ARCHITECTURE TIER

MODULE 12

Attention Mechanism

The breakthrough architecture behind modern AI

TINYTORCH ARCHITECTURE TIER

Module 12: Attention

Learning to Focus in Sequence Modeling

PREREQUISITES

Mod 01: Tensors

PREREQUISITES

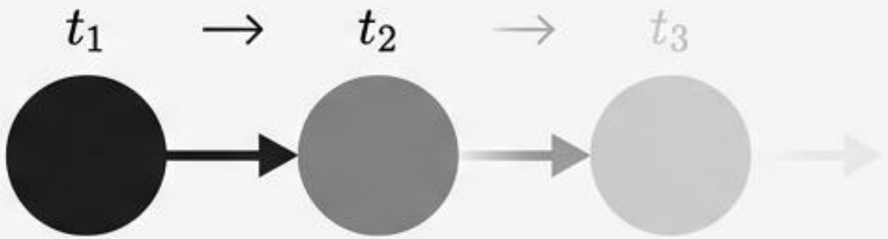
Mod 06: Autograd

PREREQUISITES

Mod 11: Embeddings

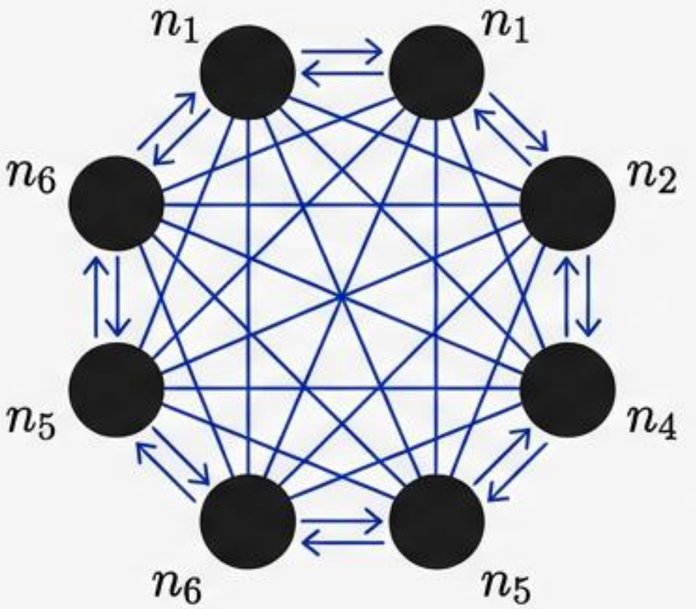
The Problem: The Sequential Bottleneck

The Old Way (RNN)



Sequential Processing.
History compressed into one fixed state.
Information degrades over distance.

The Attention Way

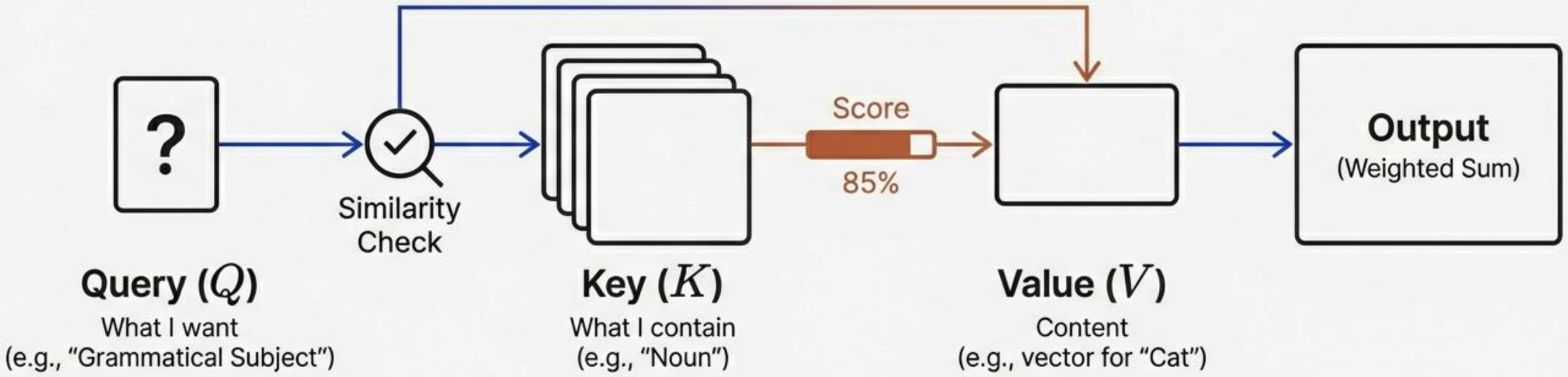


Parallel Processing.
Direct access ($O(1)$) to entire history.
No compression bottleneck.

