ARCHITECTURE TIER

MODULE 11

Embeddings

Token lookup and position encoding for sequence models

TinyTorch Module 11

Architecture Tier | Embeddings

01-08 Foundation → **11 Embeddings** → 12 Attention → 13 Transformers

Core Goal

Bridge the gap between discrete text tokens (integers) and continuous neural operations (vectors).

The Build

1. `Embedding`: Efficient table lookup
2. `PositionalEncoding`: Injecting order
3. `EmbeddingLayer`: Production integration

The Impedance Mismatch

Discrete Input (Tokens)

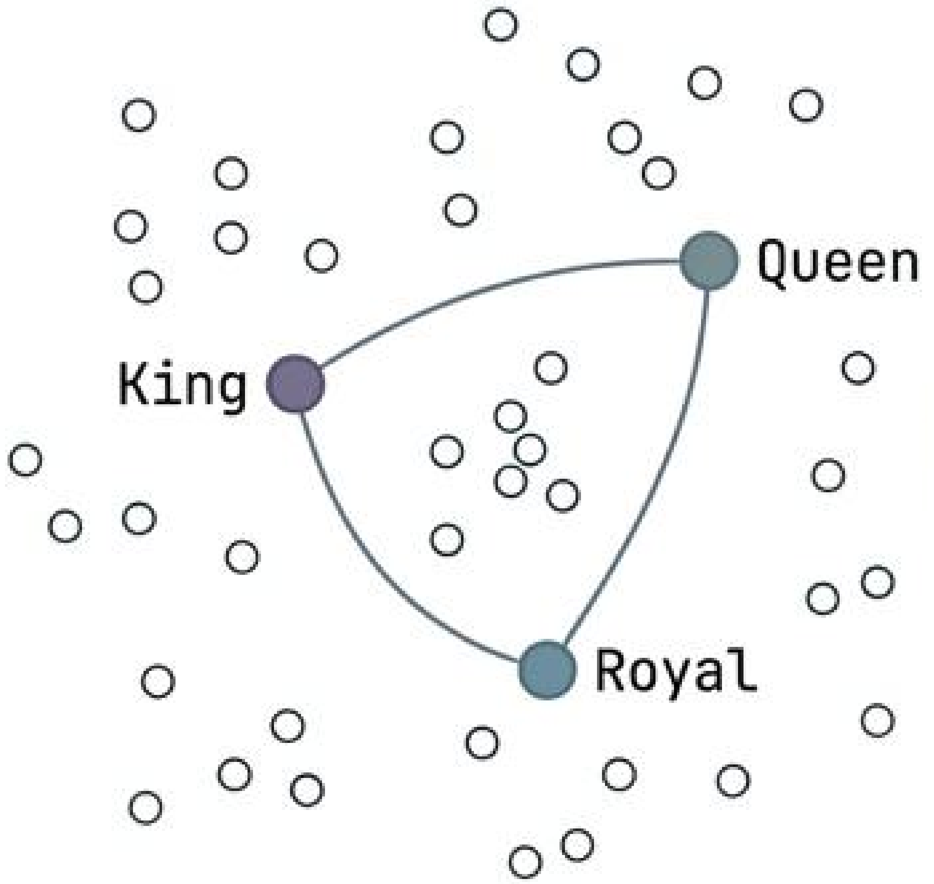
"cat"
(ID: 42)

"sat"
(ID: 7)

"mat"
(ID: 15)

Hard edges.
Indifferentiable.
No similarity (42 ≠ 43).

Continuous Space (Vectors)



Differentiable manifold.
Semantic proximity.

Challenge: We need a translation layer to map index $i \to v \in \mathbb{R}^d$.

The Systems Constraint: The Memory Wall

Why not just use One-Hot Encoding?

