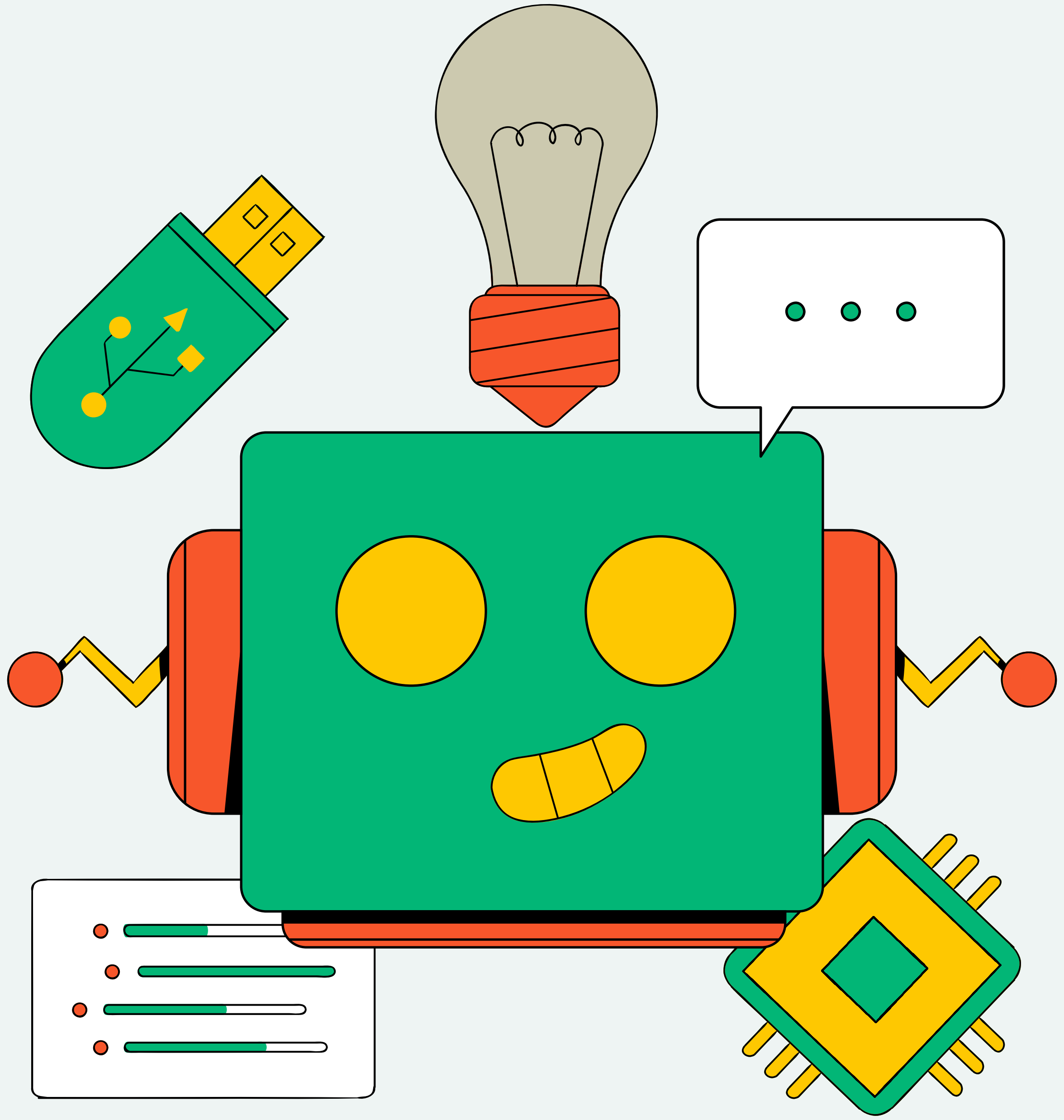
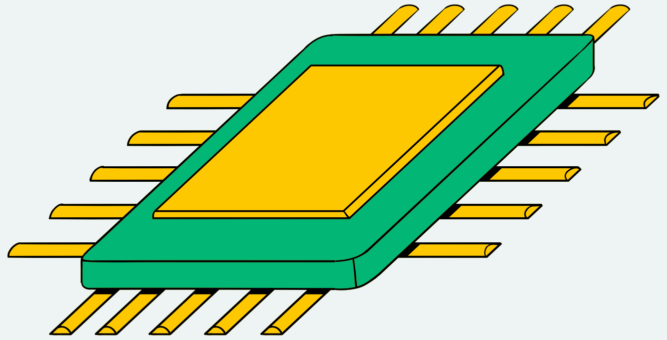
GROUP PRESENTATION

LINEAR ALGEBRA



PRESENTED BY:

- Vo Duong Xuan Nguyen – 2470738
- Nguyen Cong Thanh – 2470744
- Nguyen Vo Thai Trieu – 2470577
- Pham Quynh Tran – 2470731
- Pham The Long – 2470752



PRESENTATION OUTLINE

- Introduction
- Basic concepts and operations
- Exercises
- Conclusion

OVERVIEW

Linear algebra provides the foundation for representing data as scalars, vectors, matrices, and tensors in deep learning.

These structures appear in inputs, weights, and gradients during training.

Most deep learning computations rely on tensor operations, making linear algebra essential for understanding and implementing models efficiently.

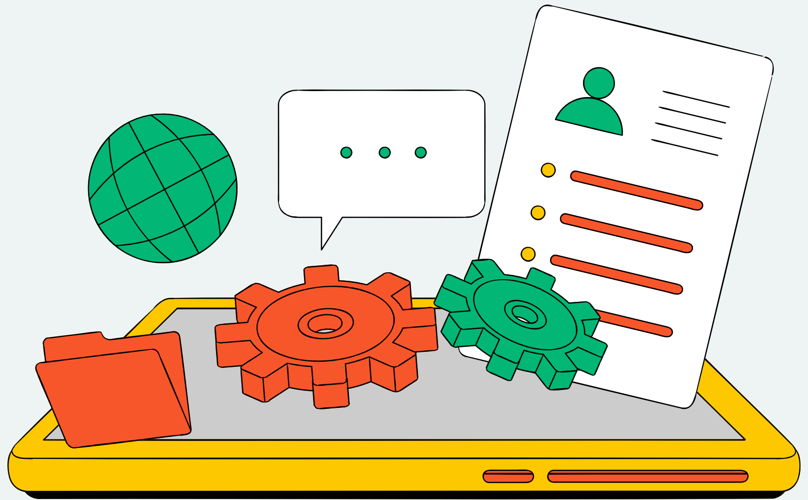
IMPORTANCE AND APPLICATION

Importance

- Linear algebra enables efficient computation and data representation in deep learning.
- Fundamental operations like matrix–vector and matrix–matrix products support transformations across layers.
- Vector norms are crucial in regularization to prevent overfitting.
- Gradient-based optimization relies on linear algebra to compute and update parameters.

Application

- Used in computing losses, class probabilities, and attention via reduction operations (sum, mean, max).
- Powers algorithms such as gradient descent, PCA, and SVD.
- Supports model debugging, interpretation, and performance tuning.



USE CASES IN COMPUTER SCIENCE

COMPUTER GRAPHICS

Geometric transformations (translation, scaling, rotation)

COMPUTER VISION

Convolutions, filtering, feature extraction

NATURAL LANGUAGE PROCESSING

Word embeddings, cosine similarity

RECOMMENDER SYSTEMS

Matrix factorization for user–item modeling

DATA COMPRESSION

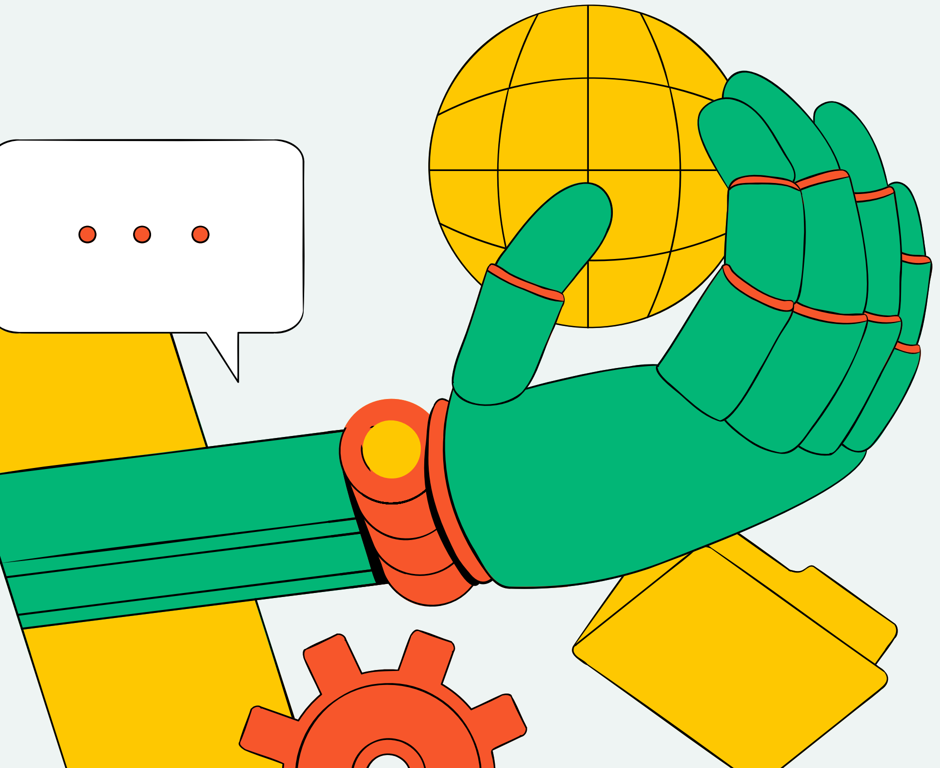
Dimensionality reduction, encoding techniques

SIGNAL PROCESSING

Filtering, Fourier transforms using matrix operations

BASIC CONCEPTS …

SCALARS - DEFINITION



A scalar is a single number - without any direction or structure.

We denote scalars by ordinary lower-cased letters (e.g. x, y and z) and the space of all (continuous) real-valued scalars by $\mathbb{R}$

SCALARS - COMPUTATION IN PRACTICE

In PyTorch, we can use **torch.tensor()** contain only one element for scalars.

```
import torch
x = torch.tensor(3.0)
y = torch.tensor(2.0)
x + y, x * y, x / y, x**y
```

```
(tensor(5.), tensor(6.), tensor(1.5000), tensor(9.))
```

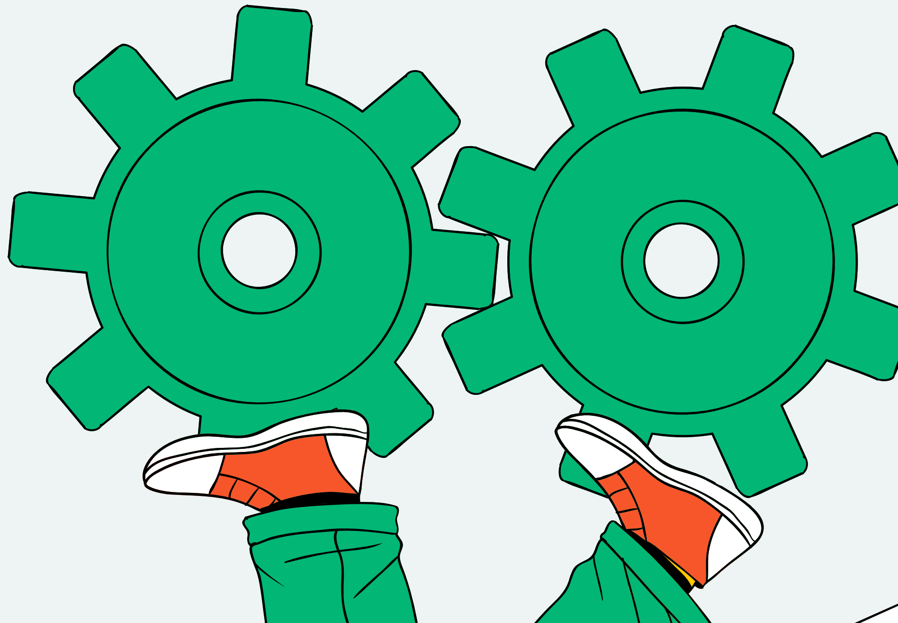
SCALARS - APPLICATIONS

E.g.: The temperature in Palo Alto is a balmy 72 degrees Fahrenheit. This is the expression to convert Fahrenheit to Celsius.

$$c = \frac{5}{9}(f - 32)$$

In this equation,

- 5, 9, and 32 are constants scalars.
- Variables c and f in general represent unknown scalars.



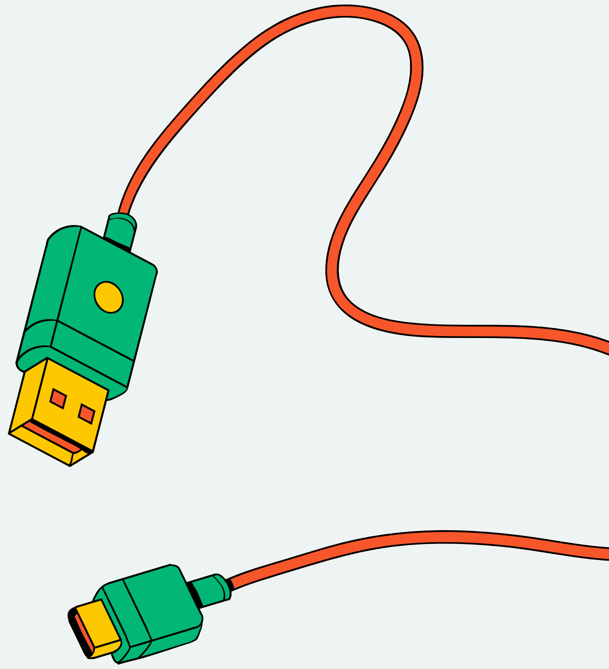
VECTORS - DEFINITION

Vector is a **fixed-length array** of scalars.
By default, we visualize vectors by stacking their elements vertically.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

, x1, ... , xn are elements of the vector

To indicate that a vector contains n elements, we write $\mathbf{x} \in \mathbb{R}^n$
or n is the dimensionality of the vector



VECTORS - COMPUTATION IN PRACTICE

In PyTorch, we can use **torch.arange()** to create a 1-dimensional tensor containing a sequence of integers starting from 0

```
x = torch.arange(3)
>> tensor([0, 1, 2])
```

```
len(x)
>> 3
```

```
x.shape
>> torch.Size([3])
```

VECTORS - APPLICATIONS

When vectors represent examples from real-world datasets, their values hold some real-world significance.

E.g., if we were training a model to predict the risk of a loan defaulting, we might associate each applicant with a vector whose components correspond to quantities like their income, length of employment, or number of previous defaults.

MATRICES - DEFINITION

While scalars are 0th-order tensors and vectors are 1st-order tensors, matrices are **2nd-order tensors**. We denote matrices by bold capital letters (e.g. **X**, **Y, Z**).

The expression $\mathbf{A} \in \mathbb{R}^{m \times n}$ indicates that a matrix **A** contains m × n real-valued scalars, arranged as m rows and n columns. When m = n, we say that a matrix is square

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}.$$

To refer to an individual element, we subscript both the row and column indices.
E.g., aij is the value that belongs to **A**’s ith row and jth column.

MATRICES - DEFINITION

We signify a matrix **A**’s transpose by **A**T and if **B** = **A**T, then bij = aji for all i and j. Thus, the transpose of an m × n matrix is an n × m matrix:

$$\mathbf{A}^{\mathrm{T}} = \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{bmatrix}$$

Symmetric matrices are the subset of square matrices that are equal to their own transposes **A** = **A**T. The following matrix is symmetric:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 0 & 4 \\ 3 & 4 & 5 \end{bmatrix}$$

MATRICES - COMPUTATION IN PRACTICE

In PyTorch, we use **torch.arange()** and **reshape()**

Ex: We create a 1D tensor with values [0, 1, 2, 3, 4, 5], then reshape it into a 2D tensor with 3 rows and 2 columns.

```
A = torch.arange(6).reshape(3, 2)
>> tensor( [[0, 1],
            [2, 3],
            [4, 5]] )
```

And we can access any matrix's transpose by A.T expression

```
A.T
>> tensor( [[0, 2, 4],
             [1, 3, 5]] )
```

MATRICES - APPLICATIONS

Matrices are useful for representing datasets. Typically, rows correspond to individual records and columns correspond to distinct attributes.

