FOUNDATION TIER

MODULE 01

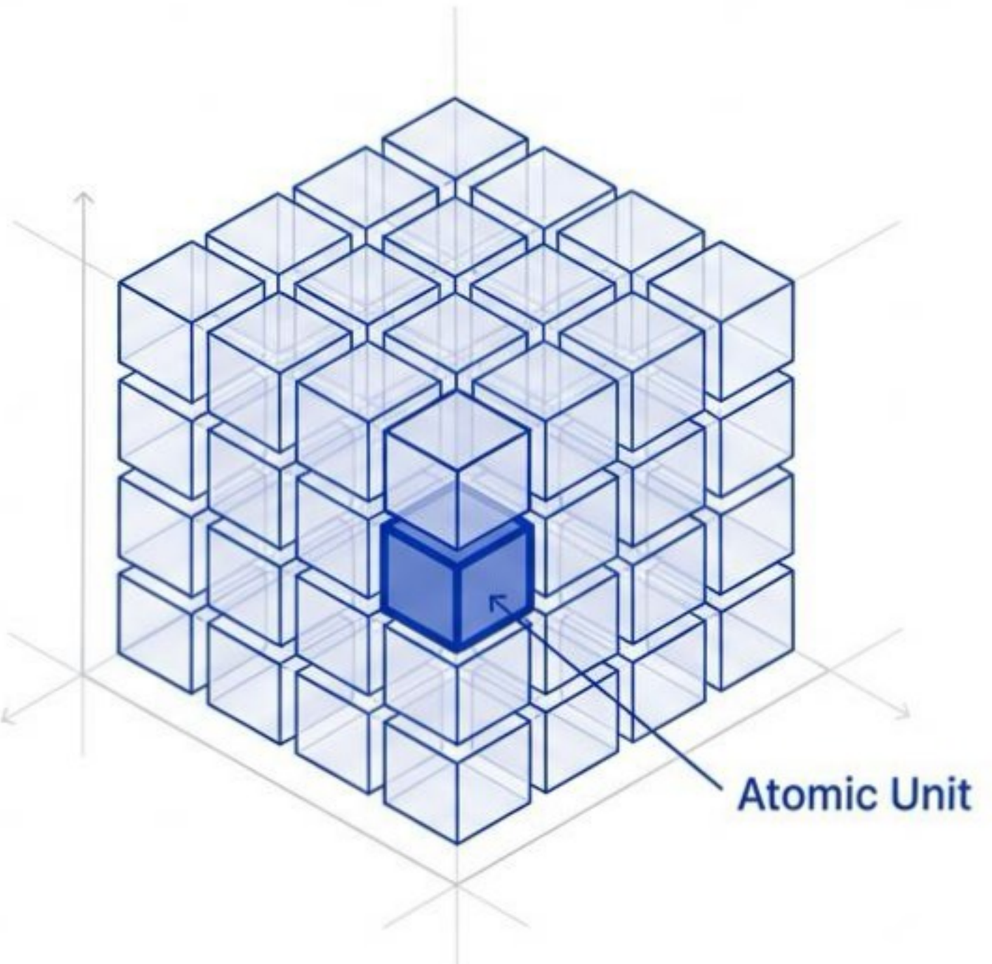
The Tensor Foundation

Building the core data structure that powers modern deep learning

Module 01: Tensor

Foundation Tier

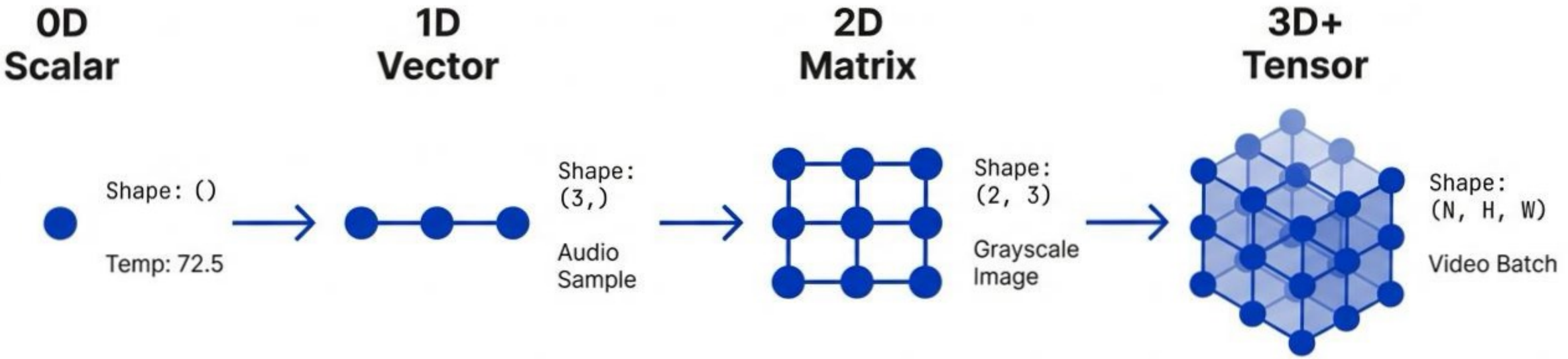
The universal data structure for machine learning. From raw math to computational graphs. This module establishes the base class upon which all subsequent deep learning operations—from activations to autograd—are built.



