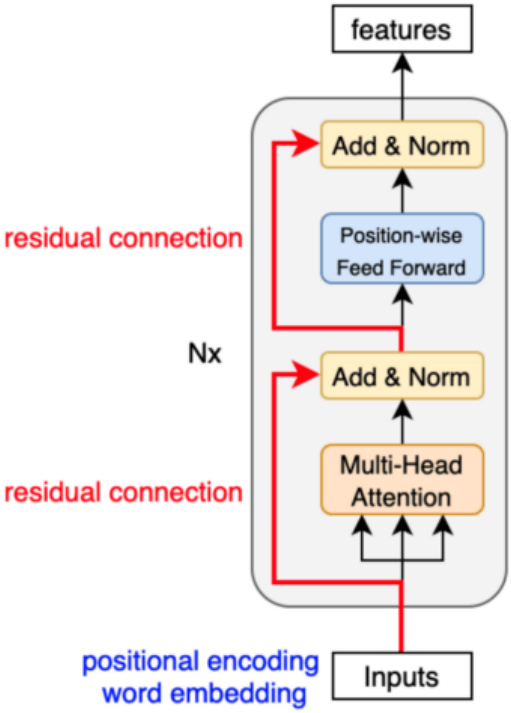
After each residual connection, there is a **layer normalization**:

$$\text{LayerNorm}(\boldsymbol{x} + \text{Sublayer}(\boldsymbol{x}))$$



Like batch normalization, layer normalization aims to reduce the effect of covariant shift. In other words, it prevents the mean and standard deviation of embedding vector elements from moving around, which makes training unstable and slow (i.e., we can't make the learning rate big enough). Unlike batch normalization, layer normalization works at each embedding vector (not at the batch level). Originally, Geoffrey Hinton's team introduced layer normalization as it was not practical to apply batch normalization to recurrent neural networks.

The transformer uses six stacked encoder blocks. The outputs from the last encoder block become the input features for the decoder.

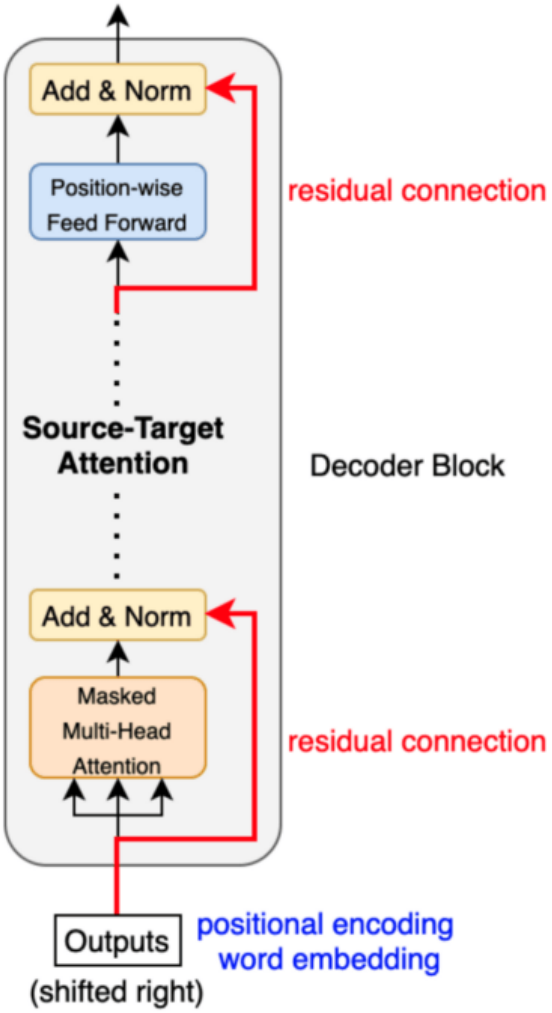


As we have seen so far, the input features are nothing but a sequence of enriched embeddings through the multi-head attention mechanisms and position-wise feed-forward networks with residual connections and layer normalizations.