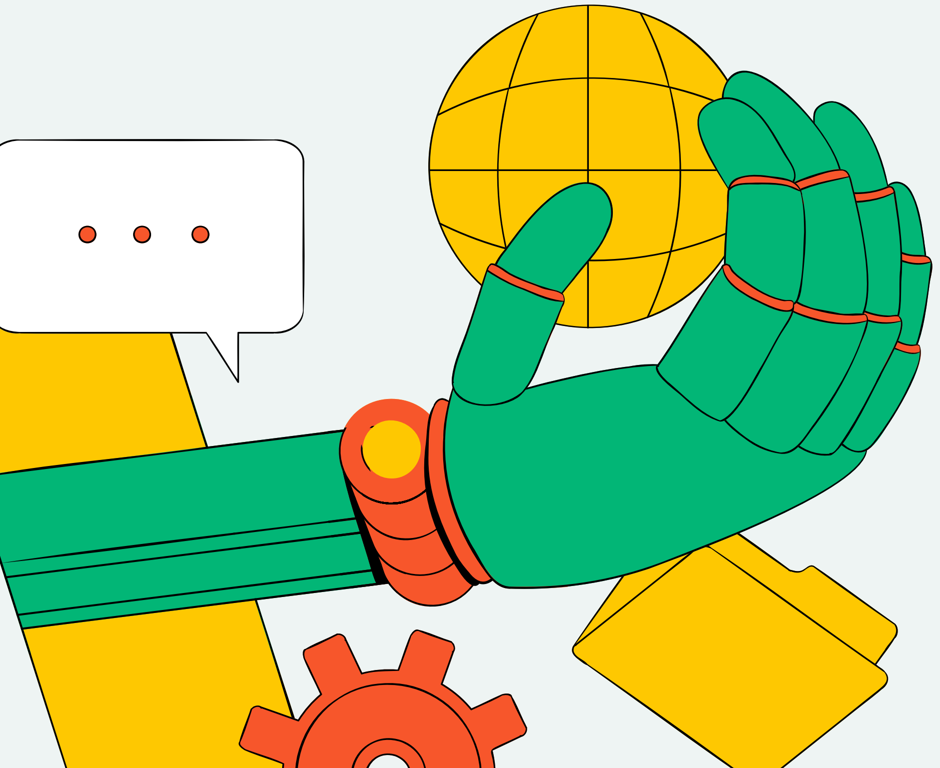
In computer graphics and video game development, matrices are used to perform transformations such as scaling, rotation, and translation of 2D and 3D objects. Matrices play a crucial role in projecting 3D scenes onto a 2D screen, creating realistic graphics.

TENSORS - DEFINITION

In deep learning, we frequently work with **tensors**, which are generalizations of **scalars** (0D), **vectors** (1D), and **matrices** (3D) to **higher-dimensional arrays**.

A tensor is simply **a multi-dimensional array**, and its number of axes (or dimensions) is referred to as its rank.

- **A scalar** is a **0D tensor** (e.g., a single number).
- **A vector** is a **1D tensor** (e.g., an array of numbers).
- **A matrix** is a **2D tensor** (e.g., a table of numbers with rows and columns).
- **A 3D tensor** could be an array of matrices, and higher-order tensors (4D, 5D, etc.) follow accordingly.



TENSORS - COMPUTATION IN PRACTICE

In PyTorch, **tensors** can be implemented as the following examples:

```
import torch

# Vector (1D tensor)
b = torch.tensor([1.0, 2.0, 3.0])
```

```
import torch

# Matrix (2D tensor)
c = torch.tensor([[1, 2], [3, 4]])
```

```
import torch

# Scalar (0D tensor)
a = torch.tensor(3.14)
```

```
import torch

# 3D Tensor: 2 matrices of shape (2x2)
d = torch.tensor([[[1, 2], [3, 4]],
                  [[5, 6], [7, 8]]])
```

TENSORS - APPLICATIONS

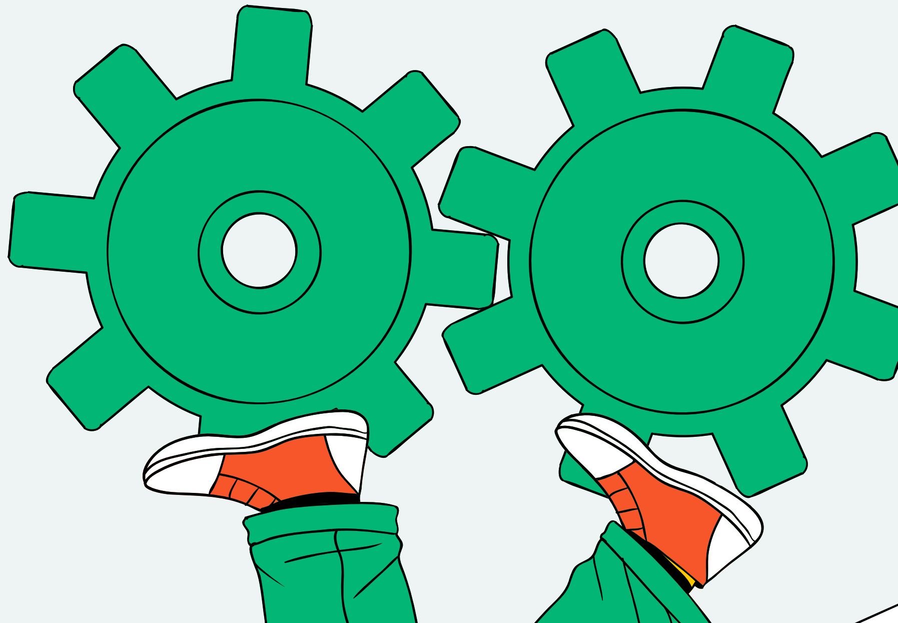
Tensors are the fundamental data structures in deep learning frameworks like PyTorch and TensorFlow. They are used to:

- Represent inputs (e.g., images, text, audio).
- Store model parameters (e.g., weights, biases).
- Track activations and gradients during training.

They allow models to process data in batch form efficiently, enabling GPU acceleration and vectorized computation.

In practical deep learning applications:

- A **grayscale image** might be a 2D tensor (height × width).
- A **color image** would be a 3D tensor (channels × height × width).
- A **batch of images** would be represented as a 4D tensor (batch_size × channels × height × width).

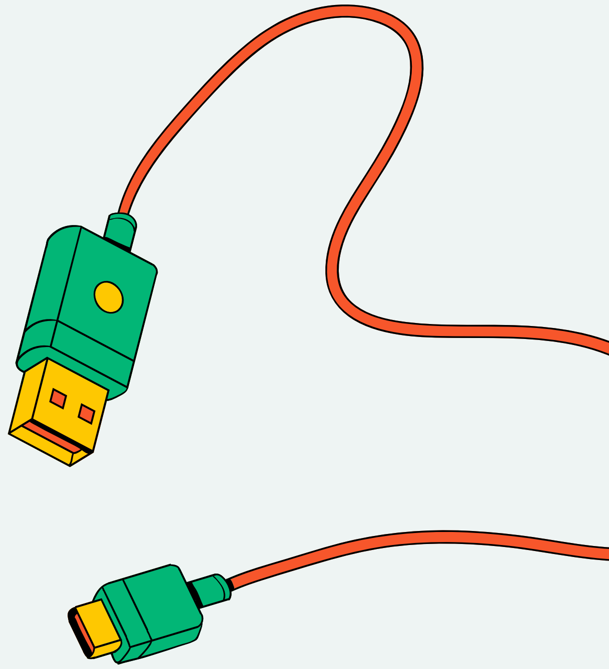


BASIC PROPERTIES OF TENSOR ARITHMETIC – DEFINITION

Tensor arithmetic refers to operations such as **addition**, **multiplication**, and **scalar operations** applied element-wise. These follow consistent algebraic rules extended to tensors.

Important concepts include:

- Element-wise: addition and multiplication.
- Broadcasting.
- Hadamard product.



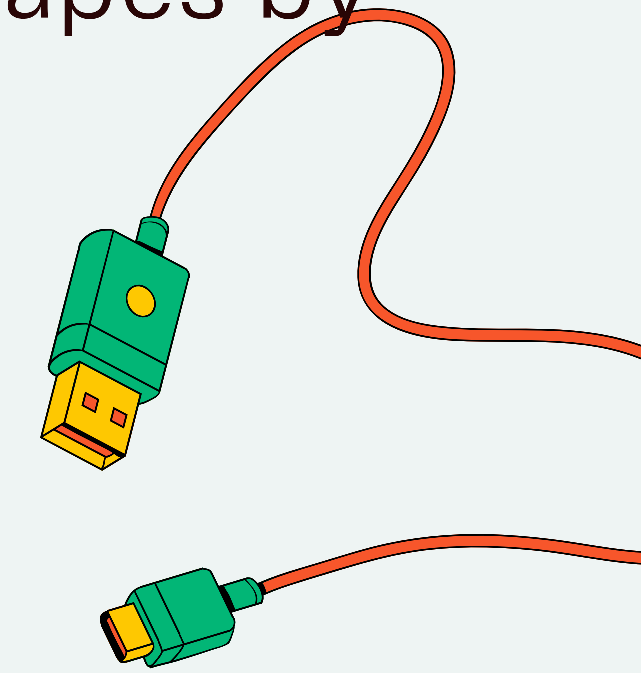
BASIC PROPERTIES OF TENSOR ARITHMETIC – DEFINITION

Important concepts include:

- **Element-wise addition and multiplication**: Operate independently on corresponding elements of two tensors.
  - Addition: $(\mathbf{A}+\mathbf{B})_{ij} = \mathbf{A}_{ij} + \mathbf{B}_{ij}$
  - Multiplication: $(\mathbf{A}\cdot\mathbf{B})_{ij} = \mathbf{A}_{ij} \cdot \mathbf{B}_{ij}$

- **Broadcasting**: Allows operations on tensors of different shapes by automatically expanding them to compatible shapes.

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \mathbf{B} = \begin{bmatrix} 10 & 20 \end{bmatrix} \Rightarrow \mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \mathbf{B} = \begin{bmatrix} 10 & 20 \\ 10 & 20 \end{bmatrix} \Rightarrow \mathbf{A}+\mathbf{B} = \begin{bmatrix} 11 & 22 \\ 13 & 24 \end{bmatrix}$$

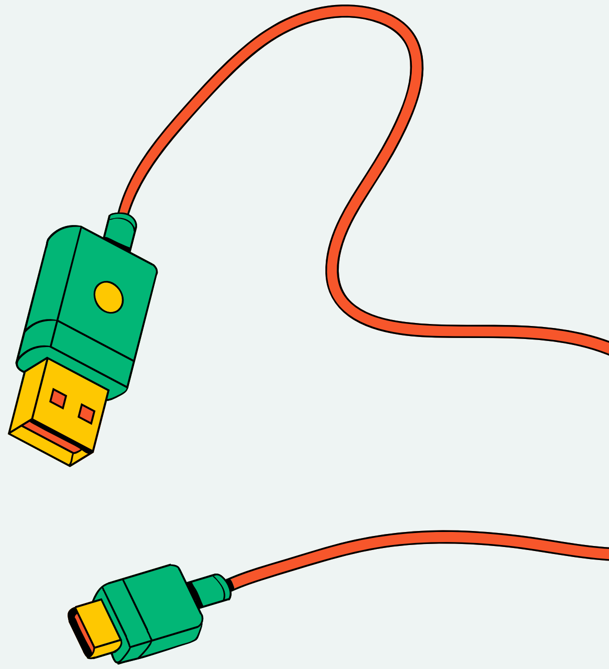


BASIC PROPERTIES OF TENSOR ARITHMETIC – DEFINITION

- **Hadamard product**: A special case of element-wise multiplication between two matrices of the same shape.

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \cdots & a_{1n}b_{1n} \\ a_{21}b_{21} & a_{22}b_{22} & \cdots & a_{2n}b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{m1} & a_{m2}b_{m2} & \cdots & a_{mn}b_{mn} \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \odot \begin{bmatrix} 10 & 20 \\ 30 & 40 \end{bmatrix} = \begin{bmatrix} 1 \times 10 & 2 \times 20 \\ 3 \times 30 & 4 \times 40 \end{bmatrix} = \begin{bmatrix} 10 & 40 \\ 90 & 160 \end{bmatrix}$$



… AND OPERATIONS

BASIC PROPERTIES OF TENSOR ARITHMETIC - COMPUTATION IN PRACTICE

In PyTorch, **element-wise operations**, **broadcasting** and **Hadamard product** can be implemented as the following examples:

```
import torch

A = torch.tensor([[1, 2], [3, 4]])
B = torch.tensor([[10, 20], [30, 40]])

# Element-wise addition
add = A + B
# Element-wise multiplication
mul = A * B
```

```
import torch

A = torch.tensor([[1, 2], [3, 4]])
B = torch.tensor([[10, 20], [30, 40]])
C = torch.tensor([10, 20])

# C is 1D but gets automatically
# expanded to match A’s shape
broadcast_result = A + C
# In PyTorch, Hadamard product is
# just element-wise multiplication
hadamard = A * B
```

BASIC PROPERTIES OF TENSOR ARITHMETIC – APPLICATIONS

- Efficient **manipulation of data** in vectorized form.
- Used in every step of **neural network training and inference**:
  - **Forward propagation** (e.g., adding bias vectors).
  - **Loss calculation** (e.g., subtracting predictions from ground truth).
  - **Backpropagation** (element-wise gradient calculations).
- Broadcasting reduces memory usage and boosts performance by avoiding explicit data replication.
- Hadamard product is used in:
  - **Attention mechanisms** (element-wise scaling of values).
  - **Feature-wise modulation** (e.g., gating or residual scaling).

REDUCTION OPERATIONS

```
A = torch.tensor([
 [0,1,0],
 [2,0,2],
 [0,3,0]
], dtype=torch.float32)
A.numel()
```

```
tensor([[0., 1., 0.], [2., 0., 2.], [0., 3., 0.]])
9
```

```
A.sum()
A.sum().numel()
```

```
tensor(8.)
1
```

- Reduction is an operation that reduces the number of dimensions of a tensor by calculating aggregate values (**sum, mean, max, min, etc**.) along one or more axes(axis) of the tensor.
- Reduction is an operation that reduces the number of elements in a tensor.

