Comprehensive Analysis of my Custom Neural Network Implementation in C++

by 000x999

September 22, 2024

Contents

1 Introduction

This document provides an in-depth analysis of my custom-built neural network implementation in C++. The neural network is designed to handle multi-layer perceptron architectures and uses backpropagation for training.

1.1 What is a Neural Network?

A neural network is a computational model inspired by the biological neural networks that constitute animal brains. At its core, a neural network is a system of interconnected nodes, or "neurons," that work together to process information and learn patterns from data.

Neural networks are designed to recognize complex patterns and solve problems that are difficult for traditional algorithms. They excel at tasks such as image and speech recognition, natural language processing, and decision making.

1.2 How Neural Networks Work

Neural networks operate on a principle of transforming input data through a series of layers to produce an output. The process can be broken down into several key steps:

1. **Input**: Data is fed into the network through the input layer.
2. **Weighted Sum**: Each neuron receives inputs from the previous layer, multiplies each input by a weight, and sums these values.
3. **Activation**: The weighted sum is passed through an activation function, which introduces non-linearity into the model.
4. **Propagation**: The output of each neuron becomes an input to neurons in the next layer.
5. **Learning**: The network compares its output to the desired output and adjusts its weights to minimize the error.

1.3 What is a Neuron?

In the context of artificial neural networks, a neuron (also called a node or unit) is the fundamental building block. It's a computational unit that:

- Receives one or more inputs
- Applies weights to these inputs
- Sums the weighted inputs
- Passes this sum through an activation function
- Produces an output

Mathematically, we can represent the operation of a neuron as:

$$y = f(\sum_{i=1}^{n} w_i x_i + b)$$

Where:

- y is the output
- f is the activation function
- w_i are the weights
- x_i are the inputs
- b is a bias term
- n is the number of inputs

2 Neural Network Structure

2.1 Network Topology

The network topology is defined using a vector of unsigned integers. Here's how it's implemented in the `Net` constructor:

```
Net::Net(const std::vector<unsigned>& topology) {
    uint16_t numLayers = static_cast<uint16_t>(topology.size());
    for (uint16_t layerNum = 0; layerNum < numLayers; ++layerNum) {
        _layers.push_back(Layer());
        uint16_t numOutputs = layerNum == topology.size() - 1 ? 0 : topology[layerNum + 1];

        for (uint16_t neuronNum = 0; neuronNum <= topology[layerNum]; ++neuronNum) {
            _layers.back().push_back(Neuron(numOutputs, neuronNum));
        }

        _layers.back().back().SetOutputVal(1.0);
    }
}
```

Implementation Details:

- The constructor takes a vector of unsigned integers (`topology`) as input, where each element represents the number of neurons in a layer.
- It uses a nested loop structure to create layers and neurons.
- The outer loop iterates over each layer, while the inner loop creates neurons for that layer.
- An extra neuron is added to each layer (except the output layer) to serve as a bias neuron.

Memory Allocation:

- The `_layers` vector is dynamically resized as layers are added using `push_back()`.
- Each `Layer` is a vector of `Neuron` objects, which are also dynamically allocated.
- This approach allows for flexible network sizes but may lead to fragmented memory if many networks are created and destroyed.

Code Dynamics:

- The number of outputs for each neuron is determined by the number of neurons in the next layer.
- The last neuron added to each layer (except the output layer) is set as a bias neuron with a fixed output of 1.0.

2.2 Neuron Structure

Each neuron is represented by the `Neuron` class:

```
struct Neuron {
private:
    double _outputVal;
    std::vector<Connections> _outputWeights;
    uint16_t _myIndex;
    double _gradient;

public:
    Neuron(uint16_t numOutputs, uint16_t myIndex);
    void SetOutputVal(double val) { _outputVal = val; }
    double GetOutputVal() const { return _outputVal; }
    void FeedNeurons(const Layer& prevLayer);
    void CalcOutputGradients(double targetVal);
    void CalcHiddenGradients(const Layer& nextLayer);
    void UpdateWeights(Layer& prevLayer);
    // ... other methods ...
};
```

Implementation Details:

- The `Neuron` class encapsulates all properties and behaviors of a single neuron.
- It stores its output value, connections to the next layer, its index in the current layer, and its gradient for backpropagation.
- The class provides methods for feeding forward, calculating gradients, and updating weights.

