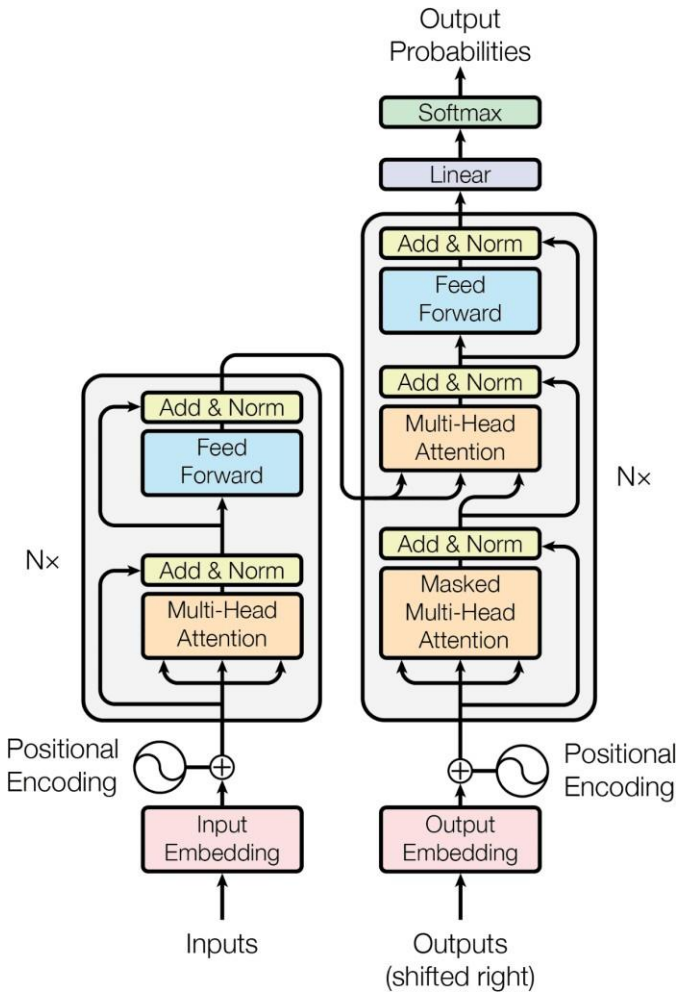
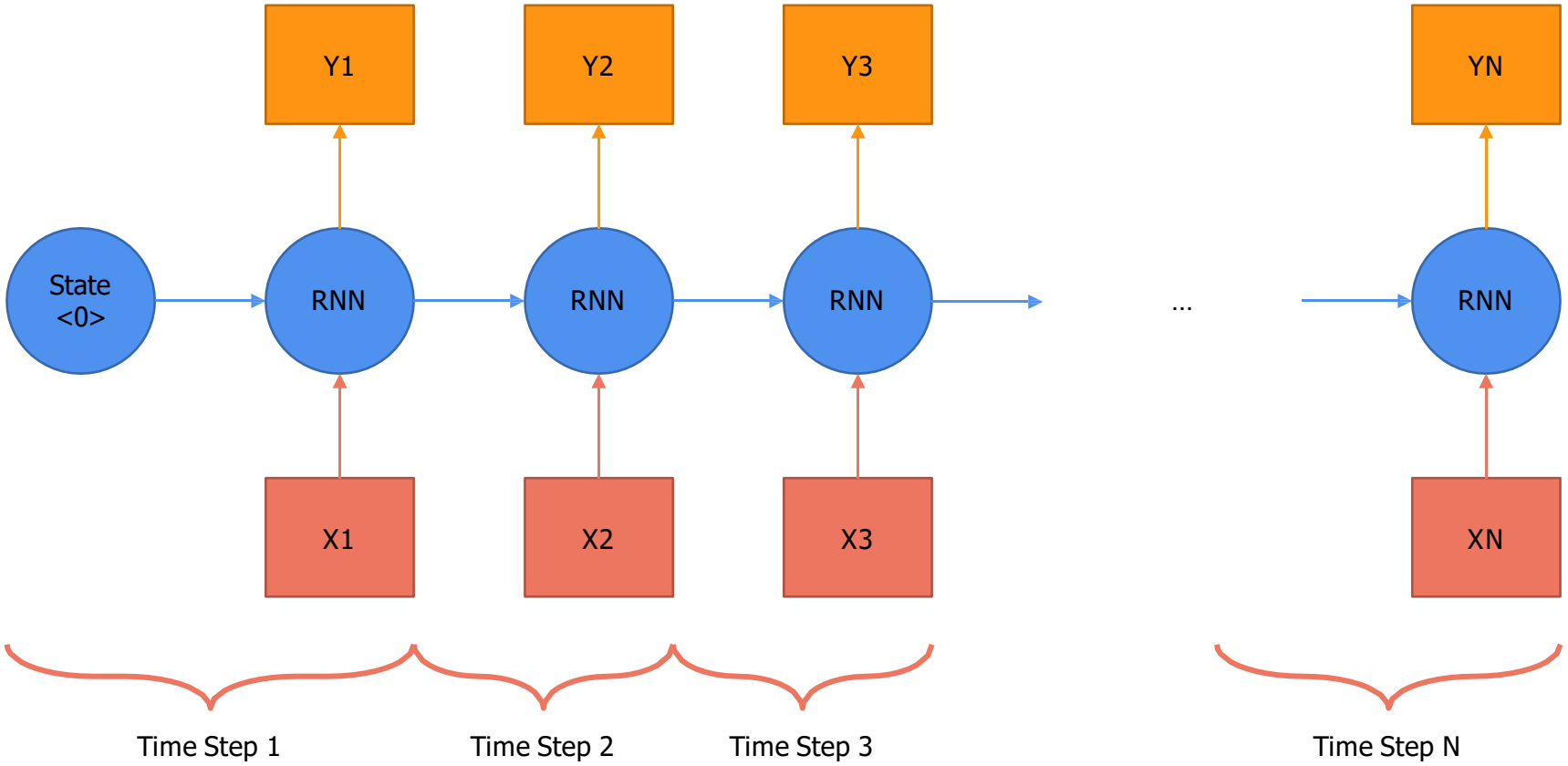


