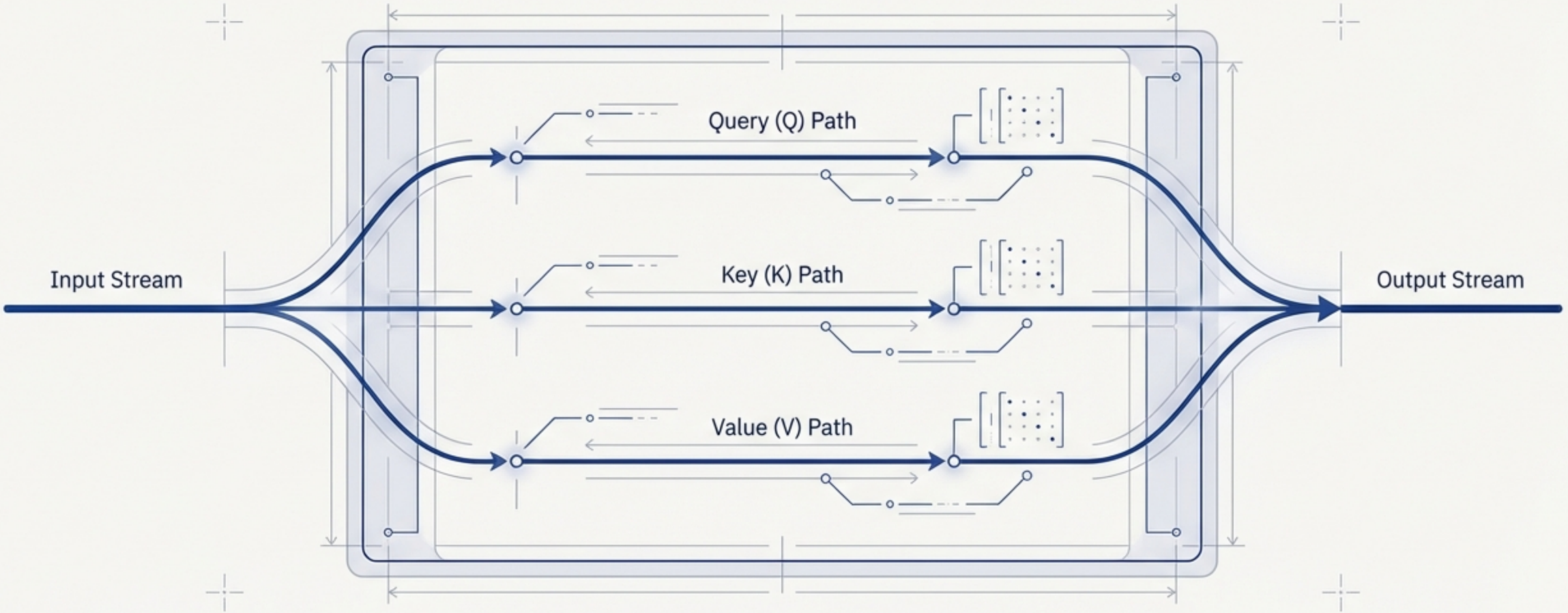
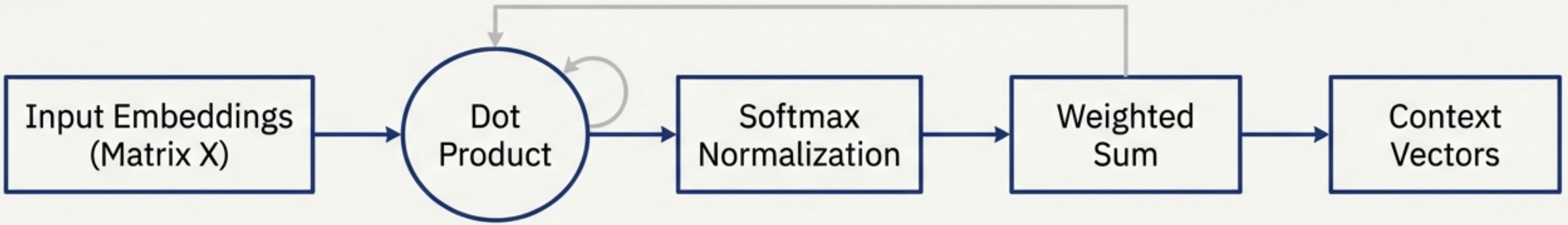
The Engine of Transformers: Building Scaled Dot-Product Attention

A step-by-step construction of the trainable self-attention mechanism used in models like GPT.



