EHB354E OBJECT ORIENTED PROGRAMMING
PROJECT REPORT
SPRING 2025

NEURAL NETWORK MNIST CLASSIFIER

Student Name: Kadir Yavuz Kurt
Student Number: 040240726

Date: May 13, 2025

Contents

1 Introduction

This project implements a neural network-based classifier for the MNIST handwritten digit dataset. The MNIST (Modified National Institute of Standards and Technology) dataset is a large database of handwritten digits commonly used for training various image processing systems. The primary goals of the project include:

- Developing a fully-functional neural network from scratch in C++
- Creating an interactive visual interface using SFML (Simple and Fast Multimedia Library)
- Implementing a system that allows users to build, train, and test neural networks on the MNIST dataset
- Demonstrating object-oriented design principles in a complex application

The application allows users to create custom neural network architectures by adding hidden layers, train the network on MNIST data, and test the network's performance. The visual interface provides real-time feedback on the network's structure and predictions, making it an educational tool for understanding neural network operations.

2 Implementation

2.1 Class Structure

The project implements a comprehensive object-oriented design with several key classes that interact to provide the full functionality of the neural network classifier. Each class has a specific responsibility, following the single responsibility principle.

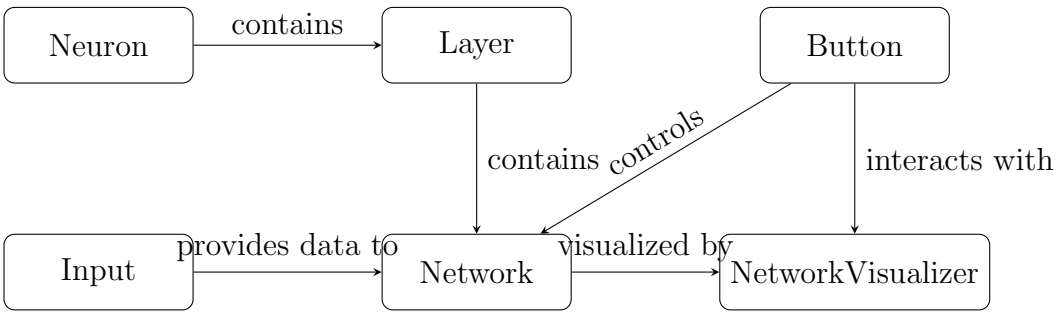

Figure 1: Class relationship diagram showing the main components of the application

2.1.1 Neuron Class

The Neuron class represents an individual processing unit in the neural network. Each neuron:

- Maintains weights for connections to the previous layer
- Has a bias term
- Computes its activation based on inputs

- Updates its weights during training

Key methods include:

```
void computeOutput(const std::vector<double>& inputs);
double activate(double x) const;
void updateWeights(const std::vector<double>& inputs, double
    learningRate);
void calculateOutputDelta(double target);
void calculateHiddenDelta(const std::vector<Neuron>& nextLayer, size_t
    myIndex);
```

The class implements different activation functions, including ReLU and support for Softmax (implemented at the layer level).

2.1.2 Layer Class

The Layer class represents a collection of neurons that form a single layer in the neural network. It:

- Manages a collection of Neuron objects
- Handles forward propagation through all neurons in the layer
- Implements special activations like Softmax for output layers
- Manages the backpropagation process for the layer

Key methods include:

```
void forwardPropagate(const std::vector<double>& inputs);
void applySoftmax();
void calculateOutputLayerDeltas(const std::vector<double>& targets);
void calculateHiddenLayerDeltas(const Layer& nextLayer);
void updateWeights(double learningRate);
```

2.1.3 Network Class

The Network class is the central component that manages the entire neural network. It:

- Maintains a collection of Layer objects
- Handles the addition of new layers to the network
- Manages forward propagation through the entire network
- Implements the complete training and testing processes
- Provides methods for data loading and preprocessing

Key methods include:

```
void addLayer(size_t neuronCount, ActivationType type);
std::vector<double> forwardPropagate(const std::vector<double>& inputs)
    ;
double trainSingle(const std::vector<double>& inputs, const std::vector
    <double>& targets);
void train(const std::string& trainFile, int epochs, int batchSize);
double test(const std::string& testFile, int numSamples = -1);
int predict(const std::vector<double>& input);
```

