Computational Linear Algebra

Siju Swamy

2024-07-16

Table of contents

Preface

Welcome to the course on **Computational Linear Algebra**. This course is designed to provide a practical perspective on linear algebra, bridging the gap between mathematical theory and real-world applications. As we delve into the intricacies of linear algebra, our focus will be on equipping you with the skills to effectively utilize these concepts in the design, development, and manipulation of data-driven processes applicable to Computer Science and Engineering.

Throughout this course, you will explore linear algebra not just as a set of abstract mathematical principles, but as a powerful tool for solving complex problems and optimizing processes. The curriculum integrates robust mathematical theory with hands-on implementation, enabling you to apply linear algebra techniques in practical scenarios.

From understanding fundamental operations to applying advanced concepts in data-driven contexts, this course aims to build a strong foundation that supports both theoretical knowledge and practical expertise. Whether you're tackling computational challenges or developing innovative solutions, the skills and insights gained here will be invaluable in your academic and professional endeavors Knuth (1984).

We look forward to guiding you through this journey of blending theory with practice and helping you harness the full potential of linear algebra in your work.

Introduction

Introduction to Computational Linear Algebra

Welcome to the Computational Linear Algebra course, a pivotal component of our Computational Mathematics for Engineering minor program. This course is meticulously designed to connect theoretical linear algebra concepts with their practical applications in Artificial Intelligence (AI) and Data Science.

Course Themes

1. **Practical Application Proficiency**
    - Our primary focus is on seamlessly translating theoretical concepts into practical solutions for real-world challenges.
    - Develop robust problem-solving skills applicable to AI, Data Science, and advanced engineering scenarios.
2. **Mathematical Expertise for Data Insights**
    - Gain in-depth proficiency in computational linear algebra, covering essential topics like matrix operations, eigendecomposition, and singular value decomposition.
    - Leverage linear algebra techniques to derive meaningful insights and make informed decisions in data science applications.
3. **Hands-On Learning**
    - Engage in immersive, project-based learning experiences with a strong emphasis on Python implementation.
    - Apply linear algebra principles to practical problems, including linear regression, principal component analysis (PCA), and neural networks.

Relevance and Impact

In today's technology-driven landscape, linear algebra forms the backbone of many critical algorithms and applications in AI and Data Science. This course will not only enhance your analytical and computational skills but also prepare you to address complex engineering problems with confidence.

By the end of this course, you will have acquired a comprehensive understanding of the role of linear algebra in computational mathematics and its practical applications. This knowledge will equip you with the tools necessary to excel in the rapidly evolving tech industry.

Let us start this educational journey together, where theoretical knowledge meets practical application, and explore the fascinating and impactful world of Computational Linear Algebra.

1 Python for Linear Algebra

1.1 Pseudocode: the new language for algorithm design

Pseudocode is a way to describe algorithms in a structured but plain language. It helps in planning the logic without worrying about the syntax of a specific programming language. In this module, we'll use Python-flavored pseudocode to describe various matrix operations.

> Caution
>
> There are varities of approaches in writing pseudocode. Students can adopt any of the standard approach to write pseudocode.

1.1.1 Matrix Sum

Mathematical Procedure:

To add two matrices A and B, both matrices must have the same dimensions. The sum C of two matrices A and B is calculated element-wise:

$$C[i][j] = A[i][j] + B[i][j]$$

Example:

Let A and B be two 2×2 matrices:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

The sum C is:

$$C = A + B = \begin{bmatrix} 1+5 & 2+6 \\ 3+7 & 4+8 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$$

Pseudocode:

```
FUNCTION matrix_sum(A, B):
    Get the number of rows and columns in matrix A
    Create an empty matrix C with the same dimensions
    FOR each row i:
        FOR each column j:
            Set C[i][j] to the sum of A[i][j] and B[i][j]
    RETURN the matrix C
END FUNCTION
```

Explanation:

1. Determine the number of rows and columns in matrix A.
2. Create a new matrix C with the same dimensions.
3. Loop through each element of the matrices and add corresponding elements.
4. Return the resulting matrix C.

1.1.2 Matrix Difference

Mathematical Procedure:

To subtract matrix B from matrix A, both matrices must have the same dimensions. The difference C of two matrices A and B is calculated element-wise:

$$C[i][j] = A[i][j] - B[i][j]$$

Example:

Let A and B be two 2×2 matrices:

$$A = \begin{bmatrix} 9 & 8 \\ 7 & 6 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

The difference C is:

$$C = A - B = \begin{bmatrix} 9-1 & 8-2 \\ 7-3 & 6-4 \end{bmatrix} = \begin{bmatrix} 8 & 6 \\ 4 & 2 \end{bmatrix}$$

Pseudocode:

```
FUNCTION matrix_difference(A, B):
    # Determine the number of rows and columns in matrix A
    rows = number_of_rows(A)
    cols = number_of_columns(A)

    # Create an empty matrix C with the same dimensions as A and B
    C = create_matrix(rows, cols)

    # Iterate through each row
    FOR i FROM 0 TO rows-1:
        # Iterate through each column
        FOR j FROM 0 TO cols-1:
            # Calculate the difference for each element and store it in C
            C[i][j] = A[i][j] - B[i][j]

    # Return the result matrix C
    RETURN C
END FUNCTION
```

In more human readable format the above pseudocode can be written as:

```
FUNCTION matrix_difference(A, B):
    Get the number of rows and columns in matrix A
    Create an empty matrix C with the same dimensions
    FOR each row i:
        FOR each column j:
            Set C[i][j] to the difference of A[i][j] and B[i][j]
    RETURN the matrix C
END FUNCTION
```

Explanation:

1. Determine the number of rows and columns in matrix A.
2. Create a new matrix C with the same dimensions.
3. Loop through each element of the matrices and subtract corresponding elements.
4. Return the resulting matrix C.

1.1.3 Matrix Product

Mathematical Procedure:

To find the product of two matrices A and B, the number of columns in A must be equal to the number of rows in B. The element $C[i][j]$ in the product matrix C is computed as:

$$C[i][j] = \sum_k A[i][k] \cdot B[k][j]$$

Example:

Let A be a 2×3 matrix and B be a 3×2 matrix:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \quad B = \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix}$$

The product C is:

$$C = A \cdot B = \begin{bmatrix} 58 & 64 \\ 139 & 154 \end{bmatrix}$$

Pseudocode:

```
FUNCTION matrix_product(A, B):
    # Get the dimensions of A and B
    rows_A = number_of_rows(A)
    cols_A = number_of_columns(A)
    rows_B = number_of_rows(B)
    cols_B = number_of_columns(B)

    # Check if multiplication is possible
    IF cols_A != rows_B:
        RAISE Error("Incompatible matrix dimensions")

    # Initialize result matrix C
    C = create_matrix(rows_A, cols_B)

    # Calculate matrix product
    FOR each row i FROM 0 TO rows_A-1:
        FOR each column j FROM 0 TO cols_B-1:
            # Compute the sum for C[i][j]
            sum = 0
            FOR each k FROM 0 TO cols_A-1:
                sum = sum + A[i][k] * B[k][j]
```

```
            C[i][j] = sum

    RETURN C
END FUNCTION
```

A more human readable version of the `pseudocode` is shown below:

```
FUNCTION matrix_product(A, B):
    Get the number of rows and columns in matrix A
    Get the number of columns in matrix B
    Create an empty matrix C with dimensions rows_A x cols_B
    FOR each row i in A:
        FOR each column j in B:
            Initialize C[i][j] to 0
            FOR each element k in the common dimension:
                Add the product of A[i][k] and B[k][j] to C[i][j]
    RETURN the matrix C
END FUNCTION
```

Explanation:

1. Determine the number of rows and columns in matrices A and B.
2. Create a new matrix C with dimensions rows$(A) \times$ columns(B).
3. Loop through each element of the resulting matrix $C[i][j]$ and calculate the dot product of i the row of A to j th column of B for each element.
4. Return the resulting matrix C.

1.1.4 Determinant

Mathematical Procedure:

To find the determinant of a square matrix A, we can use the Laplace expansion, which involves breaking the matrix down into smaller submatrices. For a 2×2 matrix, the determinant is calculated as:

$$\det(A) = A[0][0] \cdot A[1][1] - A[0][1] \cdot A[1][0]$$

For larger matrices, the determinant is calculated recursively.

Example:

Let A be a 2×2 matrix:

$$A = \begin{bmatrix} 4 & 3 \\ 6 & 3 \end{bmatrix}$$

The determinant of A is:

$$\det(A) = (4 \cdot 3) - (3 \cdot 6) = 12 - 18 = -6$$

Pseudocode:

```
FUNCTION determinant(A):
    # Step 1: Get the size of the matrix
    n = number_of_rows(A)

    # Base case for a 2x2 matrix
    IF n == 2:
        RETURN A[0][0] * A[1][1] - A[0][1] * A[1][0]

    # Step 2: Initialize determinant to 0
    det = 0

    # Step 3: Loop through each column of the first row
    FOR each column j FROM 0 TO n-1:
        # Get the submatrix excluding the first row and current column
        submatrix = create_submatrix(A, 0, j)
        # Recursive call to determinant
        sub_det = determinant(submatrix)
        # Alternating sign and adding to the determinant
        det = det + ((-1) ^ j) * A[0][j] * sub_det

    RETURN det
END FUNCTION

FUNCTION create_sub_matrix(A, row, col):
    sub_matrix = create_matrix(number_of_rows(A)-1, number_of_columns(A)-1)
    sub_i = 0
    FOR i FROM 0 TO number_of_rows(A)-1:
        IF i == row:
            CONTINUE
        sub_j = 0
        FOR j FROM 0 TO number_of_columns(A)-1:
            IF j == col:
```

```
                CONTINUE
            sub_matrix[sub_i][sub_j] = A[i][j]
            sub_j = sub_j + 1
        sub_i = sub_i + 1
    RETURN sub_matrix
END FUNCTION
```

A human readable version of the same pseudocode is shown below:

```
FUNCTION determinant(A):
    IF the size of A is 2x2:
        RETURN the difference between the product of the diagonals
    END IF
    Initialize det to 0
    FOR each column c in the first row:
        Create a sub_matrix by removing the first row and column c
        Add to det: the product of (-1)^c, the element A[0][c], and the
 ↪  determinant of the sub_matrix
    RETURN det
END FUNCTION

FUNCTION create_sub_matrix(A, row, col):
    Create an empty sub_matrix with dimensions one less than A
    Set sub_i to 0
    FOR each row i in A:
        IF i is the row to be removed:
            CONTINUE to the next row
        Set sub_j to 0
        FOR each column j in A:
            IF j is the column to be removed:
                CONTINUE to the next column
            Copy the element A[i][j] to sub_matrix[sub_i][sub_j]
            Increment sub_j
        Increment sub_i
    RETURN sub_matrix
END FUNCTION
```

Explanation:

1. If the matrix is 2×2, calculate the determinant directly.
2. For larger matrices, use the Laplace expansion to recursively calculate the determinant.

3. Create submatrices by removing the current row and column.
4. Sum the determinants of the submatrices, adjusted for the sign and the current element.

1.1.5 Rank of a Matrix

Mathematical Procedure:

The rank of a matrix A is the maximum number of linearly independent rows or columns in A. This can be found using Gaussian elimination to transform the matrix into its row echelon form (REF) and then counting the number of non-zero rows.

Example:

Let A be a 3×3 matrix:

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

After performing Gaussian elimination, we obtain:

$$\text{REF}(A) = \begin{bmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & 0 \end{bmatrix}$$

The rank of A is the number of non-zero rows, which is 2.

Pseudocode:

```
FUNCTION matrix_rank(A):
    # Step 1: Get the dimensions of the matrix
    rows = number_of_rows(A)
    cols = number_of_columns(A)

    # Step 2: Transform the matrix to row echelon form
    row_echelon_form(A, rows, cols)

    # Step 3: Count non-zero rows
    rank = 0
    FOR each row i FROM 0 TO rows-1:
        non_zero = FALSE
        FOR each column j FROM 0 TO cols-1:
            IF A[i][j] != 0:
```

```
                non_zero = TRUE
                BREAK
        IF non_zero:
            rank = rank + 1

    RETURN rank
END FUNCTION

FUNCTION row_echelon_form(A, rows, cols):
    # Perform Gaussian elimination
    lead = 0
    FOR r FROM 0 TO rows-1:
        IF lead >= cols:
            RETURN
        i = r
        WHILE A[i][lead] == 0:
            i = i + 1
            IF i == rows:
                i = r
                lead = lead + 1
                IF lead == cols:
                    RETURN
        # Swap rows i and r
        swap_rows(A, i, r)
        # Make A[r][lead] = 1
        lv = A[r][lead]
        A[r] = [m / float(lv) for m in A[r]]
        # Make all rows below r have 0 in column lead
        FOR i FROM r + 1 TO rows-1:
            lv = A[i][lead]
            A[i] = [iv - lv * rv for rv, iv in zip(A[r], A[i])]
        lead = lead + 1
END FUNCTION

FUNCTION swap_rows(A, row1, row2):
    temp = A[row1]
    A[row1] = A[row2]
    A[row2] = temp
END FUNCTION
```

A more human readable version of the above pseudocode is shown below:

```
FUNCTION rank(A):
    Get the number of rows and columns in matrix A
    Initialize the rank to 0
    FOR each row i in A:
        IF the element in the current row and column is non-zero:
            Increment the rank
            FOR each row below the current row:
                Calculate the multiplier to zero out the element below the
↪   diagonal
                Subtract the appropriate multiple of the current row from
↪   each row below
        ELSE:
            Initialize a variable to track if a swap is needed
            FOR each row below the current row:
                IF a non-zero element is found in the current column:
                    Swap the current row with the row having the non-zero
↪   element
                    Set the swap variable to True
                    BREAK the loop
            IF no swap was made:
                Decrement the rank
    RETURN the rank
END FUNCTION
```

Explanation:

1. Initialize the rank to 0.
2. Loop through each row of the matrix.
3. If the diagonal element is non-zero, increment the rank and perform row operations to zero out the elements below the diagonal.
4. If the diagonal element is zero, try to swap with a lower row that has a non-zero element in the same column.
5. If no such row is found, decrement the rank.
6. Return the resulting rank of the matrix.

1.1.6 Practice Problems

Find the rank of the following matrices.

1. $\begin{pmatrix} 1 & 1 & 3 \\ 2 & 2 & 6 \\ 2 & 5 & 3 \end{pmatrix}$.

2. $\begin{pmatrix} 2 & 0 & 2 \\ 1 & 2 & 3 \\ 3 & 2 & 7 \end{pmatrix}$

1.1.7 Solving a System of Equations

Mathematical Procedure:

To solve a system of linear equations represented as $A\mathbf{x} = \mathbf{b}$, where$A$ is the coefficient matrix, $\mathbf{x}$ is the vector of variables, and$\mathbf{b}$ is the constant vector, we can use Gaussian elimination to transform the augmented matrix $[A|\mathbf{b}]$ into its row echelon form (REF) and then perform back substitution to find the solution vector $\mathbf{x}$ Strang (2022).

Example:

Consider the system of equations:

$$\begin{cases} x + 2y + 3z & = 9 \\ 4x + 5y + 6z & = 24 \\ 7x + 8y + 9z & = 39 \end{cases}$$

The augmented matrix is:

$$[A|\mathbf{b}] = \begin{bmatrix} 1 & 2 & 3 & | & 9 \\ 4 & 5 & 6 & | & 24 \\ 7 & 8 & 9 & | & 39 \end{bmatrix}$$

After performing Gaussian elimination on the augmented matrix, we get:

$$\text{REF}(A) = \begin{bmatrix} 1 & 2 & 3 & | & 9 \\ 0 & -3 & -6 & | & -12 \\ 0 & 0 & 0 & | & 0 \end{bmatrix}$$

Performing back substitution, we solve for z,y, and x:

$$\begin{cases} z = 1 \\ y = 0 \\ x = 3 \end{cases}$$

Therefore, the solution vector is $\mathbf{x} = \begin{bmatrix} 3 \\ 0 \\ 1 \end{bmatrix}$.

Pseudocode:

```
FUNCTION solve_system_of_equations(A, b):
    # Step 1: Get the dimensions of the matrix
    rows = number_of_rows(A)
    cols = number_of_columns(A)

    # Step 2: Create the augmented matrix
    augmented_matrix = create_augmented_matrix(A, b)

    # Step 3: Transform the augmented matrix to row echelon form
    row_echelon_form(augmented_matrix, rows, cols)

    # Step 4: Perform back substitution
    solution = back_substitution(augmented_matrix, rows, cols)

    RETURN solution
END FUNCTION

FUNCTION create_augmented_matrix(A, b):
    # Combine A and b into an augmented matrix
    augmented_matrix = []
    FOR i FROM 0 TO number_of_rows(A)-1:
        augmented_matrix.append(A[i] + [b[i]])
    RETURN augmented_matrix
END FUNCTION

FUNCTION row_echelon_form(augmented_matrix, rows, cols):
    # Perform Gaussian elimination
    lead = 0
    FOR r FROM 0 TO rows-1:
        IF lead >= cols:
            RETURN
        i = r
        WHILE augmented_matrix[i][lead] == 0:
            i = i + 1
            IF i == rows:
                i = r
                lead = lead + 1
                IF lead == cols:
                    RETURN
        # Swap rows i and r
```

```
        swap_rows(augmented_matrix, i, r)
        # Make augmented_matrix[r][lead] = 1
        lv = augmented_matrix[r][lead]
        augmented_matrix[r] = [m / float(lv) for m in augmented_matrix[r]]
        # Make all rows below r have 0 in column lead
        FOR i FROM r + 1 TO rows-1:
            lv = augmented_matrix[i][lead]
            augmented_matrix[i] = [iv - lv * rv for rv, iv in
 ↪  zip(augmented_matrix[r], augmented_matrix[i])]
        lead = lead + 1
END FUNCTION

FUNCTION back_substitution(augmented_matrix, rows, cols):
    # Initialize the solution vector
    solution = [0 for _ in range(rows)]
    # Perform back substitution
    FOR i FROM rows-1 DOWNTO 0:
        solution[i] = augmented_matrix[i][cols-1]
        FOR j FROM i+1 TO cols-2:
            solution[i] = solution[i] - augmented_matrix[i][j] * solution[j]
    RETURN solution
END FUNCTION

FUNCTION swap_rows(matrix, row1, row2):
    temp = matrix[row1]
    matrix[row1] = matrix[row2]
    matrix[row2] = temp
END FUNCTION
```

Explanation:

1. Augment the coefficient matrix A with the constant matrix B.
2. Perform Gaussian elimination to reduce the augmented matrix to row echelon form.
3. Back-substitute to find the solution vector X.
4. Return the solution vector X.

1.1.8 Review Problems

Q1: Fill in the missing parts of the pseudocode to yield a meaningful algebraic operation on of two matrices.

Pseudocode:

```
FUNCTION matrix_op1(A, B):
    rows = number_of_rows(A)
    cols = number_of_columns(A)
    result = create_matrix(rows, cols, 0)

    FOR i FROM 0 TO rows-1:
        FOR j FROM 0 TO cols-1:
            result[i][j] = A[i][j] + ---

    RETURN result
END FUNCTION
```

Q2: Write the pseudocode to get useful derivable from a given a matrix by fill in the missing part.

Pseudocode:

```
FUNCTION matrix_op2(A):
    rows = number_of_rows(A)
    cols = number_of_columns(A)
    result = create_matrix(cols, rows, 0)

    FOR i FROM 0 TO rows-1:
        FOR j FROM 0 TO cols-1:
            result[j][i] = A[i][--]

    RETURN result
END FUNCTION
```

1.2 Transition from Pseudocode to Python Programming

In this course, our initial approach to understanding and solving linear algebra problems has been through pseudocode. Pseudocode allows us to focus on the logical steps and algorithms without getting bogged down by the syntax of a specific programming language. This method helps us build a strong foundation in the computational aspects of linear algebra.

However, to fully leverage the power of computational tools and prepare for real-world applications, it is essential to implement these algorithms in a practical programming language. Python is a highly versatile and widely-used language in the fields of data science, artificial intelligence, and engineering. By transitioning from pseudocode to Python, we align with the following course objectives:

1. **Practical Implementation:** Python provides numerous libraries and tools, such as NumPy and SciPy, which are specifically designed for numerical computations and linear algebra. Implementing our algorithms in Python allows us to perform complex calculations efficiently and accurately.

2. **Hands-On Experience:** Moving to Python programming gives students hands-on experience in coding, debugging, and optimizing algorithms. This practical experience is crucial for developing the skills required in modern computational tasks.

3. **Industry Relevance:** Python is extensively used in industry for data analysis, machine learning, and scientific research. Familiarity with Python and its libraries ensures that students are well-prepared for internships, research projects, and future careers in these fields.

4. **Integration with Other Tools:** Python's compatibility with various tools and platforms allows for seamless integration into larger projects and workflows. This integration is vital for tackling real-world problems that often require multi-disciplinary approaches.

5. **Enhanced Learning:** Implementing algorithms in Python helps reinforce theoretical concepts by providing immediate feedback through code execution and results visualization. This iterative learning process deepens understanding and retention of the material.

By transitioning to Python programming, we not only achieve our course objectives but also equip students with valuable skills that are directly applicable to their academic and professional pursuits.

1.3 Python Fundamentals

1.3.1 Python Programming Overview

Python is a high-level, interpreted programming language that was created by Guido van Rossum and first released in 1991. Its design philosophy emphasizes code readability and simplicity, making it an excellent choice for both beginners and experienced developers. Over the years, Python has undergone significant development and improvement, with major releases adding new features and optimizations. The language's versatility and ease of use have made it popular in various domains, including web development, data science, artificial intelligence, scientific computing, automation, and more. Python's extensive standard library and active community contribute to its widespread adoption, making it one of the most popular programming languages in the world today.

1.3.2 Variables

In Python, variables are used to store data that can be used and manipulated throughout a program. Variables do not need explicit declaration to reserve memory space. The declaration happens automatically when a value is assigned to a variable.

Basic Input/Output Functions

Python provides built-in functions for basic input and output operations. The `print()` function is used to display output, while the `input()` function is used to take input from the user.

Output with `print()` function

Example 1

```
# Printing text
print("Hello, World!")

# Printing multiple values
x = 5
y = 10
print("The value of x is:", x, "and the value of y is:", y)
```

Example 2

```
# Assigning values to variables
a = 10
b = 20.5
name = "Alice"

# Printing the values
print("Values Stored in the Variables:")
print(a)
print(b)
print(name)
```

Input with `input()` Function:

```
# Taking input from the user
name = input("Enter usr name: ")
print("Hello, " + name + "!")

# Taking numerical input
age = int(input("Enter usr age: "))
print("us are", age, "years old.")
```

> **Note**
>
> The `print()` function in Python, defined in the built-in `__builtin__` module, is used to display output on the screen, providing a simple way to output text and variable values to the console.

Combining Variables and Input/Output

us can combine variables and input/output functions to create interactive programs.

Example:

```
# Program to calculate the sum of two numbers
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))

# Calculate sum
sum = num1 + num2

# Display the result
print("The sum of", num1, "and", num2, "is", sum)
```

1.3.3 Python Programming Style

1.3.3.1 Indentation

Python uses indentation to define the blocks of code. Proper indentation is crucial as it affects the program's flow. Use 4 spaces per indentation level.

```
if a > b:
    print("a is greater than b")
else:
    print("b is greater than or equal to a")
```

1.3.3.2 Comments

Use comments to explain usr code. Comments begin with the # symbol and extend to the end of the line. Write comments that are clear and concise. See the example:

```
# This is a comment
a = 10  # This is an inline comment
```

1.3.3.3 Variable Naming

Use meaningful variable names to make usr code more understandable. Variable names should be in lowercase with words separated by underscores.

```
student_name = "John"
total_score = 95
```

1.3.3.4 Consistent Style

Follow the `PEP 8` style guide for Python code to maintain consistency and readability. Use blank lines to separate different sections of usr code. See the following example of function definition:

```

def calculate_sum(x, y):
    return x + y

result = calculate_sum(5, 3)
print(result)
```

1.4 Basic Datatypes in Python

In Python, a datatype is a classification that specifies which type of value a variable can hold. Understanding datatypes is essential as it helps in performing appropriate operations on variables. Python supports various built-in datatypes, which can be categorized into several groups.

1.4.1 Numeric Types

Numeric types represent data that consists of numbers. Python has three distinct numeric types:

1. **Integers (`int`)**:
   - Whole numbers, positive or negative, without decimals.
   - Example: `a = 10`, `b = -5`.
2. **Floating Point Numbers (`float`)**:
   - Numbers that contain a decimal point.
   - Example: `pi = 3.14`, `temperature = -7.5`.
3. **Complex Numbers (`complex`)**:
   - Numbers with a real and an imaginary part.
   - Example: `z = 3 + 4j`.

```
# Examples of numeric types
a = 10          # Integer
pi = 3.14       # Float
z = 3 + 4j      # Complex
```

1.4.2 Sequence Types

Sequence types are used to store multiple items in a single variable. Python has several sequence types, including:

1.4.2.1 String Type

Strings in Python are sequences of characters enclosed in quotes. They are used to handle and manipulate textual data.

Characteristics of Strings

- *Ordered*: Characters in a string have a defined order.
- *Immutable*: Strings cannot be modified after they are created.
- *Heterogeneous*: Strings can include any combination of letters, numbers, and symbols.

Creating Strings

Strings can be created using single quotes, double quotes, or triple quotes for multiline strings.

Example:

```
# Creating strings with different types of quotes
single_quoted = 'Hello, World!'
double_quoted = "Hello, World!"
multiline_string = """This is a
multiline string"""
```

Accessing String Characters

Characters in a string are accessed using their index, with the first character having an index of 0. Negative indexing can be used to access characters from the end.

Example:

```
# Accessing characters in a string
first_char = single_quoted[0]  # Output: 'H'
last_char = single_quoted[-1]  # Output: '!'
```

Common String Methods

Python provides various methods for string manipulation:

1. `upper()`: Converts all characters to uppercase.
2. `lower()`: Converts all characters to lowercase.
3. `strip()`: Removes leading and trailing whitespace.
4. `replace(old, new)`: Replaces occurrences of a substring with another substring.
5. `split(separator)`: Splits the string into a list based on a separator.

Example:

```
# Using string methods
text = "   hello, world!   "
uppercase_text = text.upper()       # Result: "   HELLO, WORLD!   "
stripped_text = text.strip()        # Result: "hello, world!"
replaced_text = text.replace("world", "Python")  # Result: "   hello, Python!
↪   "
words = text.split(",")             # Result: ['hello', ' world!   ']
```

1.4.2.2 List Type

Lists are one of the most versatile and commonly used sequence types in Python. They allow for the storage and manipulation of ordered collections of items.

Characteristics of Lists

- *Ordered*: The items in a list have a defined order, which will not change unless explicitly modified.
- *Mutable*: The content of a list can be changed after its creation (i.e., items can be added, removed, or modified).
- *Dynamic*: Lists can grow or shrink in size as items are added or removed.
- *Heterogeneous*: Items in a list can be of different data types (e.g., integers, strings, floats).

Creating Lists

Lists are created by placing comma-separated values inside square brackets.