Memory Allocation:

- The `_outputWeights` vector is dynamically allocated, allowing for flexible connections between layers.
- The size of each `Neuron` object will depend on the number of its output connections.

Code Dynamics:

- The `FeedNeurons` method is called during forward propagation to compute the neuron's output.
- `CalcOutputGradients` and `CalcHiddenGradients` are used during backpropagation to compute error gradients.
- `UpdateWeights` adjusts the neuron's weights based on the computed gradients.

Performance Considerations:

- The use of inline functions (like `SetOutputVal` and `GetOutputVal`) can reduce function call overhead.

2.3 Weight Updates

The weight update process is implemented in the `Neuron::UpdateWeights` method:

```
void Neuron::UpdateWeights(Layer& prevLayer) {
    const double eta = 0.15;  // learning rate
    const double alpha = 0.5; // momentum

    for (uint16_t n = 0; n < prevLayer.size(); ++n) {
        Neuron& neuron = prevLayer[n];
        double oldDeltaWeight = neuron._outputWeights[_myIndex].deltaWeight;
        double newDeltaWeight =
            eta * neuron.GetOutputVal() * _gradient
            + alpha * oldDeltaWeight;

        neuron._outputWeights[_myIndex].deltaWeight = newDeltaWeight;
        neuron._outputWeights[_myIndex].weight += newDeltaWeight;
    }
}
```

Implementation Details:

- This method updates the weights of connections coming into the current neuron.
- It uses both a learning rate (`eta`) and a momentum term (`alpha`).
- The weight update incorporates the gradient, the output of the neuron in the previous layer, and the previous weight change.

Memory Access Patterns:

- The method sequentially accesses neurons in the previous layer.
- For each neuron in the previous layer, it accesses and modifies the weight and delta weight of its connection to the current neuron.

Code Dynamics:

- The learning rate and momentum are hard-coded constants. In more advanced implementations, these could be adaptive or user-specified parameters.
- The method combines the current gradient-based update with the previous update (momentum) to determine the new weight change.

3 Training Process

3.1 Training Data Handling

The `TrainingData` class handles reading and parsing of training data:

```
struct TrainingData {
private:
    std::ifstream _dataFile;
public:
    TrainingData(const std::string FileName) {
        _dataFile.open(FileName.c_str());
        if (!_dataFile.is_open()) {
            throw std::runtime_error("Unable to open file: " + FileName);
        }
    }
    bool IsEof(void) const { return _dataFile.eof(); }
    void GetTopology(std::vector<unsigned>& topology) {
        std::string line;
        std::string label;
        std::getline(_dataFile, line);
        std::stringstream ss(line);
        ss >> label;
        if (label != "topology:") {
            std::cerr << "Error: Expected 'topology:', found '" << label << "'" << std::endl
    ;
            return;
        }
        unsigned n;
        while (ss >> n) {
            topology.push_back(n);
        }
    }
    unsigned GetNextInputs(std::vector<double>& InputVals) {
        InputVals.clear();
        std::string line;
        std::string label;
        while (std::getline(_dataFile, line)) {
            std::stringstream ss(line);
            ss >> label;
            if (label == "in:") {
                double OneVal;
                while (ss >> OneVal) {
                    InputVals.push_back(OneVal);
                }
                return InputVals.size();
            }
        }
        return 0;
    }
    unsigned GetTargetOutputs(std::vector<double>& TargetVals) {
        // Similar to GetNextInputs, but for "out:" lines
        // ... (code omitted for brevity)
    }
};
```

Implementation Details:

- The class uses an `ifstream` to read data from a file.
- It provides methods to read the network topology, input values, and target output values.
- The data file format is specific, with "topology:", "in:", and "out:" labels marking different types of data.