The Math

Vocabulary (V) = 50,257

Embedding Dimension (D) = 12,288

Batch Size (B) = 32

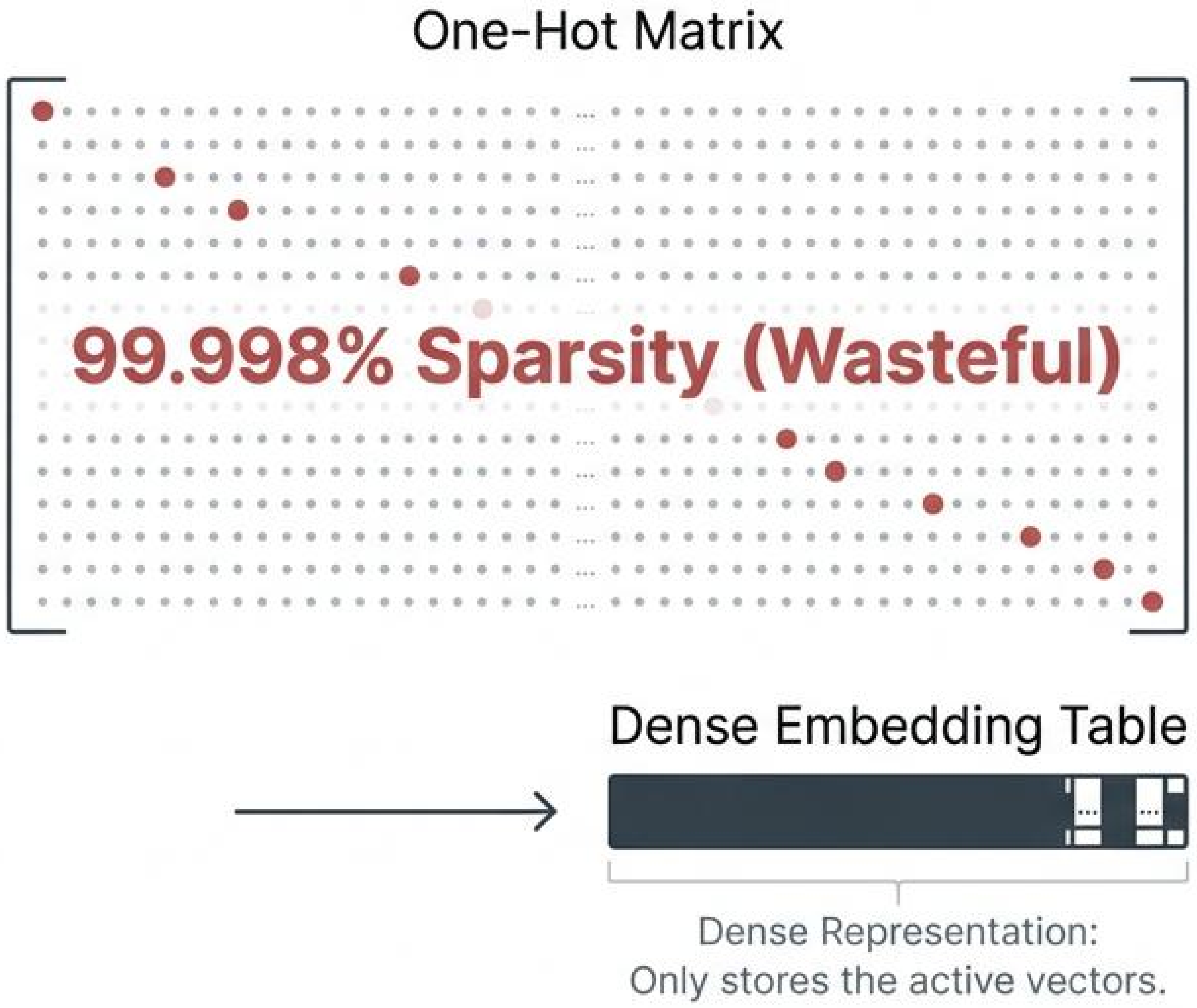
Sequence Length (L) = 2048

One-Hot Input Tensor: $B \times L \times V$

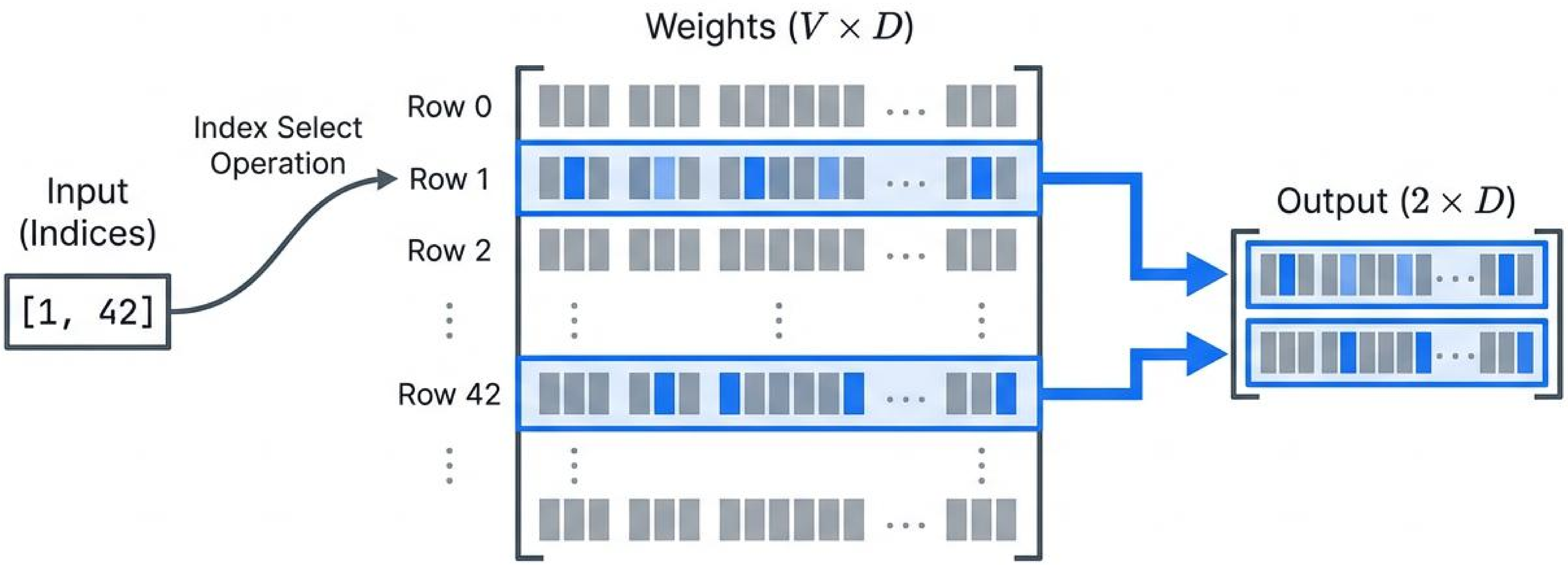
Elements: $32 \times 2048 \times 50{,}257 \approx 3.3$ billion floats

Memory: **≈ 13 GB** per batch (Input only)

Memory: **≈ 13 GB** per batch (Input only)



The Solution: A Learnable Lookup Table



Initialized randomly. Updated via backpropagation. Similar contexts → Similar vectors.

Implementing the Table: Initialization

tinytorch/core/embeddings.py

```
class Embedding:
    def __init__(self, vocab_size: int, embed_dim: int):
        self.vocab_size = vocab_size
        self.embed_dim = embed_dim

        # Xavier initialization for better gradient flow
        # Limit derived from fan-in + fan-out
        limit = math.sqrt(6.0 / (vocab_size + embed_dim))

        # The core lookup table
        self.weight = Tensor(
            np.random.uniform(-limit, limit, (vocab_size, embed_dim))
        )
```

The State: A simple matrix of learnable parameters.

Xavier Init: Keeps variance stable. Prevents vanishing gradients.

The Forward Pass: Efficient Indexing

tinytorch/core/embeddings.py

```python
def forward(self, indices: Tensor) -> Tensor:
    # Validate indices are in range
    if np.any(indices.data >= self.vocab_size):
        raise ValueError("Index out of range")
        raise ValueError("Index out of range")

    # Perform embedding lookup using advanced indexing
    # O(1) per token. No matrix multiplication.
    embedded = self.weight.data[indices.data.astype(int)]

    return Tensor(embedded)
```

Systems Insight: Fancy Indexing

- NumPy `arr[indices]` operation.
- Avoids creating intermediate One-Hot vectors.
- Speed is independent of Vocabulary Size (V).
- Handles arbitrary batch shapes (B, T, ...) automatically.

