Self-Attention Matrix Operations

Input	→	`(batch_size, seq_len, d_model)`
Q, K, V	→	`(batch_size, seq_len, d_k)`
Scores	→	`(batch_size, seq_len, seq_len)`
Output	→	`(batch_size, seq_len, d_k)`

GPU Parallelization

Matrix multiplication enables **efficient GPU computation**

Memory Requirement

$O(n^2)$ for sequence length **n**

Efficiency Trade-off

Speed vs **Memory**

Input/Output Attention Scores (seq_len × seq_len) Final Output

Computation Example

`seq_len = 5` `d_model = 3` `d_k = 3` `batch_size = 1`

1 Input Embeddings (X)

X: Input sequence with 5 tokens

```
[[0.5, 0.8, 0.3], ← Token 1
[0.2, 0.6, 0.9], ← Token 2
[0.7, 0.4, 0.1], ← Token 3
[0.3, 0.9, 0.5], ← Token 4
[0.6, 0.2, 0.8]] ← Token 5
```

Shape: (5, 3) = (seq_len, d_model)

2 Linear Projections: Q = XW_Q, K = XW_K, V = XW_V

Q (Query Matrix)

```
[[0.6, 0.4, 0.5],
[0.3, 0.7, 0.2],
[0.5, 0.3, 0.6],
[0.4, 0.8, 0.3],
[0.7, 0.2, 0.4]]
```

Shape: (5, 3) = (seq_len, d_k)

K (Key Matrix)

```
[[0.4, 0.6, 0.3],
[0.5, 0.4, 0.7],
[0.3, 0.5, 0.4],
[0.6, 0.3, 0.5],
[0.2, 0.7, 0.6]]
```

Shape: (5, 3) = (seq_len, d_k)

V (Value Matrix)

```
[[0.7, 0.3, 0.5],
[0.4, 0.6, 0.2],
[0.6, 0.4, 0.7],
[0.5, 0.7, 0.3],
[0.3, 0.5, 0.8]]
```

Shape: (5, 3) = (seq_len, d_k)

3 Compute Attention Scores: QK^T / √d_k

$$\text{Scores} = Q \times K^T / \sqrt{d\_k} = Q \times K^T / \sqrt{3} \approx Q \times K^T / 1.732$$

QK^T: Raw attention scores (before scaling)

```
[[0.67, 0.71, 0.58, 0.69, 0.64], ← Token 1 attending to all tokens
[0.52, 0.49, 0.44, 0.51, 0.53], ← Token 2 attending to all tokens
[0.61, 0.67, 0.54, 0.63, 0.56], ← Token 3 attending to all tokens
[0.68, 0.73, 0.59, 0.69, 0.70], ← Token 4 attending to all tokens
[0.50, 0.58, 0.48, 0.55, 0.48]] ← Token 5 attending to all tokens
```

Shape: (5, 5) = (seq_len, seq_len) - Critical $O(n^2)$ memory!

⚠ **Key Point:** This 5×5 attention score matrix is where the $O(n^2)$ memory complexity comes from. For sequence length n=5, we need 25 values. For n=1000, we'd need 1,000,000 values!

4 Apply Softmax (row-wise normalization)

Attention Weights (after softmax)

```
[[0.189, 0.204, 0.176, 0.201, 0.190], ← Sum = 1.0
[0.203, 0.199, 0.192, 0.201, 0.205], ← Sum = 1.0
[0.194, 0.205, 0.188, 0.199, 0.194], ← Sum = 1.0
[0.197, 0.208, 0.185, 0.197, 0.203], ← Sum = 1.0
[0.196, 0.208, 0.189, 0.201, 0.196]] ← Sum = 1.0
```

Shape: (5, 5) - Each row sums to 1.0 (probability distribution)

5 Compute Weighted Sum: Attention_Weights × V

$$\text{Output} = \text{Softmax}(QK^T / \sqrt{d\_k}) \times V$$

Final Output (context-aware representations)

```
[[0.50, 0.50, 0.50], ← Updated Token 1
[0.50, 0.50, 0.50], ← Updated Token 2
[0.50, 0.50, 0.50], ← Updated Token 3
[0.50, 0.50, 0.50], ← Updated Token 4
[0.50, 0.50, 0.50]] ← Updated Token 5
```

Shape: (5, 3) = (seq_len, d_k) - Same as input!

✅ **Result:** Each token now has a context-aware representation that incorporates information from all other tokens through the attention mechanism.

Matrix Shape Summary

Input X: (5, 3) = (seq_len, d_model)

Q, K, V: (5, 3) each = (seq_len, d_k)

Scores QK^T: (5, 5) = (seq_len, seq_len) ← $O(n^2)$ bottleneck!

Output: (5, 3) = (seq_len, d_k)