Tensor Operations with Einsum and Tensordot: Concepts, Implementation, and Image Blurring Use Case

by **Felix Hirwa Nshuti**

Introduction and Motivation

Working with multi-dimensional arrays (tensors) is a fundamental aspect of scientific computing and machine learning. While most researchers and engineers are more comfortable with high-level matrix operations or simple broadcasting rules, challenges arise when an operation

- Interacts with multiple dimensions simultaneously,
- Involves more than two tensors,
- Requires specific dimension alignments or contractions.
- Or combines several algebraic steps (e.g., transpose, reshape, sum) into one operation.

In such cases, typical solutions include reshaping, transposing, iterating through dimensions, or chaining multiple operations. These approaches often obscure the underlying mathematical intent and introduce unnecessary data movement, leading to inefficiencies.

Einsum (`np.einsum, torch.einsum`) and tensordot (`np.tensordot, torch.tensordot`) solve this problem by allowing users to express complex tensor operations directly in terms of their mathematical definitions. Instead of restructuring tensors to conform to specific operation requirements, users can specify how dimensions should interact, contract, or align. This leads to clearer, more maintainable code that closely mirrors the mathematical expressions being implemented.

The gap in common operations

Standard tensor operations (such as `matmul`, `dot`, `sum`, or element-wise operations) are limited to predefined behavior:

- Matrix multiplication (`matmul`) contracts the last axis of the first tensor with the second axis of the second tensor (following NumPy's convention). Higher-dimensional tensors are treated as batched matrices, and the same contraction rule applies to the trailing axes.
- Dot product (`dot`) contracts the last axis of both tensors. `np.dot` is fully contracted for 1D x 1D, but acts like `matmul` for higher dimensions.
- Reduction operations (like `sum`) eliminate a single axis at a time.
- Element-wise operations requires shapes to be broadcast-compatible.

These operations are insufficient when the algebraic structure of the problem involves arbitrary contractions, multi-tensor interactions, or batched operations on higher-dimensional data.
Example: Matrix multipication is just a special case of tensor contraction.

$$C_{ik} = \sum_j A_{ij} B_{jk}$$

A standard implementation would use `matmul(A, B)`. Which hides the contraction mechanism. Using `einsum`, we can express this more generally as:

$$C = \text{einsum}('ij, jk- > ik', A, B)$$

This notation makes it explicit that we are multiplying over matching indices and summing over repeated ones.

Why this matters

There are several recurring computation patterns where einsum and tensordot shine:

Practical Use Case	**How Einsum Helps**
Batched matrix multiplications	Aligns axes across batches explicitly
Attention mechanism (Transformers)	Clarifies query-key and weight-value relationships
Local filtering (e.g., image convolutions)	Defines kernel application without reshaping or looping
Statistical operations (e.g., covariance, bilinear form)	Captures both multiplication and reduction in a single expression
Tensor compositions in physics simulations	Allows direct implementation of index-based formulas

From matrix view to tensor computation

A matrix multiplication contracts along one axies.
A convolution contracts across spatial and channel axes.
An attention mechanism contracts across query-key and then weights-value axes.

All of these are fundamentally tensor contractions, not matrix multiplications.

Einsum provides a controlled syntactic way to express these arbitrary contractions.
Tensordot generalizes dot products by allowing explicit control over contraction axes.

This document builds intuition for these operations, explains their syntax and semantics, and demonstrates their use with a practical image blurring example.

What this document covers

1. Refresher on tensor shapes, axes, and broadcasting.
2. Tensordot: intuitive generalization of dot product and matrix multiplication.
3. Einsum: flexible notation for arbitrary tensor contractions.
4. Examples (dot poduct, matrix multiplication, batched operations, multi-tensor interactions).
5. Practical example: Implementing image blurring using einsum and tensordot.
6. Performance insights and when to use each operation.

Tensor Shapes, Axes, and Broadcasting Refresher

Tensor Shapes and Axes

A *tensor* is a multi-dimensional array. Its *shape* describes the size along each dimension, and its *rank* is the number of dimensions (axes). For example:

$$\text{shape of } x = (3, 4, 5) \quad \Rightarrow \quad x \text{ is a rank-3 tensor}$$

In tensor programming, the location of a dimension often conveys semantics. For instance, in image processing:

$$\text{batch} \times \text{channels} \times \text{height} \times \text{width} \quad \Rightarrow \quad (B, C, H, W)$$

Correct interpretation and alignment of axes is crucial before performing any contraction or combination operation.

