

NATIONAL RESEARCH UNIVERSITY HIGHER SCHOOL OF ECONOMICS

Faculty of Computer Science
Bachelor's Programme "HSE and University of London Double Degree Programme in Data
Science and Business Analytics"

Software project report

on the topic Neural Networks from Scratch

Student:

group БПАД232



Sign

A.V.Suvorova

Name, Surname, Patronymic

03.06.2025

Date

Supervisor:

D.V.Trushin

Name, Surname, Patronymic

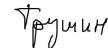
Docent / FCS HSE

Position / Company

03.06.2025

Date

Grade



Sign

Contents

1	Introduction	4
2	Overview of sources	5
3	Functional and non-functional requirements	5
3.1	Functional requirements	5
3.2	Non-functional requirements	6
4	Theoretical background	7
4.1	Layers	7
4.2	Backpropagation	8
4.2.1	Forward pass	8
4.2.2	Backward pass	8
4.2.3	Backpropagation process	8
4.3	Optimization and training	9
4.3.1	SGD	10
4.3.2	Adam	10
5	System architecture	12
5.1	Structure	12
5.2	Data flow and complexity considerations	14
6	Experimental evaluation	16
7	Conclusion	18

Abstract

This report represents the implementation of a fully-connected neural network from scratch in C++. The main goal of the project was to create a self-contained neural network library using C++20 standard language, Eigen and EigenRand libraries as submodules on the GitHub for linear algebra computations. The neural network is a multilayer perceptron trained via the backpropagation algorithm to recognize handwritten digit images from the MNIST dataset. The key features of the project are modular design of the network layers, usage of the EigenRand for efficient random initialization of weights and console-based experiments with the data from MNIST. The implemented network successfully recognizes handwritten digits with the accuracy of 88.76% after just one epoch of training. The report provides theoretical background, discusses the design features, evaluates the network's performance on the test data, highlights challenges and possible optimizations. In conclusion, the project represents a functioning neural network prototype written in C++, which can be integrated in a graphical user interface and can be a foundation of a future more complex extensions.

Keywords

Neural networks; Fully connected; Backpropagation; C++20; Eigen; EigenRand; MNIST; Machine learning; Multilayer perceptron

Glossary of Key Terms

Neuron	The basic computational unit in a neural network. It receives inputs, applies weights and a bias, and passes the result through an activation function.
Layer	A collection of neurons operating in parallel. Layers are classified as input, hidden, or output.
Activation Function	A nonlinear function applied to neuron outputs.
Forward Propagation	The process of computing the output of a network by passing input data through all layers.
Backpropagation	An algorithm for computing gradients of the loss function with respect to weights using the chain rule.
Loss Function	A function that measures the difference between predicted values and actual targets. Examples: Mean Squared Error (MSE), Cross Entropy.
Optimizer	An algorithm that updates weights using gradients.
Training	The process of learning optimal weights by minimizing the loss function using input data.
Epoch	A complete pass over the entire training dataset.
Train Accuracy	The proportion of correctly classified examples in the training set.

1 Introduction

[8] An artificial neural network (referred to as a neural network) is a complex differentiable function that defines a mapping from the initial feature space to the response space, all parameters of which can be adjusted simultaneously and interconnected (that is, the network can be trained end-to-end).

Neural network is a class of machine learning models inspired by the structure and function of a human brain. In machine learning, a neural network is a computational model consisting of layers and interconnected units called neurons, that transform the input data through weighted connections and functions. Despite the widespread use of high-level frameworks that simplify implementation of a neural network, building it from scratch can provide an insight into its underlying mechanics and mathematical logic behind it.

The goal of this project was to develop a library relying on Eigen [1] and EigenRand [2] for linear algebra computations and work with matrices, while maintaining low-level control over the learning algorithm. The following is the motivation for using C++: to achieve a deeper understanding of neural network internals by manually coding forward and backward propagation, and to potentially attain higher execution speed compared to typical implementations on Python.

The importance of this work is to go through each step of the learning process - weight initialization, updates of the gradient and others to find out how does the activation function affect learning and accuracy. Neural network was tested on the MNIST dataset, which contains hand-written digits, and the program was supposed to predict the correct digit on the picture. After just one epoch of training, using 60,000 images with handwritten digits 0-9 represented as vectors of size 784 each, using only 3 layers, ReLU and Identity activation functions, the neural network showed 88.76% accuracy in predicting the output. The successful training and high accuracy on MNIST demonstrate that the custom implementation is correct and competitive for the given problem.

