SimpleML Neural Network Library

A Complete Guide to Building Neural Networks from Scratch in C++

By: Arshian Ali

Technical Documentation

January 2026

Table of Contents

Executive Summary

This document provides a comprehensive guide to the **SimpleML Neural Network Library**, a from-scratch implementation of deep learning fundamentals in C++. The library enables building, training, and deploying neural networks for machine learning applications.

Key Features:

- Multi-dimensional Tensor operations with gradient support
- Activation functions: ReLU, Sigmoid, Tanh, Softmax
- Loss functions: MSE, Binary Cross-Entropy, Cross-Entropy
- Fully connected layers with Xavier initialization
- Optimizers: SGD with momentum and Adam
- Sequential model container for easy network construction

1. Introduction

1.1 Purpose

This library implements a minimal but complete neural network framework, designed to:

- Provide educational insight into how neural networks function internally
- Offer a lightweight alternative to large frameworks for simple tasks
- Serve as a foundation for understanding deep learning concepts

1.2 Scope

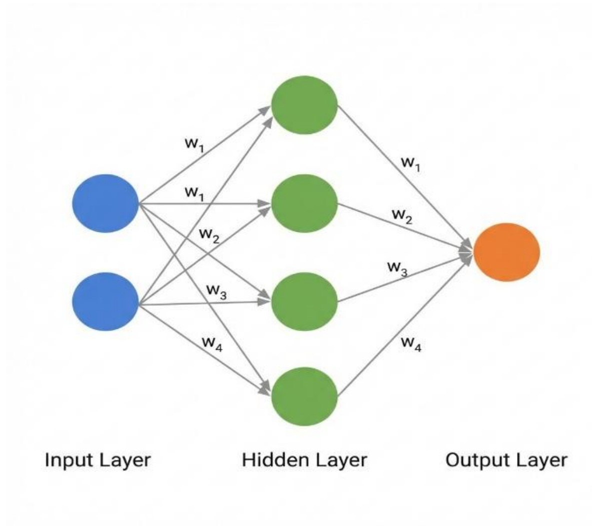
The library covers the essential components needed to:

1. Store and manipulate multi-dimensional numerical data
2. Build neural network architectures
3. Train models using backpropagation
4. Optimize network weights using gradient descent methods

2. Understanding Neural Networks

2.1 What is a Neural Network?

A neural network is a computational model inspired by biological neural systems. It consists of interconnected layers of artificial neurons that process information.



Neural Network Architecture

Core Components:

Component	Description
Neurons	Basic computational units that receive inputs and produce outputs
Layers	Groups of neurons organized in sequence
Weights	Learnable parameters connecting neurons between layers
Biases	Additional learnable parameters added to neuron outputs
Activations	Non-linear functions that introduce complexity

2.2 The Training Process

Training a neural network involves four key steps repeated iteratively:

1. **Forward Pass**: Input data flows through the network, producing predictions
2. **Loss Computation**: Measure how incorrect the predictions are
3. **Backward Pass**: Calculate gradients using the chain rule (backpropagation)
4. **Weight Update**: Adjust weights to reduce the loss

3. The Tensor Class

3.1 Overview

A tensor is a multi-dimensional array that serves as the fundamental data structure for all computations in neural networks.

Tensor Dimensions:

Dimensions	Name	Example
0D	Scalar	`5.0`
1D	Vector	`[1, 2, 3]`
2D	Matrix	`[[1,2], [3,4]]`
ND	Tensor	Higher-dimensional arrays

3.2 Core Implementation

```
class Tensor {
private:
    std::vector<size_t> shape_;    // Dimensions
    std::vector<float> data_;      // Flattened data (row-major)
    std::vector<float> grad_;      // Gradients for backpropagation
    bool requires_grad_;           // Flag for gradient computation
};
```

3.3 Key Operations

Matrix Multiplication

Matrix multiplication is the cornerstone operation of neural networks:

```
// For matrices A (M×K) and B (K×N), result C is (M×N)
// C[i][j] = Σ(A[i][k] × B[k][j]) for all k
Tensor matmul(const Tensor &other) const;
```