Memory Handling:

- The class reads data line by line, processing each line into appropriate data structures.

- It uses `std::vector` to store topology, input values, and target values, which allows for dynamic sizing but may involve memory reallocation.

Code Dynamics:

- The `GetTopology`, `GetNextInputs`, and `GetTargetOutputs` methods use string streams to parse data from file lines.
- Error checking is implemented to ensure the correct format of the input file.

3.2 Training Loop

The main training loop is implemented in the `main` function:

```
int main() {
    try {
        TrainingData data("TrainingData\\DataFile.txt");
        std::vector<unsigned> topology;
        data.GetTopology(topology);
        Net NeuralNet(topology);

        std::vector<double> InputVals, TargetVals, ResultVals;
        uint16_t TrainingPass = 0;

        while (!data.IsEof()) {
            ++TrainingPass;
            std::cout << "Pass " << TrainingPass << std::endl;

            if (data.GetNextInputs(InputVals) != topology[0]) {
                std::cout << "End of training data" << std::endl;
                break;
            }

            NeuralNet.FeedForward(InputVals);

            NeuralNet.GetResults(ResultVals);
            if (ResultVals.empty()) {
                std::cerr << "Error: No results produced by the network" << std::endl;
                break;
            }

            if (data.GetTargetOutputs(TargetVals) != topology.back()) {
                std::cerr << "Mismatch in number of target values" << std::endl;
                std::cerr << "Expected " << topology.back() << ", got " << TargetVals.size()
     << std::endl;
                break;
            }

            NeuralNet.BackProp(TargetVals);
        }
    } catch (const std::exception& e) {
        std::cerr << "An exception occurred: " << e.what() << std::endl;
        return 1;
    }

    std::cout << "Training completed" << std::endl;
    return 0;
}
```

Implementation Details:

- The main function sets up the neural network based on the topology read from the training data file.
- It then enters a loop, reading input data, performing forward propagation, backpropagation, and weight updates for each training example.

Memory Handling:

- The `Net` object and associated vectors (`InputVals`, `TargetVals`, `ResultVals`) are created on the stack.

- The neural network itself (contained within the `Net` object) uses dynamic memory allocation for its layers and neurons.

Code Dynamics:

- The training process continues until the end of the data file is reached.
- Each pass through the loop represents one training example.
- Error checking is performed to ensure the input and target data match the network's topology.

4 Neural Network Functionality

4.1 Initialization

The `Neuron` constructor initializes the weights randomly:

```
Neuron::Neuron(uint16_t numOutputs, uint16_t myIndex)
    : _myIndex(myIndex)
{
    for (uint16_t c = 0; c < numOutputs; ++c) {
        _outputWeights.push_back(Connections());
        _outputWeights.back().weight = RandomWeight();
    }
}

static double RandomWeight() { return rand() / double(RAND_MAX); }
```

Implementation Details:

- The constructor takes the number of outputs and the neuron's index as parameters.
- It initializes the `_outputWeights` vector with random weights for each connection.
- The `RandomWeight()` function generates a random weight between 0 and 1.

Memory Allocation:

- The `_outputWeights` vector is dynamically resized for each new connection.
- Each `Connections` object is created and pushed onto the vector individually.

Code Dynamics:

- The random initialization of weights is crucial for breaking symmetry in the network, allowing different neurons to learn different features.

4.2 Feed Forward

The feed-forward process is implemented in the `FeedForward` method:

```
void Net::FeedForward(const std::vector<double>& inputVals) {
    assert(inputVals.size() == _layers[0].size() - 1);

    for (uint16_t i = 0; i < inputVals.size(); ++i) {
        _layers[0][i].SetOutputVal(inputVals[i]);
    }

    for (uint16_t layerNum = 1; layerNum < _layers.size(); ++layerNum) {
        Layer& prevLayer = _layers[layerNum - 1];
        for (uint16_t n = 0; n < _layers[layerNum].size() - 1; ++n) {
            _layers[layerNum][n].FeedNeurons(prevLayer);
        }
    }
}
```

Implementation Details:

- The method takes a vector of input values and propagates them through the network.
- It first sets the output values of the input layer neurons.
- Then, it iterates through subsequent layers, calling `FeedNeurons` on each neuron.