Prerequisites: Basic Python & Linear Algebra. No ML Framework internals required.

The Universal Container

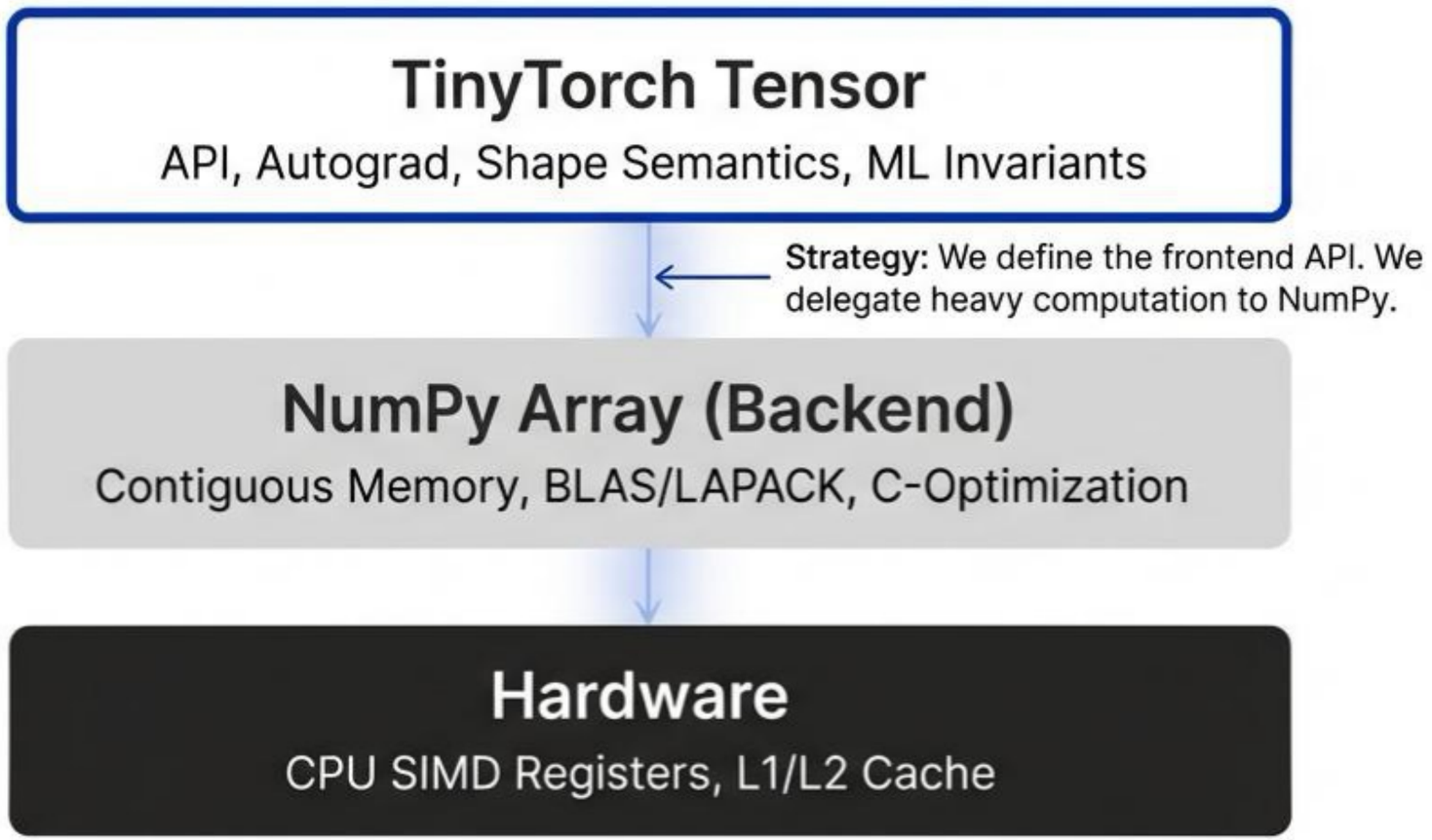
Generalizing the grid to N-dimensions.