Transformers

Paper: Attention is all you need

Recurrent Neural Networks (RNN)

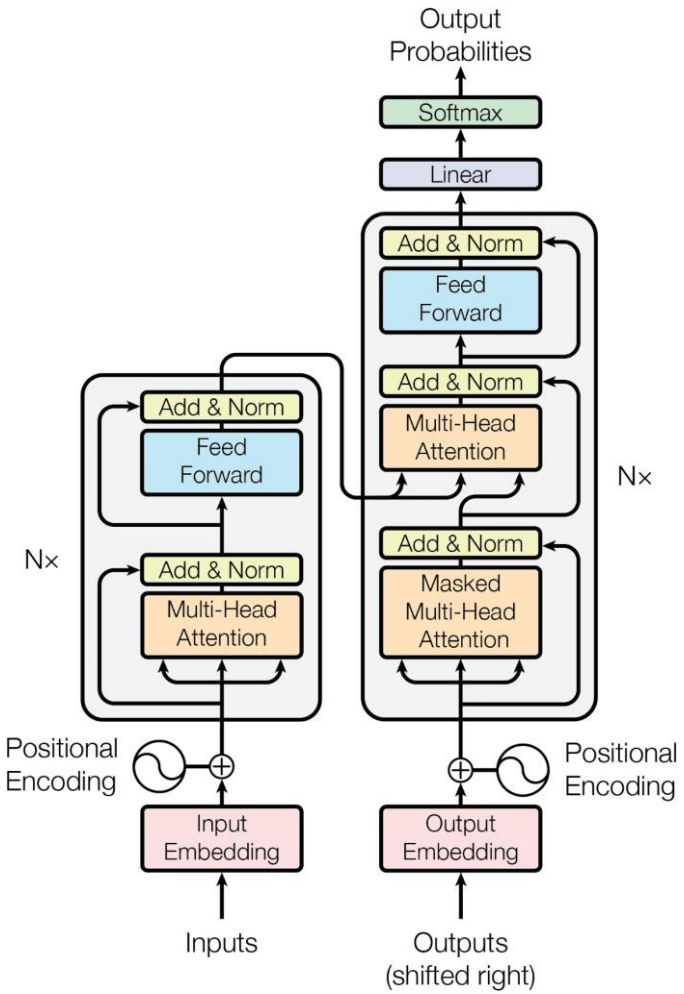


Problems with RNN (among others)

1. Slow computation for long sequences
2. Vanishing or exploding gradients
3. Difficulty in accessing information from long time ago

https://github.com/hkproj/transformer-from-scratch-notes

Transformers

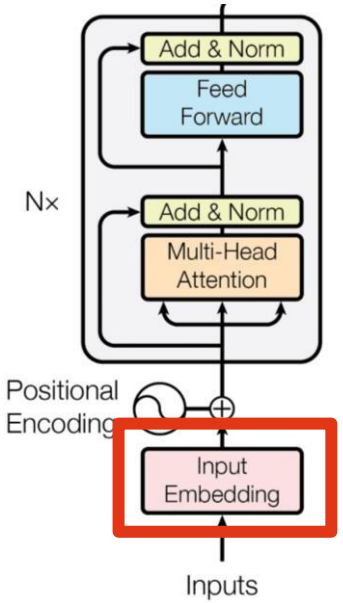


https://www.youtube.com/watch?v=bCz4OMemCcA&t=796s

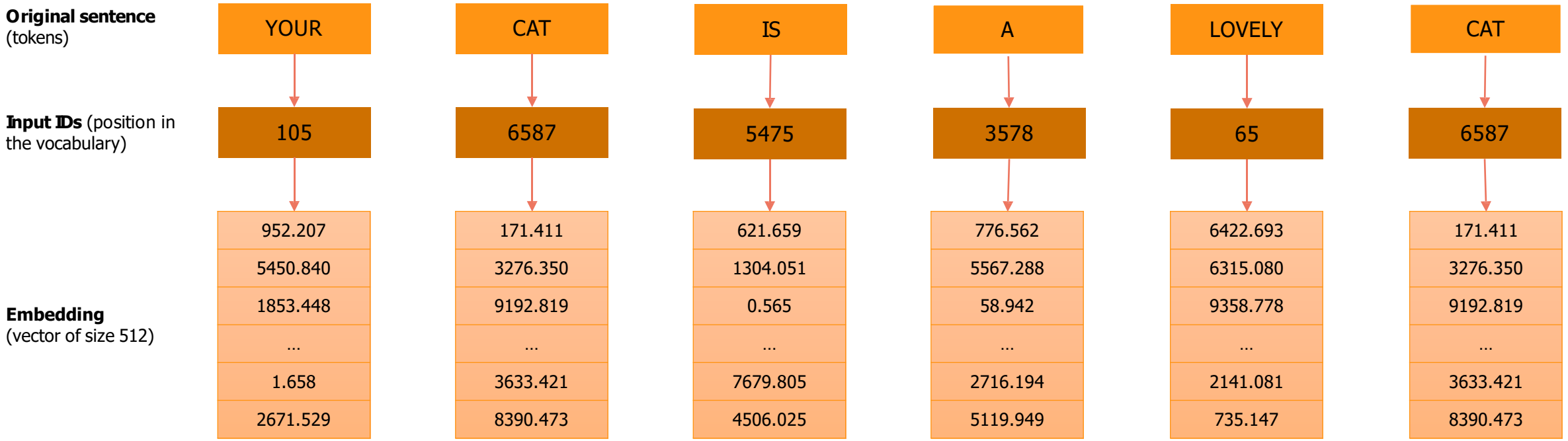
Notations

Input matrix (sequence, d_{model})