Memory Access Patterns:

- The method accesses neurons layer by layer, which can be cache-friendly if layers fit in cache lines.

Code Dynamics:

- The assertion at the beginning ensures that the number of input values matches the number of input neurons (excluding the bias neuron).

- The outer loop iterates over layers, while the inner loop iterates over neurons in each layer.

The `Neuron::FeedNeurons` method:

```
void Neuron::FeedNeurons(const Layer& prevLayer) {
    double sum = 0.0;
    for (uint16_t n = 0; n < prevLayer.size(); ++n) {
        sum += prevLayer[n].GetOutputVal() *
            prevLayer[n]._outputWeights[_myIndex].weight;
    }
    _outputVal = Neuron::TransferFunction(sum);
}
```

Implementation Details:

- This method computes the weighted sum of inputs from the previous layer.
- It then applies the activation function (`TransferFunction`) to this sum.

Code Dynamics:

- The loop iterates over all neurons in the previous layer, including the bias neuron.
- The activation function is applied to the final sum.

4.3 Backpropagation

The backpropagation process is implemented in the `BackProp` method:

```
void Net::BackProp(const std::vector<double>& targetVals) {
    // ... (error calculation code) ...

    for (uint16_t n = 0; n < outputLayer.size() - 1; ++n) {
        outputLayer[n].CalcOutputGradients(targetVals[n]);
    }

    for (uint16_t layerNum = _layers.size() - 2; layerNum > 0; --layerNum) {
        Layer& hiddenLayer = _layers[layerNum];
        Layer& nextLayer = _layers[layerNum + 1];

        for (uint16_t n = 0; n < hiddenLayer.size(); ++n) {
            hiddenLayer[n].CalcHiddenGradients(nextLayer);
        }
    }

    for (uint16_t layerNum = _layers.size() - 1; layerNum > 0; --layerNum) {
        Layer& layer = _layers[layerNum];
        Layer& prevLayer = _layers[layerNum - 1];

        for (uint16_t n = 0; n < layer.size() - 1; ++n) {
            layer[n].UpdateWeights(prevLayer);
        }
    }
}
```

Implementation Details:

- The method first calculates the output gradients for the output layer.
- It then calculates gradients for hidden layers, moving backwards through the network.
- Finally, it updates the weights for all layers, again moving backwards.

Code Dynamics:

- The process involves three main steps: output gradient calculation, hidden gradient calculation, and weight updates.
- Each step involves iterating over layers and neurons, with different computations performed at each level.

5 Training Process

5.1 Training Data

Training data is read from a file using the `TrainingData` class. Here's how the class is implemented:

```
struct TrainingData {
private:
    std::ifstream _dataFile;
public:
    TrainingData(const std::string FileName) {
        _dataFile.open(FileName.c_str());
        if (!_dataFile.is_open()) {
            throw std::runtime_error("Unable to open file: " + FileName);
        }
    }
    bool IsEof(void) const { return _dataFile.eof(); }
    void GetTopology(std::vector<unsigned>& topology) {
        std::string line;
        std::string label;
        std::getline(_dataFile, line);
        std::stringstream ss(line);
        ss >> label;
        if (label != "topology:") {
            std::cerr << "Error: Expected 'topology:', found '" << label << "'" << std::endl
    ;
            return;
        }
        unsigned n;
        while (ss >> n) {
            topology.push_back(n);
        }
    }
    unsigned GetNextInputs(std::vector<double>& InputVals) {
        InputVals.clear();
        std::string line;
        std::string label;
        while (std::getline(_dataFile, line)) {
            std::stringstream ss(line);
            ss >> label;
            if (label == "in:") {
                double OneVal;
                while (ss >> OneVal) {
                    InputVals.push_back(OneVal);
                }
                return InputVals.size();
            }
        }
        return 0;
    }
    unsigned GetTargetOutputs(std::vector<double>& TargetVals) {
        TargetVals.clear();
        std::string line;
        std::string label;
        while (std::getline(_dataFile, line)) {
            std::stringstream ss(line);
            ss >> label;
            if (label == "out:") {
                double OneVal;
                while (ss >> OneVal) {
                    TargetVals.push_back(OneVal);
                }
                return TargetVals.size();
            }
        }
        return 0;
    }
};
```

