VIETNAM NATIONAL UNIVERSITY – HO CHI MINH CITY

UNIVERSITY OF TECHNOLOGY

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

_____o0o_____

REPORT

LINEAR ALGEBRA

Supervisor: Dr. Nguyen An Khuong

Student group: Vo Duong Xuan Nguyen - 2470738

Nguyen Vo Thai Trieu - 2470577

Pham The Long - 2470752

Pham Quynh Tran - 2470731

Nguyen Cong Thanh - 2470744

HO CHI MINH CITY, APRIL 2025

Table of Contents

I. Introduction

1. Overview of Linear Algebra in Deep Learning

Linear algebra plays a central role in deep learning by providing the tools to represent and manipulate data in structured forms such as scalars, vectors, matrices, and tensors. These mathematical objects are used throughout deep learning models—from the inputs and weights to the computed gradients during training.

Rather than being purely theoretical, these concepts are directly applied in practice. Most computations in deep learning are built on tensor operations, which are extensions of linear algebra operations. For this reason, understanding linear algebra is essential for interpreting how models work and for implementing them efficiently using frameworks like PyTorch.

2. Importance and Applications

Linear algebra plays a key role in deep learning by offering clear, efficient tools for representing and performing complex operations. Core operations like matrix - vector and matrix - matrix multiplications are fundamental to neural networks, enabling data to be transformed across layers and contributing directly to the model's predictions. Additionally, vector norms, commonly used to quantify the size of vectors, also play a vital role in regularization methods that control parameter growth and help prevent overfitting.

Reduction operations - such as summing, averaging, or taking the maximum - are commonly used when computing losses, generating class probabilities, or implementing attention mechanisms. These operations allow models to compress and aggregate information across different dimensions, such as features, time steps, or data samples. Optimization algorithms like gradient descent also rely heavily on linear algebra to compute gradients and update parameters. Understanding these operations is essential for effectively developing, interpreting, and debugging deep learning models.

3. Use cases in Computer Science

Beyond deep learning, linear algebra plays a critical role in many other areas of computer science. In computer graphics, matrix operations are used to perform geometric transformations such as translation, scaling, and rotation. In computer vision, techniques like convolutions, image filtering, and feature extraction depend on efficient manipulation of matrices.

In natural language processing, words are often represented as vectors, which can be compared using dot products and cosine similarity to capture semantic meaning. In recommendation systems, matrix factorization is used to model interactions between users and items. Many core techniques in data compression, signal processing, and scientific computing are built upon linear algebraic principles. These examples illustrate the wide applicability and lasting importance of linear algebra across a broad range of computational fields.

II. Fundamental Concepts

1. Scalars

1.1. Definition

A scalar is a single number. They are plain numbers 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 $\mathbb{R}$.

1.2. Computation in Practice

In PyTorch, we can use torch.tensor() to 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
```

We will get these results:

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

1.3. Applications

For example, the temperature in Palo Alto is a balmy 72 degrees Fahrenheit. This is the expression to convert Fahrenheit to Celsius. In this equation, 5, 9, and 32 are constants scalars. The variables c and f in general represent unknown scalars.

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

2. Vectors

2.1. Definition

Vector is a fixed-length array of scalars.

By default, we visualize vectors by stacking their elements vertically.

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

$x_1, \ldots, x_n$ 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.

2.2. 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])

2.3. Applications

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

3. Matrices

3.1. Definition

While scalars are 0^{th}-order tensors and vectors are 1^{st}-order tensors, matrices are 2^{nd}-order tensors. We denote matrices by bold capital letters (e.g. **X**, **Y** and **Z**). The expression $\mathbf{A} \in \mathbb{R}^{m \times n}$ indicates that a matrix **A** contains $m \times n$ real-valued scalars, arranged as m rows and n columns. When $m = n$, we say that a matrix is square.

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

To refer to an individual element, we subscript both the row and column indices.

E.g., a_{ij} is the value that belongs to **A**'s i^{th} row and j^{h} column.

We signify a matrix **A**'s transpose by $\mathbf{A}^{\top}$ and if $\mathbf{B} = \mathbf{A}^{\top}$, then $b_{ij} = a_{ji}$ for all i and j. Thus, the transpose of an $m \times n$ matrix is an $n \times m$ matrix:

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

Symmetric matrices are the subset of square matrices that are equal to their own transposes $\mathbf{A} = \mathbf{A}^{\mathrm{T}}$. The following matrix is symmetric:

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

3.2. Computation in Practice

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

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

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

3.3. Applications

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

4. Tensors

4.1 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.

4.2 Computation in Practice

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

