**EHB354E OBJECT ORIENTED PROGRAMMING PROJECT REPORT**

**SPRING 2025**

**Neural Network Visualizer**

**Student Name: Efe Murat Uçarlı**

**Student Number: 070200760**

**Date:**

**Introduction**

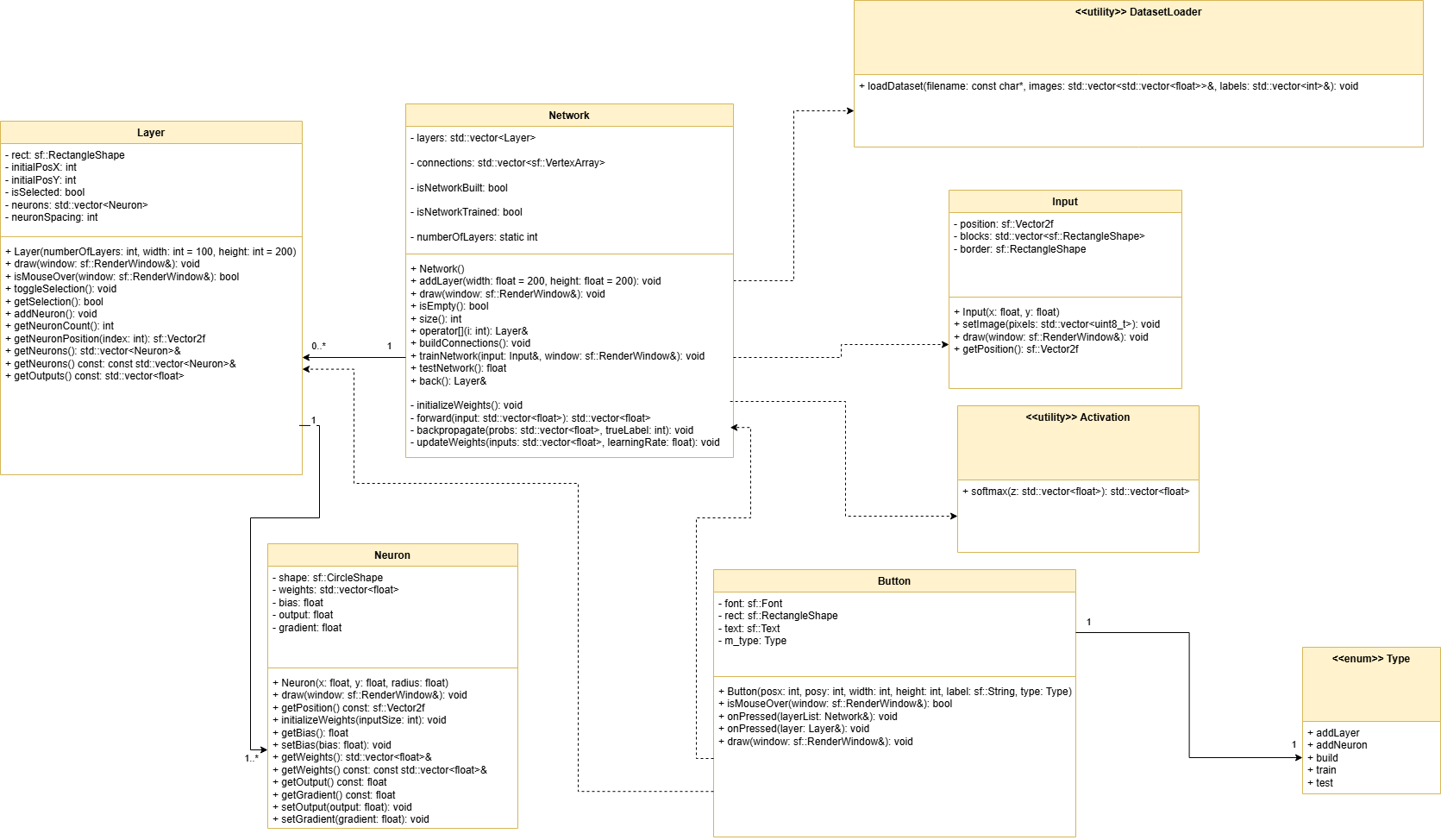
This project aims to implement and visualize a fully connected feedforward neural network using the object-oriented programming (OOP) paradigm in C++. The primary objective is to provide an interactive graphical interface through which users can construct and train a neural network model on the MNIST dataset. The application allows users to dynamically build the architecture by adding layers and neurons via UI controls. Each component of the network—such as layers, neurons, and their interconnections—is encapsulated into modular C++ classes, promoting reusability, scalability, and clear separation of functionalities.