Axis alignment: the key challenge

Many tensor operations assume specific axis positions.

- `matmul(A, B)` assumes the last axis of A contracts with the second-to-last axis of B.
- `dot(A, B)` contracts the last axis of both tensors (for 1D inputs), or behaves similarly to `matmul` for higher dimensions.

This implicit alignment becomes problematic when dimensions are not ordered as expected.
Example: Let $A \in \mathbb{R}^{(4,3,5)}$ and $B \in \mathbb{R}^{(6,5)}$. And we want to compute the matrix multiplication along the last axis of A and the second axis of B to produce a new tensor C of shape (4, 3, 6). Even though both tensors contain a dimension of size 5, they do not align correctly:

```
A @ B # ValueError: matmul: Input operand 1 has a mismatch in its core
      dimension 0, with gufunc signature (n?,k),(k,m?)->(n?,m?) (size 6 is
      different from 5)
```

Analysis of the failure: In the example above, `matmul` (or the `@` operator) treats A as a batch of matrices with shape $(3, 5)$ and B as a matrix of shape $(6, 5)$. It attempts to perform matrix multiplication:

$$(3, \mathbf{5}) \times (6, 5)$$

The inner dimensions **5** and 6 do not match, causing the error. To fix this with standard operators, one would have to transpose B to $(5, 6)$ so the alignment becomes $(3, \mathbf{5}) \times (\mathbf{5}, 6)$. This manual transposition is exactly the kind of 'noise' that `tensordot` and `einsum` eliminate.

Broadcasting

Before diving into contractions, we must recall how NumPy and PyTorch handle dimensions of different sizes that are *not* being contracted. This is **broadcasting**.
When operating on two tensors, if their shapes differ, the smaller shape is 'right-aligned' with the larger shape. Two dimensions are compatible when:

1. They are equal, or
2. One of them is 1.

Visualizing the stretch: If a dimension is 1, the tensor is implicitly 'stretched' or copied along that dimension to match the other tensor.

$$\text{A} : (4, 1, 6)$$
$$\text{B} : (3, 6)$$

Step 1: Right-align

$$\text{A} : (4, 1, 6)$$
$$\text{B} : (\ \ 3, 6)$$

Step 2: Check compatibility

- Trailing axis: $6 == 6$ (Compatible)
- Middle axis: 1 vs 3 (Compatible, A is broadcasted)
- Leading axis: 4 vs (missing) (Compatible, B is effectively broadcasted/repeated if necessary, though conceptually it just implies batching)

Resulting Shape: $(4, 3, 6)$
Relevance to Einsum: Broadcasting is automatic handling of 'free indices.' If an index appears in the input but is **not** summed over, it must follow broadcasting rules. Understanding this prevents shape mismatch errors when performing batched contractions.

The Logic of Contraction

To use `tensordot` and `einsum` effectively, we must categorize the axes (dimensions) of our tensors into two types during an operation:

1. Summation Indices (Contracted Axes)

These are the dimensions that are consumed by the operation. In the mathematical expression, these indices appear in the summation formula.

2. Free Indices (Batch/Outer Axes)

These are the dimensions that are preserved in the output. They determine the shape of the result.

$$C_{\mathbf{ik}} = \sum_j A_{\mathbf{i}j} B_{j\mathbf{k}}$$

Here, j is the summation index. It exists in both inputs but disappears in the output. The sizes of these dimensions must match exactly. And i and k are free indices. The output tensor C will have shape based on i (from A) and k (from B).
Key Intuition:

- **Tensordot** asks: "Which axes correspond to the Summation Indices"? (You define the contraction, everything else is Free).
- **Einsum** asks: "Please list down every index for inputs and output". (You explicitly label both Summation and Free indices).

Tensordot: Explicit Axis Contraction

`tensordot` is the bridge between standard matrix multiplication and full tensor contraction. Unlike `matmul`, which uses heuristic rules to guess which axes to align, `tensordot` requires you to explicitly specify the **Summation Indices**.
The signature is:

```
result = np.tensordot(a, b, axes=x)
```

The `axes` parameter controls the contraction logic. It accepts two formats:

1. Integer Argument (The 'Default' Contract)

If `axes=N` (where N is an integer), it contracts the **last** N axes of a with the **first** N axes of b.

- `axes=0`: Tensor outer product (no contraction).
- `axes=1`: Standard dot product or matrix multiplication (contracts last of a with first of b).
- `axes=2`: Contracts last two of a with first two of b (common in 2D convolutions).

