Transformers have revolutionized natural language processing with their use of self-attention mechanisms. In this comprehensive blog post, we will build an intuition about how self-attention works and why it is so powerful.

Introduction

The Transformer architecture was first introduced in the 2017 paper "**Attention is All You Need**" by researchers at Google. Unlike previous sequence models such as RNNs, the Transformer relies entirely on self-attention to model dependencies in sequential data like text.

Remarkably, this simple change led to major improvements in machine translation quality over existing methods. Since then, Transformers have been applied successfully to diverse NLP tasks like text generation, summarization, and question-answering. Their versatility has even led to applications in computer vision.

But what exactly is self-attention and why is it so effective? In this post, we'll develop an intuition behind self-attention by stepping through a concrete example.

The Limitations of RNNs

Recurrent neural networks (RNNs) used to be the dominant approach for modeling sequences. An RNN processes textual data incrementally, maintaining a "memory" of the previous context. For example, to predict the next word in a sentence, an RNN model would incorporate information about all the preceding words.

However, RNNs have certain limitations. They process data sequentially, making parallelization difficult. More critically, they struggle to learn long-range dependencies because the information gets diluted over many time steps. Attention mechanisms were proposed to mitigate this issue.

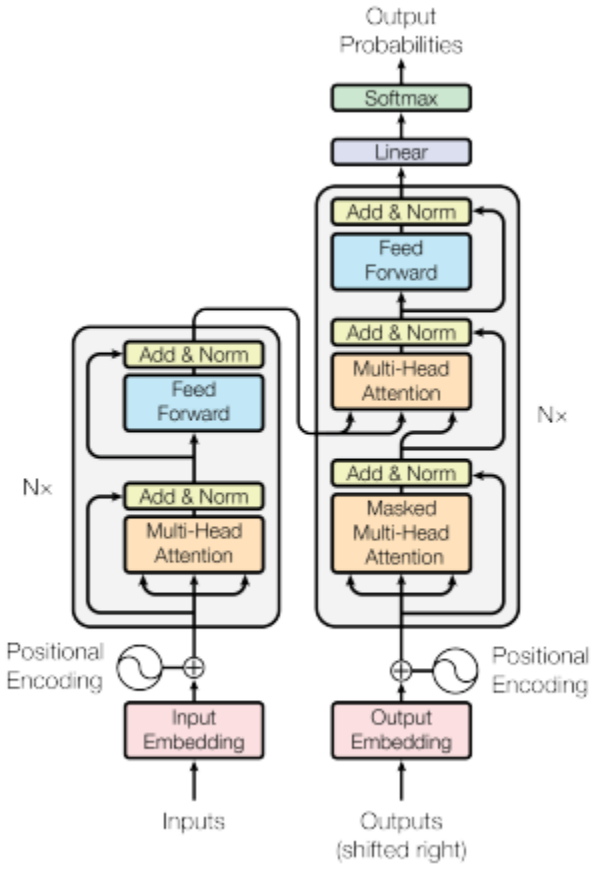
Why Use a Transformer Model?

The transformer architecture has enabled the development of new models that can be trained on large datasets and significantly outperform recurrent neural networks like LSTMs. These new models are utilized for tasks like sequence classification, question answering, language modeling, named entity recognition, summarization, and translation.

Let’s examine the key components of transformers to understand how they have become the foundation for state-of-the-art performance on different NLP tasks.