Example:

```python
# Creating a list of fruits
fruits = ["apple", "banana", "cherry"]

# Creating a mixed list
mixed_list = [1, "Hello", 3.14]
```

Accessing List Items

List items are accessed using their index, with the first item having an index of 0.

Example:

```python

# Accessing the first item
first_fruit = fruits[0]  # Output: "apple"

# Accessing the last item
last_fruit = fruits[-1]  # Output: "cherry"
```

Modifying Lists

Lists can be modified by changing the value of specific items, adding new items, or removing existing items.

Example:

```
# Changing the value of an item
fruits[1] = "blueberry"  # fruits is now ["apple", "blueberry", "cherry"]

# Adding a new item
fruits.append("orange")  # fruits is now ["apple", "blueberry", "cherry",
 ↪  "orange"]

# Removing an item
fruits.remove("blueberry")  # fruits is now ["apple", "cherry", "orange"]
```

List Methods

`Python` provides several built-in methods to work with lists:

1. `append(item)`: Adds an item to the end of the list.
2. `insert(index, item)`: Inserts an item at a specified index.
3. `remove(item)`: Removes the first occurrence of an item.
4. `pop(index)`: Removes and returns the item at the specified index.
5. `sort()`: Sorts the list in ascending order.
6. `reverse()`: Reverses the order of the list.

Example:

```
# Using list methods
numbers = [5, 2, 9, 1]

numbers.append(4)     # numbers is now [5, 2, 9, 1, 4]
numbers.sort()        # numbers is now [1, 2, 4, 5, 9]
numbers.reverse()     # numbers is now [9, 5, 4, 2, 1]
first_number = numbers.pop(0)  # first_number is 9, numbers is now [5, 4, 2,
 ↪  1]
```

1.4.2.3 Tuple Type

Tuples are a built-in sequence type in Python that is used to store an ordered collection of items. Unlike lists, tuples are immutable, which means their contents cannot be changed after creation.

Characteristics of Tuples

- *Ordered*: Tuples maintain the order of items, which is consistent throughout their lifetime.

- *Immutable*: Once a tuple is created, its contents cannot be modified. This includes adding, removing, or changing items.
- *Fixed Size*: The size of a tuple is fixed; it cannot grow or shrink after creation.
- *Heterogeneous*: Tuples can contain items of different data types, such as integers, strings, and floats.

Creating Tuples

Tuples are created by placing comma-separated values inside parentheses. Single-element tuples require a trailing comma.

Example:

```
# Creating a tuple with multiple items
coordinates = (10, 20, 30)

# Creating a single-element tuple
single_element_tuple = (5,)

# Creating a tuple with mixed data types
mixed_tuple = (1, "Hello", 3.14)
```

Accessing Tuple Items

Tuple items are accessed using their index, with the first item having an index of 0. Negative indexing can be used to access items from the end.

Example:

```
# Accessing the first item
x = coordinates[0]  # Output: 10

# Accessing the last item
z = coordinates[-1]  # Output: 30
```

Modifying Tuples

Since tuples are immutable, their contents cannot be modified. However, us can create new tuples by combining or slicing existing ones.

Example:

```python
# Combining tuples
new_coordinates = coordinates + (40, 50)  # Result: (10, 20, 30, 40, 50)

# Slicing tuples
sub_tuple = coordinates[1:3]  # Result: (20, 30)
```

Tuple Methods

Tuples have a limited set of built-in methods compared to lists:

1. `count(item)`: Returns the number of occurrences of the specified item.
2. `index(item)`: Returns the index of the first occurrence of the specified item.

   Example:

```python
# Using tuple methods
numbers = (1, 2, 3, 1, 2, 1)

# Counting occurrences of an item
count_1 = numbers.count(1)  # Result: 3

# Finding the index of an item
index_2 = numbers.index(2)  # Result: 1
```

1.4.3 Mapping Types

Mapping types in Python are used to store data in key-value pairs. Unlike sequences, mappings do not maintain an order and are designed for quick lookups of data.

1.4.3.1 Dictionary (`dict`)

The primary mapping type in Python is the `dict`. Dictionaries store data as key-value pairs, where each key must be unique, and keys are used to access their corresponding values.

Characteristics of Dictionaries

- *Unordered*: The order of items is not guaranteed and may vary.
- *Mutable*: us can add, remove, and change items after creation.
- *Keys*: Must be unique and immutable (e.g., strings, numbers, tuples).
- *Values*: Can be of any data type and can be duplicated.

Creating Dictionaries

Dictionaries are created using curly braces `{}` with key-value pairs separated by colons `:`.

Example:

```
# Creating a dictionary
student = {
    "name": "Alice",
    "age": 21,
    "major": "Computer Science"
}
```

Accessing and Modifying Dictionary Items

Items in a dictionary are accessed using their keys. us can also modify, add, or remove items.

Example:

```
# Accessing a value
name = student["name"]  # Output: "Alice"

# Modifying a value
student["age"] = 22  # Updates the age to 22

# Adding a new key-value pair
student["graduation_year"] = 2024

# Removing a key-value pair
del student["major"]
```

Dictionary Methods

Python provides several built-in methods to work with dictionaries:

1. `keys()`: Returns a view object of all keys.
2. `values()`: Returns a view object of all values.
3. `items()`: Returns a view object of all key-value pairs.
4. `get(key, default)`: Returns the value for the specified key, or a default value if the key is not found.
5. `pop(key, default)`: Removes and returns the value for the specified key, or a default value if the key is not found.

Example:

```
# Using dictionary methods
keys = student.keys()        # Result: dict_keys(['name', 'age',
 ↪  'graduation_year'])
values = student.values()    # Result: dict_values(['Alice', 22, 2024])
items = student.items()      # Result: dict_items([('name', 'Alice'), ('age',
 ↪  22), ('graduation_year', 2024)])
name = student.get("name")  # Result: "Alice"
age = student.pop("age")    # Result: 22
```

1.4.4 Set Types

Sets are a built-in data type in Python used to store unique, unordered collections of items. They are particularly useful for operations involving membership tests, set operations, and removing duplicates.

Characteristics of Sets

- *Unordered* : The items in a set do not have a specific order and may change.
- *Mutable* : us can add or remove items from a set after its creation.
- *Unique* : Sets do not allow duplicate items; all items must be unique.
- *Unindexed* : Sets do not support indexing or slicing.

Creating Sets

Sets are created using curly braces `{}` with comma-separated values, or using the `set()` function.

Example:

```
# Creating a set using curly braces
fruits = {"apple", "banana", "cherry"}

# Creating a set using the set() function
numbers = set([1, 2, 3, 4, 5])
```

Accessing and Modifying Set Items

While us cannot access individual items by index, us can check for membership and perform operations like adding or removing items.

Example:

```python
# Checking membership
has_apple = "apple" in fruits  # Output: True

# Adding an item
fruits.add("orange")

# Removing an item
fruits.remove("banana")  # Raises KeyError if item is not present
```

Set Operations Sets support various mathematical set operations, such as `union`, `intersection`, and `difference`.

Example:

```python
# Union of two sets
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union = set1 | set2  # Result: {1, 2, 3, 4, 5}

# Intersection of two sets
intersection = set1 & set2  # Result: {3}

# Difference between two sets
difference = set1 - set2  # Result: {1, 2}

# Symmetric difference (items in either set, but not in both)
symmetric_difference = set1 ^ set2  # Result: {1, 2, 4, 5}
```

Set Methods

Python provides several built-in methods for set operations:

1. `add(item)`: Adds an item to the set.
2. `remove(item)`: Removes an item from the set; raises KeyError if item is not present.
3. `discard(item)`: Removes an item from the set if present; does not raise an error if item is not found.
4. `pop()`: Removes and returns an arbitrary item from the set.
5. `clear()`: Removes all items from the set.

Example:

```
# Using set methods
set1 = {1, 2, 3}

set1.add(4)        # set1 is now {1, 2, 3, 4}
set1.remove(2)     # set1 is now {1, 3, 4}
set1.discard(5)    # No error, set1 remains {1, 3, 4}
item = set1.pop()  # Removes and returns an arbitrary item, e.g., 1
set1.clear()      # set1 is now an empty set {}
```

1.4.4.1 ## Frozen Sets

Frozen sets are a built-in data type in Python that are similar to sets but are immutable. Once created, a frozen set cannot be modified, making it suitable for use as a key in dictionaries or as elements of other sets.

Characteristics of Frozen Sets

- *Unordered* : The items in a frozen set do not have a specific order and may change.
- *Immutable* : Unlike regular sets, frozen sets cannot be altered after creation. No items can be added or removed.
- *Unique* : Like sets, frozen sets do not allow duplicate items; all items must be unique.
- *Unindexed* : Frozen sets do not support indexing or slicing.

Creating Frozen Sets

Frozen sets are created using the `frozenset()` function, which takes an iterable as an argument.

Example:

```
# Creating a frozen set
numbers = frozenset([1, 2, 3, 4, 5])

# Creating a frozen set from a set
fruits = frozenset({"apple", "banana", "cherry"})
```

Accessing and Modifying Frozen Set Items

Frozen sets do not support modification operations such as adding or removing items. However, us can perform membership tests and other set operations.

Example:

```python
# Checking membership
has_apple = "apple" in fruits  # Output: True

# Since frozenset is immutable, us cannot use add() or remove() methods
```

Set Operations with Frozen Sets

Frozen sets support various mathematical set operations similar to regular sets, such as union, intersection, and difference. These operations return new frozen sets and do not modify the original ones.

Example:

```python
# Union of two frozen sets
set1 = frozenset([1, 2, 3])
set2 = frozenset([3, 4, 5])
union = set1 | set2  # Result: frozenset({1, 2, 3, 4, 5})

# Intersection of two frozen sets
intersection = set1 & set2  # Result: frozenset({3})

# Difference between two frozen sets
difference = set1 - set2  # Result: frozenset({1, 2})

# Symmetric difference (items in either set, but not in both)
symmetric_difference = set1 ^ set2  # Result: frozenset({1, 2, 4, 5})
```

Frozen Set Methods

Frozen sets have a subset of the methods available to regular sets. The available methods include:

1. `copy()` : Returns a shallow copy of the frozen set.
2. `difference(other)` : Returns a new frozen set with elements in the original frozen set but not in other.
3. `intersection(other)` : Returns a new frozen set with elements common to both frozen sets.
4. `union(other)` : Returns a new frozen set with elements from both frozen sets.
5. `symmetric_difference(other)` : Returns a new frozen set with elements in either frozen set but not in both.

Example:

```python
# Using frozen set methods
set1 = frozenset([1, 2, 3])
set2 = frozenset([3, 4, 5])

# Getting the difference
difference = set1.difference(set2)  # Result: frozenset({1, 2})

# Getting the intersection
intersection = set1.intersection(set2)  # Result: frozenset({3})

# Getting the union
union = set1.union(set2)  # Result: frozenset({1, 2, 3, 4, 5})

# Getting the symmetric difference
symmetric_difference = set1.symmetric_difference(set2)  # Result:
↪ frozenset({1, 2, 4, 5})
```

1.5 Control Structures in Python

Control structures in Python allow us to control the flow of execution in our programs. They help manage decision-making, looping, and the execution of code blocks based on certain conditions. Python provides several key control structures: `if` statements, `for` loops, `while` loops, and control flow statements like `break`, `continue`, and `pass`.

1.5.1 Conditional Statements

Conditional statements are used to execute code based on certain conditions. The primary conditional statement in Python is the `if` statement, which can be combined with `elif` and `else` to handle multiple conditions.

Syntax:

```python
if condition:
    # Code block to execute if condition is True
elif another_condition:
    # Code block to execute if another_condition is True
else:
    # Code block to execute if none of the above conditions are True
```

Example: Program to classify a person based on his/her age.

```python
age = 20

if age < 18:
    print("us are a minor.")
elif age < 65:
    print("us are an adult.")
else:
    print("us are a senior citizen.")
```

1.5.2 Looping Statements

Looping statements are used to repeat a block of code multiple times. Python supports for loops and while loops.

1.5.2.1 For Loop

The `for` loop iterates over a sequence (like a list, tuple, or string) and executes a block of code for each item in the sequence.

Syntax:

```python
for item in sequence:
    # Code block to execute for each item
```

Example: Program to print names of fruits saved in a list.

```python
# Iterating over a list
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

1.5.2.2 While Loop

The `while` loop repeatedly executes a block of code as long as a specified condition is True.

Syntax:

```python
while condition:
    # Code block to execute while condition is True
```

Example: Print all counting numbers less than 5.

```
# Counting from 0 to 4
count = 0
while count < 5:
    print(count)
    count += 1
```

1.5.3 Control Flow Statements

Control flow statements alter the flow of execution within loops and conditionals.

1.5.3.1 Break Statement

The `break` statement exits the current loop, regardless of the loop's condition.

Example: Program to exit from the printing of whole numbers less than 10, while trigger 5.

```
for i in range(10):
    if i == 5:
        break
    print(i)
# Output: 0 1 2 3 4
```

1.5.3.2 Continue Statement

The `continue` statement skips the rest of the code inside the current loop iteration and proceeds to the next iteration.

Example: Program to print all the whole numbers in the range 5 except 2.

```
for i in range(5):
    if i == 2:
        continue
    print(i)
# Output: 0 1 3 4
```

1.5.3.3 Pass Statement

The `pass` statement is a placeholder that does nothing and is used when a statement is syntactically required but no action is needed.

Example: Program to print all the whole numbers in the range 5 except 3.

```
for i in range(5):
    if i == 3:
        pass  # Placeholder for future code
    else:
        print(i)
# Output: 0 1 2 4
```

> **Cautions When Using Control Flow Structures**
>
> Control flow structures are essential in Python programming for directing the flow of execution. However, improper use of these structures can lead to errors, inefficiencies, and unintended behaviors. Here are some cautions to keep in mind:
>
> **Infinite Loops**
>
> - **Issue**: A `while` loop with a condition that never becomes `False` can lead to an infinite loop, which will cause the program to hang or become unresponsive.
> - **Caution**: Always ensure that the condition in a `while` loop will eventually become `False`, and include logic within the loop to modify the condition.
>
> **Example:**
>
> ```
> # Infinite loop example
> count = 0
> while count < 5:
>     print(count)
>     # Missing count increment, causing an infinite loop
> ```

1.6 Functions in Python Programming

Functions are a fundamental concept in Python programming that enable code reuse, modularity, and organization. They allow us to encapsulate a block of code that performs a specific task, which can be executed whenever needed. Functions are essential for writing clean, maintainable, and scalable code, making them a cornerstone of effective programming practices.

What is a Function?

A function is a named block of code designed to perform a specific task. Functions can take inputs, called parameters or arguments, and can return outputs, which are the results of the computation or task performed by the function. By defining functions, we can write code once and reuse it multiple times, which enhances both efficiency and readability.

Defining a Function

In Python, functions are defined using the `def` keyword, followed by the function name, parentheses containing any parameters, and a colon. The function body, which contains the code to be executed, is indented below the function definition.

Syntax:

```
def function_name(parameters):
    # Code block
    return result
```

Example:

```
def greet(name):
    """
    Returns a greeting message for the given name.
    """
    return f"Hello, {name}!"
```

1.6.0.1 Relevance of functions in Programming

1. *Code Reusability* : Functions allow us to define a piece of code once and reuse it in multiple places. This reduces redundancy and helps maintain consistency across our codebase.
2. *Modularity* : Functions break down complex problems into smaller, manageable pieces. Each function can be focused on a specific task, making it easier to understand and maintain the code.
3. *Abstraction* : Functions enable us to abstract away the implementation details. We can use a function without needing to know its internal workings, which simplifies the code we write and enhances readability.
4. *Testing and Debugging* : Functions allow us to test individual components of our code separately. This isolation helps in identifying and fixing bugs more efficiently.

5. *Library Creation* : Functions are the building blocks of libraries and modules. By organizing related functions into libraries, we can create reusable components that can be shared and utilized across different projects.

   Example: Creating a Simple Library

Stage 1: Define Functions in a Module

```python
# my_library.py

def add(a, b):
    """
    Returns the sum of two numbers.
    """
    return a + b

def multiply(a, b):
    """
    Returns the product of two numbers.
    """
    return a * b
```

Stage 2: Use the Library in Another Program

```python
# main.py

import my_library

result_sum = my_library.add(5, 3)
result_product = my_library.multiply(5, 3)

print(f"Sum: {result_sum}")
print(f"Product: {result_product}")
```

1.7 Object-Oriented Programming (OOP) in Python

Object-Oriented Programming (OOP) is a programming paradigm that uses "objects" to design and implement software. It emphasizes the organization of code into classes and objects, allowing for the encapsulation of data and functionality. OOP promotes code reusability, scalability, and maintainability through key principles such as encapsulation, inheritance, and polymorphism.

1.7.1 Key Concepts of OOP

1. **Classes and Objects**

- **Class**: A class is a blueprint for creating objects. It defines a set of attributes and methods that the created objects will have. A class can be thought of as a template or prototype for objects.
- **Object**: An object is an instance of a class. It is a specific realization of the class with actual values for its attributes.

1.7.1.1 Example

```python
# Defining a class
class Dog:
    def __init__(self, name, age):
        self.name = name  # Attribute
        self.age = age    # Attribute

    def bark(self):
        return "Woof!"   # Method

# Creating an object of the class
my_dog = Dog(name="Buddy", age=3)

# Accessing attributes and methods
print(my_dog.name)  # Output: Buddy
print(my_dog.age)   # Output: 3
print(my_dog.bark())  # Output: Woof!
```

2. **Encapsulation**

Encapsulation is the concept of bundling data (attributes) and methods (functions) that operate on the data into a single unit, or class. It restricts direct access to some of the object's components and can help protect the internal state of the object from unintended modifications.

Example: Controll the access to member variables using encapsulation.

```python
class Account:
    def __init__(self, balance):
        self.__balance = balance  # Private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def get_balance(self):
        return self.__balance

# Creating an object of the class
my_account = Account(balance=1000)
my_account.deposit(500)

print(my_account.get_balance())  # Output: 1500
# print(my_account.__balance)  # This will raise an AttributeError
```

3. **Inheritance**

Inheritance is a mechanism in which a new class (child or derived class) inherits attributes and methods from an existing class (parent or base class). It allows for code reuse and the creation of a hierarchy of classes.

Example: Demonstrating usage of attributes of base class in the derived classes.

```python
# Base class
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return "Some sound"

# Derived class
class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)  # Calling the constructor of the base class
        self.breed = breed

    def speak(self):
        return "Woof!"
```

```python
# Another derived class
class Cat(Animal):
    def __init__(self, name, color):
        super().__init__(name)  # Calling the constructor of the base class
        self.color = color

    def speak(self):
        return "Meow!"

# Creating objects of the derived classes
dog = Dog(name="Buddy", breed="Golden Retriever")
cat = Cat(name="Whiskers", color="Gray")

print(f"{dog.name} is a {dog.breed} and says {dog.speak()}")  # Output: Buddy
↪  is a Golden Retriever and says Woof!
print(f"{cat.name} is a {cat.color} cat and says {cat.speak()}")  # Output:
↪  Whiskers is a Gray cat and says Meow!
```

4. **Polymorphism**

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables a single interface to be used for different data types. In Python, polymorphism is often achieved through method overriding, where a method in a derived class has the same name as a method in the base class but implements different functionality.

Example:

```python
class Bird:
    def fly(self):
        return "Flies in the sky"

class Penguin(Bird):
    def fly(self):
        return "Cannot fly, swims instead"

# Creating objects of different classes
bird = Bird()
penguin = Penguin()

print(bird.fly())      # Output: Flies in the sky
print(penguin.fly())  # Output: Cannot fly, swims instead
```

1.8 Working with Files in Python

File handling is an essential part of programming that allows us to work with data stored in files. Python provides built-in functions and methods to create, read, write, and manage files efficiently. This section will cover basic file operations, including opening, reading, writing, and closing files.

Opening a File

In Python, we use the `open()` function to open a file. This function returns a file object, which provides methods and attributes to interact with the file. The `open()` function requires at least one argument: the path to the file. we can also specify the mode in which the file should be opened.

Syntax:

```
file_object = open(file_path, mode)
```

Where,

- `file_path` : Path to the file (can be a relative or absolute path).
- `mode` : Specifies the file access mode (e.g., 'r' for reading, 'w' for writing, 'a' for appending).

Example:

```
# Opening a file in read mode
file = open('example.txt', 'r')
```

Reading from a File

Once a file is opened, we can read its contents using various methods. Common methods include `read()`, `readline()`, and `readlines()`.

- `read()` : Reads the entire file content.
- `readline()` : Reads a single line from the file.
- `readlines()` : Reads all the lines into a list.

Example:

```python
# Reading the entire file
file_content = file.read()
print(file_content)

# Reading a single line
file.seek(0)  # Move cursor to the start of the file
line = file.readline()
print(line)

# Reading all lines
file.seek(0)
lines = file.readlines()
print(lines)
```

Writing to a File

To write data to a file, we need to open the file in write ('w') or append ('a') mode. When opened in `write` mode, the file is truncated (i.e., existing content is deleted). When opened in `append` mode, new data is added to the end of the file.

Example:

```python
# Opening a file in write mode
file = open('example.txt', 'w')

# Writing data to the file
file.write("Hello, World!\n")
file.write("Python file handling example.")

# Closing the file
file.close()
```

Closing a File

It is important to close a file after performing operations to ensure that all changes are saved and resources are released. We can close a file using the `close()` method of the file object.

Example:

```python
f_1 = open('example.txt', 'w') # open the file example.txt to f_1
f_1.close() # close the file with handler 'f_1'
```

Using Context Managers

Context managers provide a convenient way to handle file operations, automatically managing file opening and closing. We can use the `with` statement to ensure that a file is properly closed after its block of code is executed.

Example:

```
# Using context manager to open and write to a file
with open('example.txt', 'w') as file:
    file.write("This is written using a context manager.")
```

1.9 From Theory to Practice

In this section, we transition from theoretical concepts to practical applications by exploring how fundamental matrix operations can be used in the field of image processing. By leveraging the knowledge gained from understanding matrix addition, subtraction, multiplication, and other operations, we can tackle real-world problems such as image blending, sharpening, filtering, and transformations. This hands-on approach not only reinforces the theoretical principles but also demonstrates their utility in processing and enhancing digital images. Through practical examples and coding exercises, you'll see how these mathematical operations are essential tools in modern image manipulation and analysis.

1.9.1 Applications of Matrix Operations in Digital Image Processing

Matrix operations play a pivotal role in digital image processing, enabling a wide range of techniques for manipulating and enhancing images. By leveraging fundamental matrix operations such as addition, subtraction, multiplication, and transformations, we can perform essential tasks like image blending, filtering, edge detection, and geometric transformations. These operations provide the mathematical foundation for various algorithms used in image analysis, compression, and reconstruction. Understanding and applying matrix operations is crucial for developing efficient and effective image processing solutions, making it an indispensable skill in fields like computer vision, graphics, and multimedia applications.

1.9.1.1 Matrix Addition in Image Blending

Matrix addition is a fundamental operation in image processing, particularly useful in the technique of image blending. Image blending involves combining two images to produce a single image that integrates the features of both original images. This technique is commonly used in applications such as image overlay, transition effects in videos, and creating composite images.

Concept

When working with grayscale images, each image can be represented as a matrix where each element corresponds to the intensity of a pixel. By adding corresponding elements (pixels) of two matrices (images), we can blend the images together. The resultant matrix represents the blended image, where each pixel is the sum of the corresponding pixels in the original images.

Example:

Consider two 2x2 grayscale images represented as matrices:

```
image1= [[100, 150],[200, 250]]
image2=[[50, 100],[100, 150]]
```

To blend these images, we add the corresponding pixel values as:

```
blended_image[i][j] = image1[i][j] + image2[i][j]
```

Ensure that the resulting pixel values do not exceed the maximum value allowed (255 for 8-bit images).

Python Implementation of image blending

Below is the Python code for blending two images using matrix addition:

```
def matrix_addition(image1, image2):
    rows = len(image1)
    cols = len(image1[0])
    blended_image = [[0] * cols for _ in range(rows)]

    for i in range(rows):
        for j in range(cols):
            blended_pixel = image1[i][j] + image2[i][j]
            blended_image[i][j] = min(blended_pixel, 255)  # Clip to 255

    return blended_image

# Example matrices (images)
image1 = [[100, 150], [200, 250]]
```

```python
image2 = [[50, 100], [100, 150]]

blended_image = matrix_addition(image1, image2)
print("Blended Image:")
for row in blended_image:
    print(row)
```

```
Blended Image:
[150, 250]
[255, 255]
```

Image blending is a powerful technique with numerous real-time applications. It is widely used in creating smooth transitions in video editing, overlaying images in augmented reality, and producing composite images in photography and graphic design. By understanding and applying matrix operations, we can develop efficient algorithms that seamlessly integrate multiple images, enhancing the overall visual experience. The practical implementation of matrix addition in image blending underscores the importance of mathematical foundations in achieving sophisticated image processing tasks in real-world applications.

> Image Blending as Basic Arithmetic with Libraries
>
> In upcoming chapters, we will explore how specific libraries for image handling simplify the process of image blending to a basic arithmetic operation—adding two objects. Using these libraries, such as PIL (Python Imaging Library) or OpenCV, allows us to leverage efficient built-in functions that streamline tasks like resizing, matrix operations, and pixel manipulation.

Let's summarize a few more matrix operations and its uses in digital image processing tasks in the following sections.

1.9.1.2 Matrix Subtraction in Image Sharpening

Matrix subtraction is a fundamental operation in image processing, essential for techniques like image sharpening. Image sharpening enhances the clarity and detail of an image by increasing the contrast along edges and boundaries.

Concept

In grayscale images, each pixel value represents the intensity of light at that point. Image sharpening involves subtracting a smoothed version of the image from the original. This process accentuates edges and fine details, making them more prominent.

Example:

Consider a grayscale image represented as a matrix:

```
original_image [[100, 150, 200],[150, 200, 250],[200, 250, 100]]
```

To sharpen the image, we subtract a blurred version (smoothed image) from the original. This enhances edges and fine details:

```
sharpened_image[i][j] = original_image[i][j] - blurred_image[i][j]
```

Python Implementation

Below is a simplified Python example of image sharpening using matrix subtraction:

```python
# Original image matrix (grayscale values)
original_image = [
    [100, 150, 200],
    [150, 200, 250],
    [200, 250, 100]
]

# Function to apply Gaussian blur (for demonstration, simplified as average
 ↪ smoothing)
def apply_blur(image):
    blurred_image = []
    for i in range(len(image)):
        row = []
        for j in range(len(image[0])):
            neighbors = []
            for dx in [-1, 0, 1]:
                for dy in [-1, 0, 1]:
                    ni, nj = i + dx, j + dy
                    if 0 <= ni < len(image) and 0 <= nj < len(image[0]):
                        neighbors.append(image[ni][nj])
            blurred_value = sum(neighbors) // len(neighbors)
            row.append(blurred_value)
        blurred_image.append(row)
    return blurred_image

# Function for matrix subtraction (image sharpening)
def image_sharpening(original_image, blurred_image):
```

```
    sharpened_image = []
    for i in range(len(original_image)):
        row = []
        for j in range(len(original_image[0])):
            sharpened_value = original_image[i][j] - blurred_image[i][j]
            row.append(sharpened_value)
        sharpened_image.append(row)
    return sharpened_image

# Apply blur to simulate smoothed image
blurred_image = apply_blur(original_image)

# Perform matrix subtraction for image sharpening
sharpened_image = image_sharpening(original_image, blurred_image)

# Print the sharpened image
print("Sharpened Image:")
for row in sharpened_image:
    print(row)
```

1.9.1.3 Matrix Multiplication in Image Filtering (Convolution)

Matrix multiplication, specifically convolution in the context of image processing, is a fundamental operation used for various tasks such as smoothing, sharpening, edge detection, and more. Convolution involves applying a small matrix, known as a kernel or filter, to an image matrix. This process modifies the pixel values of the image based on the values in the kernel, effectively filtering the image.

Concept In grayscale images, each pixel value represents the intensity of light at that point. Convolution applies a kernel matrix over the image matrix to compute a weighted sum of neighborhood pixels. This weighted sum determines the new value of each pixel in the resulting filtered image.

Example:

Consider a grayscale image represented as a matrix:

```
original_image= [[100, 150, 200, 250],
 [150, 200, 250, 300],
 [200, 250, 300, 350],
 [250, 300, 350, 400]]
```

To perform smoothing (averaging) using a simple kernel:

```
[[1/9, 1/9, 1/9],
 [1/9, 1/9, 1/9],
 [1/9, 1/9, 1/9]]
```

The kernel is applied over the image using convolution:

```
smoothed_image[i][j] = sum(original_image[ii][jj] * kernel[k][l] for all (ii,
↪  jj) in neighborhood around (i, j))
```

Python Implementation

Here's a simplified Python example demonstrating convolution for image smoothing without external libraries:

```
# Original image matrix (grayscale values)
original_image = [
    [100, 150, 200, 250],
    [150, 200, 250, 300],
    [200, 250, 300, 350],
    [250, 300, 350, 400]
]

# Define a simple kernel/filter for smoothing (averaging)
kernel = [
    [1/9, 1/9, 1/9],
    [1/9, 1/9, 1/9],
    [1/9, 1/9, 1/9]
]

# Function for applying convolution (image filtering)
def apply_convolution(image, kernel):
    height = len(image)
    width = len(image[0])
    ksize = len(kernel)
    kcenter = ksize // 2  # Center of the kernel

    # Initialize result image
    filtered_image = [[0]*width for _ in range(height)]
```

```
    # Perform convolution
    for i in range(height):
        for j in range(width):
            sum = 0.0
            for k in range(ksize):
                for l in range(ksize):
                    ii = i + k - kcenter
                    jj = j + l - kcenter
                    if ii >= 0 and ii < height and jj >= 0 and jj < width:
                        sum += image[ii][jj] * kernel[k][l]
            filtered_image[i][j] = int(sum)

    return filtered_image

# Apply convolution to simulate smoothed image (averaging filter)
smoothed_image = apply_convolution(original_image, kernel)

# Print the smoothed image
print("Smoothed Image:")
for row in smoothed_image:
    print(row)
```

1.9.1.4 Determinant: Image Transformation

Concept The determinant of a transformation matrix helps understand how transformations like scaling affect an image. A transformation matrix determines how an image is scaled, rotated, or sheared.

Example:

Here, we compute the determinant of a scaling matrix to understand how the scaling affects the image area.