Trade-off: Memory scales Quadratically ($O(n^2)$).

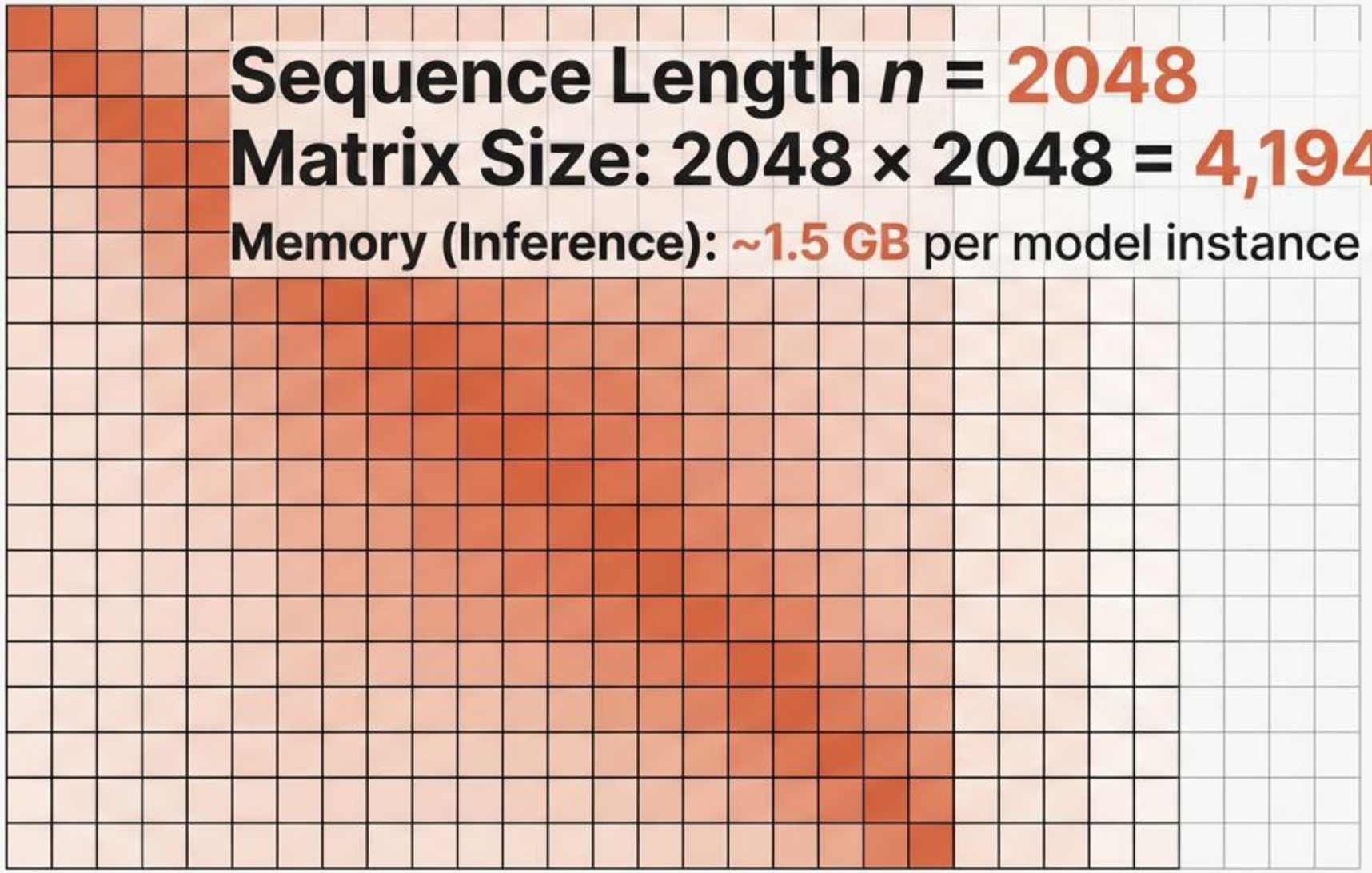
The Abstraction: Information Retrieval

The Fuzzy Database Lookup Metaphor



We treat sequence modeling as a **database lookup**. Unlike a hash map (exact match), **Attention** uses a **differentiable "soft" lookup** based on vector similarity.

