CMML

A C-Language Matrix & Machine Learning Library

Documentation Booklet

Release: v0.5

Module: Cmatrix.h

Released: 21 November, 2025

INDEX

1. ABSTRACT

This project presents a custom Matrix and Machine Learning Utility Library implemented in the C programming language. The library provides essential tools for creating, manipulating, and performing mathematical operations on matrices, along with machine-learning- oriented utilities such as activation functions, Gaussian initialization, broadcasting, and vector operations. It aims to simplify mathematical computation in C by offering a structured, reusable, and efficient set of functions. The project also demonstrates fundamental linear- algebra algorithms such as matrix multiplication, transposition, determinants, and inverse calculations. This library serves as a foundational module for future machine learning or numerical computing projects written in C.

2. Introduction

2.1 Overview of the Project

The **C-Language Matrix & Machine Learning Library (CMML)** is a specialized compact, portable C library designed to implement the core mathematical engines that drive modern Artificial Intelligence. While high-level languages like Python rely on pre-compiled libraries (such as NumPy or TensorFlow) to handle heavy computations, this project rebuilds those capabilities from scratch using the **C programming language.**

The primary focus of this library is to enable users to build and grasp the fundamental principles of machine learning and the functionalities that drive these concepts.

Motivation behind CMML is "learning by implementation." Machine Learning is fundamentally based on Linear Algebra—specifically, operations performed on matrices and vectors. By creating a custom library, this project provides a transparent view of how data is stored in memory, how matrices are manipulated at a low level, and how complex algorithms like matrix inversion and Gaussian broadcasting are executed.

The system is designed to be:

- **Lightweight:** It requires no external dependencies or heavy frameworks.
- **Modular:** Functions are grouped logically, allowing for easy expansion.
- **Educational:** It prioritizes clear logic and safe memory management over obscure optimizations.

2.2 Description of Modules

Module 1 — Memory Management & Data Structure

This foundational module defines the library's core data type and implements robust memory handling utilities. The matrix structure encapsulates matrix dimensions (row, col) and a flattened float *value buffer for storage. Memory allocation is centralized via `malloc_safe(size_t n)`, which handles allocation failures in a consistent, fail-fast manner. Complementary lifecycle functions such as `free_matrix()` ensure resources are released cleanly and help prevent memory leaks across the codebase.

Key items: struct matrix, `malloc_safe()`, `free_matrix()`

Module 2 — Initialization & Constructors

This module provides all constructors and common initializers used to create and duplicate matrix objects. It encapsulates allocation patterns and common initialization strategies so callers can create matrices with minimal boilerplate.

Primary functions:

- `new_matrix(row, col)` — allocate and initialize an empty matrix.
- `zeros(row, col)` — allocate and fill with zeros.
- `new_random_matrix(row, col, min, max)` —uniform random initialization.
- `new_gaussian_matrix(row, col)` — Gaussian-distributed initialization (uses normal_rand).
- `array_matrix(arr, row, col)` — wrap an existing array into a matrix copy.
- `copy_matrix(const matrix *old)` — deep copy of an existing matrix.
- `free_matrix(matrix *m)` — release matrix memory (see Module 1).

Module 3 — Accessors, Utilities & Data Manipulation

This module groups read-only inspection utilities and functions that safely mutate or reshape matrix data. The accessors and debugging helpers make it easy to validate matrix shapes and contents, while the manipulation APIs support controlled updates and sub-region extraction.

Accessors & utilities:

- `get_rows(), get_cols(), get_matrix_element()` — dimension and element queries.
- `print_matrix(), print_shape()` — human-readable output for debugging.
- `matrix_check_equal(a, b, tolerance)` — approximate equality check for tests.
- Data manipulation:
- `set_matrix(), fill_matrix()` — element assignment and bulk fill.
- `reshape_matrix(), reshape_matrix_inplace()` — safe reshaping with dimension checks.
- `get_row(), get_col(), get_slice()` — extract 1×N, N×1, or submatrix views (as new matrices).

Module 4 — Arithmetic Operations (Element-wise Math)

This module implements element-wise mathematical operations that mirror standard numerical-library behavior. Each core operation is provided in two variants: one that returns a new matrix and an "in-place" variant that mutates an existing matrix to reduce memory overhead.

Functions include:

- add_matrix, add_matrix_inplace
- subtract_matrix, subtract_matrix_inplace
- hadamard_matrix, hadamard_matrix_inplace

- Scalar operations: scalar_multiply, scalar_multiply_inplace, scalar_add, scalar_add_inplace

---

Module 5 — Linear Algebra Engine

This module contains higher-level algebraic routines and the row-operations needed to implement them. It provides the computational core required for common matrix computations and linear-system tasks.

Capabilities & functions:

- Matrix multiplication: `multiply_matrix()` (row×column dot-product implementation).
- Vector math: `dot_product()` for vector-shaped matrices.
- Transformations: `transpose_matrix()`.
- Solvers & algebraic tools: `determinant_matrix()`, `inverse_matrix()`, `cofactor_matrix()`, `adjoint_matrix()`, `minor_matrix()`.
- Elementary row operations: `swap_rows()`, `add_rows()`, `multiply_row()` (used internally for elimination-based algorithms).
- The determinant and inverse routines use direct algebraic approaches (row-reduction and cofactors/adjoints) to remain straightforward and pedagogical, making the code easy to study and validate.

Module 6 — Machine Learning Utilities

This module bridges numeric routines to common machine-learning workflows, offering convenient and flexible helpers for model initialization, preprocessing, and inference.