REDUCTION OPERATIONS

Common reduction operations

```
A.sum() # sum of elements
A.prod() # product of elements
A.mean() # mean (average) of elements
A.std() # standard deviation of elements
```

```
tensor(8.)
tensor(0.)
tensor(0.8889)
tensor(1.1667
```

REDUCTION OPERATIONS

Reduction operations don't always have to reduce tensors to a single scalar; they can also reduce tensors along specific axes:

- Axis 0 refers to reducing along columns (vertical direction).
- Axis 1 refers to reducing along rows (horizontal direction).

```
A = torch.tensor([
 [1,1,1,1],
 [2,2,2,2],
 [3,3,3,3]
], dtype=torch.float32)
```

```
A.shape
A.sum(axis=0)          Sum along each column
A.sum(axis=0).shape
```

```
torch.Size([3, 4])
tensor([6., 6., 6., 6.])
torch.Size([4])
```

```
A.sum(axis=1)          Sum along each row
A.sum(axis=1).shape
```

```
tensor([ 4., 8., 12.])
torch.Size([3])
```

REDUCTION OPERATIONS

Reducing multiple axes simultaneously produces the same result as reducing the entire tensor directly

```
A.sum(axis[0,1])
A.sum()
```

```
tensor(24.)
tensor(24.)
```

```
A = torch.tensor([
 [1,1,1,1],
 [2,2,2,2],
 [3,3,3,3]
], dtype=torch.float32)
```

REDUCTION OPERATIONS

The mean is simply the sum of all elements divided by the total number of elements:

```
A.mean()
A.sum() / A.numel()      total value divided by total quantity
```

```
tensor(2.)
tensor(2.)
```

```
A.mean(axis=0)
A.sum(axis=0) / A.shape[0]
```

```
tensor([2., 2., 2., 2.])
tensor([2., 2., 2., 2.])
```

```
A.mean(axis=1)
A.sum(axis=1) / A.shape[1])
```

```
tensor([1., 2., 3.])
tensor([1., 2., 3.])
```

```
A = torch.tensor([
  [1,1,1,1],
  [2,2,2,2],
  [3,3,3,3]
], dtype=torch.float32)
```

NON-REDUCED SUMS

When using functions such as sum() or mean(), by default the tensor is reduced by removing the axes along which the sum or mean was calculated.
However, sometimes we want to retain the original dimensions (axes), making subsequent calculations more convenient. This can be achieved by setting the parameter keepdims=True.

```
A = torch.tensor([[ 0,  1,  2,  3],
                  [ 4,  5,  6,  7],
                  [ 8,  9, 10, 11],
                  [12, 13, 14, 15],
                  [16, 17, 18, 19]], dtype=torch.float32)
A.shape
torch.Size([5, 4])  This is a 2-dimensional tensor.
A.sum(axis=1)
A.sum(axis=1).shape
tensor([ 6., 22., 38., 54., 70.])
torch.Size([5])  This is a 1-dimensional tensor.
```

The result has lost axis 1 (column) because each row now has only 1 number.

```
A.sum(axis=1, keepdims=True)
A.sum(axis=1, keepdims=True).shape
tensor([[ 6.],
        [22.],
        [38.],
        [54.],
        [70.]])
torch.Size([5, 1])  This is a 2-dimensional tensor.
```

Still keeps the tensor 2-dimensional, even though each row has only 1 element.

NON-REDUCED SUMS

Keeping this dimension can be particularly useful for subsequent calculations involving broadcasting, such as **row-wise normalization**:

```
A = torch.tensor([[ 0, 1, 2, 3],
 [ 4, 5, 6, 7],
 [ 8, 9, 10, 11],
 [12, 13, 14, 15],
 [16, 17, 18, 19]], dtype=torch.float32)
A.shape
torch.Size([5, 4])
A.sum(axis=1, keepdims=True)
A.sum(axis=1, keepdims=True).shape
tensor([[ 6.],
      [22.],
      [38.],
      [54.],
      [70.]])
torch.Size([5, 1])
```

```
sum_A = A.sum(axis=1, keepdims=True)
A_normalized = A / sum_A
tensor([
 [0.0000, 0.1667, 0.3333, 0.5000],
 [0.1818, 0.2273, 0.2727, 0.3182],
 [0.2105, 0.2368, 0.2632, 0.2895],
 [0.2222, 0.2407, 0.2593, 0.2778],
 [0.2286, 0.2429, 0.2571, 0.2714]
])
```

First line A is [0, 1, 2, 3]

First line sum_A is [6]

First line A_normalized = [0/6, 1/6, 2/6, 3/6] → [0.0, 0.1667, 0.3333, 0.5]

CUMULATIVE SUM

The cumsum(axis) function computes the cumulative sum of elements along a specified axis without removing that axis.
Example of cumulative sum along axis 0 (vertically, row-wise):

```
A = torch.tensor([[ 0, 1, 2, 3],
 [ 4, 5, 6, 7],
 [ 8, 9, 10, 11],
 [12, 13, 14, 15],
 [16, 17, 18, 19]], dtype=torch.float32)
A.cumsum(axis=0)
A.shape == A.cumsum(axis=0).shape
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  6.,  8., 10.],
        [12., 15., 18., 21.],
        [24., 28., 32., 36.],
        [40., 45., 50., 55.]])
True
```

Row 1 remains unchanged: [0, 1, 2, 3]
Row 2 becomes [0+4, 1+5, 2+6, 3+7] → [4, 6, 8, 10]
Row 3 becomes [4+8, 6+9, 8+10, 10+11] → [12, 15, 18, 21]
...

The result always has the same shape as the original tensor

DOT PRODUCT – DEFINITION

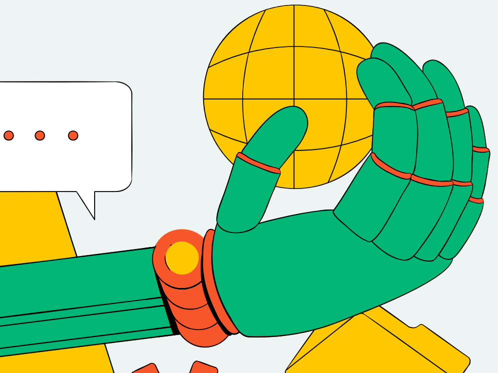
The dot product is a fundamental operation that returns a scalar from two vectors of equal length.

Given two vectors:

$$\mathbf{x} = [x_1, x_2, \dots, x_n], \mathbf{y} = [y_1, y_2, \dots, y_n] \in \mathbb{R}^n$$

→ Dot product:

$$\mathbf{x}^{\mathrm{T}}\mathbf{y} = \sum_{i=1}^{n} x_i y_i$$



DOT PRODUCT - COMPUTATION IN PRACTICE

In PyTorch, we can use **torch.dot()** to calculate the dot product.

```
import torch
x = torch.tensor([0., 1., 2.])
y = torch.ones(3, dtype=torch.float32)
result = torch.dot(x, y)
```

```
>> 3
```

... There is also an alternative way for calculating dot product:

```
result = torch.sum(x * y)
```

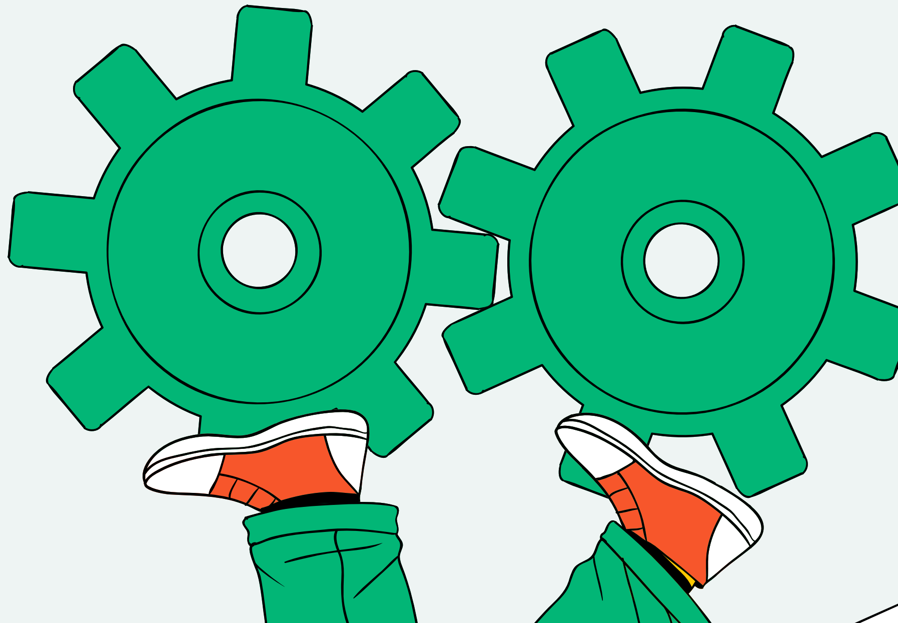
```
>> 3
```

DOT PRODUCT - APPLICATIONS

- Weighted averages: Dot product $\mathbf{x}^{\mathrm{T}}\mathbf{y}$ represents a weighted average when $\sum w_i = 1$

- Cosine similarity: For unit vectors, $\mathbf{x}^{\mathrm{T}}\mathbf{y}$ equals the cosine of the angle between them, used in search and recommendation systems.

- Neural networks: Dot products compute linear combinations in dense layers, enabling prediction and feature learning.



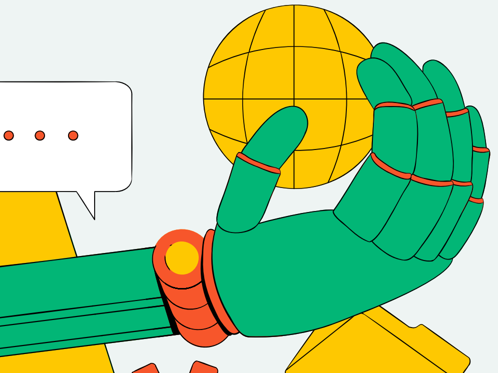
MATRIX-VECTOR PRODUCT - DEFINITION

Given a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ and a vector $\mathbf{x} \in \mathbb{R}^n$

- **A** can be visualized as $\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^T \\ \mathbf{a}_2^T \\ \vdots \\ \mathbf{a}_m^T \end{bmatrix}$

- The **matrix-vector product Ax** is simply a column vector of length m, each element is a **dot product**:

$$\mathbf{Ax} = \begin{bmatrix} \mathbf{a}_1^T \\ \mathbf{a}_2^T \\ \vdots \\ \mathbf{a}_m^T \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{a}_1^T \mathbf{x} \\ \mathbf{a}_2^T \mathbf{x} \\ \vdots \\ \mathbf{a}_m^T \mathbf{x} \end{bmatrix}$$



MATRIX-VECTOR PRODUCT - COMPUTATION IN PRACTICE

In PyTorch, we can use **torch.mv()** to compute matrix-vector product.

```
import torch
A = torch.arange(6, dtype=torch.float32).reshape(2, 3)
x = torch.tensor([1, 2, 3], dtype=torch.float32)
y = torch.mv(A, x)
```

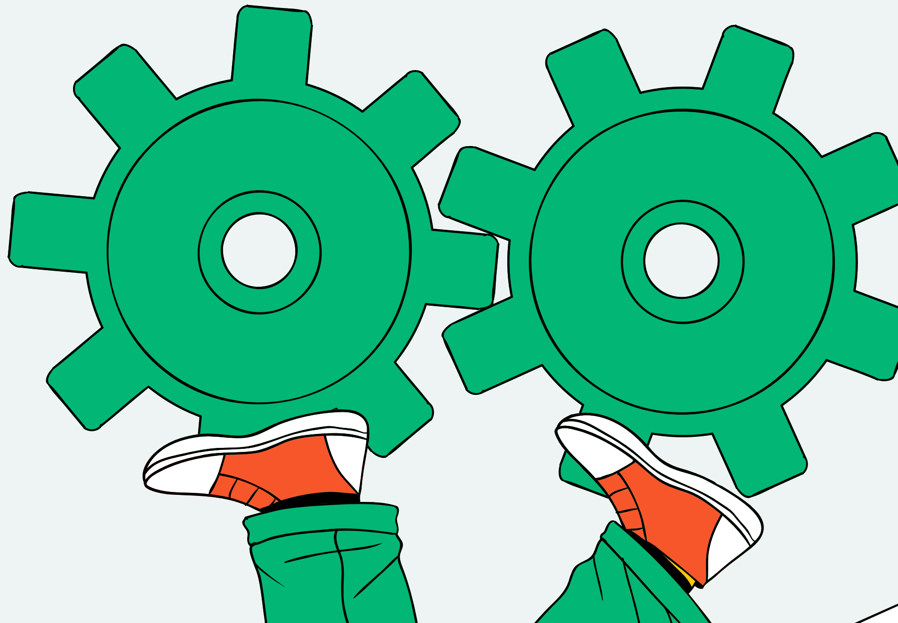
→ A is $\begin{bmatrix}0 & 1 & 2\\3 & 4 & 5\end{bmatrix}$

→ x is $\begin{bmatrix}1\\2\\3\end{bmatrix}$

```
>> tensor([ 8, 26])
```

MATRIX-VECTOR PRODUCT - APPLICATIONS

- Linear systems: Solving is foundational in optimization and numerical analysis.
- Neural networks: In dense layers, the output for each input vector is computed via a matrix–vector product between the weight matrix and the input.
- Transformations: Matrix–vector multiplication expresses linear transformations such as rotations, scalings, and projections in geometry and graphics.



MATRIX-MATRIX MULTIPLICATION - DEFINITION

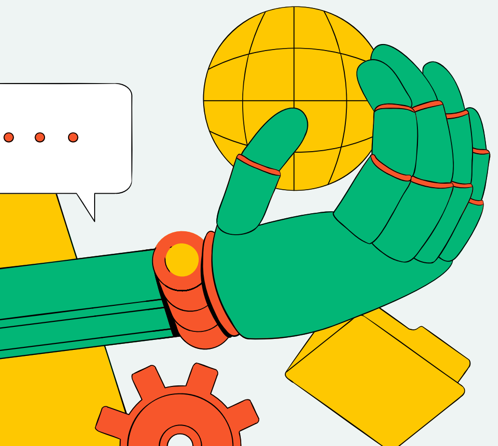
Given two matrices $\mathbf{A} \in \mathbb{R}^{n \times k}$ and $\mathbf{B} \in \mathbb{R}^{k \times m}$

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1k} \\ a_{21} & a_{22} & \dots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nk} \end{bmatrix} \rightarrow \quad \mathbf{A} = \begin{bmatrix} \mathbf{a}_1^T \\ \mathbf{a}_2^T \\ \vdots \\ \mathbf{a}_m^T \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1m} \\ b_{21} & b_{22} & \dots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k1} & b_{k2} & \dots & b_{km} \end{bmatrix} \rightarrow \quad \mathbf{B} = [\mathbf{b}_1 \quad \mathbf{b}_2 \quad \dots \quad \mathbf{b}_m]$$

→ Matrix-matrix product $\mathbf{C} \in \mathbb{R}^{n \times m}$

$$\mathbf{C} = \mathbf{A}\mathbf{B} = \begin{bmatrix} \mathbf{a}_1^T \\ \mathbf{a}_2^T \\ \vdots \\ \mathbf{a}_n^T \end{bmatrix} [\mathbf{b}_1 \quad \mathbf{b}_2 \quad \dots \quad \mathbf{b}_m] = \begin{bmatrix} \mathbf{a}_1^T\mathbf{b}_1 & \mathbf{a}_1^T\mathbf{b}_2 & \dots & \mathbf{a}_1^T\mathbf{b}_m \\ \mathbf{a}_2^T\mathbf{b}_1 & \mathbf{a}_2^T\mathbf{b}_2 & \dots & \mathbf{a}_2^T\mathbf{b}_m \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{a}_n^T\mathbf{b}_1 & \mathbf{a}_n^T\mathbf{b}_2 & \dots & \mathbf{a}_n^T\mathbf{b}_m \end{bmatrix}$$



MATRIX-MATRIX PRODUCT - COMPUTATION IN PRACTICE

In PyTorch, we can use **torch.mm()** contain only one element for scalars.

```
import torch
A = torch.arange(6, dtype=torch.float32).reshape(2, 3)
B = torch.tensor([[1, 2], [3, 4], [5, 6]], dtype=torch.float32)
C = torch.mm(A, B)
```

→ $$\mathbf{A} = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix}$$

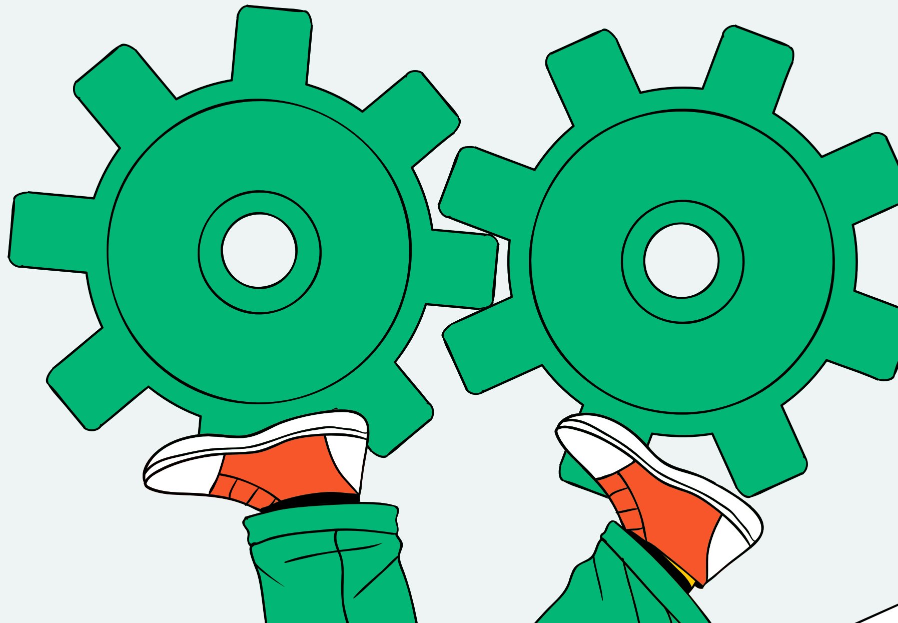
→ $$\mathbf{B} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

```
>> tensor([[13, 16],
           [40, 52]])
```

$$\mathbf{C} = \begin{bmatrix} 13 & 16 \\ 40 & 52 \end{bmatrix}$$

MATRIX-MATRIX MULTIPLICATION - APPLICATIONS

- Batch operations in machine learning: Matrix–matrix multiplication enables parallel computation over mini-batches, unlike matrix–vector multiplication.
- Composing linear transformations: Multiple transformations (e.g., rotation and scaling) can be combined via matrix–matrix products.
- Optimization and modeling: Algorithms such as PCA and SVD rely heavily on efficient matrix–matrix operations.