Systems Constraints: The $O(n^2)$ Price Tag



Sequence Length n = 2048

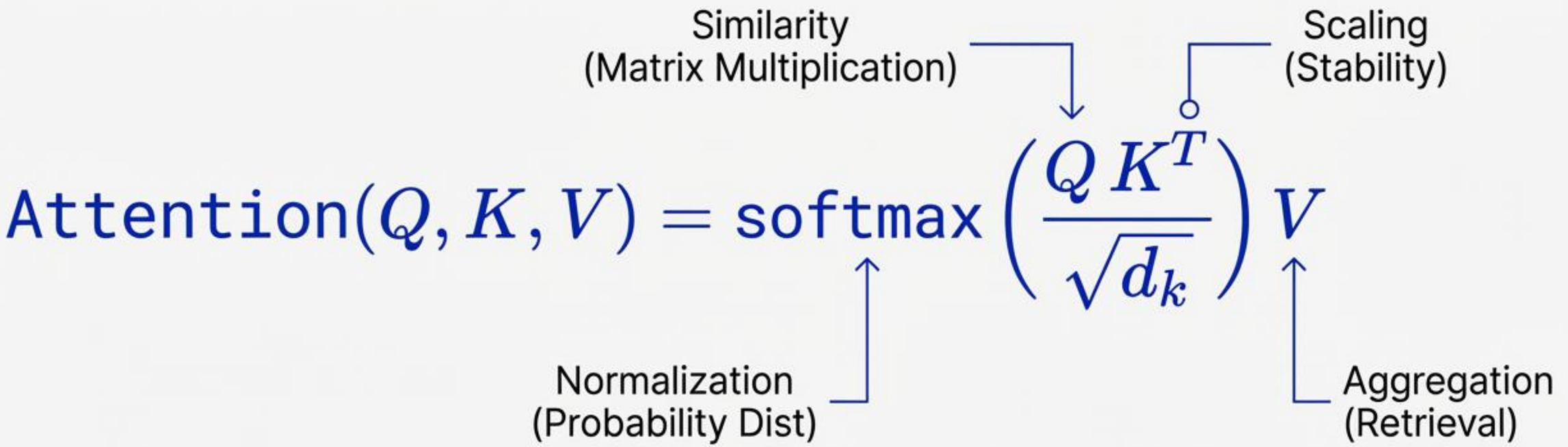
Matrix Size: 2048 × 2048 = 4,194,304 elements

Memory (Inference): ~1.5 GB per model instance (Attention Weights Only)

The Invariant:
Doubling sequence length quadruples memory usage.

This matrix is the primary bottleneck for Long Context modeling.

The Formula: Scaled Dot-Product Attention



This single formula maps directly to 4 lines of TinyTorch code.

Step 1: Similarity Scores

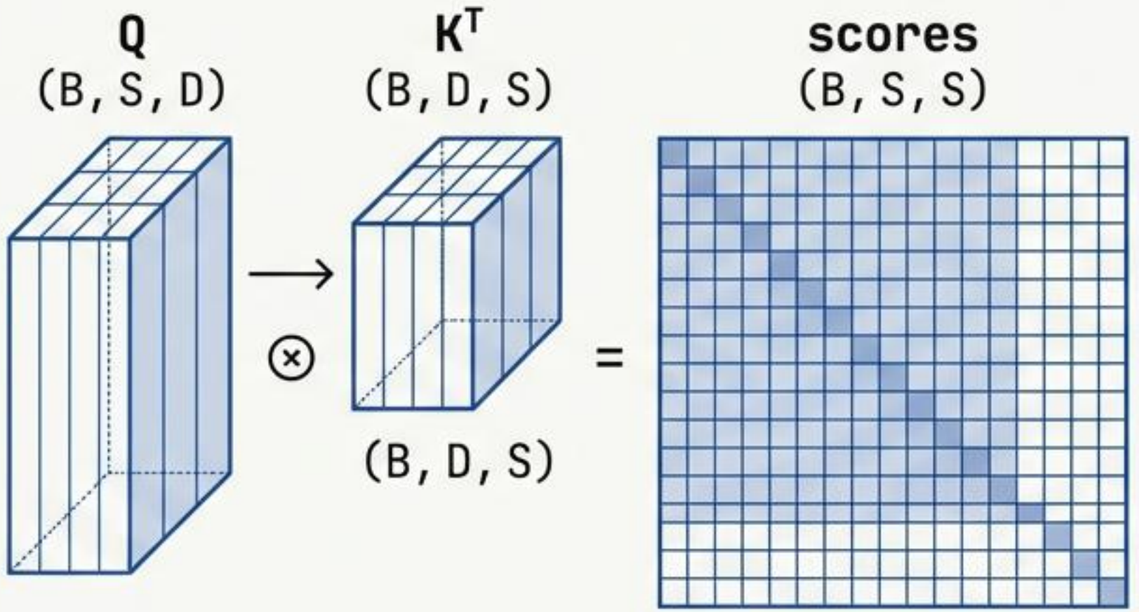
Code Implementation

```
# Q: (batch, seq, d_model)
# K: (batch, seq, d_model)

# Transpose K for matmul alignment
K_t = K.transpose(-2, -1)

# The O(n^2) operation
scores = Q.matmul(K_t)
```

Matrix Multiplication Visual



Every Query compares against every Key.