The Problem: Loss of Order (Bag of Words)

Sentence A: `The cat sat on the mat`

Sentence B: `The mat sat on the cat`

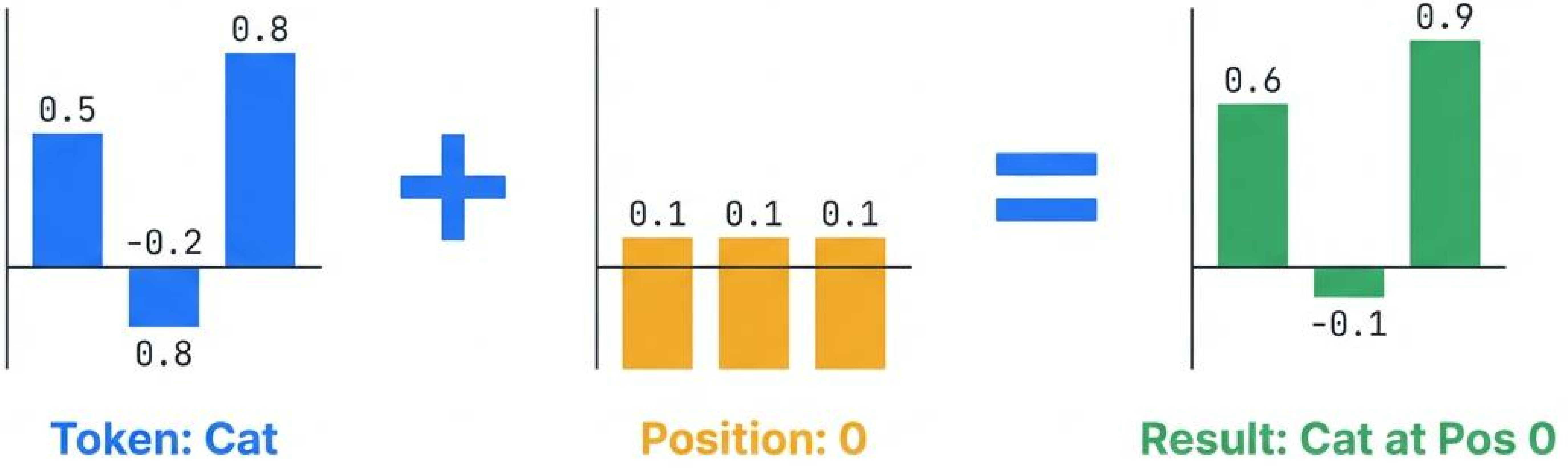
Embedding Lookup
(Summed/Averaged) $\longrightarrow \Sigma$

Vector A == Vector B

Without explicit position information, the model cannot distinguish **subject from object**.

Injecting Position

Final Vector = Embedding(Token) + Embedding(Position)



Invariant: Shapes must match perfectly (B, T, D)

Approach 1: Learned Positional Encoding

- Treat positions [0, 1, 2, ...] exactly like tokens.

  Learn a specific vector for “Position 1”, “Position 2”, etc.

```
class PositionalEncoding:
    def __init__(self, max_seq_len: int, embed_dim: int):
        # A second lookup table, just for positions
        limit = math.sqrt(2.0 / embed_dim)
        self.position_embeddings = Tensor(
            np.random.uniform(-limit, limit, (max_seq_len, embed_dim))
        )
```

Pros

Flexible.

Adapts to task.