A Tensor is a unified interface for all these shapes. Whether representing a single temperature reading or a batch of 4K video frames, the Tensor class handles the arithmetic and memory management uniformly.

The TinyTorch Architecture

A high-level wrapper around optimized C-backends.



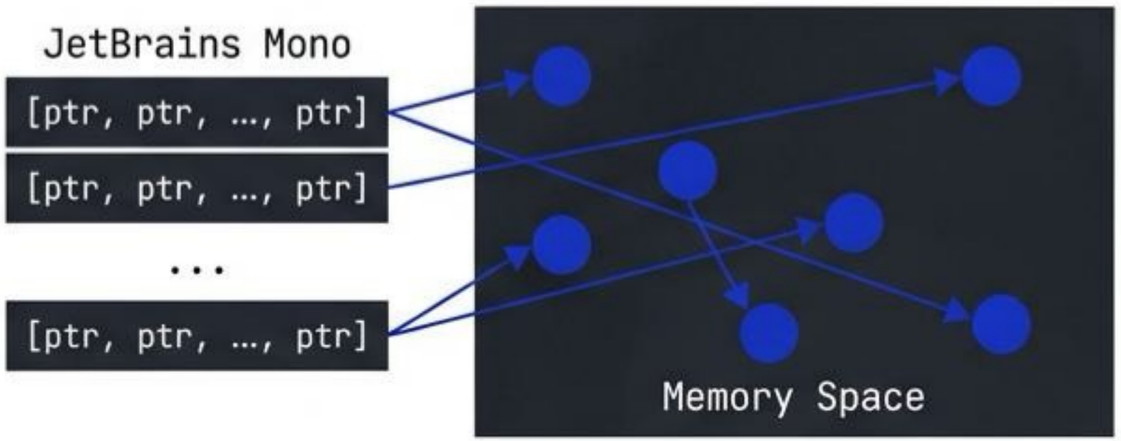
We do not write raw assembly.

We leverage NumPy's battle-tested C implementation for numerical stability and speed, speed, wrapping it to enforce Deep Learning specific rules (like float32 precision and gradient tracking).

Why Not Python Lists?

System Constraint: Memory Locality

Python List [Object, Object, ...]



Pointer Chasing (Slow)

Requires dereferencing every element individually.

Tensor / Array

Contiguous Block (Fast)

Packed bytes (float32). CPU SIMD vectors can process chunks in parallel.

Machine learning requires high throughput. Python lists store pointers to objects scattered in memory, causing cache misses. Tensors enforce contiguous memory layout, enabling vectorized operations that are orders of magnitude faster.

Implementation: The Wrapper

Initializing the Foundation

```
class Tensor:
    def __init__(self, data):
        # Enforce standard ML precision (32-bit floats)
        self.data = np.array(data, dtype=np.float32)

        # Metadata properties
        self.shape = self.data.shape
        self.size = self.data.size
        self.dtype = self.data.dtype
```

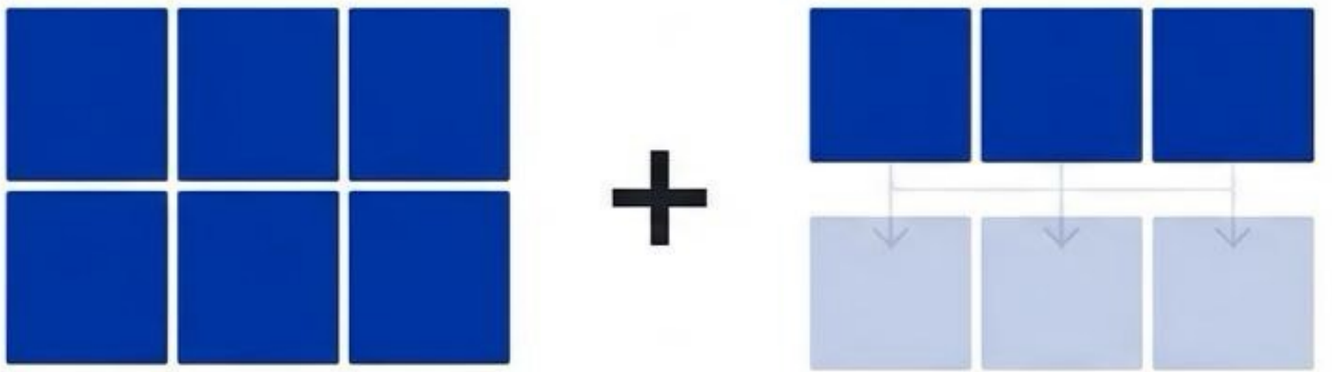
The backend storage (NumPy array).