```
def calculate_determinant(matrix):
    a, b = matrix[0]
    c, d = matrix[1]
    return a * d - b * c

# Example transformation matrix (scaling)
transformation_matrix = [[2, 0], [0, 2]]
```

```
determinant = calculate_determinant(transformation_matrix)
print(f"Determinant of the transformation matrix: {determinant}")
```

This value indicates how the transformation scales the image area.

1.9.1.5 Rank: Image Rank and Data Compression

Concept The rank of a matrix indicates the number of linearly independent rows or columns. In image compression, matrix rank helps approximate an image with fewer data.

Example:

Here, we compute the rank of a matrix representing an image. A lower rank might indicate that the image can be approximated with fewer data.

```
def matrix_rank(matrix):
    def is_zero_row(row):
        return all(value == 0 for value in row)

    def row_echelon_form(matrix):
        A = [row[:] for row in matrix]
        m = len(A)
        n = len(A[0])
        rank = 0
        for i in range(min(m, n)):
            if A[i][i] != 0:
                for j in range(i + 1, m):
                    factor = A[j][i] / A[i][i]
                    for k in range(i, n):
                        A[j][k] -= factor * A[i][k]
                rank += 1
        return rank

    return row_echelon_form(matrix)

# Example matrix (image)
image_matrix = [[1, 2], [3, 4]]
rank = matrix_rank(image_matrix)
print(f"Rank of the image matrix: {rank}")
```

1.10 Matrix Operations Using `Python` Libraries

1.10.1 Introduction

In this section, we will explore the computational aspects of basic matrix algebra using Python. We will utilize the `SymPy` library for symbolic mathematics, which allows us to perform matrix operations and convert results into LaTeX format. Additionally, the `Pillow` (PIL) library will be used for image manipulation to demonstrate practical applications of these matrix operations in digital image processing. By the end of this section, you'll understand how to implement matrix operations and apply them to real-world problems such as image blending, sharpening, filtering, and solving systems of equations.

1.10.2 Introduction to SymPy

`SymPy` is a powerful Python library designed for symbolic mathematics. It provides tools for algebraic operations, equation solving, and matrix handling in a symbolic form. This makes it ideal for educational purposes and theoretical work where exact results are needed.

1.10.2.1 Key Matrix Functions in SymPy

- **Matrix Addition**: Adds two matrices element-wise.
- **Matrix Subtraction**: Subtracts one matrix from another element-wise.
- **Matrix Multiplication**: Multiplies two matrices using the dot product.
- **Matrix Power**: Raises a matrix to a given power using matrix multiplication.

  Example 1: Matrix Addition

Pseudocode

```
FUNCTION matrix_add():
    # Define matrices A and B
    A = [[1, 2], [3, 4]]
    B = [[5, 6], [7, 8]]

    # Check if matrices A and B have the same dimensions
    if dimensions_of(A) != dimensions_of(B):
         raise ValueError("Matrices must have the same dimensions")

    # Initialize result matrix with zeros
    result = [[0 for _ in range(len(A[0]))] for _ in range(len(A))]
```

```
    # Add corresponding elements from A and B
    for i in range(len(A)):
        for j in range(len(A[0])):
            result[i][j] = A[i][j] + B[i][j]

# Return the result matrix
    return result
ENDFUNCTION
```

Python implementation of the above pseudocode is given below:

```python
import sympy as sy
sy.init_printing()
# Define matrices A and B
A = sy.Matrix([[1, 2], [3, 4]])
B = sy.Matrix([[5, 6], [7, 8]])

# Add matrices
C = A + B

# Print the result in symbolic form
print("Matrix Addition Result:")
display(C)

# Convert to LaTeX code for documentation or presentation
#latex_code = sy.latex(C)
#print("LaTeX Code for Addition Result:")
#print(latex_code)
```

```
Matrix Addition Result:
```

$$\begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$$

Example 2: Matrix Subtraction

Pseudocode:

```
# Define matrices A and B
A = [[5, 6], [7, 8]]
B = [[1, 2], [3, 4]]

# Check if matrices A and B have the same dimensions
if dimensions_of(A) != dimensions_of(B):
    raise ValueError("Matrices must have the same dimensions")

# Initialize result matrix with zeros
result = [[0 for _ in range(len(A[0]))] for _ in range(len(A))]

# Subtract corresponding elements from A and B
for i in range(len(A)):
    for j in range(len(A[0])):
        result[i][j] = A[i][j] - B[i][j]

# Return the result matrix
return result
```

Python implementation of the above pseudocode is given below:

```
from sympy import Matrix
# Define matrices A and B
A = Matrix([[5, 6], [7, 8]])
B = Matrix([[1, 2], [3, 4]])

# Subtract matrices
C = A - B

# Print the result in symbolic form
print("Matrix Subtraction Result:")
display(C)

# Convert to LaTeX code for documentation or presentation
latex_code = sy.latex(C)
print("LaTeX Code for Subtraction Result:")
print(latex_code)
```

```
Matrix Subtraction Result:
LaTeX Code for Subtraction Result:
\left[\begin{matrix}4 & 4\\4 & 4\end{matrix}\right]
```

$$\begin{bmatrix} 4 & 4 \\ 4 & 4 \end{bmatrix}$$

Example 3: Matrix Multiplication

Pseudocode:

```
# Define matrices A and B
A = [[1, 2], [3, 4]]
B = [[5, 6], [7, 8]]

# Check if the number of columns in A equals the number of rows in B
if len(A[0]) != len(B):
    raise ValueError("Number of columns in A must equal number of rows in B")

# Initialize result matrix with zeros
result = [[0 for _ in range(len(B[0]))] for _ in range(len(A))]

# Multiply matrices A and B
for i in range(len(A)):
    for j in range(len(B[0])):
        for k in range(len(B)):
            result[i][j] += A[i][k] * B[k][j]

# Return the result matrix
return result
```

Python implementation of the above pseudocode is given below:

```
# Define matrices A and B
A = Matrix([[1, 2], [3, 4]])
B = Matrix([[5, 6], [7, 8]])

# Multiply matrices
M = A * B

# Print the result in symbolic form
print("Matrix Multiplication Result:")
display(M)

# Convert to LaTeX code for documentation or presentation
```

```
latex_code = sy.latex(M)
print("LaTeX Code for Multiplication Result:")
print(latex_code)
```

```
Matrix Multiplication Result:
LaTeX Code for Multiplication Result:
\left[\begin{matrix}19 & 22\\43 & 50\end{matrix}\right]
```

$$\begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

Example 3: Matrix Multiplication

Pseudocode:

```
# Define matrices A and B
A = [[1, 2], [3, 4]]
B = [[5, 6], [7, 8]]

# Check if the number of columns in A equals the number of rows in B
if len(A[0]) != len(B):
    raise ValueError("Number of columns in A must equal number of rows in B")

# Initialize result matrix with zeros
result = [[0 for _ in range(len(B[0]))] for _ in range(len(A))]

# Multiply matrices A and B
for i in range(len(A)):
    for j in range(len(B[0])):
        for k in range(len(B)):
            result[i][j] += A[i][k] * B[k][j]

# Return the result matrix
return result
```

Python code for implementing the above pseudocode is shown below:

```
# Define matrices A and B
A = Matrix([[1, 2], [3, 4]])
B = Matrix([[5, 6], [7, 8]])

# Multiply matrices
M = A * B

# Print the result in symbolic form
print("Matrix Multiplication Result:")
display(M)

# Convert to LaTeX code for documentation or presentation
latex_code = sy.latex(M)
print("LaTeX Code for Multiplication Result:")
print(latex_code)
```

```
Matrix Multiplication Result:
LaTeX Code for Multiplication Result:
\left[\begin{matrix}19 & 22\\43 & 50\end{matrix}\right]
```

$$\left[\begin{matrix}19 & 22\\43 & 50\end{matrix}\right]$$

Example 4: Matrix Power

Pseudocode:

```
# Define matrix A and power n
A = [[1, 2], [3, 4]]
n = 2

# Initialize result matrix as identity matrix
result = identity_matrix_of(len(A))

# Compute A raised to the power of n
for _ in range(n):
    result = matrix_multiply(result, A)

# Return the result matrix
return result
```

Python implementation of the above pseudocode is shown below:

```
# Define matrix A
A = Matrix([[1, 2], [3, 4]])

# Compute matrix A raised to the power of 2
n = 2
C = A**n

# Print the result in symbolic form
print("Matrix Power Result:")
display(C)

# Convert to LaTeX code for documentation or presentation
latex_code = sy.latex(C)
print("LaTeX Code for Power Result:")
print(latex_code)
```

```
Matrix Power Result:
LaTeX Code for Power Result:
\left[\begin{matrix}7 & 10\\15 & 22\end{matrix}\right]
```

$$\left[\begin{matrix}7 & 10\\15 & 22\end{matrix}\right]$$

1.10.2.2 Introduction to PIL for Image Manipulation

The `PIL` (Python Imaging Library), now known as `Pillow`, provides essential tools for opening, manipulating, and saving various image file formats. In this session, we will use Pillow to perform image operations such as resizing and blending to demonstrate the practical applications of these matrix operations in digital image processing.

Matrix operations have significant applications in digital image processing. These operations can manipulate images in various ways, from blending to filtering. Below we will discuss how matrix addition, subtraction, and multiplication are used in real-time image processing tasks.

1. **Matrix Addition: Image Blending**

Matrix addition can be used to blend two images by adding their pixel values. This process can be straightforward or involve weighted blending.

Example 1: Simple Image Blending

```
import numpy as np
from PIL import Image
import urllib.request
urllib.request.urlretrieve('http://lenna.org/len_top.jpg',"input.jpg")
img1 = Image.open("input.jpg") #loading first image

urllib.request.urlretrieve('https://www.keralatourism.org/images/destination
↪ /large/thekkekudi_cave_temple_in_pathanamthitta20131205062431_315_1.jpg'
↪ ,"input2.jpg")

img2 = Image.open("input2.jpg")# loading second image

# Resize second image to match the size of the first image
img2 = img2.resize(img1.size)
# Convert images to numpy arrays
arr1 = np.array(img1)
arr2 = np.array(img2)

# Add the images
blended_arr = arr1 + arr2

# Clip the values to be in the valid range [0, 255]
blended_arr = np.clip(blended_arr, 0, 255).astype(np.uint8)

# Convert back to image
blended_img = Image.fromarray(blended_arr)

# Save or display the blended image
#blended_img.save('blended_image.jpg')
#blended_img.show()
#blended_img #display the blended image
```

The input and output images are shown below:

```
img1
```

```
img2
```

```
blended_img
```

Example 2: Weighted Image Blending

```
# Blend with weights
alpha = 0.7
blended_arr = alpha * arr1 + (1 - alpha) * arr2

# Clip the values to be in the valid range [0, 255]
blended_arr = np.clip(blended_arr, 0, 255).astype(np.uint8)

# Convert back to image
blended_img = Image.fromarray(blended_arr)

# Save or display the weighted blended image
#blended_img.save('weighted_blended_image.jpg')
#blended_img.show()
blended_img
```

1.10.2.3 Matrix Subtraction: Image Sharpening

Matrix subtraction can be used to sharpen images by subtracting a blurred version of the image from the original.

Example 1: Sharpening by Subtracting Blurred Image

```
from PIL import Image, ImageFilter
# Convert image to grayscale for simplicity
img_gray = img1.convert('L')
arr = np.array(img_gray)

# Apply Gaussian blur
blurred_img = img_gray.filter(ImageFilter.GaussianBlur(radius=5))
blurred_arr = np.array(blurred_img)

# Sharpen the image by subtracting blurred image
sharpened_arr = arr - blurred_arr

# Clip the values to be in the valid range [0, 255]
sharpened_arr = np.clip(sharpened_arr, 0, 255).astype(np.uint8)

# Convert back to image
sharpened_img = Image.fromarray(sharpened_arr)
```

```
# Save or display the sharpened image
#sharpened_img.save('sharpened_image.jpg')
#sharpened_img.show()
sharpened_img
```

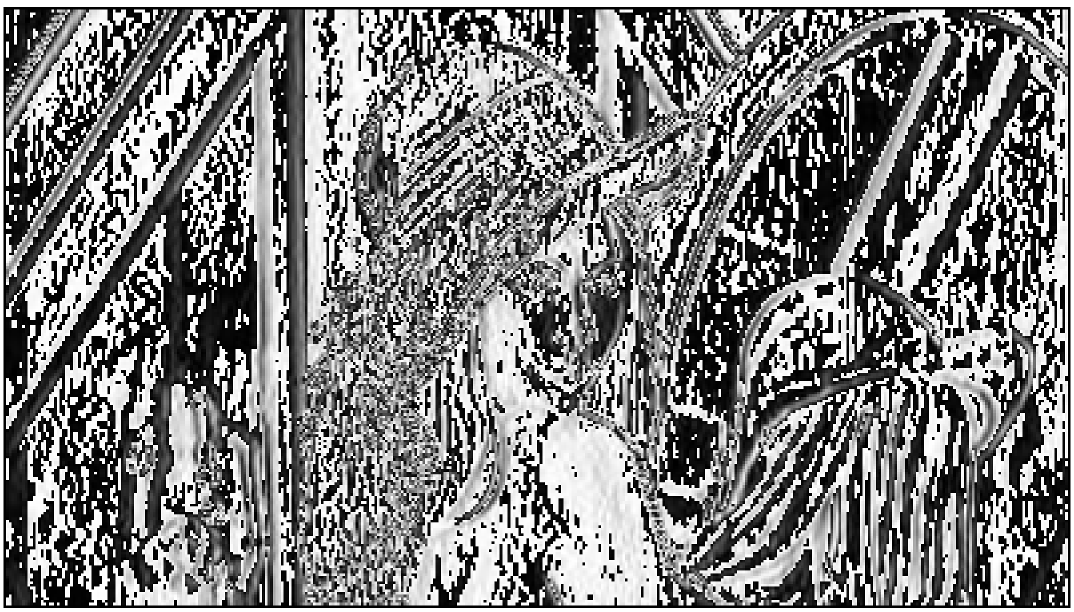
1.10.2.4 Matrix Multiplication: Image Filtering (Convolution)

Matrix multiplication is used in image filtering to apply convolution kernels for various effects.

Example 1: Applying a Convolution Filter

```
# Define a simple convolution kernel (e.g., edge detection)
kernel = np.array([
    [1, 0, -1],
    [1, 0, -1],
    [1, 0, -1]
])

# Convert the image to grayscale for simplicity
img_gray = img1.convert('L')
arr = np.array(img_gray)
```

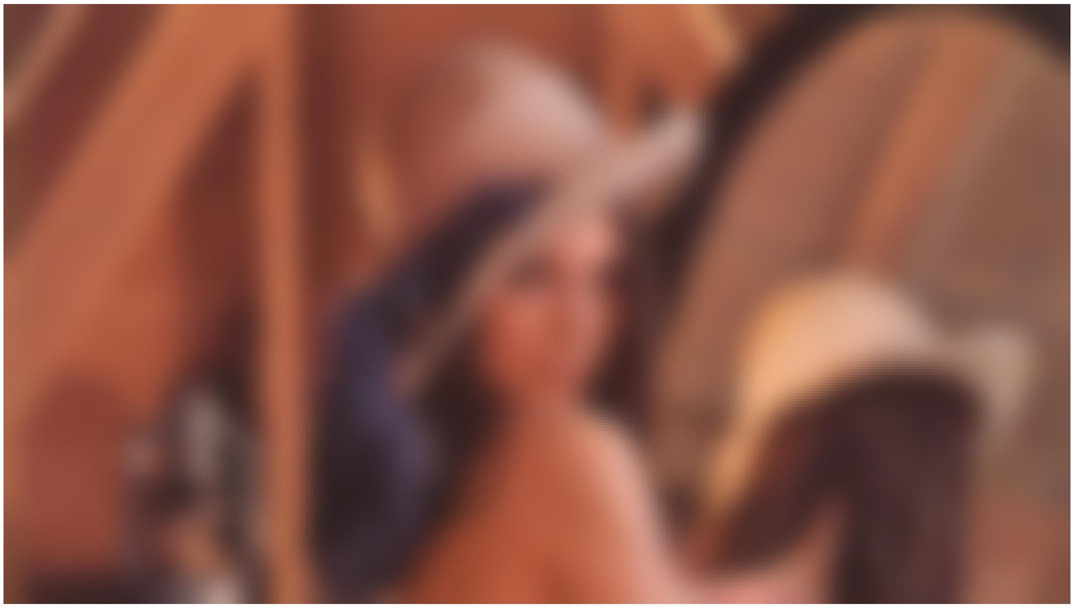
```
# Apply convolution
filtered_arr = np.zeros_like(arr)
for i in range(1, arr.shape[0] - 1):
    for j in range(1, arr.shape[1] - 1):
        region = arr[i-1:i+2, j-1:j+2]
        filtered_arr[i, j] = np.sum(region * kernel)

# Convert back to image
filtered_img = Image.fromarray(np.clip(filtered_arr, 0,
 ↪  255).astype(np.uint8))

# Save or display the filtered image
#filtered_img.save('filtered_image.jpg')
#filtered_img.show()
filtered_img
```



Example 2: Applying a Gaussian Blur Filter

```
# Define a Gaussian blur filter
blurred_img = img1.filter(ImageFilter.GaussianBlur(radius=5))

# Save or display the blurred image
#blurred_img.save('blurred_image.jpg')
```

```
#blurred_img.show()
blurred_img
```



1.10.2.5 Solving Systems of Equations and Applications

Introduction

Solving systems of linear equations is crucial in various image processing tasks, such as image transformation, camera calibration, and object detection. In this section, we will demonstrate how to solve systems of linear equations using Python and explore practical applications in image processing.

Example 1: Solving a System of Equations

Consider the system:

$$\begin{cases} 2x + 3y & = 13 \\ 4x - y & = 7 \end{cases}$$

Python Implementation

```python
from sympy import Matrix
# Define the coefficient matrix and constant matrix
A = Matrix([[2, 3], [4, -1]])
B = Matrix([13, 7])
# Solve the system of equations
solution = A.solve_least_squares(B)
# Print the solution
print("Solution to the System of Equations:")
print(solution)
```

```
Solution to the System of Equations:
Matrix([[17/7], [19/7]])
```

1.11 Conclusion

In this chapter, we transitioned from understanding fundamental matrix operations to applying them in practical scenarios, specifically in the realm of image processing. We began by covering essential matrix operations such as addition, subtraction, multiplication, and determinant calculations, providing both pseudocode and detailed explanations. This foundational knowledge was then translated into Python code, demonstrating how to perform these operations computationally.

We further explored the application of these matrix operations to real-world image processing tasks. By applying techniques such as image blending, sharpening, filtering, and transformation, we illustrated how theoretical concepts can be used to manipulate and enhance digital images effectively. These practical examples highlighted the significance of matrix operations in solving complex image processing challenges.

By integrating theoretical understanding with practical implementation, this chapter reinforced how matrix operations form the backbone of many image processing techniques. This blend of theory and practice equips you with essential skills for tackling advanced problems and developing innovative solutions in the field of image processing and beyond.

2 Transforming Linear Algebra to Computational Language

2.1 Introduction

In the first module, we established a solid foundation in matrix algebra by exploring pseudocode and implementing fundamental matrix operations using `Python`. We practiced key concepts such as matrix addition, subtraction, multiplication, and determinants through practical examples in image processing, leveraging the `SymPy` library for symbolic computation.

As we begin the second module, **"Transforming Linear Algebra to Computational Language,"** our focus will shift towards applying these concepts with greater depth and actionable insight. This module is designed to bridge the theoretical knowledge from matrix algebra with practical computational applications. You will learn to interpret and utilize matrix operations, solve systems of equations, and analyze the rank of matrices within a variety of real-world contexts.

A new concept we will introduce is the **Rank-Nullity Theorem**, which provides a fundamental relationship between the rank of a matrix and the dimensions of its null space. This theorem is crucial for understanding the solution spaces of linear systems and the properties of linear transformations. By applying this theorem, you will be able to gain deeper insights into the structure of solutions and the behavior of matrix transformations.

This transition will not only reinforce your understanding of linear algebra but also enhance your ability to apply these concepts effectively in computational settings. Through engaging examples and practical exercises, you will gain valuable experience in transforming abstract mathematical principles into tangible solutions, setting a strong groundwork for advanced computational techniques.

2.2 Relearning of Terms and Operations in Linear Algebra

In this section, we will revisit fundamental matrix operations such as addition, subtraction, scaling, and more through practical examples. Our goal is to transform theoretical linear algebra into modern computational applications. We will demonstrate these concepts using `Python`, focusing on practical and industrial applications.

2.2.1 Matrix Addition and Subtraction in Data Analysis

Matrix addition and subtraction are fundamental operations that help in combining datasets and analyzing differences.

Simple Example: Combining Quarterly Sales Data

We begin with quarterly sales data from different regions and combine them to get the total sales. The sales data is given in Table 2.1. A ar plot of the total sales is shown in Fig 2.1.

Table 2.1: Quarterly Sales Data

Region	Q1	Q2	Q3	Q4
A	2500	2800	3100	2900
B	1500	1600	1700	1800

From Scratch `Python` Implementation:

```python
import numpy as np
import matplotlib.pyplot as plt

# Quarterly sales data
sales_region_a = np.array([2500, 2800, 3100, 2900])
sales_region_b = np.array([1500, 1600, 1700, 1800])

# Combine sales data
total_sales = sales_region_a + sales_region_b

# Visualization
quarters = ['Q1', 'Q2', 'Q3', 'Q4']
plt.bar(quarters, total_sales, color='skyblue')
plt.xlabel('Quarter')
plt.ylabel('Total Sales')
plt.title('Combined Quarterly Sales Data for Regions A and B')
plt.show()
```

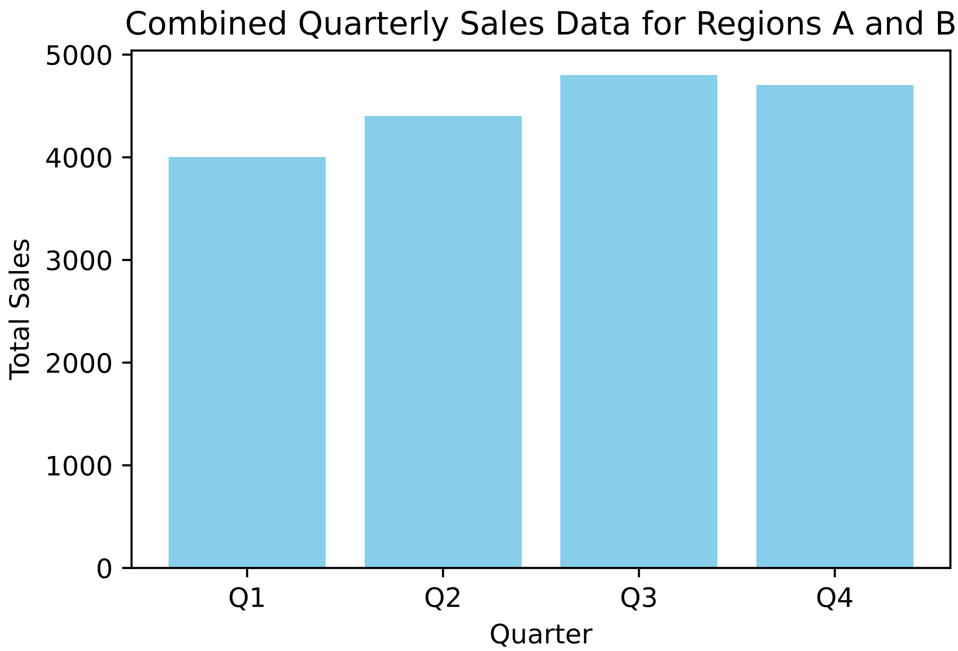


Figure 2.1: Computing Total Sales using `Numpy` aggregation method

In the above `Python` code, we have performed the aggregation operation with the `NumPy` method. Same can be done in a more data analysis style using `pandas` inorder to handle tabular data meaningfully. In this approach, quarterly sales data of each region is stored as `DataFrames`(like an excel sheet). The we combine these two `DataFrames` into one. After that create a new row with index 'Total' and populate this row with sum of quarterly sales in Region A and Region B. Finally a bar plot is created using this 'Total' sales. Advantage of this approach is that we don't need the `matplotlib` library to create visualizations!. The EDA using this approach is shown in Fig 2.2.

