# Tensor Operations: einsum & tensordot

Powerful Functions for Multi-dimensional Array Manipulation

# Introduction and Mathematical Foundation

## What are `einsum` and `tensordot`?

- Two powerful functions for tensor operations in NumPy and PyTorch.
- They provide concise ways to perform dot products, outer products, transpositions, matrix math, and more complex tensor contractions.

## Mathematical Foundation

### Einstein Summation Convention

The Einstein summation convention implies summation over repeated indices. For example, $C_{ik}=A_{ij}B_{jk}$ represents: $C_{ik}=\sum_j A_{ij}B_{jk}$ This convention simplifies tensor notation by removing explicit summation symbols.

### Tensor Contraction

Tensor contraction generalizes matrix multiplication to higher-dimensional arrays. It involves element-wise multiplication followed by summation along specific dimensions.

# Einsum - Core Concepts

## Einstein Summation (`einsum`)

- A versatile function in NumPy and PyTorch using a special string notation.
- **Basic Idea**: `einsum('input_specs -> output_spec', tensor1, tensor2, ...)`
  - Indices repeated in inputs but not in the output are summed out (contracted).
  - Indices appearing in an input but not the output are also summed out.
  - The order of indices in `output_spec` defines the final tensor's axis order.
  - If `-> output_spec` is missing, `einsum` infers it from unsummed input indices, ordered alphabetically.

**Syntax:**

numpy.einsum(subscripts, *operands, ...)

torch.einsum(equation, *operands)

# Einsum - Vector Operations

## `einsum` for Vector Operations

Let u=[1,2,3] and v=[4,5,6].

- **Element-wise Product (Hadamard Product)**: $w_i = u_i \cdot v_i$
  - `einsum('i,i->i', u, v)`
  - Result: `[4, 10, 18]`
- **Dot Product (Inner Product)**: $s = \sum_i u_i v_i$
  - `einsum('i,i->', u, v)` or `einsum('i,i', u, v)`
  - Result: `32`
- **Outer Product**: $M_{ij} = u_i v_j$
  - `einsum('i,j->ij', u, v)`

Result:
[[ 4,  5,  6],

[ 8, 10, 12],

[12, 15, 18]]

- **Sum of Elements**: $s = \sum_i u_i$
  - `einsum('i->', u)`
  - Result: `6`

# einsum - Basic Matrix Operations

## `einsum` for Matrix Operations

Let A be (2×3) and B be (3×2).

- **Matrix Transpose**: MjiT=Mij
  - `einsum('ij->ji', A_mat)`
- **Matrix-Vector Product**: yi=∑jMijuj
  - `einsum('ij,j->i', A_mat, u_vec_ops)`
- **Vector-Matrix Product**: zj=∑iwiMij
  - `einsum('i,ij->j', w_matvec, C_mat)`
- **Matrix Multiplication**: Pik=∑j(M1)ij(M2)jk
  - `einsum('ij,jk->ik', A_mat, B_mat)` or `einsum('ij,jk', A_mat, B_mat)`

# einsum - Advanced Matrix Operations

## Advanced Matrix Operations with `einsum`

- **Matrix Multiplication with Reduction**:
    - Sum columns of P: `einsum('ij,jk->i', M1, M2)`
    - Sum rows of P: `einsum('ij,jk->k', M1, M2)`
    - Sum all elements of P: `einsum('ij,jk->', M1, M2)`
- **Trace of a Matrix**: $tr(M)=\sum iMii$
    - `einsum('ii->', C_mat)`
- **Diagonal Extraction**: $di=Mii$
    - `einsum('ii->i', C_mat)`
- **Frobenius Norm (Squared)**: $||A||F2=\sum i\sum jAij2$
    - `einsum('ij,ij->', A_mat, A_mat)`

# einsum - Axis Operations & Hadamard

## Axis Operations & Hadamard Product with `einsum`

- **Sum along an Axis**:
  - Sum over columns (for each row): `einsum('ij->i', A_mat)`
  - Sum over rows (for each column): `einsum('ij->j', A_mat)`
- **Element-wise Matrix Product (Hadamard)**: Pij=(M1)ij(M2)ij
  - `einsum('ij,ij->ij', C_mat, D_mat)`

# einsum - Chained & Tensor Operations

## Chained Operations and Higher-Rank Tensors with `einsum`

- **Chained Operations**: `einsum` can combine multiple tensor operations efficiently.
  - Example: (M1M2)w: `einsum('ij,jk,k->i', M1, M2, w)`
  - Example: M1M2M3: `einsum('ij,jk,kl->il', M1, M2, M3)`
- **Tensor Operations (Rank 3+)**: `einsum` excels here.
  - **Ellipsis (`...`) Operator**: Placeholder for leading batch dimensions.
    - Example: Batch matrix multiplication `einsum('...ij,...jk->...ik', T_A, T_B)` for `T_A` and `T_B` with shared batch dimensions.
  - **Permutation of Axes**: Reorders tensor dimensions.
    - Example: Tijk→Tkij′: `einsum('ijk->kij', Tensor_permute)`

# tensordot - Introduction & Examples

## `tensordot`

- Performs tensor contractions by summing over specified axes.
- Available in NumPy and PyTorch.