Recapping Our Initial Prototype: Self-Attention without Trainable Weights

- In our previous model, we computed context vectors directly from input embeddings.
- The process involved three fixed steps:
  1. **Attention Scores**: Calculated via dot product between a token's embedding (as the 'query') and all other input embeddings.
  2. **Attention Weights**: Scores were normalized using a softmax function to sum to 1.
  3. **Context Vectors**: A weighted sum of the original input embeddings, using the attention weights.



This was a good start, but it lacked a crucial element: **trainability**. The mechanism couldn't learn
This was a good start, but it lacked a crucial element: **trainability**. The mechanism couldn't learn
which aspects of the inputs were important for determining relationships. The model was fixed.

The Upgrade: Projecting Inputs into Specialized Roles with Q, K, V

To make our attention mechanism trainable, we introduce three new components: dedicated weight matrices that the model learns during training.

`W_Q` (Query Matrix)

Transforms input embeddings into **“queries.”** A query represents the current token's perspective, asking a question like, "Given my role, what should I pay attention to?"

`W_K` (Key Matrix)

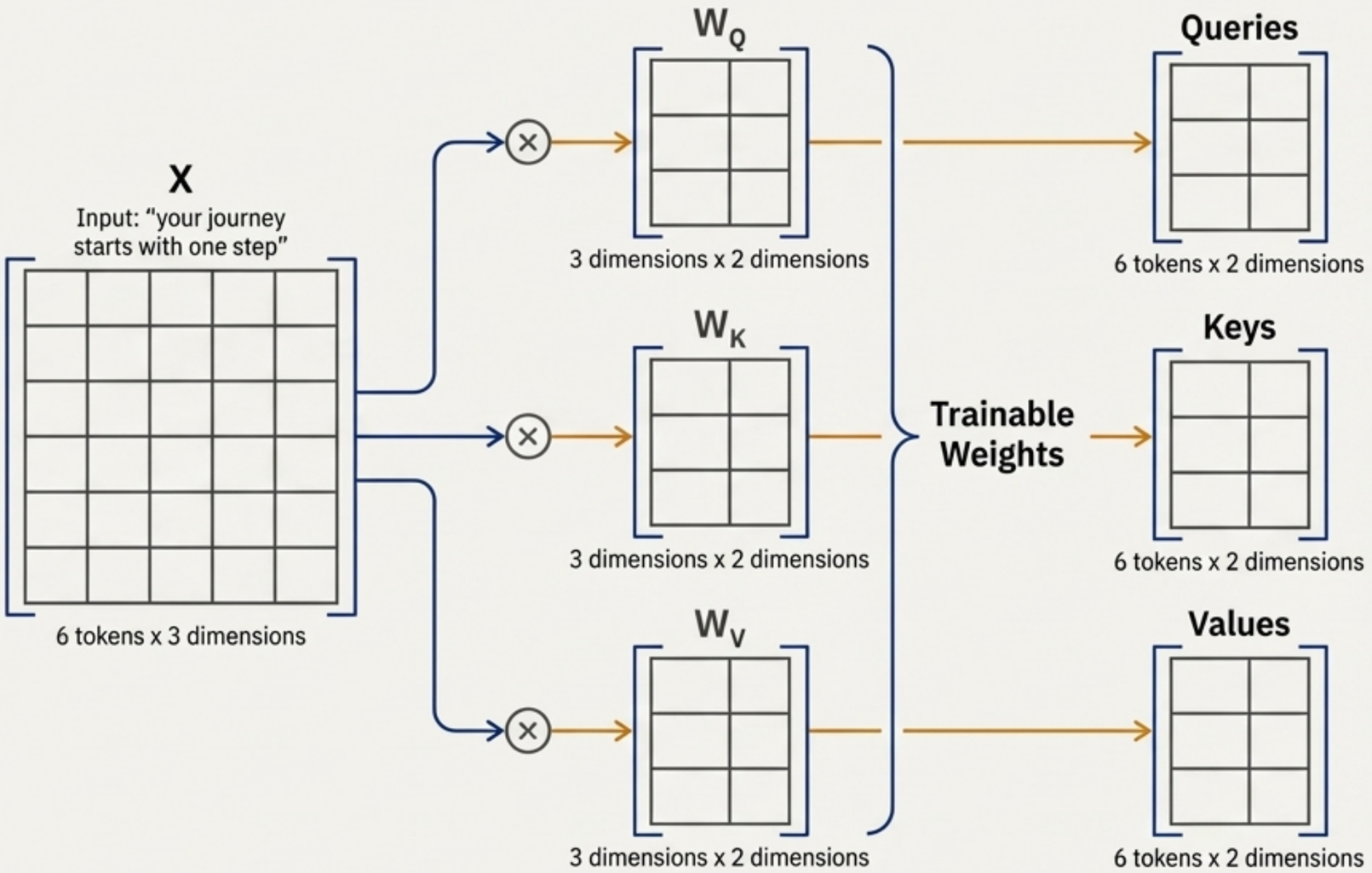
Transforms input embeddings into **“keys.”** A **key** acts like a label for each token, providing information that a query can match against.