This implementation executes fundamental operations without external deep learning frameworks, the logic is entirely written by author, with the exception of computations made by Eigen. The design is modular, which provides the possibility of code extension by adding new activation functions or loss functions in the future. The report serves as the documentation for the project, containing the evaluation of the network's performance. In this project a neural network was implemented through several modules such as Layer, Activation Function, Loss function, MNIST loader, Model, Optimizer and technical modules for tests, reading from files and writing into them and other utilities.

The structure of the report is the following: section *Overview of sources* reviews related work and libraries for neural networks. Section *Functional and non-functional requirements* details the capabilities of the program and technical constraints and choices. The *Theoretical background* section outlines the mathematical foundation of fully connected neural networks and the backpropagation algorithm used for training. The *System architecture* section describes the software design, including class structure, data flow, and complexity considerations. *Experimental evaluation* presents the results of running the network on the MNIST dataset. Finally, the *Conclusion* summarizes the achievements and challenges of the project and suggests directions for future work, such as adding a GUI or extending the network's capabilities. The report ends with *References* to relevant literature.

2 Overview of sources

Implementing a neural network from scratch is an educational and challenging task. Over time, various libraries and frameworks have emerged in order to simplify the development of neural networks. There are some dominating the field - PyTorch[5] and TensorFlow[7], they offer Python APIs and C++ backends to construct and train neural networks without dealing directly with linear algebra operations. While these sources represent an excellent performance and are easy in use, they do not provide an insight into the details of weight updates and matrix multiplications. Hence, in contrast, this project allows to understand the inner parts of the neural network.

There are some examples of open source libraries written in C/C++, that are aimed at neural network implementation. One of them is FANN[3] - a library written in C with C++ wrappers, which provides a pre-built implementation of multilayer networks and training algorithms and supports training via backpropagation. Despite the availability of the above libraries, there are gaps that this project is aimed to close. Many of them do not implement everything from scratch and rely on another back-end components and/or are not using modern C++ practices. By using C++20, Eigen and EigenRand, this project provides modern language features, efficient math operations and fast random generation of weights. The EigenRand can help to generate efficient random distributions - normal and uniform directly to Eigen matrices, while some other projects may use simple random initializations.

To sum up, the existing tools provide many ways to create neural networks, but sometimes hiding the details of implementation or using outdated language standards. This project serves as an educational code for from-scratch implementation, while integrating numerical libraries in order to create a neural network.

3 Functional and non-functional requirements

3.1 Functional requirements

1. *Neural network construction:* The program should allow creation of a fully connected neural network with a specified architecture. The code in `main.cpp` define the number of layers, the number of neurons in each layer and types of activation functions used. For example, a network with input dimension 784 (for 28×28 pixel images), one hidden layer of 128 neurons, and an output layer of 10 neurons (because there are 10 digits) can be constructed for the MNIST classification task.
2. *Training on dataset:* The program should train the neural network on a given dataset using the backpropagation algorithm. It reads the input data and iteratively adjusts the weights and biases in order to minimize the loss function (MSE in this case). The training data is represented in a MNIST dataset, where each input is an image with a handwritten number, flattened to a 784-dimensional vector and the output is one of 10 classes - digits from 0 to 9.
3. *Prediction:* After the training, the model should take new input data (unused images in training from MNIST dataset) and produce a prediction, which is the result of a forward pass through the network, outputting raw scores. Finally, the program should correctly predict the class of the image (guess the digit).
4. *Accuracy evaluation:* The program should evaluate the trained network's performance on a test dataset by comparing predictions to correct values. It should compute the percentage of correctly classified examples. In `main.cpp`, after training, the code tests the network on the MNIST test set, consisting of 10,000 images, and reports the accuracy achieved. This verifies that the network has learned effectively.
5. *Command-line interface:* The current version of the application is console-based. The program runs from the command line and outputs progress bar, value of the loss function on each step and final accuracy. In future version graphical user interface will be implemented, through which user will be able to select the number of layers, neurons in each layer and types of activation function used.
6. *Data handling:* The program should be capable of reading data from files. The code should include the functionality to load this data into appropriate Eigen structures such as matrices and vectors for processing. If data is not found, the program should output an error message.

