Simplified Self-Attention (Formalized, No Trainable Parameters)

This phase answers:

> "What exactly is self-attention, mathematically and computationally?"

1. Input Representation (Start Point)

Assume we already have embeddings.

Input tensor

```
X ∈ ℝ^(n × d)
```

- n = number of tokens (sequence length)
- d = embedding dimension

Example:

```
X.shape = (n, d)
```

Each row = one token embedding.

2. Similarity = Attention Scores

Key idea:

Tokens that are **similar** should influence each other more.

We measure similarity using a **dot product**.

Formula

$$S = XX^T$$

Shape

```
(n, d) @ (d, n) → (n, n)
```

Each element:

$$S_{ij} = x_i \cdot x_j$$

Meaning:

> "How much token *i* relates to token *j*"

3. Why Dot Product?

Because it:

- Is fast
- Works well in high dimensions
- Measures alignment between vectors
- Is differentiable

(Transformers are optimized for matrix multiplication.)

4. Normalize Scores → Attention Weights

Raw scores are:

- Unbounded
- Not comparable across rows

So we apply **softmax row-wise**.

Formula

$$A = \text{softmax}(S)$$

Properties

- Each row sums to **1**
- Values are probabilities
- Represents **importance**

5. Weighted Sum = Context Vectors

Now each token builds its new representation:

Formula

$$Y = AX$$

Shape

```
(n, n) @ (n, d) → (n, d)
```

Each output vector is:

> A mixture of all token embeddings, weighted by relevance

6. Full Simplified Self-Attention Formula

$$\boxed{\text{SelfAttention}(X) = \text{softmax}(XX^T)X}$$

This is the **purest form** of self-attention.

7. Complete NumPy Implementation

```python
import numpy as np

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)

def self_attention_simple(X):
    # X: (n, d)
    scores = X @ X.T        # (n, n)
    weights = softmax(scores)  # (n, n)
    output = weights @ X    # (n, d)
    return output, weights
```

8. Shape Walkthrough (Interview-Level Clarity)

Step	Operation	Shape
Input	`X`	`(n, d)`
Scores	`X @ X.T`	`(n, n)`

Weights	`softmax(scores)`	`(n, n)`
Output	`weights @ X`	`(n, d)`

If you can explain this table, you **understand self-attention**.

9. Visualization (How to Think About It)

Imagine the attention matrix as a heatmap:

- Rows → current token
- Columns → tokens being attended to
- Bright cell → strong attention

This matrix is **dynamic**, changes with input.

10. Limitations of This Version (Very Important)

This simplified attention:

No Treats similarity as fixed
No Cannot learn what "important" means
No No notion of roles (query vs key vs value)
No No sequence direction
No No parameter learning

This is why we need **trainable projections**.

11. Questions and their Answers

1. Why is the attention matrix `(n × n)`?

Because **every token attends to every other token**.

- We have `n` tokens in the sequence.
- For **each token**, we compute its similarity with **all tokens** (including itself).

Breakdown:

- Rows → **query tokens** (the token doing the attending)
- Columns → **key tokens** (the tokens being attended to)

So:

```
n tokens × n tokens = n × n matrix
```

Each cell `(i, j)` answers:

> "How much does token *i* care about token *j*?"

That's why the attention matrix must be **square**.

2. What does one row of attention weights represent?

One row represents **the attention distribution for a single token**.

More precisely:

- A row corresponds to **one token as the focus**
- The values in that row show **how much importance it assigns to every token in the sequence**

Because we apply **softmax row-wise**:

- All values are positive
- The row sums to **1**

So one row answers:

> "How should this token combine information from all tokens?"

It's a **probability distribution over tokens**.

3. Why does the output shape match the input shape?

Because self-attention **updates token representations**, it does not change:

- The number of tokens
- The embedding dimension

What happens conceptually:

- Each token becomes a **context-aware version of itself**
- But it's still one vector per token
- And still of the same size

Mathematically:

```
(n × n) @ (n × d) → (n × d)
```

So:

- Same number of tokens (n)
- Same embedding size (d)
- But richer information

This makes self-attention:

- Easy to stack
- Compatible with residual connections
- Perfect for deep transformer layers

One-Sentence Summary

> Self-attention keeps the input shape unchanged because it **refines each token using global context**, rather than creating or removing tokens.