`W_V` (Value Matrix)

Transforms input embeddings into **“values.”** A **value** represents the actual content or meaning of a token that will be used to construct the final output.

The Big Idea: Instead of using the raw input embeddings for everything, we project them into three distinct, specialized subspaces. This allows the model to learn nuanced relationships.

Step 1: Creating Queries, Keys, and Values



Explanation

We multiply our input matrix X with each of the trainable weight matrices (W_Q, W_K, W_V). This projects our 6 input tokens from a 3D space into a new, 2D space for each of their roles.

From this point forward, we will work exclusively with these new Q, K, and V representations.

Code Snippet (PyTorch)

```
# D_in=3, D_out=2
W_q = nn.Parameter(torch.rand(D_in, D_out))
W_k = nn.Parameter(torch.rand(D_in, D_out))
W_v = nn.Parameter(torch.rand(D_in, D_out))

queries = inputs @ W_q
keys = inputs @ W_k
values = inputs @ W_v
```

Step 2: Calculating Attention Scores via Dot Product

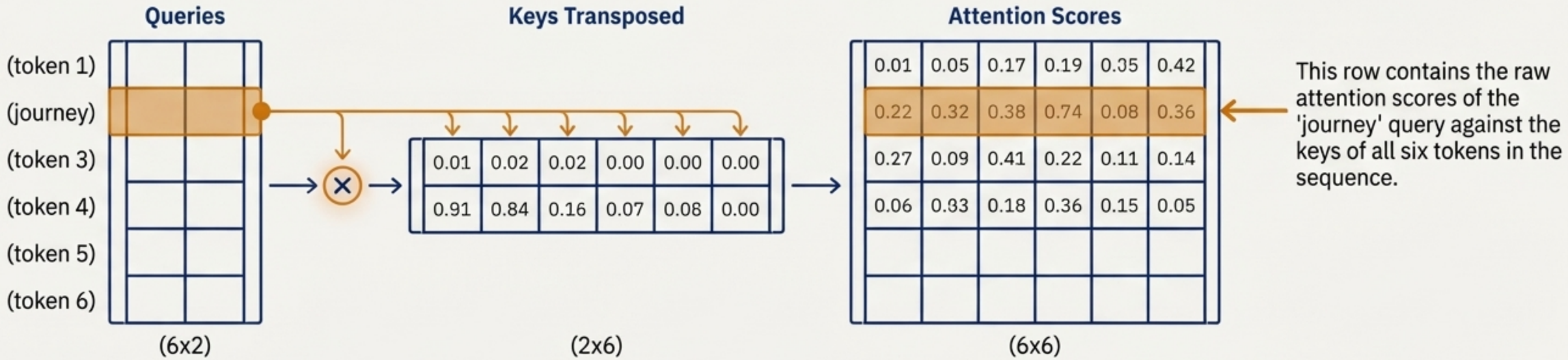
Intuition

The core of attention is determining the relationship between each token's query and every other token's key.