```
import torch

# Scalar (0D tensor)

a = torch.tensor(3.14)

# Vector (1D tensor)

b = torch.tensor([1.0, 2.0, 3.0])

# Matrix (2D tensor)

c = torch.tensor([[1, 2], [3, 4]])

# 3D Tensor: 2 matrices of shape (2x2)

d = torch.tensor([[[1, 2], [3, 4]],

                  [[5, 6], [7, 8]]])
```

4.3 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).

III. Operations and Properties

1. Basic Properties of Tensor Arithmetic

1.1 Definition

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

Important concepts include:

- Element-wise addition and multiplication: Operate independently on corresponding elements of two tensors

+ Addition: $(A + B)_{ij} = A_{ij} + B_{ij}$

+ Multiplication: $(A \cdot B)_{ij} = A_{ij} \cdot B_{ij}$

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

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, B = [10 \quad 20] \Longrightarrow A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, B = \begin{bmatrix} 10 & 20 \\ 10 & 20 \end{bmatrix} \Longrightarrow A + B = \begin{bmatrix} 11 & 22 \\ 13 & 24 \end{bmatrix}$$

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

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

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

1.2 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

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
```

1.3 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)

2. Reduction Operations

2.1. Definition

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

Simply put, reduction is an operation that reduces the number of elements in a tensor. Reduction allows us to perform operations across the elements within a single tensor.

2.2. Computation in practice

Reduction operations decrease the number of elements in a tensor by aggregating values, such as calculating the sum, mean, or standard deviation. To illustrate this, consider the following tensor:

```
import torch

# Assume we have the following rank-2 tensor of shape 3 x 3:

A = torch.tensor([
```

```
    [0,1,0],
    [2,0,2],
    [0,3,0]
], dtype=torch.float32)
```

First, let's use the simplest reduction operation, sum():

```
print(A.sum())
>> tensor(8.)
```

The sum() method calculates the total of all elements, returning a tensor containing just a single scalar. We can confirm the reduction in tensor size:

```
print(A.numel())
print(A.sum().numel())
print(A.sum().numel() <= A.numel())
>> 9
1
True
```

As expected, the tensor resulting from the sum() operation has fewer elements, confirming it as a reduction operation.

Other common reduction operations include:

```
print(A.sum())   # sum of elements
print(A.prod())  # product of elements
print(A.mean())  # mean (average) of elements
print(A.std())   # standard deviation of elements
>> tensor(8.)
```

```
tensor(0.)

tensor(0.8889)

tensor(1.1667)
```

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).

For instance, consider a rank-2 tensor with shape 3 x 4:

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

Performing a sum along each column (axis=0):

```
print(A.shape)

print(A.sum(axis=0))

print(A.sum(axis=0).shape)

>> torch.Size([3, 4])

tensor([6., 6., 6., 6.])

torch.Size([4])
```

The resulting tensor shape (4,) confirms the reduction along axis 0.

Performing a sum along each row (axis=1):

```
print(A.sum(axis=1))
print(A.sum(axis=1).shape)
>> tensor([ 4., 8., 12.])
torch.Size([3])
```

The resulting tensor shape (3,) confirms the reduction along axis 1.

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

```
print(A.sum(axis=[0,1]))
print(A.sum())
>> tensor(24.)
tensor(24.)
```

Similarly, computing the mean (average) of tensor elements is straightforward:

```
print(A.mean())
print(A.sum() / A.numel())
>> tensor(2.)
tensor(2.)
```

You can also calculate the mean along specific axes, for example:

```
print(A.mean(axis=0))
print(A.sum(axis=0) / A.shape[0])
>> tensor([2., 2., 2., 2.])
tensor([2., 2., 2., 2.])
```

```
print(A.mean(axis=1))
print(A.sum(axis=1) / A.shape[1])
>> tensor([1., 2., 3.])
tensor([1., 2., 3.])
```

This clearly demonstrates how reduction operations allow you to aggregate tensor data flexibly across different axes, simplifying computations in practice.