2. Sequence Argument (The 'Explicit' Alignment)

This is the most powerful feature for solving alignment issues. You pass a tuple of two lists: `axes=([a_axes], [b_axes])`.

- `a_axes`: The list of indices in a to sum over.
- `b_axes`: The list of indices in b to sum over.

The operation proceeds by 'zipping' these axes together. The dimensions at these specified indices must have matching sizes.

Solving the Alignment Problem

Recall our motivating failure:

$$A \in \mathbb{R}^{(4,3,\mathbf{5})}, \quad B \in \mathbb{R}^{(6,\mathbf{5})}$$

We want to contract the dimension of size 5.

- In A, the dimension of size 5 is at index **2**.
- In B, the dimension of size 5 is at index **1**.

Using `tensordot`, we explicitly align axis 2 of A with axis 1 of B:

```
import numpy as np

A = np.random.randn(4, 3, 5) # Shape : (4, 3, 5)
B = np.random.randn(6,5) # Shape : (6, 5)

# Contract A's axis 2 with B's axis 1
C = np.tensordot(A, B, axes=([2], [1]))
```

Calculating the Output Shape: The output shape is formed by concatenating the **Free Indices** of A followed by the **Free Indices** of B.

1. Remove the contracted axes:
$$A : (4, 3, \cancel{5}) \to (4, 3)$$
$$B : (6, \cancel{5}) \to (6)$$

2. Concatenate the remaining axes:
$$(4, 3) + (6) \Rightarrow (4, 3, 6)$$

The result is a tensor of shape $(4, 3, 6)$. The "noise" of manual transposition is gone; we simply pointed to the axes that matched.

Einsum: The Domain Specific Language for Tensors

While `tensordot` solves the alignment problem, it requires you to mentally calculate axis indices (e.g., "axis 2 matches axis 1"). This becomes brittle if the tensor rank changes.
Einsum (Einstein Summation) solves this by using labels instead of indices. You name the axes, and the library figures out the alignment.

The Syntax

The function signature is:

```
# 'equation_string', operand_A, operand_B, ...
result = np.einsum('subscripts->output', A, B)
```

The `equation_string` is split into two parts by the arrow `->`:

1. **LHS (Left of Arrow):** Labels the axes of the input tensors. Comma `,` separates operands.
2. **RHS (Right of Arrow):** Labels the axes of the desired output tensor.

The Three Rules of Einsum

To read or write any einsum string, apply these three rules:

- **Rule 1 Alignment (Repeated index across inputs):** If an index appears in more than one input tensor, the corresponding axes are **aligned** (multiplied element-wise).
- **Rule 2 Summation (Index missing from output):** If an index appears in the input(s) but is **not** present in the output expression, that axis is **summed over** that axis.
- **Rule 3 Free index (Index present in output):** If an index appears in the input and is also present in the output expression, that axis is **preserved** in the result (no summation).

Solving the Alignment Problem

Let's revisit our recurring example:
$$A \in \mathbb{R}^{(4,3,5)}, \quad B \in \mathbb{R}^{(6,5)}$$

We want to contract the dimension of size 5.
Step 1: Label the axes

- Tensor A has 3 axes. Let's call them `i, j, k`. (Size: 4, 3, 5)
- Tensor B has 2 axes. Let's call them `l, m`. (Size: 6, 5)

Step 2: Identify alignments We know the last axis of A (size 5, label `k`) must match the last axis of B (size 5, label `m`). In einsum, we force alignment by giving them the **same label**. Let's use `k` for both.

$$A : \texttt{ijk}, \quad B : \texttt{lk}$$

Step 3: Define the output We want to sum over `k` (the shared axis). Therefore, `k` must **not** appear in the output. The remaining indices are `i, j, l`.

$$\texttt{output} : \texttt{ijl}$$

The Code:

```
# 'ijk' matches A (4,3,5)
# 'lk'  matches B (6,5) -> notice k is aligned to axis 1
# '->ijl' means we keep i,j,l and sum over k
C = np.einsum('ijk,lk->ijl', A, B)
```

This produces the same $(4, 3, 6)$ result as `tensordot`, but the intent is explicit: "Align axis k, sum over it, and keep i, j, l".