Step 2: Scaling for Stability

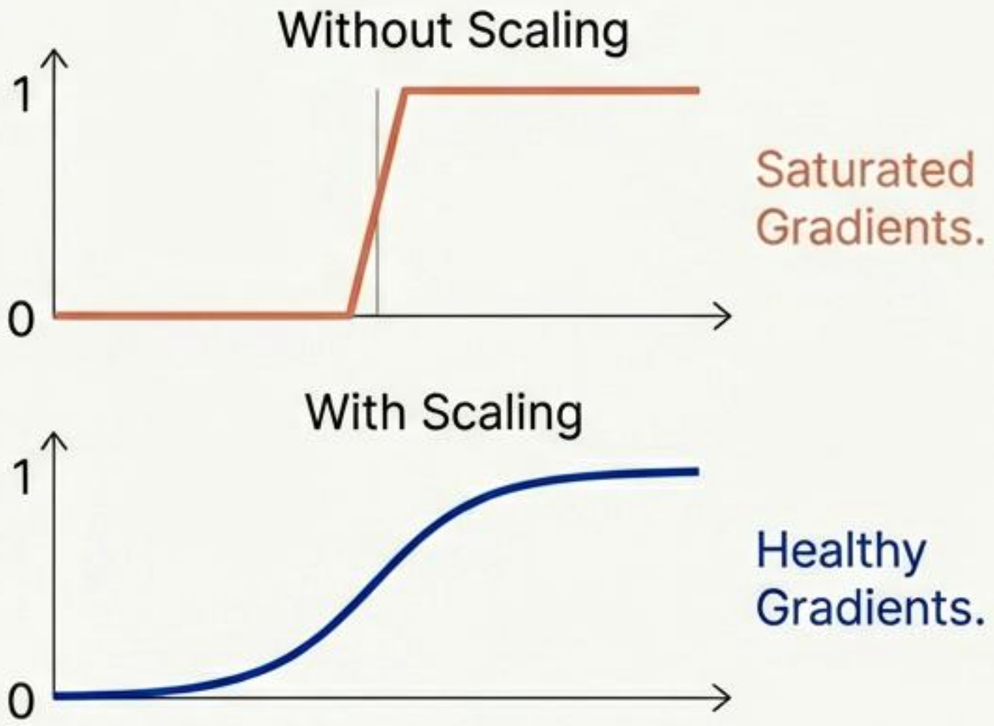
Code Implementation

```
d_model = Q.shape[-1]

scale_factor = 1.0 /
math.sqrt(d_model)

# Normalize variance
scores = scores * scale_factor
```

Scaling & Softmax Visual



Large dimensions → Large dot products → Vanishing Gradients. Scaling maintains variance.