2.3. Applications

Reduction operations are fundamental in data processing and deep learning workflows:

- Loss computation: Functions like mean() and sum() are used to aggregate per-sample losses into a scalar for backpropagation.

- Aggregation: max(), min(), or sum() help reduce features across dimensions, enabling operations like global pooling in CNNs.

- Dimensionality checks: Reduction along axes allows model diagnostics, feature summarization, and shape compatibility in broadcasting.

3. Non-Reduction Sum

3.1. Definition

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.

3.2. Computation in practice

Consider a tensor A defined as follows:

```
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)
print( A.shape)
>> torch.Size([5, 4])
```

This tensor has the shape (5,4), meaning it has 5 rows (axis 0) and 4 columns (axis 1). When summing along axis 1 without specifying keepdims, the resulting shape reduces to (5,), collapsing each row into a single scalar value:

```
print(A.sum(axis=1))
print(A.sum(axis=1).shape)
>> tensor([ 6., 22., 38., 54., 70.])
torch.Size([5])
```

If we set keepdims=True, the summed axis remains as a dimension of size 1, resulting in shape (5,1). Keeping this dimension can be particularly useful for subsequent calculations involving broadcasting, such as row-wise normalization:

```
sum_A = A.sum(axis=1, keepdims=True)
print(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]])
```

Another useful function, cumsum(axis), calculates the cumulative sum along the specified axis without reducing dimensions. For example, applying cumulative sum vertically (along axis 0) results in a tensor of the same shape as the original:

```
print(A.cumsum(axis=0))
>> tensor([[ 0., 1., 2., 3.],
[ 4., 6., 8., 10.],
[12., 15., 18., 21.],
[24., 28., 32., 36.],
[40., 45., 50., 55.]])
```

The resulting tensor retains the original shape (5,4), clearly demonstrating that cumsum does not remove any dimensions.

3.3 Applications

Non-reduction sums with keepdims=True and cumulative operations like cumsum() are useful in shape-aware computations:

- Normalization: Keeping dimensions allows consistent broadcasting when scaling or normalizing tensor rows or columns.

- Time-series modeling: cumsum() is used to track accumulated values such as cumulative returns or moving sums across time steps.

- Batch-wise processing: Maintaining original shapes helps apply per-sample operations while preserving compatibility for batch computations.

4. Dot Products

4.1 Definition

The dot product, also referred to as the inner product in Euclidean space $\mathbb{R}^n$, is a fundamental operation that returns a scalar from two vectors of equal length. Given two vectors

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

Dot product (or inner product, $\langle \mathbf{x}, \mathbf{y} \rangle$) is defined as**:**

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

The result of dot product is a scalar value.

4.2 Computation in practice

In practice, the dot product can be computed in multiple equivalent ways. For example, using PyTorch:

```
import torch

x = torch.tensor([0., 1., 2.])

y = torch.ones(3, dtype=torch.float32)

result = torch.dot(x, y)
```

Here, the dot product calculates:

$$0 \cdot 1 + 1 \cdot 1 + 2 \cdot 1 = 3$$

Alternatively, we can obtain this result by:

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

4.3 Applications

Dot products are applied in a wide range of contexts:

- Weighted averages: Given a vector $\mathbf{x} \in \mathbb{R}^n$ and a weight vector $\mathbf{w} \in \mathbb{R}^n$ such that $\sum w_i = 1$ and $w_i \geq 1$, the dot product $\mathbf{x}^{\mathrm{T}}\mathbf{w}$ represents a weighted average of $\mathbf{x}$'s components.

- Cosine similarity: After normalizing vectors to unit length, the dot product $\mathbf{x}^{\mathrm{T}}\mathbf{y}$ becomes the cosine of the angle between $\mathbf{x}$ and $\mathbf{y}$, which is a common similarity measure in information retrieval and recommendation systems.

