What Problem Does Self-Attention Solve?

1. The Core Problem (Why Attention Exists)

Consider this sentence:

> **“The animal didn’t cross the street because it was tired.”**

Question:

What does **“it”** refer to?

- The street? (No)
- The animal? (yes)

To answer this, the model must:

- Remember earlier words
- Compare meanings across the sentence
- Decide **which word matters most**

Old models struggled:

- **RNNs** forget long-distance info
- **CNNs** have limited receptive fields

Self-Attention solves this:

> Every word can **directly look at every other word** and decide **how important they are**

2. What Self-Attention *Really* Is

Think of **each word as asking a question**:

> “Which other words in this sentence are relevant to me?”

Then it:

1. Scores all other words
2. Converts scores into probabilities
3. Takes a **weighted average** of their information

That's it.

Self-attention = smart weighted averaging

3. Mental Model (Very Important)

For each token:

```
Token i →
    looks at all tokens →
        assigns importance →
            blends their information →
                produces a new representation
```

This happens **for every token in parallel**.

4. Tiny Toy Example (Conceptual)

Sentence:

```
["I", "love", "deep", "learning"]
```

Imagine:

- `"love"` strongly attends to `"I"`
- `"learning"` strongly attends to `"deep"`

Each word updates its meaning using context.

5. First Visualization (Conceptual Attention Map)

Think of an attention matrix:

	I	love	deep	learning
I	✔			
love	✔	✔		
deep			✔	✔
learning			✔	✔

Each row =
"How much this word cares about other words"

6. Mini NumPy Experiment

Let's simulate **pure similarity-based attention**.

Step 1: Fake embeddings

```
import numpy as np

# 4 tokens, embedding size = 3
X = np.array([
    [1.0, 0.0, 1.0],   # I
    [1.0, 1.0, 0.0],   # love
    [0.0, 1.0, 1.0],   # deep
    [0.0, 0.0, 1.0],   # learning
])
```

Step 2: Similarity (dot product)

```
scores = X @ X.T
print(scores)
```

1This tells us:

> "How similar is each word to every other word?"

Step 3: Softmax (importance)

```
def softmax(x):
    exp_x = np.exp(x - np.max(x, axis=-1, keepdims=True))
    return exp_x / exp_x.sum(axis=-1, keepdims=True)

weights = softmax(scores)
print(weights)
```

Now:

- Each row sums to **1**
- These are **attention weights**

Step 4: Contextualized output

```
context = weights @ X
print(context)
```

Each row is now:

Original word + information blended from other words

This is self-attention in its simplest form

7. Key Takeaways

- Self-attention lets **every token talk to every other token**
- It computes **importance scores**
- Then performs **weighted averaging**
- No sequence order yet
- No learning yet
- But the *core idea is already here*

8. Questions and their Answers

1. What does each row of the attention matrix represent?

Each row answers this question:

"For this word, how much should I pay attention to every other word?"

Easy way to think:

- Pick **one word**
- Look across the row
- The numbers tell you **how important other words are to it**

Example:

If the row is for the word **"love"**:

```
[I   love   deep   learning]
0.4  0.3    0.2      0.1
```

That means:

- "love" cares a lot about **"I"**
- a bit about **itself**
- less about the others

One row = one word's attention to all words

2. Why do we apply softmax?

Because raw scores are **not meaningful yet**.

Problems with raw scores:

- They can be negative
- They don't sum to anything useful
- You can't say "how important" one is compared to another

What softmax does:

- Converts scores into **probabilities**
- Makes all values:
    - Positive
    - Add up to **1**

Easy analogy:

Softmax turns **scores** into **attention percentages**

Like:

> "I care 50% about this word, 30% about that one…"

Softmax makes attention **comparable and interpretable**

3. What does `weights @ X` actually do?

This is the **most important step**.

Simple explanation:

It creates a **new version of each word** by mixing information from all words.

Think of it like this:

- `weights` = how much to listen to each word
- `X` = the information each word has
- `weights @ X` = **listen + combine**

Example:

If a word gives:

- 70% attention to "deep"
- 30% attention to "learning"

Then its new meaning becomes:

> "Mostly 'deep', with some 'learning' mixed in"

Real-world analogy:

Like making tea:

- `weights` = how much of each ingredient
- `X` = ingredients
- result = **final flavor**

This step creates **context-aware word meanings**