Transformer Design

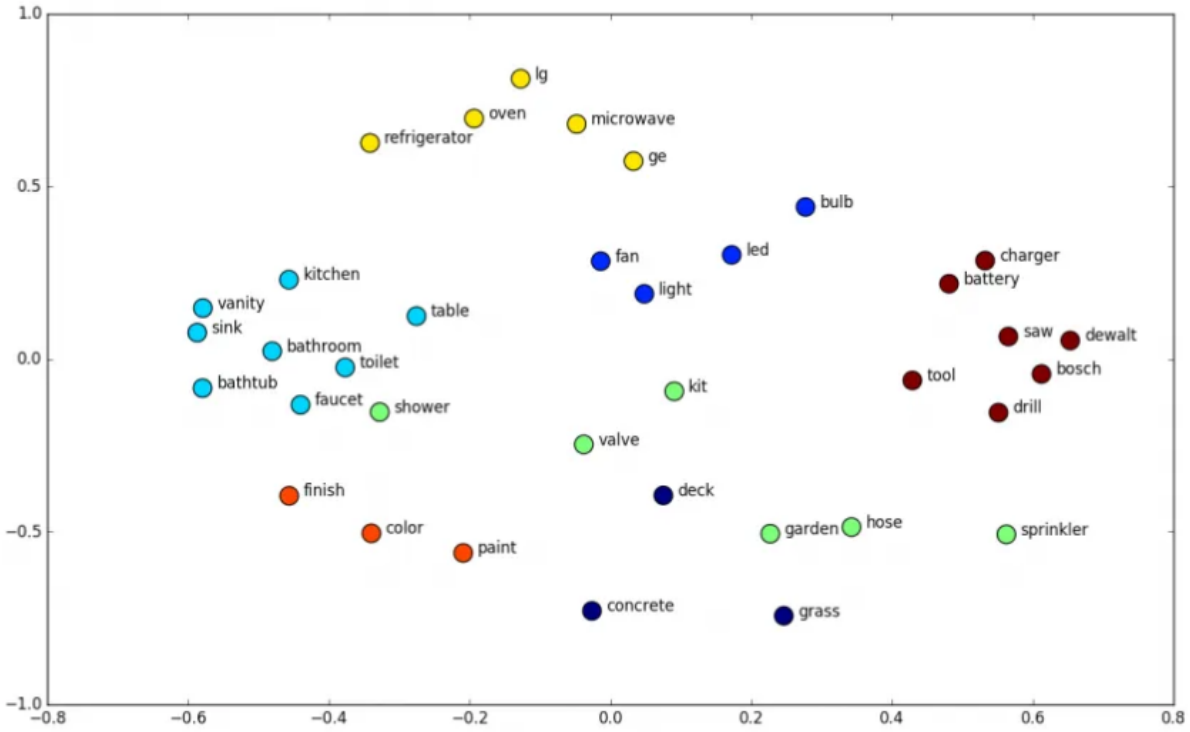
A transformer consists of an encoder and a decoder. The encoder's role is to encode the inputs (i.e. sentences) into a state, often containing multiple tensors. This state is then passed to the decoder to generate the outputs. In machine translation, the encoder converts a source sentence, e.g. "**Hello world**", into a state, such as a vector, that captures its semantic meaning. The decoder then utilizes this state to produce the translated target sentence, e.g. "**Bonjour le monde**." Both the encoder and decoder primarily employ Multi-Head Attention and Feedforward Networks, which are the main focus of this article.



Key Transformer Components

1. **Input Embedding**
   Embedding aims to create a vector representation of words where words with similar meanings will be close in terms of Euclidean distance. For instance, the words "bathroom" and "shower" are related to the same concept, so their word vectors are close in Euclidean space as they convey similar meanings.

For the encoder, the authors opted for an embedding size of 512 (i.e. each word is represented by a 512-dimensional vector).



2. **Positional Encoding**

The position of a word plays a crucial role in understanding the sequence we want to model. Therefore, we add positional information about the word's location in the sequence to its vector. The authors used the following sinusoidal functions (see Figure 2) to represent a word's position within a sequence.

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

We will explain positional encoding in more detail with an example.

```
The big yellow cat
1    2    3     4
```

We note the position of each word in the sequence.