- Neural networks: In fully connected layers, dot products are used to compute linear combinations of inputs and weights, forming the basis for prediction and representation learning.

5. Matrix - Vector Products

5.1 Definition

Let $\mathbf{A} \in \mathbb{R}^{m \times n}$ be a matrix and $\mathbf{x} \in \mathbb{R}^n$ be a vector. We visualize our matrix in terms of its row vectors

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

Where each $\mathbf{a}_i^T \in \mathbb{R}^n$ is a row vector representing the ith row of the matrix $\mathbf{A}$.

The matrix - vector product $\mathbf{Ax}$ is simply a column vector of length m, whose ith element is the dot product $\mathbf{a}_i^T\mathbf{x}$:

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

Alternatively, we can think $\mathbf{Ax}$ is a transformation of vectors from $\mathbb{R}^n$ to $\mathbb{R}^m$.

5.2 Computation in Practice

In PyTorch, the matrix–vector product can be computed using torch.mv():

```
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)
```

Here, the matrix **A** is:

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix} \text{ and } x = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

The result is:

$$y_1 = 0 \cdot 1 + 1 \cdot 2 + 2 \cdot 3 = 8$$

$$y_2 = 3 \cdot 1 + 4 \cdot 2 + 5 \cdot 3 = 26$$

5.3 Applications

Matrix–vector products are widely used in scientific computing and machine learning:

- Linear systems: Solving $\mathbf{Ax} = \mathbf{b}$ 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.

6. Matrix - Matrix Multiplication

6.1 Definition

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

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1k} \\ a_{21} & a_{22} & \dots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nk} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1k} \\ b_{21} & b_{22} & \dots & b_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{nk} \end{bmatrix}$$

Let $\mathbf{a}_i^T \in \mathbb{R}^k$ denote the row vector representing the i[th] row of the matrix **A** and let $\mathbf{b}_j \in \mathbb{R}^k$ denote the column vector from the j[th] column of the matrix **B**:

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

To form the matrix product $\mathbf{C} \in \mathbb{R}^{n \times m}$, we simply compute each element c_{ij} as the dot product between the i[th] row of **A** and the j[th] column of **B**, i.e., $\mathbf{a}_i^T \mathbf{b}_j$:

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

We can think of the matrix - matrix multiplication **AB** as performing m matrix - vector products or m × n dot products and stitching the results together to form an m × n matrix.

6.2 Computation in Practice

Using PyTorch, matrix–matrix multiplication is performed by torch.mm():

```
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)
```

Here:

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix}, \mathbf{B} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

Then:

$$\mathbf{C} = \begin{bmatrix} 0 \cdot 1 + 1 \cdot 3 + 2 \cdot 5 & 0 \cdot 2 + 1 \cdot 4 + 2 \cdot 6 \\ 3 \cdot 1 + 4 \cdot 3 + 5 \cdot 5 & 3 \cdot 2 + 4 \cdot 4 + 5 \cdot 6 \end{bmatrix} = \begin{bmatrix} 13 & 16 \\ 40 & 52 \end{bmatrix}$$

6.3 Applications

Matrix–matrix multiplication is foundational in:

- Batch operations in machine learning: Instead of computing output for one input at a time (matrix–vector), matrix–matrix multiplication enables parallel computation for entire mini-batches.

- Composing linear transformations: Combining multiple transformations (e.g., rotation + scaling) can be expressed as matrix–matrix products.

- Optimization and modeling: Many algorithms (e.g., PCA, SVD) rely on large-scale matrix operations.

7. Norms

7.1. Definition

A norm is a function $\|\cdot\|$ that maps a vector (or matrix) to a scalar, representing its size or length, and satisfies three properties:

- Scalability: $\|a\mathbf{x}\| = |a|\|\mathbf{x}\|$ for any scalar $a \in \mathbb{R}$.

- Triangle inequality: $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$.

- Non-negativity: $\|\mathbf{x}\| \geq 0$, and $\|\mathbf{x}\| = 0$ if and only if $\mathbf{x} = 0$.