Highlights:

Activation & elementwise transforms: `apply_function()`, `apply_function_inplace()` accept a float (*f)(float) pointer so any scalar activation (Sigmoid, ReLU, etc.) may be applied across a matrix.

Broadcasting: `broadcast_add(), broadcast_add_inplace()` add a column-vector bias to every row of a matrix (NumPy-like semantics).
Selection & initialization: argmax returns the index of the largest element, and `normal_rand()` produces Gaussian-distributed values for weight initialization.
This module is designed to make the library immediately useful for lightweight neural-network experiments and educational ML projects implemented in C.

---

3. Feasibility Study

3.1 Physical (Technical) Feasibility

Physical feasibility assesses the hardware and software resources required to implement the system.

- **Hardware Independence:** Being written in standard C, the CMML library is highly portable. It does not require high-end GPUs or specialized processors to run basic matrix operations. It can be compiled and executed on any machine with a standard C compiler (GCC/Clang), ranging from low-power embedded devices to high-performance servers.
- **Software Requirements:** The project has zero external dependencies. It relies solely on the C Standard Library (stdlib.h, math.h), ensuring that it can be integrated into any system without "dependency hell" or complex installation procedures.
- **Conclusion:** The project is physically feasible as it operates efficiently on minimal hardware resources.

3.2 Behavioral (Operational) Feasibility

Behavioral feasibility determines if the end-users (in this case, developers and students) will be able to adapt to the new system.

- **Standardization:** The library uses standard mathematical structures (rows, columns) that are intuitive to anyone familiar with linear algebra.
- **Ease of Integration:** The API design mirrors common mathematical operations (e.g., `add_matrix(), multiply_matrix()).` A developer

familiar with mathematical notation or other libraries like NumPy will find the transition to CMML straightforward.

- **Error Handling:** The inclusion of safe memory wrappers (malloc_safe) and dimension checks ensures that the system behaves predictably, reducing user frustration during debugging.
- **Conclusion:** The project is operationally feasible as it requires no special training beyond standard C programming knowledge.

3.3 Economic Feasibility

Economic feasibility analyzes the cost-benefit ratio of the project.

- **Development Cost:** As an open-source project developed using free tools (VS Code, GCC compiler), the direct financial cost is zero. The primary investment is the developer's time and intellectual effort.
- **Operational Cost:** Unlike Python-based solutions that may require heavy runtime environments (interpreters, virtual environments), this compiled C library incurs negligible operational costs in terms of memory and CPU cycles.
- **Benefit:** The educational value of understanding low-level ML implementation outweighs the development effort. Furthermore, it provides a free alternative to complex commercial libraries for small-scale embedded projects.
- **Conclusion:** The project is economically feasible with a high return on investment regarding performance and educational value.

4. Background Study

4.1 Existing Systems

To understand the necessity of this project, we must first analyze the current landscape of mathematical and machine learning tools available to developers. The existing ecosystem can be broadly categorized into three groups:

- **The Python Ecosystem (NumPy, TensorFlow, PyTorch):** These libraries are currently the industry standard for Data Science and Machine Learning. They function as high-level wrappers around pre-compiled C/C++ backends. While they are user-friendly, they suffer from significant "dependency hell"—requiring gigabytes of installation space for interpreters and libraries. Furthermore, they suffer from slower runtime performance for simple loops due to interpretation overhead and, most critically, they act as "black boxes" where the actual mathematical algorithms are hidden behind compiled binaries, limiting deep learning.
- **C++ Libraries (Eigen, Armadillo):** Libraries like Eigen are widely used in game engines and high-frequency trading for their speed. However, they rely heavily on "Template Meta-Programming," a complex feature of C++ that creates a steep learning curve. The syntax is often extremely difficult for beginners to read or debug, making them unsuitable for students attempting to understand the core logic of matrix manipulation.
- **Standard C Libraries (BLAS / LAPACK):** The Basic Linear Algebra Subprograms (BLAS) and Linear Algebra Package (LAPACK) have been the standard since the 1970s. While highly optimized, they are considered outdated for modern educational use. They lack essential Machine Learning utilities (such as activation functions or broadcasting), and their APIs are notoriously unfriendly, often using cryptic function names (e.g., dgemm, sgetrf) that obscure the code's intent.

4.2 Gaps & Limitations

Popular high-level libraries and ecosystems present several challenges for learners and small C projects: they often hide implementation details behind compiled binaries, preventing students from studying core algorithms; they introduce substantial dependency overhead (large runtimes and many packages) that makes simple projects cumbersome to set up; and they obscure manual memory management, reducing exposure to pointers and lifecycle concerns.

At the same time, high-performance C++ and C libraries tend to be either very complex (heavy template use and advanced APIs) or too low-level, and standard C tooling generally lacks convenient ML helpers such as activation functions, broadcasting, Gaussian initialization, and argmax.

Compounding these issues, many educational C examples provide poor documentation and weak error handling, which makes them brittle and difficult to extend.

5. Objectives

5.1 Primary Objective:

The Primary purpose of this project was to build something low-level in which I can learn the basic concepts of machine learning. I wanted a simple, transparent codebase to understand how complex ML algorithms operate on actual mathematical parameters. As noted in Sections 4.1 and 4.2, current industry-standard tools and libraries are excellent, but they are often too abstract and require deep domain insight; because of their complexity they demand substantial time to learn. I therefore created a C library that addresses these issues by using C — a low-level language — and by providing very basic mathematical tools so that users do not need to start from zero and can instead focus on learning ML concepts.

5.2 Secondary Objective (Performance & Control):

A secondary goal was performance: because C is a low-level language it offers excellent runtime performance and fine-grained benchmarking capabilities. The library is intended for testing ML algorithms on a minimal scale where near-complete customization and explicit control over parameters are required.