CRITICAL: Python defaults to float64 (double). We explicitly downcast to float32 to save 50% memory and match GPU standards.

The `Tensor` class is initialized by wrapping raw data. We immediately convert everything to `float32`, the lingua franca of deep learning. This balances numerical precision with memory efficiency.

Broadcasting Semantics

How mismatched shapes align automatically.



Matrix (2, 3) Vector (3,)

`(2, 3) + (3,) → (2, 3)`

The Invariants

1. **Right-to-Left Alignment:** Align shapes starting from the last dimension.
2. **Compatibility Rule:** Dimensions are compatible if they are equal OR if one of them is 1.
3. **Expansion:** Missing dimensions on the left are treated as 1.

Examples

`(32, 512, 768) + (768,)`	✅ Valid (Bias addition)
`(2, 3) + (2,)`	❌ Invalid (3 != 2)

Swiss Engineering Editorial

The Efficiency of Broadcasting

Virtual expansion vs. Physical allocation.

Scenario: Adding a bias vector `(768,)` to a batch of inputs `(32, 512, 768)`.

Option A: Naive Copying

Replicating data 16,384 times in memory.

Slow & Wasteful

Option B: Broadcasting

Reusing the same data pointer.

TinyTorch / NumPy approach

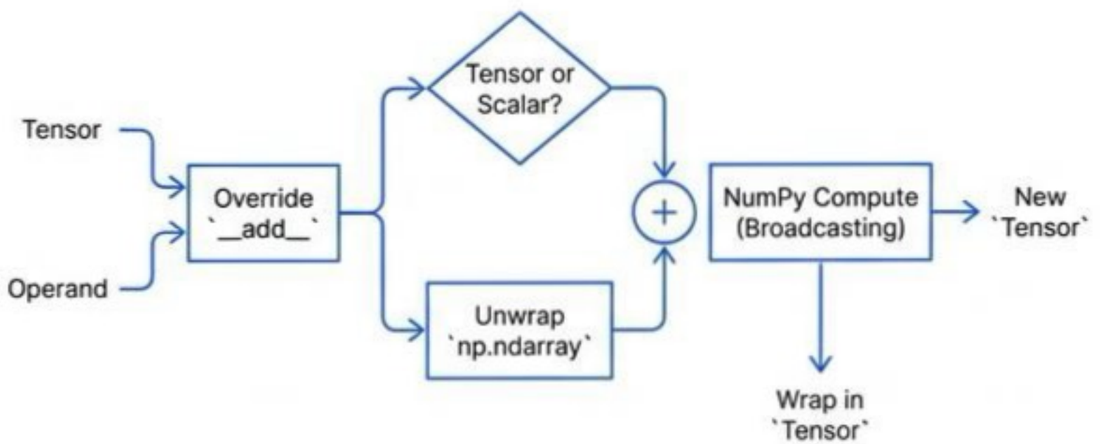
Broadcasting allows us to perform massive operations without massive allocations. We conceptually expand the data without physically copying bytes, saving memory bandwidth and cache space.

Swiss Engineering Editorial

Implementing Arithmetic

Delegating to the backend.

```python
def __add__(self, other):
    """Add two tensors element-wise with broadcasting support."""
    if isinstance(other, Tensor):
        # Unwrap, compute with NumPy, re-wrap
        return Tensor(self.data + other.data)
    else:
        # Scalar broadcast
        return Tensor(self.data + other)
```

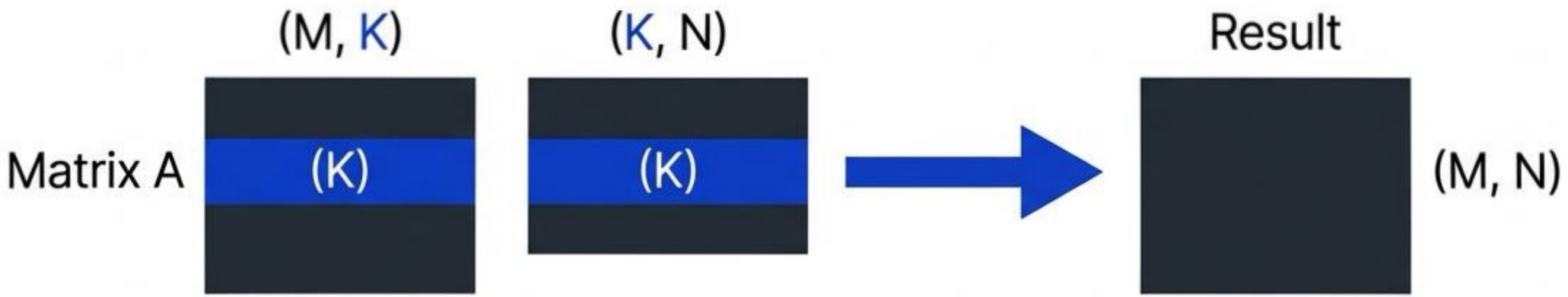


We override Python's magic methods (like `__add__`). The logic is simple:

1. Check if the operand is a Tensor or a Scalar.
2. Unwrap the `np.ndarray` from `self.data`.
3. Let NumPy handle the complex broadcasting rules.
4. Wrap the result back into a new `Tensor`.