3.2 Non-functional requirements

1. *Programming language and standards:* The project is implemented in C++20, which ensures code safety and provides features such as `std::filesystem` for file handling.
2. *Assembly automation tool:* Files of this project are built using CMake version 3.10.
3. *Version control system:* The project was implemented using Git as a version control system. The code is stored in a remote GitHub[6] repository.
4. *Code formatting:* The project files are formatted using clang-format based on style LLVM including additional changes for convenience.
5. *Extensibility:* The code of the project is designed to be extensible. New layer types such as Dropout layer can be added, new activation and loss functions can be introduced in order to improve the performance. While the current implementation is using MNIST data, with minor corrections it can be extended to other tasks and larger neural networks.
6. *Error handling:* The program should be able to handle various errors on each step of the execution. The form of error handling is an error message as a console output.
7. *Minimal RAM requirement:* 2GB+

4 Theoretical background

4.1 Layers

As it was mentioned in Introduction, the neural network is a complex differentiable function or, more precisely, a sequence of differentiable parametric transformations. The type "fully connected neural network" is often called "dense neural network" since it contains layers with artificial neurons, where each neuron in one layer is connected to each neuron in the next layer, creating a *dense* structure.

It is considered to be convenient to represent a complex function as a superposition of simple ones, and neural networks usually appear to the programmer as a constructor consisting of simple blocks - *layers*. Here are two of the simplest varieties:

- *Linear layer* — linear transformation over incoming data. Its trainable parameters are a matrix A and a vector b :

$$x \mapsto xA + b \quad (A \in \mathbb{R}^{d \times k}, x \in \mathbb{R}^d, b \in \mathbb{R}^k).$$

Such a layer transforms d -dimensional vectors into k -dimensional ones.

- *Activation function* — non-linear transformation applied elementwise to the input data. Due to activation functions, neural networks are able to generate more informative feature descriptions by transforming data in a non-linear manner. For example, ReLU (Rectified Linear Unit):

$$\text{ReLU}(x) = \max(0, x)$$

or the Sigmoid function:

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

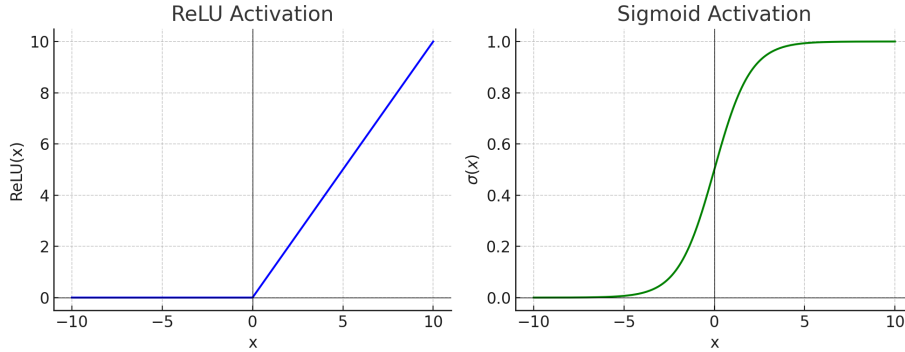


Figure 1: ReLU and Sigmoid activation functions

Even the most complex neural networks are usually assembled from relatively simple blocks like these. Thus, they can be represented as a computational graph, where transformations correspond to intermediate vertices. Such an architecture was used to implement layers in this project. The operation of a single neuron can be described as follows: given an input vector $\mathbf{x} = (x_1, x_2, \dots, x_n)$, the neuron computes the linear combination

$$\mathbf{A} \cdot \mathbf{x} + b,$$

where \mathbf{A} is a weight vector and $b \in \mathbb{R}$ is a bias term (scalar). The result is then passed through an activation function $\sigma(\cdot)$, producing the output

$$\sigma(\mathbf{A} \cdot \mathbf{x} + b).$$

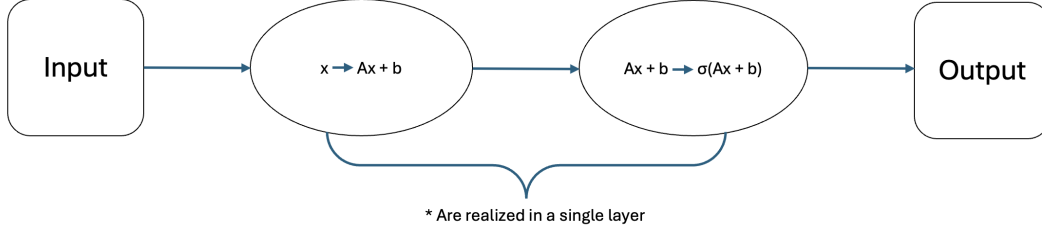


Figure 2: Computation flow: affine transformation and activation in a single layer

4.2 Backpropagation

4.2.1 Forward pass

Information can flow through the graph in two directions - forward and backward. The application of a neural network to data (calculating the output from a given input) is often referred to as forward propagation. At this stage, the initial data representation is transformed into the target one and intermediate data representations are sequentially constructed — the results of applying layers to previous representations.