Note: for large-scale, GPU-accelerated workloads, modern ML frameworks that leverage specialized hardware will usually outperform a simple CPU C implementation; CMML is aimed at teaching, experimentation, and small-scale benchmarking.

EXAMPLE : -

(Python/NumPy) ***In Python***, the memory allocation, data types, and loop logic are hidden behind a single function call. The user does not know "how" it happens.

```
# The internal logic is hidden (Black Box)
import numpy as np
weights = np.random.randn(10, 1)  # Hidden memory allocation & loop
output  = 1 / (1 + np.exp(-weights)) # Hidden element-wise math
```

(CMML) In C, the user explicitly controls the initialization algorithm and the activation pass, reinforcing an understanding of the computational cost.

```
// The logic is explicit and transparent
matrix *weights = new_gaussian_matrix(10, 1);   // Controlled initialization
matrix *output  = apply_function(weights, sigmoid); // Explicit element-
wise activation
```

The above example excellently showcases the balance of reduced complexity with precise, controlled low-level implementation.

6. Design & Coding

6.1 Core Architecture and Design Philosop hy

6.1.1 **Data Representation** The core of the library is the matrix structure. To maximize cache efficiency and simplify memory management, the library uses a "flattened" 1D array approach rather than a pointer-to-pointer (float**) approach. This reduces memory fragmentation and allows for contiguous memory blocks

Structure Definition:

```
typedef struct matrix {
    int row;
    int col;
    float *value;
} matrix;} matrix;
```

For a 2D array stored in a single continuous 1D memory block, each element is mapped using:

$$\textbf{index} = (\boldsymbol{i} \times \textbf{total\_columns}) + \boldsymbol{j}$$

Example:

If a 2D array is defined as array[6][6] and the element array[2][3] = 5, then in the flattened 1D representation this element is stored at:

$$\textbf{index} = (\mathbf{2} \times \mathbf{6}) + \mathbf{3} = \mathbf{15}$$

So the flattened array location is array[15].

6.1.2 Error Handling Strategy The library uses two distinct strategies for handling errors, depending on the severity and nature of the failure, and both approaches are consistently applied throughout the codebase.

1) Critical Failures (Memory Allocation):

All memory allocation is routed through the malloc_safe(size_t n) function, which checks whether malloc returned NULL.

If allocation fails, the library does not attempt to continue execution; instead, it immediately crashes the program after printing a detailed error message. Errors are reported to stderr using the ERROR_LOG macro, which automatically includes the filename and line number, making the issue easy to trace during debugging.

This fail-fast approach ensures that the program never proceeds with invalid or uninitialized memory and avoids undefined behavior.

2) Logical & Runtime Errors:

Logical errors and invalid arguments are handled gracefully. Most public functions perform argument validation using the CHECK_NULL_MATRIX macro; when a NULL or otherwise invalid argument is detected the macro prints an error to stderr (again using ERROR_LOG) and returns a context-appropriate value (NULL, -1, 0, or nothing for void functions).

This approach prevents crashes for normal argument errors while still providing diagnostic information for debugging via **stderr**.

fatal conditions stop execution immediately, while **ordinary logical mistakes are reported clearly but handled without crashing**. This gives both reliability for low-level operations and flexibility for higher-level ML experimentation.

6.1.3 API Naming Scheme and "_inplace" Suffix

the API follows consistent naming scheming to make function easy to understand, increase readability, maintainability and self-documentation increasing overall user experience

Most functions use a **verb-like prefix** followed by the **object or operation** being performed
E.x :-
new_matrix(), copy_matrix(), multiply_matrix(), and transpose_matrix()

"_inplace" Suffix :-

An important part of the naming scheme is the use of the _inplace suffix, which indicates that the function modifies the matrix directly rather than returning a new matrix.

There are many core function available with _inplace suffix
for example :-
add_matrix(a, b) will create and return a *new* matrix containing the result, and return pointer to it.
add_matrix_inplace(a, b) will store the result directly into a and avoid allocating new memory and return void.

This separation gives users full control over both memory usage and program behavior:

- **Normal versions** (constructive) are safer and easier to reason about because they do not modify the original data.
- **_inplace versions** (destructive) are faster and more memory-efficient, especially for large matrices or inside performance-critical loops.

Overall, the API naming scheme is intentionally designed to be **clear, predictable, and user-friendly**, while the _inplace suffix provides an explicit and easy-to-understand way to distinguish between safe high-level operations and optimized low-level operations.

6.2 Function Module Documentation

6.2.1 **CONSTRUCTORS & DESTRUCTORS →**
(Lifecycle Management)

This module handles the creation, initialization, and destruction of matrix objects. It ensures that all matrices are allocated safely using the library's memory wrappers and prevents memory leaks through proper cleanup routines.

1. `new_matrix()`

- **Prototype:** `matrix* new_matrix(int row, int col);`
- **Description:** The primary constructor. It allocates memory for the matrix structure and the internal value array to hold (row * col) floating-point elements.
- **Logic:** It uses `malloc_safe()` to safely allocate memory and to handle errors.

- **Errors / edge cases:**
    - If allocation fails, `malloc_safe()` prints an error and terminates the program.
    - No explicit runtime validation for non-positive dimensions in the current implementation (caller responsibility).

NOTE:- It does not initialize the matrix(internally it only performs allocation without initialization).

2. free_matrix()

- **Prototype:** `void free_matrix(matrix* m)`
- **Logic:** The destructor function. It safely frees the internal value array first, followed by the structure pointer itself. It includes a NULL check to prevent crashes if called on an already freed or uninitialized pointer.