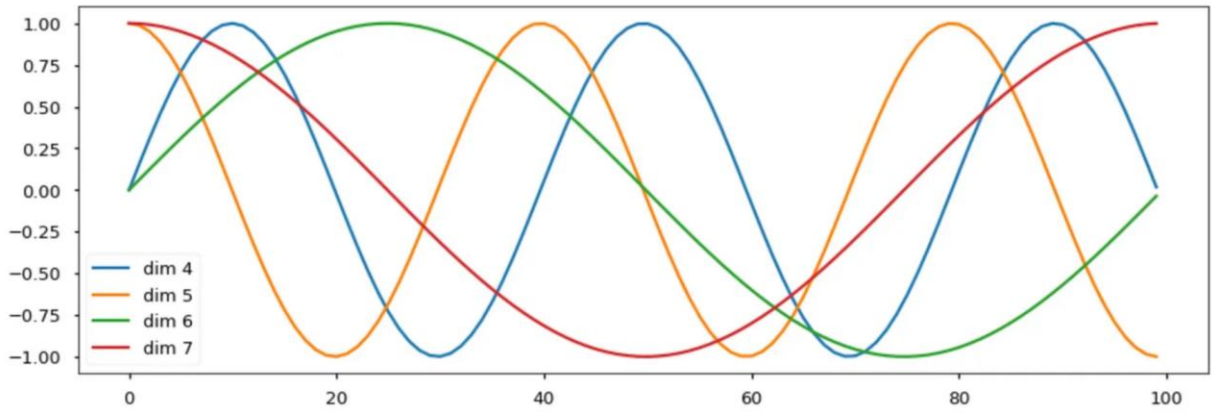
We define d_{model} = 512, which represents the size of the embedding vector of each word (i.e. the vector dimension). We can now rewrite the two positional encoding equations as:

$$p_t^{2i} = \sin(\lambda_t.t)$$

$$p_t^{2i+1} = \cos(\lambda_t.t)$$

$$\lambda_t = \frac{1}{10000^{\frac{2i}{d_m odel}}}$$

We can see that the wavelength (i.e. frequency) lambda_t decreases as the dimension increases, this forms a progression along the wave from 2pi to 10000.2pi.

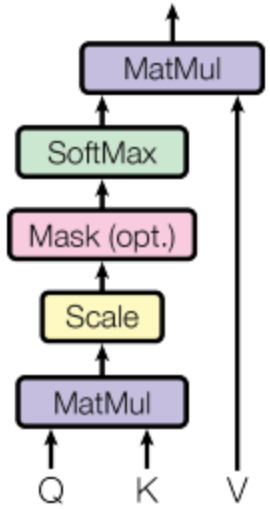


In this model, the absolute positional information of a word in a sequence is added directly to its initial vector. For this, the positional encoding must have the same size dmodel as the initial word vector.

3. Attention Mechanism

3.1. Scaled Dot-Product Attention

Scaled Dot-Product Attention



Let's explain the attention mechanism. The key goal of attention is to estimate the relative relevance of the keywords compared to the query word for the same entity. For this, the attention mechanism takes a query vector Q representing a word, the keys K comprising all other words in the sentence, and values V representing the word vectors.

In our case, V = Q (for the two self-attention layers). In other words, the attention mechanism provides the significance of a word in a given sentence.

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

When we compute the normalized dot product between the query and the keys, we get a tensor that represents the relative importance of each other word for the query. To go deeper into mathematics, we can try to understand why the authors used a dot product to calculate the relation between two words.

A word is represented by a vector in Euclidian space, in this case, a vector of size 512.

When computing the dot product between **Q** and **KT**, we calculate the product between Q's orthogonal projection onto K. In other words, we estimate the alignment between the query and keyword vectors, returning a weight for each word in the sentence.

We then normalize by dk to counteract large **Q** and **K** magnitudes which can push the softmax function into regions with tiny gradients. The softmax function regularizes the terms and rescales

them between 0 and 1 (i.e., converts the dot product to a probability distribution), with the goal of normalizing all weights between 0 and 1.

The matrices **Q**, **K** and **V** are just the input sentence.

$$\text{softmax}\left(\frac{Q \times K^T}{\sqrt{512}}\right) = \text{(6, 6)}$$

Q: (6, 512) × K^T: (512, 6)

	YOUR	CAT	IS	A	LOVELY	CAT	Σ
YOUR	0.268	0.119	0.134	0.148	0.179	0.152	1
CAT	0.124	0.278	0.201	0.128	0.154	0.115	1
IS	0.147	0.132	0.262	0.097	0.218	0.145	1
A	0.210	0.128	0.206	0.212	0.119	0.125	1
LOVELY	0.146	0.158	0.152	0.143	0.227	0.174	1
CAT	0.195	0.114	0.203	0.103	0.157	0.229	1

*all values are random.

*for simplicity I considered only one head, which makes $d_{model} = d_k$.