The Engine: MatMul

The computational heart of Neural Networks.



Rule: Inner dimensions must match.

`Row (from A) · Column (from B) = Scalar (in Result)`

This operation underpins every Linear layer, Convolution, and Attention mechanism.
A single shape mismatch here breaks the entire network graph.

Swiss Engineering Editorial

Implementing MatMul Validation

Defensive programming for shape errors.

```python
def matmul(self, other):
    if not isinstance(other, Tensor):
        raise TypeError(f"Expected Tensor, got {type(other)}")

    # Critical Shape Validation
    if self.shape[-1] != other.shape[-2]:
        raise ValueError(
            f"Inner dimensions must match: "
            f"{self.shape[-1]} ≠ {other.shape[-2]}"
    )
```


```python
    # Critical Shape Validation
    if self.shape[-1] != other.shape[-2]:
        raise ValueError(
            f"Inner dimensions must match: "
            f"{self.shape[-1]} ≠ {other.shape[-2]}"
    )

    # Dispatch to implementation...
```


Crash early and loudly. We provide a descriptive error message explaining exactly *which* dimensions mismatched, saving hours of debugging time.

Inside the Black Box

The naive $O(n^3)$ implementation.

```
# For 2D matrices (Educational View)

M, K = a.shape
K2, N = b.shape
result = np.zeros((M, N))

for i in range(M):
    for j in range(N):
        # Dot product of row i and column j
        # Cost: K multiplications + K additions
        result[i, j] = np.dot(a[i, :], b[:, j])
```

This nested loop structure reveals the cost. To compute M*N output pixels, we must perform K operations for each one.

Complexity = $O(M \times N \times K)$.

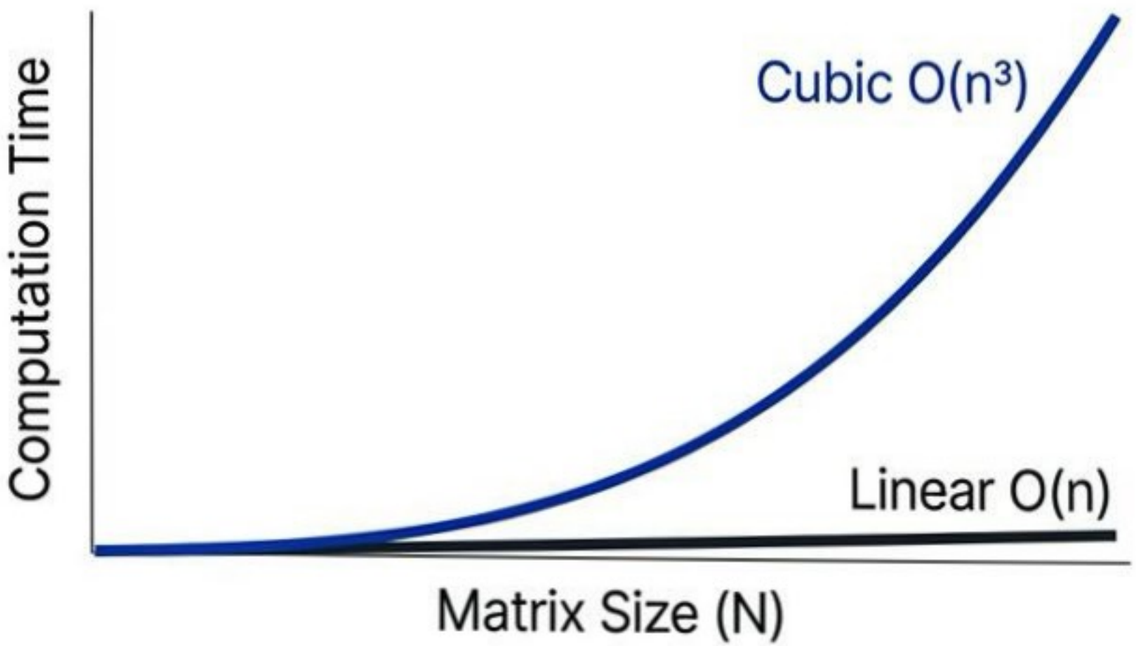
For square matrices of size N, this is **$O(N^3)$**.