```
import pandas as pd
import matplotlib.pyplot as plt

# DataFrames for quarterly sales data
df_a = pd.DataFrame({'Q1': [2500], 'Q2': [2800], 'Q3': [3100], 'Q4': [2900]},
↪  index=['Region A'])
df_b = pd.DataFrame({'Q1': [1500], 'Q2': [1600], 'Q3': [1700], 'Q4': [1800]},
↪  index=['Region B'])

# Combine data
df_combined = df_a.add(df_b, fill_value=0)
```

```
df_combined.loc["Total"] = df_combined.sum(axis=0)
# Visualization
df_combined.loc["Total"].plot(kind='bar', color=['green'])
plt.xlabel('Quarter')
plt.ylabel('Total Sales')
plt.title('Combined Quarterly Sales Data for Regions A and B')
plt.show()
```

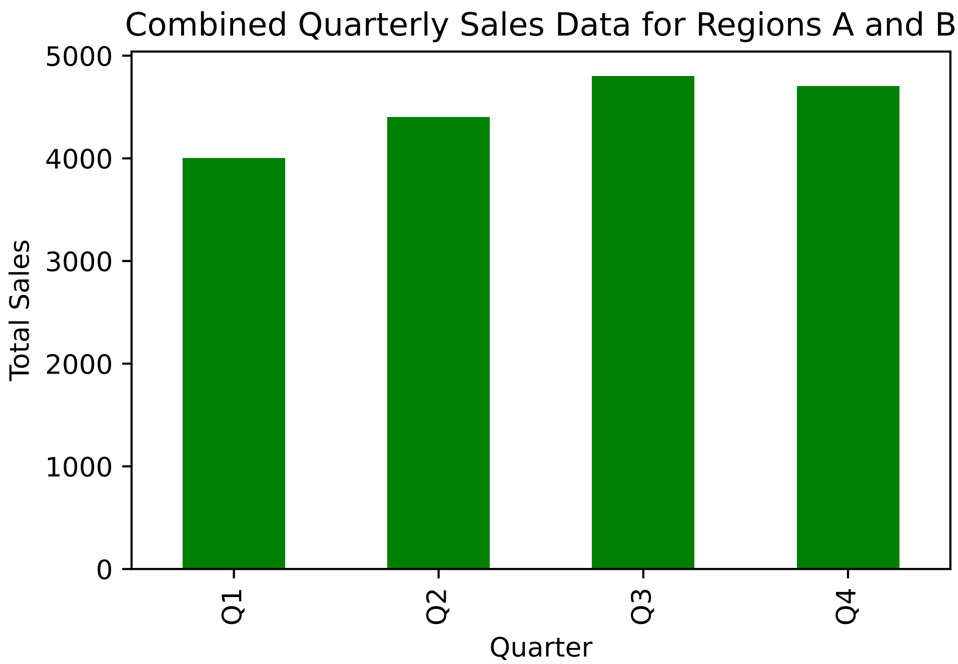


Figure 2.2: Computation of Total Sales using `Pandas` method

We can extend this in to more advanced examples. Irrespective to the size of the data, for representation and aggregation tasks matrix models are best options and are used in industry as a standard. Let us consider an advanced example to analyse difference in stock prices. For this example we are using a simulated data. The python code for this simulation process is shown in Fig 2.3.

```
import numpy as np
import matplotlib.pyplot as plt

# Simulated observed and predicted stock prices
observed_prices = np.random.uniform(100, 200, size=(100, 5))
```

```
predicted_prices = np.random.uniform(95, 210, size=(100, 5))

# Calculate the difference matrix
price_differences = observed_prices - predicted_prices

# Visualization
plt.imshow(price_differences, cmap='coolwarm', aspect='auto')
plt.colorbar()
plt.title('Stock Price Differences')
plt.xlabel('Stock Index')
plt.ylabel('Day Index')
plt.show()
```

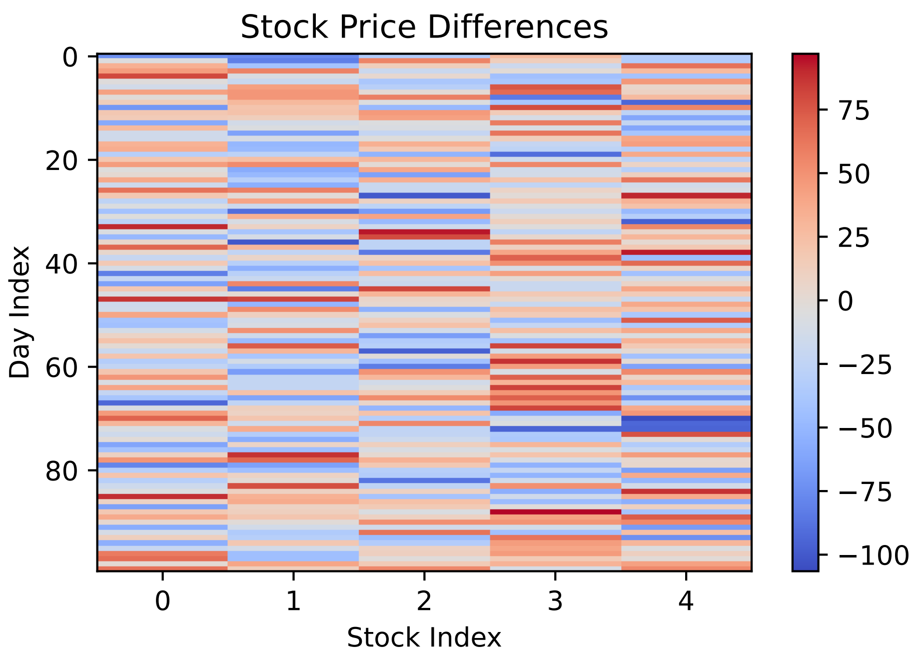


Figure 2.3: Demonstration of Stock Price simulated from a Uniform Distribution

Another important matrix operation relevant to data analytics and Machine Learning application is scaling. This is considered as a statistical tool to make various features (attributes) in to same scale so as to avoid unnecessary misleading impact in data analysis and its intepretation. In Machine Learning context, this pre-processing stage is inevitable so as to make the model relevant and usable.

Simple Example: Normalizing Employee Performance Data

Table 2.2: Employee Performance Data

Employee	Metric A	Metric B
X	80	700
Y	90	800
Z	100	900
A	110	1000
B	120	1100

Using simple python code we can simulate the model for `min-max` scaling. The formula for `min-max` scaling is:

$$min_max(X) = \frac{X - min(X)}{max(X) - min(X)}$$

For example, while applying the `min-max` scaling in the first value of Metric A, the scaled value is

$$min_max(80)\frac{80 - 80}{120 - 80} = 0$$

Similarly

$$min_max(100)\frac{100 - 80}{120 - 80} = 0.5$$

When we apply this formula to Metric A and Metric B, the scaled output from Table 2.2 will be as follows:

Table 2.3: Employee Performance Data

Employee	Metric A	Metric B
X	0.00	0.00
Y	0.25	0.25
Z	0.50	0.50
A	0.75	0.75
B	1.00	1.00

It is interesting to look into the scaled data! In the orginal table (Table 2.2) it is looked like Metric B is superior. But from the scaled table (Table 2.3), it is clear that both the Metrics are representing same relative information. This will help us to identify the redundency in measure and so skip any one of the Metric before analysis!.

The same can be achieved through a matrix operation. The `Python` implementation of this scaling process is shown in Fig 2.4.

```python
import numpy as np
import matplotlib.pyplot as plt

# Employee performance data with varying scales
data = np.array([[80, 700], [90, 800], [100, 900], [110, 1000], [120, 1100]])

# Manual scaling
min_vals = np.min(data, axis=0)
max_vals = np.max(data, axis=0)
scaled_data = (data - min_vals) / (max_vals - min_vals)

# Visualization
plt.figure(figsize=(8, 5))
plt.subplot(1, 2, 1)
plt.imshow(data, cmap='viridis')
plt.title('Original Data')
plt.colorbar()

plt.subplot(1, 2, 2)
plt.imshow(scaled_data, cmap='viridis')
plt.title('Scaled Data')
plt.colorbar()

plt.show()
```

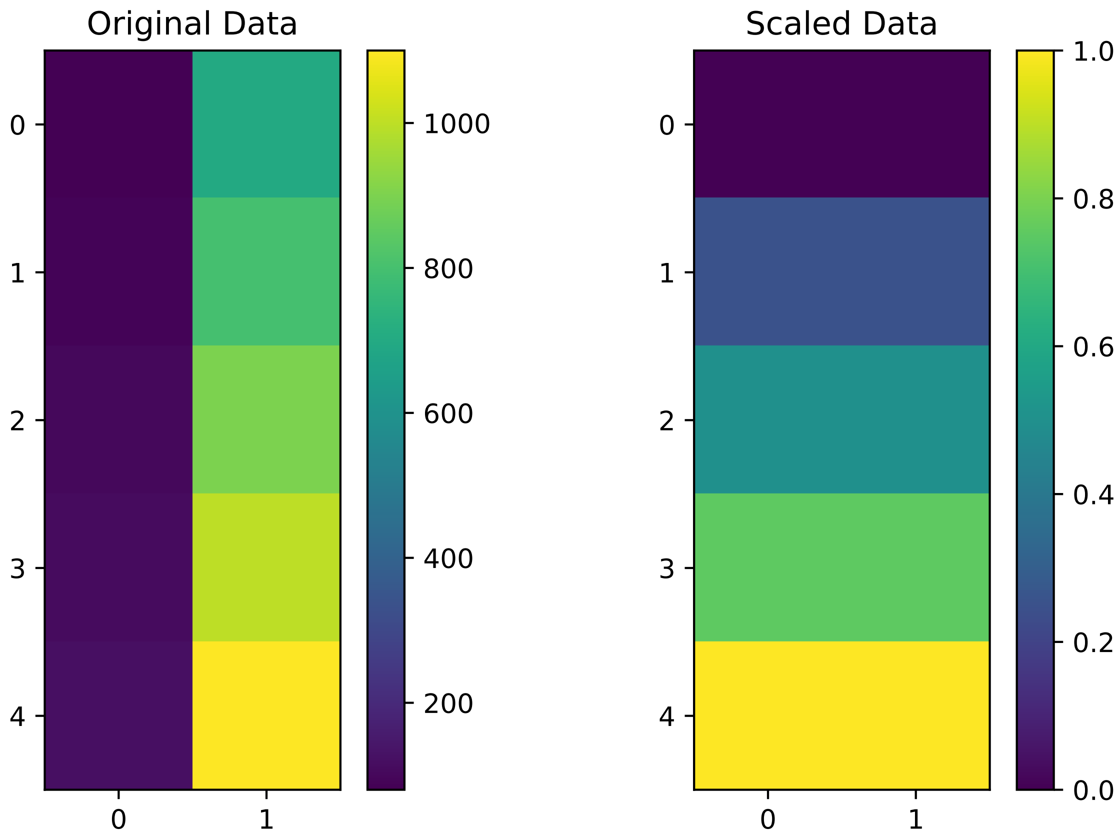


Figure 2.4: Total sales using `pandas` method

From the first sub plot, it is clear that there is a significant difference in the distributions (Metric A and Metric B values). But the second sub plot shows that both the distributions have same pattern and the values ranges between 0 and 1. In short the visualization is more appealing and self explanatory in this case.

> **Note**
>
> The `min-max` scaling method will confine the feature values (attributes) into the range $[0, 1]$. So in effect all the features are scaled proportionally to the data spectrum.

Similarly, we can use the `standard scaling` (transformation to normal distribution) using the transformation $\frac{x-\bar{x}}{\sigma}$. Scaling table is given as a practice task to the reader. The python code for this operation is shown in Fig 2.5.

```python
# Standard scaling from scratch
def standard_scaling(data):
    mean = np.mean(data, axis=0)
    std = np.std(data, axis=0)
    scaled_data = (data - mean) / std
    return scaled_data

# Apply standard scaling
scaled_data_scratch = standard_scaling(data)

print("Standard Scaled Data (from scratch):\n", scaled_data_scratch)

# Visualization
plt.figure(figsize=(6, 5))
plt.subplot(1, 2, 1)
plt.imshow(data, cmap='viridis')
plt.title('Original Data')
plt.colorbar()

plt.subplot(1, 2, 2)
plt.imshow(scaled_data_scratch, cmap='viridis')
plt.title('Scaled Data')
plt.colorbar()

plt.show()
```

```
Standard Scaled Data (from scratch):
 [[-1.41421356 -1.41421356]
 [-0.70710678 -0.70710678]
 [ 0.          0.        ]
 [ 0.70710678  0.70710678]
 [ 1.41421356  1.41421356]]
```

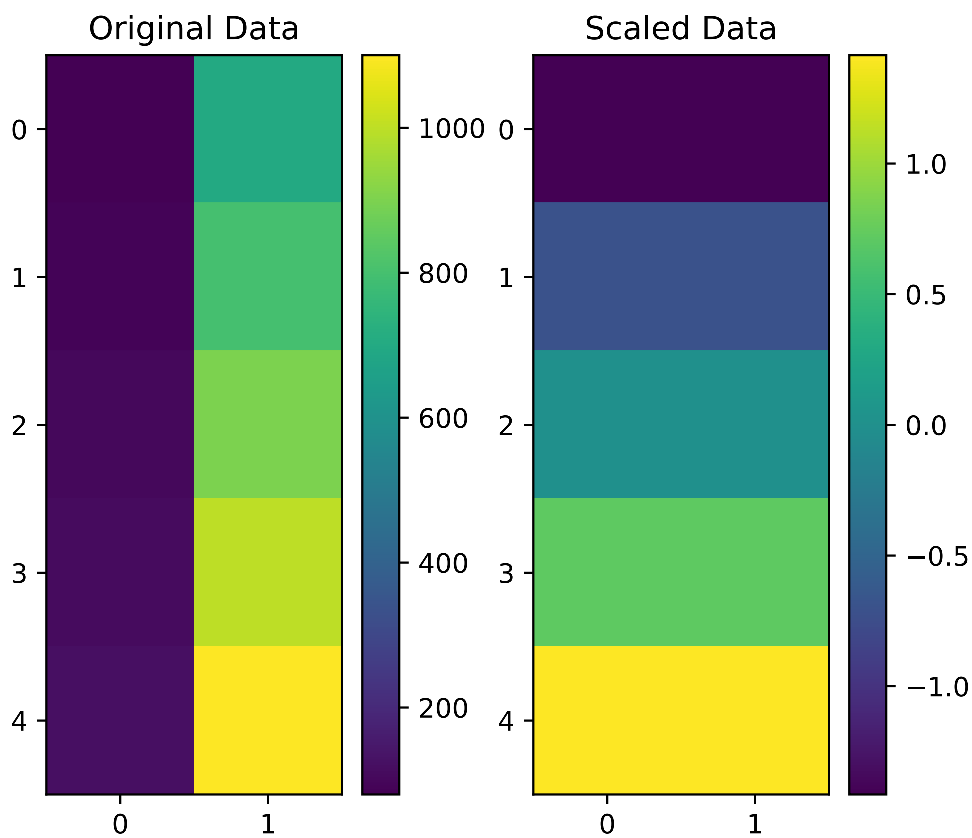


Figure 2.5: Min-max scaling using basic python

To understand the effect of standard scaling, let us consider Fig 2.6. This plot create the frequency distribution of the data as a histogram along with the density function. From the first sub-plot, it is clear that the distribution has multiple modes (peaks). When we apply the standard scaling, the distribution become un-modal(only one peek). This is demonstrated in the second sub-plot.

```
# Standard scaling from scratch
import seaborn as sns
# Create plots
plt.figure(figsize=(6, 5))

# Plot for original data
plt.subplot(1, 2, 1)
sns.histplot(data, kde=True, bins=10, palette="viridis")
plt.title('Original Data Distribution')
```

```python
plt.xlabel('Value')
plt.ylabel('Frequency')

# Plot for standard scaled data
plt.subplot(1, 2, 2)
sns.histplot(scaled_data_scratch, kde=True, bins=10, palette="viridis")
plt.title('Standard Scaled Data Distribution')
plt.xlabel('Value')
plt.ylabel('Frequency')

plt.tight_layout()
plt.show()
```

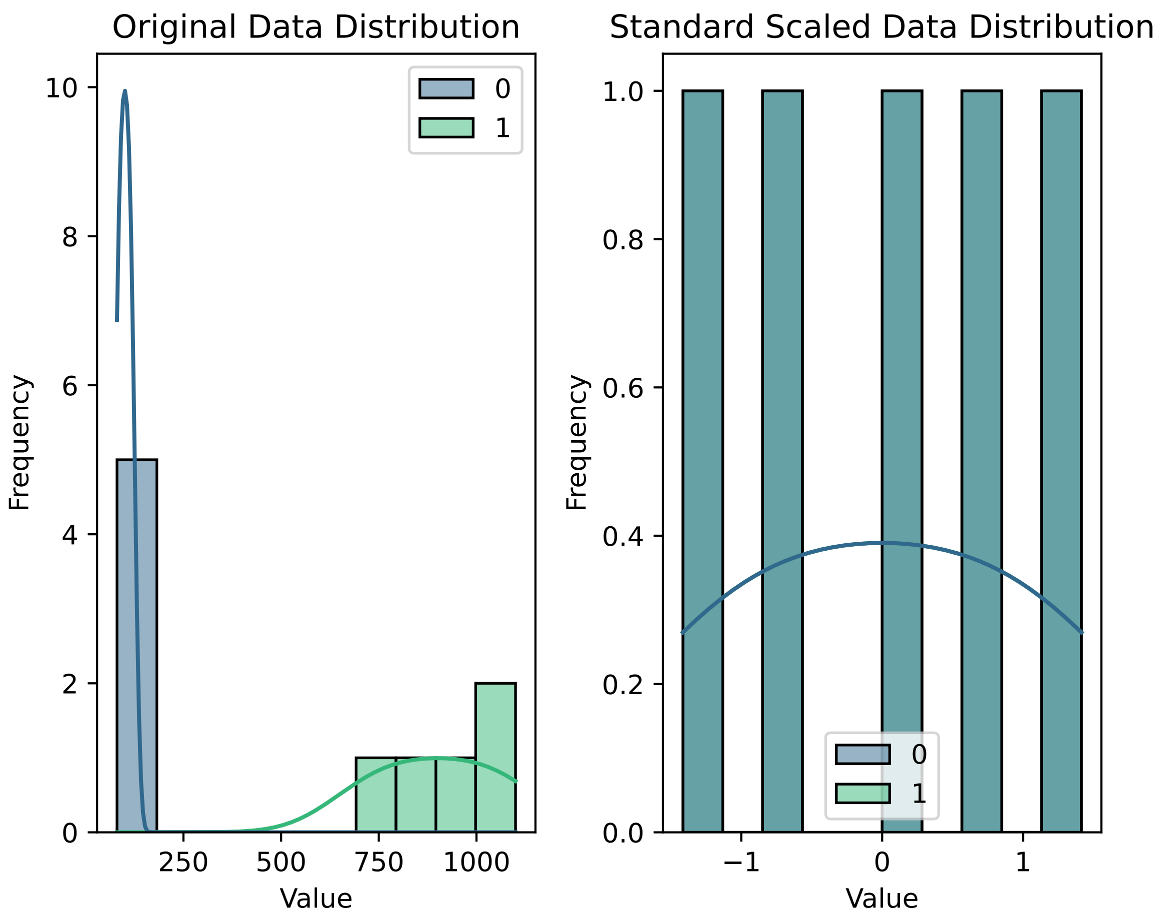


Figure 2.6: Impact of standard scaling on the distribution

A scatter plot showing the compare the impact of scaling on the given distribution is shown in Fig 2.7.

```
# Plot original and scaled data
plt.figure(figsize=(6, 5))

# Original Data
plt.subplot(1, 3, 1)
plt.scatter(data[:, 0], data[:, 1], color='blue')
plt.title('Original Data')
plt.xlabel('Metric A')
plt.ylabel('Metric B')

# Standard Scaled Data
plt.subplot(1, 3, 2)
plt.scatter(scaled_data_scratch[:, 0], scaled_data_scratch[:, 1],
 ↪  color='green')
plt.title('Standard Scaled Data')
plt.xlabel('Metric A (Standard Scaled)')
plt.ylabel('Metric B (Standard Scaled)')

# Min-Max Scaled Data
plt.subplot(1, 3, 3)
plt.scatter(scaled_data[:, 0], scaled_data[:, 1], color='red')
plt.title('Min-Max Scaled Data')
plt.xlabel('Metric A (Min-Max Scaled)')
plt.ylabel('Metric B (Min-Max Scaled)')

plt.tight_layout()
plt.show()
```

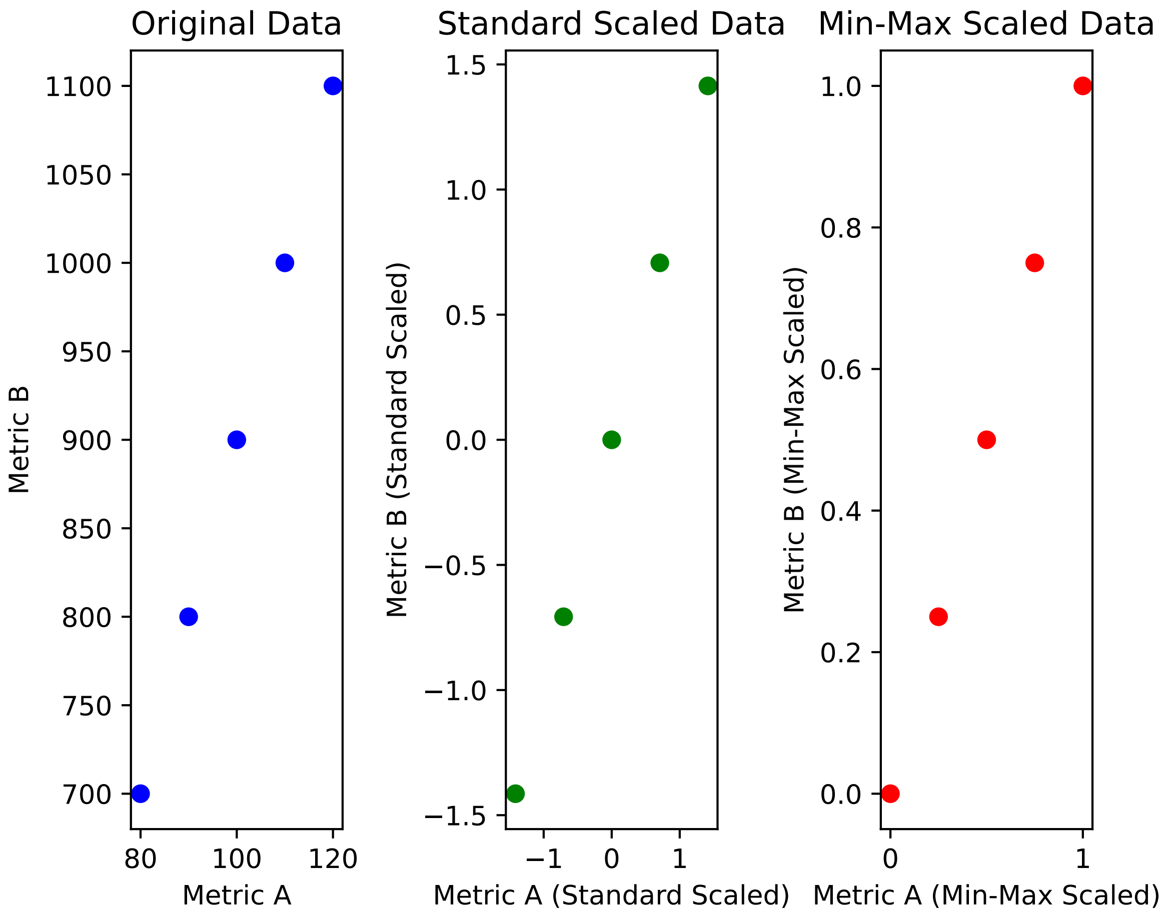


Figure 2.7: Comparison of impact of scaling on the distribution

From the Fig 2.7, it is clear that the scaling does not affect the pattern of the data, instead it just scale the distribution proportionally!

We can use the `scikit-learn` library for do the same thing in a very simple handy approach. The `python` code for this job is shown below.

```
from sklearn.preprocessing import MinMaxScaler

# Min-max scaling using sklearn
scaler = MinMaxScaler()
min_max_scaled_data_sklearn = scaler.fit_transform(data)

print("Min-Max Scaled Data (using sklearn):\n", min_max_scaled_data_sklearn)
```

```
Min-Max Scaled Data (using sklearn):
 [[0.   0.  ]
 [0.25 0.25]
 [0.5  0.5 ]
 [0.75 0.75]
 [1.   1.  ]]
```

```
from sklearn.preprocessing import StandardScaler

# Standard scaling using sklearn
scaler = StandardScaler()
scaled_data_sklearn = scaler.fit_transform(data)

print("Standard Scaled Data (using sklearn):\n", scaled_data_sklearn)
```

```
Standard Scaled Data (using sklearn):
 [[-1.41421356 -1.41421356]
 [-0.70710678 -0.70710678]
 [ 0.          0.        ]
 [ 0.70710678  0.70710678]
 [ 1.41421356  1.41421356]]
```

A scatter plot showing the impact on scaling is shown in Fig 2.8. This plot compare the `mmin-max` and `standard-scaling`.