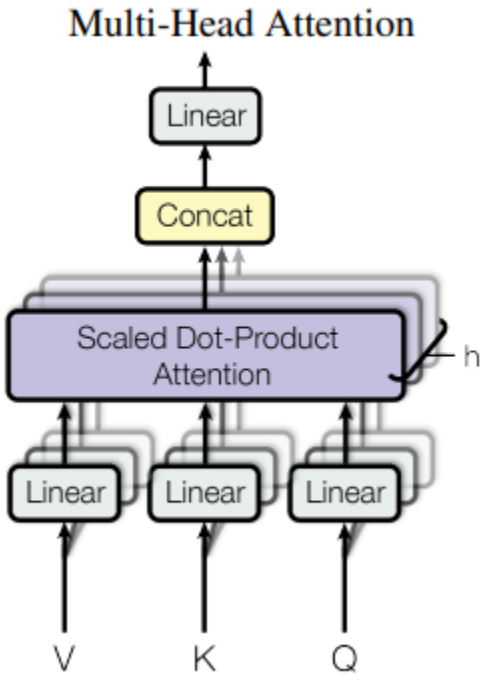
Finally, we multiply the weights (i.e., importances) by the values V to reduce irrelevant words and focus on the most significant words.

	YOUR	CAT	IS	A	LOVELY	CAT
YOUR	0.268	0.119	0.134	0.148	0.179	0.152
CAT	0.124	0.278	0.201	0.128	0.154	0.115
IS	0.147	0.132	0.262	0.097	0.218	0.145
A	0.210	0.128	0.206	0.212	0.119	0.125
LOVELY	0.146	0.158	0.152	0.143	0.227	0.174
CAT	0.195	0.114	0.203	0.103	0.157	0.229

(6, 6) × **V** (6, 512) = **Attention** (6, 512)

Each row in this matrix captures not only the meaning (given by the embedding) or the position in the sentence (represented by the positional encodings) but also each word's interaction with other words.

3.2. Multi-Head Attention

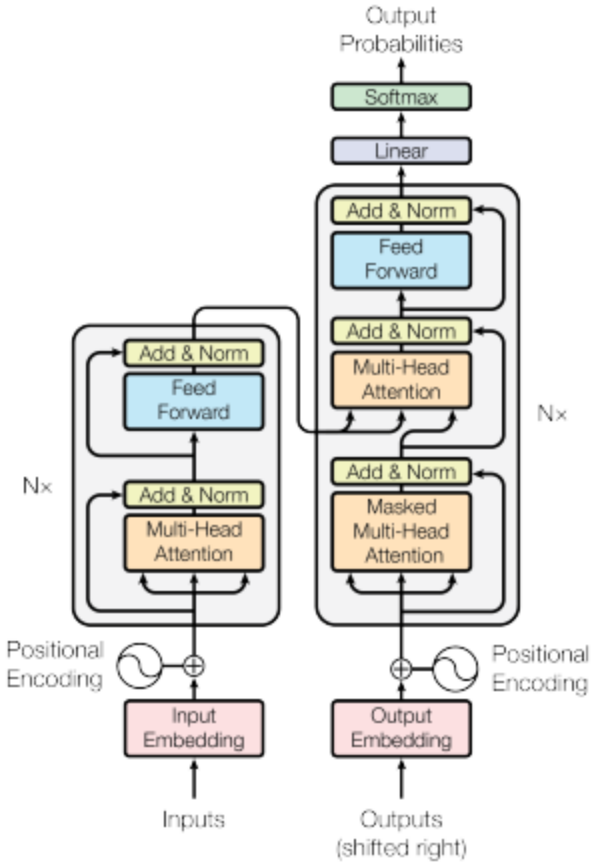


The key idea is that attention is applied multiple times in parallel on different projections of the input queries, keys, and values. This allows the model to learn different types of dependencies between the input words.

The input **queries (Q**), **keys (K),** and **values (V)** are each linearly projected h times into smaller subspaces. For example, h=8 times into 64-dimensional spaces.
Attention is then applied in each of these h projected subspaces in parallel, yielding h different attention outputs.