Let’s look at how the decoder uses the input features.

5. Decoder Block Internals

The decoder block is similar to the encoder block, except it calculates the source-target attention.



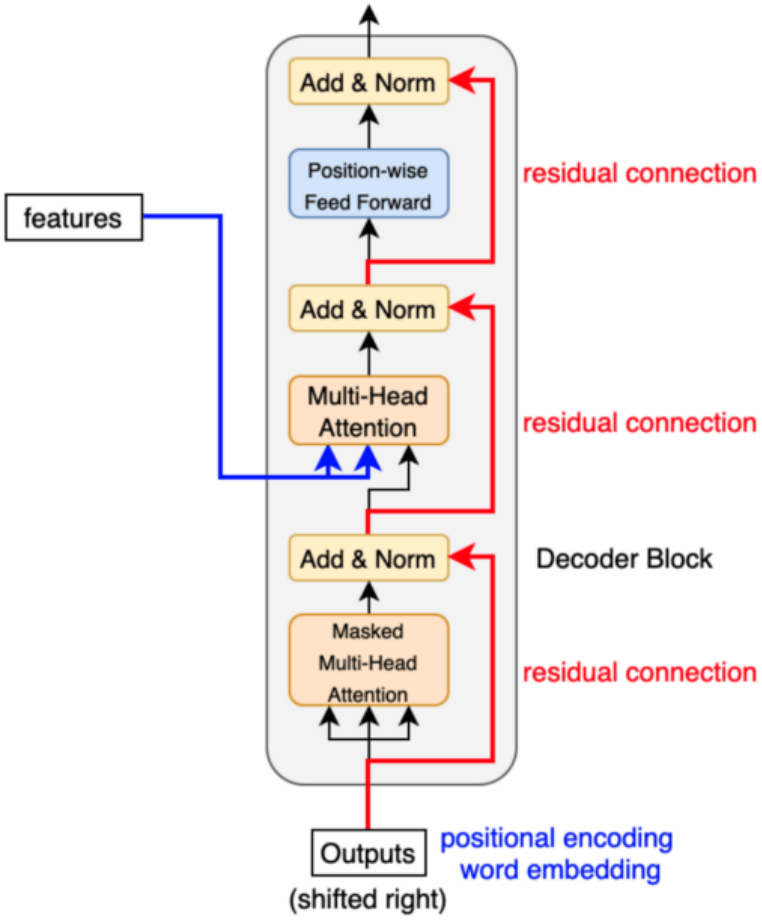
As mentioned before, an input to the decoder is an output shifted right, which becomes a sequence of embeddings with positional encoding. So, we can think of the decoder block as another encoder generating enriched embeddings useful for translation outputs.

Masked multi-head attention means the multi-head attention receives inputs with masks so that the attention mechanism does not use information from the hidden (masked) positions. The paper mentions that they used the mask inside the attention calculation by setting attention scores to negative infinity (or a very large negative number). The softmax within the attention mechanisms effectively assigns zero probability to masked positions.

Intuitively, it is as if we were gradually increasing the visibility of input sentences by the masks:

```
(1, 0, 0, 0, 0, …, 0) => (<SOS>)

(1, 1, 0, 0, 0, …, 0) => (<SOS>, 'Bonjour')

(1, 1, 1, 0, 0, …, 0) => (<SOS>, 'Bonjour', 'le')

(1, 1, 1, 1, 0, …, 0) => (<SOS>, 'Bonjour', 'le', 'monde')

(1, 1, 1, 1, 1, …, 0) => (<SOS>, 'Bonjour', 'le', 'monde', '!')
```

The source-target attention is another multi-head attention that calculates the attention values between the features (embeddings) from the input sentence and the features from the output (yet partial) sentence.