```
# Plot original and scaled data
plt.figure(figsize=(6, 5))

# Original Data
plt.subplot(1, 3, 1)
plt.scatter(data[:, 0], data[:, 1], color='blue')
plt.title('Original Data')
plt.xlabel('Metric A')
plt.ylabel('Metric B')

# Standard Scaled Data
plt.subplot(1, 3, 2)
plt.scatter(scaled_data_sklearn[:, 0], scaled_data_sklearn[:, 1],
 ↪  color='green')
plt.title('Standard Scaled Data')
plt.xlabel('Metric A (Standard Scaled)')
```

```
plt.ylabel('Metric B (Standard Scaled)')

# Min-Max Scaled Data
plt.subplot(1, 3, 3)
plt.scatter(min_max_scaled_data_sklearn[:, 0], min_max_scaled_data_sklearn[:,
 ↪  1], color='red')
plt.title('Min-Max Scaled Data')
plt.xlabel('Metric A (Min-Max Scaled)')
plt.ylabel('Metric B (Min-Max Scaled)')

plt.tight_layout()
plt.show()
```

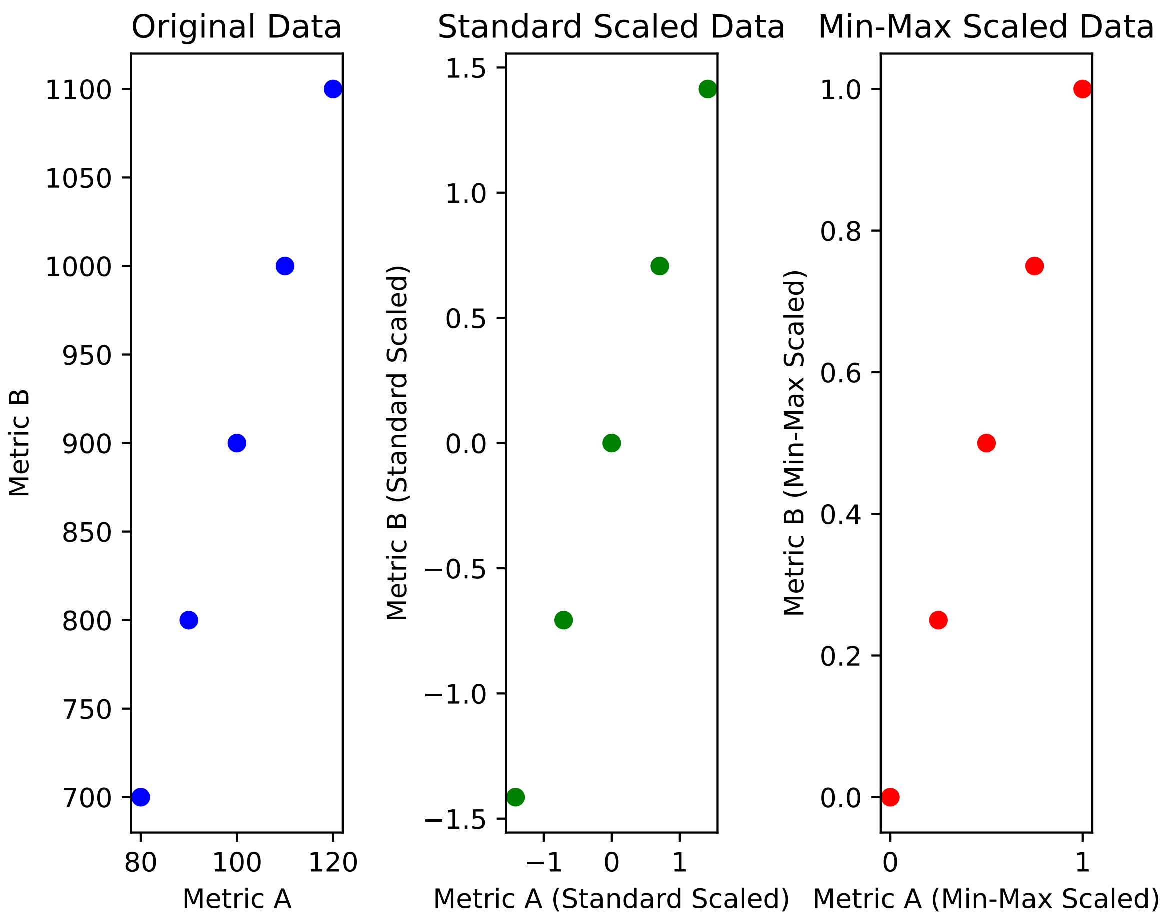


Figure 2.8: Camparison of Min-max and standard scalings with original data

2.2.2 More on Matrix Product and its Applications

In the first module of our course, we introduced matrix products as scalar projections, focusing on how matrices interact through basic operations. In this section, we will expand on this by exploring different types of matrix products that have practical importance in various fields. One such product is the *Hadamard product*, which is particularly useful in applications ranging from image processing to neural networks and statistical analysis. We will cover the definition, properties, and examples of the Hadamard product, and then delve into practical applications with simulated data.

2.2.2.1 Hadamard Product

The Hadamard product (or element-wise product) of two matrices is a binary operation that combines two matrices of the same dimensions to produce another matrix of the same dimensions, where each element is the product of corresponding elements in the original matrices.

! Definition (Hadamard Product):

For two matrices A and B of the same dimension $m \times n$, the Hadamard product $A \circ B$ is defined as:

$$(A \circ B)_{ij} = A_{ij} \cdot B_{ij}$$

where $\cdot$ denotes element-wise multiplication.

i Properties of Hadamard Product

1. **Commutativity**:
$$A \circ B = B \circ A$$
2. **Associativity**:
$$(A \circ B) \circ C = A \circ (B \circ C)$$
3. **Distributivity**:
$$A \circ (B + C) = (A \circ B) + (A \circ C)$$

Some simple examples to demonstrate the Hadamard product is given below.

Example 1: Basic Hadamard Product

Given matrices:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \quad B = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$$

The Hadamard product $A \circ B$ is:

$$A \circ B = \begin{pmatrix} 1 \cdot 5 & 2 \cdot 6 \\ 3 \cdot 7 & 4 \cdot 8 \end{pmatrix} = \begin{pmatrix} 5 & 12 \\ 21 & 32 \end{pmatrix}$$

Example 2: Hadamard Product with Larger Matrices

Given matrices:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}, \quad B = \begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix}$$

The Hadamard product $A \circ B$ is:

$$A \circ B = \begin{pmatrix} 1 \cdot 9 & 2 \cdot 8 & 3 \cdot 7 \\ 4 \cdot 6 & 5 \cdot 5 & 6 \cdot 4 \\ 7 \cdot 3 & 8 \cdot 2 & 9 \cdot 1 \end{pmatrix} = \begin{pmatrix} 9 & 16 & 21 \\ 24 & 25 & 24 \\ 21 & 16 & 9 \end{pmatrix}$$

In the following code chunks the computational process of Hadamard product is implemented in `Python`. Here both the from the scratch and use of external module versions are included.

1. Compute Hadamard Product from Scratch (without Libraries)

Here's how you can compute the Hadamard product manually:

```python
# Define matrices A and B
A = [[1, 2, 3], [4, 5, 6]]
B = [[7, 8, 9], [10, 11, 12]]

# Function to compute Hadamard product
def hadamard_product(A, B):
    # Get the number of rows and columns
    num_rows = len(A)
    num_cols = len(A[0])

    # Initialize the result matrix
    result = [[0]*num_cols for _ in range(num_rows)]

    # Compute the Hadamard product
    for i in range(num_rows):
        for j in range(num_cols):
            result[i][j] = A[i][j] * B[i][j]
```

```python
    return result

# Compute Hadamard product
hadamard_product_result = hadamard_product(A, B)

# Display result
print("Hadamard Product (From Scratch):")
for row in hadamard_product_result:
    print(row)
```

```
Hadamard Product (From Scratch):
[7, 16, 27]
[40, 55, 72]
```

2. Compute Hadamard Product Using `SymPy`

Here's how to compute the Hadamard product using `SymPy`:

```python
import sympy as sp

# Define matrices A and B
A = sp.Matrix([[1, 2, 3], [4, 5, 6]])
B = sp.Matrix([[7, 8, 9], [10, 11, 12]])

# Compute Hadamard product using SymPy
Hadamard_product_sympy = A.multiply_elementwise(B)

# Display result
print("Hadamard Product (Using SymPy):")
print(Hadamard_product_sympy)
```

```
Hadamard Product (Using SymPy):
Matrix([[7, 16, 27], [40, 55, 72]])
```

Practical Applications

Application 1: Image Masking

The Hadamard product can be used for image masking. Here's how you can apply a mask to an image and visualize it as shown in Fig 2.9.

```python
import matplotlib.pyplot as plt
import numpy as np

# Simulated large image (2D array) using NumPy
image = np.random.rand(100, 100)

# Simulated mask (binary matrix) using NumPy
mask = np.random.randint(0, 2, size=(100, 100))

# Compute Hadamard product
masked_image = image * mask

# Plot original image and masked image
fig, ax = plt.subplots(1, 2, figsize=(12, 5))
ax[0].imshow(image, cmap='gray')
ax[0].set_title('Original Image')
ax[1].imshow(masked_image, cmap='gray')
ax[1].set_title('Masked Image')
plt.show()
```

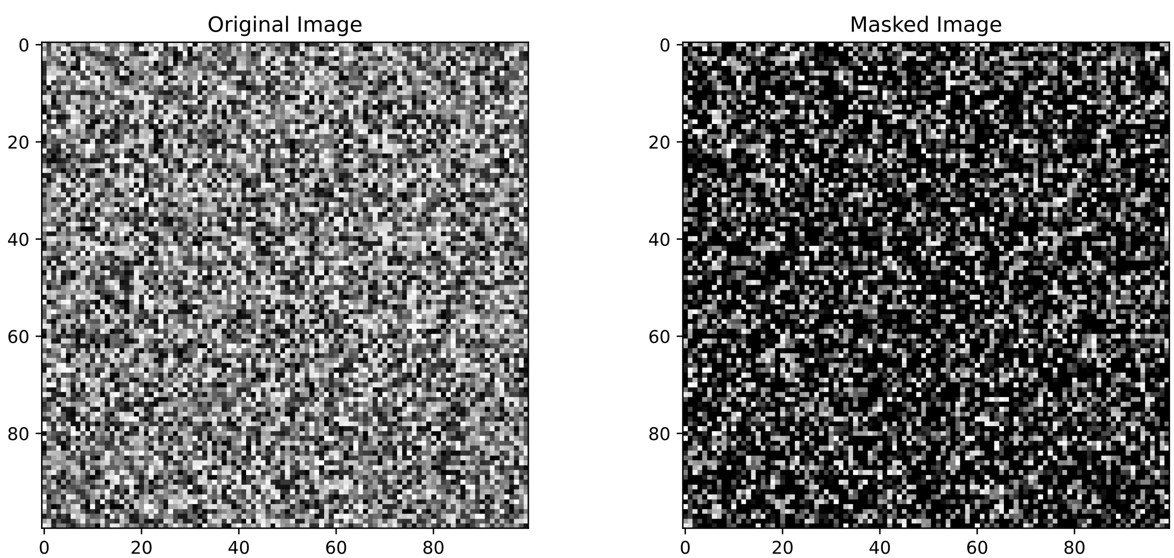


Figure 2.9: Demonstration of Masking in DIP using Hadamard Product

Application 2: Element-wise Scaling in Neural Networks

The Hadamard product can be used for dropout[1] in neural networks. A simple simulated example is given below.

```
# Simulated large activations (2D array) using NumPy
activations = np.random.rand(100, 100)

# Simulated dropout mask (binary matrix) using NumPy
dropout_mask = np.random.randint(0, 2, size=(100, 100))

# Apply dropout
dropped_activations = activations * dropout_mask

# Display results
print("Original Activations:")
print(activations)
print("\nDropout Mask:")
print(dropout_mask)
print("\nDropped Activations:")
print(dropped_activations)
```

```
Original Activations:
[[0.6531628  0.83828723 0.11495104 ... 0.57363979 0.77368058 0.88208445]
 [0.66763628 0.96987022 0.98753793 ... 0.16475706 0.63929072 0.33189477]
 [0.02473686 0.25289626 0.64734498 ... 0.13418643 0.69848738 0.98461407]
 ...
 [0.64511232 0.14902076 0.16164124 ... 0.44909864 0.58728166 0.96646049]
 [0.14945165 0.27439787 0.39108701 ... 0.41744138 0.31147021 0.81408627]
 [0.24841395 0.67896233 0.75853492 ... 0.25917307 0.64807412 0.29599092]]

Dropout Mask:
[[0 1 1 ... 0 1 0]
 [1 0 1 ... 1 0 1]
 [0 1 1 ... 1 0 0]
 ...
 [1 0 1 ... 1 1 0]
 [1 1 1 ... 0 0 1]
 [0 1 1 ... 1 0 0]]

Dropped Activations:
[[0.         0.83828723 0.11495104 ... 0.         0.77368058 0.        ]
```

[1] A regularization techniques in Deep learning. This approach deactivate some selected neurons to control model over-fitting

```
 [0.66763628 0.         0.98753793 ... 0.16475706 0.         0.33189477]
 [0.         0.25289626 0.64734498 ... 0.13418643 0.         0.        ]
 ...
 [0.64511232 0.         0.16164124 ... 0.44909864 0.58728166 0.        ]
 [0.14945165 0.27439787 0.39108701 ... 0.         0.         0.81408627]
 [0.         0.67896233 0.75853492 ... 0.25917307 0.         0.        ]]
```

Application 3: Statistical Data Analysis

In statistics, the Hadamard product can be applied to scale covariance matrices. Here's how we can compute the covariance matrix using matrix operations and apply scaling. Following `Python` code demonstrate this.

```python
import sympy as sp
import numpy as np

# Simulated large dataset (2D array) using NumPy
data = np.random.rand(100, 10)

# Compute the mean of each column
mean = np.mean(data, axis=0)

# Center the data
centered_data = data - mean

# Compute the covariance matrix using matrix product operation
cov_matrix = (centered_data.T @ centered_data) / (centered_data.shape[0] - 1)
cov_matrix_sympy = sp.Matrix(cov_matrix)

# Simulated scaling factors (2D array) using SymPy Matrix
scaling_factors = sp.Matrix(np.random.rand(10, 10))

# Compute Hadamard product
scaled_cov_matrix = cov_matrix_sympy.multiply(scaling_factors)

# Display results
print("Covariance Matrix:")
print(cov_matrix_sympy)
print("\nScaling Factors:")
print(scaling_factors)
print("\nScaled Covariance Matrix:")
print(scaled_cov_matrix)
```

```
Covariance Matrix:
Matrix([[0.0829059586487339, 0.0150029959722171, 0.00365684586530435, 0.00206780555712605, 0

Scaling Factors:
Matrix([[0.638439144815528, 0.520774228383380, 0.0363261442169694, 0.774607219606415, 0.35570

Scaled Covariance Matrix:
Matrix([[0.0451904772333656, 0.0466537971670436, -0.0104654305469453, 0.0685631538496497, 0.0
```

2.2.2.2 Practice Problems

Problem 1: Basic Hadamard Product

Given matrices:

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

$$B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

Find the Hadamard product $C = A \circ B$.

Solution:

$$C = \begin{bmatrix} 1 \cdot 5 & 2 \cdot 6 \\ 3 \cdot 7 & 4 \cdot 8 \end{bmatrix} = \begin{bmatrix} 5 & 12 \\ 21 & 32 \end{bmatrix}$$

Problem 2: Hadamard Product with Identity Matrix

Given matrices:

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

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

Find the Hadamard product $C = A \circ I$.

Solution:

$$C = \begin{bmatrix} 1 \cdot 1 & 2 \cdot 0 & 3 \cdot 0 \\ 4 \cdot 0 & 5 \cdot 1 & 6 \cdot 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 5 & 0 \end{bmatrix}$$

Problem 3: Hadamard Product with Zero Matrix

Given matrices:

$$A = \begin{bmatrix} 3 & 4 \\ 5 & 6 \end{bmatrix}$$

$$Z = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Find the Hadamard product $C = A \circ Z$.

Solution:

$$C = \begin{bmatrix} 3 \cdot 0 & 4 \cdot 0 \\ 5 \cdot 0 & 6 \cdot 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Problem 4: Hadamard Product of Two Identity Matrices

Given identity matrices:

$$I_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Find the Hadamard product $C = I_2 \circ I_3$ (extend I_2 to match dimensions of I_3).

Solution:

Extend I_2 to I_3:

$$I_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 \cdot 1 & 0 \cdot 0 & 0 \cdot 0 \\ 0 \cdot 0 & 1 \cdot 1 & 0 \cdot 0 \\ 0 \cdot 0 & 0 \cdot 0 & 0 \cdot 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Problem 5: Hadamard Product with Random Matrices

Given random matrices:

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

$$B = \begin{bmatrix} 0 & 5 \\ 6 & 2 \end{bmatrix}$$

Find the Hadamard product $C = A \circ B$.

Solution:

$$C = \begin{bmatrix} 2 \cdot 0 & 3 \cdot 5 \\ 1 \cdot 6 & 4 \cdot 2 \end{bmatrix} = \begin{bmatrix} 0 & 15 \\ 6 & 8 \end{bmatrix}$$

Problem 6: Hadamard Product of 3x3 Matrices

Given matrices:

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

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

Find the Hadamard product $C = A \circ B$.

Solution:

$$C = \begin{bmatrix} 1 \cdot 9 & 2 \cdot 8 & 3 \cdot 7 \\ 4 \cdot 6 & 5 \cdot 5 & 6 \cdot 4 \\ 7 \cdot 3 & 8 \cdot 2 & 9 \cdot 1 \end{bmatrix} = \begin{bmatrix} 9 & 16 & 21 \\ 24 & 25 & 24 \\ 21 & 16 & 9 \end{bmatrix}$$

Problem 7: Hadamard Product of Column Vectors

Given column vectors:

$$u = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

$$v = \begin{bmatrix} 5 \\ 6 \end{bmatrix}$$

Find the Hadamard product $w = u \circ v$.

Solution:

$$w = \begin{bmatrix} 2 \cdot 5 \\ 3 \cdot 6 \end{bmatrix} = \begin{bmatrix} 10 \\ 18 \end{bmatrix}$$

Problem 8: Hadamard Product with Non-Square Matrices

Given matrices:

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

$$B = \begin{bmatrix} 7 & 8 \\ 9 & 10 \end{bmatrix}$$

Find the Hadamard product $C = A \circ B$ (extend B to match dimensions of A).

Solution:

Extend B to match dimensions of A:

$$B = \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 7 & 8 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 \cdot 7 & 2 \cdot 8 \\ 3 \cdot 9 & 4 \cdot 10 \\ 5 \cdot 7 & 6 \cdot 8 \end{bmatrix} = \begin{bmatrix} 7 & 16 \\ 27 & 40 \\ 35 & 48 \end{bmatrix}$$

Problem 9: Hadamard Product in Image Processing

Given matrices representing image pixel values:

$$A = \begin{bmatrix} 10 & 20 \\ 30 & 40 \end{bmatrix}$$

$$B = \begin{bmatrix} 0.5 & 1.5 \\ 2.0 & 0.5 \end{bmatrix}$$

Find the Hadamard product $C = A \circ B$.

Solution:

$$C = \begin{bmatrix} 10 \cdot 0.5 & 20 \cdot 1.5 \\ 30 \cdot 2.0 & 40 \cdot 0.5 \end{bmatrix} = \begin{bmatrix} 5 & 30 \\ 60 & 20 \end{bmatrix}$$

Problem 10: Hadamard Product in Statistical Data

Given matrices representing two sets of statistical data:

$$A = \begin{bmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \end{bmatrix}$$

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

Find the Hadamard product $C = A \circ B$.

Solution:

$$C = \begin{bmatrix} 5 \cdot 1 & 6 \cdot 2 & 7 \cdot 3 \\ 8 \cdot 4 & 9 \cdot 5 & 10 \cdot 6 \end{bmatrix} = \begin{bmatrix} 5 & 12 & 21 \\ 32 & 45 & 60 \end{bmatrix}$$

2.2.2.3 Inner Product of Matrices

The inner product of two matrices is a generalized extension of the dot product, where each matrix is treated as a vector in a high-dimensional space. For two matrices A and B of the same dimension $m \times n$, the inner product is defined as the sum of the element-wise products of the matrices.

! Definition (Inner product)

For two matrices A and B of dimension $m \times n$, the inner product $\langle A, B \rangle$ is given by:

$$\langle A, B \rangle = \sum_{i=1}^{m} \sum_{j=1}^{n} A_{ij} \cdot B_{ij}$$

where $\cdot$ denotes element-wise multiplication.

! Properties

1. **Commutativity**:
$$\langle A, B \rangle = \langle B, A \rangle$$
2. **Linearity**:
$$\langle A + C, B \rangle = \langle A, B \rangle + \langle C, B \rangle$$
3. **Positive Definiteness**:
$$\langle A, A \rangle \geq 0$$
with equality if and only if A is a zero matrix.

Some simple examples showing the mathematical process of calculating the inner product is given bellow.

Example 1: Basic Inner Product

Given matrices:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \quad B = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$$

The inner product $\langle A, B \rangle$ is:

$$\langle A, B\rangle = 1\cdot 5 + 2\cdot 6 + 3\cdot 7 + 4\cdot 8 = 5 + 12 + 21 + 32 = 70$$

Example 2: Inner Product with Larger Matrices

Given matrices:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}, \quad B = \begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix}$$

The inner product $\langle A, B\rangle$ is calculated as:

$$\begin{aligned} \langle A, B\rangle &= 1\cdot 9 + 2\cdot 8 + 3\cdot 7 + 4\cdot 6 + 5\cdot 5 + 6\cdot 4 + 7\cdot 3 + 8\cdot 2 + 9\cdot 1 \\ &= 9 + 16 + 21 + 24 + 25 + 24 + 21 + 16 + 9 \\ &= 175 \end{aligned}$$

2.2.2.4 Practice Problems

Problem 1: Inner Product of 2x2 Matrices

Given matrices:

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

$$B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

Solution:

$$\begin{aligned} \langle A, B\rangle &= \sum_{i,j} A_{ij} B_{ij} \\ &= 1\cdot 5 + 2\cdot 6 + 3\cdot 7 + 4\cdot 8 \\ &= 5 + 12 + 21 + 32 \\ &= 70 \end{aligned}$$

Problem 2: Inner Product of 3x3 Matrices

Given matrices:

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

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

Solution:

$$\begin{aligned} \langle A, B \rangle &= \sum_{i,j} A_{ij} B_{ij} \\ &= 1 \cdot 8 + 0 \cdot 7 + 2 \cdot 6 + \\ &\quad 3 \cdot 5 + 4 \cdot 4 + 5 \cdot 3 + \\ &\quad 6 \cdot 2 + 7 \cdot 1 + 8 \cdot 0 \\ &= 8 + 0 + 12 + 15 + 16 + 15 + 12 + 7 + 0 \\ &= 85 \end{aligned}$$

Problem 3: Inner Product of Diagonal Matrices

Given diagonal matrices:

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

$$B = \begin{bmatrix} 5 & 0 & 0 \\ 0 & 6 & 0 \\ 0 & 0 & 7 \end{bmatrix}$$

Solution:

$$
\begin{aligned}
\langle A, B \rangle &= \sum_{i,j} A_{ij} B_{ij} \\
&= 2 \cdot 5 + 0 \cdot 0 + 0 \cdot 0 + \\
&\quad 0 \cdot 0 + 3 \cdot 6 + 0 \cdot 0 + \\
&\quad 0 \cdot 0 + 0 \cdot 0 + 4 \cdot 7 \\
&= 10 + 0 + 0 + 0 + 18 + 0 + 0 + 0 + 28 \\
&= 56
\end{aligned}
$$

Problem 4: Inner Product of Column Vectors

Given column vectors:

$$u = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

$$v = \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}$$

Solution:

$$
\begin{aligned}
\langle u, v \rangle &= \sum_{i} u_i v_i \\
&= 1 \cdot 4 + 2 \cdot 5 + 3 \cdot 6 \\
&= 4 + 10 + 18 \\
&= 32
\end{aligned}
$$

Problem 5: Inner Product with Random Matrices

Given matrices:

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

$$B = \begin{bmatrix} 5 & 7 \\ 8 & 6 \end{bmatrix}$$

Solution:

$$
\begin{aligned}
\langle A, B \rangle &= \sum_{i,j} A_{ij} B_{ij} \\
&= 3 \cdot 5 + 2 \cdot 7 + \\
&\quad 1 \cdot 8 + 4 \cdot 6 \\
&= 15 + 14 + 8 + 24 \\
&= 61
\end{aligned}
$$

Problem 6: Inner Product of 2x3 and 3x2 Matrices

Given matrices:

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

$$
B = \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix}
$$

Solution:

$$
\begin{aligned}
\langle A, B \rangle &= \sum_{i,j} A_{ij} B_{ij} \\
&= 1 \cdot 7 + 2 \cdot 8 + 3 \cdot 11 + \\
&\quad 4 \cdot 9 + 5 \cdot 10 + 6 \cdot 12 \\
&= 7 + 16 + 33 + 36 + 50 + 72 \\
&= 214
\end{aligned}
$$

Problem 7: Inner Product with Transpose Operation

Given matrices:

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

$$
B = \begin{bmatrix} 6 & 7 \\ 8 & 9 \end{bmatrix}
$$

Solution:

$$\begin{aligned}
\langle A, B\rangle &= \sum_{i,j} A_{ij}B_{ij} \\
&= 2\cdot 6 + 3\cdot 7 + \\
&\quad 4\cdot 8 + 5\cdot 9 \\
&= 12 + 21 + 32 + 45 \\
&= 110
\end{aligned}$$

Problem 8: Inner Product of Symmetric Matrices

Given symmetric matrices:

$$A = \begin{bmatrix} 1 & 2 \\ 2 & 3 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 5 \\ 5 & 6 \end{bmatrix}$$

Solution:

$$\begin{aligned}
\langle A, B\rangle &= \sum_{i,j} A_{ij}B_{ij} \\
&= 1\cdot 4 + 2\cdot 5 + \\
&\quad 2\cdot 5 + 3\cdot 6 \\
&= 4 + 10 + 10 + 18 \\
&= 42
\end{aligned}$$

Problem 9: Inner Product with Complex Matrices

Given matrices:

$$A = \begin{bmatrix} 1+i & 2-i \\ 3+i & 4-i \end{bmatrix}$$

$$B = \begin{bmatrix} 5-i & 6+i \\ 7-i & 8+i \end{bmatrix}$$

Solution:

$$
\begin{aligned}
\langle A, B\rangle &= \sum_{i,j} \mathrm{Re}(A_{ij}\overline{B_{ij}}) \\
&= (1+i)\cdot(5+i) + (2-i)\cdot(6-i) + \\
&\quad (3+i)\cdot(7+i) + (4-i)\cdot(8+i) \\
&= (5+i+5i-i^2) + (12-i-6i+i^2) + \\
&\quad (21+i+7i-i^2) + (32+i-8i-i^2) \\
&= 5+5+12-6+21+32-2 \\
&= 62
\end{aligned}
$$

Problem 10: Inner Product of 4x4 Matrices

Given matrices:

$$
A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}
$$

$$
B = \begin{bmatrix} 16 & 15 & 14 & 13 \\ 12 & 11 & 10 & 9 \\ 8 & 7 & 6 & 5 \\ 4 & 3 & 2 & 1 \end{bmatrix}
$$

Solution:

$$
\begin{aligned}
\langle A, B\rangle &= \sum_{i,j} A_{ij}B_{ij} \\
&= 1\cdot 16 + 2\cdot 15 + 3\cdot 14 + 4\cdot 13 + \\
&\quad 5\cdot 12 + 6\cdot 11 + 7\cdot 10 + 8\cdot 9 + \\
&\quad 9\cdot 8 + 10\cdot 7 + 11\cdot 6 + 12\cdot 5 + \\
&\quad 13\cdot 4 + 14\cdot 3 + 15\cdot 2 + 16\cdot 1 \\
&= 16+30+42+52+60+66+70+72+ \\
&\quad 72+70+66+60+52+42+30+16 \\
&= 696
\end{aligned}
$$

Now let's look into the computational part of *inner product*.

1. Compute Inner Product from Scratch (without Libraries)

Here's how you can compute the inner product from the scratch:

```
# Define matrices A and B
A = [[1, 2, 3], [4, 5, 6]]
B = [[7, 8, 9], [10, 11, 12]]

# Function to compute inner product
def inner_product(A, B):
    # Get the number of rows and columns
    num_rows = len(A)
    num_cols = len(A[0])

    # Initialize the result
    result = 0

    # Compute the inner product
    for i in range(num_rows):
        for j in range(num_cols):
            result += A[i][j] * B[i][j]

    return result

# Compute inner product
inner_product_result = inner_product(A, B)

# Display result
print("Inner Product (From Scratch):")
print(inner_product_result)
```

```
Inner Product (From Scratch):
217
```

2. Compute Inner Product Using `NumPy`

Here's how to compute the inner product using Numpy:

```
import numpy as np
# Define matrices A and B
A = np.array([[1, 2, 3], [4, 5, 6]])
B = np.array([[7, 8, 9], [10, 11, 12]])
# calculating innerproduct
inner_product = (A*B).sum() # calculate element-wise product, then column sum

print("Inner Product (Using numpy):")
print(inner_product)
```

```
Inner Product (Using numpy):
217
```

The same operation can be done using `SymPy` functions as follows.