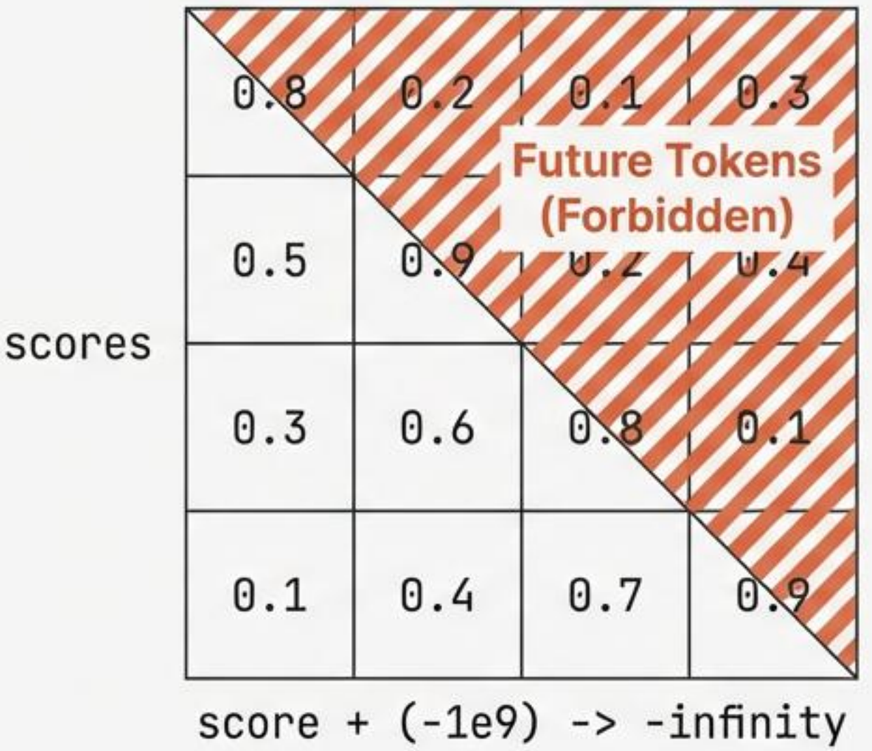
Step 3: Causal Masking

Code Implementation

```
if mask is not None:
    # mask is 0 for masked, 1 for unmasked
    mask_data = mask.data

    # Add -infinity to forbidden positions
    adder_mask = (1.0 - mask_data) * -1e9
    scores = scores + Tensor(adder_mask)
```

Causal Masking Visual



Applying an additive mask of -1e9 to future token positions, effectively excluding them from attention after the softmax operation.

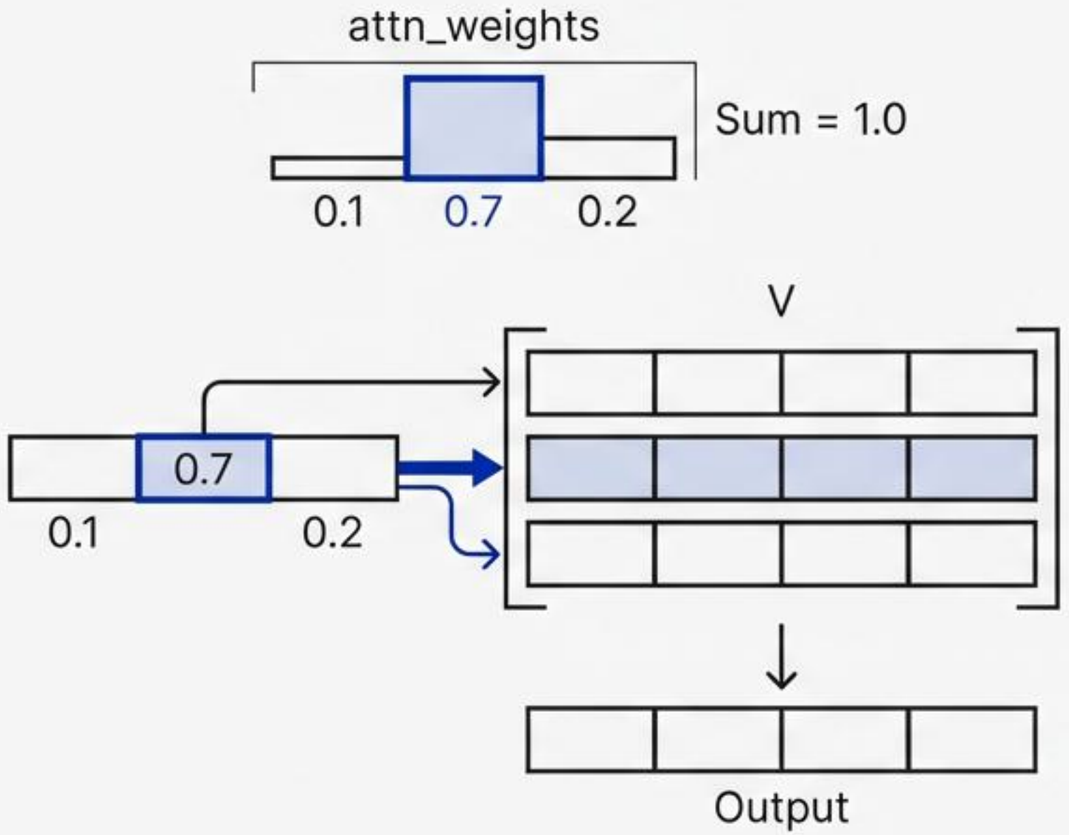
Step 4: Normalization & Retrieval

Code Implementation

```
# Normalize to probabilities
softmax = Softmax()
attn_weights = softmax(scores, dim=-1)

# Retrieve weighted values
output = attn_weights.matmul(V)
```