Note: In production code, we replace this loop with `np.matmul(a, b)`.

Swiss Engineering Editorial

The O(n³) Reality

Why MatMul dominates training time.



Data Points Callout

1000 x 1000 Matrix Multiply:
~0.1 seconds

2000 x 2000 Matrix Multiply:
~0.8 seconds

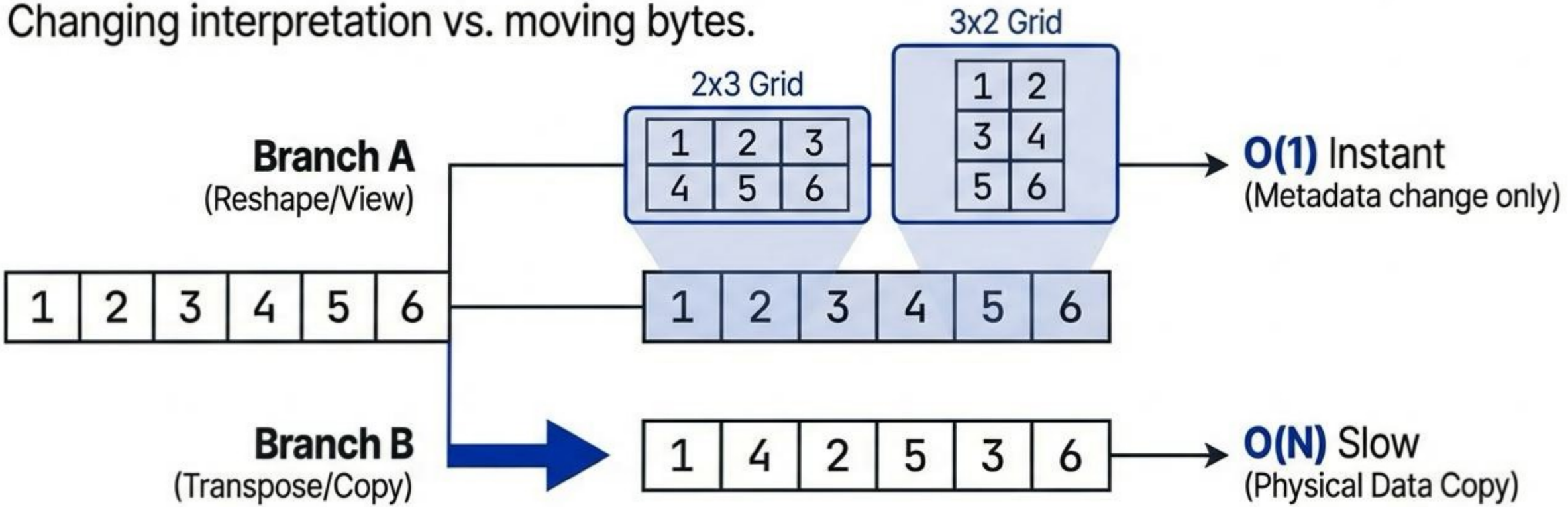
8x increase in time for a 2x increase in size.

Takeaway: Matmul consumes >90% of compute time in modern Transformers. This cubic scaling dictates the need for GPUs.

Swiss Engineering Editorial

Views vs. Copies

Changing interpretation vs. moving bytes.



⚠ Modifying a View modifies the original tensor. Modifying a Copy does not.

Swiss Engineering Editorial

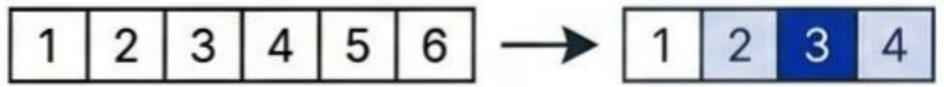
Implementing Reshape

Inference logic for dynamic shapes.

```python
def reshape(self, *shape):
    new_shape = list(shape)

    # Handle -1 inference
    if -1 in new_shape:
        known_size = 1
        for dim in new_shape:
            if dim != -1: known_size *= dim

        # Calculate unknown dimension
        unknown_dim = self.size // known_size
        new_shape[new_shape.index(-1)] = unknown_dim

    return Tensor(np.reshape(self.data, tuple(new_shape)))
```

Explanation

The `-1` argument allows lazy dimension calculation (e.g., `(Batch, -1)`). We calculate the missing dimension by dividing the total size by the product of known dimensions.