```
import sympy as sp
import numpy as np
# Define matrices A and B
A = sp.Matrix([[1, 2, 3], [4, 5, 6]])
B = sp.Matrix([[7, 8, 9], [10, 11, 12]])

# Compute element-wise product
elementwise_product = A.multiply_elementwise(B)

# Calculate sum of each column
inner_product_sympy = np.sum(elementwise_product)

# Display result
print("Inner Product (Using SymPy):")
print(inner_product_sympy)
```

```
Inner Product (Using SymPy):
217
```

A vector dot product (in Physics) can be calculated using `SymPy .dot()` function as shown below.

Let $A = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$ and $B = \begin{pmatrix} 4 & 5 & 6 \end{pmatrix}$, then the dot product, $A \cdot B$ is computed as:

```python
import sympy as sp
A=sp.Matrix([1,2,3])
B=sp.Matrix([4,5,6])
display(A.dot(B)) # calculate fot product of A and B
```

32

> **A word of caution**
>
> In `SymPy` , `sp.Matrix([1,2,3])` create a column vector. But `np.array([1,2,3])` creates a row vector. So be careful while applying matrix/ dot product operations on these objects.

The same dot product using `numpy` object can be done as follows:

```python
import numpy as np
A=np.array([1,2,3])
B=np.array([4,5,6])
display(A.dot(B.T))# dot() stands for dot product B.T represents the
 ↪ transpose of B
```

```
np.int64(32)
```

Practical Applications

Application 1: Signal Processing

In signal processing, the inner product can be used to measure the similarity between two signals. Here the most popular measure of similarity is the `cosine` similarity. This measure is defined as:

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

Now consider two digital signals are given. It's cosine similarity measure can be calculated with a simulated data as shown below.

```python
import numpy as np

# Simulated large signals (1D array) using NumPy
signal1 = np.sin(np.random.rand(1000))
```

```
signal2 = np.cos(np.random.rand(1000))

# Compute inner product
inner_product_signal = np.dot(signal1, signal2)
#cosine_sim=np.dot(signal1,signal2)/(np.linalg.norm(signal1)*np.linalg.norm(
↪ signal2))
# Display result
cosine_sim=inner_product_signal/(np.sqrt(np.dot(signal1,signal1))*np.sqrt(np
↪ .dot(signal2,signal2)))
print("Inner Product (Using numpy):")
print(inner_product_signal)
print("Similarity of signals:")
print(cosine_sim)
```

```
Inner Product (Using numpy):
392.62000268841626
Similarity of signals:
0.8689213793892914
```

Application 2: Machine Learning - Feature Similarity

In machine learning, the inner product is used to calculate the similarity between feature vectors.

```
import numpy as np

# Simulated feature vectors (2D array) using NumPy
features1 = np.random.rand(100, 10)
features2 = np.random.rand(100, 10)

# Compute inner product for each feature vector
inner_products = np.einsum('ij,ij->i', features1, features2) # use Einstien's
↪ sum

# Display results
print("Inner Products of Feature Vectors:")
display(inner_products)
```

```
Inner Products of Feature Vectors:
```

```
array([2.48886033, 3.43945648, 2.38530585, 2.08518548, 3.40538291,
       3.08881997, 2.54774385, 2.43295258, 2.50780792, 1.8560207 ,
       3.06411358, 2.26048811, 2.03134131, 1.81299756, 1.14507113,
       2.94901178, 2.75576454, 2.74002185, 1.8272226 , 2.58531693,
       1.19590323, 2.21817239, 1.89038745, 2.54850376, 3.91705499,
       3.46785415, 2.51064793, 2.02607531, 2.9717878 , 1.79783316,
       1.94793807, 2.36596347, 2.79641365, 2.73322973, 2.35513161,
       1.96174937, 2.40119398, 2.33250281, 2.49837413, 3.69676824,
       2.7461938 , 2.95986562, 3.55920368, 4.03507279, 2.88940507,
       2.5993641 , 2.76814909, 3.10670645, 2.2457387 , 2.39073579,
       1.8101891 , 2.52428113, 1.81653064, 2.09484409, 1.25551096,
       2.32619789, 1.67463344, 3.42870115, 2.24114176, 2.59109337,
       2.15239433, 2.36326495, 2.58930612, 3.56443309, 1.9019121 ,
       1.74618531, 2.11408339, 2.72646816, 2.50110499, 2.08643111,
       2.42503712, 2.16812433, 1.80379391, 1.82089971, 3.42084513,
       2.59491103, 1.89301808, 3.09804981, 1.84899366, 2.40860833,
       2.34468849, 3.19607324, 1.9487296 , 3.54428811, 2.58308956,
       2.54039416, 1.86061794, 1.82552294, 2.83780514, 3.46611961,
       2.63983103, 2.19804622, 2.15164376, 1.86292563, 2.45612335,
       2.57643077, 2.49426072, 4.73403731, 2.83851871, 4.06494811])
```

Application 3: Covariance Matrix in Statistics

The inner product can be used to compute covariance matrices for statistical data analysis. If X is a given distribution and $x = X - \bar{X}$. Then the covariance of X can be calculated as $cov(X) = \frac{1}{n-1}(x \cdot x^T)$ [2]. The python code a simulated data is shown below.

```python
import sympy as sp
import numpy as np

# Simulated large dataset (2D array) using NumPy
data = np.random.rand(100, 10)

# Compute the mean of each column
mean = np.mean(data, axis=0)

# Center the data
centered_data = data - mean

# Compute the covariance matrix using matrix product operation
```

[2]Remember that the covariance of X is defined as $Cov(X) = \frac{\sum(X - \bar{X})^2}{n-1}$

```python
cov_matrix = (centered_data.T @ centered_data) / (centered_data.shape[0] - 1)
cov_matrix_sympy = sp.Matrix(cov_matrix)

# Display results
print("Covariance Matrix:")
display(cov_matrix_sympy)
```

```
Covariance Matrix:
```

$$\left[\begin{matrix} 0.0702190471848429 & 0.00394267598912726 & -0.0241177850497233 & -0.0021984094106544 & 0.00984 \\ 0.00394267598912726 & 0.0762839838945567 & 0.00565134755565279 & 0.0200714303162392 & 0.000325 \\ -0.0241177850497233 & 0.00565134755565279 & 0.0922149807786352 & -0.00190862124342186 & -0.00662 \\ -0.0021984094106544 & 0.0200714303162392 & -0.00190862124342186 & 0.0892403994436236 & 0.00309 \\ 0.00984141838732026 & 0.000325839400694629 & -0.00662646939412883 & 0.00309944925332746 & 0.0613 \\ 0.00163614928095014 & -0.00481724643200719 & -0.0078070601824087 & 0.00377520705008828 & -0.0083 \\ 0.0129289228170729 & -0.00198619124779562 & 0.00581793052349881 & -0.000290004766692818 & -0.0013 \\ 0.00192249653907526 & -0.00476347369990653 & -0.00933666364132328 & 0.0172667002235969 & 0.0133 \\ 0.0058037097347326 & -0.00497663740176259 & -0.00873685961082437 & -0.00541190249590404 & 0.00398 \\ 0.014126309216061 & -0.00107787427749236 & -0.00988374728069452 & -0.00202505745225932 & -0.0011 \end{matrix}\right.$$

These examples demonstrate the use of inner product and dot product in various applications.

2.2.2.5 Outer Product

The outer product of two vectors results in a matrix, and it is a way to combine these vectors into a higher-dimensional representation.

> **Definition (Outer Product)**
>
> For two vectors $\mathbf{u}$ and $\mathbf{v}$ of dimensions m and n respectively, the outer product $\mathbf{u} \otimes \mathbf{v}$ is an $m \times n$ matrix defined as:
>
> $$(\mathbf{u} \otimes \mathbf{v})_{ij} = u_i \cdot v_j$$
>
> where $\cdot$ denotes the outer product operation. In matrix notation, for two column vectors u, v,
>
> $$u \otimes v = uv^T$$

Properties

1. **Linearity**:
$$(\mathbf{u} + \mathbf{w}) \otimes \mathbf{v} = (\mathbf{u} \otimes \mathbf{v}) + (\mathbf{w} \otimes \mathbf{v})$$
2. **Distributivity**:
$$\mathbf{u} \otimes (\mathbf{v} + \mathbf{w}) = (\mathbf{u} \otimes \mathbf{v}) + (\mathbf{u} \otimes \mathbf{w})$$
3. **Associativity**:
$$(\mathbf{u} \otimes \mathbf{v}) \otimes \mathbf{w} = \mathbf{u} \otimes (\mathbf{v} \otimes \mathbf{w})$$

Some simple examples of outer product is given below.

Example 1: Basic Outer Product

Given vectors:

$$\mathbf{u} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \quad \mathbf{v} = \begin{pmatrix} 3 \\ 4 \\ 5 \end{pmatrix}$$

The outer product $\mathbf{u} \otimes \mathbf{v}$ is:

$$\mathbf{u} \otimes \mathbf{v} = \begin{pmatrix} 1 \cdot 3 & 1 \cdot 4 & 1 \cdot 5 \\ 2 \cdot 3 & 2 \cdot 4 & 2 \cdot 5 \end{pmatrix} = \begin{pmatrix} 3 & 4 & 5 \\ 6 & 8 & 10 \end{pmatrix}$$

Example 2: Outer Product with Larger Vectors

Given vectors:
$$\mathbf{u} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, \quad \mathbf{v} = \begin{pmatrix} 4 \\ 5 \end{pmatrix}$$

The outer product $\mathbf{u} \otimes \mathbf{v}$ is:

$$\mathbf{u} \otimes \mathbf{v} = \begin{pmatrix} 1 \cdot 4 & 1 \cdot 5 \\ 2 \cdot 4 & 2 \cdot 5 \\ 3 \cdot 4 & 3 \cdot 5 \end{pmatrix} = \begin{pmatrix} 4 & 5 \\ 8 & 10 \\ 12 & 15 \end{pmatrix}$$

2.2.2.6 Practice Problems

Find the outer product of A and B where A and B are given as follows:

Problem 1:

Find the outer product of:

$$A = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

$$B = \begin{bmatrix} 3 & 4 \end{bmatrix}$$

Solution:

$$\begin{aligned} A \otimes B &= \begin{bmatrix} 1 \\ 2 \end{bmatrix} \otimes \begin{bmatrix} 3 & 4 \end{bmatrix} \\ &= \begin{bmatrix} 1 \cdot 3 & 1 \cdot 4 \\ 2 \cdot 3 & 2 \cdot 4 \end{bmatrix} \\ &= \begin{bmatrix} 3 & 4 \\ 6 & 8 \end{bmatrix} \end{aligned}$$

Problem 2:

Find the outer product of:

$$A = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 5 & 6 \end{bmatrix}$$

Solution:

$$\begin{aligned} A \otimes B &= \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \otimes \begin{bmatrix} 4 & 5 & 6 \end{bmatrix} \\ &= \begin{bmatrix} 1 \cdot 4 & 1 \cdot 5 & 1 \cdot 6 \\ 2 \cdot 4 & 2 \cdot 5 & 2 \cdot 6 \\ 3 \cdot 4 & 3 \cdot 5 & 3 \cdot 6 \end{bmatrix} \\ &= \begin{bmatrix} 4 & 5 & 6 \\ 8 & 10 & 12 \\ 12 & 15 & 18 \end{bmatrix} \end{aligned}$$

Problem 3:

Find the outer product of:

$$A = \begin{bmatrix} 1 & 2 \end{bmatrix}$$

$$B = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$$

Solution:

$$\begin{aligned} A \otimes B &= \begin{bmatrix} 1 & 2 \end{bmatrix} \otimes \begin{bmatrix} 3 \\ 4 \end{bmatrix} \\ &= \begin{bmatrix} 1 \cdot 3 & 1 \cdot 4 \\ 2 \cdot 3 & 2 \cdot 4 \end{bmatrix} \\ &= \begin{bmatrix} 3 & 4 \\ 6 & 8 \end{bmatrix} \end{aligned}$$

Problem 4:

Find the outer product of:

$$A = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & -1 \end{bmatrix}$$

Solution:

$$\begin{aligned} A \otimes B &= \begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & -1 \end{bmatrix} \\ &= \begin{bmatrix} 0 \cdot 1 & 0 \cdot -1 \\ 1 \cdot 1 & 1 \cdot -1 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 0 \\ 1 & -1 \end{bmatrix} \end{aligned}$$

Problem 5:

Find the outer product of:

$$A = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

$$B = \begin{bmatrix} 5 & -2 \end{bmatrix}$$

Solution:

$$\begin{aligned} A \otimes B &= \begin{bmatrix} 2 \\ 3 \end{bmatrix} \otimes \begin{bmatrix} 5 & -2 \end{bmatrix} \\ &= \begin{bmatrix} 2 \cdot 5 & 2 \cdot -2 \\ 3 \cdot 5 & 3 \cdot -2 \end{bmatrix} \\ &= \begin{bmatrix} 10 & -4 \\ 15 & -6 \end{bmatrix} \end{aligned}$$

Problem 6:

Find the outer product of:

$$A = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

$$B = \begin{bmatrix} 2 & -1 & 0 \end{bmatrix}$$

Solution:

$$\begin{aligned} A \otimes B &= \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 2 & -1 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 1 \cdot 2 & 1 \cdot -1 & 1 \cdot 0 \\ 0 \cdot 2 & 0 \cdot -1 & 0 \cdot 0 \\ 1 \cdot 2 & 1 \cdot -1 & 1 \cdot 0 \end{bmatrix} \\ &= \begin{bmatrix} 2 & -1 & 0 \\ 0 & 0 & 0 \\ 2 & -1 & 0 \end{bmatrix} \end{aligned}$$

Problem 7:

Find the outer product of:

$$A = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

$$B = \begin{bmatrix} 2 & 0 \\ 3 & -1 \end{bmatrix}$$

Solution:

$$\begin{aligned} A \otimes B &= \begin{bmatrix} 1 \\ -1 \end{bmatrix} \otimes \begin{bmatrix} 2 & 0 \\ 3 & -1 \end{bmatrix} \\ &= \begin{bmatrix} 2 & 3 & 0 & -1 \\ -2 & -3 & 0 & 1 \end{bmatrix} \end{aligned}$$

Problem 8:

Find the outer product of:

$$A = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & -2 & 3 \end{bmatrix}$$

Solution:

$$\begin{aligned} A \otimes B &= \begin{bmatrix} 3 \\ 4 \end{bmatrix} \otimes \begin{bmatrix} 1 & -2 & 3 \end{bmatrix} \\ &= \begin{bmatrix} 3 \cdot 1 & 3 \cdot -2 & 3 \cdot 3 \\ 4 \cdot 1 & 4 \cdot -2 & 4 \cdot 3 \end{bmatrix} \\ &= \begin{bmatrix} 3 & -6 & 9 \\ 4 & -8 & 12 \end{bmatrix} \end{aligned}$$

Problem 9:

Find the outer product of:

$$A = \begin{bmatrix} 2 \\ 3 \\ -1 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & -2 \end{bmatrix}$$

Solution:

$$\begin{aligned} A \otimes B &= \begin{bmatrix} 2 \\ 3 \\ -1 \end{bmatrix} \otimes \begin{bmatrix} 4 & -2 \end{bmatrix} \\ &= \begin{bmatrix} 2 \cdot 4 & 2 \cdot -2 \\ 3 \cdot 4 & 3 \cdot -2 \\ -1 \cdot 4 & -1 \cdot -2 \end{bmatrix} \\ &= \begin{bmatrix} 8 & -4 \\ 12 & -6 \\ -4 & 2 \end{bmatrix} \end{aligned}$$

Problem 10:

Find the outer product of:

$$A = \begin{bmatrix} 0 \\ 5 \end{bmatrix}$$

$$B = \begin{bmatrix} 3 & 1 \end{bmatrix}$$

Solution:

$$\begin{aligned} A \otimes B &= \begin{bmatrix} 0 \\ 5 \end{bmatrix} \otimes \begin{bmatrix} 3 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 0 \cdot 3 & 0 \cdot 1 \\ 5 \cdot 3 & 5 \cdot 1 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 0 \\ 15 & 5 \end{bmatrix} \end{aligned}$$

---

1. Compute Outer Product of Vectors from Scratch (without Libraries)

Here's how you can compute the outer product manually:

```
# Define vectors u and v
u = [1, 2]
v = [3, 4, 5]

# Function to compute outer product
def outer_product(u, v):
    # Initialize the result
    result = [[a * b for b in v] for a in u]
    return result

# Compute outer product
outer_product_result = outer_product(u, v)

# Display result
print("Outer Product of Vectors (From Scratch):")
for row in outer_product_result:
    print(row)
```

```
Outer Product of Vectors (From Scratch):
[3, 4, 5]
[6, 8, 10]
```

2. Compute Outer Product of Vectors Using SymPy

Here's how to compute the outer product using `SymPy`:

```
import sympy as sp

# Define vectors u and v
u = sp.Matrix([1, 2])
v = sp.Matrix([3, 4, 5])

# Compute outer product using SymPy
outer_product_sympy = u * v.T

# Display result
print("Outer Product of Vectors (Using SymPy):")
display(outer_product_sympy)
```

`Outer Product of Vectors (Using SymPy):`

$$\begin{bmatrix} 3 & 4 & 5 \\ 6 & 8 & 10 \end{bmatrix}$$

Outer Product of Matrices

The outer product of two matrices extends the concept from vectors to higher-dimensional tensors. For two matrices A and B, the outer product results in a higher-dimensional tensor and is generally expressed as block matrices.

> **Definition (Outer Product of Matrices)**
>
> For two matrices A of dimension $m \times p$ and B of dimension $q \times n$, the outer product $A \otimes B$ results in a tensor of dimension $m \times q \times p \times n$. The entries of the tensor are given by:
>
> $$(A \otimes B)_{ijkl} = A_{ij} \cdot B_{kl}$$
>
> where $\cdot$ denotes the outer product operation.

> **Properties**
>
> 1. **Linearity**:
> $$(A + C) \otimes B = (A \otimes B) + (C \otimes B)$$
> 2. **Distributivity**:
> $$A \otimes (B + D) = (A \otimes B) + (A \otimes D)$$
> 3. **Associativity**:
> $$(A \otimes B) \otimes C = A \otimes (B \otimes C)$$

Here are some simple examples to demonstrate the mathematical procedure to find outer product of matrices.

Example 1: Basic Outer Product of Matrices

Given matrices:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \quad B = \begin{pmatrix} 5 \\ 6 \end{pmatrix}$$

The outer product $A \otimes B$ is:

$$A \otimes B = \begin{pmatrix} 1 \cdot 5 & 1 \cdot 6 \\ 2 \cdot 5 & 2 \cdot 6 \\ 3 \cdot 5 & 3 \cdot 6 \\ 4 \cdot 5 & 4 \cdot 6 \end{pmatrix} = \begin{pmatrix} 5 & 6 \\ 10 & 12 \\ 15 & 18 \\ 20 & 24 \end{pmatrix}$$

Example 2: Outer Product with Larger Matrices

Given matrices:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}, \quad B = \begin{pmatrix} 7 \\ 8 \end{pmatrix}$$

The outer product $A \otimes B$ is:

$$A \otimes B = \begin{pmatrix} 1 \cdot 7 & 1 \cdot 8 \\ 2 \cdot 7 & 2 \cdot 8 \\ 3 \cdot 7 & 3 \cdot 8 \\ 4 \cdot 7 & 4 \cdot 8 \\ 5 \cdot 7 & 5 \cdot 8 \\ 6 \cdot 7 & 6 \cdot 8 \end{pmatrix} = \begin{pmatrix} 7 & 8 \\ 14 & 16 \\ 21 & 24 \\ 28 & 32 \\ 35 & 40 \\ 42 & 48 \end{pmatrix}$$

Example 3: Compute the outer product of the following vectors $\mathbf{u} = [0, 1, 2]$ and $\mathbf{v} = [2, 3, 4]$.

To find the outer product, we calculate each element (i, j) as the product of the (i)-th element of $\mathbf{u}$ and the (j)-th element of $\mathbf{v}$. Mathematically:

$$\mathbf{u} \otimes \mathbf{v} = \begin{bmatrix} 0 \cdot 2 & 0 \cdot 3 & 0 \cdot 4 \\ 1 \cdot 2 & 1 \cdot 3 & 1 \cdot 4 \\ 2 \cdot 2 & 2 \cdot 3 & 2 \cdot 4 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 2 & 3 & 4 \\ 4 & 6 & 8 \end{bmatrix}$$

1. Compute Outer Product of Matrices from Scratch (without Libraries)

Here's how you can compute the outer product manually:

```
# Define matrices A and B
A = [[1, 2], [3, 4]]
B = [[5], [6]]

# Function to compute outer product
def outer_product_matrices(A, B):
    m = len(A)
    p = len(A[0])
```

```python
    q = len(B)
    n = len(B[0])
    result = [[0] * (n * p) for _ in range(m * q)]

    for i in range(m):
        for j in range(p):
            for k in range(q):
                for l in range(n):
                    result[i*q + k][j*n + l] = A[i][j] * B[k][l]

    return result

# Compute outer product
outer_product_result_matrices = outer_product_matrices(A, B)

# Display result
print("Outer Product of Matrices (From Scratch):")
for row in outer_product_result_matrices:
    print(row)
```

```
Outer Product of Matrices (From Scratch):
[5, 10]
[6, 12]
[15, 20]
[18, 24]
```

Here is the `Python` code to compute the outer product of these vectors using the `NumPy` function `.outer()`:

```python
import numpy as np

# Define vectors
u = np.array([[1,2],[3,4]])
v = np.array([[5],[4]])

# Compute outer product
outer_product = np.outer(u, v)

print("Outer Product of u and v:")
display(outer_product)
```

```
Outer Product of u and v:

array([[ 5,  4],
       [10,  8],
       [15, 12],
       [20, 16]])
```

Example 3: Real-world Application in Recommendation Systems

In recommendation systems, the outer product can represent user-item interactions. A simple context is here. Let the user preferences of items is given as $u = [4, 3, 5]$ and the item scores is given by $v = [2, 5, 4]$. Now the recommendation score can be calculated as the outer product of these two vectors. Calculation of this score is shown below. The outer product $\mathbf{u} \otimes \mathbf{v}$ is calculated as follows:

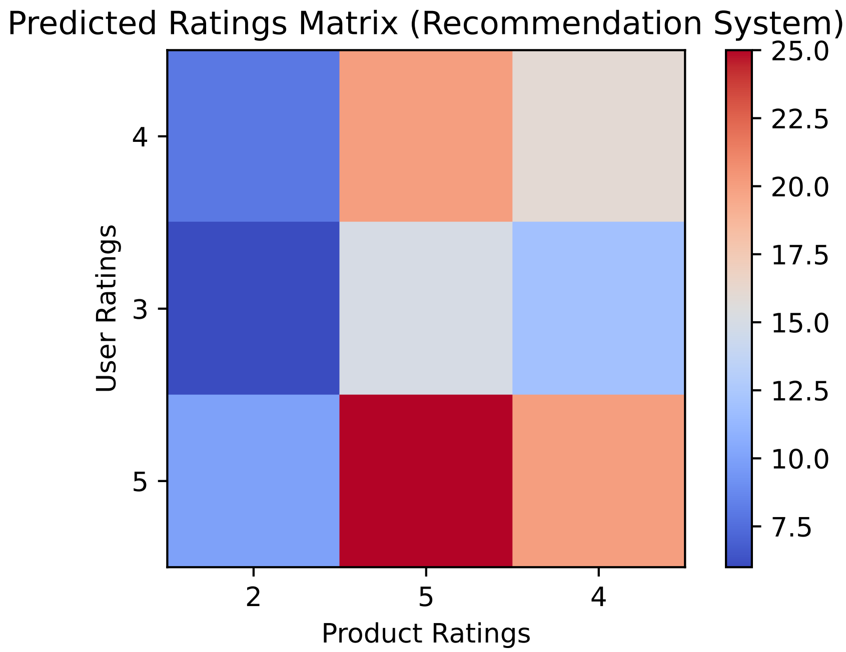
$$\mathbf{u} \otimes \mathbf{v} = \begin{bmatrix} 4 \cdot 2 & 4 \cdot 5 & 4 \cdot 4 \\ 3 \cdot 2 & 3 \cdot 5 & 3 \cdot 4 \\ 5 \cdot 2 & 5 \cdot 5 & 5 \cdot 4 \end{bmatrix} = \begin{bmatrix} 8 & 20 & 16 \\ 6 & 15 & 12 \\ 10 & 25 & 20 \end{bmatrix}$$

The python code for this task is given below.