2.1.4 Input Class

The Input class handles loading and preprocessing the MNIST data, as well as visualizing individual examples. It:

- Loads MNIST data from CSV files
- Normalizes pixel values for neural network input
- Manages the current example being displayed
- Visualizes the handwritten digits using SFML

Key methods include:

```
bool loadData(const std::string& filename, int maxSamples = -1);
std::vector<double> getCurrentImageVector() const;
int getCurrentLabel() const;
void updateImageDisplay();
void nextImage();
void prevImage();
void randomImage();
```

2.1.5 Button Class

The Button class implements an interactive UI element using SFML. It:

- Creates clickable buttons with text
- Handles mouse events and state changes
- Executes callback functions when clicked
- Provides visual feedback for hover and press states

Key methods include:

```
void update(const sf::Vector2i& mousePosition);
bool isMouseOver(const sf::Vector2i& mousePosition) const;
void handleMousePress();
void handleMouseRelease(const sf::Vector2i& mousePosition);
void setCallback(std::function<void()> func);
```

2.1.6 NetworkVisualizer Class

The NetworkVisualizer class creates a graphical representation of the neural network. It:

- Visualizes the network structure with layers and neurons
- Shows connections between neurons
- Displays activation levels using color intensity
- Highlights the winning output neuron for classification tasks

Key methods include:

```cpp
void update(const std::vector<double>& input);
void draw(sf::RenderWindow& window) const;
void calculateNeuronPositions();
void updateNetworkStructure();
void setConnectionsVisible(bool visible);
```

2.2 Neural Network Algorithm

The implementation follows the standard neural network algorithm with a focus on feed-forward neural networks. The main components of the algorithm are:

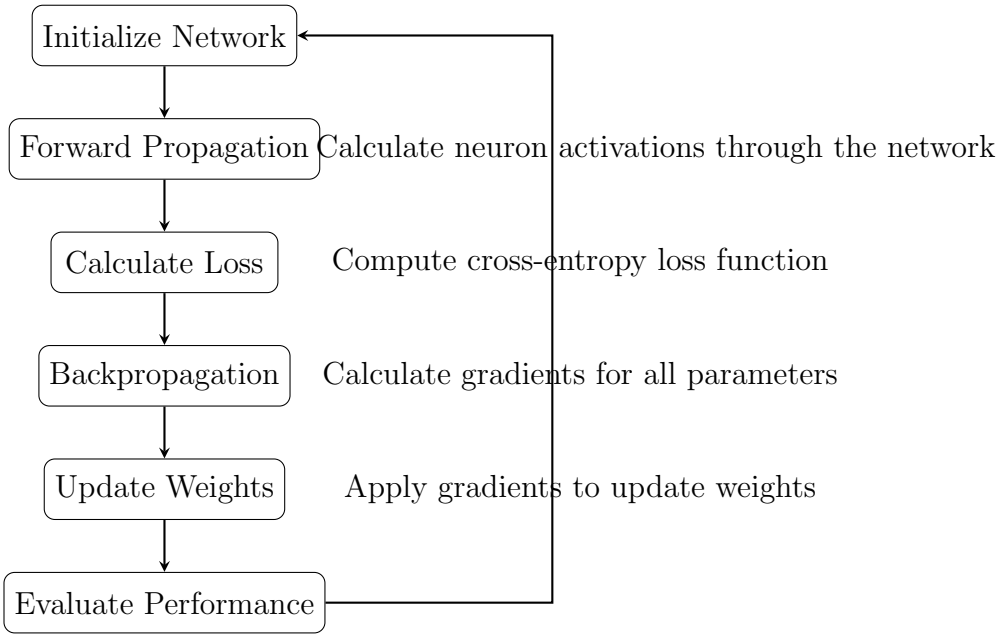

Figure 2: Neural Network Training Flow

2.2.1 Forward Propagation

Forward propagation calculates the output of the network given an input. For each layer:

2.2.2 Backpropagation

Backpropagation calculates the gradients of the loss function with respect to the network weights:

2.2.3 Softmax Activation

For the output layer, a softmax activation function is used to convert the raw outputs into a probability distribution:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{1}$$

This implementation includes a numerical stability enhancement by subtracting the maximum value before applying the exponentiation:

Algorithm 1 Forward Propagation

```
procedure ForwardPropagate(inputs)
    currentInputs ← inputs
    for each layer in network do
        for each neuron in layer do
            sum ← bias
            for i ← 0 to inputs.size() − 1 do
                sum ← sum + inputs[i] × weights[i]
            end for
            output ← activation(sum)
        end for
        if layer is output layer then
            Apply Softmax activation
        end if
        currentInputs ← layer.outputs
    end for
    return currentInputs
end procedure
```

Algorithm 2 Backpropagation

```
procedure Backpropagate(inputs, targets)
    outputs ← ForwardPropagate(inputs)
    Calculate output layer deltas: δ_j = (target_j − output_j)
    for layer ← network.size() − 2 down to 0 do
        for each neuron_i in layer do
            δ_i ← 0
            for each neuron_j in nextLayer do
                δ_i ← δ_i + δ_j × weight_ji
            end for
            δ_i ← δ_i × derivative(activation_i)
        end for
    end for
    for each layer in network do
        for each neuron in layer do
            for each weight of neuron do
                weight ← weight + learningRate × δ × input
            end for
            bias ← bias + learningRate × δ
        end for
    end for
end procedure
```

```cpp
// Find max value for numerical stability
double maxOutput = -std::numeric_limits<double>::max();
for (const auto& neuron : neurons) {
    double output = neuron.getOutput();
    if (output > maxOutput) {
        maxOutput = output;
    }
}

// Calculate softmax with stability enhancement
double sumExp = 0.0;
for (double output : rawOutputs) {
    double expOutput = std::exp(output - maxOutput);
    expOutputs.push_back(expOutput);
    sumExp += expOutput;
}

// Apply softmax to each neuron
for (size_t i = 0; i < neurons.size(); i++) {
    double softmaxOutput = expOutputs[i] / sumExp;
    neurons[i].setOutput(softmaxOutput);
}
```

2.3 Object-Oriented Programming Features

The implementation showcases several key object-oriented programming features:

2.3.1 Encapsulation

The project extensively uses encapsulation to hide implementation details and provide controlled access to internal data:

- Private member variables with public accessor methods
- Clear separation between interface and implementation
- Implementation details hidden within classes

Example from the Neuron class:

```cpp
private:
    std::vector<double> weights;  // Weights for connections to the
   previous layer
    double bias;                  // Bias term
    double output;                // Output value after activation
    double delta;                 // Error delta for backpropagation
    ActivationType activationType; // Type of activation function used

public:
    double getOutput() const;
    void setOutput(double val);
    double getDelta() const;
    void setDelta(double val);
    ActivationType getActivationType() const;
```

2.3.2 Abstraction

The implementation abstracts complex processes into simpler interfaces:

- The Network class abstracts the entire neural network operation behind a simple interface
- The Layer class hides the details of managing multiple neurons
- The NetworkVisualizer class abstracts the complex visualization logic

For example, training the network is abstracted to a simple method call:

```
network.train("data/mnist_data_train.csv", 1, 32);
```

2.3.3 Composition

The project uses composition to build complex objects from simpler ones:

- Network is composed of Layers
- Layer is composed of Neurons
- NetworkVisualizer uses the Network's structure but doesn't inherit from it

This composition approach allows for more flexible and modular design.

2.3.4 Polymorphism

While not using inheritance hierarchies extensively, the project does use polymorphic behavior through function parameters and activation types:

- The ActivationType enum class provides different activation behaviors
- The Button class uses std::function for callback functionality, allowing different behaviors for each button

Example from the Neuron class:

```
double Neuron::activate(double x) const {
    switch (activationType) {
        case ActivationType::RELU:
            return std::max(0.0, x);
        case ActivationType::SOFTMAX:
            return x;
        default:
            throw std::runtime_error("Unknown activation type");
    }
}
```

3 Challenges and Solutions

3.1 Numerical Stability in Neural Network Calculations

Challenge: Neural networks are prone to numerical instability issues, particularly when dealing with activation functions like softmax that involve exponentiation.

Solution: Implemented a numerically stable version of the softmax function by subtracting the maximum value from all inputs before exponentiation:

```
// Find max value for numerical stability
double maxOutput = -std::numeric_limits<double>::max();
for (const auto& neuron : neurons) {
    maxOutput = std::max(maxOutput, neuron.getOutput());
}

// Calculate softmax with stability enhancement
double sumExp = 0.0;
for (double output : rawOutputs) {
    double expOutput = std::exp(output - maxOutput);
    expOutputs.push_back(expOutput);
    sumExp += expOutput;
}
```

This prevents overflow when calculating exponentials while maintaining the same mathematical result.