Slate Grey

Cons

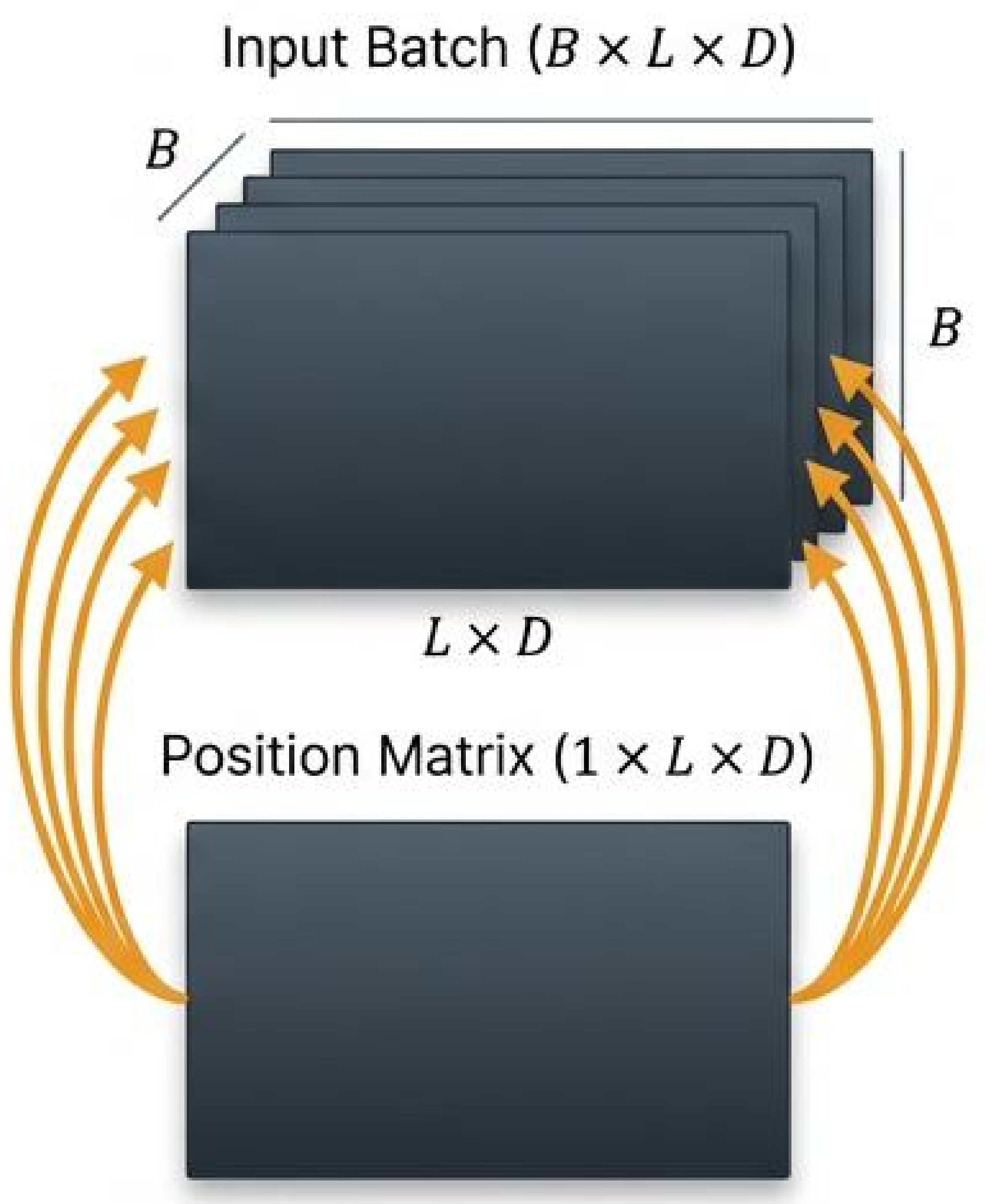
Fixed max length (L_max).

Extra parameters.