Normalization & Retrieval Visual



Complete Implementation

tinytorch/core/attention.py

```python
def scaled_dot_product_attention(Q, K, V, mask=None):
    d_model = Q.shape[-1]

    # 1. Similarity (O(n^2))
    scores = Q.matmul(K.transpose(-2, -1))

    # 2. Scale
    scores = scores * (1.0 / math.sqrt(d_model))

    # 3. Mask (Causal)
    if mask is not None:
        scores = scores + Tensor((1.0 - mask.data) * -1e9)

    # 4. Normalize & Retrieve
    attn_weights = Softmax()(scores, dim=-1)
    output = attn_weights.matmul(V)

    return output, attn_weights
```

Geometry

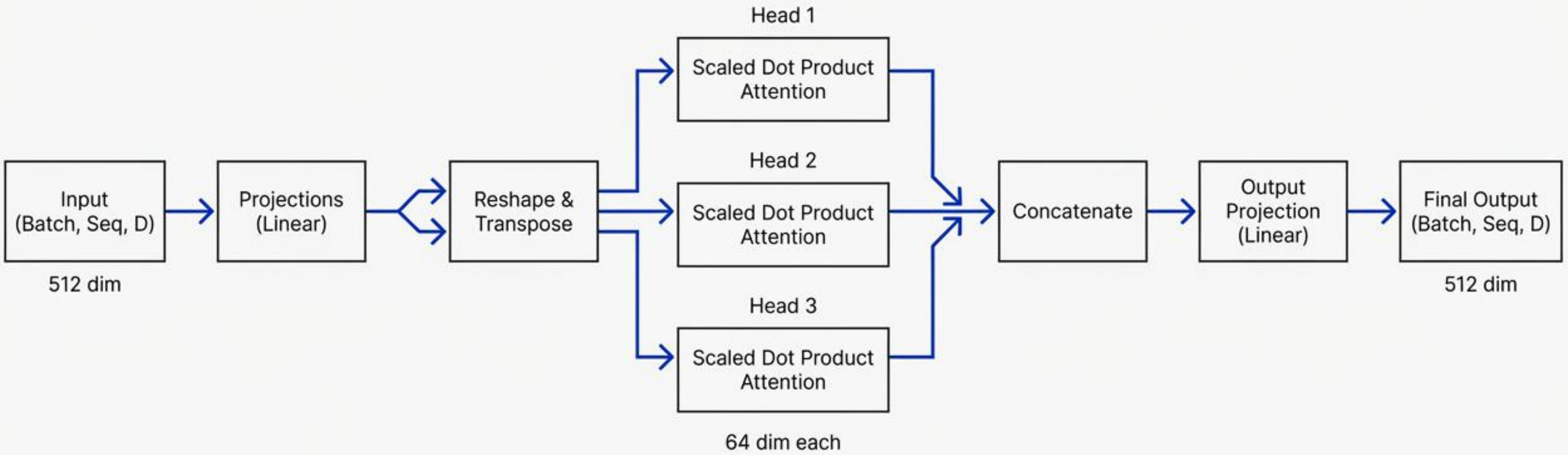
Stability

Probability

Why One Head Isn’t Enough

- Single Head: Averages all relationships into one blurry view.
- Multi-Head: Runs parallel subspaces. Specialized attention for Syntax, Semantics, and Structure.
- Constraint: Total parameter count remains constant (Split `d_model` across heads).

Architecture: Split, Attend, Concatenate



MHA Implementation: Initialization

Defining the projections

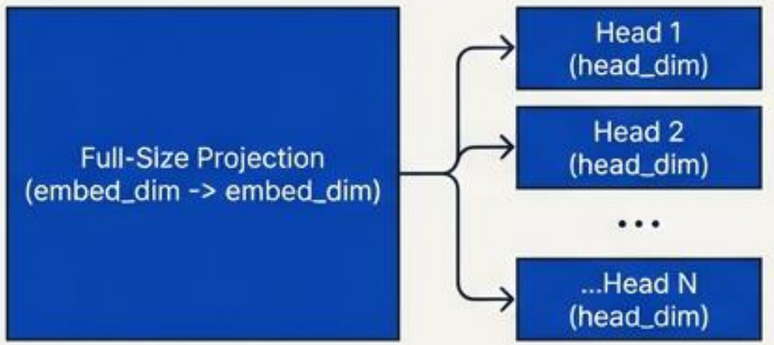
```python
class MultiHeadAttention:
    def __init__(self, embed_dim, num_heads):
        # Validation
        if embed_dim % num_heads != 0:
            raise ValueError('Dim must divide by heads')

        self.head_dim = embed_dim // num_heads

        # Independent projections
        self.q_proj = Linear(embed_dim, embed_dim)
        self.k_proj = Linear(embed_dim, embed_dim)
        self.v_proj = Linear(embed_dim, embed_dim)

        self.out_proj = Linear(embed_dim, embed_dim)
```

Implementation Note: We define full-size projections (embed_dim -> embed_dim). The "Splitting" into heads happens logically in the forward pass via reshaping, not by creating 8 separate small Linear layers.