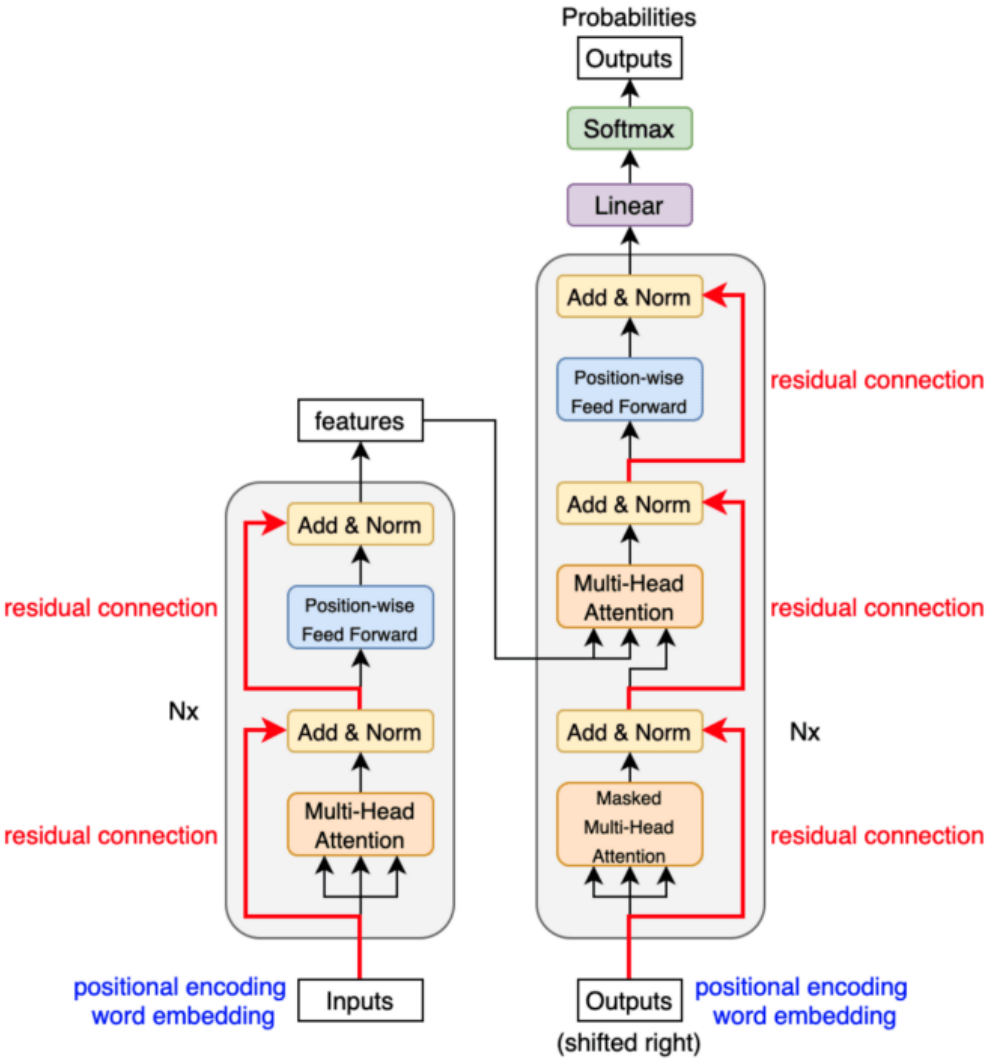


So, the decoder block enriches the embeddings using features from the input and partial output sentences.

6. Conclusion

The transformer architecture assumes no recurrence or convolution pattern when processing input data. As such, the transformer architecture is suitable for any sequence data. As long as we can express our input as sequence data, we can apply the same approach, including computer vision (sequences of image patches) and reinforcement learning (sequences of states, actions, and rewards).

In the case of the original transformer, the mission is to translate, and it uses the architecture to learn to enrich embedding vectors with relevant information for translation.



I hope the transformer architecture looks as simple to you by now as the paper author believes it is.