This class reads the network topology and training data from a file. The file format is:

```
topology: 5 10 8 1
in: 0.1 0.2 0.3 0.4 0.5
out: 0
in: 0.9 0.8 0.7 0.6 0.5
out: 1
...
```

This format allows for easy specification of the network topology and input-output pairs for training.

5.2 Training Loop

The main training loop is implemented in the `main` function:

```cpp
int main() {
    try {
        TrainingData data("TrainingData\\DataFile.txt");
        std::vector<unsigned> topology;
        data.GetTopology(topology);
        Net NeuralNet(topology);

        std::vector<double> InputVals;
        std::vector<double> TargetVals;
        std::vector<double> ResultVals;
        uint16_t TrainingPass = 0;

        while (!data.IsEof()) {
            ++TrainingPass;
            std::cout << "Pass " << TrainingPass << std::endl;

            if (data.GetNextInputs(InputVals) != topology[0]) {
                std::cout << "End of training data" << std::endl;
                break;
            }

            NeuralNet.FeedForward(InputVals);

            NeuralNet.GetResults(ResultVals);
            if (ResultVals.empty()) {
                std::cerr << "Error: No results produced by the network" << std::endl;
                break;
            }

            if (data.GetTargetOutputs(TargetVals) != topology.back()) {
                std::cerr << "Mismatch in number of target values" << std::endl;
                std::cerr << "Expected " << topology.back() << ", got " << TargetVals.size()
     << std::endl;
                break;
            }

            NeuralNet.BackProp(TargetVals);
        }
    } catch (const std::exception& e) {
        std::cerr << "An exception occurred: " << e.what() << std::endl;
        return 1;
    }

    std::cout << "Training completed" << std::endl;

    return 0;
}
```

This loop reads input data, performs feed-forward propagation, gets the network's output, reads target values, and performs backpropagation. This process is repeated for each training example in the data file.

5.3 Weight Updates

Weights are updated in the `Neuron::UpdateWeights` method:

```
void Neuron::UpdateWeights(Layer& prevLayer) {
    const double eta = 0.15;  // learning rate
    const double alpha = 0.5; // momentum

    for (uint16_t n = 0; n < prevLayer.size(); ++n) {
        Neuron& neuron = prevLayer[n];
        double oldDeltaWeight = neuron._outputWeights[_myIndex].deltaWeight;
        double newDeltaWeight =
            eta * neuron.GetOutputVal() * _gradient
            + alpha * oldDeltaWeight;

        neuron._outputWeights[_myIndex].deltaWeight = newDeltaWeight;
        neuron._outputWeights[_myIndex].weight += newDeltaWeight;
    }
}
```

This implements the weight update rule:

$$\Delta w_{ij} = \eta \cdot a_i \cdot \delta_j + \alpha \cdot \Delta w_{ij}^{prev}$$

Where:

- η is the learning rate (set to 0.15)
- a_i is the activation of the neuron in the previous layer
- δ_j is the gradient of the current neuron
- α is the momentum factor (set to 0.5)
- Δw_{ij}^{prev} is the previous weight change

The learning rate controls how quickly the network adapts to the training data. A smaller value leads to slower but potentially more stable learning, while a larger value can lead to faster but potentially unstable learning.

The momentum term helps the network to escape local minima and can speed up learning in regions where the gradient is small.

