Object-Oriented Programming

BLG 252E

Exercise 3

Res. Assist. M. Alpaslan Tavukçu
tavukcu22@itu.edu.tr

Instructors:
Assoc. Prof. Dr. Feza BUZLUCA
Assoc. Prof. Dr. Cihan TOPAL
Assis. Prof. Dr. Sanem KABADAYI

Faculty of Computer and Informatics Engineering
Department of Computer Engineering

1. Neural Network Forward Pass Library

In this exercise, you will implement a simplified neural network library in C++. This project will strengthen your understanding of both object-oriented programming principles.

No prior knowledge of neural networks is required to complete this exercise. You can implement the specified components by following the technical requirements without deep understanding of neural networks. However, gaining domain knowledge will enhance your implementation and understanding, so additional resources are provided below:

- **3Blue1Brown: Neural Networks**:https://www.youtube.com/watch?v=aircAruvnKk

1.1. Background

1.1.1. Forward Flow in Neural Networks

Forward flow (or forward propagation) is the fundamental process by which information travels through a neural network to generate a prediction or output. It represents the core operational sequence when using a trained neural network.

Steps of Forward Flow:

1. **Input Layer**
    - Receives the initial data values
    - Passes these values unchanged to the first processing layer
2. **Hidden Layers**
   Each hidden layer transforms its input data through four sequential operations:
    - **Weighting**: Multiplies each input value by corresponding weight values
    - **Summing**: Combines all weighted inputs into a single value for each neuron
    - **Biasing**: Adds a constant value (bias) to each sum
    - **Activation**: Applies a non-linear function to determine the neuron's output signal
3. **Layer-to-Layer Transition**
    - Output values from each layer serve as input values to the subsequent layer
    - Each successive layer performs further transformations on the data
4. **Output Layer**

- Performs the final transformation and produces the network's result
- The format of this output depends on the problem being solved (classification, regression, etc.)

1.1.2. Forward Flow Algorithm

The following pseudo-algorithm describes how forward flow operates using the classes defined in this library:

```
// Forward Flow Algorithm

// 1. Initialize with input data
Create a Matrix containing the input values

// 2. Process through the Network
Network.forward(input_matrix):
    If network contains no layers:
        Return input_matrix unchanged

    // Process through first layer
    current_output = first_layer.forward(input_matrix)

    // Process through remaining layers sequentially
    For each subsequent layer in the network:
        current_output = current_layer.forward(current_output)

    Return current_output as the final result

// 3. Layer processing
Layer.forward(input_matrix):
    // Apply weights to input values
    weighted_input = input_matrix * weights_matrix

    // Apply bias and activation function
    Create result_matrix with same dimensions as weighted_input
    For each position [i,j] in weighted_input:
        Apply bias: value = weighted_input[i][j] + bias[j]
        Apply activation: result_matrix[i][j] = activation_function(
            value)

    Return result_matrix

// 4. Activation function application
Activation.function(value):
    If activation type is "relu":
        Return maximum of (0, value)
    Else if activation type is "sigmoid":
        Return 1 / (1 + e^(-value))
```

In this process:

- Matrix class handles mathematical operations
- Layer class applies transformations to data
- Activation class provides non-linear functions
- Network class coordinates sequential processing through all layers