The input layer receives the raw data. In our case - an image of size 28×28 pixels is flattened into a vector of size 784. Each layer performs its computations - linear part and then activation function, transforming the data. The first weights' matrix and bias vector are initialized through EigenRand library.

4.2.2 Backward pass

During the backward pass, information about the prediction error of the target representation moves from the final representation (from the loss function) to the original one through all transformations. The mechanism of back propagation of error, which plays an important role in the training of neural networks, involves the reverse movement along the computational graph of the network. This is called a process of training - adjusting the weights and biases to minimize the loss function.

The loss function represents the error between the network's predictions and the target values. There are two types of loss functions implemented in this project - Cross Entropy and MSE. The Mean Squared Error (MSE) is defined as:

$$\text{MSE} = \frac{1}{n} \|\mathbf{y}_{\text{pred}} - \mathbf{y}_{\text{true}}\|^2,$$

where \mathbf{y}_{pred} is the predicted output vector, \mathbf{y}_{true} is the ground truth vector, and n is the number of elements.

The gradient of the MSE with respect to the predictions is:

$$\nabla_{\mathbf{y}_{\text{pred}}} \text{MSE} = \frac{2}{n} (\mathbf{y}_{\text{pred}} - \mathbf{y}_{\text{true}}).$$

and the Cross Entropy loss is defined as:

$$\mathcal{L}_{\text{CE}} = - \sum_{i=1}^n y_i^{\text{true}} \cdot \log(y_i^{\text{pred}}),$$

where $\mathbf{y}_{\text{true}}, \mathbf{y}_{\text{pred}} \in \mathbb{R}^n$ are the true and predicted probability distributions, respectively.

The gradient of the cross-entropy loss with respect to the predictions is:

$$\nabla_{\mathbf{y}_{\text{pred}}} \mathcal{L}_{\text{CE}} = - \frac{\mathbf{y}_{\text{true}}}{\mathbf{y}_{\text{pred}}}.$$

The key idea is to propagate the the error backward from the output layer to the input layer.

4.2.3 Backpropagation process

Each training epoch consists of performing the following steps for each training example (or batch of examples) and updating weights accordingly. Over many epochs, the network's predictions hopefully become more accurate as the weights converge to values that produce low value of loss function:

1. *Forward pass*
2. *Compute loss*
3. *Output layer gradient*
4. *Backward pass*
5. *Gradient of weights and biases*
6. *Update weights*

The appropriate parameters for optimization of a neural network can be found by minimizing the loss function described in the previous section. To achieve the optimal value, the *backpropagation* algorithm is used, which calculates the gradient of the loss function with respect to the weights and the bias of a network. It iterates through the layers backwards — right after the data have been forward passed — calculating the gradients for each neuron of each layer. By applying the chain rule, it is possible to make these calculations for every layer and not only the last one [9].

Let:

- a^ℓ — the output (activation) of layer ℓ
- W^ℓ — the weight matrix connecting layer $\ell - 1$ to layer ℓ
- b^ℓ — the bias vector of layer ℓ
- δ^ℓ — the error of layer ℓ
- $\sigma(\cdot)$ — the activation function
- L — the loss function

The error at the output layer L is calculated as:

$$\delta^L = \frac{\partial L}{\partial a^L} \circ \sigma'(z^L) \quad (1)$$

where \circ denotes element-wise multiplication and $z^L = W^L a^{L-1} + b^L$ is the input to the activation function at the output layer.

For any hidden layer ℓ , the error is computed by propagating the error backward:

$$\delta^\ell = ((W^{\ell+1})^\top \delta^{\ell+1}) \circ \sigma'(z^\ell) \quad (2)$$

The gradient of the loss with respect to the biases of layer ℓ is simply:

$$\frac{\partial L}{\partial b^\ell} = \delta^\ell \quad (3)$$

The gradient of the loss with respect to the weights of layer ℓ is:

$$\frac{\partial L}{\partial W^\ell} = \delta^\ell (a^{\ell-1})^\top \quad (4)$$

These formulas are the core of the backpropagation algorithm and are used to compute the gradients needed for gradient descent updates.