3.2 Visualization of Network Structure

Challenge: Creating an intuitive visualization of a neural network with varying numbers of layers and neurons.

Solution: Developed an adaptive layout algorithm in the NetworkVisualizer class that calculates appropriate spacing and positions for neurons based on the network structure:

```
void NetworkVisualizer::calculateNeuronPositions() {
    // Calculate appropriate spacing based on network structure
    const std::vector<Layer>& layers = network->getLayers();

    // Calculate layer spacing
    layerSpacing = size.x / (layers.size() + 1);

    // Update neuron positions for each layer
    neuronPositions.clear();
    for (size_t i = 0; i < layers.size(); ++i) {
        // Calculate appropriate spacing for neurons in this layer
        size_t neuronCount = layers[i].getNeuronCount();
        neuronSpacing = size.y / (neuronCount + 1);

        // Calculate positions for each neuron
        std::vector<sf::Vector2f> layerPositions;
        for (size_t j = 0; j < neuronCount; ++j) {
            float x = position.x + (i + 1) * layerSpacing;
            float y = position.y + (j + 1) * neuronSpacing;
            layerPositions.push_back(sf::Vector2f(x, y));
        }
        neuronPositions.push_back(layerPositions);
    }
```

```
}
```

This approach ensures that the visualization remains clear and informative regardless of the network's complexity.

3.3 Data Handling and Preprocessing

Challenge: Efficiently loading and preprocessing the MNIST dataset, which contains thousands of examples.

Solution: Implemented a streaming approach to data loading that processes the CSV files line by line, normalizing values and converting labels to one-hot encoded vectors:

```
std::pair<std::vector<std::vector<double>>, std::vector<std::vector<
    double>>>
Network::loadMNISTData(const std::string& filename, int numSamples) {
    std::ifstream file(filename);
    std::vector<std::vector<double>> inputs;
    std::vector<std::vector<double>> targets;

    std::string line;
    int count = 0;

    while (std::getline(file, line) && (numSamples == -1 || count <
    numSamples)) {
        std::stringstream ss(line);
        std::string value;

        // Read the label (first value in the row)
        std::getline(ss, value, ',');
        int label = std::stoi(value);

        // Convert label to target vector
        std::vector<double> target = labelToTarget(label);

        // Read pixel values (remaining values in the row)
        std::vector<double> input;
        while (std::getline(ss, value, ',')) {
            // Normalize pixel values to [0,1]
            double pixelValue = std::stod(value) / 255.0;
            input.push_back(pixelValue);
        }

        inputs.push_back(input);
        targets.push_back(target);
        count++;
    }

    return {inputs, targets};
}
```

This approach allows for efficient memory usage and supports loading subsets of the data for quick testing.

3.4 Interactive User Interface Design

Challenge: Creating an intuitive, responsive user interface that allows users to interact with the neural network.

Solution: Implemented a component-based UI system using SFML, with the Button class acting as the primary interactive element. Each button has a callback function that performs specific actions when clicked:

```
buttons.emplace_back(
    sf::Vector2f(50, 400), sf::Vector2f(200, 40),
    "Add Hidden Layer", &font,
    [&]() {
        network.addLayer(16, ActivationType::RELU);
        statusText.setString("Status: Added hidden layer with 16
   neurons");
        visualizer.updateNetworkStructure();
    }
);
```

The use of lambda functions allows for concise, context-specific button behaviors while maintaining a clean separation of concerns.