Encoder

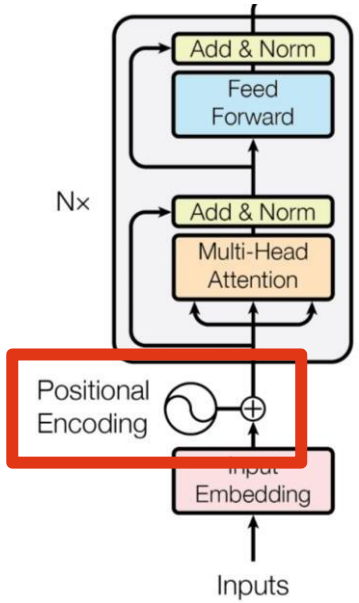

Add & Norm
Feed Forward
Nx
Add & Norm
Multi-Head Attention
Positional Encoding
Input Embedding
Inputs

What is an input embedding?



We define $d_{model} = 512$, which represents the size of the embedding vector of each word

Encoder


Add & Norm
Feed Forward
Nx
Add & Norm
Multi-Head Attention
Positional Encoding
Embedding
Inputs

What is positional encoding?

- We want each word to carry some information about its position in the sentence.
- To inject information about the position or order of elements in a sequence
- We want the model to treat words that appear close to each other as “close” and words that are distant as “distant”.
- We want the positional encoding to represent a pattern that can be learned by the model.

- Why sine cosine?

1. Sine and cosine are periodic functions, so they’ll repeat after some time
2. Constrained values: lets the value b/w a certain range
3. Easier to extrapolate for longer sequences, even if we haven't seen certain lengths, we can interpret them during inference

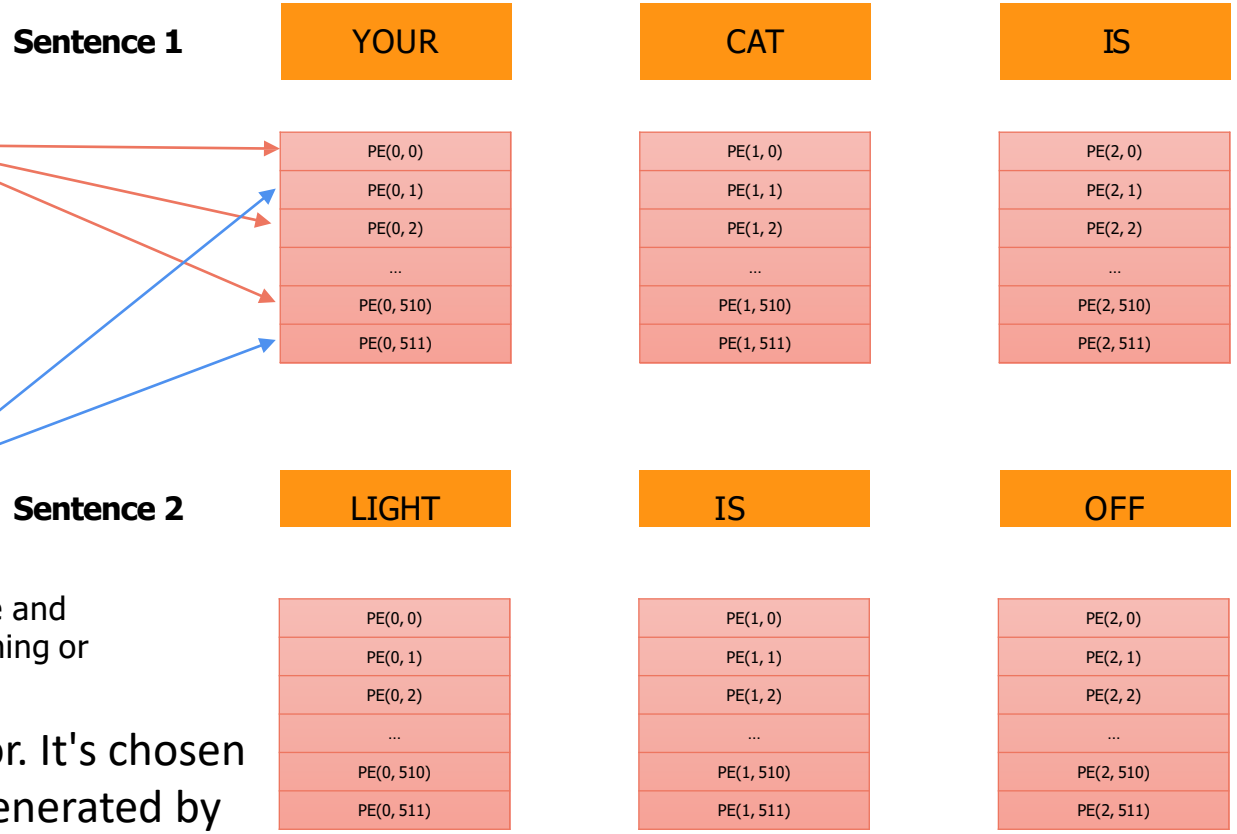
What is positional encoding?