3. new_gaussian_matrix()

- **Prototype:** `matrix* new_gaussian_matrix(int row, int col)`
- **Description:** Creates a matrix filled with random numbers following a Standard Normal Distribution (mean = 0, std_dev = 1). This is critical for Machine Learning weight initialization.

- **Logic:** It iterates through the matrix and assigns values using the `normal_rand()` helper function, which implements the Box-Muller transform.

```
C
//
// Initialization loop for Gaussian weights
for (int i = 0; i < count; ++i) {
    m->value[i] = normal_rand();
}
```

Note:- The helper function `normal_rand()` internally uses the standard rand() function. Therefore, an explicit invocation of srand() (e.g., `srand(time(NULL))`) in the main program is absolutely needed to generate unique random sequences.

- **Mathematical Formula:** Given two uniformly distributed random numbers U_1 and U_2, the function calculates:

$$Z_0 = \sqrt{-2lnU_1}\, cos(2\pi U_2)$$

$$Z_1 = \sqrt{-2lnU_1}\, sin(2\pi U_2)$$

The code implements this logic to generate the Gaussian noise values.

4. new_random_matrix()

- **Prototype:** `matrix* new_random_matrix(int row, int col, float min, float max)`
- **Description:** Generates a matrix filled with random values uniformly distributed between a specified min and max range.

Note:- `new_random_matrix()` internally uses the standard `rand()` function. Therefore, an explicit invocation of srand() (e.g., `srand(time(NULL))`) in the main program is absolutely needed to generate unique random sequences.

5. eye()

- **Prototype:** `matrix* eye(int n)`
- **Description:** Constructs an "Identity Matrix" of size (n x n). It initializes a zero-filled square matrix and sets the diagonal elements (where i == j) to 1.0.

6. zeros()

- **Prototype:** `matrix* zeros(int row, int col)`
- **Description:** Allocates a new matrix of specific dimensions and initializes every element to 0.0. This is often used to create bias vectors or placeholder buffers.

7. array_matrix()

- **Prototype:** `matrix* array_matrix(float *arr, int row, int col)`
- **Description:** Converts a standard C array (float*) into a library-compatible matrix object.
- **Logic:** It performs a deep copy, meaning the library creates its own separate memory block and copies the values from the input array. This ensures that the original array can be modified or freed without affecting the new matrix.

8. copy_matrix()

- **Prototype:** `matrix* copy_matrix(const matrix *old)`
- **Description:** Creates a deep copy of an existing matrix object. It replicates both the structural metadata (row, col) and the actual data values using memcpy for performance.

6.2.2 **ACCESSORS & UTILITIES →**

(Inspection and Debugging)

This module provides essential tools for inspecting matrix properties, visualizing data, and validating equality. These functions are primarily used for debugging and ensuring data integrity throughout the library's operations.

1. `get_rows()` & `get_cols()`

- **Prototypes:**
  - `int get_rows(matrix* m)`
  - `int get_cols(matrix* m)`
- **Description:** distinct accessor functions that return the number of rows or columns in a given matrix.
- **Safety:** Both functions perform a NULL check on the input pointer. If the matrix is invalid, they return 0 to prevent segmentation faults.

2. `get_matrix_element()`

- **Prototype:** `float get_matrix_element(matrix* m, int r, int c)`
- **Description:** Retrieves the value at a specific row r and column c from the flattened array.
- **Safety & Error Handling:** Unlike standard C array access, this function enforces strict **bounds checking**.
  - If the requested indices are outside the matrix dimensions (r >= row or c >= col), it logs a specific "Index out of bounds" error to stderr and returns NAN (Not A Number) to indicate failure.

3. `print_matrix()`

- **Prototype:** `void print_matrix(matrix *m)`
- **Description:** A visualization utility that iterates through the matrix and prints its contents to the console in a formatted grid. It uses the %g format

specifier to automatically switch between standard and scientific notation for readability.

4. print_shape()

- **Prototype:** `void print_shape(matrix* m)`
- **Description:** A lightweight debugging tool that prints the dimensions of the matrix in the format (rows x cols). Useful for verifying dimensionality before performing matrix multiplication.

5. matrix_check_equal()

- **Prototype:**

  ```
  int matrix_check_equal(matrix* a, matrix* b, float tolerance)
  ```

- **Description:** Compares two matrices for equality. It returns 1 (true) if they are equal and 0 (false) otherwise.
- **Logic (Floating Point Comparison):**
  - First, it checks if the dimensions (row and col) match. If not, it returns 0 immediately.
  - Second, instead of checking for exact equality (==), which is unreliable for floating-point arithmetic, it checks if the difference between elements is within a specified tolerance.

```
// Robust floating-point comparison
if (fabsf(a->value[i] - b->value[i]) > tolerance) {
    return 0; // Not equal
}
```

6.2.3 **DATA MANIPULATION →**

(Setters, Reshaping, Filling)

This module handles the modification of matrix contents and structural transformations. It includes functions to set values, reshape dimensions without changing data, and extract specific sub-regions (slicing) for analysis

1. `set_matrix()` & `fill_matrix()`

- **Prototypes:**
  - `void set_matrix(matrix *m, int r, int c, float value)`
  - `void fill_matrix(matrix *m, float value)`
- **Description:**
  - `set_matrix()` updates a single element at (r, c). It performs strict bounds checking and logs an error if the index is invalid.
  - `fill_matrix()` sets **every** element in the matrix to a specific scalar value (e.g., clearing a matrix to 0.0).