Norm – Definition

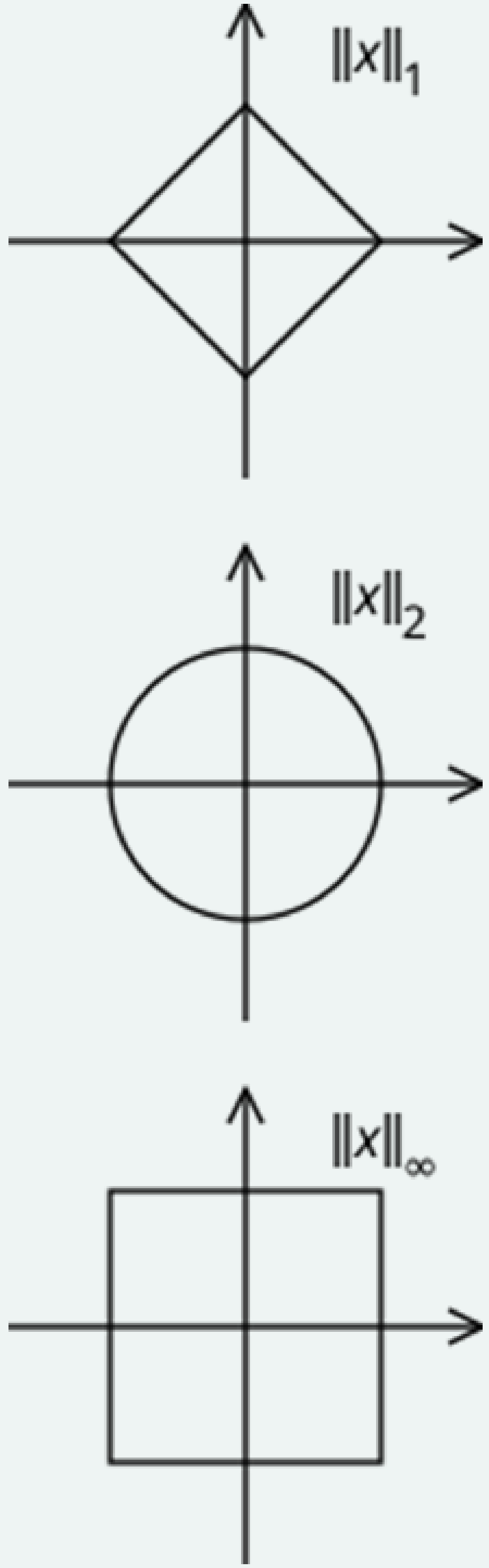
The norm of a vector is a measure of its length in a vector space.

The most common norm is the Euclidean norm (L2 norm), denoted as $\| x \|_2 = \sqrt{\sum_{i=1}^{n} x_i^{\,2}}$

The L1 norm is also common and the measure is called the Manhattan distance: $\| x \|_1 = \sum_{i=1}^{n} |x_i|$

The general Lp norms: $\| x \|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}$

as p approaches ∞ the infinity norm or maximum norm: $\|x\|_\infty = max_i |x_i|$



Norm – Definition

A norm is a function || · || that maps a vector to a scalar and satisfies the following three properties:

1. Given any vector x, if we scale (all elements of) the vector by a scalar α∈R, its norm scales accordingly:

$$\| \alpha x \| = |\alpha| \, \| x \|$$

2. For any vectors x and y: norms satisfy the triangle inequality:

$$\| x + y \| \leq \| x \| + \| y \|$$

3. The norm of a vector is nonnegative and it only vanishes if the vector is zero:

$$\| x \| > 0 \; \forall \, x \neq 0$$

Norm of matrice – The Frobenius norm

$$\|\mathbf{X}\|_{\mathrm{F}} = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij}^2}.$$

Norm – Computation in practice

A norm is a function || · || that maps a vector to a scalar and satisfies the following three properties:

```
x = torch.tensor([3.0, 4.0])
l2_norm = torch.norm(x)
l1_norm = torch.norm(x, p=1)
linf_norm = torch.norm(x, p=float('inf'))
print("L2 norm:", l2_norm.item())
print("L1 norm:", l1_norm.item())
print("L-infinity norm:", linf_norm.item())
```

```
>> L2 norm: 5.0
>> L1 norm: 7.0
>> L-infinitiy norm: 4.0
```

NORM - APPLICATIONS

Distance metrics: The L2 norm is the basis for Euclidean distance.

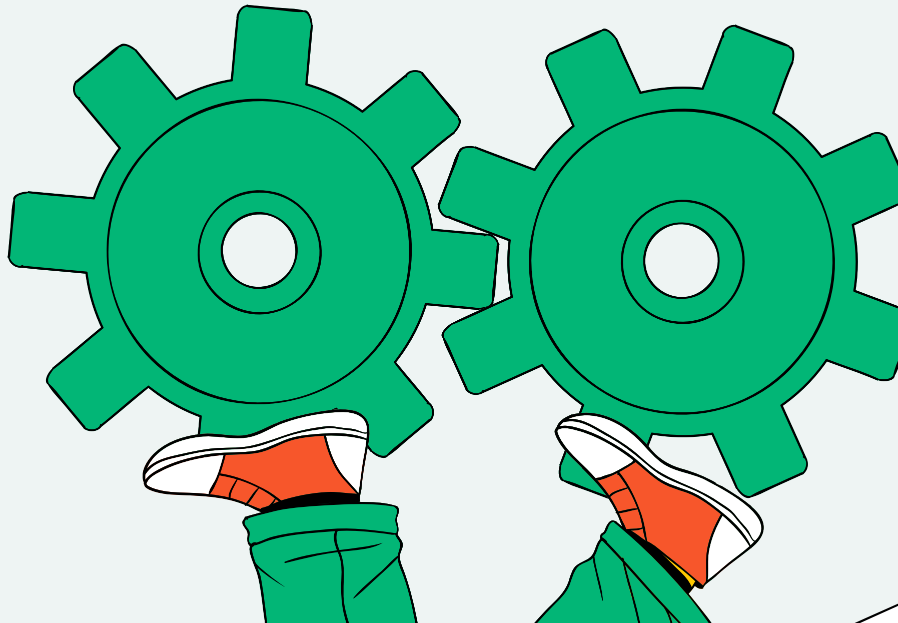
$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|}$$

Error measurement: Norms are used to quantify the difference between predicted and actual values.

EX:

predicted = [8, 9, 5]

actual = [3, 6, 9]



NORM - APPLICATIONS

Data normalization: Vectors are scaled to unit length using their norm.

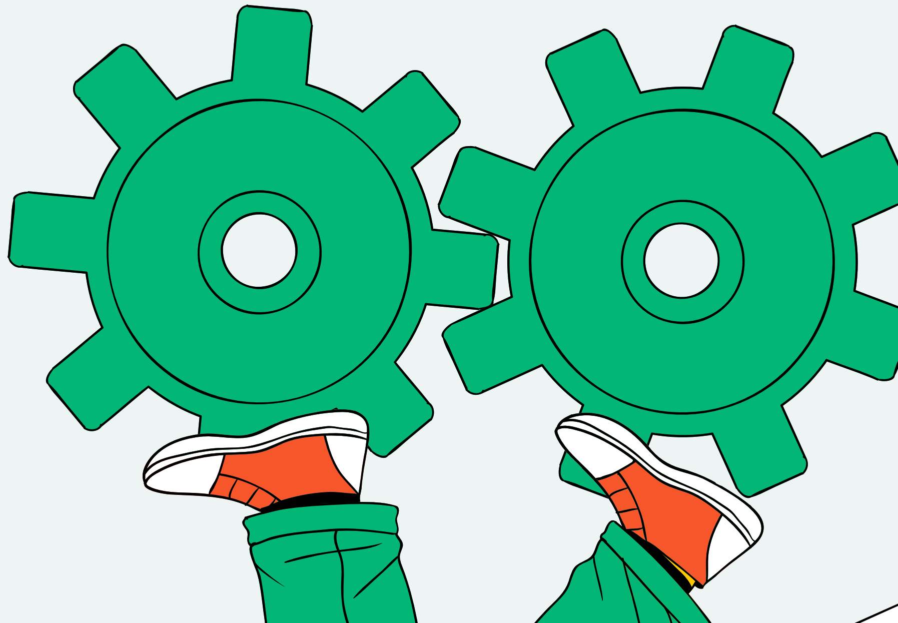
$$\hat{\mathbf{u}} = \frac{\mathbf{u}}{\|\mathbf{u}\|} = \left(\frac{u_1}{\|\mathbf{u}\|}, \frac{u_2}{\|\mathbf{u}\|}, \ldots, \frac{u_n}{\|\mathbf{u}\|}\right)$$

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|}$$

Regularization: L1 and L2 norms are used in machine learning to reduce overfitting.

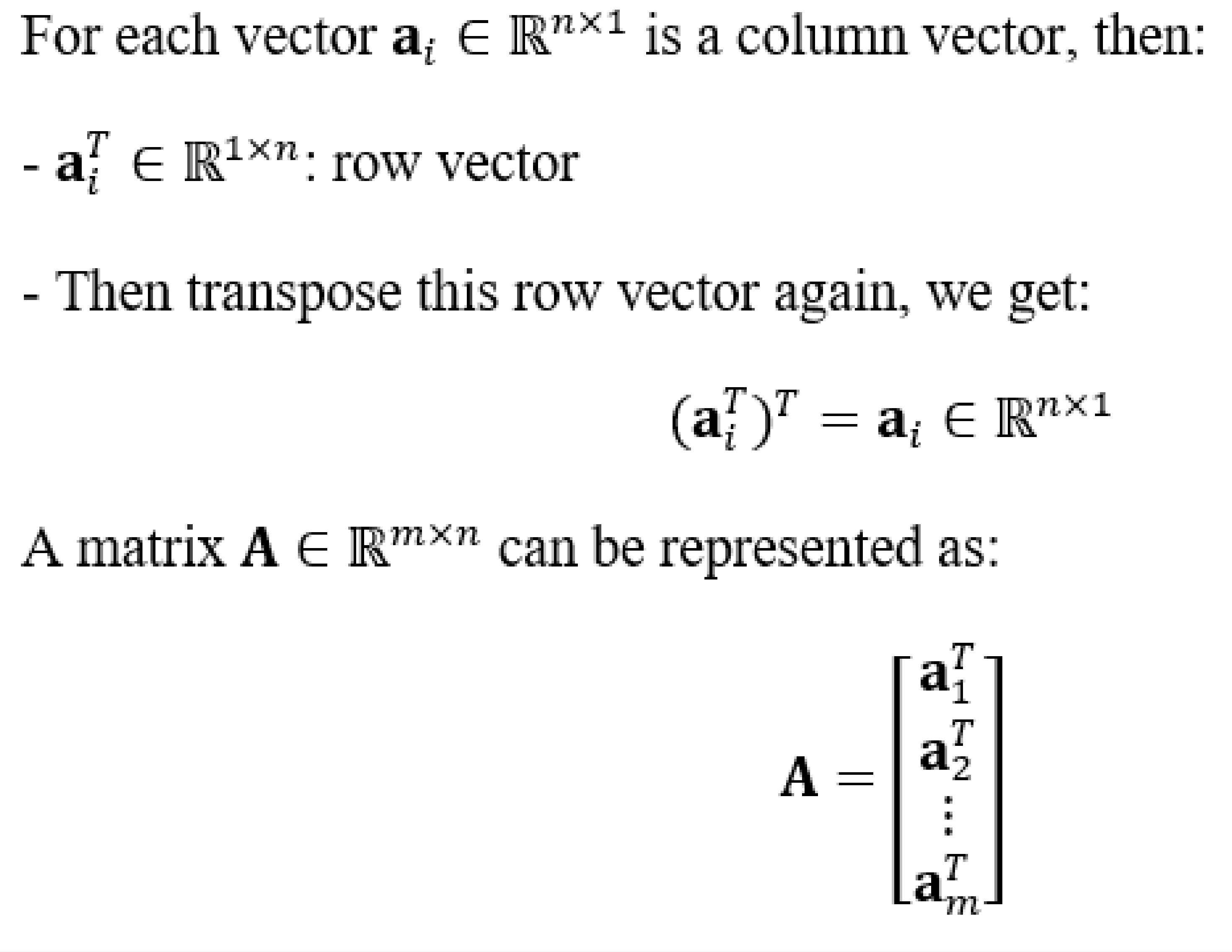
$$L = L_{\text{data}} + \lambda \cdot \|\mathbf{w}\|_1$$

$$L = L_{\text{data}} + \lambda \cdot \|\mathbf{w}\|_2^2$$



EXERCISES

EX-1: PROVE THAT THE TRANSPOSE OF THE TRANSPOSE OF A MATRIX IS THE MATRIX ITSELF $(\mathbf{A}^T)^T = \mathbf{A}$



For each vector $\mathbf{a}_i \in \mathbb{R}^{n\times 1}$ is a column vector, then:

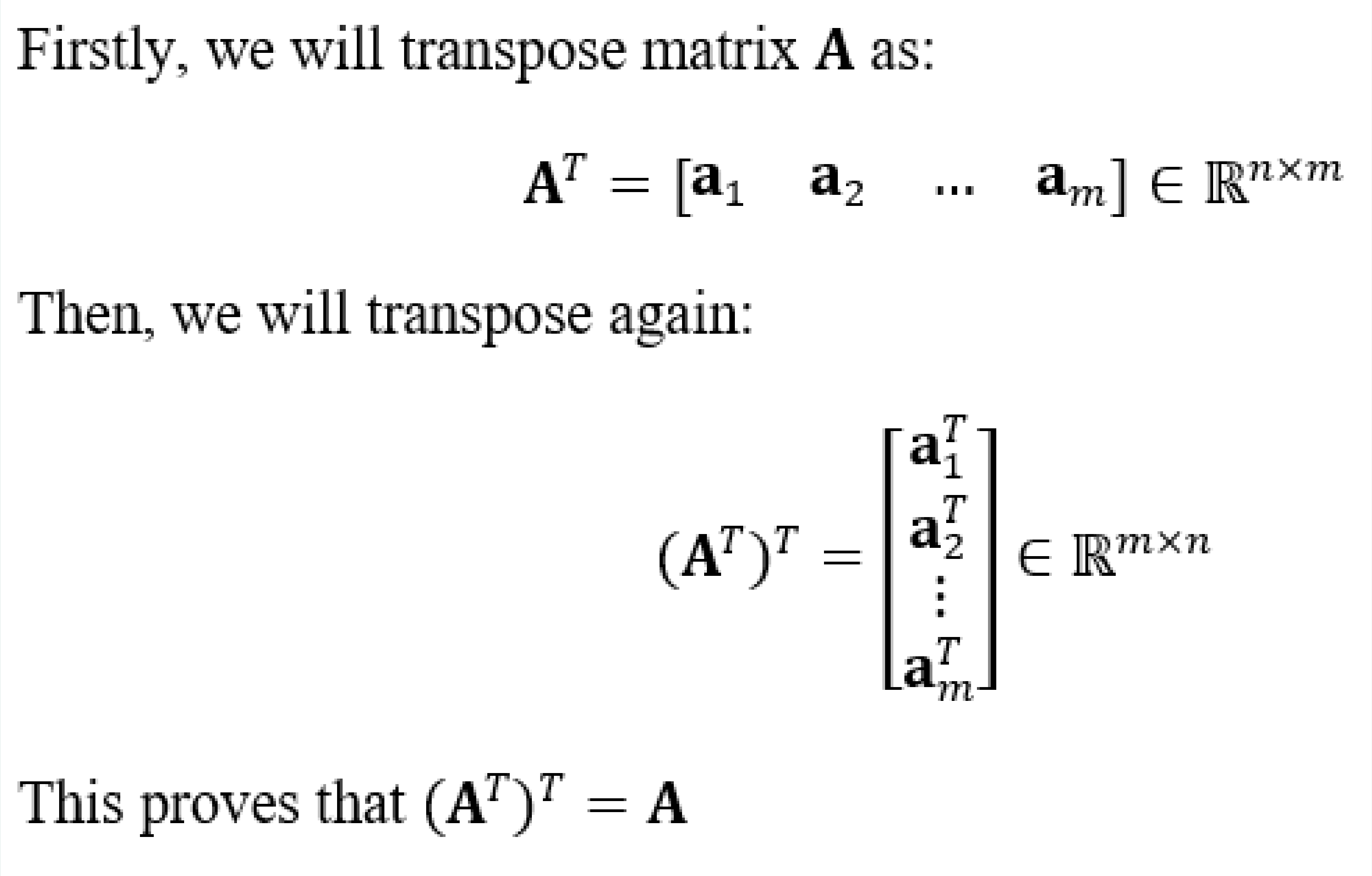
- $\mathbf{a}_i^T \in \mathbb{R}^{1\times n}$: row vector