What is positional encoding?

These sinusoidal functions are designed to create unique and distinct positional embeddings for different positions in a sequence, allowing the model to distinguish between tokens based on their position. The formula is fixed and does not adapt to the specific content of the tokens.

$$PE(pos, 2i) = \sin \frac{pos}{10000^{\frac{2i}{d_{model}}}}$$

$$PE(pos, 2i+1) = cos \frac{pos}{10000^{\frac{2i}{d_{model}}}}$$



We only need to compute the positional encodings once and then reuse them for every sentence, no matter if it is training or inference.

The 10000 in the denominator is a scaling factor. It's chosen to be a large number so that the frequencies generated by the exponent are relatively small. This ensures that the positional encodings don't dominate the word embeddings

https://kikaben.com/transformers-positional-encoding/

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

- **i = 0:**
  - PE(pos, 2i) = PE(pos, 0) = sin(pos / 10000^(2*0/d_model)) = sin(pos / 10000^0)
  - PE(pos, 2i+1) = PE(pos, 1) = cos(pos / 10000^(2*0/d_model)) = cos(pos / 10000^0)
- **i = 1:**
  - PE(pos, 2i) = PE(pos, 2) = sin(pos / 10000^(2*1/d_model)) = sin(pos / 10000^(2/d_model))
  - PE(pos, 2i+1) = PE(pos, 3) = cos(pos / 10000^(2*1/d_model)) = cos(pos / 10000^(2/d_model))
- **i = 2:**
  - PE(pos, 2i) = PE(pos, 4) = sin(pos / 10000^(2*2/d_model)) = sin(pos / 10000^(4/d_model))
  - PE(pos, 2i+1) = PE(pos, 5) = cos(pos / 10000^(2*2/d_model)) = cos(pos / 10000^(4/d_model))

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

Therefore, position 0 is a vector with alternatively repeating 0s and 1s.

Position 1 is as follows:

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

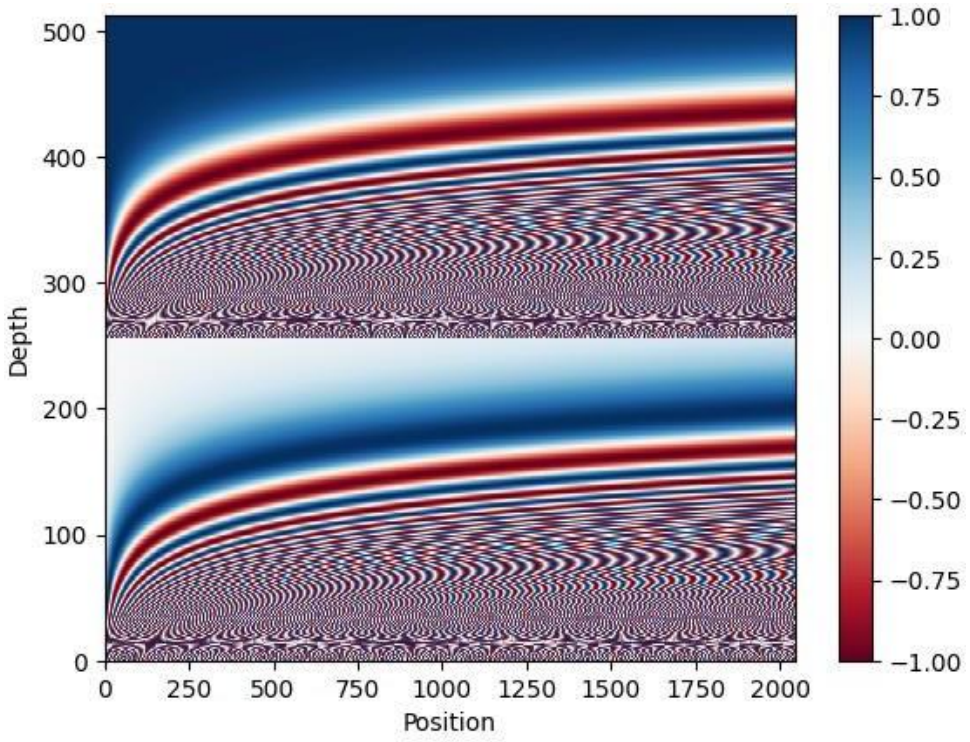
Pre Computation of PE

positional encodings are typically pre-computed. The sine and cosine functions are deterministic, meaning that for a given position and dimension, they will always produce the same value. Therefore, you only need to calculate them once for each possible position up to a maximum sequence length. These pre-computed values are then stored and added to the word embeddings as needed. You don't need to recalculate them for every new word, unless the word is at a position beyond your pre-computed range.

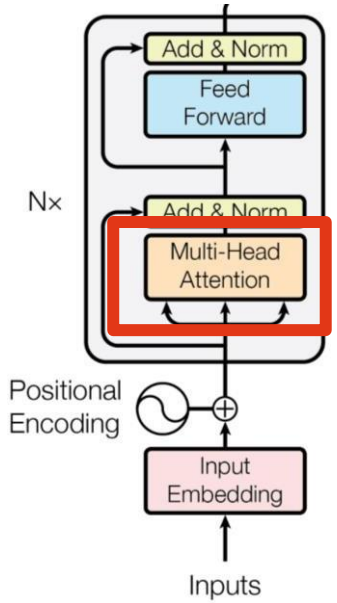
This pre-computation makes positional encoding very efficient. It's a one-time cost that significantly improves the self-attention mechanism's ability to handle word order without adding significant computational overhead during training or inference.