The model training process involves forward and backward propagation implemented from scratch, without reliance on external deep learning libraries. The user can initiate training and testing directly through the GUI, while real-time predictions and classification results are simultaneously displayed in the console, providing immediate feedback on the model's behavior.

With object-oriented programming and a user-friendly interface built in SFML, this project offers a simple and visual way to understand and experiment with neural network training and testing processes.

**Implementation**

Here, you should describe the classes you implemented. Point out the object-oriented programming features you used in your implementation.

Try to use block diagrams, flow charts and any other visuals that will help explain your algorithm to a reader who does not know anything about your project.

This project was developed using object-oriented programming (OOP) principles, ensuring modularity, encapsulation, and reusability. The following classes and utility modules were implemented to separate concerns between the graphical interface and the core logic of the neural network. The UML class diagram above visualizes their relationships and responsibilities.

**Network**

The Network class represents the logical structure of a feedforward neural network. It consists of multiple Layer instances, each containing a set of Neuron objects. It encapsulates key operations such as weight initialization, forward propagation, loss computation, backpropagation, and weight updates. Each phase of the training process is modularized into private helper methods, following the single responsibility and separation of concerns principles. The class also exposes public methods for training (trainNetwork) and testing (testNetwork), without containing any graphical logic.

The Network class demonstrates several object-oriented programming concepts in its design. Encapsulation is achieved by organizing the training process into private helper methods, keeping internal logic hidden from external access. The class also exhibits aggregation through its composition of multiple Layer objects, reflecting a has-a relationship. Furthermore, the overall structure of the training pipeline is modular, with each phase—such as weight initialization, forward pass, and backpropagation—implemented as distinct, manageable steps. This modularity not only enhances code clarity but also facilitates debugging and future extensions.

**Button**

The Button class is part of the graphical user interface and encapsulates both its visual representation and associated behavior. Each button is linked to a specific Type, defined as an enumeration with values such as addLayer, train, or test. When a button is clicked, it triggers functionality corresponding to its assigned type. The use of an enum rather than a separate class for Type is intentional and justified by design considerations: the set of possible actions is finite and well-defined, and these actions are not expected to evolve with complex behavior or additional state. Therefore, representing them as an enumeration ensures simplicity, clarity, and efficiency.

The Button class interacts with the Network and Layer classes through overloaded onPressed() methods, illustrating interface-driven interaction and maintaining loose coupling between components. This design promotes modularity while keeping the GUI logic abstracted from the core computation.

The Button class encapsulates both the visual and interactive aspects of the graphical user interface elements used in the application. Each button is linked to a specific Type enumeration, such as addLayer, train, or test, and is responsible for triggering predefined behaviors upon being clicked. This class demonstrates encapsulation by keeping its graphical representation and internal logic self-contained. It also employs polymorphism through method overloading, as the onPressed() function is defined with different signatures to interact with either a Network or a Layer object. Additionally, the class reflects the principle of abstraction by decoupling user interface events from the underlying computational logic, enabling the GUI to remain independent from the network’s internal workings.

**Layer**

The Layer class represents a single layer in the neural network and manages a dynamic collection of Neuron objects. In addition to maintaining the logical structure of a layer, it also handles layout-related attributes such as position, spacing, and selection state, enabling seamless integration with the graphical interface. Methods are provided to add neurons, retrieve their outputs, and interact with individual neuron instances. The class demonstrates encapsulation by managing its internal neuron list and layout properties privately. It also illustrates aggregation, as each layer consists of multiple neurons that it contains and manages. Furthermore, with its draw() method and visual representation capabilities, the class is designed in a way that supports future abstraction or extension through polymorphism, should a more generalized rendering system be introduced.

**Neuron**

The Neuron class models a single computational unit within the neural network. It maintains key internal variables such as weights, bias, output, and gradient, all of which are essential for forward and backward propagation during training. These values are accessed and modified exclusively through dedicated getter and setter methods, ensuring encapsulation and enforcing data hiding to prevent unintended external manipulation. The weights are initialized dynamically based on the input size, allowing for flexibility in network configuration. Additionally, each neuron is capable of rendering itself visually using SFML primitives, combining both computational logic and its graphical representation. This responsibility-driven design allows the neuron to manage its internal state while also supporting its role in the user interface.

**Input**

The Input class serves as a graphical user interface component responsible for visualizing the 28×28 grayscale image input fed into the neural network. It renders a grid of 10×10 blocks, each corresponding to a pixel, and updates this visual representation in real time based on the data provided. Although it does not participate in the computational aspects of training or inference, it plays an important role in enhancing user interaction by offering immediate visual feedback. The class exemplifies encapsulation by keeping all visual state and rendering logic self-contained. It also adheres to the single responsibility principle, as its sole purpose is to manage and display the input image without involving itself in unrelated processing or control logic.