- Then transpose this row vector again, we get:

$$(\mathbf{a}_i^T)^T = \mathbf{a}_i \in \mathbb{R}^{n\times 1}$$

A matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ can be represented as:

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^T \\ \mathbf{a}_2^T \\ \vdots \\ \mathbf{a}_m^T \end{bmatrix}$$



Firstly, we will transpose matrix $\mathbf{A}$ as:

$$\mathbf{A}^T = [\mathbf{a}_1 \quad \mathbf{a}_2 \quad \dots \quad \mathbf{a}_m] \in \mathbb{R}^{n\times m}$$

Then, we will transpose again:

$$(\mathbf{A}^T)^T = \begin{bmatrix} \mathbf{a}_1^T \\ \mathbf{a}_2^T \\ \vdots \\ \mathbf{a}_m^T \end{bmatrix} \in \mathbb{R}^{m\times n}$$

This proves that $(\mathbf{A}^T)^T = \mathbf{A}$

EX-2: GIVEN TWO MATRICES A AND B, SHOW THAT SUM AND TRANSPOSITION COMMUTE: $\mathbf{A}^\top + \mathbf{B}^\top = (\mathbf{A} + \mathbf{B})^\top$

Given any two matrices A and B of the same size $\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix}$

EX-2: GIVEN TWO MATRICES A AND B, SHOW THAT SUM AND TRANSPOSITION COMMUTE: $\mathbf{A}^\top + \mathbf{B}^\top = (\mathbf{A}+\mathbf{B})^\top$

Given any two matrices A and B of the same size $\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix}$

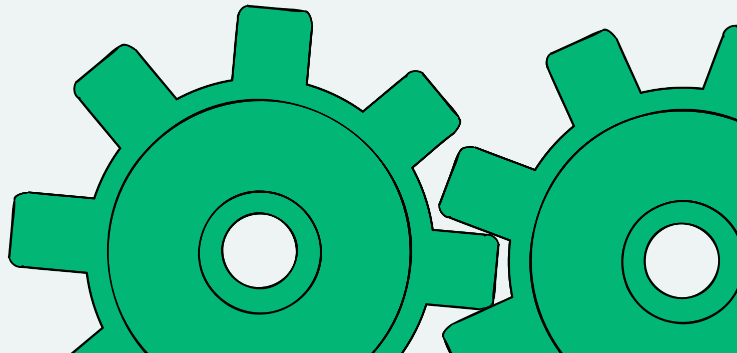
$$\mathbf{A}+\mathbf{B} = \begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & \cdots & a_{1n}+b_{1n} \\ a_{21}+b_{21} & a_{22}+b_{22} & \cdots & a_{2n}+b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}+b_{m1} & a_{m2}+b_{m2} & \cdots & a_{mn}+b_{mn} \end{bmatrix} \quad (\mathbf{A}+\mathbf{B})^{\mathrm{T}} = \begin{bmatrix} a_{11}+b_{11} & a_{21}+b_{21} & \cdots & a_{m1}+b_{m1} \\ a_{12}+b_{12} & a_{22}+b_{22} & \cdots & a_{m2}+b_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n}+b_{1n} & a_{2n}+b_{2n} & \cdots & a_{mn}+b_{mn} \end{bmatrix} \tag{1}$$

EX-2: GIVEN TWO MATRICES A AND B, SHOW THAT SUM AND TRANSPOSITION COMMUTE: $\mathbf{A}^\top + \mathbf{B}^\top = (\mathbf{A}+\mathbf{B})^\top$

Given any two matrices A and B of the same size $\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix}$

$$\mathbf{A}+\mathbf{B} = \begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & \cdots & a_{1n}+b_{1n} \\ a_{21}+b_{21} & a_{22}+b_{22} & \cdots & a_{2n}+b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}+b_{m1} & a_{m2}+b_{m2} & \cdots & a_{mn}+b_{mn} \end{bmatrix} \quad (\mathbf{A}+\mathbf{B})^{\mathrm{T}} = \begin{bmatrix} a_{11}+b_{11} & a_{21}+b_{21} & \cdots & a_{m1}+b_{m1} \\ a_{12}+b_{12} & a_{22}+b_{22} & \cdots & a_{m2}+b_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n}+b_{1n} & a_{2n}+b_{2n} & \cdots & a_{mn}+b_{mn} \end{bmatrix} \tag{1}$$

$$\mathbf{A}^{\mathrm{T}} = \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{bmatrix}, \quad \mathbf{B}^{\mathrm{T}} = \begin{bmatrix} b_{11} & b_{21} & \cdots & b_{m1} \\ b_{12} & b_{22} & \cdots & b_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ b_{1n} & b_{2n} & \cdots & b_{mn} \end{bmatrix} \quad \mathbf{A}^{\mathrm{T}}+\mathbf{B}^{\mathrm{T}} = \begin{bmatrix} a_{11}+b_{11} & a_{21}+b_{21} & \cdots & a_{m1}+b_{m1} \\ a_{12}+b_{12} & a_{22}+b_{22} & \cdots & a_{m2}+b_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n}+b_{1n} & a_{2n}+b_{2n} & \cdots & a_{mn}+b_{mn} \end{bmatrix} \tag{2}$$



EX-2: GIVEN TWO MATRICES A AND B, SHOW THAT SUM AND TRANSPOSITION COMMUTE: $\mathbf{A}^\top + \mathbf{B}^\top = (\mathbf{A}+\mathbf{B})^\top$

Given any two matrices A and B of the same size $\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix}$

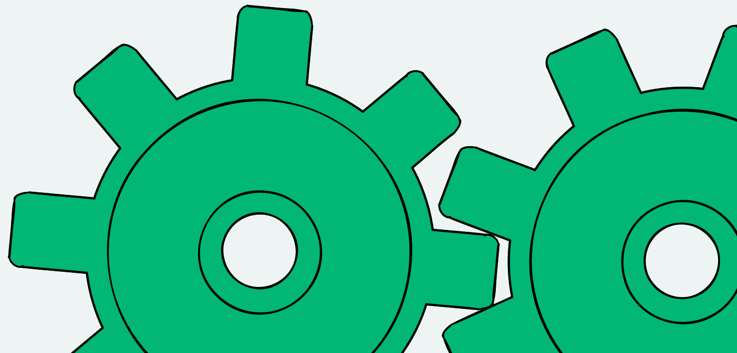
$$\mathbf{A}+\mathbf{B} = \begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & \cdots & a_{1n}+b_{1n} \\ a_{21}+b_{21} & a_{22}+b_{22} & \cdots & a_{2n}+b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}+b_{m1} & a_{m2}+b_{m2} & \cdots & a_{mn}+b_{mn} \end{bmatrix} \quad (\mathbf{A}+\mathbf{B})^\mathrm{T} = \begin{bmatrix} a_{11}+b_{11} & a_{21}+b_{21} & \cdots & a_{m1}+b_{m1} \\ a_{12}+b_{12} & a_{22}+b_{22} & \cdots & a_{m2}+b_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n}+b_{1n} & a_{2n}+b_{2n} & \cdots & a_{mn}+b_{mn} \end{bmatrix} \tag{1}$$

$$\mathbf{A}^\mathrm{T} = \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{bmatrix}, \quad \mathbf{B}^\mathrm{T} = \begin{bmatrix} b_{11} & b_{21} & \cdots & b_{m1} \\ b_{12} & b_{22} & \cdots & b_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ b_{1n} & b_{2n} & \cdots & b_{mn} \end{bmatrix} \mathbf{A}^\mathrm{T}+\mathbf{B}^\mathrm{T} = \begin{bmatrix} a_{11}+b_{11} & a_{21}+b_{21} & \cdots & a_{m1}+b_{m1} \\ a_{12}+b_{12} & a_{22}+b_{22} & \cdots & a_{m2}+b_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n}+b_{1n} & a_{2n}+b_{2n} & \cdots & a_{mn}+b_{mn} \end{bmatrix} \tag{2}$$

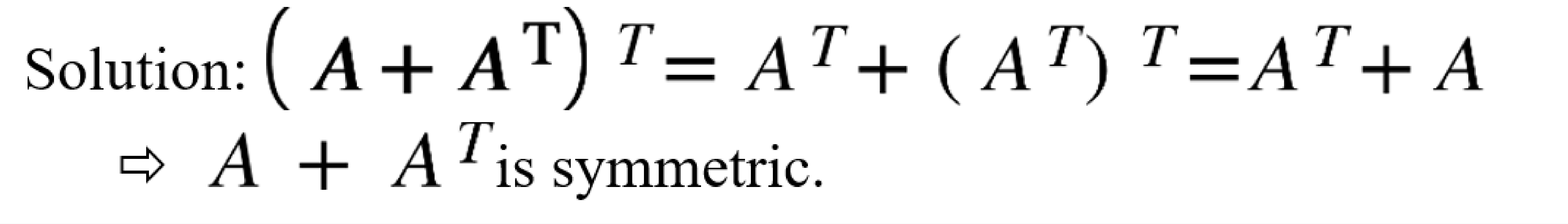
Since (1) and (2) are identical, we conclude that: $\mathbf{A}^\top + \mathbf{B}^\top = (\mathbf{A}+\mathbf{B})^\top$

EX-2: GIVEN TWO MATRICES A AND B, SHOW THAT SUM AND TRANSPOSITION COMMUTE: $\mathbf{A}^\top + \mathbf{B}^\top = (\mathbf{A} + \mathbf{B})^\top$

```
import torch
A = torch.randn(100, 200,
dtype=torch.float64)
B = A.clone()
A.T + B.T == (A + B).T

tensor([[True, True, True, ..., True, True, True],
[True, True, True, ..., True, True, True], [True,
True, True, ..., True, True, True], ..., [True, True,
True, ..., True, True, True], [True, True, True, ...,
True, True, True], [True, True, True, ..., True,
True, True]])
```



EXERCISE 3: GIVEN ANY SQUARE MATRIX A, IS $A + A^T$ ALWAYS SYMMETRIC? CAN YOU PROVE THE RESULT BY USING ONLY THE RESULTS OF THE PREVIOUS TWO EXERCISES?



Solution: $\left(\mathbf{A}+\mathbf{A}^{\mathrm{T}}\right)^{T} = A^{T} + (A^{T})^{T} = A^{T} + A$

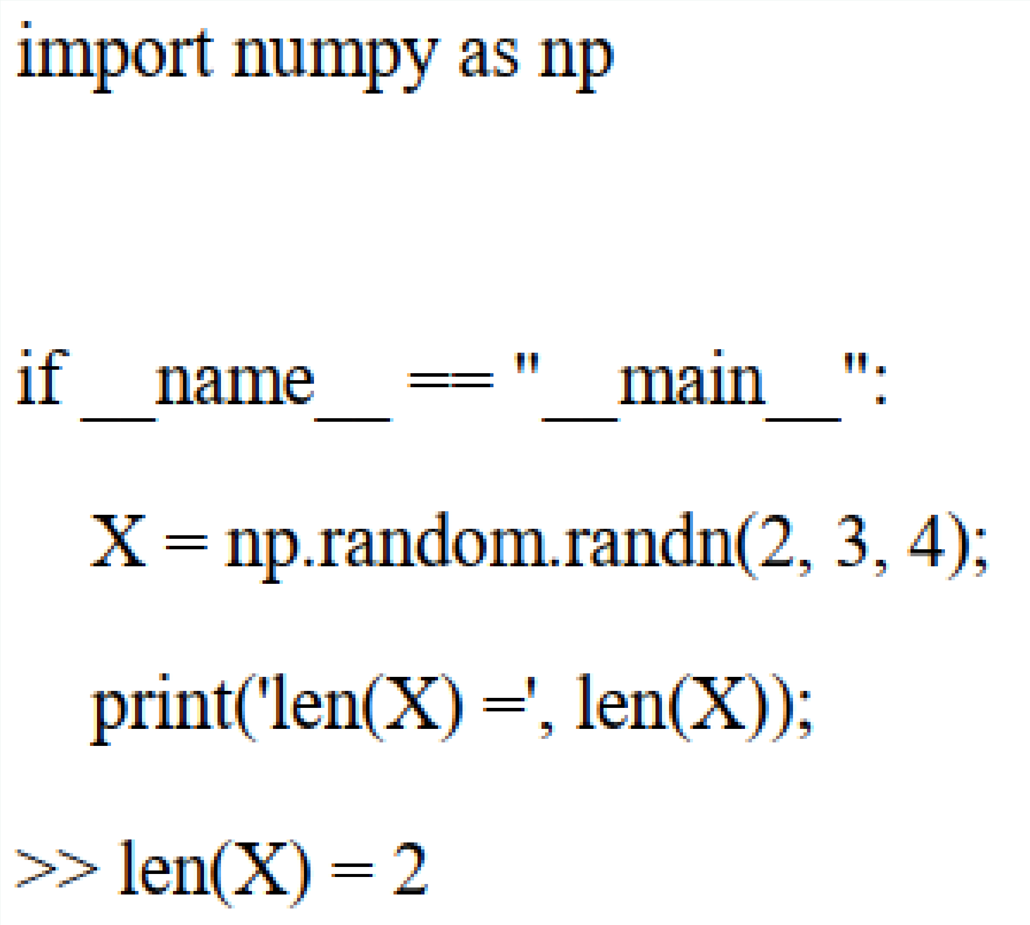
⇨ $A + A^{T}$ is symmetric.

EX-4: We defined the tensor X of shape (2, 3, 4) in this section. What is the output of len(X)? Write your answer without implementing any code, then check your answer using code.

When we define a tensor X with shape (2, 3, 4), it means:

- 2 blocks (or batches),
- each with 3 rows,
- and each row has 4 elements.