Why trigonometric functions?

Trigonometric functions like **cos** and **sin** naturally represent a pattern that the model can recognize as continuous, so relative positions are easier to see for the model. By watching the plot of these functions, we can also see a regular pattern, so we can hypothesize that the model will see it too.



Encoder


Add & Norm
Feed Forward
Nx
Add & Norm
Multi-Head Attention
Positional Encoding
Input Embedding
Inputs

What is Self-Attention?

Self-Attention allows the model to relate words to each other.

In this simple case we consider the sequence length **seq** = 6 and $\mathbf{d_{model}} = \mathbf{d_k} = 512$.

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

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

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

	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

(6, 6)

* all values are random.

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

How to compute Self-Attention?

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

	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)

X

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.

Self-Attention in detail

- Self-Attention is permutation invariant since it does not cater the order of words, and treats each word independently
  (if you do not consider the contribution of the positional encoding)

Order does not affect the output of the self attention

If u rearrange the inputs, attention values will remain the same

- Self-Attention requires no parameters. Up to now the interaction between words has been driven by their embedding and the positional encodings. This will change later.
- We expect values along the diagonal to be the highest.
- If we don't want some positions to interact, we can always set their values to $-\infty$ before applying the *softmax* in this matrix and the model will not learn those interactions. We will use this in the decoder.

	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

Self-Attention in detail

Permutation invariance is a problem because word order is crucial for understanding language. The sentence "The cat sat" is very different from "Sat cat the." Standard self-attention doesn't inherently understand this.

How is this solved?

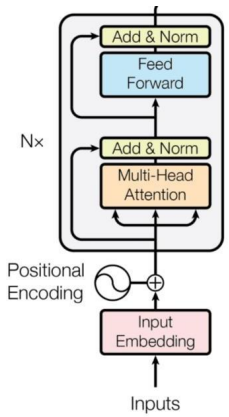
Positional encodings are added to the input embeddings to address this limitation. These encodings provide information about the absolute position of each word, allowing the self-attention mechanism to consider word order when calculating attention weights. With positional encodings, the attention weights for the permuted sentence would be *different* from the original sentence, reflecting the change in meaning due to the change in word order.