- A **Query** asks: "How relevant is each key to me?"
- The **Dot Product** answers: It measures the alignment or similarity between a query vector and a key vector. A high dot product means high relevance.

The Mechanics

To get the attention score for every token against every other token, we perform a matrix multiplication: **Queries @ Keys.T**



Code Snippet (PyTorch)

```
# queries shape: (6, 2), keys shape: (6, 2)
attn_scores = queries @ keys.T
# attn_scores shape: (6, 6)
```

Deep Dive: Why We Scale Scores by the Square Root of Key Dimension ($\sqrt{d_k}$)

The Problem

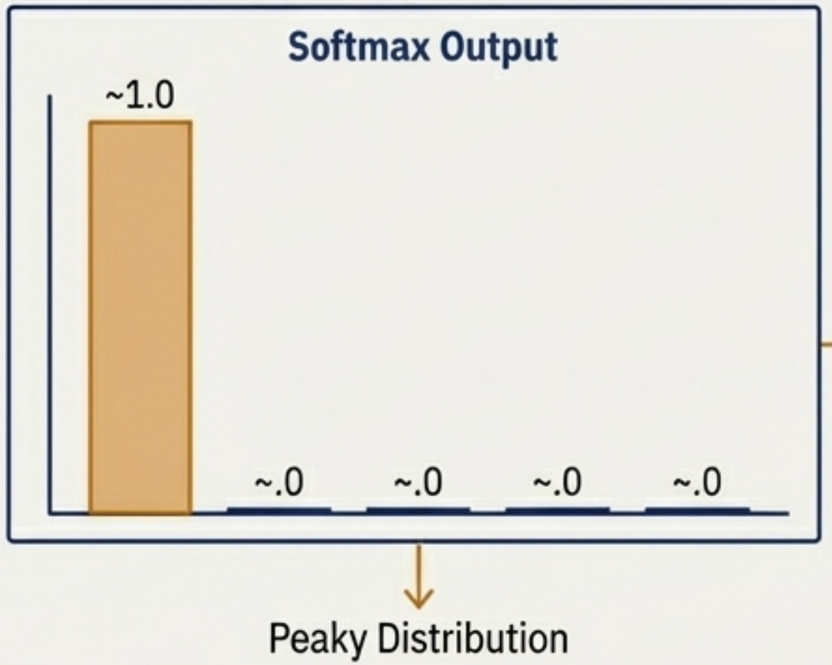
Raw dot products can grow very large, especially with high-dimensional vectors. Large inputs cause the softmax function to produce 'peaky' distributions, where one token gets nearly all the attention (a probability of ~1.0) and others get nearly zero. This makes learning unstable, as gradients can vanish.

The Solution

We scale the attention scores *before* applying softmax. Dividing by $\sqrt{d_k}$ accomplishes two things:

1. **Gradient Stability**: It counteracts large dot product values, leading to a softer, more distributed attention probability.
2. **Variance Control**: Dividing by $\sqrt{d_k}$ ensures the variance of the scaled scores remains close to 1, regardless of the dimension d_k, keeping learning dynamics consistent.