So, len(X) should return 2.



```
import numpy as np


if __name__ == "__main__":
    X = np.random.randn(2, 3, 4);
    print('len(X) =', len(X));
>> len(X) = 2
```

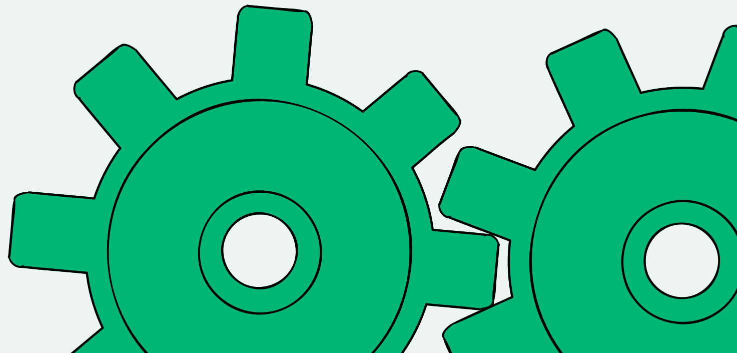
E5: For a tensor X of arbitrary shape, does $len(X)$ always correspond to the length of a certain axis of X? What is that axis?

To determine whether $len(X)$ corresponds to the length of a specific axis, we consider tensors of various orders:

- **Scalar**: A scalar has no axes, no dimensions, $len(X)$ is undefined or not applicable in standard linear algebra. Therefore, $len(X)$ cannot correspond to any axis.

- **Vector**: A vector has one axis, with shape (d_1), where d_1 is the number of components. $len(X) = d_1$ corresponds to the first (and only) axis, indexed as 0.

- **Matrix**: A matrix has two axes, with shape (d_1, d_2), where d_1 is the number of rows and d_2 is the number of columns. Because matrices are often viewed as collections of row vectors and the number of rows is a primary structural characteristic, $len(X) = d_1$ corresponds to the first axis (index 0).

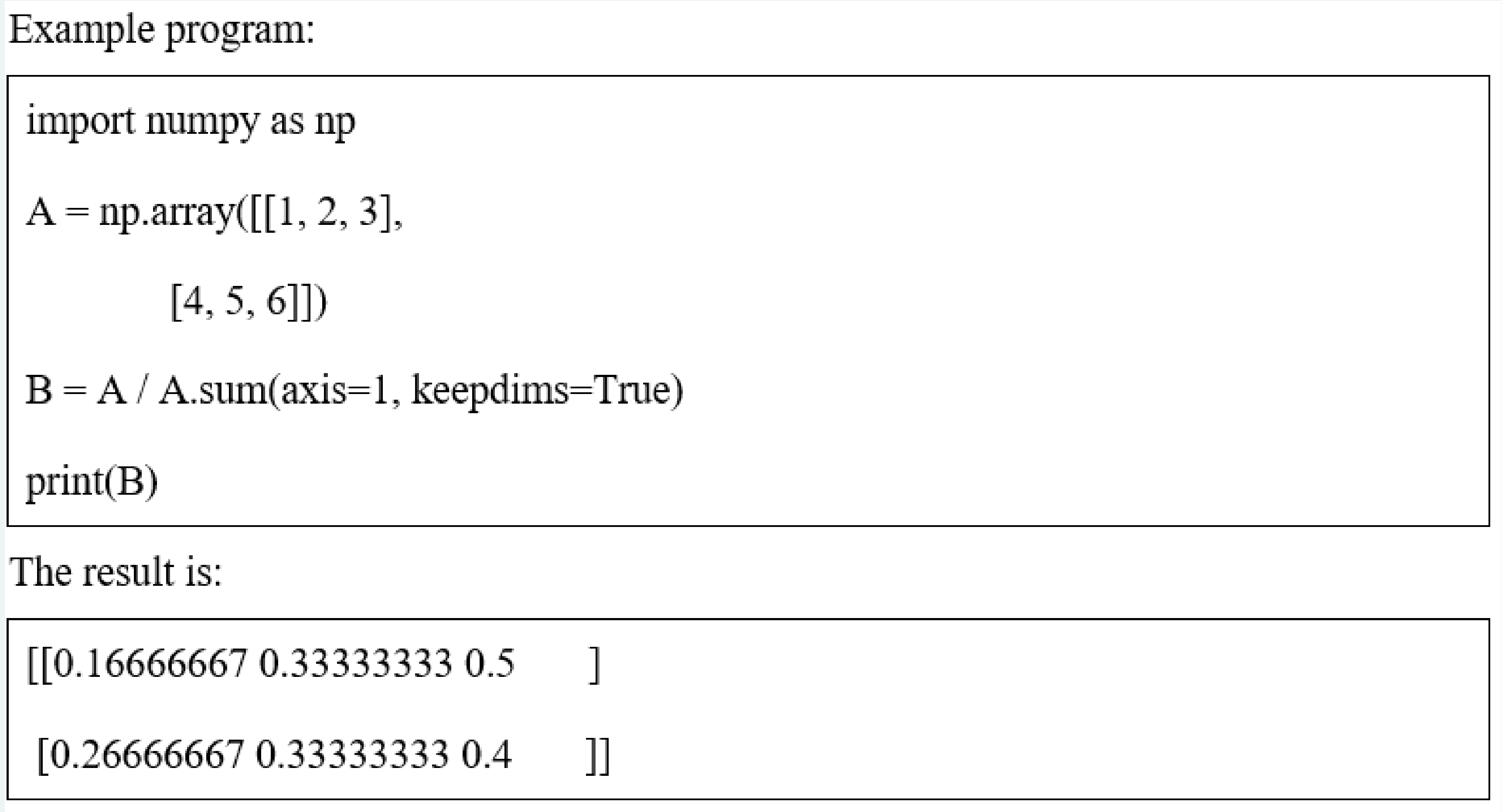


E5: For a tensor X of arbitrary shape, does $len(X)$ always correspond to the length of a certain axis of X? What is that axis?

– **Tensor**: A tensor has shape $(d_1, d_2,...,d_N)$, with N axes. For a general N–th order tensor, $len(X)$ is most naturally associated with d_1, the size of the first axis, as this aligns with the hierarchical organization of tensors where the first index governs the outermost grouping. $len(X) = d_1$ corresponds to the first axis (index 0).

For **a tensor** X **of arbitrary shape** with **at least one dimension**, $len(X)$ corresponds to the length of the first axis (index 0), which is the size of the outermost dimension in the tensor’s shape, denoted d_1 for shape $(d_1, d_2,...,d_N)$.
For **a scalar**, $len(X)$ is undefined, as it has no axes.

EX-6: RUN $\mathbf{A}/\mathbf{A}.\,\mathrm{sum}(\mathrm{axis} = 1)$ AND SEE WHAT HAPPENS. CAN YOU ANALYZE THE RESULT?

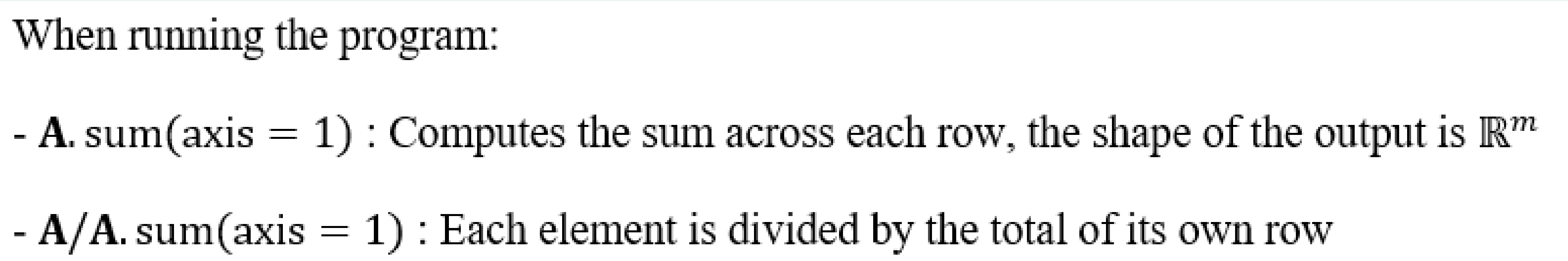


Example program:

```
import numpy as np

A = np.array([[1, 2, 3],

           [4, 5, 6]])

B = A / A.sum(axis=1, keepdims=True)

print(B)
```

The result is:

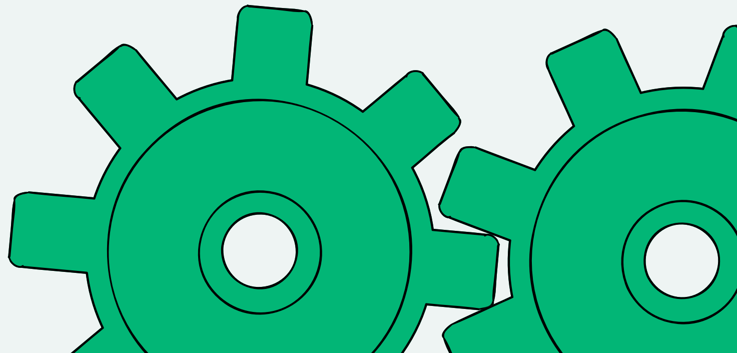
```
[[0.16666667 0.33333333 0.5       ]

 [0.26666667 0.33333333 0.4       ]]
```



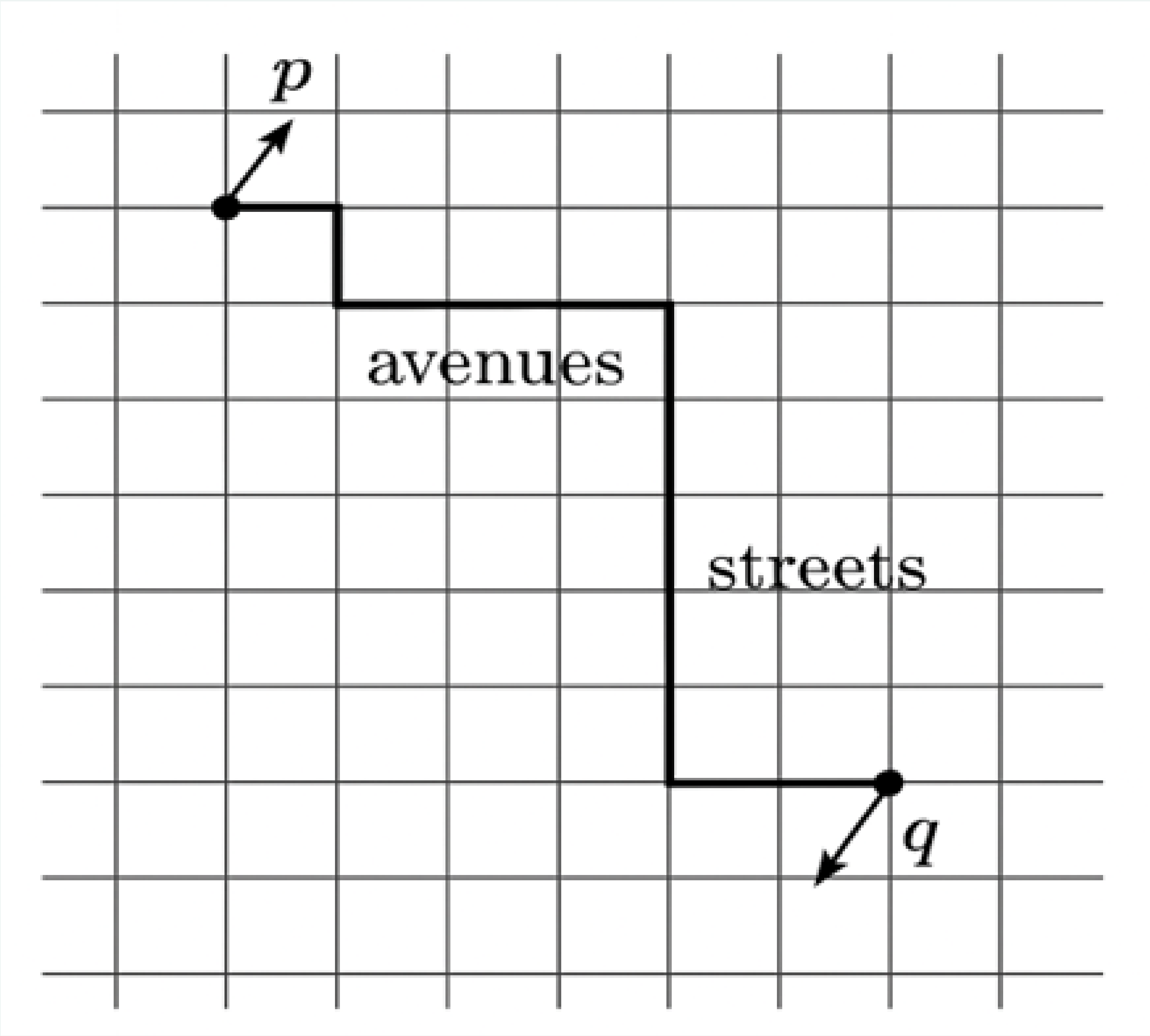
When running the program:

- $\mathbf{A}.\,\mathrm{sum}(\mathrm{axis} = 1)$: Computes the sum across each row, the shape of the output is $\mathbb{R}^m$

- $\mathbf{A}/\mathbf{A}.\,\mathrm{sum}(\mathrm{axis} = 1)$: Each element is divided by the total of its own row

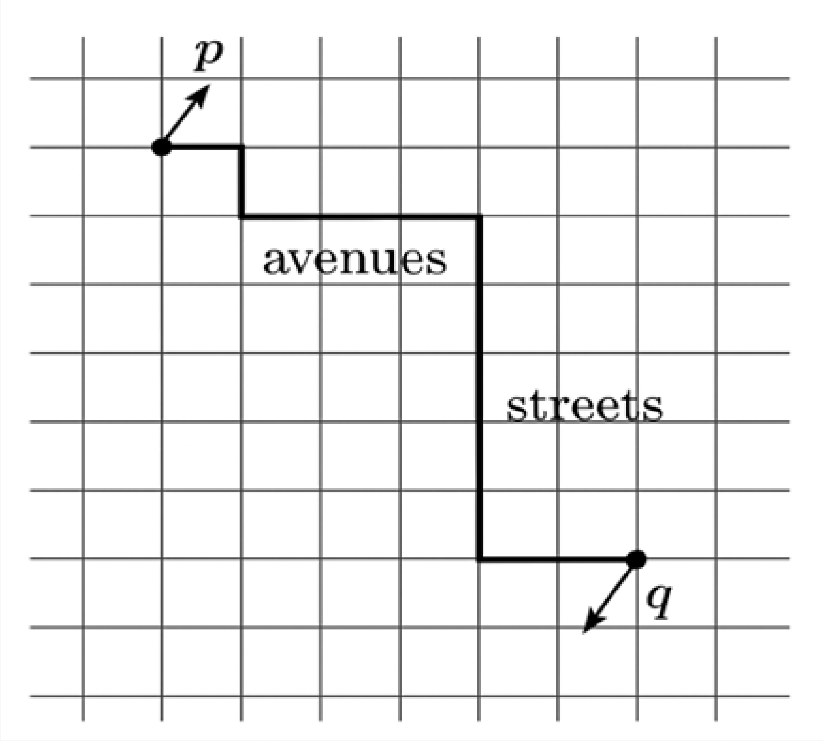


EX-7:WHEN TRAVELING BETWEEN TWO POINTS IN DOWNTOWN MANHATTAN, WHAT IS THE DISTANCE THAT YOU NEED TO COVER IN TERMS OF THE COORDINATES, I.E., IN TERMS OF AVENUES AND STREETS? CAN YOU TRAVEL DIAGONALLY?



Point p: starting point.
Point q: destination.
The vertical axes are avenues,
the horizontal axes are streets

EX-7:WHEN TRAVELING BETWEEN TWO POINTS IN DOWNTOWN MANHATTAN, WHAT IS THE DISTANCE THAT YOU NEED TO COVER IN TERMS OF THE COORDINATES, I.E., IN TERMS OF AVENUES AND STREETS? CAN YOU TRAVEL DIAGONALLY?