Why explicit mode is safer

It is possible to use einsum without the arrow (implicit mode), e.g., `einsum('ij,jk', A, B)`. However, explicit mode (`->`) is recommended for two reasons:

1. It forces you to consciously decide the order of output dimensions (preventing accidental transposes).
2. It allows you to sum over an axis without contracting it against another tensor (e.g., `'ijk->ij'` is just a sum reduction over k).

Common Patterns: Examples of Einsum and Tensordot

To build fluency, it helps to see standard operations translated into `einsum` and `tensordot`.

1. Dot Product (Vector-Vector)

Both inputs are 1D vectors. We contract the only available axis.

$$c = \sum_i a_i b_i$$

```
a = np.random.rand(5)
b = np.random.rand(5)

# Standard
res = np.dot(a, b)

# Einsum: 'i' appears in both -> multiply. Missing in output -> sum.
res = np.einsum('i,i->', a, b)

# Tensordot: Contract 1 axis
res = np.tensordot(a, b, axes=1)
```

2. Matrix Multiplication

Standard row-by-column multiplication.

$$C_{ik} = \sum_j A_{ij} B_{jk}$$

```
A = np.random.rand(3, 5)
B = np.random.rand(5, 4)

# Standard
res = A @ B

# Einsum: sum over inner dimension 'j'
res = np.einsum('ij,jk->ik', A, B)

# Tensordot: contract last of A, first of B
res = np.tensordot(A, B, axes=1)
```

3. Outer Product

Compute the product of every element in a with every element in b, resulting in a matrix. No summation occurs.

$$C_{ij} = a_i b_j$$

```
a = np.random.rand(3)
b = np.random.rand(4)

# Standard
res = np.outer(a, b)

# Einsum: distinct indices 'i' and 'j', both kept in output
res = np.einsum('i,j->ij', a, b)

# Tensordot: axes=0 means no contraction (outer product)
res = np.tensordot(a, b, axes=0)
```

4. Batch Matrix Multiplication

This is where `einsum` shines. Suppose we have a batch of matrices (e.g., 10 matrices of size 3x5) and we want to multiply them by another batch.

$$C_{bik} = \sum_j A_{bij} B_{bjk}$$

Standard libraries often require specific functions like `torch.bmm`. Einsum handles it naturally by adding a "batch" index.

```
A = np.random.rand(10, 3, 5)
B = np.random.rand(10, 5, 4)

# Einsum: 'b' is a free index (shared but not summed)
# 'j' is the summation index
```

```
res = np.einsum('bij,bjk->bik', A, B)

# Tensordot cannot do this easily in one step because
# it sums over all shared axes by default.
```

5. Transpose and Permutation

Einsum can reorder axes without any multiplication.

$$B_{ji} = A_{ij}$$

```
A = np.random.rand(3, 5)

# Standard
res = A.T

# Einsum: swap 'i' and 'j' in the output
res = np.einsum('ij->ji', A)
```

6. Trace and Diagonal

Summing the diagonal elements of a matrix.

$$\text{tr}(A) = \sum_i A_{ii}$$

```
A = np.random.rand(5, 5)

# Standard
res = np.trace(A)

# Einsum: repeat index 'i' on input implies diagonal
# Omitting 'i' on output implies summation
res = np.einsum('ii->', A)

# Get Diagonal (no sum)
diag = np.einsum('ii->i', A)
```

Practical Application: Channel Mixing in Deep Learning

Let's apply these concepts to a common task in computer vision and deep learning: applying a linear transformation to image channels (a "1x1 convolution").

The Scenario

We have a batch of RGB images. We want to convert these 3 color channels into 16 feature channels using a weight matrix.

The Data Shapes (using standard NumPy/TensorFlow convention):

- **Input Images (I):** Shape (B, H, W, C_{in}). Let's use (32 images, 64 height, 64 width, 3 RGB).

- **Weights Matrix (W):** Shape (C_{in}, C_{out}). Let's use $(3 \to 16)$.

The Goal: For every single pixel (b, h, w) across the entire batch, we want to take its 3-element color vector and multiply it by the 3×16 weight matrix to get a new 16-element vector. The target output shape is $(32, 64, 64, 16)$.
The mathematical operation for a single pixel's output channel k:

$$\text{Output}_{b,h,w,k} = \sum_{c=1}^{3} I_{b,h,w,c} \cdot W_{c,k}$$

We must contract over the C_{in} (size 3) axis.