1.1.3. UML Diagram - Relationships Between Objects

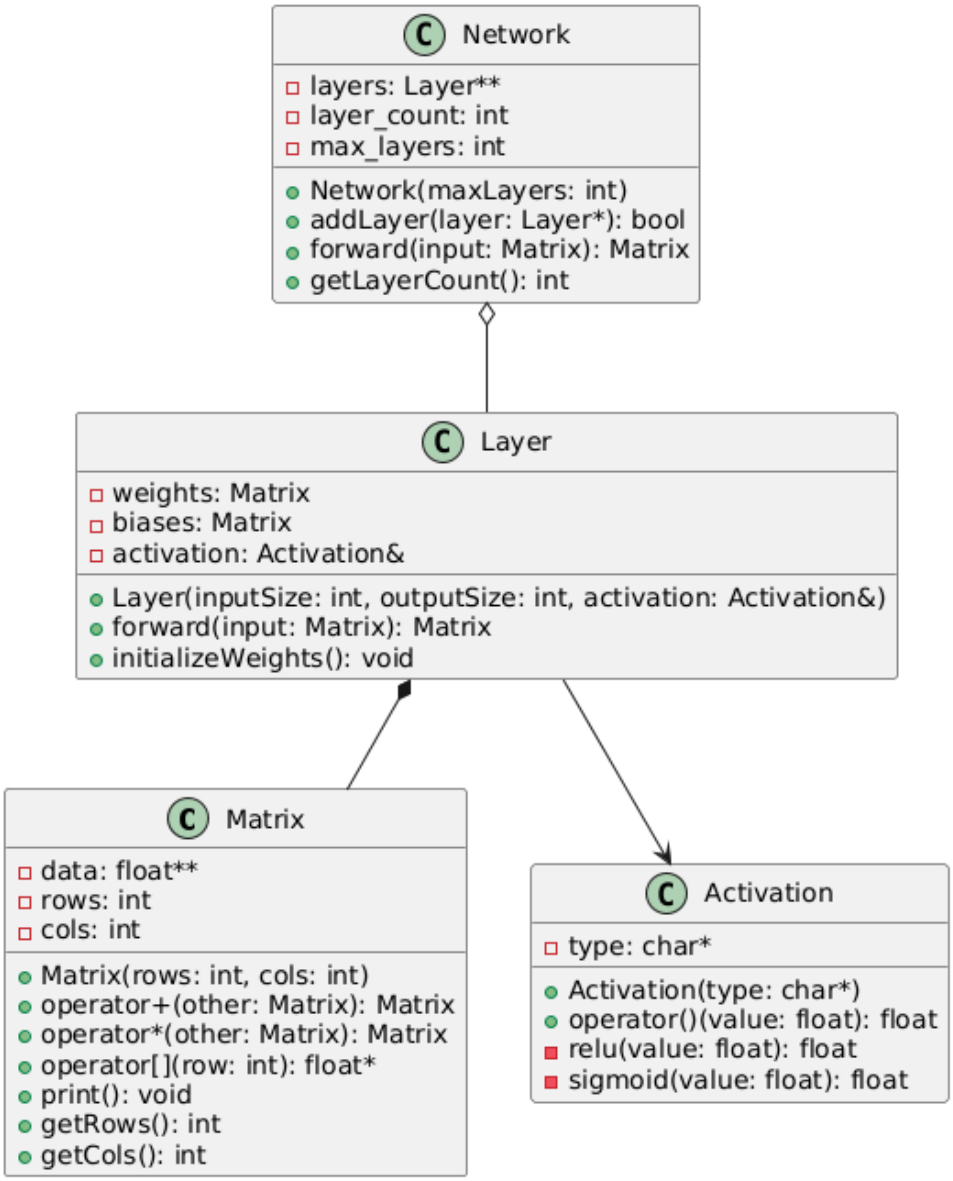

Figure 1.1: UML Diagram of the NN Library

2. Implementation

2.1. Matrix Class

The Matrix class provides a mathematical structure for storing and manipulating two-dimensional numerical data.

2.1.1. Attributes

- **data**: Two-dimensional array storing floating-point values. Each element data[i][j] represents the value at row i, column j.
- **rows**: Integer value indicating the number of rows in the matrix.

  > **Note**: Used for dimensional validation and bounds checking during operations.

- **cols**: Integer value indicating the number of columns in the matrix.

  > **Note**: Used for dimensional validation and bounds checking during operations.

2.1.2. Methods

- **Constructor**: Creates a new matrix with specified dimensions and initializes all values to zero.

  > **Note**: Requires proper memory allocation for the two-dimensional structure.

- **operator+**: Adds corresponding elements of two matrices, returning a new result matrix.

  > **Note**: Matrices must have identical dimensions for addition to be valid.

- **operator***: Performs standard matrix multiplication following linear algebra rules.

  > **Note**: For multiplication of matrices A and B, A's column count must equal B's row count.

- **operator[]**: Enables access to individual elements using standard notation: matrix[row][column].

  > **Note**: Implemented as a two-step indexing operation returning first a row, then an element.

- **print()**: Displays matrix contents in a readable format.

  Note: Useful for debugging and visualizing matrix data.