```python
import numpy as np
import matplotlib.pyplot as plt

# Define the user and product ratings vectors
user_ratings = np.array([4, 3, 5])
product_ratings = np.array([2, 5, 4])

# Compute the outer product
predicted_ratings = np.outer(user_ratings, product_ratings)

# Print the predicted ratings matrix
print("Predicted Ratings Matrix:")
display(predicted_ratings)

# Plot the result
plt.imshow(predicted_ratings, cmap='coolwarm', interpolation='nearest')
plt.colorbar()
plt.title('Predicted Ratings Matrix (Recommendation System)')
plt.xlabel('Product Ratings')
plt.ylabel('User Ratings')
plt.xticks(ticks=np.arange(len(product_ratings)), labels=product_ratings)
```

```
plt.yticks(ticks=np.arange(len(user_ratings)), labels=user_ratings)
plt.show()
```

```
Predicted Ratings Matrix:

array([[ 8, 20, 16],
       [ 6, 15, 12],
       [10, 25, 20]])
```



Additional Properties & Definitions

1. **Definition and Properties**

   Given two vectors:

   - $\mathbf{u} \in \mathbb{R}^m$
   - $\mathbf{v} \in \mathbb{R}^n$

   The outer product $\mathbf{u} \otimes \mathbf{v}$ results in an $m \times n$ matrix where each element (i, j) of the matrix is calculated as:

   $$(\mathbf{u} \otimes \mathbf{v})_{ij} = u_i \cdot v_j$$

2. **Non-Symmetry**

The outer product is generally not symmetric. For vectors $\mathbf{u}$ and $\mathbf{v}$, the matrix $\mathbf{u} \otimes \mathbf{v}$ is not necessarily equal to $\mathbf{v} \otimes \mathbf{u}$:

$$\mathbf{u} \otimes \mathbf{v} \neq \mathbf{v} \otimes \mathbf{u}$$

3. **Rank of the Outer Product**

   The rank of the outer product matrix $\mathbf{u} \otimes \mathbf{v}$ is always 1, provided neither $\mathbf{u}$ nor $\mathbf{v}$ is a zero vector. This is because the matrix can be expressed as a single rank-1 matrix.

4. **Distributive Property**

   The outer product is distributive over vector addition. For vectors $\mathbf{u}_1, \mathbf{u}_2 \in \mathbb{R}^m$ and $\mathbf{v} \in \mathbb{R}^n$:

$$(\mathbf{u}_1 + \mathbf{u}_2) \otimes \mathbf{v} = (\mathbf{u}_1 \otimes \mathbf{v}) + (\mathbf{u}_2 \otimes \mathbf{v})$$

5. **Associativity with Scalar Multiplication**

   The outer product is associative with scalar multiplication. For a scalar α and vectors $\mathbf{u} \in \mathbb{R}^m$ and $\mathbf{v} \in \mathbb{R}^n$:

$$\alpha(\mathbf{u} \otimes \mathbf{v}) = (\alpha\mathbf{u}) \otimes \mathbf{v} = \mathbf{u} \otimes (\alpha\mathbf{v})$$

6. **Matrix Trace**

   The trace of the outer product of two vectors is given by:

$$\mathrm{tr}(\mathbf{u} \otimes \mathbf{v}) = (\mathbf{u}^T\mathbf{v}) = (\mathbf{v}^T\mathbf{u})$$

   Here, tr denotes the trace of a matrix, which is the sum of its diagonal elements.

7. **Matrix Norm**

   The Frobenius norm of the outer product matrix can be expressed in terms of the norms of the original vectors:

$$\|\mathbf{u} \otimes \mathbf{v}\|_F = \|\mathbf{u}\|_2 \cdot \|\mathbf{v}\|_2$$

   where $\|\cdot\|_2$ denotes the Euclidean norm.

Example Calculation in `Python`

Here's how to compute and visualize the outer product properties using `Python`:

```python
import numpy as np
import matplotlib.pyplot as plt
```

```python
# Define vectors
u = np.array([1, 2, 3])
v = np.array([4, 5])

# Compute outer product
outer_product = np.outer(u, v)

# Display results
print("Outer Product Matrix:")
print(outer_product)

# Compute and display rank
rank = np.linalg.matrix_rank(outer_product)
print(f"Rank of Outer Product Matrix: {rank}")

# Compute Frobenius norm
frobenius_norm = np.linalg.norm(outer_product, 'fro')
print(f"Frobenius Norm: {frobenius_norm}")

# Plot the result
plt.imshow(outer_product, cmap='viridis', interpolation='nearest')
plt.colorbar()
plt.title('Outer Product Matrix')
plt.xlabel('Vector v')
plt.ylabel('Vector u')
plt.xticks(ticks=np.arange(len(v)), labels=v)
plt.yticks(ticks=np.arange(len(u)), labels=u)
plt.show()
```

```
Outer Product Matrix:
[[ 4  5]
 [ 8 10]
 [12 15]]
Rank of Outer Product Matrix: 1
Frobenius Norm: 23.958297101421877
```

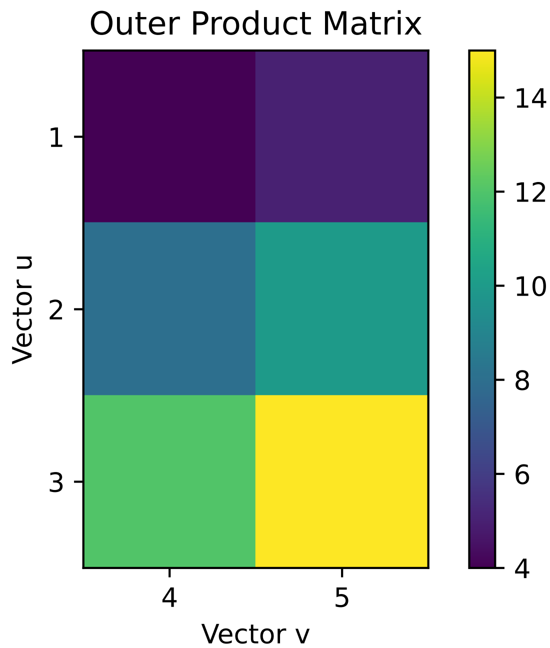


Figure 2.10: Demonstration of Outer Product and its Properties

2.2.2.7 Kronecker Product

In mathematics, the Kronecker product, sometimes denoted by $\otimes$, is an operation on two matrices of arbitrary size resulting in a *block matrix*. It is a specialization of the tensor product (which is denoted by the same symbol) from vectors to matrices and gives the matrix of the tensor product linear map with respect to a standard choice of basis. The Kronecker product is to be distinguished from the usual matrix multiplication, which is an entirely different operation. The Kronecker product is also sometimes called *matrix direct product*.

> **Note**
>
> If A is an $m \times n$ matrix and B is a $p \times q$ matrix, then the Kronecker product $A \otimes B$ is the $pm \times qn$ block matrix defined as: Each a_{ij} of A is replaced by the matrix $a_{ij}B$. Symbolically this will result in a block matrix defined by:
>
> $$A \otimes B = \left[A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & \cdots & a_{1n}B \\ a_{21}B & a_{22}B & \cdots & a_{2n}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}B & a_{m2}B & \cdots & a_{mn}B \end{bmatrix} \right]$$

Properties of the Kronecker Product

1. **Associativity**

   The Kronecker product is associative. For matrices $A \in \mathbb{R}^{m\times n}$, $B \in \mathbb{R}^{p\times q}$, and $C \in \mathbb{R}^{r\times s}$:

   $$(A \otimes B) \otimes C = A \otimes (B \otimes C)$$

2. **Distributivity Over Addition**

   The Kronecker product distributes over matrix addition. For matrices $A \in \mathbb{R}^{m\times n}$, $B \in \mathbb{R}^{p\times q}$, and $C \in \mathbb{R}^{p\times q}$:

   $$A \otimes (B + C) = (A \otimes B) + (A \otimes C)$$

3. **Mixed Product Property**

   The Kronecker product satisfies the mixed product property with the matrix product. For matrices $A \in \mathbb{R}^{m\times n}$, $B \in \mathbb{R}^{p\times q}$, $C \in \mathbb{R}^{r\times s}$, and $D \in \mathbb{R}^{r\times s}$:

   $$(A \otimes B)(C \otimes D) = (AC) \otimes (BD)$$

4. **Transpose**

   The transpose of the Kronecker product is given by:

   $$(A \otimes B)^T = A^T \otimes B^T$$

5. **Norm**

   The Frobenius norm of the Kronecker product can be computed as:

   $$\|A \otimes B\|_F = \|A\|_F \cdot \|B\|_F$$

   where $\|\cdot\|_F$ denotes the Frobenius norm.

Frobenius Norm

The Frobenius norm, also known as the Euclidean norm for matrices, is a measure of a matrix's magnitude. It is defined as the square root of the sum of the absolute squares of its elements. Mathematically, for a matrix A with elements a_{ij}, the Frobenius norm is given by:

$$\|A\|_F = \sqrt{\sum_{i,j} |a_{ij}|^2}$$

Example 1: Calculation of Frobenius Norm

Consider the matrix A:

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

To compute the Frobenius norm:

$$\|A\|_F = \sqrt{1^2 + 2^2 + 3^2 + 4^2} = \sqrt{1 + 4 + 9 + 16} = \sqrt{30} \approx 5.48$$

Example 2: Frobenius Norm of a Sparse Matrix

Consider the sparse matrix B:

$$B = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

To compute the Frobenius norm:

$$\|B\|_F = \sqrt{0^2 + 0^2 + 0^2 + 5^2 + 0^2 + 0^2} = \sqrt{25} = 5$$

Example 3: Frobenius Norm in a Large Matrix

Consider the matrix C of size \$3 $\times$3 \$:

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

To compute the Frobenius norm:

$$\begin{aligned} \|C\|_F &= \sqrt{1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 6^2 + 7^2 + 8^2 + 9^2} \\ &= \sqrt{1 + 4 + 9 + 16 + 25 + 36 + 49 + 64 + 81} & &= \sqrt{285} \approx 16.88 \end{aligned}$$

Applications of the Frobenius Norm

- *Application 1: Image Compression:* In image processing, the Frobenius norm can measure the difference between the original and compressed images, indicating how well the compression has preserved the original image quality.
- *Application 2: Matrix Factorization:* In numerical analysis, Frobenius norm is used to evaluate the error in matrix approximations, such as in Singular Value Decomposition (SVD). A lower Frobenius norm of the error indicates a better approximation.
- *Application 3: Error Measurement in Numerical Solutions:* In solving systems of linear equations, the Frobenius norm can be used to measure the error between the true solution and the computed solution, providing insight into the accuracy of numerical methods.

The `linalg` sub module of `NumPy` library can be used to calculate various norms. Basically norm is the generalized form of Euclidean distance.

```
import numpy as np

# Example 1: Simple Matrix
A = np.array([[1, 2], [3, 4]])
frobenius_norm_A = np.linalg.norm(A, 'fro')
print(f"Frobenius Norm of A: {frobenius_norm_A:.2f}")

# Example 2: Sparse Matrix
B = np.array([[0, 0, 0], [0, 5, 0], [0, 0, 0]])
frobenius_norm_B = np.linalg.norm(B, 'fro')
print(f"Frobenius Norm of B: {frobenius_norm_B:.2f}")

# Example 3: Large Matrix
C = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
frobenius_norm_C = np.linalg.norm(C, 'fro')
print(f"Frobenius Norm of C: {frobenius_norm_C:.2f}")
```

```
Frobenius Norm of A: 5.48
Frobenius Norm of B: 5.00
Frobenius Norm of C: 16.88
```

Frobenius norm of Kronecker product

Let us consider two matrices,

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

and

$$B = \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix}$$

The Kronecker product $C = A \otimes B$ is:

$$C = \begin{bmatrix} 1 \cdot B & 2 \cdot B \\ 3 \cdot B & 4 \cdot B \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} & \begin{bmatrix} 0 \cdot 2 & 5 \cdot 2 \\ 6 \cdot 2 & 7 \cdot 2 \end{bmatrix} \\ \begin{bmatrix} 0 \cdot 3 & 5 \cdot 3 \\ 6 \cdot 3 & 7 \cdot 3 \end{bmatrix} & \begin{bmatrix} 0 \cdot 4 & 5 \cdot 4 \\ 6 \cdot 4 & 7 \cdot 4 \end{bmatrix} \end{bmatrix}$$

This expands to:

$$C = \begin{bmatrix} 0 & 5 & 0 & 10 \\ 6 & 7 & 12 & 14 \\ 0 & 15 & 0 & 20 \\ 18 & 21 & 24 & 28 \end{bmatrix}$$

Computing the Frobenius Norm

To compute the Frobenius norm of C:

$$\|C\|_F = \sqrt{\sum_{i=1}^{4} \sum_{j=1}^{4} |c_{ij}|^2}$$

$$\|C\|_F = \sqrt{0^2 + 5^2 + 0^2 + 10^2 + 6^2 + 7^2 + 12^2 + 14^2 + 0^2 + 15^2 + 0^2 + 20^2 + 18^2 + 21^2 + 24^2 + 28^2}$$

$$\|C\|_F = \sqrt{0 + 25 + 0 + 100 + 36 + 49 + 144 + 196 + 0 + 225 + 0 + 400 + 324 + 441 + 576 + 784}$$

$$\|C\|_F = \sqrt{2896}$$
$$\|C\|_F \approx 53.87$$

2.2.2.8 Practice Problems

Find the Kronecker product of A and B where A and B are given as follows:

Problem 1:

Find the Kronecker product of:

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

$$B = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Solution:

$$\begin{aligned} A \otimes B &= \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \otimes \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 1 \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} & 2 \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \\ 3 \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} & 4 \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \end{bmatrix} \\ &= \begin{bmatrix} 0 & 1 & 0 & 2 \\ 1 & 0 & 2 & 0 \\ 0 & 3 & 0 & 4 \\ 3 & 0 & 4 & 0 \end{bmatrix} \end{aligned}$$

Problem 2:

Find the Kronecker product of:

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$B = \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix}$$

Solution:

$$
\begin{aligned}
A \otimes B &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix} \\
&= \begin{bmatrix} 1 \cdot \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix} & 0 \cdot \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix} \\ 0 \cdot \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix} & 1 \cdot \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix} \end{bmatrix} \\
&= \begin{bmatrix} 2 & 3 & 0 & 0 \\ 4 & 5 & 0 & 0 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 4 & 5 \end{bmatrix}
\end{aligned}
$$

Problem 3:

Find the Kronecker product of:

$$A = \begin{bmatrix} 1 & 2 \end{bmatrix}$$

$$B = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$$

Solution:

$$
\begin{aligned}
A \otimes B &= \begin{bmatrix} 1 & 2 \end{bmatrix} \otimes \begin{bmatrix} 3 \\ 4 \end{bmatrix} \\
&= \begin{bmatrix} 1 \cdot \begin{bmatrix} 3 \\ 4 \end{bmatrix} & 2 \cdot \begin{bmatrix} 3 \\ 4 \end{bmatrix} \end{bmatrix} \\
&= \begin{bmatrix} 3 & 6 \\ 4 & 8 \end{bmatrix}
\end{aligned}
$$

Problem 4:

Find the Kronecker product of:

$$A = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & -1 \\ 2 & 0 \end{bmatrix}$$

Solution:

$$
\begin{aligned}
A \otimes B &= \begin{bmatrix} 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & -1 \\ 2 & 0 \end{bmatrix} \\
&= \begin{bmatrix} 0 \cdot \begin{bmatrix} 1 & -1 \\ 2 & 0 \end{bmatrix} & 1 \cdot \begin{bmatrix} 1 & -1 \\ 2 & 0 \end{bmatrix} \end{bmatrix} \\
&= \begin{bmatrix} 0 & 0 & 1 & -1 \\ 0 & 0 & 2 & 0 \end{bmatrix}
\end{aligned}
$$

Problem 5:

Find the Kronecker product of:

$$A = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & -2 \end{bmatrix}$$

Solution:

$$
\begin{aligned}
A \otimes B &= \begin{bmatrix} 2 \\ 3 \end{bmatrix} \otimes \begin{bmatrix} 4 & -2 \end{bmatrix} \\
&= \begin{bmatrix} 2 \cdot \begin{bmatrix} 4 & -2 \end{bmatrix} \\ 3 \cdot \begin{bmatrix} 4 & -2 \end{bmatrix} \end{bmatrix} \\
&= \begin{bmatrix} 8 & -4 \\ 12 & -6 \end{bmatrix}
\end{aligned}
$$

Problem 6:

Find the Kronecker product of:

$$A = \begin{bmatrix} 1 & -1 \\ 0 & 2 \end{bmatrix}$$

$$B = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Solution:

$$
\begin{aligned}
A \otimes B &= \begin{bmatrix} 1 & -1 \\ 0 & 2 \end{bmatrix} \otimes \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \\
&= \begin{bmatrix} 1 \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} & -1 \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \\ 0 \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} & 2 \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \end{bmatrix} \\
&= \begin{bmatrix} 0 & 1 & 0 & -1 \\ 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 2 \\ 0 & 0 & 2 & 0 \end{bmatrix}
\end{aligned}
$$

Problem 7:

Find the Kronecker product of:

$$A = \begin{bmatrix} 2 \end{bmatrix}$$

$$B = \begin{bmatrix} 3 & 4 \\ 5 & 6 \end{bmatrix}$$

Solution:

$$
\begin{aligned}
A \otimes B &= \begin{bmatrix} 2 \end{bmatrix} \otimes \begin{bmatrix} 3 & 4 \\ 5 & 6 \end{bmatrix} \\
&= 2 \cdot \begin{bmatrix} 3 & 4 \\ 5 & 6 \end{bmatrix} \\
&= \begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}
\end{aligned}
$$

Problem 8:

Find the Kronecker product of:

$$A = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Solution:

$$
\begin{aligned}
A \otimes B &= \begin{bmatrix} 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} 0 \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & 1 \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \end{bmatrix} \\
&= \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}
\end{aligned}
$$

Problem 9:

Find the Kronecker product of:

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

Solution:

$$
\begin{aligned}
A \otimes B &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \\
&= \begin{bmatrix} 1 \cdot \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} & 0 \cdot \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \\ 0 \cdot \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} & 1 \cdot \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \end{bmatrix} \\
&= \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}
\end{aligned}
$$

Problem 10:

Find the Kronecker product of:

$$A = \begin{bmatrix} 2 & -1 \\ 3 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 0 & 5 \\ -2 & 3 \end{bmatrix}$$

Solution:

$$
\begin{aligned}
A \otimes B &= \begin{bmatrix} 2 & -1 \\ 3 & 4 \end{bmatrix} \otimes \begin{bmatrix} 0 & 5 \\ -2 & 3 \end{bmatrix} \\
&= \begin{bmatrix} 2 \cdot \begin{bmatrix} 0 & 5 \\ -2 & 3 \end{bmatrix} & -1 \cdot \begin{bmatrix} 0 & 5 \\ -2 & 3 \end{bmatrix} \\ 3 \cdot \begin{bmatrix} 0 & 5 \\ -2 & 3 \end{bmatrix} & 4 \cdot \begin{bmatrix} 0 & 5 \\ -2 & 3 \end{bmatrix} \end{bmatrix} \\
&= \begin{bmatrix} 0 & 10 & 0 & -5 \\ -4 & 6 & 2 & -3 \\ 0 & 15 & 0 & 20 \\ -6 & 9 & -8 & 12 \end{bmatrix}
\end{aligned}
$$

2.2.2.9 Connection Between Outer Product and Kronecker Product

1. **Conceptual Connection:**

   - The **outer product** is a special case of the **Kronecker product**. Specifically, if $\mathbf{A}$ is a column vector and $\mathbf{B}$ is a row vector, then $\mathbf{A}$ is a $m \times 1$ matrix and $\mathbf{B}$ is a $1 \times n$ matrix. The Kronecker product of these two matrices will yield the same result as the outer product of these vectors.
   - For matrices $\mathbf{A}$ and $\mathbf{B}$, the Kronecker product involves taking the outer product of each element of $\mathbf{A}$ with the entire matrix $\mathbf{B}$.

2. **Mathematical Formulation:**

   - Let $\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$ and $\mathbf{B} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$. Then:

   $$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} \\ a_{21}\mathbf{B} & a_{22}\mathbf{B} \end{bmatrix}$$

   - If $\mathbf{A} = \mathbf{u}\mathbf{v}^T$ where $\mathbf{u}$ is a column vector and $\mathbf{v}^T$ is a row vector, then the Kronecker product of $\mathbf{u}$ and $\mathbf{v}^T$ yields the same result as the outer product $\mathbf{u} \otimes \mathbf{v}$.

ℹ Note

Summary

- The **outer product** is a specific case of the **Kronecker product** where one of the matrices is a vector (either row or column).

- The **Kronecker product** generalizes the outer product to matrices and is more versatile in applications involving tensor products and higher-dimensional constructs.

2.2.2.10 Matrix Multiplication as Kronecker Product

Given matrices $\mathbf{A}$ and $\mathbf{B}$, where: - $\mathbf{A}$ is an $m \times n$ matrix - $\mathbf{B}$ is an $n \times p$ matrix

The product $\mathbf{C} = \mathbf{AB}$ can be expressed using Kronecker products as:

$$\mathbf{C} = \sum_{k=1}^{n} (\mathbf{A}_{:,k} \otimes \mathbf{B}_{k,:})$$

where: - $\mathbf{A}_{:,k}$ denotes the k-th column of matrix $\mathbf{A}$ - $\mathbf{B}_{k,:}$ denotes the k-th row of matrix $\mathbf{B}$

Example:

Let:

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

and:

$$\mathbf{B} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

To find $\mathbf{C} = \mathbf{AB}$ using Kronecker products:

1. **Compute the Kronecker Product of Columns of A and Rows of B:**

   - For column $\mathbf{A}_{:,1} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$ and row $\mathbf{B}_{1,:} = \begin{bmatrix} 0 & 1 \end{bmatrix}$:

   $$\mathbf{A}_{:,1} \otimes \mathbf{B}_{1,:} = \begin{bmatrix} 0 & 1 \\ 0 & 3 \end{bmatrix}$$

   - For column $\mathbf{A}_{:,2} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$ and row $\mathbf{B}_{2,:} = \begin{bmatrix} 1 & 0 \end{bmatrix}$:

   $$\mathbf{A}_{:,2} \otimes \mathbf{B}_{2,:} = \begin{bmatrix} 2 & 0 \\ 4 & 0 \end{bmatrix}$$

2. **Sum the Kronecker Products:**

$$\mathbf{C} = \begin{bmatrix} 0 & 1 \\ 0 & 3 \end{bmatrix} + \begin{bmatrix} 2 & 0 \\ 4 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 4 & 3 \end{bmatrix}$$

In the previous block we have discussed the Frobenius norm and its applications. Now came back to the discussions on the Kronecker product. The Kronecker product is particularly useful in scenarios where interactions between different types of data need to be modeled comprehensively. In recommendation systems, it allows us to integrate user preferences with item relationships to improve recommendation accuracy.

In addition to recommendation systems, Kronecker products are used in various fields such as:

- Signal Processing: For modeling multi-dimensional signals.
- Machine Learning: In building features for complex models.
- Communication Systems: For modeling network interactions.

By understanding the Kronecker product and its applications, we can extend it to solve complex problems and enhance systems across different domains. To understand the practical use of Kronecker product in a Machine Learning scenario let us consider the following problem statement and its solution.

Problem statement

In the realm of recommendation systems, predicting user preferences for various product categories based on past interactions is a common challenge. Suppose we have data on user preferences for different products and categories. We can use this data to recommend the best products for each user by employing mathematical tools such as the Kronecker product. The User Preference and Category relationships are given in Table 2.4 and Table 2.5 .

Table 2.4: User Preference

User/Item	Electronics	Clothing	Books
User 1	5	3	4
User 2	2	4	5
User 3	3	4	4

Table 2.5: Category Relationships

Category/Feature	Feature 1	Feature 2	Feature 3
Electronics	1	0	0
Clothing	0	1	1
Books	0	1	1

Predict user preferences for different product categories using the Kronecker product matrix.

Solution Procedure

1. *Compute the Kronecker Product:* Calculate the Kronecker product of matrices U and C to obtain matrix K.

   To model the problem, we use the Kronecker product of the user preference matrix U and the category relationships matrix C. This product allows us to predict the user's rating for each category by combining their preferences with the category features.

Formulating Matrices

User Preference Matrix (U): - Dimension: 3×3 (3 users, 3 items) - from the User preference data, we can create the User Preference Matrix as follows:

$$U = \begin{pmatrix} 5 & 3 & 4 \\ 2 & 4 & 5 \\ 3 & 4 & 4 \end{pmatrix}$$

Category Relationships Matrix (C): - Dimension: 3×3 (3 categories) - from the Category Relationships data, we can create the Category Relationship Matrix as follows:

$$C = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

Kronecker Product Calculation

The Kronecker product K of U and C is calculated as follows:

1. **Matrix Dimensions:**

- U is 3×3 (3 users, 3 items).
- C is 3×3 (3 categories, 3 features).

2. **Calculate Kronecker Product:**

- For each element u_{ij} in U, multiply by the entire matrix C.

The Kronecker product K is computed as:

$$K = U \otimes C$$

Explicitly, the Kronecker product K is:

$$K = \begin{pmatrix} 5 \cdot C & 3 \cdot C & 4 \cdot C \\ 2 \cdot C & 4 \cdot C & 5 \cdot C \\ 3 \cdot C & 4 \cdot C & 4 \cdot C \end{pmatrix}$$

As an example the blocks in first row are:

$$5 \cdot C = \begin{pmatrix} 5 & 0 & 0 \\ 0 & 5 & 5 \\ 0 & 5 & 5 \end{pmatrix}, \quad 3 \cdot C = \begin{pmatrix} 3 & 0 & 0 \\ 0 & 3 & 3 \\ 0 & 3 & 3 \end{pmatrix}, \quad 4 \cdot C = \begin{pmatrix} 4 & 0 & 0 \\ 0 & 4 & 4 \\ 0 & 4 & 4 \end{pmatrix}$$

Combining these blocks:

$$K = \begin{pmatrix} 5 & 0 & 0 & 3 & 0 & 0 & 4 & 0 & 0 \\ 0 & 5 & 5 & 0 & 3 & 3 & 0 & 4 & 4 \\ 0 & 5 & 5 & 0 & 3 & 3 & 0 & 4 & 4 \\ 2 & 0 & 0 & 4 & 0 & 0 & 5 & 0 & 0 \\ 0 & 2 & 2 & 0 & 4 & 4 & 0 & 5 & 5 \\ 0 & 2 & 2 & 0 & 4 & 4 & 0 & 5 & 5 \\ 3 & 0 & 0 & 4 & 0 & 0 & 4 & 0 & 0 \\ 0 & 3 & 3 & 0 & 4 & 4 & 0 & 4 & 4 \\ 0 & 3 & 3 & 0 & 4 & 4 & 0 & 4 & 4 \end{pmatrix}$$

2. **Interpret the Kronecker Product Matrix:** The resulting matrix K represents all possible combinations of user preferences and category features.

3. **Predict Ratings:** For each user, use matrix K to predict the rating for each category by summing up the values in the corresponding rows.

4. **Generate Recommendations:** Identify the top categories with the highest predicted ratings for each user.

The `python` code to solve this problem computationally is given below.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Define the matrices
U = np.array([[5, 3, 4],
              [2, 4, 5],
              [3, 4, 4]])

C = np.array([[1, 0, 0],
              [0, 1, 1],
              [0, 1, 1]])

# Compute the Kronecker product
K = np.kron(U, C)

# Create a DataFrame to visualize the Kronecker product matrix
df_K = pd.DataFrame(K,
                    columns=['Electronics_F1', 'Electronics_F2',
 ↪  'Electronics_F3',
                             'Clothing_F1', 'Clothing_F2', 'Clothing_F3',
                             'Books_F1', 'Books_F2', 'Books_F3'],
                    index=['User 1 Electronics', 'User 1 Clothing', 'User 1
 ↪  Books',
                           'User 2 Electronics', 'User 2 Clothing', 'User 2
                            ↪  Books',
                           'User 3 Electronics', 'User 3 Clothing', 'User 3
                            ↪  Books'])

# Print the Kronecker product matrix
print("Kronecker Product Matrix (K):\n", df_K)

# Predict ratings and create recommendations
def recommend(user_index, top_n=3):
    """ Recommend top_n categories for a given user based on Kronecker
     ↪  product matrix. """
    user_ratings = K[user_index * len(C):(user_index + 1) * len(C), :]
    predicted_ratings = np.sum(user_ratings, axis=0)
    recommendations = np.argsort(predicted_ratings)[::-1][:top_n]
    return recommendations

# Recommendations for User 1
```

```python
user_index = 0  # User 1
top_n = 3
recommendations = recommend(user_index, top_n)

print(f"\nTop {top_n} recommendations for User {user_index + 1}:")
for rec in recommendations:
    print(df_K.columns[rec])
```

```
Kronecker Product Matrix (K):
                     Electronics_F1  Electronics_F2  Electronics_F3  \
User 1 Electronics                5               0               0
User 1 Clothing                   0               5               5
User 1 Books                      0               5               5
User 2 Electronics                2               0               0
User 2 Clothing                   0               2               2
User 2 Books                      0               2               2
User 3 Electronics                3               0               0
User 3 Clothing                   0               3               3
User 3 Books                      0               3               3

                    Clothing_F1  Clothing_F2  Clothing_F3  Books_F1  Books_F2  \
User 1 Electronics            3            0            0         4         0
User 1 Clothing               0            3            3         0         4
User 1 Books                  0            3            3         0         4
User 2 Electronics            4            0            0         5         0
User 2 Clothing               0            4            4         0         5
User 2 Books                  0            4            4         0         5
User 3 Electronics            4            0            0         4         0
User 3 Clothing               0            4            4         0         4
User 3 Books                  0            4            4         0         4

                    Books_F3
User 1 Electronics         0
User 1 Clothing            4
User 1 Books               4
User 2 Electronics         0
User 2 Clothing            5
User 2 Books               5
User 3 Electronics         0
User 3 Clothing            4
User 3 Books               4
```

```
Top 3 recommendations for User 1:
Electronics_F2
Electronics_F3
Books_F3
```