**Utility Modules**

The utility components of the project, namely DatasetLoader and Activation, provide essential functionality that supports the main system without being tied to any specific class structure. The DatasetLoader module includes the loadDataset() function, which reads training data from a CSV file and populates corresponding data structures. Similarly, the Activation module implements the softmax() function, which is used in the output layer of the network for converting logits into probabilities. Both utilities are designed to be stateless and reusable, and follow modular design principles by keeping general-purpose logic separate from the core class definitions. This enhances reusability, as these functions can be called from multiple contexts without introducing tight coupling. Additionally, they support effective dependency management, as they are only included where needed, minimizing unnecessary interdependencies across the system. Such a structure facilitates scalable improvements in the future, including the integration of new network architectures (e.g., convolutional networks), exporting model outputs to files, or expanding graphical functionality—all without requiring extensive refactoring of the existing codebase.

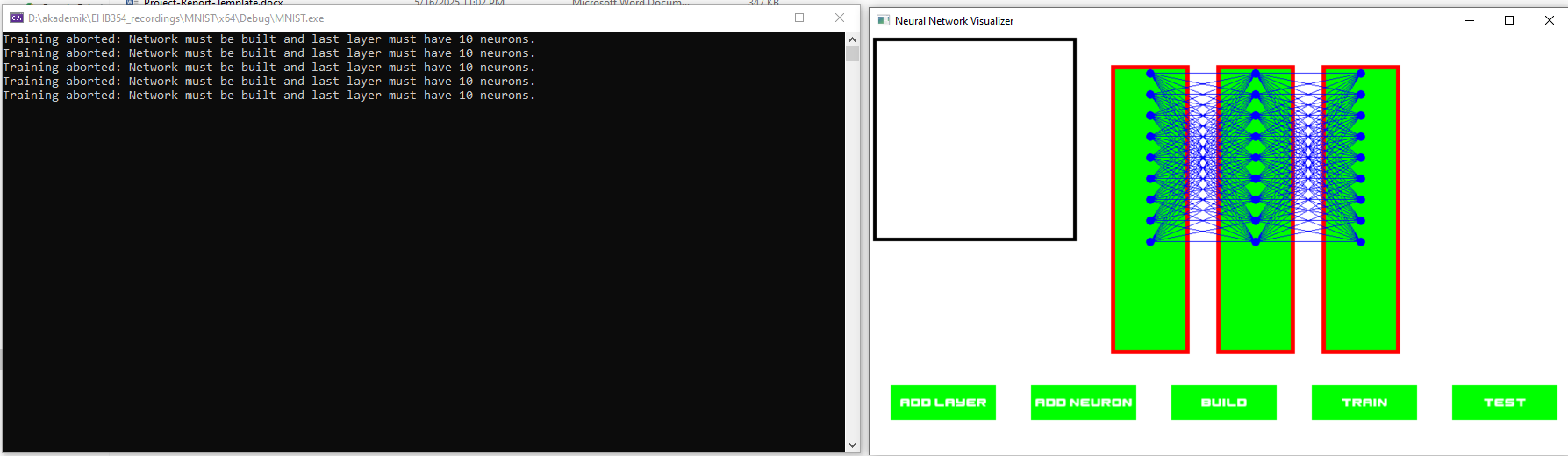
**TrainNetwork Algorithm**

The trainNetwork function is responsible for training the neural network using the MNIST dataset. Training only proceeds if the network has been structurally built (isNetworkBuilt == true) and the last layer contains exactly 10 neurons, which corresponds to the 10 digit classes (0–9) in the MNIST dataset. If this condition is not met, training is aborted and an error message is printed to the console. If the condition is satisfied, the training process begins by loading the dataset and initializing all weights and biases. Then, for each epoch, the function iterates over all samples in the training set. Each input image is converted into grayscale blocks and displayed on the screen in real-time through the Input object.