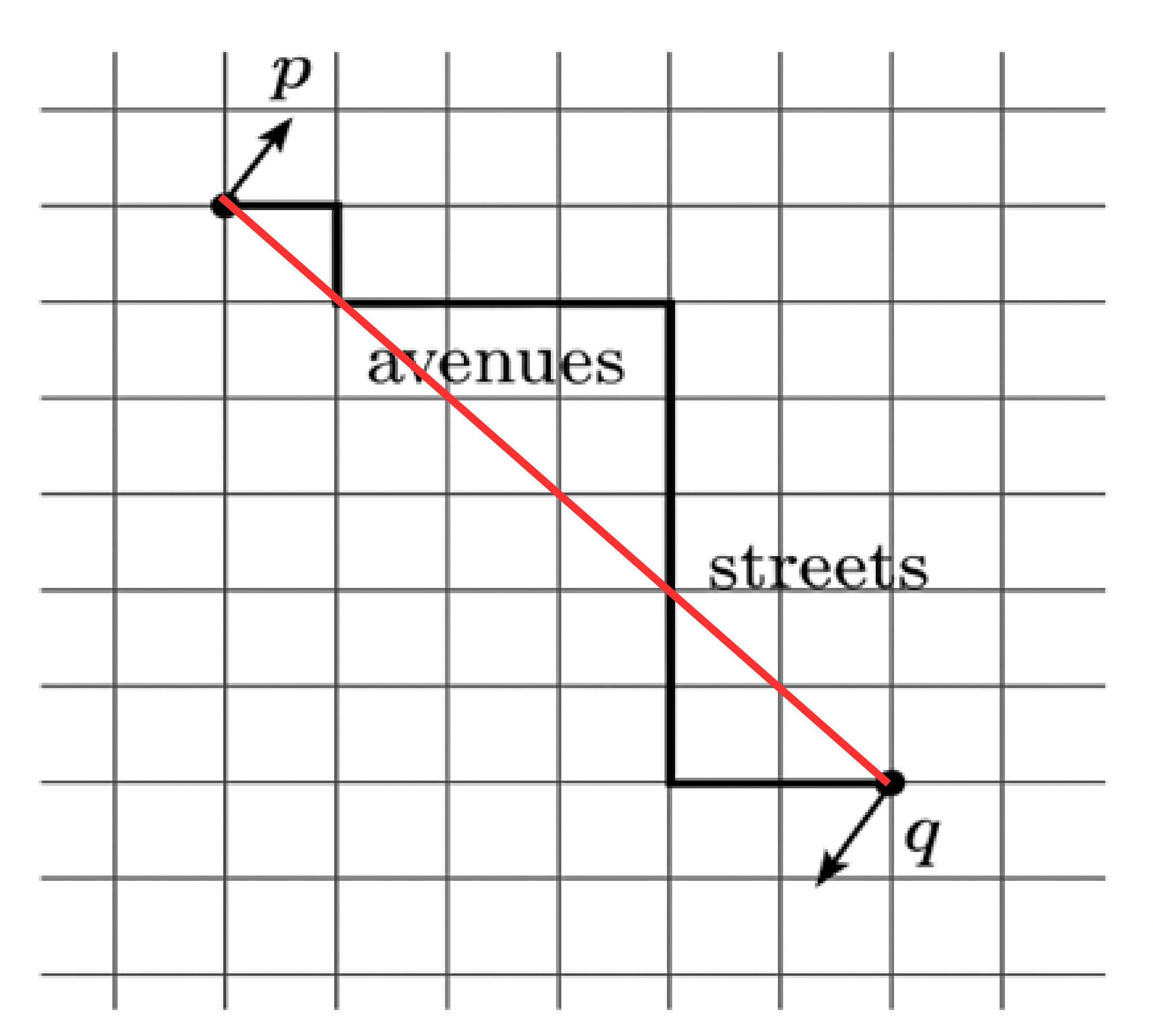
Point p: starting point.
Point q: destination.
The vertical axes are avenues, the horizontal axes are streets

Given two points $p=(p_1,...,p_n)\,,q=(q_1,...,q_n)$ in an n-dimensional space with real coordinates, , where the coordinates represent intersections on a grid (for example, avenues and streets in Manhattan), the Manhattan distance d between these points is calculated as follows:

$$d\,(p,q)=\parallel p-q\parallel=\sum_{i=1}^{n}|p_i-q_i|$$

This equation sums the absolute differences in their coordinates p (avenues) and q (streets)

EX-7:WHEN TRAVELING BETWEEN TWO POINTS IN DOWNTOWN MANHATTAN, WHAT IS THE DISTANCE THAT YOU NEED TO COVER IN TERMS OF THE COORDINATES, I.E., IN TERMS OF AVENUES AND STREETS? CAN YOU TRAVEL DIAGONALLY?



In the context of **Manhattan**, or any other location with a **grid-like structure**, this formula effectively measures the shortest path where only **orthogonal turns (right angles)** are allowed, mimicking the actual movement along the city’s streets and avenues.

Traveling diagonally in the sense of **moving directly through buildings is not possible** under this metric in a strict grid system. However, certain streets like **Broadway** or other diagonal roads may provide some leeway where such direct paths are possible.

Exercise 8: Consider a tensor of shape (2, 3, 4). What are the shapes of the summation outputs along axes 0, 1, and 2 ?

SOLUTION:
A TENSOR OF SHAPE (2, 3, 4).
AXIS 0: 2 ELEMENTS (BLOCKS)
AXIS 1: 3 ELEMENTS (ROWS)
AXIS 2: 4 ELEMENTS (COLUMNS)

SUMMATION ALONG AXIS 0:

- SUMMING OVER AXIS 0 REMOVES THE FIRST DIMENSION (2 → 1).
- THE REMAINING AXES: (3, 4)

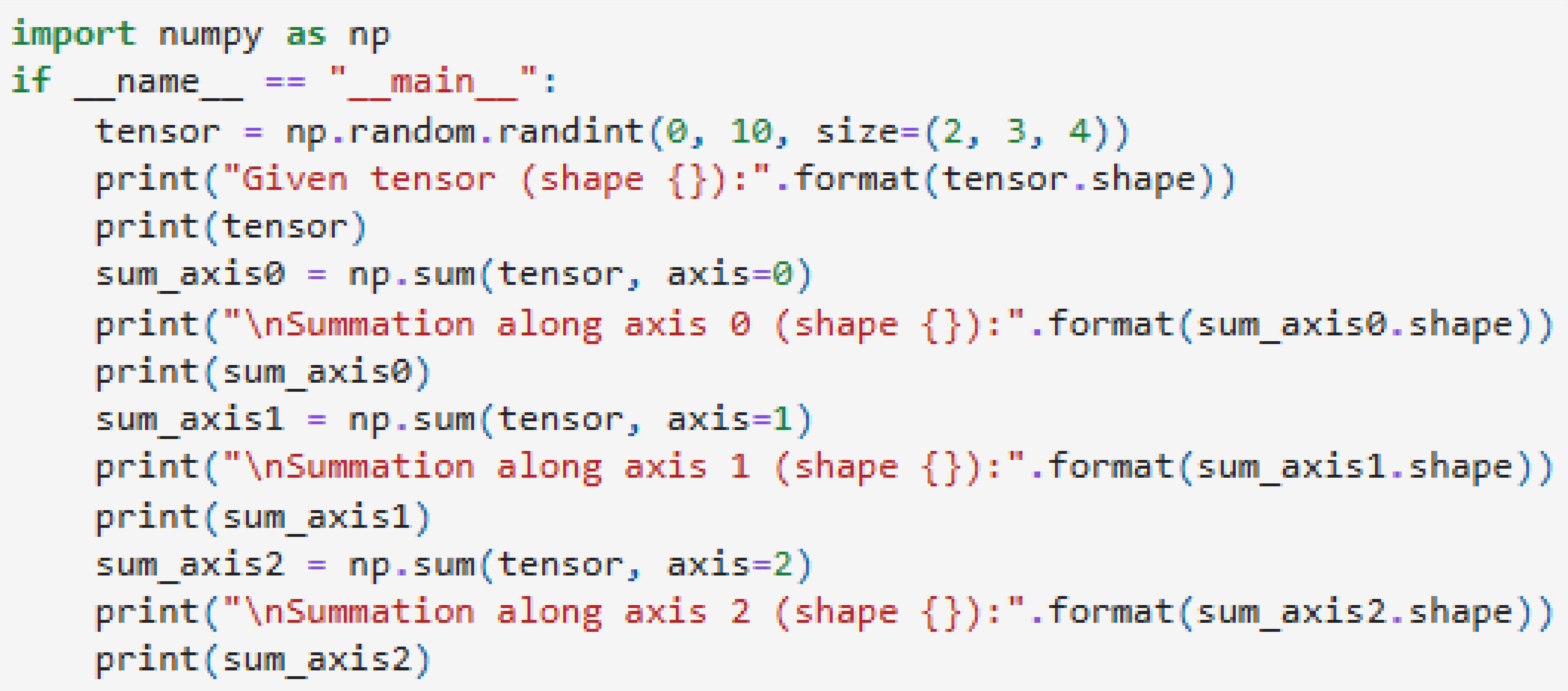
SUMMATION ALONG AXIS 1:

· SUMMING OVER AXIS 1 REMOVES THE SECOND DIMENSION (3 → 1).
· REMAINING SHAPE: (2, 4)

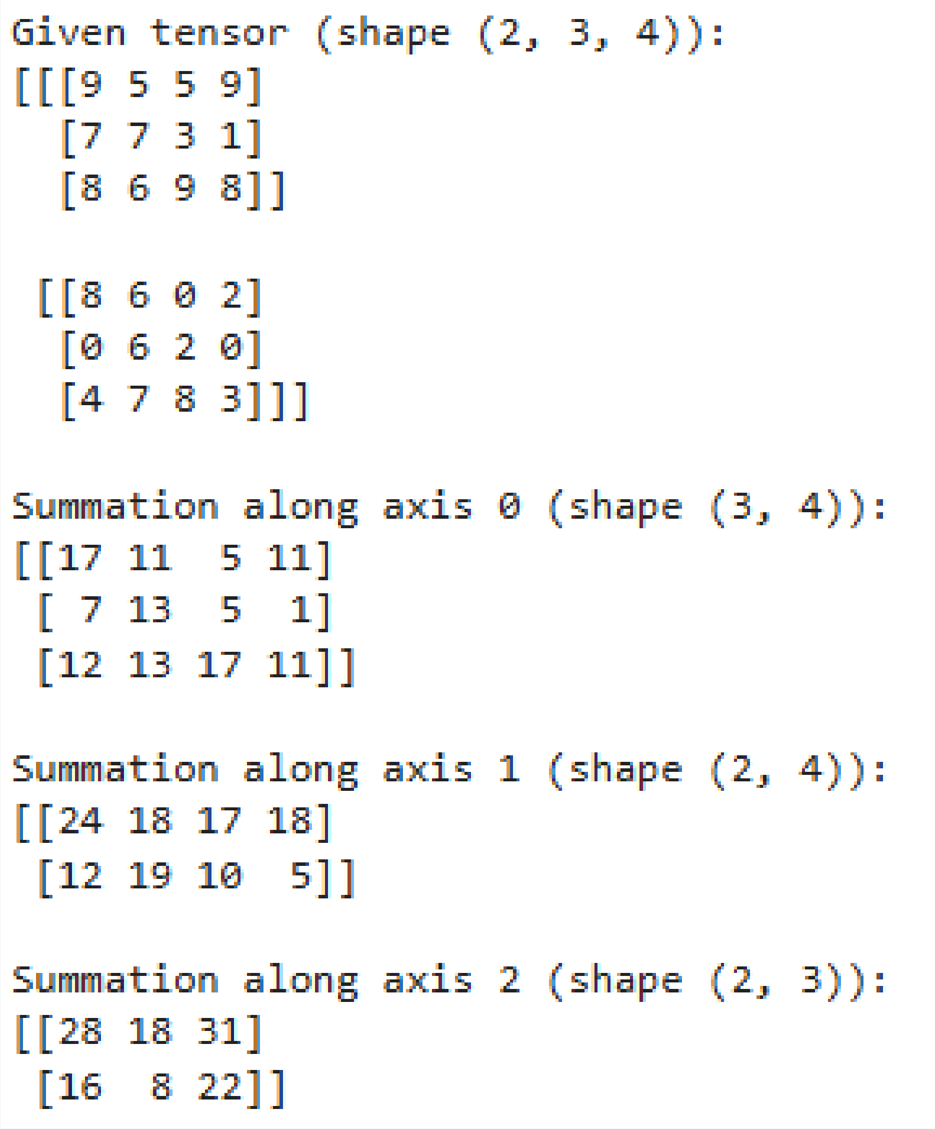
SUMMATION ALONG AXIS 2:

· SUMMING OVER AXIS 2 REMOVES THE THIRD DIMENSION (4 → 1).
· REMAINING SHAPE: (2, 3)

Exercise 8: Consider a tensor of shape (2, 3, 4). What are the shapes of the summation outputs along axes 0, 1, and 2 ?



```python
import numpy as np
if __name__ == "__main__":
    tensor = np.random.randint(0, 10, size=(2, 3, 4))
    print("Given tensor (shape {}):".format(tensor.shape))
    print(tensor)
    sum_axis0 = np.sum(tensor, axis=0)
    print("\nSummation along axis 0 (shape {}):".format(sum_axis0.shape))
    print(sum_axis0)
    sum_axis1 = np.sum(tensor, axis=1)
    print("\nSummation along axis 1 (shape {}):".format(sum_axis1.shape))
    print(sum_axis1)
    sum_axis2 = np.sum(tensor, axis=2)
    print("\nSummation along axis 2 (shape {}):".format(sum_axis2.shape))
    print(sum_axis2)
```



```
Given tensor (shape (2, 3, 4)):
[[[9 5 5 9]
  [7 7 3 1]
  [8 6 9 8]]

 [[8 6 0 2]
  [0 6 2 0]
  [4 7 8 3]]]

Summation along axis 0 (shape (3, 4)):
[[17 11  5 11]
 [ 7 13  5  1]
 [12 13 17 11]]

Summation along axis 1 (shape (2, 4)):
[[24 18 17 18]
 [12 19 10  5]]

Summation along axis 2 (shape (2, 3)):
[[28 18 31]
 [16  8 22]]
```

EX-9: Feed a tensor with three or more axes to the linalg.norm function and observe its output. What does this function compute for tensors of arbitrary shape?

Assume a tensor X has shape (2, 3, 4):

$$X = \begin{bmatrix} \begin{bmatrix} 9 & 3 & 6 & 8 \\ 4 & 8 & 9 & 5 \\ 0 & 7 & 8 & 3 \end{bmatrix} \\ \begin{bmatrix} 4 & 3 & 4 & 7 \\ 2 & 0 & 7 & 1 \\ 5 & 4 & 4 & 7 \end{bmatrix} \end{bmatrix}$$

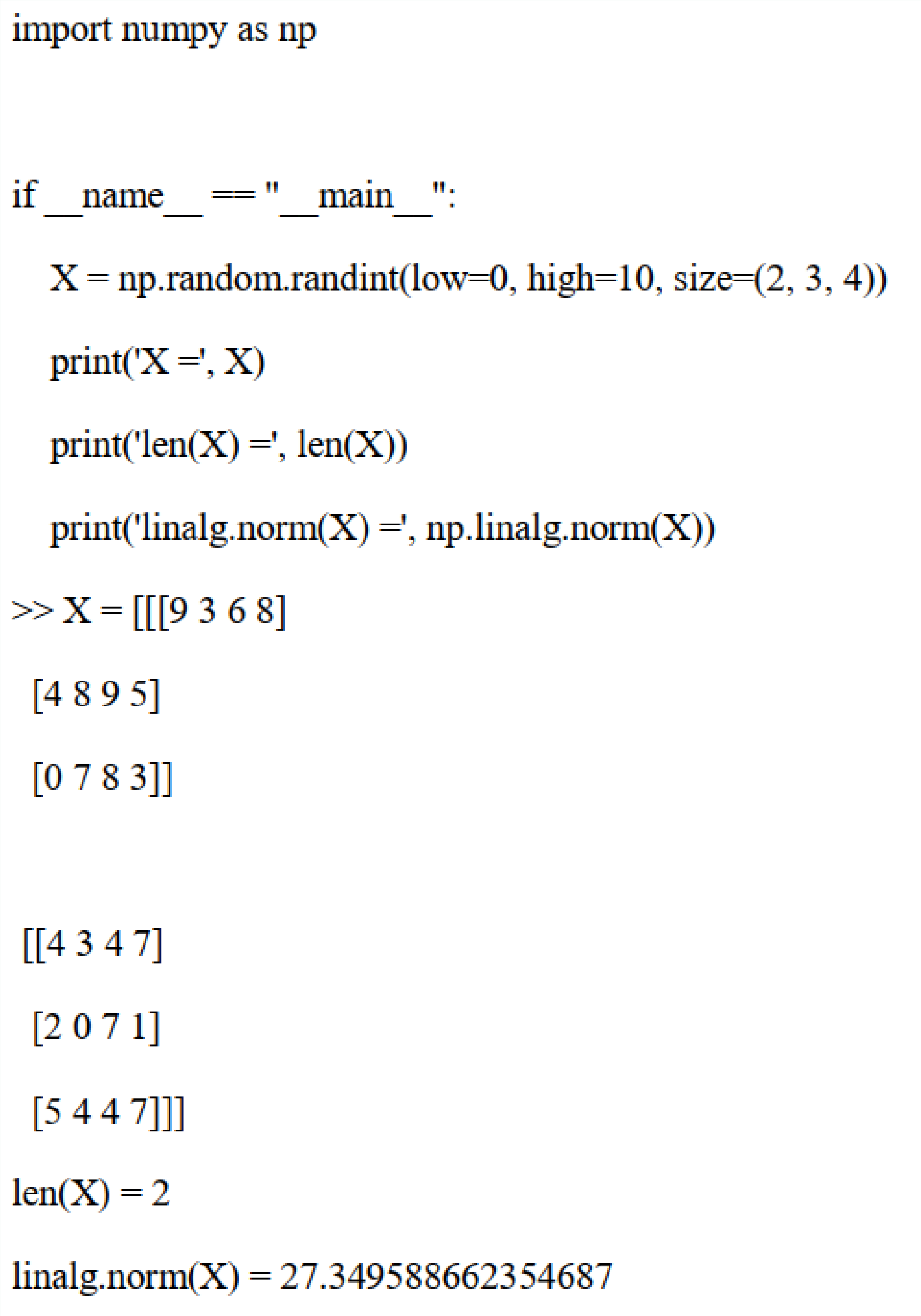
The **linalg.norm(X)** function will:

- Flatten the entire tensor into a 1D vector (24 numbers).
- Compute the sum of the squares of all 24 numbers.
- Take the square root of that sum.

