[redacted]

[redacted]

[redacted]

 What is Self-Attention?

Self-attention is the core mechanism in the Transformer architecture (Vaswani et al., 2017) that allows the model to weigh the importance of different words in a sequence when encoding a particular word.

In simpler terms:

For example, in the sentence: *"The cat sat on the mat because it was tired,"* the word *"it"* should pay more attention to *"cat"* (not *"mat"*) to understand what *"it"* refers to.

It helps the model figure out which words to pay more attention to when processing a specific word.

Lets make it more simple, imagine you're reading a sentence and trying to understand what each word means in context.

For example:

> *“The cat sat on the mat because it was tired.”*

When you read *“it”*, your brain automatically connects it to *“cat”* — you pay attention to the right part of the sentence to understand the meaning.

That’s basically what self-attention does inside a Transformer:
 it helps the model figure out which other words are important for understanding each word.

How Does Self-Attention Work?

For each word in the input sequence:

1. Compute three vectors:
    - Query (Q)
    - Key (K)
    - Value (V)

   These are obtained by multiplying the word embedding with learned weight matrices.
2. Calculate attention scores:
    - Compute similarity between Query of the current word and Key of all words.
    - This gives a score indicating how much focus to put on each word.
3. Normalize scores:
    - Apply softmax to turn scores into probabilities (attention weights).
4. Compute weighted sum:
    - Multiply attention weights with Value vectors of all words and sum them up.
    - This gives the final representation of the current word, enriched by its context.

Lets simple it, for every word (or token) in a sentence:

1. Look around → Each word “looks” at the other words in the sentence.
2. Decide what’s important → The model figures out how strongly each word should be connected to the others.
3. Mix the information → Each word updates its meaning by blending in information from the words it paid attention to.

So, when the Transformer processes *“it”*, it “realizes” that *“cat”* is important, not *“mat”*.

Formula (Scaled Dot-Product Attention)







- Q: Query matrix
- K: Key matrix
- V: Value matrix
- d_k: Dimension of key (used for scaling)

Benefits of Self-Attention

Captures long-range dependencies (no matter how far words are)
Enables parallel processing (unlike RNNs)
It figures out meaning dynamically depending on the sentence.

Visualization (conceptual)

```
vbnetCopyEditInput:   The   cat   sat   on   the   mat
Weights: 0.1   0.5   0.2   0.05  0.1  0.05
```

For the word *“cat”*, it may pay more attention to itself and nearby words, while *“sat”* may look at both *“cat”* and *“mat”*.

Summary in one line

Self-attention helps each word in a sentence understand its meaning by looking at — and learning from — all the other words.

Let’s focus just on Q (Query):

What is Query (Q)?

The Query vector is like the “question” that one word asks about how much it should care about other words in the sentence.

For each word:

- The model generates a Query vector.
- This vector is compared against the Key vectors of all words (including itself).
- The comparison gives attention scores that tell which words matter most when understanding this word.

Simple example

Sentence:

> *“The cat sat on the mat.”*

Let’s say we’re working on the word “cat”.

- We create Query(cat) → this is like asking:
  “Who in this sentence is important for me to understand my meaning?”

We then compare Query(cat) with Key(The), Key(cat), Key(sat), Key(on), Key(the), Key(mat). This gives scores like:

- The → 0.1
- cat → 0.4
- sat → 0.3
- on → 0.1

- the → 0.05
- mat → 0.05

So, “cat” pays most attention to itself and a little to “sat.”

How is Query created?

For each word: Query=Embedding×W^Q

- Embedding → the vector representing the word.
- W^Q → a learned weight matrix.
- Result → Query vector.

Let’s focus just on K (Key):

What is Key (K)?

The Key vector is like a label or tag that tells other words:
“Here’s the kind of information I carry.”