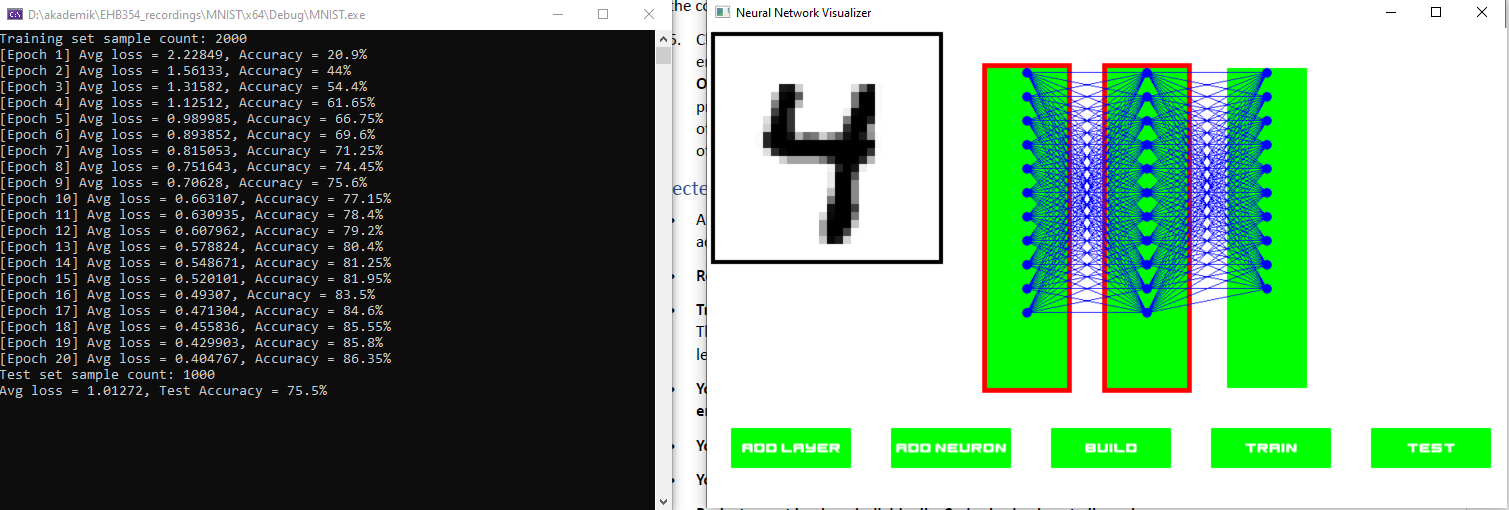
For each sample:

1. A forward pass is executed to compute the predicted class probabilities.
2. The cross-entropy loss is calculated based on the true label.
3. Backpropagation is applied to compute gradients.
4. Weights and biases are updated accordingly.

After each epoch, the average loss and training accuracy are printed to the console. At the end of training, the network is marked as trained.



**Results**



Using a learning rate of 0.01 and training for 20 epochs, the network achieved a training accuracy of 86.35% and a test accuracy of 75.5%.

**Challenges and Solutions**

Identify the challenges faced during the project. Outline debugging processes, problem-solving strategies, and how obstacles were overcome.

One of the initial challenges in the project was designing a clear and structured class hierarchy. The system was structured such that the Network class encapsulates multiple Layer objects, each containing a collection of Neuron instances. Additionally, GUI buttons were designed to trigger relevant network operations.

The implementation began with the graphical interface components, developed in stages: starting with the Button, followed by the Neuron, Layer, and Input classes. These components were responsible for managing user interaction and visual representation. Subsequently, the Network class was implemented to define the core logic of the neural network. Unlike the other components, Network does not include any graphical functionality; instead, it encapsulates the internal structure of the model by managing layers and performing computations related to training and inference.

After completing the user interface, the focus shifted to the implementation of the training algorithm. During the development of the trainNetwork method, several functional errors were encountered. To address these, a divide-and-conquer strategy was employed. The training process was broken down into distinct phases—weight initialization, forward pass, loss computation, backpropagation, and weight updates—each encapsulated as a private method within the Network class. This modular design allowed each step to be verified independently, simplifying the debugging process and ensuring correctness before integration into the full training pipeline.

To support debugging, both cout statements and the Visual Studio debugger were utilized extensively. While cout outputs helped in tracking the flow of execution and intermediate values, the debugger allowed close inspection of variable states within each method. Through this process, an early issue was identified where the weight vectors remained unchanged during training. By observing variable behavior in the debugger, the root cause was found and corrected, ensuring that weight updates were properly applied.

**Conclusion and Future Enhancements**

Summarize the overall success and impact of the project. Provide recommendations for future improvements or expansions.

This project successfully demonstrates the implementation of a neural network system developed in accordance with object-oriented programming (OOP) principles. The architecture effectively separates graphical interface components from the core computational logic, resulting in a modular and extensible design. Currently, the system is configured to operate on a fixed dataset using a fully connected architecture. In future iterations, support for different dataset formats and model types—such as convolutional networks or regression-based models—could significantly enhance flexibility and applicability. Additionally, while training and testing results are currently displayed in the console, incorporating these outputs into the graphical interface would improve usability. Providing users with options to export performance metrics to external files (e.g., TXT or CSV) could also be beneficial for analysis and documentation purposes.