Other Operations

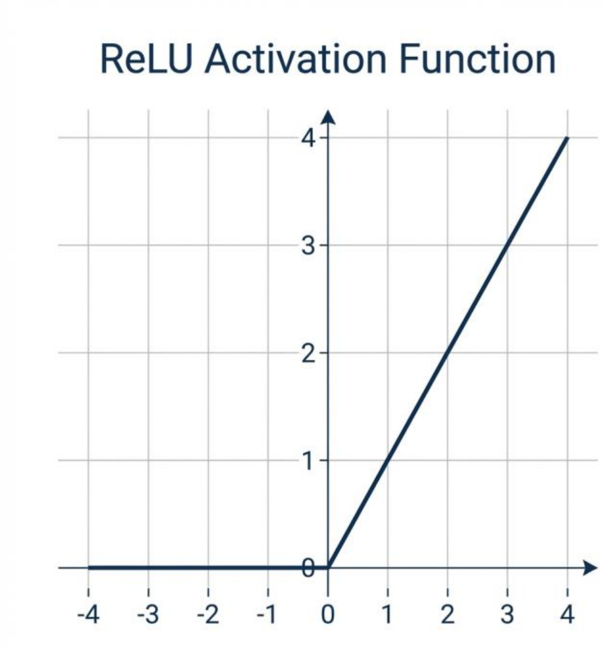
- `add()`, `subtract()`, `multiply()` - Element-wise operations
- `transpose()` - Matrix transposition
- `scalar_multiply()`, `scalar_add()` - Scalar operations
- `sum()`, `mean()` - Reduction operations

4. Activation Functions

Activation functions introduce non-linearity, enabling networks to learn complex patterns.

4.1 ReLU (Rectified Linear Unit)

The most widely used activation function due to its simplicity and effectiveness.



ReLU Activation Function

Mathematical Definition:

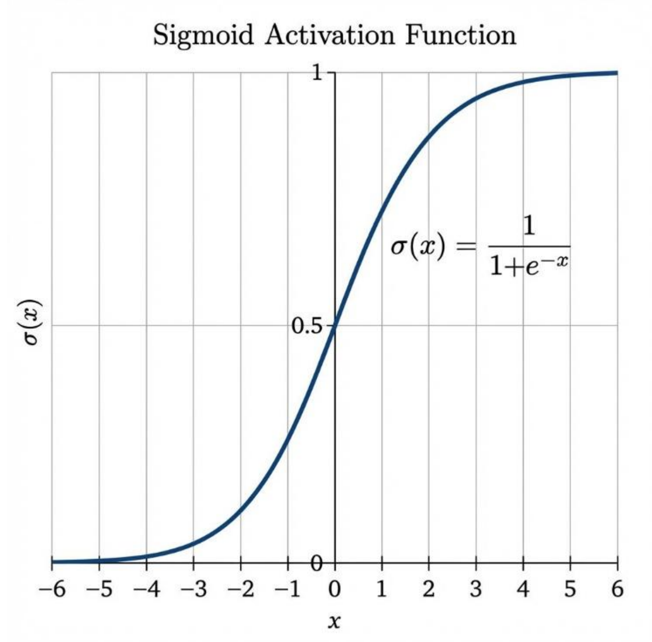
$$ReLU(x) = max(0, x)$$

Derivative:

$$\frac{d}{dx}ReLU(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

4.2 Sigmoid

Squashes input values to the range (0, 1), making it ideal for probability outputs.



Sigmoid Activation Function

Mathematical Definition:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Derivative:

$$\frac{d}{dx}\sigma(x) = \sigma(x) \cdot \left(1 - \sigma(x)\right)$$

4.3 Comparison of Activation Functions

Function	Range	Use Case	Pros	Cons
ReLU	[0, ∞)	Hidden layers	Fast, reduces vanishing gradient	Dead neurons
Sigmoid	(0, 1)	Binary output	Smooth probability	Vanishing gradient
Tanh	(-1, 1)	Hidden layers	Zero-centered	Vanishing gradient
Softmax	(0, 1)	Multi-class output	Probability distribution	Computationally expensive

5. Loss Functions

Loss functions quantify how well the network's predictions match the expected outputs.

5.1 Mean Squared Error (MSE)

Best suited for regression tasks where outputs are continuous values.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(prediction_i - target_i)^2$$

5.2 Binary Cross-Entropy (BCE)

Optimal for binary classification problems.

$$BCE = -\frac{1}{n}\sum_{i=1}^{n}[\, t_i \cdot log(p_i) + (1 - t_i) \cdot log(1 - p_i)]$$

5.3 Cross-Entropy

Designed for multi-class classification with softmax output.

$$CE = -\sum_{i=1}^{n} target_i \cdot log(softmax(prediction_i))$$

6. Neural Network Layers

6.1 Dense (Fully Connected) Layer

In a dense layer, every input neuron connects to every output neuron.