**Syntax:**

numpy.tensordot(a, b, axes)

torch.tensordot(a, b, dims)

**`axes` / `dims` Parameter is Key:**

- **Integer `N`**: Sums over the last `N` axes of tensor `a` and the first `N` axes of `b`.
  - Example: `torch.tensordot(A_ex3, B_ex3, dims=2)` where `A_ex3` ends with matching dimensions to `B_ex3`'s start (e.g., `A(...,4,5), B(4,5,...)`). Result shape: `(2,3,6,7)`.
- **Tuple of two lists `([a_axes_list], [b_axes_list])`**:
  - Sums over specified axes: `a_axes_list[k]` is contracted with `b_axes_list[k]`.
  - Resulting tensor has `a`'s remaining axes, then `b`'s remaining axes, in that order.
  - Example: `torch.tensordot(A_td_torch, B_td_torch, dims=([0,3], [0,2]))` (contract `A[0]` with `B[0]`, `A[3]` with `B[2]`). Result shape: `(3,4,3,6)`.

# Practical Applications

### 1. Batch Processing in Neural Networks (Convolution-like Operation)

`einsum` can express parts of convolutions.

```
# Simulate batch of images and a convolutional kernel

# Input patch: bchw (batch, channels_in, kernel_h, kernel_w)

# Kernel: ochw (kernel_out_channels, channels_in, kernel_h, kernel_w)

conv_like_op = np.einsum('bcij,ocij->boij', image_patch, kernel_conv)

# Output: bohw (batch, kernel_out_channels, kernel_h, kernel_w)
```

# Practical Applications - Cont.

**2. Attention Mechanisms in Transformers**

`einsum` is crucial for attention scores and weighted sums.

```
# Compute attention scores: Q @ K^T

# Q: bqd, K: bkd -> Scores: bqk

attention_scores = np.einsum('bqd,bkd->bqk', Q_att, K_att)

# Apply attention to values: weights @ V

# weights: bqk, V: bkd -> Output: bqd

attention_output = np.einsum('bqk,bkd->bqd', attention_weights_simplified, V_att)
```

# Practical Applications - Cont.

**3. Statistical Operations (Covariance Matrix)**

`einsum` simplifies statistical computations.

```
# Covariance matrix: (1/(N-1)) * X_centered^T @ X_centered

# Data_centered: sn (samples, features)

cov_matrix_einsum = np.einsum('sn,sm->nm', data_centered, data_centered) / (num_samples - 1)

# Output: nm (features, features)
```

# Performance & Best Practices

## Performance Tips and Best Practices

### 1. Memory Layout and Performance Considerations

For very large arrays, memory layout (Fortran vs. C order) can influence performance. `einsum` operations are generally efficient, but specific performance can vary.

### 2. `optimize=True` Parameter in `numpy.einsum`

NumPy's `einsum` has an `optimize` parameter to significantly improve performance for complex operations (three or more operands) by finding an optimal contraction order.

```
# Complex multi-tensor operation: 'ijk,jkl,klm->ilm'

# Use optimize=True for potential speedups:

np.einsum('ijk,jkl,klm->ilm', A_complex_perf, B_complex_perf, C_complex_perf, optimize=True)
```

# Performance & Best Practices - Cont.

### 3. Memory Efficiency

Be mindful of memory implications for intermediate arrays, especially with large tensors. `einsum` strives for efficiency.

## Common Patterns

`einsum` excels with clear, descriptive index strings.

- **Matrix transpose**: `'ij->ji'`
- **Batch matrix multiplication**: `'bij,bjk->bik'`
- **Vector dot product**: `'i,i->'`
- **Matrix trace (sum of diagonal)**: `'ii->'`
- **Sum all elements (tensor)**: `'...->'`
- **Quadratic form (vector-matrix-vector)**: `'i,ij,j->'`

# Conclusion

## Key Takeaways:

1. **Concise Notation**: `einsum` offers a powerful and concise domain-specific language for tensor operations using Einstein summation convention.
2. **Versatility**: Both functions perform a wide array of operations including dot products, outer products, transpositions, permutations, and complex tensor contractions.
3. **Performance**: `einsum` can be highly performant, especially with the `optimize` flag for complex contractions. `tensordot` is also optimized for its specific tasks.
4. **Readability**: While initially cryptic, `einsum` strings become very readable for those familiar with the notation, clearly expressing tensor manipulation intent.
5. **Framework Support**: Both are available in NumPy for CPU-bound tasks and PyTorch (often with GPU acceleration) for deep learning.

## When to Use Each:

- **`einsum`**: For flexible control over indices (summing, keeping, reordering), complex multi-tensor contractions, or operations hard to express concisely with standard routines (e.g., attention mechanisms).
- **`tensordot`**: When contracting specific pairs of axes between two tensors. It's a good general-purpose tensor dot product when the `axes` argument clearly defines the contraction.
- **Standard operators (`@`, `np.dot`, `np.multiply`, etc.)**: For very common and simple operations, as they are often the most readable and highly optimized.

## Best Practices:

1. **Clarity**: Add comments to complex `einsum` strings.
2. **Optimization**: For `numpy.einsum` with three or more tensors, use `optimize=True`.
3. **Profiling**: For critical sections, profile `einsum` against alternatives.
4. **Readability vs. Conciseness**: Balance `einsum`'s conciseness with the readability of standard operations.
5. **Start Simple**: Begin by reformulating simple operations with `einsum` and gradually move to complex ones.