2.2. Activation Class

The Activation class implements non-linear functions that enable neural networks to learn complex patterns.

2.2.1. Attributes

- **type**: String identifier specifying which activation function to use.

  Note: Common types include "relu" and "sigmoid", each serving different purposes.

2.2.2. Methods

- **Constructor**: Creates an activation function object with the specified function type.

  Note: Initializes the internal state to use the correct function implementation.

- **operator()**: Applies the selected activation function to an input value.

  Note: Using operator overloading allows the object to be used like a mathematical function.

- **relu()**: Implements the Rectified Linear Unit function: f(x) = max(0, x).

  Note: Returns the input value for positive inputs, zero for negative inputs.

- **sigmoid()**: Implements the Sigmoid function: f(x) = $\frac{1}{1+e^{-x}}$.

  Note: Produces a value between 0 and 1, useful for representing probabilities.

2.3. Layer Class

The Layer class represents a single processing unit within the neural network, containing parameters and transformation logic.

2.3.1. Attributes

- **weights**: Matrix storing connection strengths between this layer and the previous layer.

  Note: Dimensions are [inputSize $\times$ outputSize], representing connections between layers.

- **biases**: Matrix storing offset values for each neuron in this layer.

  Note: Typically a [1 $\times$ outputSize] matrix with one value per output neuron.

- **activation**: Reference to an Activation object defining the non-linear function to apply.

  Note: The layer uses but does not own this object

2.3.2. Methods

- **Constructor**: Creates a layer with specified input size, output size, and activation function.

  Note: Initializes weights and biases matrices with appropriate dimensions.

- **forward()**: Transforms input data by applying weights, biases, and the activation function.

  Note: Implements the formula: output = activation(input $\times$ weights + biases).

- **initializeWeights()**: Sets initial weight values to small random numbers.

2.4. Network Class

The Network class coordinates multiple layers to form a complete neural network.

2.4.1. Attributes

- **layers**: Array of Layer pointers representing the sequence of processing units.

  Note: Stores references to existing Layer objects without taking ownership.

- **layer_count**: Integer tracking how many layers have been added to the network.

  Note: Used for bounds checking and iteration through the layer sequence.

2.4.2. Methods

- **addLayer()**: Incorporates an existing Layer object into the network sequence.

  Note: Maintains the correct order of layers for sequential processing.

- **forward()**: Processes input data through all network layers in sequence.

  Note: Passes each layer’s output as input to the next layer, returning final results.

- **getLayerCount()**: Returns the number of layers in the network.

  Note: Provides information about the network’s structure.

3. Example Usage

```cpp
#include <iostream>
#include "../include/Matrix.h"
#include "../include/Activation.h"
#include "../include/Layer.h"
#include "../include/Network.h"

int main()
{
    // Create activation functions
    Activation relu{"relu"};
    Activation sigmoid{"sigmoid"};

    // Create a simple neural network for XOR problem
    // Input layer: 2 neurons
    // Hidden layer: 3 neurons with ReLU activation
    // Output layer: 1 neuron with Sigmoid activation

    // Create layers
    Layer hiddenLayer{2, 3, relu};
    Layer outputLayer{3, 1, sigmoid};

    // Create network with a maximum of 2 layers and add layers
    Network network{2};
    network.addLayer(&hiddenLayer);
    network.addLayer(&outputLayer);

    std::cout << "Neural Network with " << network.getLayerCount() << "
        layers created." << std::endl;

    // Create input data for XOR problem
    // [0,0], [0,1], [1,0], [1,1]
    Matrix input{4, 2};
    input[0][0] = 0.0f; input[0][1] = 0.0f;
    input[1][0] = 0.0f; input[1][1] = 1.0f;
    input[2][0] = 1.0f; input[2][1] = 0.0f;
    input[3][0] = 1.0f; input[3][1] = 1.0f;

    std::cout << "Input data:" << std::endl;
    input.print();

    // Forward pass through the network
    Matrix output = network.forward(input);

    std::cout << "Network output:" << std::endl;
    output.print();
```

```
45
46      std::cout << "Note: Since weights are randomly initialized, the
            output won't match XOR truth table yet." << std::endl;
47      std::cout << "Training would be needed to adjust weights for correct
            predictions." << std::endl;
48
49      return 0;
50  }
```