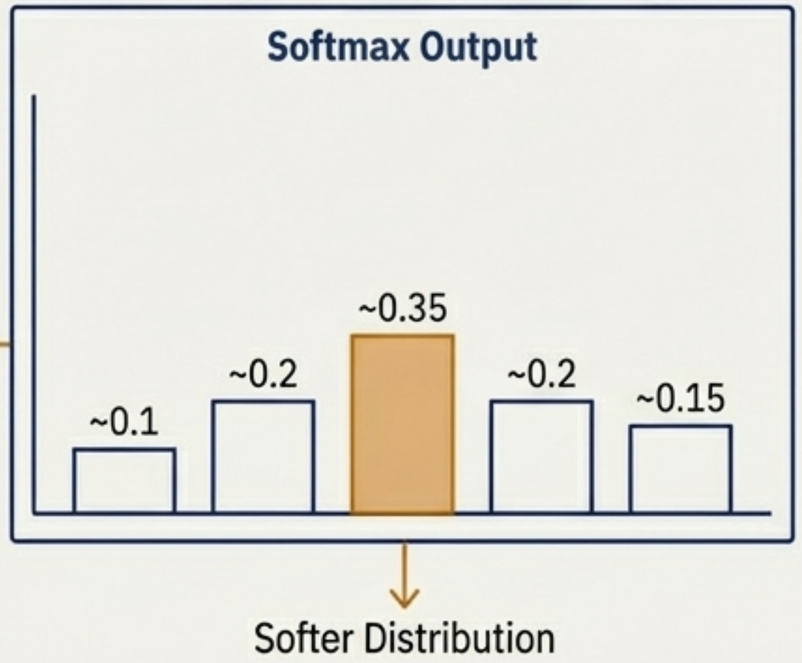
Unscaled Scores → Unstable Gradients



Code Evidence

```
Dimension: 100
Variance before scaling: 107.4
Variance after scaling by sqrt(100): 1.074
```

Scaled Scores → Stable Learning



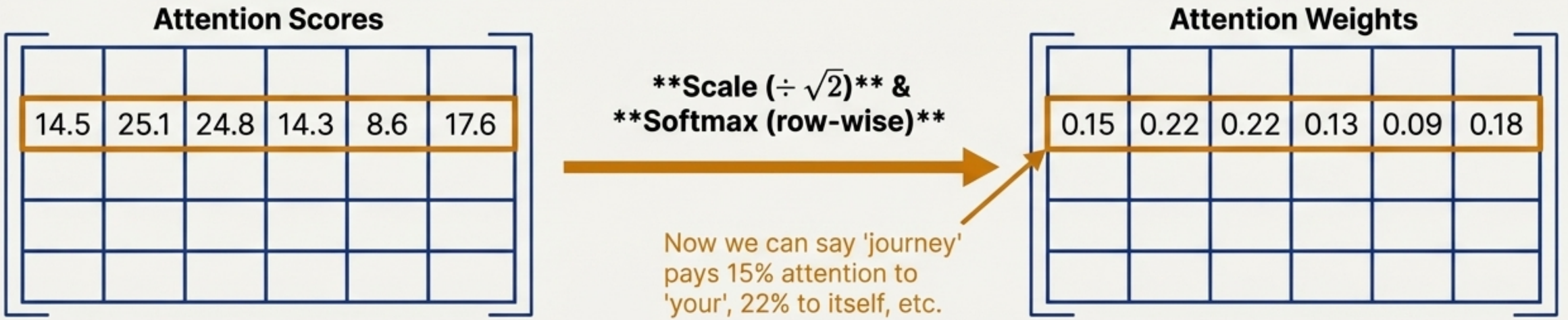
Step 3: Normalizing Scores into Attention Weights

$$\text{Attention Weights} = \text{softmax}\left(\frac{\text{Queries @ Keys.T}}{\sqrt{d_k}}\right)$$

The Outcome

1. **Scaling**: We first divide every element in the `Attention Scores` matrix by $\sqrt{d_k}$ (in our example, $\sqrt{2}$).
2. **Normalization**: We then apply the softmax function to each *row* independently.

The Result: An `Attention Weights` matrix (6×6) where each row is a probability distribution. All values in a row are between 0 and 1, and the entire row sums to 1.



