Self-Attention Mechanism - Simplified

```
import torch
import numpy as np

#with the next line of code, we are telling pytorch that whenever it
generates random number, to do in the same way every time we run this
code.
#This makes the code repeatable and predictable, which is helpful for
debugging or comparing results.

torch.manual_seed(42)
```

```
<torch._C.Generator at 0x2670a9935b0>
```

Step 1: Define Input Tokens

- In the code below, we are representing words as numbers - embedding.
- Each word in the sentence is turned into a **vector** (list of numbers) that describes its meaning (context) in a way that computer can understand.
- Each of these is a 3D vector. Real LLMs use much higher dimensions, but for simplicity, we are keeping it small.

```
# Example: "The cat ran"
inputs = torch.tensor([
    [0.2, 0.1, 0.5],  # "The"
    [0.9, 0.1, 0.3],  # "cat"
    [0.4, 0.8, 0.2],  # "ran"
])
print("Input vectors:", inputs)
```

```
Input vectors: tensor([[0.2000, 0.1000, 0.5000],
        [0.9000, 0.1000, 0.3000],
        [0.4000, 0.8000, 0.2000]])
```

Step 2: Compute Attention Scores

- We measure similarity between words using dot products.
- We are using the word "ran" as the focus word - the one we want to understand better using self-attention.
- input[2] means give me the third word vector.

```
# We call this a query because in attention , the word is asking:
"Hey, which of you(words) are important for me"
query = inputs[2]  # focus on "ran"

# We are creating an empty box that can store 3 numbers - one for each
word in our sentence.
```

```
attn_scores = torch.empty(inputs.shape[0])

# This loop goes through each word vector and compares it with query
vector using dot product.
# Dot product tells us how similar the two vectors are. The higher the
number, more related are the words to "ran"
for i, x_i in enumerate(inputs):
    attn_scores[i] = torch.dot(query, x_i)

print("Attention Scores:", attn_scores)
```

```
Attention Scores: tensor([0.2600, 0.5000, 0.8400])
```

The above output would mean

- "The" - 0.2600 (not so important to "ran")
- "cat" - 0.5000 (somewhat important)
- "ran" - 0.8400 (very important to itself)

Step 3: Normalize Scores using Softmax

Turn raw scores into probabilities that add up to 1.

```
# Below line converts raw attention scores into probabilities -
attention weights
# Weights indicate how much focus each word would get.
# Using softmax makes them all positive and they add up to 1.

attn_weights = torch.softmax(attn_scores, dim=0)
print("Normalized Attention Weights:", attn_weights

# checks if the weights add up to 1
print("Sum of weights:", attn_weights.sum().item())
```

```
Normalized Attention Weights: tensor([0.2465, 0.3133, 0.4402])
Sum of weights: 0.9999999403953552
```

The above output means that

- "The" gets 26% attention
- "cat" gets 31% attention
- "ran" gets 44% attention

Step 4: Compute Context Vector

Each word contributes to the meaning of 'ran' based on its weight.

```
context_vec = torch.zeros(query.shape)
for i, x_i in enumerate(inputs):
```

```
    context_vec += attn_weights[i] * x_i

print("Context Vector for 'ran':", context_vec)
```

```
Context Vector for 'ran': tensor([0.5074, 0.4081, 0.3053])
```

- Self-attention lets a word like **"ran"** look at **"cat"** and **"the"** to understand the full context.
- We used dot product to get **scores**, softmax to get **weights**, and then blended them to get a **context vector**.
- That's the basic idea behind self-attention used in LLMs like GPT!

```

```