2. `reshape_matrix()` & `reshape_matrix_inplace()`

- **Prototypes:**
  - `matrix* reshape_matrix(const matrix* m, int new_rows, int new_cols)`
  - `int reshape_matrix_inplace(matrix *m, int new_rows, int new_cols)`
- **Description:** changes the dimensions of the matrix (e.g., converting a 4x4 matrix into 2x8) while preserving the underlying data.
- **Validation Logic (Crucial):** Both functions strictly enforce that the total number of elements must remain constant.
  - ***Condition****:* new_rows * new_cols == current_rows * current_cols.

- If this condition is not met, the functions log an "Invalid Matrix Dimension" error and fail (**returning NULL or -1**).

3. `get_row()` & `get_col()`

- **Prototypes:**
  - `matrix* get_row(matrix* m, int r)`
  - `matrix* get_col(matrix* m, int c)`
- **Description:** Extracts a single row (returned as 1 x Cols) or a single column (returned as Rows x 1) **into a new matrix object**. Both include bounds checking to ensure r and c exist.

4. `get_slice()` (Complex Operation)

- **Prototype:**
  - `matrix* get_slice(matrix* m, int r_start, int r_end, int c_start, int c_end)`
- **Description:** Extracts a rectangular sub-region from the matrix. This function is powerful but requires precise parameter usage.
- **Parameter Rules:**
  - **Inclusive Start:** r_start and c_start are the indices of the first element to include.
  - **Exclusive End:** r_end and c_end are the indices **after** the last element to include (similar to Python's list slicing [start:end]).
  - **Constraint:** The slice size (r_end - r_start) must be positive.
- **Algorithm:** The function calculates the memory offset for each row and uses memcpy to copy the relevant segment into the new matrix, ensuring efficient extraction.

```
// Loop to copy slice row-by-row
for (int i = r_start; i < r_end; ++i) {
    // Source pointer arithmetic: Base + (Current Row * Total Cols) + Col Offset
    memcpy(&s->value[(i - r_start) * s->col],
           &m->value[i * col + c_start],
           sizeof(float) * s->col);
}
```

6.2.4 ARITHMETIC OPERATIONS→

(Element-wise Math)

This module implements fundamental element-wise mathematical operations. A key feature of this module is the dual-implementation strategy: every operation is available in both a standard form (returning a new matrix) and an in-place form (modifying the destination matrix) to allow for memory optimization in iterative Machine Learning algorithms.

1. add_matrix() & subtract_matrix()

- **Prototypes:**
  - `matrix* add_matrix(const matrix *a, const matrix *b)`
  - `int add_matrix_inplace(matrix *dest, matrix *src)`
  - `matrix* subtract_matrix(const matrix *a, const matrix *b)`
  - `int subtract_matrix_inplace(matrix *dest, matrix *src)`
- **Description:** Performs element-wise addition ($C_{ij} = A_{ij} + B_{ij}$) or subtraction ($C_{ij} = A_{ij} - B_{ij}$) of two matrices.
  - The Safe variants return a new matrix pointer.
  - The In-Place variants modify the first matrix given in argument directly and return 0 for success or -1 for failure.
- **Validation**: Strictly requires both matrices to have identical dimensions ($Rows_A = Rows_B$ and $Cols_A = Cols_B$)

  If dimensions mismatch, the function logs a dimension error and returns NULL or -1.

2. hadamard_matrix() (Element-wise Product)

- **Prototypes:**
  - `matrix* hadamard_matrix(const matrix *a, const matrix *b)`
  - `int hadamard_matrix_inplace(matrix *dest, matrix *src)`

- **Description**: Computes the Hadamard Product (denoted as $A \odot B$) Unlike standard matrix multiplication (Dot Product), this operation multiplies corresponding elements of two matrices.
- **Mathematical Formula:**

$$C_{ij} = A_{ij} \times B_{ij}$$

- **Usage:** This operation is frequently used in Neural Networks for masking (Dropout layers) or applying forget-gates in LSTM/GRU architectures.
- **Validation**: Strictly requires both matrices to have identical dimensions ($\boldsymbol{Rows_A = Rows_B}$ $\boldsymbol{and}$ $\boldsymbol{Cols_A = Cols_B}$)

  If dimensions mismatch, the function logs a dimension error and returns NULL or -1.

3. scalar_multiply() & scalar_add()

- **Prototypes:**
  - `matrix* scalar_multiply(matrix *a, const float b)`
  - `int scalar_multiply_inplace(matrix *a, const float b)`
  - `matrix* scalar_add(matrix *a, const float b)`
  - `int scalar_add_inplace(matrix *a, const float b)`
- **Description:** Applies a single floating-point operation to every element in the matrix independently.
  - scalar_multiply(): Scales the matrix magnitude
    - (e.g., $A_{ij} = A_{ij} \times 0.5$).
  - scalar_add(): Adds a constant offset to the entire matrix (e.g., shifting values).
    - (e.g., $A_{ij} = A_{ij} + 0.5$).
- **Logic:** The function iterates through the flattened array, modifying each element directly.

```
// Example: Scalar Multiplication Logic
for (int i = 0; i < count; ++i) {
    new->value[i] = a->value[i] * b;
}
```

6.2.5 **MATRIX ALGEBRA→**

(Dot Products, Transpose, Linear Algebra)

A. Core Matrix Operations

1. `multiply_matrix()`

- **Prototype:**
  - `matrix* multiply_matrix(const matrix *a, const matrix *b)`
- **Description:** Computes the standard matrix product (Row-by-Column multiplication) of two matrices.
- **Algorithm:** The function performs the dot product of every row in matrix with every column in matrix .
- **Validation:** Requires that the number of columns in A equals the number of rows in B $(Cols\_A == Rows\_B)$
- **Time Complexity:** $O(M \times N \times P)$ W here the result is an $M \times P$ matrix and N is the common dimension.