Step 4: Synthesizing Context with Value Vectors

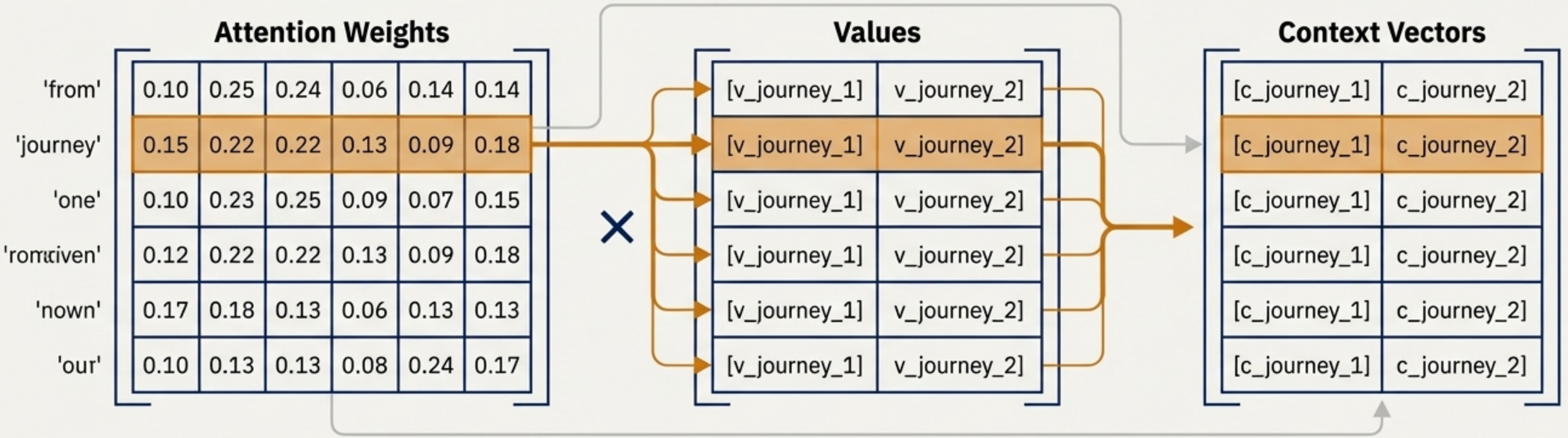
The Intuition

The `Attention Weights` matrix tells us *how much* attention each token should pay to every other token. The `Values` matrix provides the actual substance or content *to be attended to*. We now combine them.