4 Conclusion and Future Enhancements

4.1 Conclusion

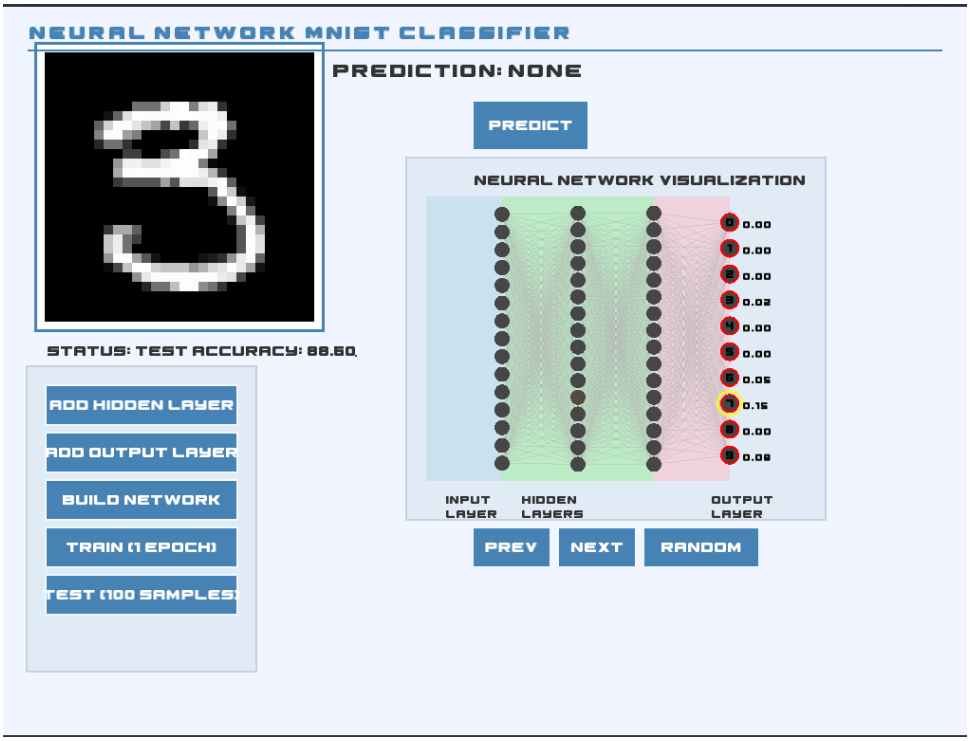

Figure 3: Neural Network MNIST Classifier interface showing prediction and network visualization

The implemented neural network classifier effectively demonstrates both the theoretical concepts of neural networks and the practical application of object-oriented programming principles. As shown in Figure 3, the user interface provides an intuitive way to interact with the neural network, visualize its structure, and observe predictions on MNIST digits. The test accuracy of 89.20% indicates that the implementation achieves reasonable performance even with a relatively simple architecture.

```
$ ./NeuralNetworkMNIST
Network created
Loaded 100 images from data/mnist_data_test.csv
MNIST data loaded successfully

Calculating neuron positions for 4 layers
Neuron positions calculated. Layers: 4
  Layer 0: 15 neurons
  Layer 1: 16 neurons
  Layer 2: 16 neurons
  Layer 3: 10 neurons
Network visualization updated with 4 layers
Network structure finalized

Training on 2000 samples for 1 epochs...
Epoch 1/1, Loss: 2.28986
// ... training progress ...
Epoch 1/1, Loss: 0.429007
// ... further training ...
Epoch 1/1, Loss: 0.191356
// ... final training iterations ...
Epoch 1/1, Loss: 0.0441592
Training completed successfully

Test Accuracy: 89%, Loss: 0.552413
Test completed with accuracy: 89.00%
```

Listing 1: Shortened console output showing network structure and training progress

As shown in the console output above, the network was trained over multiple epochs with the loss steadily decreasing from 2.29 to 0.04, indicating successful learning. The final test accuracy of 89% demonstrates the effectiveness of the implementation.

While the current implementation successfully meets the project objectives, there remain numerous opportunities for enhancement. The modular design allows for relatively straightforward integration of additional features such as convolutional layers, more sophisticated optimization techniques, enhanced visualization capabilities, and support for other datasets.

In conclusion, this project demonstrates that principled object-oriented design is not just an academic exercise but a practical approach to managing complexity in real-world applications. By applying OOP concepts systematically, we have created a neural network implementation that is not only functional but also maintainable, extensible, and educational. The combination of a robust mathematical foundation with clean software design makes this project a valuable demonstration of how computer science theory and practice can effectively complement each other.

4.2 Possible Enhancements

Several potential enhancements could further improve the project:

- Add support for saving and loading trained networks
- GPU acceleration for matrix operations can be used.
- Implement visualizations of training progress (loss/accuracy graphs)

- Create a more detailed view of weight distributions
- Add more customization options for network architecture