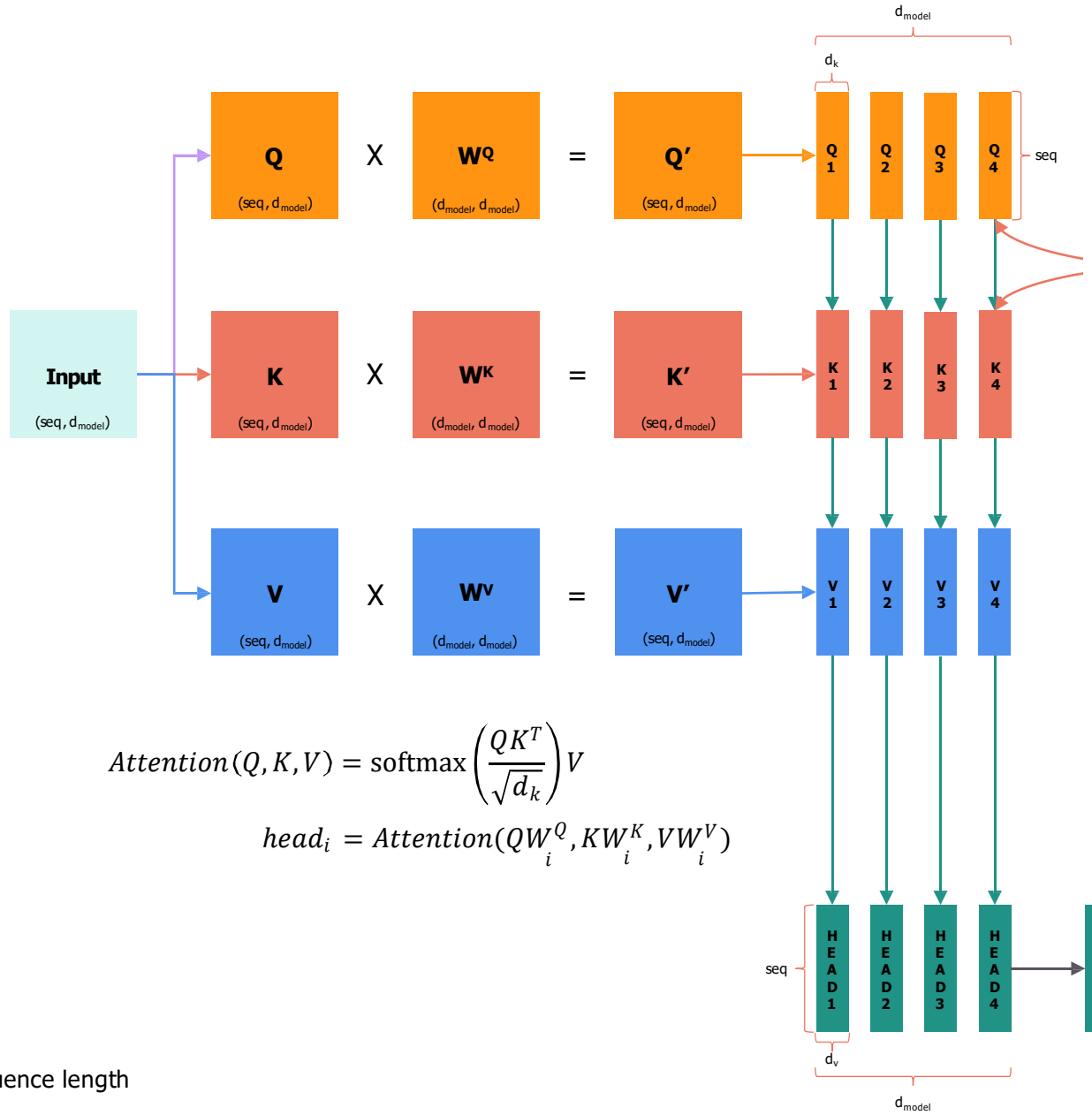
Multi-head Attention

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

$$MultiHead(Q, K, V) = Concat(head_1 \ \ldots head_h)W^O$$

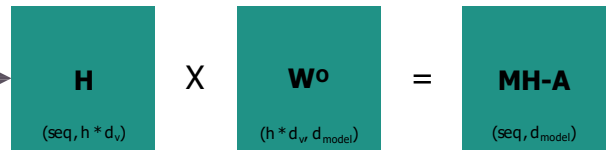
$$head_i = Attention(QW^Q, KW_i^K, VW^V)$$





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

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$



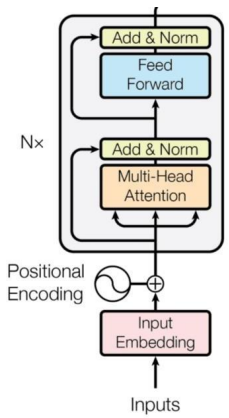
$$MultiHead(Q,K,V) = Concat(head_1 \ \ldots head_h)W^O$$

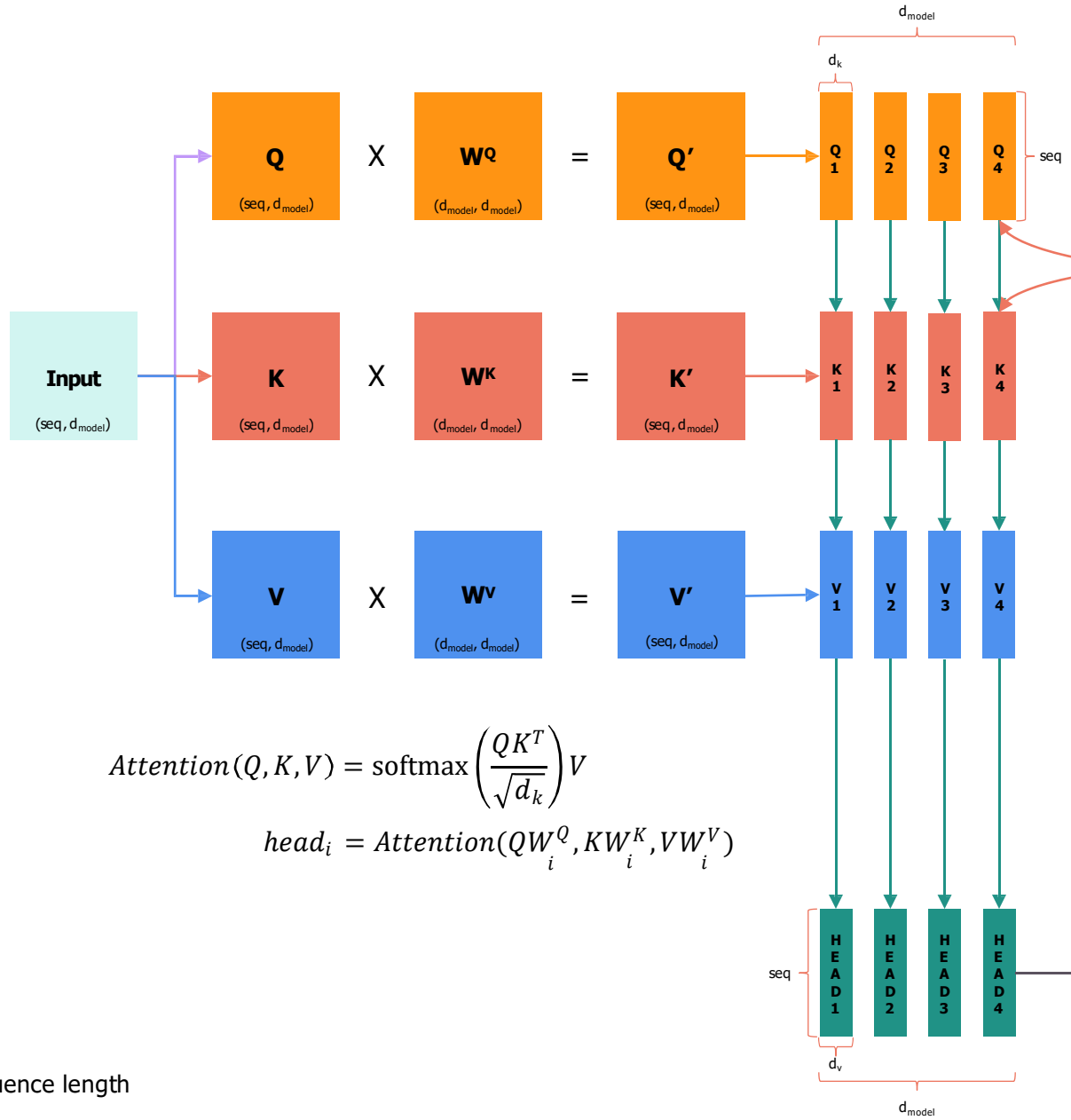
seq	= sequence length
d_{model}	= size of the embedding vector
h	= number of heads
$d_k = d_v$	= d_{model} / h

query: The "query" is like a word looking for other words to pay attention to.

key: The "key" is like a word being looked at by other words.