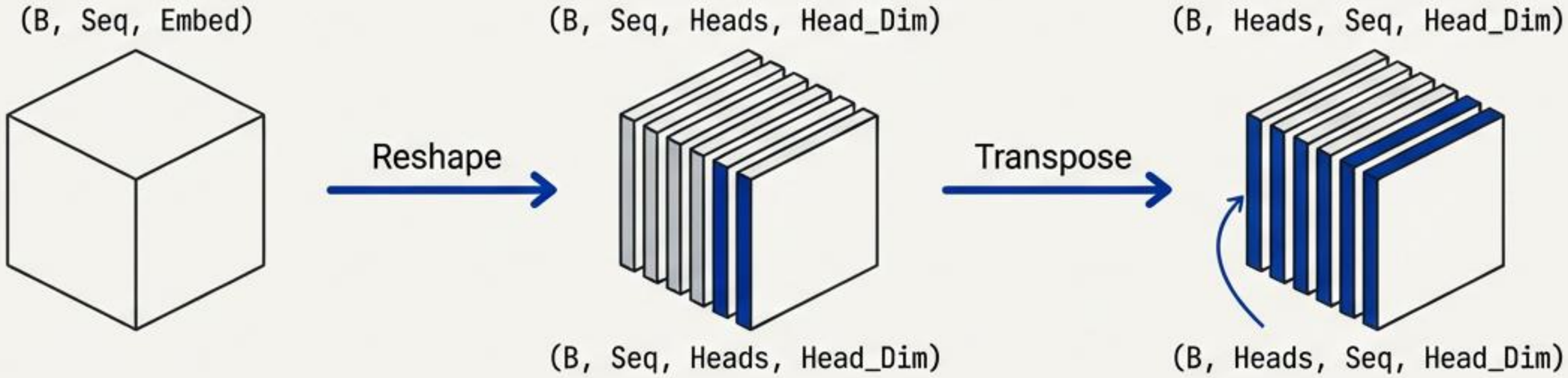
MHA Forward: The Reshape Dance

Goal: Isolate heads for parallel processing.

```
# 1. Project
Q = self.q_proj.forward(x) # (B, Seq, Embed)

# 2. Reshape to expose heads
# (B, Seq, Heads, Head_Dim)
Q = Q.reshape(batch, seq, self.num_heads, self.head_dim)

# 3. Transpose for parallel attention
# Swap Seq and Heads: (B, Heads, Seq, Head_Dim)
Q = Q.transpose(1, 2)
```



Moves ‘Heads’ dimension outside so MatMul treats them as parallel batches.

MHA Forward: Merge and Output

```
# 4. Apply Attention (broadcasts across heads)
attended, _ = scaled_dot_product_attention(Q, K, V, mask)

# 5. Transpose back: (B, Seq, Heads, Head_Dim)
attended = attended.transpose(1, 2)

# 6. Contiguous Reshape: (B, Seq, Embed)
concat_output = attended.reshape(batch, seq, self.embed_dim)

# 7. Mix
output = self.out_proj.forward(concat_output)
```

Crucial Step: The final linear projection allows heads to share information.

Systems Analysis: The Memory Wall

Scenario: GPT-3 Scale (96 Layers, 2048 Context)

Implication: Training requires massive memory buffers just for the attention gradients.

Common Implementation Pitfalls

Shape Mismatch

RuntimeError: Inner dimensions must match.

Fix: Forgot `K.transpose(-2, -1)`.

Softmax Axis

Weights don't sum to 1.0.

Fix: Must specify `dim=-1`.

Mask Broadcasting

Broadcasting failure in Multi-Head.