2. `dot_product()`

- **Prototype:**

```
float dot_product(const matrix* a, const matrix* b)
```

- **Description:** Calculates the scalar dot product of two vectors.
- **Validation:** Strictly enforces that inputs must be vectors (either $1 \times N$ *or* $N \times 1$). If passed a full matrix, it logs an error and returns NAN.
- **Formula:** $\sum(\boldsymbol{A_i} \times \boldsymbol{B_i})$**.**
- **Complexity:** $O(N)$.

3. `transpose_matrix()`

- **Prototype:** `matrix* transpose_matrix(matrix *a)`
- **Description:** Returns a new matrix where the rows of the original become columns ($B_{ji} = A_{ij}$).

- **Time Complexity:** $O(Rows \times Cols)$

B. Linear Algebra Solvers

4. `determinant_matrix()`

- **Prototype:** `float determinant_matrix(const matrix *m)`
- **Description:** Calculates the scalar determinant of a square matrix.
- **Algorithm(Gaussian Elimination) :** Unlike simple recursive implementations which have factorial complexity ($O(N!)$), this library implements **Gaussian Elimination** to convert the matrix into **Row Echelon Form** (Upper Triangular).
    1. The function iterates through the diagonal pivots.
    2. If a pivot is zero, it scans below to find a non-zero element and calls `swap_rows()`.
    3. It performs row reductions to eliminate values below the pivot.
    4. Once triangular, the determinant is the product of the diagonal elements.
- **Time Complexity:** $\boldsymbol{O(N^3)}$ (Efficient).

```
// Row reduction step in Gaussian Elimination
for (int j = i + 1; j < row; ++j) {
    float entry = k->value[j * col + i];
    add_rows(k, j, i, (-1) * (entry / pivot));
}
```

5. `inverse_matrix()`

- **Prototype:** `matrix* inverse_matrix(matrix* a)`
- **Description:** Computes the multiplicative inverse A^{-1} such that

$$A \times A^{-1} = I.$$

- **Algorithm (Adjoint Method):**The function utilizes the explicit mathematical definition of the inverse:

$$A^{-1} = \frac{1}{|A|} \cdot \text{adj}(A)$$

Where $|A|$ is the determinant and $\text{adj}(A)$ is the Adjoint matrix which are calculated via `adjoint_matrix()` and `determinant_matrix()`.

- **Validation:** Checks if the matrix is square and if the determinant is non-zero (non-singular). If the determinant is 0, inversion is impossible, and it returns NULL.
- **Time Complexity:** $\boldsymbol{O(N^5)}$ (Due to calculating determinants for minors). While computationally expensive, this method is implemented for educational clarity regarding cofactors.

C. Decomposition Tools (Adjoint/Cofactor)

These functions are primarily internal helpers for `inverse_matrix()` but are exposed for educational dissection of linear algebra concepts.

6. `minor_matrix()`

- **Prototype:** `matrix* minor_matrix(matrix* a, int i, int j)`
- **Description:** Returns a sub-matrix of size $(N-1) \times (N-1)$ by removing i row and j column from the original matrix.

7. `cofactor_matrix()`

- **Prototype:** `matrix* cofactor_matrix(matrix* a)`
- **Description:** Creates a matrix where every element C_{ij} is the determinant of the minor at (i, j), multiplied by the alternating sign $(-1)^{i+j}$.

8. `adjoint_matrix()`

- **Prototype:** `matrix* adjoint_matrix(matrix* a)`
- **Description:** Returns the Transpose of the Cofactor matrix. This is the final step before calculating the Inverse.

D. Elementary Row Operations

These functions implement the three fundamental operations of Gaussian Elimination. They modify the matrix **in-place**.

9. `swap_rows()`

- **Prototype:** `void swap_rows(matrix* m, int r1, int r2)`
- **Description:** Swaps the content of row $r1$ and row $r2$.
- **Optimization:** Uses `memcpy()` to swap entire rows of memory at once rather than iterating element-by-element, significantly improving performance for large matrices.

10. `multiply_row()`

- **Prototype:** `void multiply_row(matrix* m, int r, float scalar)`
- **Description:** Multiplies every element in row by a non-zero scalar value ($R_i = kR_i$).

11. `add_rows()`

- **Prototype:**

```
void add_rows(matrix* m, int target_r, int source_r, float scale)
```

- **Description:** Adds a multiple of the source row to the target row $\left(R_{target} = R_{target} + k \cdot R_{source}\right)$. This is the core operation used to eliminate variables in linear solver

6.2.6 **MACHINE LEARNING UTILITIES→**

(Activation, Broadcasting, Initialization Tools)

This module bridges the gap between pure mathematics and Machine Learning applications. It provides the specialized tools necessary for building Neural Networks, including activation function mapping, bias broadcasting, and classification helpers.