Total Elements (6) / Known Dimension (2) = Unknown Dimension (3)

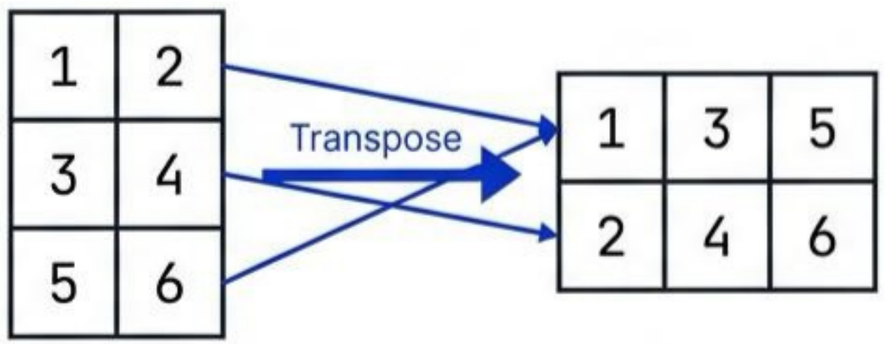
Invariant: 'Total Elements' must remain constant.

Swiss Engineering Editorial

Implementing Transpose

Axis manipulation.

```
def transpose(self, dim0=None, dim1=None):
    # Default: swap last two dims (Matrix Transpose)
    axes = list(range(len(self.shape)))

    if dim0 is None:
        dim0, dim1 = -2, -1

    axes[dim0], axes[dim1] = axes[dim1], axes[dim0]

    return Tensor(np.transpose(self.data, axes))
```



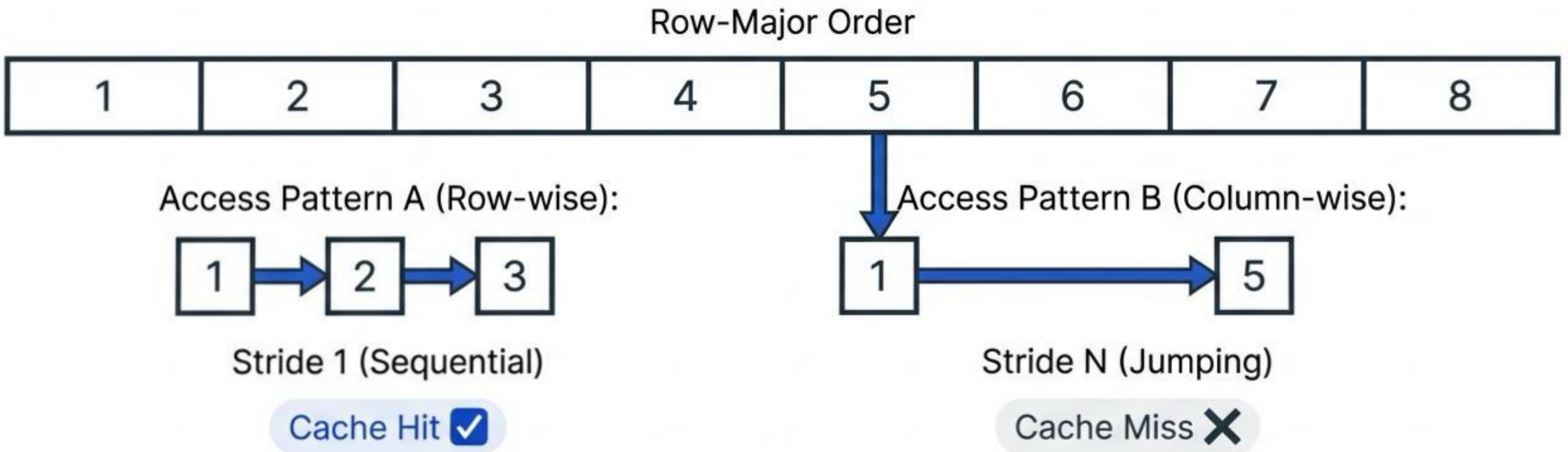
Critically used in Backpropagation (`W.T`) and Attention mechanisms.

⚠ Modifying a View modifies the original tensor. Modifying a Copy does not.

Swiss Engineering Editorial

Memory Layout & Cache Locality

Why layout dictates performance.