These h outputs are concatenated and linearly projected again to get the final values.
The projections allow the model to focus on different positional and semantic relationships between words since each projected subspace captures different information.
Doing this in parallel (multi-head) instead of sequentially improves efficiency.
The projection matrices are learned during training to discover the most useful projections.
So, in summary, multi-head attention applies the attention mechanism in multiple parallel subspaces to learn different types of dependencies between words in an efficient way.

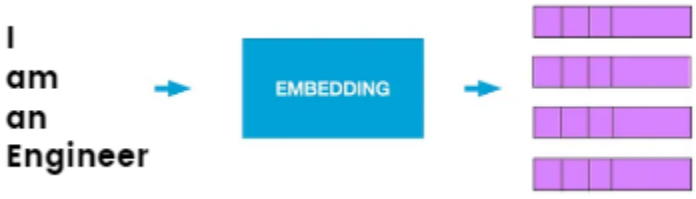
Let's dive into the mechanics of encoder-decoder architecture.



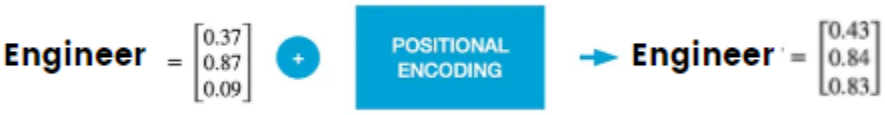
Source: [Attention is all you need.]

In this section, we'll explain how the encoder and decoder work together to translate an English sentence into a French one, step by step.

1. **Encoder**
    1.1. Convert a sequence of tokens to a sequence of vectors by using embeddings.



    1.2. Add position information in each word vector



The key advantage of recurrent neural networks is their knack for understanding relationships between sequences and remembering information. On the other hand, Transformers employ positional encoding to factor in where words are located in a sequence.

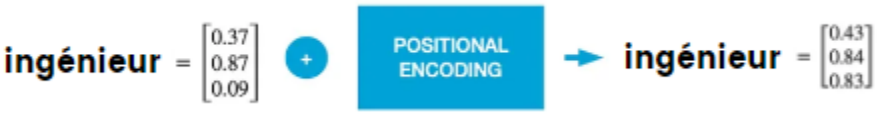
1.3. Apply Multi Head Attention

1.4. Use Feed Forward Network

2. **Decoder**

2.1. Utilize embeddings to transform a French sentence into vectors.

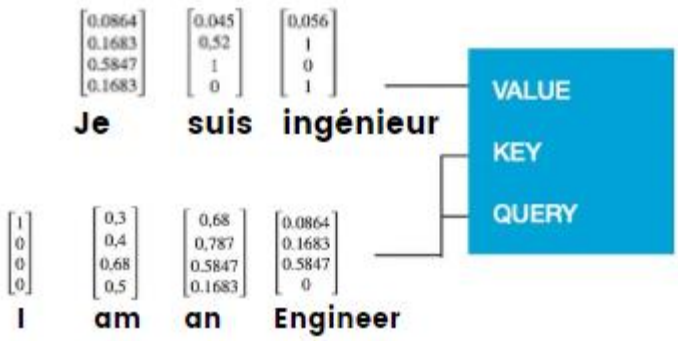
2.2. Add positional details within each word vector.



2.3. Apply Multi Head Attention

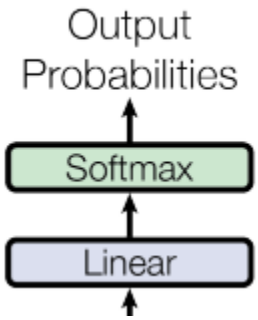
2.4. Apply Feed Forward Network

2.5. Apply Multi-Head Attention to the encoder output.



We can observe that the Transformer combines the encoder's output with the decoder's input. This enables it to discern the relationship between the vectors that encode the English and French sentences.

2.6. Apply Feed Forward Network again.
2.7. Compute the probability for the next word by using linear + softmax block. The decoder returns the highest probability as the next word at the output.



In our case, the next word after **“Je”** is **“suis”**.

Result

The transformer model outperforms all the models on different benchmarks also there was no difference seen between the translation provided by the algorithm and by humans.