This is called Frobenius norm (standard Euclidean norm). Mathematically:

$$\|\mathbf{X}\|_F = \sqrt{\sum_{i=1}^{2}\sum_{j=1}^{3}\sum_{k=1}^{4}\mathbf{X}[i]\,[j]\,[k]} = \sqrt{9^2 + 3^2 + 6^2 + \ \dots} \approx 27.349589$$

EX-9: Feed a tensor with three or more axes to the linalg.norm function and observe its output. What does this function compute for tensors of arbitrary shape?



```
import numpy as np


if __name__ == "__main__":

   X = np.random.randint(low=0, high=10, size=(2, 3, 4))

   print('X =', X)

   print('len(X) =', len(X))

   print('linalg.norm(X) =', np.linalg.norm(X))

>> X = [[[9 3 6 8]

  [4 8 9 5]

  [0 7 8 3]]


 [[4 3 4 7]

  [2 0 7 1]

  [5 4 4 7]]]

len(X) = 2

linalg.norm(X) = 27.349588662354687
```

E10: Consider three large matrices, say $\mathbf{A} \in \mathbb{R}^{2^{10} \times 2^{16}}, \mathbf{B} \in \mathbb{R}^{2^{16} \times 2^{5}}, \mathbf{C} \in \mathbb{R}^{2^{5} \times 2^{14}}$ initialized with Gaussian random variables. You want to compute the product $\mathbf{ABC}$. Is there any difference in memory footprint and speed, depending on whether you compute $(\mathbf{AB})\mathbf{C}$ or $\mathbf{A}(\mathbf{BC})$. Why?

To determine if there's a difference in memory footprint and speed, we analyze the matrix multiplications in terms of operation counts and intermediate matrix sizes:

# Speed

- For $(\mathbf{AB})\mathbf{C}$:

  + $\mathbf{AB}$: Operations = $2^{10} \times 2^{16} \times 2^{5} = 2^{31}$
  + $(\mathbf{AB})\mathbf{C}$: Operations = $2^{10} \times 2^{5} \times 2^{14} = 2^{29}$
  => Total Operations = $2^{31} + 2^{29} = 5 \times 2^{29}$

- For $\mathbf{A}(\mathbf{BC})$:

  + $\mathbf{BC}$: Operations = $2^{16} \times 2^{5} \times 2^{14} = 2^{35}$
  + $\mathbf{A}(\mathbf{BC})$: Operations = $2^{10} \times 2^{16} \times 2^{14} = 2^{40}$
  =>Total Operations = $2^{35} + 2^{40} = 33 \times 2^{35}$

Comparing operation counts:

$\mathbf{A}(\mathbf{BC})$ requires about 422.4 times more operations, making $(\mathbf{AB})\mathbf{C}$ much faster.

E10: Consider three large matrices, say $\mathbf{A} \in \mathbb{R}^{2^{10} \times 2^{16}}, \mathbf{B} \in \mathbb{R}^{2^{16} \times 2^{5}}, \mathbf{C} \in \mathbb{R}^{2^{5} \times 2^{14}}$ initialized with Gaussian random variables. You want to compute the product $\mathbf{ABC}$. Is there any difference in memory footprint and speed, depending on whether you compute $(\mathbf{AB})\mathbf{C}$ or $\mathbf{A}(\mathbf{BC})$. Why?

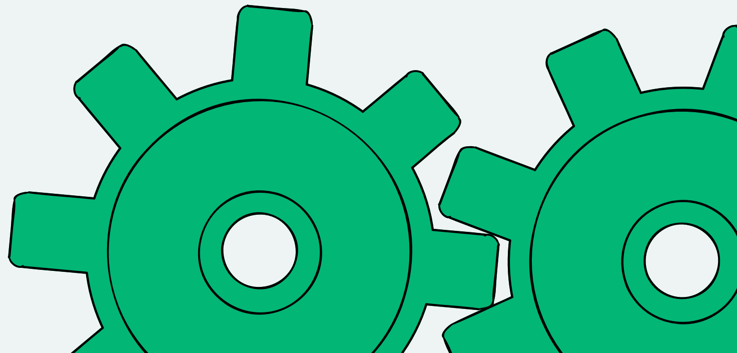
# Memory Footprint

Memory usage depends on the size of intermediate matrices:

- For $(\mathbf{AB})\mathbf{C}$:
  + Intermediate $\mathbf{AB}$: $2^{10} \times 2^{5} = 2^{15}$ elements
  + Final result: $2^{10} \times 2^{14} = 2^{24}$ elements
  => Peak memory: 2^{24} if AB is overwritten
- For $\mathbf{A}(\mathbf{BC})$:
  + Intermediate $\mathbf{BC}$: $2^{14} \times 2^{16} = 2^{30}$ elements
  + Final result: $2^{10} \times 2^{14} = 2^{24}$ elements
  => Peak memory: 2^{30} if $\mathbf{BC}$ is overwritten

Comparing intermediate storage:

$\mathbf{A}(\mathbf{BC})$ requires significantly more memory for the intermediate result when comparing with $(\mathbf{AB})\mathbf{C}$.



E10: Consider three large matrices, say $\mathbf{A} \in \mathbb{R}^{2^{10} \times 2^{16}}, \mathbf{B} \in \mathbb{R}^{2^{16} \times 2^{5}}, \mathbf{C} \in \mathbb{R}^{2^{5} \times 2^{14}}$ initialized with Gaussian random variables. You want to compute the product $\mathbf{ABC}$. Is there any difference in memory footprint and speed, depending on whether you compute $(\mathbf{AB})\mathbf{C}$ or $\mathbf{A}(\mathbf{BC})$. Why?

Why Differences?

The difference arises because the intermediate matrix sizes and operation counts depend on the order of multiplication:

- In $(\mathbf{AB})\mathbf{C}$, the intermediate $(\mathbf{AB})$ is small ($2^{10} \times 2^{5}$), and the number of operations is minimized.
- In $\mathbf{A}(\mathbf{BC})$, the intermediate $(\mathbf{BC})$ is large ($2^{16} \times 2^{14}$), and the operation count is much higher due to the large dimensions involved in each step.

EX-11: CONSIDER THREE LARGE MATRICES, SAY $\mathbf{A} \in \mathbb{R}^{2^{10}\times 2^{16}}$,$\mathbf{B} \in \mathbb{R}^{2^{16}\times 2^{5}}$ AND $\mathbf{C} \in \mathbb{R}^{2^{5}\times 2^{16}}$. IS THERE ANY DIFFERENCE IN SPEED DEPENDING ON WHETHER YOU COMPUTE $\mathbf{AB}$ OR $\mathbf{AC}^T$? WHY? WHAT CHANGES IF YOU INITIALIZE $\mathbf{C} = \mathbf{B}^T$ WITHOUT CLONING MEMORY? WHY?

When computing **AB**:

- Shape: $(2^{10} \times 2^{16}) \cdot (2^{16} \times 2^{5})$

- Result: $2^{10} \times 2^{5}$

- Number of scalar multiplications: $2^{10} \times 2^{16} \times 2^{5}$=$2^{31}$

When computing $\mathbf{AC}^T$:

- $\mathbf{C} \in \mathbb{R}^{2^5\times 2^{16}} \rightarrow \boldsymbol{C}^T \in \mathbb{R}^{2^{16}\times 2^{5}}$

- So $\mathbf{AC}^T$ has the same result as **AB**

- Number of scalar multiplications: $2^{10} \times 2^{16} \times 2^{5}$=$2^{31}$

→ There is no difference between $\mathbf{AC}^T$ and **AB** about computing, but $\mathbf{AC}^T$ may need more computing time due to extra step (transposing).

If we initialize $\mathbf{C} = \mathbf{B}^T$, this will create a view, not a new matrix. So:

- No extra memory is allocated

- **C** and **B** share the same data buffer, just with different strides.

EX-12:CONSIDER THREE MATRICES, SAY A,B,C $\in \mathbb{R}^{100x200}$. CONSTRUCT A TENSOR WITH THREE AXES BY STACKING [A,B,C]. WHAT IS THE DIMENSIONALITY? SLICE OUT THE SECOND COORDINATE OF THE THIRD AXIS TO RECOVER B. CHECK THAT YOUR ANSWER IS CORRECT.

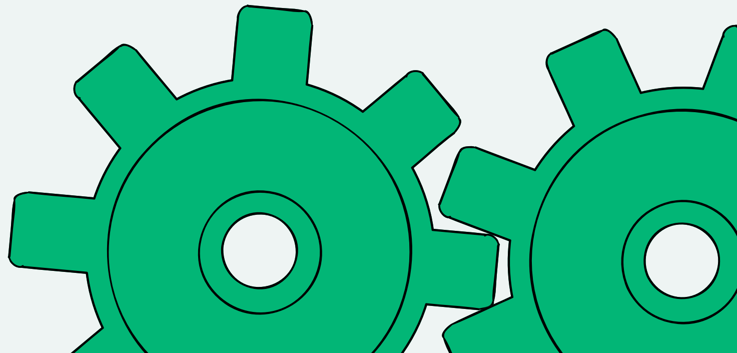
```
iimport torch
A = torch.randn(100, 200, dtype=torch.float64)
B = torch.randn(100, 200, dtype=torch.float64)
C = torch.randn(100, 200, dtype=torch.float64)
```

EX-12:CONSIDER THREE MATRICES, SAY A,B,C $\in \mathbb{R}^{100x200}$. CONSTRUCT A TENSOR WITH THREE AXES BY STACKING [A,B,C]. WHAT IS THE DIMENSIONALITY? SLICE OUT THE SECOND COORDINATE OF THE THIRD AXIS TO RECOVER B. CHECK THAT YOUR ANSWER IS CORRECT.

```
iimport torch
A = torch.randn(100, 200, dtype=torch.float64)
B = torch.randn(100, 200, dtype=torch.float64)
C = torch.randn(100, 200, dtype=torch.float64)

tensor_0 = torch.stack([A, B, C])
tensor_0.shape

torch.Size([3, 100, 200])
```

By default, torch.stack uses **axis =0**, which adds a new dimension at the front, resulting in a tensor shape of 3x100x200

- Axis 0 contains the matrices
    - tensor[0] = A
    - tensor[1] = B
    - tensor[2] = C
- Axis 1: row (100)
- Axis 2: column (200)



EX-12:CONSIDER THREE MATRICES, SAY A,B,C $\in \mathbb{R}^{100x200}$. CONSTRUCT A TENSOR WITH THREE AXES BY STACKING [A,B,C]. WHAT IS THE DIMENSIONALITY? SLICE OUT THE SECOND COORDINATE OF THE THIRD AXIS TO RECOVER B. CHECK THAT YOUR ANSWER IS CORRECT.

```
iimport torch
A = torch.randn(100, 200, dtype=torch.float64)
B = torch.randn(100, 200, dtype=torch.float64)
C = torch.randn(100, 200, dtype=torch.float64)

tensor_1 = torch.stack([A, B, C],axis=2)
tensor_1.shape

torch.Size([100, 200, 3])
```

- Axis 0: row (100),
- Axis 1: column (200)
- Axis 2 contains the matrices,
  - tensor[0] = A
  - **tensor[1] = B (second)**
  - tensor[2] = C

- When using torch.stack([A, B, C], **axis=2**), it will stack A, B, and C along axis 2 (**the third axis**)

EX-12:CONSIDER THREE MATRICES, SAY A,B,C $\in \mathbb{R}^{100x200}$. CONSTRUCT A TENSOR WITH THREE AXES BY STACKING [A,B,C]. WHAT IS THE DIMENSIONALITY? SLICE OUT THE SECOND COORDINATE OF THE THIRD AXIS TO RECOVER B. CHECK THAT YOUR ANSWER IS CORRECT.

```
iimport torch
A = torch.randn(100, 200, dtype=torch.float64)
B = torch.randn(100, 200, dtype=torch.float64)
C = torch.randn(100, 200, dtype=torch.float64)

tensor_1 = torch.stack([A, B, C],axis=2)
tensor_1.shape

torch.Size([100, 200, 3])
```

- Axis 0: row (100),
- Axis 1: column (200)
- Axis 2 contains the matrices,
  - tensor[0] = A
  - **tensor[1] = B (second)**
  - tensor[2] = C

- When using torch.stack([A, B, C], **axis=2**), it will stack A, B, and C along axis 2 (**the third axis**)

The correct answer for the dimension (shape) would be $\mathbb{R}^{100x200x3}; \mathbb{R}^{3x100x200}$

EX-12:CONSIDER THREE MATRICES, SAY A,B,C $\in \mathbb{R}^{100x200}$. CONSTRUCT A TENSOR WITH THREE AXES BY STACKING [A,B,C]. WHAT IS THE DIMENSIONALITY? SLICE OUT THE SECOND COORDINATE OF THE THIRD AXIS TO RECOVER B. CHECK THAT YOUR ANSWER IS CORRECT.

```
iimport torch
A = torch.randn(100, 200, dtype=torch.float64)
B = torch.randn(100, 200, dtype=torch.float64)
C = torch.randn(100, 200, dtype=torch.float64)
tensor_1 = torch.stack([A, B, C],axis=2)
tensor_1[:, :, 1] == B
```

```
tensor([[True, True, True, ..., True, True, True], [True,
True, True, ..., True, True, True], [True, True, True, ...,
True, True, True], ..., [True, True, True, ..., True, True,
True], [True, True, True, ..., True, True, True], [True,
True, True, ..., True, True, True]])
```

Slice the second coordinate of the third axis (axis 2) is index = 1, corresponding to B

The result returned is a matrix of True values. Therefore, the statement is correct.

CONCLUSION

WHAT WE COVERED

- Reviewed fundamental linear algebra concepts (scalars, vectors, matrices, tensors) and their role in data representation.

WHAT WE FOCUSED ON

- Highlighted core operations: dot product, reductions, matrix – vector and matrix – matrix multiplication, with applications for each sections.

WHY IT MATTERS

- Understanding linear algebra helps us design better models, interpret their behavior, and make them run more efficiently – making it a must – have skill in deep learning.

Q/A