In the self-attention process, every word has:

- A Query → *what it wants to know*
- A Key → *what it offers to others*
- A Value → *the actual content it provides*

Example in action

Sentence:

> *“The cat sat on the mat.”*

Let’s say the model is working on “cat”:

- It has a Query(cat) → asking: *“Who is important for me?”*

Then, it compares Query(cat) to the Keys of all words:

- Key(The)
- Key(cat)
- Key(sat)
- Key(on)
- Key(the)
- Key(mat)

These Keys are like ID cards saying what each word is about.
By comparing Query to Keys, the model decides which words deserve attention.

How is Key created?

For each word: $\text{Key} = \text{Embedding} \times W^K$

- Embedding → vector of the word.
- W^K → learned weight matrix.
- Result → Key vector.

Intuition

Query → What this word is looking for.
Key → What each word offers as a “summary.”
Together → The model measures how well the Query and Key match to decide attention weights.

Let’s focus just on V (Value):

What is Value (V)?

The Value vector carries the actual information that will be passed along once attention is decided.

In simpler words:

- The Query figures out what to look for.
- The Key explains what each word offers.
- The Value provides the actual content that will be mixed into the final output.

So, after the model compares Queries and Keys and decides which words to focus on,

 it collects the Values (weighted by the attention scores) to build a richer meaning for each word.

Example

Sentence:

> *“The cat sat on the mat.”*

Let’s say we’re focusing on “cat”:

- We calculate:
  - Query(cat)
  - Compare Query(cat) with Key(The), Key(cat), Key(sat), Key(on), Key(mat)
  - Get attention scores → say, 0.1, 0.5, 0.3, 0.05, 0.05

Finally:

- We take the Values from each word → Value(The), Value(cat), Value(sat), Value(on), Value(mat)
- Multiply them by the attention scores
- Sum them → this gives the updated, context-aware meaning of “cat.”

How is Value created?

For each word: $\text{Value} = \text{Embedding} \times W^{V}$

- Embedding → vector of the word.
- W^V → learned weight matrix.
- Result → Value vector.

Intuition

Query → “What am I looking for?”
Key → “What can I offer to others?”
Value → “Here’s my full info if you decide I matter.”

List of Popular Self-Attention Models

Model	D (Embedding Size)	# Heads	Head Dim (D / heads)	Notes
GPT-2 Small	768	12	64	117M parameters
GPT-2 Medium	1024	16	64	345M parameters
GPT-2 Large	1280	20	64	762M parameters
GPT-2 XL	1600	25	64	1.5B parameters
GPT-3 (175B)	12288	96	128	Huge, closed weights
GPT-4 (est.)	~12288–32768	?	?	Specs not fully disclosed
BERT Base	768	12	64	110M parameters

Model	D (Embedding Size)	# Heads	Head Dim (D / heads)	Notes
BERT Large	1024	16	64	340M parameters
DistilBERT	768	12	64	Smaller BERT
RoBERTa Base	768	12	64	BERT optimized
RoBERTa Large	1024	16	64	More training data
T5 Small	512	8	64	Text-to-text model
T5 Base	768	12	64	Encoder-decoder model
T5 Large	1024	16	64	
XLNet Base	768	12	64	Permutation-based
XLNet Large	1024	16	64	
ALBERT Base	768	12	64	Shared weights
ALBERT Large	1024	16	64	
TinyBERT	312	12	26	For mobile devices
MobileBERT	512	4	128	Highly efficient
Longformer	768–1024	12–16	64	Long sequence support
ViT-B/16 (Vision Transformer)	768	12	64	Used for image patches
ViT-L/32	1024	16	64	Larger image model

Observations

- Most models keep head size at 64 for stability and optimization.
- `D` is typically divisible by `# heads` (to avoid errors in attention reshaping).
- Bigger models (GPT-3, GPT-4) scale up both `D` and `# heads` heavily.
- Lightweight models (MobileBERT, TinyBERT) trade off `D` and `# heads` for speed and memory.

Q. What is learned weight matrix ?

Related posts:

1. Transformer Architecture in LLM
2. Input Embedding in Transformers
3. Positional Encoding in Transformers
4. Multi-Head Attention in Transformers
5. Why 512 Dimensions in Transformer Model Architecture