6 Performance Metrics

6.1 Error Calculation

The network error is calculated as the Root Mean Square (RMS) error in the `Net::BackProp` method:

```
void Net::BackProp(const std::vector<double>& targetVals) {
    Layer& outputLayer = _layers.back();
    _error = 0.0;

    for (uint16_t n = 0; n < outputLayer.size() - 1; ++n) {
        double delta = targetVals[n] - outputLayer[n].GetOutputVal();
        _error += delta * delta;
    }
    _error /= outputLayer.size() - 1;
    _error = sqrt(_error);

    // ... rest of the backpropagation code ...
}
```

This implements the RMS error formula:

$$E = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(t_i - y_i)^2}$$

Where:

- n is the number of output neurons
- t_i is the target value for output neuron i
- y_i is the actual output of neuron i

RMS error provides a measure of the average magnitude of the error, with larger errors being penalized more heavily due to the squaring.

6.2 Performance Monitoring

The current implementation prints debug information for each step of the process. For example, in the `Net::FeedForward` method:

```
void Net::FeedForward(const std::vector<double>& inputVals) {
    // ... feed forward code ...

    std::cout << "Debug: FeedForward result: " << _layers.back()[0].GetOutputVal() << std::
    endl;
}
```

Similar debug output is included throughout the code to monitor:

- Input values
- Output values
- Target values
- Weight updates
- Overall network error

This detailed output allows for close monitoring of the training process and can be invaluable for debugging and optimizing the network's performance.

7 Overall Performance Analysis

Computational Complexity:

- The time complexity of forward propagation and backpropagation is O(W), where W is the total number of weights in the network.
- For a fully connected network with L layers and N neurons per layer (on average), W is approximately $N^2 * (L - 1)$, making the overall complexity O(N^2 * L) per training example.

Memory Usage:

- The space complexity is O(W), as the network needs to store all weights and their gradients.
- Additional memory is used for intermediate computations during forward and backward passes.

Scalability:

- The current implementation faces challenges with very deep or wide networks due to increased computation time and memory usage.
- The sequential nature of the training loop limits scalability for large datasets.

Potential Optimizations:

- Implementing mini-batch gradient descent to improve training efficiency.
- Utilizing parallel computing techniques (e.g., OpenMP, CUDA) for matrix operations.
- Optimizing memory access patterns to improve cache utilization.
- Implementing more sophisticated optimization algorithms (e.g., Adam, RMSprop) for faster convergence.
- Adding support for sparse networks to handle larger models more efficiently.

8 Limitations and Potential Improvements

8.1 Limitations

- **Fixed learning rate and momentum**: The current implementation uses constant values for these hyperparameters, which may not be optimal for all problems or throughout the entire training process.
- **Single activation function**: Only the hyperbolic tangent function is implemented, which may not be suitable for all types of data or problems.
- **Limited to feed-forward architectures**: The implementation doesn't support more complex architectures like convolutional or recurrent networks.
- **No built-in cross-validation or early stopping**: This can lead to overfitting if the network is trained for too long.
- **Limited input data format**: The current implementation only supports a specific file format for training data.

8.2 Potential Improvements

- **Implement adaptive learning rates**: Techniques like Adam or RMSprop could be implemented to automatically adjust the learning rate during training.
- **Add support for different activation functions**: Implementing ReLU, sigmoid, and other activation functions would make the network more versatile.
- **Implement regularization techniques**: L1 and L2 regularization could be added to help prevent overfitting.
- **Add support for different network architectures**: Implementing convolutional layers, recurrent layers, or skip connections would allow the network to handle a wider range of problems.

9 Conclusion

This custom neural network implementation provides a comprehensive foundation for understanding the fundamentals of neural networks and the backpropagation algorithm. The C++ implementation offers a clear view of the inner workings of neural networks, from the structure of individual neurons to the process of training the entire network.

Key strengths:

- A modular, object-oriented design that closely mirrors the conceptual structure of neural networks
- Detailed implementation of forward propagation and backpropagation algorithms
- A flexible network topology that can be easily modified
- Integration of momentum in weight updates to improve training dynamics

While there are several areas for potential improvement, this implementation serves as a solid starting point for further exploration and development in the field of neural networks. It can be extended and optimized to handle more complex problems and architectures, making it a valuable resource for both educational purposes and practical applications.