Slate Grey

Applying Learned Positions

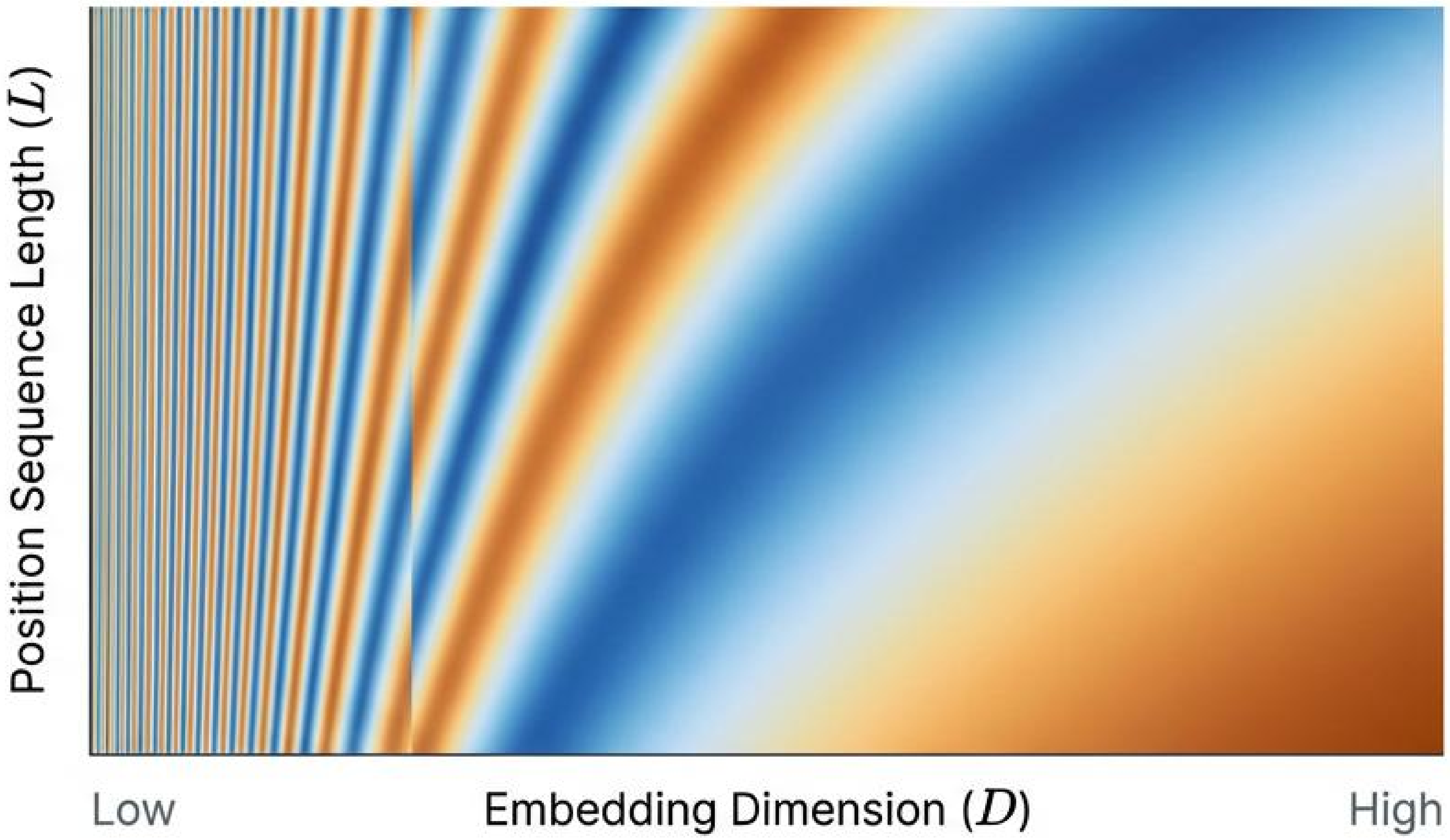
```
def forward(self, x: Tensor) -> Tensor:
    batch_size, seq_len, embed_dim = x.shape

    # 1. Slice: Get positions 0 to seq_len
    pos_embeddings = self.position_embeddings[:seq_len]

    # 2. Reshape for broadcasting: (1, seq_len, embed_dim)
    pos_data = pos_embeddings.data[np.newaxis, :, :]

    # 3. Add to input (Broadcasting copies to all batch items)
    return x + Tensor(pos_data)
```



Broadcasting in NumPy/TinyTorch

Approach 2: Sinusoidal Encodings

Handling extrapolation (infinite length) with zero parameters.