The "Standard" Approach (and its limits)

A naive attempt using standard matrix multiplication might look like this:

```
# I shape: (32, 64, 64, 3)
# W shape: (3, 16)
# Result = I @ W
```

In NumPy, this actually works because `matmul` treats the leading dimensions $(32, 64, 64)$ as a giant batch. It contracts the last axis of I (3) with the first of W (3).
But what if the data format changes? In PyTorch, image data is often ordered "channels-first": (B, C_{in}, H, W).

```
# I_torch shape: (32, 3, 64, 64)
# W shape: (3, 16)
# Result = I_torch @ W
```

This fails. `matmul` tries to contract the last axis of I (64) with the first of W (3). A shape mismatch error occurs. You would need to transpose `I_torch` repeatedly to make `matmul` work, obscuring the code's intent.

The Tensordot Solution (Robust)

`tensordot` doesn't guess. We explicitly tell it where the channel axes are. Let's use the channels-first example where `matmul` failed.

$$I : (32, \mathbf{3}, 64, 64) \quad W : (\mathbf{3}, 16)$$

The dimension to contract (size 3) is at axis 1 for I, and axis 0 for W.

```
# I shape: (32, 3, 64, 64)
# W shape: (3, 16)
# Contract I's axis 1 with W's axis 0
output = np.tensordot(I, W, axes=([1], [0]))
```

Analyzing the Output Shape: `tensordot` appends the free axes of the second input to the free axes of the first.

- Free axes of I: $(32, 64, 64)$
- Free axes of W: (16)
- Result shape: $(32, 64, 64, 16)$

It works robustly, but note that the output is now "channels-last", despite the input being "channels-first". We might need another transpose to fix the layout.

The Einsum Solution (Elegant)

Einsum handles the contraction and the resulting layout in a single step by allowing us to define the output order explicitly.

```
# I shape: (B, C_in, H, W) -> label 'bchw'
# W shape: (C_in, C_out)  -> label 'ck' (k for new channels)

# We want to contract 'c'.
# We want the output to remain channels-first: (B, C_out, H, W) -> 'bkhw'

output = np.einsum('bchw,ck->bkhw', I, W)
```

This single line is a complete description of the operation.

- `b, h, w`: Free indices (batch and spatial dimensions are preserved).
- `c`: The summation index (connects input channels to weights).
- `k`: The new free index (the resulting feature channels).
- `->bkhw`: Ensures the output is precisely in the desired PyTorch format.

By changing the string to `'bchw,ck->bhwk'`, you could instantly output TensorFlow format instead. This flexibility is why 'einsum' is preferred for complex multi-dimensional operations.

Advanced Case: Spatial Convolution (Blurring)

You might ask: "How do we apply a spatial filter, like a blur kernel?" Unlike channel mixing, spatial convolution requires mixing pixels with their neighbors.

The Limitation of Pure Einsum

`einsum` calculates products based on *indices*. It cannot easily express "index i times index $i+1$" (sliding windows) inside the string. To perform a 3×3 convolution (blur) using `einsum`, we first use a technique called **im2col** (or Unfold).

The Strategy: Unfold $\rightarrow$ Contract

We view the local neighborhoods as a new dimension.

1. **Unfold:** Transform the image from (B, C, H, W) to a shape that exposes patches, e.g., $(B, C, \text{patches}, K, K)$.
2. **Contract:** Use `einsum` to multiply patches by the kernel.

```
import numpy as np
from numpy.lib.stride_tricks import sliding_window_view

# Input: (1, 1, 5, 5) Image (Batch, Channel, H, W)
# Kernel: (3, 3) Blur filter
input_img = np.random.rand(1, 1, 5, 5)
kernel = np.ones((3, 3)) / 9.0  # Simple average blur

# 1. Create Sliding Windows (The "Unfold" step)
# We want windows of size 3x3 sliding across axes 2 (H) and 3 (W).
```

```
# This creates a "view" of the array without copying data.
# Output Shape: (Batch, Channel, H_out, W_out, 3, 3)
windows = sliding_window_view(input_img, window_shape=(3, 3), axis=(2, 3))

# 2. Einsum (The Blurring Operation)
# 'bchwxy': batch, channel, h_out, w_out, kernel_x, kernel_y
# 'xy': kernel dimensions (3, 3)
# '->bchw': output image (Valid padding)
blurred = np.einsum('bchwxy,xy->bchw', windows, kernel)
```

This pattern-restructuring memory so that a complex sliding operation becomes a simple dot product-is fundamental to how libraries like cuDNN and TVM optimize operations. `einsum` handles the contraction logic once the memory layout is prepared.