1. `broadcast_add() & broadcast_add_inplace()`

- **Prototypes:**
  - `matrix* broadcast_add(matrix* m, matrix* b)`
  - `int broadcast_add_inplace(matrix* m, matrix* b)`
- **Description:** Implements the "Broadcasting" mechanism found in libraries like NumPy. It adds a column vector b to every column of the matrix m.
- **Usage in ML:** This is essential for the "Add Bias" step in a Dense Layer ($Y = WX + b$) . If the output Y represents a batch of data (Neurons × BatchSize), the single bias vector (Neurons × 1) must be added to *every* sample in the batch.
- **Validation:** strictly enforces that the bias b is a Column Vector ($N \times 1$) and that its height matches the matrix rows ($Rows_b = Rows_m$). If mismatch, it logs a "Broadcasting Error".
- **Algorithm**:The function iterates through each row i. It retrieves the bias value b_i and adds it to every column j in that row.

```
// Adding the bias of neuron 'i' to all batch samples 'j'
    float bias = b->value[i];
    for (int j = 0; j < col; ++j){
        c->value[i * col + j] += bias;
    }
```

2. apply_function() & apply_function_inplace()

- **Prototypes:**
  - `matrix* apply_function(matrix* a, float (*f)(float))`
  - `int apply_function_inplace(matrix* a, float (*f)(float))`
- **Description:** A higher-order function that applies a transformation to every element in the matrix.
- **Function Pointers:** It accepts a **function pointer** `float (*f)(float)` as an argument. This allows the user to pass any custom mathematical function (e.g., Sigmoid, ReLU, Tanh) without rewriting the loop logic.
  - *Example:* apply_function(weights, sigmoid)
- **Logic:** It iterates through the flattened array and executes `a->value[i] = f(a->value[i])`.

3. argmax()

- **Prototype:** `int argmax(matrix* a)`
- **Description:** Finds the index of the maximum value in the matrix.
- **Usage:** Used in the final output layer of a Classification Network. For example, in digit recognition (MNIST), it identifies which neuron (0-9) has the highest activation probability.
- **Logic:** It performs a linear scan ($O(N)$) of the matrix. It tracks the `max_value()` and `max_index()`, returning the index corresponding to the highest element found.

7. Small Example/Demonstration

7.1 Machine Learning workflows Example: -

To illustrate the library's capability to handle Machine Learning workflows, the following example simulates the **Forward Pass of a Single Dense Layer**. It demonstrates weight initialization, matrix multiplication, bias broadcasting, and non-linear activation.

The Scenario:

- **Input (X):** A batch of **3 samples**, each having **4 features** (Dimensions: 4×3).
- **Layer:** A dense layer with **2 Neurons**.
- **Operation:** $Output = \text{Sigmoid}\big((W \cdot X) + b\big)$

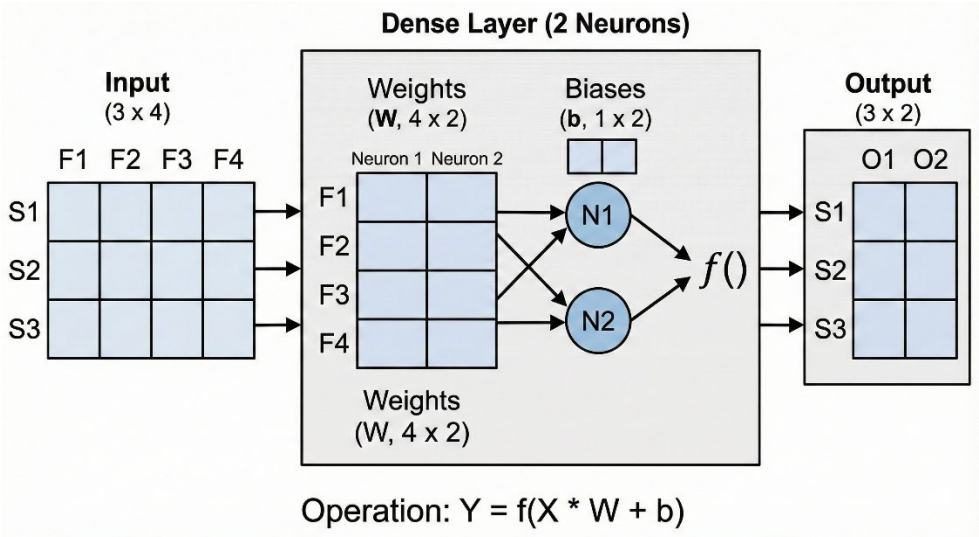

Figure: - Visual representation of the Dense Layer Forward Pass (3 Samples, 4 Features, 2 Neurons). *Note: This diagram illustrates the data flow using standard row-vector notation ($Input \times Weights$). The CMML library implementation utilizes the equivalent column-vector notation ($Weights \times Input$) typically found in linear algebra textbooks.*

The Solution Code:

```c
#include "matrix.h"
#include <stdio.h>
#include <math.h>

// 1. Define an Activation Function (Sigmoid)
float sigmoid(float x) {
    return 1.0f / (1.0f + expf(-x));
}

int main() {
    // ------------------------------------------------------------
    // STEP 1: LOAD DATA (Batch of 3 samples, 4 features each)
    // ------------------------------------------------------------
    // We use Column-Vectors: 4 Rows (Features) x 3 Cols (Samples)
    float data[] = {
        1.0, 2.0, 3.0,  // Feature 1 values
        0.5, 0.5, 0.0,  // Feature 2 values
        -1.0, 0.0, 1.0, // Feature 3 values
        0.0, 1.0, 0.0   // Feature 4 values
    };
    matrix *X = array_matrix(data, 4, 3);



    // ------------------------------------------------------------
    // STEP 2: INITIALIZE LAYER (Weights & Biases)
    // ------------------------------------------------------------
    // Weights: 2 Neurons x 4 Inputs. Initialized with Gaussian Noise.
    matrix *W = new_gaussian_matrix(2, 4);

    // Bias: 2 Neurons x 1. Initialized to zeros.
    matrix *b = zeros(2, 1);
    // Let's manually set biases to 0.5 for demonstration
    fill_matrix(b, 0.5f);
```

```
    // ------------------------------------------------------------
    // STEP 3: FORWARD PASS (Linear Step)
    // ------------------------------------------------------------
    // Z = W * X  (Result is 2x3 matrix)
    matrix *Z = multiply_matrix(W, X);
    // Add Bias (Broadcasting 2x1 bias across 3 columns of Z)
    broadcast_add_inplace(Z, b);

    // ------------------------------------------------------------
    // STEP 4: ACTIVATION (Non-Linear Step)
    // ------------------------------------------------------------
    // A = Sigmoid(Z)
    matrix *A = apply_function(Z, sigmoid);

    // ------------------------------------------------------------
    // STEP 5: RESULTS & CLEANUP
    // ------------------------------------------------------------
    printf("Input Data (4 Features x 3 Samples):\n");
    print_matrix(X);

    printf("Layer Output (Probability Scores for 2 Neurons):\n");
    print_matrix(A);

    free_matrix(X); free_matrix(W); free_matrix(b);
    free_matrix(Z); free_matrix(A);
    return 0;
}
```