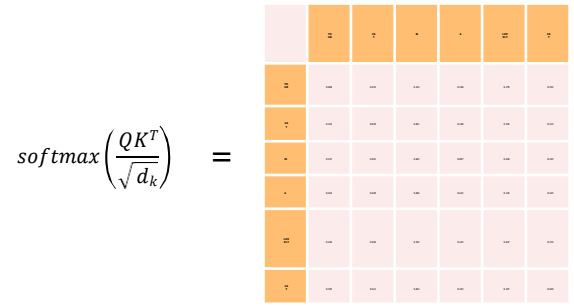
value: the "value" is like the information or meaning of a word

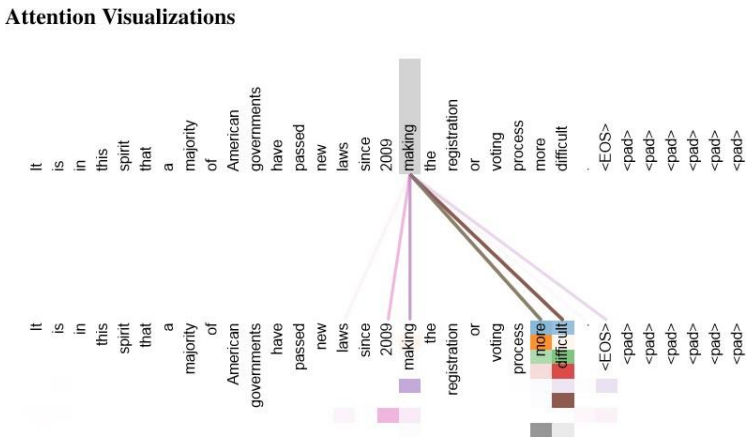




$$Attention(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$





$$MultiHead(Q,K,V) = Concat(head_1 \ \ldots head_h)W^O$$

seq = sequence length

d_{model} = size of the embedding vector

h = number of heads

$d_k = d_v$ = d_{model} / h

The purpose of using multiple heads is to allow the model to focus on different aspects or patterns in the input sequence in parallel.

1.Attention: Attention mechanism is used in deep learning models to selectively focus on important parts of input data while ignoring irrelevant parts. In the context of natural language processing (NLP), attention mechanism has been used in sequence-to-sequence models to align relevant parts of the input sentence with each word in the output sentence.

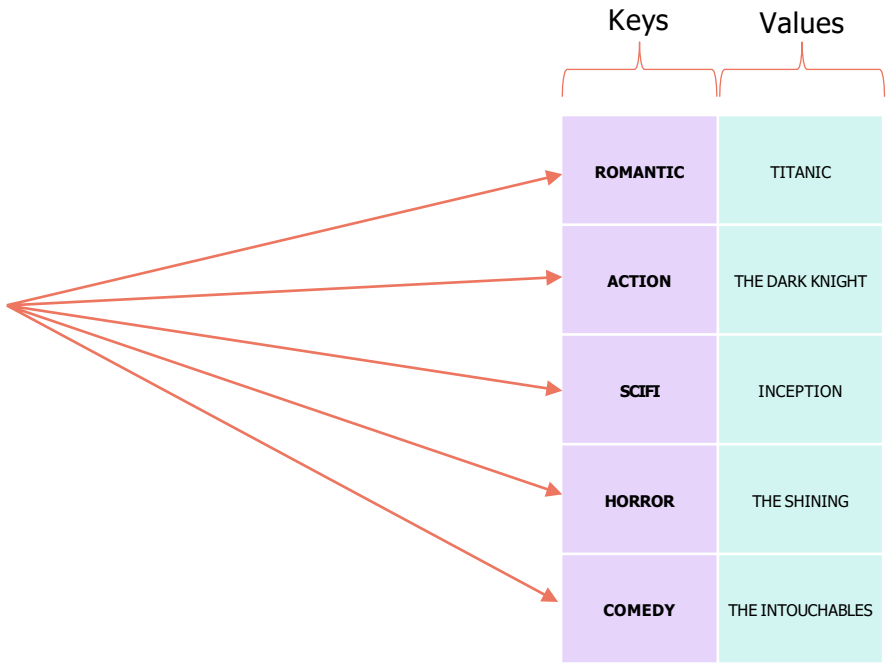
2.Self-attention: Self-attention is a special case of attention mechanism where the input sequence is the same as the output sequence. In other words, each word in the input sequence attends to all other words in the same sequence to generate a representation of itself. Self-attention is used in transformers to process input sequences by generating a set of weighted vectors for each word in the sequence.

3.Multi-head attention: Multi-head attention is an extension of self-attention where the attention mechanism is applied multiple times in parallel with different weight matrices to capture different aspects of the input sequence. This allows the model to attend to multiple parts of the input sequence at the same time and learn more complex representations. Multi-head attention is used in transformers to process input sequences and generate the context vectors that are used to generate the output sequence

Why query, keys and values?

The Internet says that these terms come from the database terminology or the Python-like dictionaries.

Query = "**love**"



*this could be a Python dictionary or a database table.