The "Pain" of the Standard Approach

To perform this same operation using standard matrix multiplication (or `np.dot`), we are forced to flatten the spatial structure. The intuition of "height times width" is lost to satisfy the requirements of linear algebra libraries.

```
# --- The Matmul/Dot Way ---
# We have `windows` of shape (B, C, H_out, W_out, 3, 3)
# We have `kernel` of shape (3, 3)

# Step 1: Flatten the kernel to a vector
# Shape: (9,)
k_flat = kernel.flatten()

# Step 2: Reshape the windows to merge the last two dimensions
# We need the last axis to match the size of k_flat (9).
# Shape: (B, C, H_out, W_out, 9)
w_flat = windows.reshape(*windows.shape[:-2], -1)

# Step 3: Perform the contraction
# Contracts the last axis of w_flat with k_flat
result_dot = w_flat @ k_flat

# Comparison:
# Matmul requires manual reshaping and mental tracking of flattened dimensions.
# Einsum ('bchwxy,xy->bchw') handles the 3x3 block naturally without
    flattening.
```

In this case, `einsum` acts as a self-documenting tool. It says: "Multiply the window axes x, y with the kernel axes x, y." The standard approach says: "Reshape these tensors into vectors and hope the indices align."

Performance Insights and Best Practices

While `einsum` is syntactically superior for complex operations, understanding its performance characteristics is vital for writing efficient systems code.

1. The Memory Advantage (Fusion)

A major benefit of `einsum` is that it performs the contraction in a single pass, avoiding large intermediate allocations.

Example: Element-wise multiplication followed by sum

$$\text{res} = \sum_{i,j} A_{ij} \cdot B_{ij}$$

```python
# Approach 1: Naive NumPy
# Creates a temporary array (A * B) matching the size of A.
# High memory footprint for large tensors.
res = (A * B).sum()

# Approach 2: Einsum
# Multiplies and sums in a single loop. No intermediate array created.
res = np.einsum('ij,ij->', A, B)
```

2. The Speed Trade-off (BLAS vs. C-Loops)

In pure NumPy:

- `np.matmul` and `np.tensordot` typically dispatch directly to highly optimized BLAS routines (like OpenBLAS or MKL). These are vectorized and cache-friendly.
- `np.einsum` parses the string and executes C-loops. While optimized, it can sometimes be slower than a direct BLAS call for simple matrix multiplications.

Note on Deep Learning Frameworks (PyTorch/TF/JAX): Modern compilers often optimize `einsum` calls by generating custom CUDA kernels or mapping them to the most efficient underlying cuBLAS/cuDNN calls. Therefore, the performance penalty is often negligible in frameworks like PyTorch compared to pure NumPy.

3. Contraction Paths (The Order Matters)

When contracting three or more tensors, the order of operations drastically affects complexity.

$$D = A \times B \times C$$

Is it faster to do $(A \times B) \times C$ or $A \times (B \times C)$?

```python
# NumPy's einsum can optimize this path automatically.
# 'optimize=True' (default in recent versions) calculates the
# optimal contraction order (Greedy algorithm) before execution.
res = np.einsum('ij,jk,kl->il', A, B, C, optimize=True)
```

Summary: When to use what?

Operation	Recommendation
Standard Matrix Mult	Use `@` or `matmul`. It is universally understood.
Specific Axis Contraction	Use `tensordot`. It is explicit and extremely fast.
Batched Operations	Use `einsum`. It avoids complex reshapes.
Complex Reductions	Use `einsum`. It is the most readable way to express dimensions.

Conclusion

Mastering axis alignment is the threshold between writing basic scripts and developing robust, dimensional-agnostic machine learning systems.
We started by identifying the limitations of standard operators like `matmul` and `dot`: they rely on implicit conventions that often fail when tensors grow beyond two dimensions. We then introduced two powerful alternatives:

1. **Tensordot:** The mechanic's tool. It offers precise, index-based control over contraction, making it ideal for programmable generation of operations or when you need raw speed in loop-heavy environments.
2. **Einsum:** The mathematician's tool. It provides a readable, self-documenting syntax that mirrors the algebraic notation found in research papers. It handles permutation, reduction, and contraction in a single step.

By treating data not just as lists of numbers but as tensors with semantic axes (batch, channel, spatial), you can write code that is both more concise and less prone to "silent" shape errors. Whether you are implementing a new attention mechanism or optimizing a convolution kernel, the ability to explicitly align axes is a superpower in scientific computing.