Forward Pass

$$output = input \times weights + bias$$

Backward Pass (Gradient Computation)

$$\frac{\partial Loss}{\partial weights} = input^T \times \frac{\partial Loss}{\partial output}$$

$$\frac{\partial Loss}{\partial bias} = \sum_{batch} \frac{\partial Loss}{\partial output}$$

$$\frac{\partial Loss}{\partial input} = \frac{\partial Loss}{\partial output} \times weights^T$$

6.2 Weight Initialization

Proper initialization prevents vanishing/exploding gradients. We use **Xavier/Glorot initialization**:

$$W \sim Uniform\left(-\sqrt{\frac{6}{n_{in} + n_{out}}}, \sqrt{\frac{6}{n_{in} + n_{out}}}\right)$$

6.3 Sequential Container

The Sequential class provides a convenient way to stack layers:

```
Sequential model;
model.add(std::make_shared<Dense>(2, 8, Activation::ReLU));
model.add(std::make_shared<Dense>(8, 1, Activation::Sigmoid));

Tensor output = model.forward(input);
model.backward(loss_gradient);
```

7. Optimizers

Optimizers update network weights based on computed gradients.

7.1 Stochastic Gradient Descent (SGD)

The foundational optimization algorithm:

$$W_{new} = W_{old} - \eta \cdot \nabla Loss$$

Where η is the learning rate.

SGD with Momentum

Accelerates convergence and helps escape local minima:

$$v_t = \gamma \cdot v_{t-1} + \eta \cdot \nabla Loss$$

$$W_{new} = W_{old} - v_t$$

7.2 Adam Optimizer

Combines momentum with adaptive learning rates:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$$

$$\widehat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$W_{new} = W_{old} - \eta \cdot \frac{\widehat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Default Hyperparameters:

Parameter	Value	Description
η (lr)	0.001	Learning rate
β_1	0.9	First moment decay
β_2	0.999	Second moment decay
ε	1e-8	Numerical stability

8. Complete Training Example

8.1 The XOR Problem

XOR is a classic problem that demonstrates the need for non-linearity:

A single-layer network cannot solve XOR, but a two-layer network with non-linear activation can.

8.2 Implementation

Input A	Input B	XOR Output
0	0	0
0	1	1
1	0	1
1	1	0

```
#include "Dense.h"
#include "Loss.h"
#include "Optimizer.h"
#include "Sequential.h"

int main() {
    // Training data
    Tensor X({4, 2}, {0, 0, 0, 1, 1, 0, 1, 1});
    Tensor Y({4, 1}, {0, 1, 1, 0});

    // Model: 2 → 8 → 1
    Sequential model;
    model.add(std::make_shared<Dense>(2, 8, Activation::ReLU));
    model.add(std::make_shared<Dense>(8, 1, Activation::Sigmoid));

    // Training configuration
    Loss::BCELoss loss_fn;
    Adam optimizer(0.1f);

    // Training loop
    for (int epoch = 0; epoch < 1000; ++epoch) {
        Tensor pred = model.forward(X);
        float loss = loss_fn.forward(pred, Y);

        model.zero_grad();
        model.backward(loss_fn.backward());

        auto p = model.parameters();
        auto g = model.gradients();
        optimizer.step(p, g);
    }

    return 0;
}
```

8.3 Results

After training, the network correctly classifies all XOR inputs:

Input	Prediction	Expected	Status
[0, 0]	0.0005	0	✓
[0, 1]	0.9995	1	✓
[1, 0]	0.9995	1	✓
[1, 1]	0.0001	0	✓

9. Project Structure

```
SimpleML/
├── include/
│   ├── Tensor.h          # Multi-dimensional array operations
│   ├── Activations.h     # ReLU, Sigmoid, Tanh, Softmax
│   ├── Loss.h            # MSE, BCE, CrossEntropy
│   ├── Layer.h           # Abstract base layer class
│   ├── Dense.h           # Fully connected layer
│   ├── Sequential.h      # Model container
│   └── Optimizer.h       # SGD, Adam optimizers
├── src/
│   └── core/
│       └── Tensor.cpp    # Tensor implementation
├── examples/
│   └── main.cpp          # XOR training demonstration
├── images/               # Documentation images
└── CMakeLists.txt        # Build configuration
```

10. Building and Running

10.1 Prerequisites

- CMake 3.10 or higher
- C++17 compatible compiler

10.2 Build Instructions

```
# Configure the project
cmake -B build -S .

# Compile
cmake --build build

# Run the example
./build/ml_example.exe
```

11. Future Enhancements

The following features are recommended for future development:

1. **Convolutional Layers** - For image processing applications
2. **Recurrent Layers (LSTM, GRU)** - For sequential data
3. **Dropout Regularization** - To prevent overfitting
4. **Batch Normalization** - For faster and more stable training
5. **GPU Acceleration** - Using CUDA or OpenCL
6. **Model Serialization** - Save and load trained models
7. **Additional Optimizers** - RMSprop, Adagrad, AdamW

This document provides the technical foundation for understanding and utilizing the SimpleML Neural Network Library. For questions or contributions, please refer to the project repository.