Sinusoidal Positional Encoding Visualization

Even Dims: $\sin(pos/10000^{2i/d})$

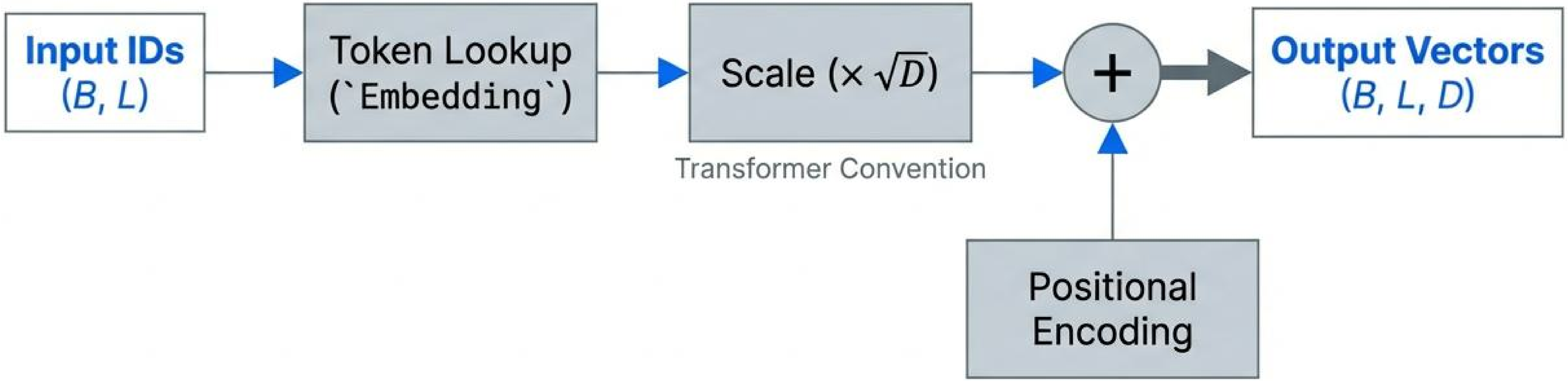
Odd Dims: $\cos(pos/10000^{2i/d})$

Implementing Sinusoidal Math

```
def create_sinusoidal_embeddings(max_seq_len, embed_dim):
def create_sinusoidal_embeddings(max_seq_len, embed_dim):
    # 1. Create position indices (Column vector)
    position = np.arange(max_seq_len)[:, np.newaxis]

    # 2. Calculate frequencies (Exponential decay)
    div_term = np.exp(np.arange(0, embed_dim, 2) *
                      -(math.log(10000.0) / embed_dim))
    # 3. Apply Sin/Cos to Even/Odd columns
    pe = np.zeros((max_seq_len, embed_dim))
    pe[:, 0::2] = np.sin(position * div_term)
    pe[:, 1::2] = np.cos(position * div_term)

    return Tensor(pe)
```

Pure NumPy. Deterministic. No trainable `self.weight`.

The Complete System: EmbeddingLayer



Production Wrapper

Encapsulates complexity. Provides a clean API matching PyTorch's `nn.Transformer` inputs.

Integrating the Pipeline

```
class EmbeddingLayer:
    def __init__(self, vocab, dim, pos_encoding='learned'):
        self.token_embedding = Embedding(vocab, dim)
        if pos_encoding == 'learned':
            self.pos_encoding = PositionalEncoding(...)
        # ... handling for sinusoidal ...

    def forward(self, tokens):
        # 1. Base Lookup
        x = self.token_embedding(tokens)

        # 2. Scale (Transformer Invariant)
        x = x * math.sqrt(self.embed_dim)

        # 3. Inject Position
        return self.pos_encoding(x)
```

****Composition over Inheritance.** We build the layer by combining smaller, focused components.

Systems Analysis: Memory Footprint

Embedding tables are often the largest parameter block in a model.

Model	Vocab	Dimension	Memory (Approx)
Small BERT	30k	768	**92 MB**
GPT-2	50k	1024	**206 MB**
GPT-3	50k	12,288	**2.4 GB**

Trade-off

Constraint: Increasing D_{model} improves semantic capacity but linearly increases RAM usage.