The most common norms include:

- l_2 norm (Euclidean norm): $\|\mathbf{x}\|_2 = \sqrt{\sum x_i^2}$

- l_1 norm (Manhattan norm): $\|\mathbf{x}\|_1 = \sum|x_i|$

- l_p norm (general case): $\|\mathbf{x}\|_p = (\sum|x_i|^p)^{1/p}$

In matrices, a common norm is the Frobenius norm, defined as: $\|\mathbf{X}\|_F = \sqrt{\sum_{i,j} x_{i,j}^2}$, which behaves like an l_2 norm applied to matrix elements.

7.2. Computation in Practice

- l_2 norm of a vector:

```
u = torch.tensor([3.0, -4.0])
torch.norm(u)
>> tensor(5.)
```

- l_1 norm (sum of absolute values):

```
torch.abs(u).sum()
>> tensor(7.)
```

- Frobenius norm of a matrix (e.g., a 4x3 matrix of ones):

```
torch.norm(torch.ones((4, 3)))
>> tensor(3.4641)
```

These norms are readily available through deep learning frameworks like PyTorch, JAX, TensorFlow, etc.

7.3. Applications

Norms are mostly used in optimization and machine learning:

- In loss functions: norms quantify distances between predicted and true values (e.g., minimizing l_1 or l_2 loss).

- In regularization: norms (like l_2) are used to penalize large weights, preventing overfitting.

- In representation learning: norms measure distances in embedding spaces to ensure semantic closeness or separation.

- In matrix analysis: norms like the Frobenius norm quantify matrix sizes and are used in low-rank approximations and PCA.

These operations support objectives such as minimizing error, maximizing likelihood, or enforcing structure in learned models.

IV. Exercises and Solutions

Exercise 1: Prove that the transpose of the transpose of a matrix is the matrix itself: $(\mathbf{A}^T)^T = \mathbf{A}$

Solution:

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 $(\mathbf{A}^T)^T = \mathbf{A}$

Exercise 2: Given two matrices **A** and **B**, show that sum and transposition commute: $\mathbf{A}^\top + \mathbf{B}^\top = (\mathbf{A} + \mathbf{B})^\top$

Solution:

Given any two matrices **A** and **B** of the same size $\boldsymbol{m \times n}$

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

$(\mathbf{A} + \mathbf{B})^\top$:

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

The transpose of the sum of two matrices $(\mathbf{A} + \mathbf{B})^\top$ is the conversion of rows into columns and columns into rows, so we have:

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

$\mathbf{A}^\top + \mathbf{B}^\top$:

The transpose matrix A, B:

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

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

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

Exercise 3: Given any square matrix $\mathbf{A}$, is $\mathbf{A} + \mathbf{A}^T$ always symmetric? Can you prove the result by using only the results of the previous two exercises?

Solution:

$(\mathbf{A} + \mathbf{A}^T)^T = \mathbf{A}^T + (\mathbf{A}^T)^T = \mathbf{A}^T + \boldsymbol{A}$

→ $\mathbf{A} + \mathbf{A}^T$ always symmetric.

Exercise 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.

Solution:

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.

Check the result using Python:

```
import numpy as np


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

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

>> len(X) = 2
```

Exercise 5: 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?

Solution:

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).

- Tensor: A tensor has shape $(d_1, d_2, \dots, d_N)$, 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 $(d_1, d_2, \dots, d_N)$. For a scalar, $len(X)$ is undefined, as it has no axes.

Exercise 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       ]]
```

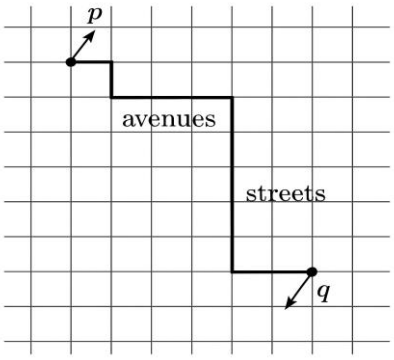
Analyze the result:

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

Exercise 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?



Solution:

Given two points in an n-dimensional space with real coordinates, $p = (p_1, \ldots, p_n)$, $q = (q_1, \ldots, q_n)$ 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:

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

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

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:

We are given a tensor of shape (2, 3, 4). This means:

- Axis 0: 2 elements (e.g. or 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 shape of the remaining axes: (3, 4)
- Result shape: (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)

Check the result using Python:

```
import numpy as np


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)):

[[[6 6 5 7]
```

```
 [4 8 9 1]
 [1 2 8 3]]

[[8 5 7 9]
 [2 4 0 2]
 [6 8 2 6]]]

Summation along axis 0 (shape (3, 4)):
[[14 11 12 16]
 [ 6 12  9  3]
 [ 7 10 10  9]]

Summation along axis 1 (shape (2, 4)):
[[11 16 22 11]
 [16 17  9 17]]

Summation along axis 2 (shape (2, 3)):
[[24 22 14]
 [29  8 22]]
```

Exercise 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?

Solution:

Assume this 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:

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

Check the result using Python:

```
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

Exercise 10: 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 $\mathbf{C} \in \mathbb{R}^{2^{5} \times 2^{14}}$, 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?

Solution:

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 $= 2^{31} + 2^{29} = 5.2^{29}$

- 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 $= 2^{35} + 2^{40} = 33.2^{35}$

Comparing operation counts:

$\mathbf{A}(\mathbf{BC})$ requires about $\frac{33.2^{35}}{5.2^{29}} = 422.4$ times more operations, making $(\mathbf{AB})\mathbf{C}$ much faster.

Memory Footprint

Memory usage depends on the size of intermediate matrices (assuming the final result is stored separately):

- 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 $(\mathbf{AB})\mathbf{C}$.

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.

Exercise 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?

Solution:

When computing $\mathbf{AB}$:

- 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 $\mathbf{AB}$

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

→ There is no difference between $\mathbf{AC}^T$ and $\mathbf{AB}$ 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

- $\mathbf{C}$ and $\mathbf{B}$ share the same data buffer, just with different strides.

Exercise 12: Consider three matrices, say $\mathbf{A},\mathbf{B},\mathbf{C} \in \mathbb{R}^{100x200}$. Construct a tensor with three axes by stacking [$\mathbf{A},\mathbf{B},\mathbf{C}$]. What is the dimensionality? Slice out the second coordinate of the third axis to recover $\mathbf{B}$. Check that your answer is correct.

Solution:

Create matrices $\mathbf{A},\mathbf{B},\mathbf{C} \in \mathbb{R}^{\mathbf{100x200}}$:

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

Each matrix has a shape of 100x200: 100 rows, 200 columns.

```
print(A.shape)
print(B.shape)
print(C.shape)
>> torch.Size([100, 200])
torch.Size([100, 200])
torch.Size([100, 200])
```

Stack the matrices:

axis =0

```
tensor = torch.stack([A, B, C])
print(tensor.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, and tensor[2] = C

Axis 1: row (100), Axis 2: column (200)

axis =2

```
tensor = torch.stack([A, B, C], axis=2)
print(tensor.shape)
>> torch.Size([3, 100, 200])
```

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 $\mathbb{R}^{3x100x200}$ or $\mathbb{R}^{100x200x3}$

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

```
print(tensor[:, :, 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]])
```

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

V. Conclusion

In this report, we have reviewed the foundational concepts of linear algebra and highlighted their central role in deep learning. From scalars and vectors to complex tensor operations, we examined how linear algebra enables the representation and manipulation of data in a mathematically coherent and computationally efficient manner.

We discussed core operations such as reduction (e.g., sum, mean), dot products, matrix - vector and matrix - matrix multiplication, all of which are extensively used in modern neural network computations. Understanding these operations not only supports the design of learning models, but also helps make their behavior easier to understand, optimize, and implement in practice.

As deep learning continues to evolve, linear algebra remains a fundamental skill - bridging the gap between theoretical insights and practical model development.

VI. Reference

1. Zhang, A., Lipton, Z. C., Li, M., & Smola, A. J. (2020). *Dive into Deep Learning*. Retrieved from https://d2l.ai/chapter_preliminaries/linear-algebra.html

2. GeeksforGeeks. (2023). *Applications of Linear Algebra*. Retrieved from https://www.geeksforgeeks.org/applications-of-linear-algebra/