The Mechanics

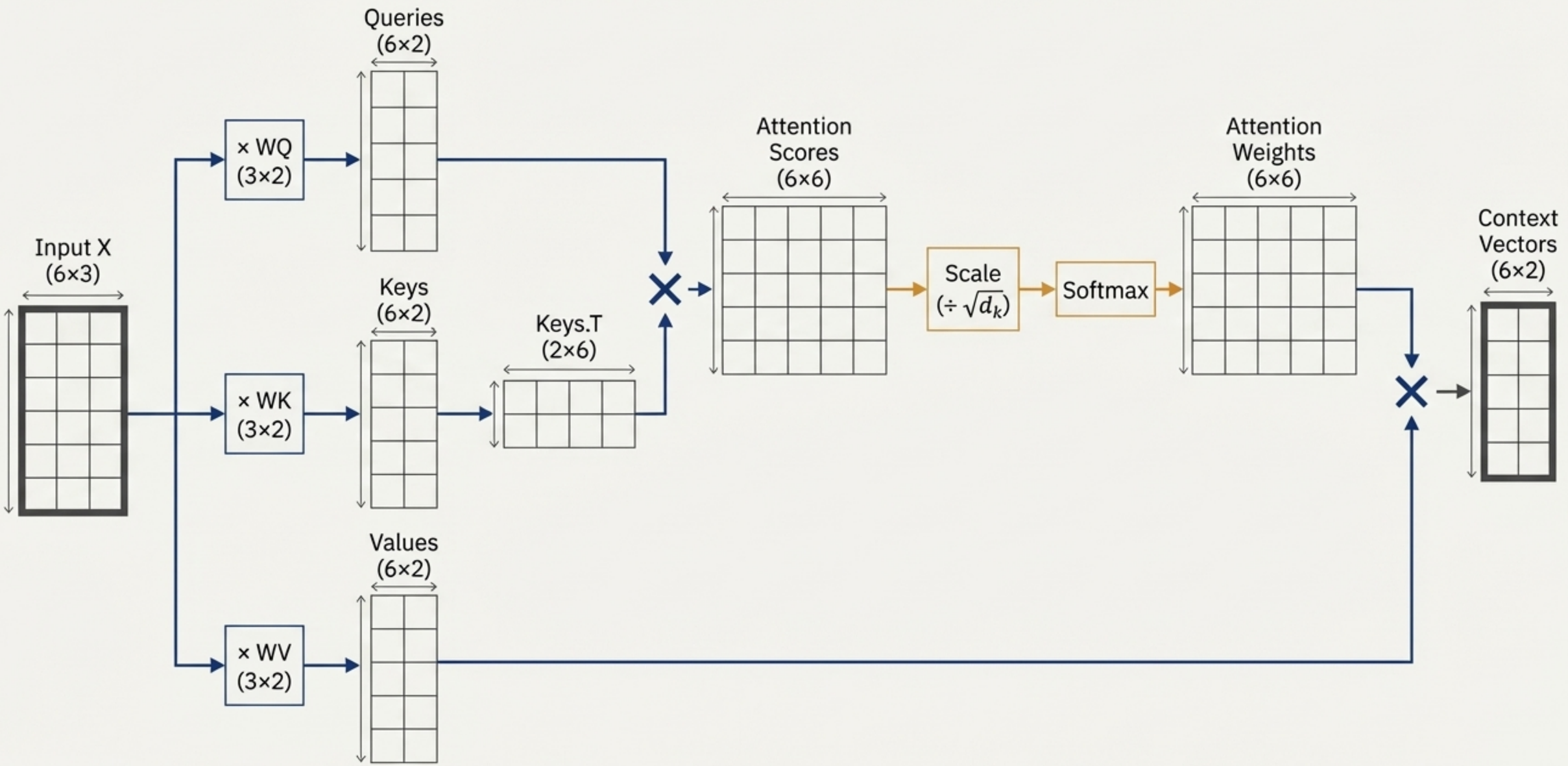
We compute the final context vectors by multiplying the `Attention Weights` matrix with the `Values` matrix.

`Context Vectors = Attention Weights @ Values`



Each row of the output is a new context vector. For example, the context vector for 'journey' is a weighted sum of ALL value vectors in the sequence. The weights are determined by the 'journey' row in the attention matrix. This new vector is enriched—it contains information not just about 'journey' itself, but about its relationship to every other word in the context.