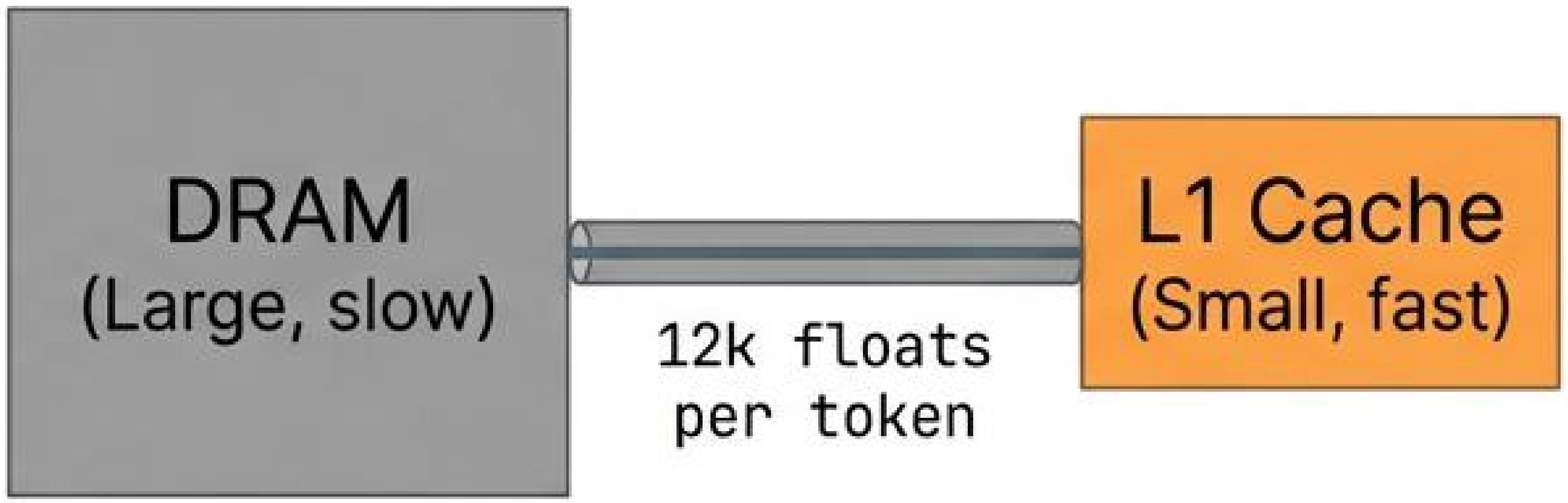
`Memory ≈ V × D × 4 bytes`

Systems Analysis: Throughput

Speed

- Lookup Cost: $O(1)$ per token.
- Independence: Fetching `vector[100]` is just as fast as `vector[100000]`.

The Bottleneck



Training spends ~15% of time in Embeddings, mostly waiting for memory transfer, not compute.

Insight

Sparse Gradients: In a batch of 65k tokens, we only touch a fraction of the vocab. Efficient backprop exploits this.

Failure Modes & Debugging

- `IndexError: index 50001 is out of bounds`

  Cause: Tokenizer vocab size mismatch. The tokenizer produced an ID larger than `Embedding(vocab_size=...)`.

  `RuntimeError: The size of tensor a (512) must match ... (768)`

  Cause: Dimension Mismatch. `Embedding` dim ≠ `PositionalEncoding` dim.

  `ValueError: Sequence length 1024 exceeds maximum`

  Cause: Using Learned PE with input longer than training config.
  Fix: Truncate input or switch to Sinusoidal.

TinyTorch vs. PyTorch

Validating the Abstraction

TinyTorch

```
# Wrapper handles position automatically
embed = EmbeddingLayer(50000, 512)
vectors = embed(tokens)
```

PyTorch

```
# Manual composition required
tok_emb = nn.Embedding(50000, 512)
pos_emb = nn.Embedding(2048, 512)

# User must manually sum
vectors = tok_emb(tokens) +
  pos_emb(positions)
```

Note: PyTorch offers flexible building blocks (`nn.Embedding`). TinyTorch provides the educational wrapper (`EmbeddingLayer`) to show the full system.

Module Summary

What We Built

- **Dense Vectors**
  Translated discrete integers (N) to continuous semantic space ($\mathbb{R}^d$).
- **Efficient Lookup**
  Used NumPy fancy indexing for $O(1)$ retrieval.
- **Position Awareness**
  Solved the "Bag of Words" problem via additive encodings.

Invariants Recap

Input: `(Batch, Seq_Len)` [Integer]
Output: `(Batch, Seq_Len, Embed_Dim)` [Float]

We now have context-free, position-aware vectors ready for processing.

What's Next?

Current State: Isolated Vectors.

Next State: Context-Aware.

Module 12: **Attention**

How do tokens “talk” to each other? We will implement `Scaled Dot-Product Attention`.