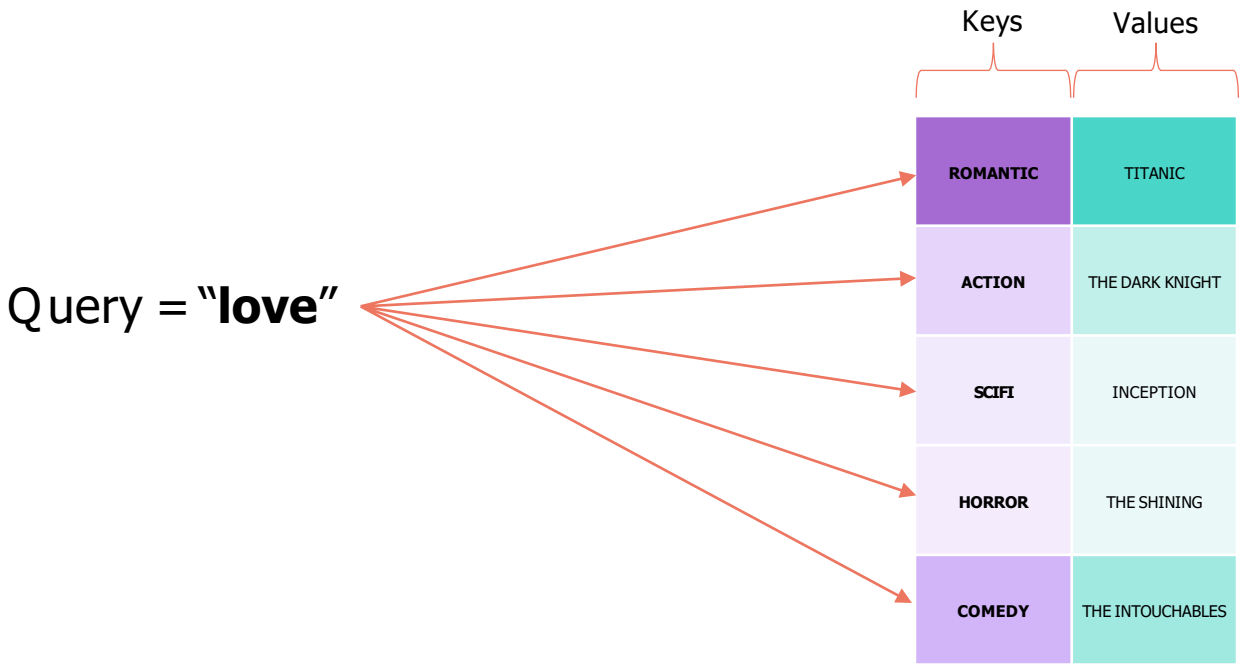
query: The "query" is like a word looking for other words to pay attention to.

key: The "key" is like a word being looked at by other words.

value: the "value" is like the information or meaning of a word

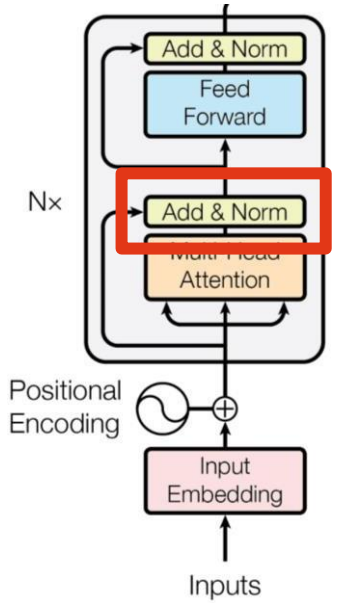
Why query, keys and values?

The Internet says that these terms come from the database terminology or the Python-like dictionaries.



*this could be a Python dictionary or a database table.

Encoder


Add & Norm
Feed Forward
Nx
Add & Norm
Attention
Positional Encoding
Input Embedding
Inputs

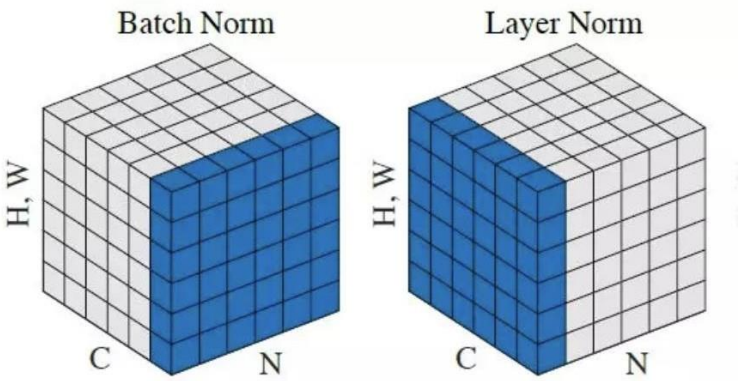
What is layer normalization?

Layer normalization ensures that each part of the sequence (each word or token) is normalized independently, providing stability and ensuring the model generalizes well across the entire sequence

Batch of 3 items

ITEM 1	ITEM 2	ITEM 3
50.147	1242.223	9.370
3314.825	688.123	4606.674
...	...	...
...	...	...
8463.361	434.944	944.705
8.021	149.442	21189.444
μ_1	μ_2	μ_3
σ_1^2	σ_2^2	σ_3^2

- **Batch normalization** looks at all the data samples in the batch and normalizes for each feature.
- **Layer normalization** normalizes the values for each instance, taking into account each instance separately



$$\hat{x}_j = \frac{x_j - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

We also introduce two parameters, usually called **gamma** (multiplicative) and **beta** (additive) that introduce some fluctuations in the data, because maybe having all values between 0 and 1 may be too restrictive for the network. The network will learn to tune these two parameters to introduce fluctuations when necessary.