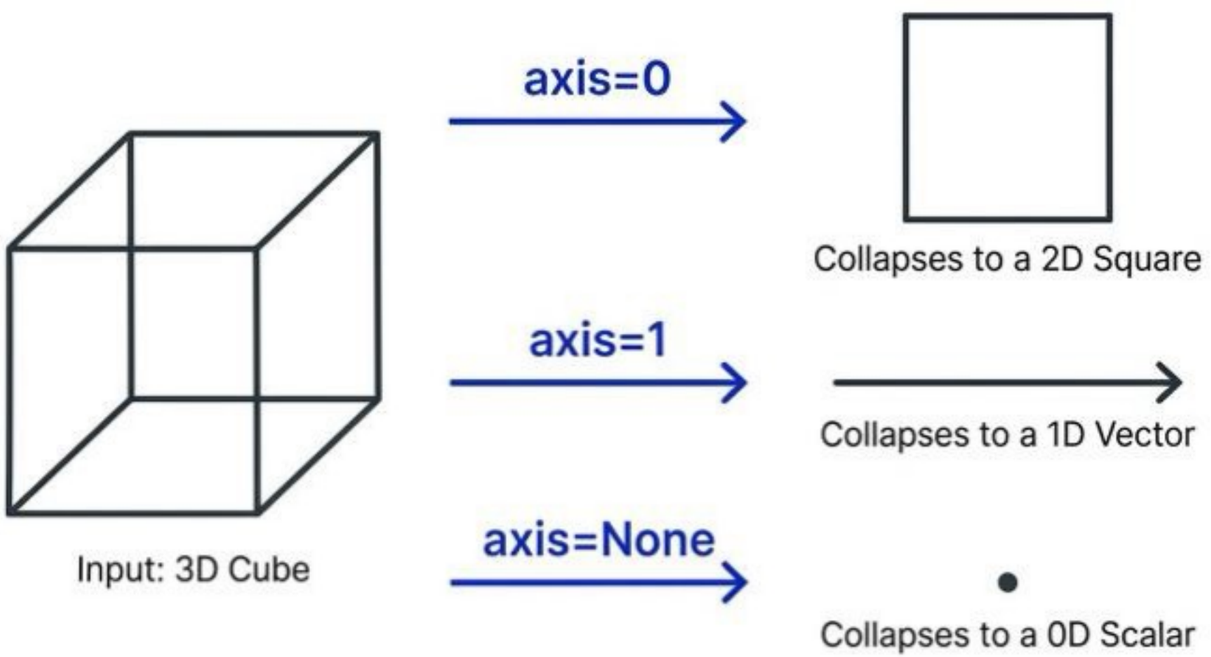


Strided access (columns) can be ~3x slower than sequential access (rows) due to CPU cache behavior. Transposing a large matrix physically disrupts this locality.

Reductions

Collapsing dimensions.

Visual Diagram



Key Parameters List

- `axis`: The dimension to remove.
- `keepdims`: If True, keeps the dimension with size 1 (essential for broadcasting back to original shape).

Implementing Sum and Mean

Aggregating information.

```
def sum(self, axis=None, keepdims=False):
    return Tensor(np.sum(self.data, axis=axis,
        keepdims=keepdims))

def mean(self, axis=None, keepdims=False):
    return Tensor(np.mean(self.data, axis=axis,
        keepdims=keepdims))
```

Applications

- `loss.mean()`: Average error over a batch (Scalar).
- `x.sum(axis=1)`: Summing features across a vector (Vector).

Integration: A Linear Layer

Putting the operators together.

```
# A simple Forward Pass
x = Tensor([[1, 2, 3], [4, 5, 6]]) # Batch of 2
W = Tensor([[0.1, 0.2], ...])      # Weights
b = Tensor([0.1, 0.2])             # Bias

# The Computation
# 1. Matmul (Shape Check & Dot Product)
# 2. Add (Broadcasting Bias)
hidden = x.matmul(W) + b
```

We have successfully replicated the fundamental building block of Deep Learning.

1. `matmul`: Transforms the feature space.
2. `+ b`: Broadcasts the bias across the batch.

Swiss Engineering Editorial

TinyTorch vs. PyTorch

Identical API, Different Engines.

TinyTorch

```
x = Tensor(...); y = x.matmul(w)
```

Backend
NumPy

Speed
1x (Baseline)

PyTorch

```
x = torch.tensor(...); y = x @ w
```

Backend
C++ / CUDA

Speed
100x (**GPU Accelerated**)

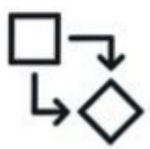
The API, broadcasting rules, and shape semantics are identical. The mental model you build here applies directly to production code.

Concept → Code Map

Module Summary

Universal Data → `Tensor.__init__`
(float32 wrapper)

Alignment → `__add__`
(Broadcasting logic)



Transformation → `matmul`
(Shape checks + Dot product)

Perspective → `reshape`
(Views vs Copies)

Aggregation → `sum, mean`
(Reductions)

What's Next?

From Dead Data to Learning Systems.

We have a Tensor that stores data and performs math. But it is currently "dead"—it cannot learn.

Missing Piece: We need to know how changes in weights (`W`) affect the loss (`L`).

Next Module: Autograd

We will teach this Tensor to track its own history and compute gradients automatically.