Results:

```
Input Data (4 Features x 3 Samples):
 1   2   3
.5  .5   0
-1   0   1
 0   1   0

Layer Output (Probability Scores for 2 Neurons):
0.014296   0.0001771   0.000129
0.166343   0.0854908   0.192811


[Execution Time: 00:00:00.0156317]
```

7.2 Performance Benchmark Example: -

This demonstration highlights the library's efficiency by performing a computationally expensive $O(N^3)$ matrix multiplication on a large dataset (1,000,000 elements per matrix).

The Tests Code:

```c
#include <stdio.h>
#include <time.h>      // Required for benchmarking
#include "matrix.h"

int main() {
    int N = 1000; // Defines a 1000x1000 matrix (1 Million elements)

    printf("------------------------------------------------\n");
    printf("[BENCHMARK] Stress Test: Matrix Multiplication\n");
    printf("------------------------------------------------\n");
    printf("Initializing two %dx%d matrices...\n", N, N);

    // 1. Setup: Allocate heavy memory (approx 8MB each)
    matrix *A = new_random_matrix(N, N, -1.0, 1.0);
    matrix *B = new_random_matrix(N, N, -1.0, 1.0);

    printf("Start Calculation...\n");

    // 2. Start Timer
    clock_t start = clock();
```

```
    // 3. Run Computation (Standard O(N^3) Algorithm)
    // For N=1000, this performs 2 Billion floating-point operations
    matrix *C = multiply_matrix(A, B);

    // 4. Stop Timer
    clock_t end = clock();
    double time_taken = (double)(end - start) / CLOCKS_PER_SEC;

    printf("Done!\n");
    printf("Time Elapsed: %.4f seconds\n", time_taken);

    // 5. Cleanup
    free_matrix(A);
    free_matrix(B);
    free_matrix(C);

    return 0;
}
```

Results: -

```
------------------------------------------------
[BENCHMARK] Stress Test: Matrix Multiplication
------------------------------------------------
Initializing two 1000x1000 matrices...
Start Calculation...
Done!
Time Elapsed: 4.4910 seconds
```

Note: This performance metric was obtained running on a **single CPU core** (Single-Threaded). The execution utilized standard scalar arithmetic, leaving the multi-core parallel processing capabilities of the hardware unutilized.

8. Future Enhancements

To evolve **CMML** from an educational prototype into a robust, industry-applicable framework, the following development roadmap is proposed.

8.1 Expansion of Machine Learning Capabilities

- **Backpropagation & Training Engine:** The immediate next step is to implement the backward pass (Chain Rule) and automatic gradient calculation. This includes adding derivative definitions for activation functions and standard Loss Functions (MSE, Cross-Entropy).
- **Optimizers:** Implementation of standard optimization algorithms such as **Stochastic Gradient Descent (SGD)** and **Adam** to automate weight updates during training loops.

8.2 Migration to C++ for Developer Experience

- **Operator Overloading:** Transitioning the codebase to C++ will allow for operator overloading (e.g., using C = A * B instead of multiply_matrix(A, B)). This significantly improves code readability for mathematical researchers.
- **RAII Memory Management:** Utilizing C++ Destructors and Smart Pointers will automate memory deallocation, eliminating the risk of memory leaks inherent in manual C implementations.
- **Templates:** Implementing template classes to support multiple data types (double for high precision, int8 for quantized edge-computing models) using a single codebase.

8.3 Advanced Neural Architecture Support

- **Convolutional Operations:** Extending the library to support Conv2D operations and MaxPooling. This is essential for supporting Convolutional Neural Networks (CNNs), which are the industry standard for Image Processing and Computer Vision.

8.4 Ecosystem Integration

- **Python Bindings:** Developing a **Python Wrapper API** (similar to NumPy's backend). This would allow users to write high-level Python code while leveraging the raw performance of the CMML C-engine for computation.
- **Model Persistence:** Adding File I/O functionality to serialize (save) trained weight matrices to disk, enabling the deployment of pre-trained models in production environments.

9. Conclusion

The CMML (C-Language Matrix & Machine Learning Library) project shows that it's possible to break down and use high-level Machine Learning ideas in low-level C. This library gives you a clear, open view of the mathematical engines that power AI by getting rid of the heavy abstractions that are common in modern frameworks.

The project's main goals of being open and clear about its goals have been met. It has a useful, lightweight toolkit with no dependencies that lets users follow the execution of complicated algorithms, like dynamic memory allocation and Gaussian broadcasting, without the "black box" obscurity of interpreted languages.

The current implementation puts algorithmic readability ahead of optimization on a large scale, but the benchmarks show that the system works well and is reliable for its intended use. The modular architecture created in this project is a strong base that shows that making a Machine Learning system from scratch is not only possible but also the best way to learn how AI works in the real world.