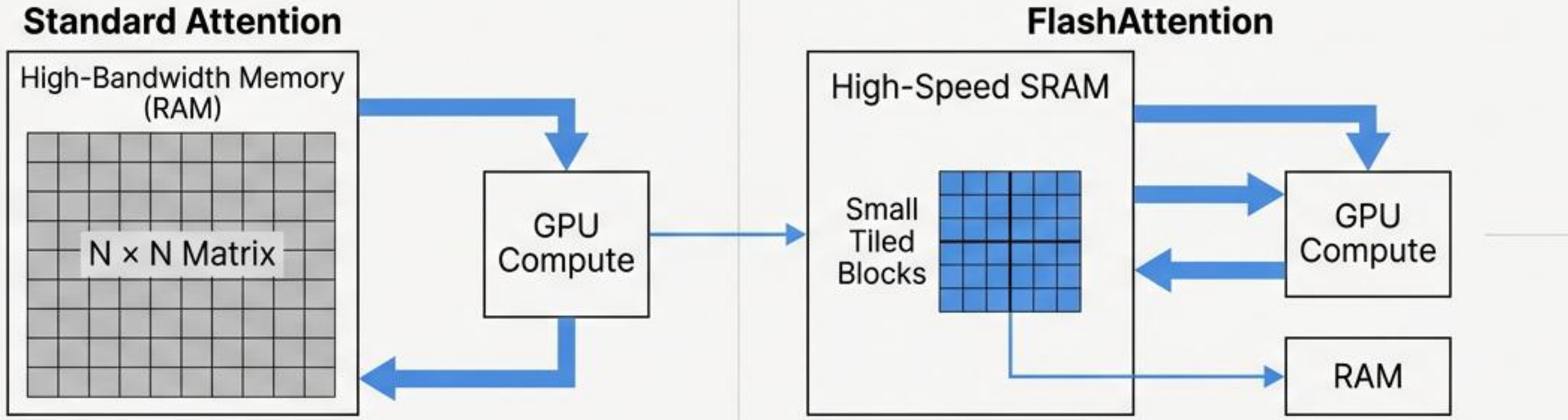
Fix: Mask must be 4D `(Batch, 1, Seq, Seq)` to broadcast over Heads.

Concept to Code Map

Concept	TinyTorch Code	Why?
Similarity	`Q.matmul(K.transpose(-2, -1))`	Vectorized Comparison
Stability	`scores * (1.0 / sqrt(d))`	Prevent Saturation
Causality	`scores + (1-mask) * -1e9`	Hide Future Tokens
Selection	`softmax(scores) @ V`	Differentiable Retrieval
Parallelism	`transpose(1, 2)`	Treat Heads as Batch

Beyond $O(n^2)$: Production Optimizations

The code we wrote is the **mathematical baseline (Exact Attention). Production systems use optimized kernels** to bypass the **memory wall.**



Big $N \times N$ Matrix stored in RAM.

Small Tiled blocks computed in high-speed SRAM.

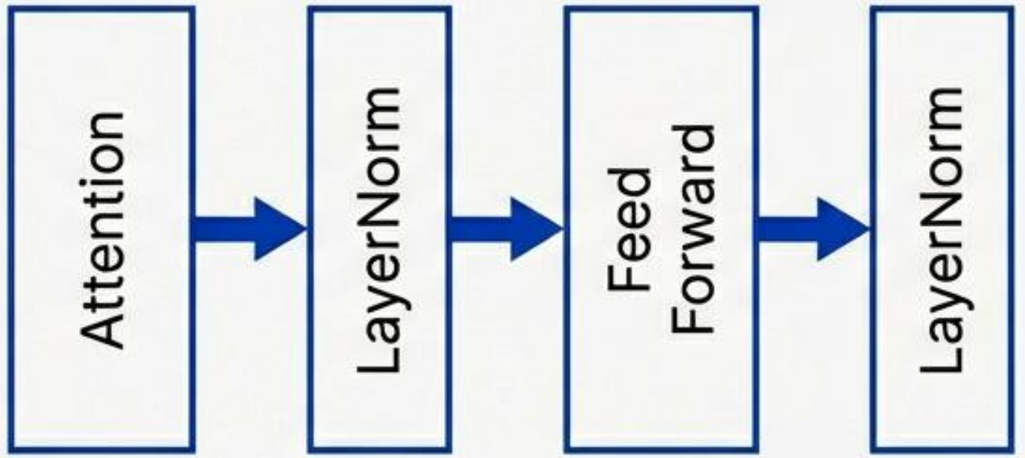
- **FlashAttention:** Tiling to avoid materializing full matrix ($O(n^2) \rightarrow O(n)$ memory).
- **Sparse Attention**: Computing only local or strided blocks.
- **Linear Attention:** Approximating the kernel.

Summary & Next Steps

Takeaways

- Attention enables direct, parallel access to history.
- The cost is Quadratic Memory ($O(n^2)$).
- MultiHeadAttention uses reshaping to learn diverse relationships.

Coming Up: Module 13



We will wrap this Attention block with FFNs and Residuals to build the GPT architecture.