The Complete Scaled Dot-Product Attention Mechanism

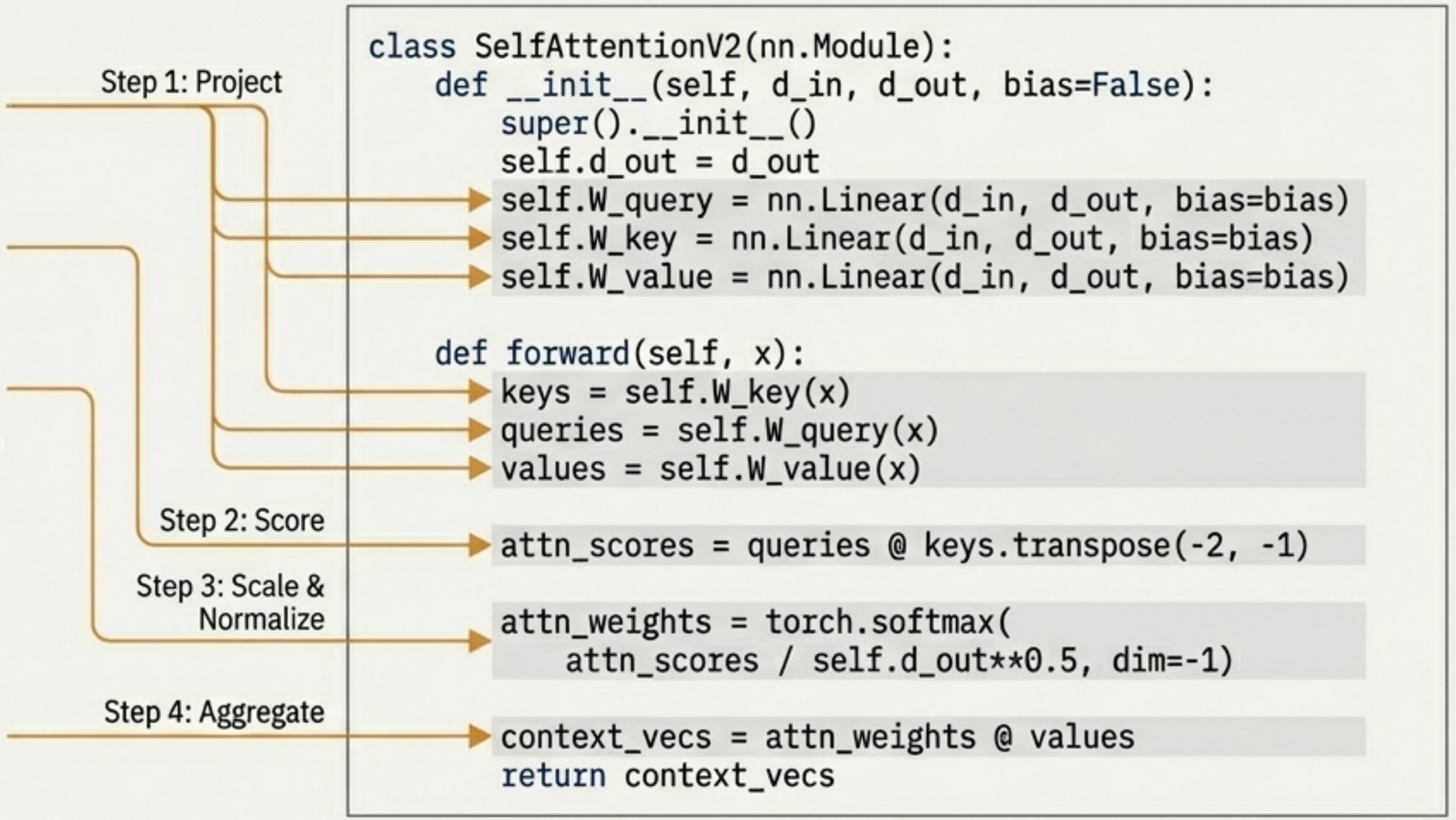


From Whiteboard to Code: Implementing the SelfAttention Class

Conceptual Flow (Recap)

1. ****Project****:
   ****Project****: Create Q, K, V from inputs.
2. ****Score****: Compute `Q @ K.T`.
3. ****Scale & Normalize****:
   Divide by `sqrt(d_k)` and apply `softmax`.
4. ****Aggregate****:
   Multiply by `V` to get context vectors.

PyTorch Implementation



The entire mechanism is encapsulated in a reusable PyTorch module. Note the use of `nn.Linear` for a more optimized and stable weight initialization.

A Final Analogy: The Database Retrieval System

The terms "Query," "Key," and "Value" are not arbitrary. They are borrowed from information retrieval systems, which provides a powerful intuition.

This is your search term. It represents the current token's need for information. It's what the model is currently focusing on.

These are the indices or labels of all the documents in the database. The system matches your query against these keys to find the most relevant items.

These are the actual documents or content. Once the most relevant keys are identified (via attention weights), the system retrieves the corresponding values to construct the answer.

Self-attention allows every token to act as a query, retrieving a custom-blended set of values from all other tokens in the sequence based on key-matching.

What This Engine Powers and What Comes Next

The Power of Scaled Dot-Product Attention

This trainable mechanism is the fundamental building block that allows Transformers to learn complex, context-dependent relationships between tokens in a sequence. It is the core of models like GPT.

The Road Ahead

- **Causal Attention**: How do we adapt this mechanism for decoder models to prevent them from 'cheating' by looking at future tokens during training?
- **Multi-Head Attention**: What if we run multiple attention mechanisms in parallel, each focusing on different types of relationships? This is the next major upgrade to our engine.