A simple visualization of this recomendation system is shown in Fig 2.11.

```python
# Visualization
def plot_recommendations(user_index):
    """ Plot the predicted ratings for each category for a given user. """
    user_ratings = K[user_index * len(C):(user_index + 1) * len(C), :]
    predicted_ratings = np.sum(user_ratings, axis=0)
    categories = df_K.columns
    plt.figure(figsize=(6, 5))
    plt.bar(categories, predicted_ratings)
    plt.xlabel('Categories')
    plt.ylabel('Predicted Ratings')
    plt.title(f'Predicted Ratings for User {user_index + 1}')
    plt.xticks(rotation=45)
    plt.show()

# Plot recommendations for User 1
plot_recommendations(user_index)
```

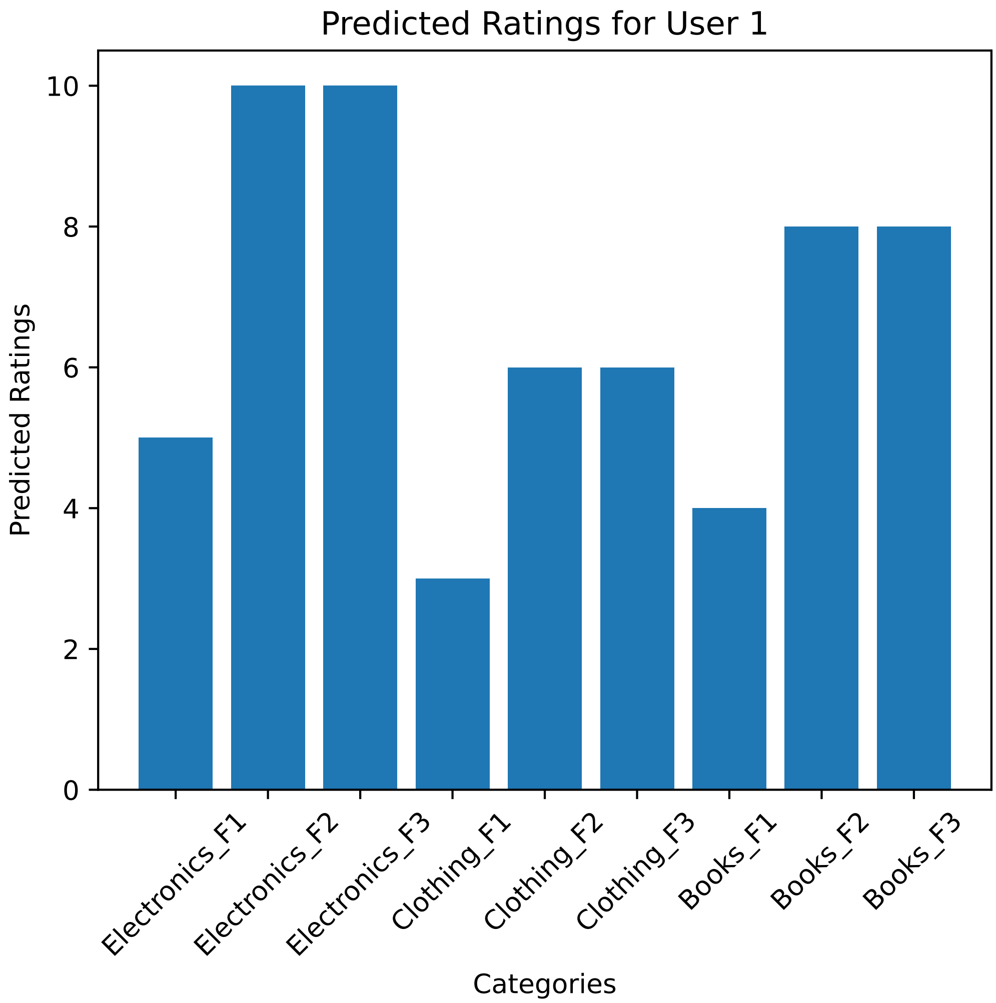


Figure 2.11: EDA for the Recommendation System

This micro project illustrate one of the popular use of Kronecker product on ML application.

2.2.3 Matrix Measures of Practical Importance

Matrix measures, such as rank and determinant, play crucial roles in linear algebra. While both rank and determinant provide valuable insights into the properties of a matrix, they serve different purposes. Understanding their roles and applications is essential for solving complex problems in computer science, engineering, and applied mathematics.

2.2.3.1 Determinant

Determinant of a 2×2 matrix $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ is defined as $|A| = ad - bc$. Determinant of higher order square matrices can be found using the Laplace method or Sarrus method.

The determinant of a matrix provides information about the matrix's invertibility and scaling factor for volume transformation. Specifically:

1. *Invertibility:* A matrix is invertible if and only if its determinant is non-zero.
2. *Volume Scaling:* The absolute value of the determinant gives the scaling factor by which the matrix transforms volume.
3. *Parallelism:* If the determinant of a matrix composed of vectors is zero, the vectors are linearly dependent, meaning they are parallel or redundant.
4. *Redundancy:* A zero determinant indicates that the vectors span a space of lower dimension than the number of vectors, showing redundancy.

! Least Possible Values of Determinant

1. *Least Positive Determinant:* For a 1×1 matrix, the smallest non-zero determinant is any positive value, typically ϵ, where ϵ is a small positive number.
2. Least Non-Zero Determinant: For higher-dimensional matrices, the smallest non-zero determinant is a non-zero value that represents the smallest area or volume spanned by the matrix's rows or columns. For example a 2×2 matrix with determinant ϵ could be:
$$B = \begin{pmatrix} \epsilon & 0 \\ 0 & \epsilon \end{pmatrix}$$
Here, ϵ is a small positive number, indicating a very small but *non-zero* area.

Now let's look into the most important matrix measure for advanced application in Linear Algebra.

As we know the matrix is basically a representation tool that make things abstract- remove unnecessary details. Then the matrix itself can be represented in many ways. This is the real story telling with this most promising mathematical structure. Consider a context of collecting feedback about a product in three aspects- cost, quality and practicality. For simplicity in calculation, we consider responses from 3 customers only. The data is shown in Table 2.6.

Table 2.6: User rating of a consumer product

User	Cost	Quality	Practicality
User-1	1	4	5

User	Cost	Quality	Practicality
User-2	3	2	5
User-3	2	1	3

It's perfect and nice looking. But both mathematics and a computer can't handle this table as it is. So we create an abstract representation of this data- the rating matrix. Using the traditional approach, let's represent this rating data as:

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

Now both the column names and row indices were removed and the data is transformed into the abstract form. This representation has both advantages and disadvantages. Be positive! So we are focused only in the advantages.

Just consider the product. Its sales fully based on its features. So the product sales perspective will be represented in terms of the features- cost, quality and practicality. These features are columns of our rating matrix. Definitly peaple will have different rating for these features. Keeping all these in mind let's introduce the concept of *linear combination*. This leads to a new matrix product as shown below.

$$\begin{aligned} Ax &= \begin{bmatrix} 1 & 4 & 5 \\ 3 & 2 & 5 \\ 2 & 1 & 3 \end{bmatrix} x \\ &= \begin{bmatrix} 1 & 4 & 5 \\ 3 & 2 & 5 \\ 2 & 1 & 3 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \\ &= \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix} x_1 + \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix} x_1 + \begin{bmatrix} 4 \\ 2 \\ 1 \end{bmatrix} x_2 + \begin{bmatrix} 5 \\ 5 \\ 3 \end{bmatrix} x_3 \end{aligned}$$

As the number of users increases, the product sales perspective become more informative. In short the span of the features define the feature space of the product. In real cases, a manufacture wants to know what are the features really inflence the customers. This new matrix product will help the manufactures to identify that features!

So we are going to define this new matrix product as the feature space, that will provide more insights to this context as:

$$A = CR$$

Where C is the column space and R is the row reduced Echelon form of A. But the product is not the usual scalar projection, Instead the weight of linear combination of elements in the column space.

Let's formally illustrate this in our example. From the first observation itself, it is clear that last column is just the sum of first and second columns (That is in our context the feature 'practicality' is just depends on 'cost' and 'quality'. meaningful?). So only first columns are independent and so spans the column space.

$$C = \begin{bmatrix} 1 & 4 \\ 3 & 2 \\ 2 & 1 \end{bmatrix}$$

Now look into the matrix R. Applying elementary row tansformations, A will transformed into:

$$R = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

Hence we can form a decomposition for the given rating matrix, A as:

$$\begin{aligned} A &= CR \\ &= \begin{bmatrix} 1 & 4 \\ 3 & 2 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ & & \end{bmatrix} \end{aligned}$$

This decomposition says that there are only two independent features (columns) and the third feature (column) is the sum of first two features (columns).

! Interpretation of the R matrix

Each column in the R matrix represents the weights for linear combination of vectors in the column space to get that column in A. In this example, third column of R is $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$. This means that third column of A will be $1 \times C_1 + 1 \times C_2$ of the column space, C!

This first matrix decompostion donate a new type of matrix product (outer product) and a new measure- the number of independent columns and number of independent rows. This count is called the *rank* of the matrix A. In the case of features, if the rank of the column space is less than the number of features then definitly a less number of feature set will perfectly represent the data. This will help us to reduce the dimension of the dataset and there by reducing computational complexities in data analysis and machine Learning jobs.

In the above discussion, we consider only the columns of A. Now we will mention the row space. It is the set of all linearly independent rows of A. For any matrix A, both the row space and column space are of same rank. This correspondance is a helpful result in many practical applications.

Now we consider a stable equation, $Ax = 0$. With the usual notation of dot product, it implies that x is orthogonal to A. Set of all those independent vectors which are orthogonal to A constitute a new space of interest. It is called the *null space* of A. If A represents a linear transformation, then the null space will be populated by those non-zero vectors which are *nullified* by the transformation A. As a summary of this discussion, the row space and null space of a matrix A creates an orthogonal system. Considering the relationship between A and A^T, it is clear that row space of A is same as the column space of A^T and vice verse are. So we can restate the orthogonality as: 'the null space of A is orthogonal to the column space of A^T' and 'the null space of A^T is orthogonal to the column space of A'. Mathematically this property can be represents as follows.

> **Note**
>
> $$\begin{aligned} \mathcal{N}(A) &\perp \mathcal{C}(A^T) \\ \mathcal{N}(A^T) &\perp \mathcal{C}(A) \end{aligned}$$

In the given example, solving $Ax = 0$ we get $x = \begin{bmatrix} 1 & 1 & -1 \end{bmatrix}^T$.

So the rank of $\mathcal{N}(A) = 1$. Already we have rank of $A = 2$. This leads to an interesting result:

$$\text{Rank}(A) + \text{Rank}(\mathcal{N}(A)) = 3$$

This observation can be framed as a theorem.

2.2.4 Rank Nullity Theorem

The rank-nullity theorem is a fundamental theorem in linear algebra that is important for understanding the connections between mathematical operations in engineering, physics, and computer science. It states that the sum of the rank and nullity of a matrix equals the number of columns in the matrix. The rank is the maximum number of linearly independent columns, and the nullity is the dimension of the nullspace.

Theorem 2.1 (Rank Nullitty Theorem)**.** *The Rank-Nullity Theorem states that for any $m \times n$ matrix A, the following relationship holds:*

$$Rank(A) + Nullity(A) = n$$

*where: - **Rank** of A is the dimension of the column space of A, which is also equal to the dimension of the row space of A. - **Nullity** of A is the dimension of the null space of A, which is the solution space to the homogeneous system* $A\mathbf{x} = \mathbf{0}$.

Steps to Formulate for Matrix A

1. **Find the Rank of** A: The rank of a matrix is the maximum number of linearly independent columns (or rows). It can be determined by transforming A into its row echelon form or reduced row echelon form (RREF).
2. **Find the Nullity of** A: The nullity is the dimension of the solution space of $A\mathbf{x} = \mathbf{0}$. This can be found by solving the homogeneous system and counting the number of free variables.
3. **Apply the Rank-Nullity Theorem**: Use the rank-nullity theorem to verify the relationship.

Example 1: Calculate the rank and nullity of $A = \begin{bmatrix} 1 & 4 & 5 \\ 3 & 2 & 5 \\ 2 & 1 & 3 \end{bmatrix}$ and verify the rank nullity theorem.

1. **Row Echelon Form**:

   Perform Gaussian elimination on A:

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

   Perform row operations to get it to row echelon form:

   - Subtract 3 times row 1 from row 2:

$$\begin{bmatrix} 1 & 4 & 5 \\ 0 & -10 & -10 \\ 2 & 1 & 3 \end{bmatrix}$$

- Subtract 2 times row 1 from row 3:

$$\begin{bmatrix} 1 & 4 & 5 \\ 0 & -10 & -10 \\ 0 & -7 & -7 \end{bmatrix}$$

- Add $\frac{7}{10}$ times row 2 to row 3:

$$\begin{bmatrix} 1 & 4 & 5 \\ 0 & -10 & -10 \\ 0 & 0 & 0 \end{bmatrix}$$

The matrix is now in row echelon form.

Rank is the number of non-zero rows, which is 2.

2. **Find the Nullity**: The matrix A has 3 columns. The number of free variables in the solution of $A\mathbf{x} = \mathbf{0}$ is $3 - \text{Rank}$.

   So,

$$\text{Nullity}(A) = 3 - 2 = 1$$

3. **Apply the Rank-Nullity Theorem**:

$$\text{Rank}(A) + \text{Nullity}(A) = 2 + 1 = 3$$

   This matches the number of columns of A, confirming the theorem.

2.2.5 Fundamental Subspaces

In section (**note-ortho?**), we have seen that for any matrix A, there is two pairs of inter-related orthogonal spaces. This leads to the concept of Fundamental sup spaces.

Matrices are not just arrays of numbers; they can represent linear transformations too. A linear transformation maps vectors from one vector space to another while preserving vector addition and scalar multiplication. The matrix A can be viewed as a representation of a linear transformation T from $\mathbb{R}^n$ to $\mathbb{R}^m$ where:

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

In this context:

- The column space of A represents the range of T, which is the set of all possible outputs.

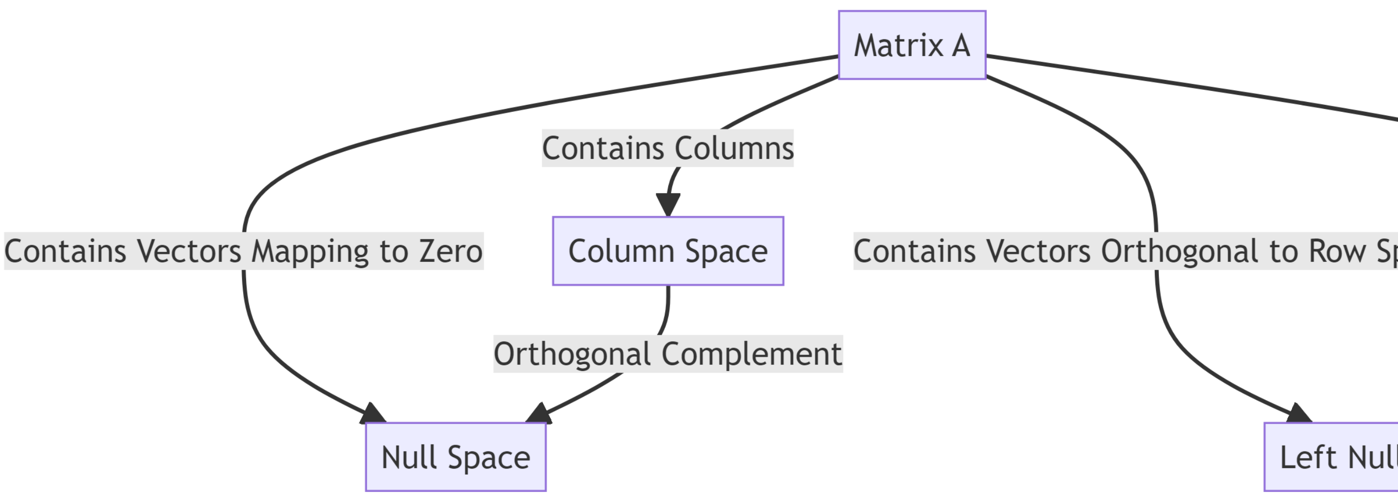
- The null space of A represents the kernel of T, which is the set of vectors that are mapped to the zero vector.

The Four Fundamental Subspaces

Understanding the four fundamental subspaces helps in analyzing the properties of a linear transformation. These subspaces are:

Definition 2.1 (Four Fundamental Subspaces)**.** Let $T : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ be a linear transformation and A represents the matrix of transformation. The four fundamental subspaces are defined as:

1. **Column Space (Range)**: The set of all possible outputs of the transformation. For matrix A, this is the span of its columns. It represents the image of $\mathbb{R}^n$ under T.
2. **Null Space (Kernel)**: The set of all vectors that are mapped to the zero vector by the transformation. For matrix A, this is the solution space of $A\mathbf{x} = \mathbf{0}$.
3. **Row Space**: The span of the rows of A. This space is crucial because it helps in understanding the rank of A. The dimension of the row space is equal to the rank of A, which represents the maximum number of linearly independent rows.
4. **Left Null Space**: The set of all vectors $\mathbf{y}$ such that $A^T\mathbf{y} = \mathbf{0}$. It provides insight into the orthogonal complement of the row space.



This idea is depicted as a 'Big picture of the four sub spaces of a matrix' in the Strang's text book on Linear algebra for every one (Strang 2020). This 'Big Picture' is shown in Fig- 2.12.

A video session from Strang's session is here:

https://youtu.be/rwLOfdfc4dw?si=DsJb8KJTF05hHc76

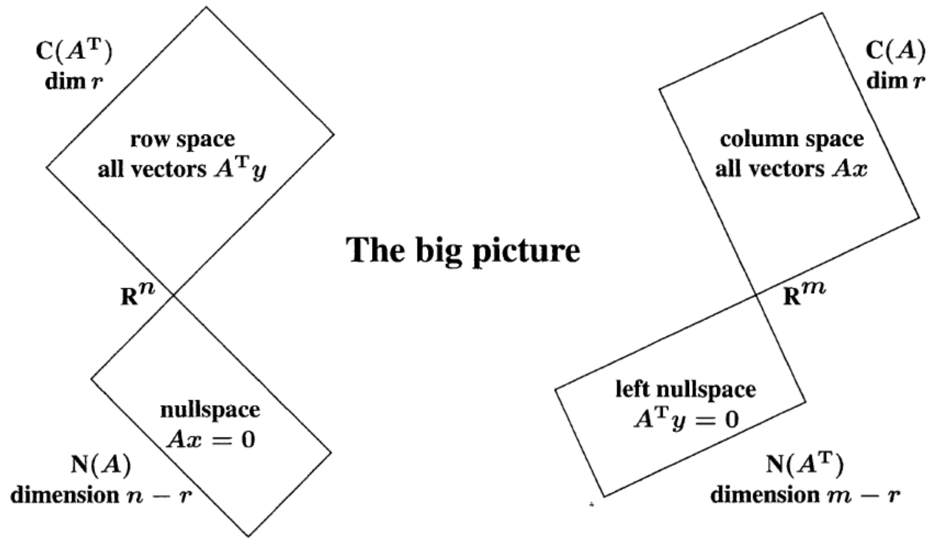


Figure 2.12: The Big Pictue of Fundamental Subspaces

2.2.5.1 Practice Problems

Problem 1: Express the vector $(1, -2, 5)$ as a linear combination of the vectors $(1, 1, 1)$, $(1, 2, 3)$ and $(2, -1, 1)$.

Problem 2: Show that the feature vector $(2, -5, 3)$ is not linearly associated with the features $(1, -3, 2)$, $(2, -4, -1)$ and $(1, -5, 7)$.

Problem 3: Show that the feature vectors $(1, 1, 1)$, $(1, 2, 3)$ and $(2, -1, 1)$ are non-redundant.

Problem 4: Prove that the features $(1, -1, 1)$, $(0, 1, 2)$ and $(3, 0, -1)$ form basis for the feature space.

Problem 5: Check whether the vectors $(1, 2, 1)$, $(2, 1, 4)$ and $(4, 5, 6)$ form a basis for $\mathbb{R}^3$.

Problem 6: Find the four fundamental subspaces of the feature space created by $(1, 2, 1)$, $(2, 1, 4)$ and $(4, 5, 6)$.

Problem 7: Find the four fundamental subspaces and its dimensions of the matrix $\begin{bmatrix} 1 & 2 & 4 \\ 2 & 1 & 5 \\ 1 & 4 & 6 \end{bmatrix}$.

Problem 8: Express $A = \begin{bmatrix} 1 & 2 & -1 \\ 3 & 1 & -1 \\ 2 & -1 & 0 \end{bmatrix}$ as the Kronecker product of the column space and the row space in the form $A = C \otimes R$.

Problem 9: Find the four fundamental subspaces of $A = \begin{bmatrix} 1 & 2 & 0 & 2 & 5 \\ -2 & -5 & 1 & -1 & -8 \\ 0 & -3 & 3 & 4 & 1 \\ 3 & 6 & 0 & -7 & 2 \end{bmatrix}$.

Problem 10: Find the four fundamental subspaces of $A = \begin{bmatrix} -1 & 2 & -1 & 5 & 6 \\ 4 & -4 & -4 & -12 & -8 \\ 2 & 0 & -6 & -2 & 4 \\ -3 & 1 & 7 & -2 & 12 \end{bmatrix}$.

Problem 11: Express $A = \begin{bmatrix} 2 & 3 & -1 & -1 \\ 1 & -1 & -2 & -4 \\ 3 & 1 & 3 & -2 \\ 6 & 3 & 0 & -7 \end{bmatrix}$ in $A = C \otimes R$, where C is the column space and R is the row space of A.

Problem 12: Express $A = \begin{bmatrix} 0 & 1 & -3 & -1 \\ 1 & 0 & 1 & 1 \\ 3 & 1 & 0 & 2 \\ 1 & 1 & -2 & 0 \end{bmatrix}$ in $A = C \otimes R$, where C is the column space and R is the row space of A.

Problem 13: Show that the feature vectors $(2, 3, 0)$, $(1, 2, 0)$ and $(8, 13, 0)$ are redundant and hence find the relationship between them.

Problem 14: Show that the feature vectors $(1, 2, 1)$, $(4, 1, 2)$, $(-3, 8, 1)$ and $(6, 5, 4)$ are redundant and hence find the relationship between them.

Problem 15: Show that the feature vectors $(1, 2, -1, 0)$, $(1, 3, 1, 2)$, $(4, 2, 1, 0)$ and $(6, 1, 0, 1)$ are redundant and hence find the relationship between them.

! Important

Three Parts of the *Fundamental theorem* The fundamental theorem of linear algebra relates all four of the fundamental subspaces in a number of different ways. There are main parts to the theorem:
Part 1:(Rank nullity theorem) The column and row spaces of an $m \times n$ matrix A both have dimension r, the rank of the matrix. The nullspace has dimension $n - r$, and the left nullspace has dimension $m - r$.
Part 2:(Orthogonal subspaces) The nullspace and row space are orthogonal. The left nullspace and the column space are also orthogonal.

Part 3:(Matrix decomposition) The final part of the fundamental theorem of linear algebra constructs an orthonormal basis, and demonstrates a singular value decomposition: any matrix M can be written in the form $M = U\Sigma V^T$, where $U_{m\times m}$ and $V_{n\times n}$ are unitary matrices, $\Sigma_{m\times n}$ matrix with nonnegative values on the diagonal.
This part of the fundamental theorem allows one to immediately find a basis of the subspace in question. This can be summarized in the following table.

Subspace	Subspace of	Symbol	Dimension	Basis
Column space	$\mathbb{R}^m$	$\mathrm{im}(A)$	r	First r columns of U
Nullspace (kernel)	$\mathbb{R}^n$	$\ker(A)$	$n-r$	Last $n-r$ columns of V
Row space	$\mathbb{R}^n$	$\mathrm{im}(A^T)$	r	First r columns of V
Left nullspace (kernel)	$\mathbb{R}^m$	$\ker(A^T)$	$m-r$	Last $m-r$ columns of U

2.2.5.2 Rank and Solution of System of Linear Equations

In linear algebra, the rank of a matrix is a crucial concept for understanding the structure of a system of linear equations. It provides insight into the solutions of these systems, helping us determine the number of independent equations and the nature of the solution space.

Definition 2.2 (Rank and System Consistency)**.** The rank of a matrix A is defined as the maximum number of linearly independent rows or columns. When solving a system of linear equations represented by $A\mathbf{x} = \mathbf{b}$, where A is an $m \times n$ matrix and $\mathbf{b}$ is a vector, the rank of A plays a crucial role in determining the solution's existence and uniqueness.

Consistency of the System

1. **Consistent System:** A system of linear equations is consistent if there exists at least one solution. This occurs if the rank of the coefficient matrix A is equal to the rank of the augmented matrix $[A|\mathbf{b}]$. Mathematically, this can be expressed as:

$$\mathrm{rank}(A) = \mathrm{rank}([A|\mathbf{b}])$$

   If this condition is met, the system has solutions. The solutions can be:

   - **Unique** if the rank equals the number of variables.
   - **Infinitely many** if the rank is less than the number of variables.

2. **Inconsistent System:** A system is inconsistent if there are no solutions. This occurs when:

$$\mathrm{rank}(A) \neq \mathrm{rank}([A|\mathbf{b}])$$

In this case, the equations represent parallel or conflicting constraints that cannot be satisfied simultaneously.

References

Knuth, Donald E. 1984. "Literate Programming." *Comput. J.* 27 (2): 97–111. https://doi.org/10.1093/comjnl/27.2.97.

Strang, Gilbert. 2020. *Linear Algebra for Everyone*. SIAM.

———. 2022. *Introduction to Linear Algebra*. SIAM.