4.3 Optimization and training

The neural network can be trained with an optimizer. Training means finding the appropriate weights and biases for the whole network. Thus, optimization means updating weights and biases. Updates can be applied in many ways, but in this project two algorithms for optimization are used: SGD and Adam.

4.3.1 SGD

Stochastic gradient descent is a time-consuming algorithm, since new weights are computed only after the end of an epoch, meaning that they converge slowly. However, SGD accurately follows the path to optimization making it a preferable algorithm. Rather than computing the gradient over the whole dataset, this algorithm computes the gradient using a mini-batch, which makes updates more frequent and fast. This implies some drawbacks: fluctuations during the training process due to the high variance make the process less stable, and a need of a careful tuning of a learning rate. Let:

- θ - model parameters (weights, biases), which we want to update
- η - learning rate
- $\nabla_{\theta} L$ - gradient of the loss with respect to θ

Hence,

$$\theta := \theta - \eta \cdot \nabla_{\theta} L$$

Algorithm 1 Stochastic Gradient Descent (SGD)

Require: Initial parameters θ_0 , learning rate η

```

1:  $t \leftarrow 0$  ▷ Initialize timestep
2: while  $\theta_t$  not converged do
3:    $t \leftarrow t + 1$  ▷ Increment timestep
4:   Sample mini-batch  $\mathcal{B}_t$  ▷ Draw random mini-batch
5:    $g_t \leftarrow \nabla_{\theta} L(\theta_{t-1}; \mathcal{B}_t)$  ▷ Compute gradient on mini-batch
6:    $\theta_t \leftarrow \theta_{t-1} - \eta \cdot g_t$  ▷ Update parameters
7: end while
8: return  $\theta_t$  ▷ Return final parameters

```

4.3.2 Adam

Adaptive moment estimation is “an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments. The method is straightforward to implement, is computationally efficient, has little memory requirements, is invariant to diagonal rescaling of the gradients, and is well suited for problems that are large in terms of data and/or parameters. The method is also appropriate for non-stationary objectives and problems with very noisy and/or sparse gradients. The hyper-parameters have intuitive interpretations and typically require little tuning.” [4]

Adam algorithm has two key features: momentum and adaptive learning rates. The former means an optimization technique used to speed up gradient descent and to reduce oscillations and help with convergence. It is used to speed up training in flat regions and to prevent getting stuck in a local minima. The latter means adjusting the learning rate per parameter. In a simple gradient descent the learning rate η is the same for all parameters and stays the same unless it is manually corrected, however parameters may have different convergence speed and some of them may oscillate - this issues are solved using adaptive optimizers. Adam uses moving average of gradients and squares of gradients:

$$\begin{aligned}
 g_t &= \nabla_{\theta} L(\theta_t) && \text{— gradient on a step } t \\
 m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t && \text{— first moment vector (average)} \\
 v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 && \text{— second moment vector (variance)} \\
 \hat{m}_t &= \frac{m_t}{1 - \beta_1^t} && \text{— adjusted bias} \\
 \hat{v}_t &= \frac{v_t}{1 - \beta_2^t} \\
 \theta_t &= \theta_{t-1} - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}
 \end{aligned}$$

where:

- β_1, β_2 — exponential decay rates for moment estimates (usually 0.9 and 0.999)

- ϵ — small constant (usually 10^{-8})

Algorithm 2 Adam Optimizer

Require: Initial parameters θ_0 , stepsize η , decay rates $\beta_1, \beta_2 \in [0, 1)$, small constant ϵ

```

1: Initialize  $m_0 \leftarrow 0, v_0 \leftarrow 0, t \leftarrow 0$ 
2: while  $\theta_t$  not converged do
3:    $t \leftarrow t + 1$ 
4:    $g_t \leftarrow \nabla_{\theta} L(\theta_{t-1})$  ▷ Compute gradient
5:    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$  ▷ 1st moment estimate
6:    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$  ▷ 2nd moment estimate
7:    $\hat{m}_t \leftarrow \frac{m_t}{1 - \beta_1^t}$  ▷ Bias-corrected 1st moment
8:    $\hat{v}_t \leftarrow \frac{v_t}{1 - \beta_2^t}$  ▷ Bias-corrected 2nd moment
9:    $\theta_t \leftarrow \theta_{t-1} - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$  ▷ Update parameters
10: end while
11: return  $\theta_t$ 

```

Hyperparameters are those parameters, that affect the learning process and can be chosen by the user. The choice must be wise - number of layers and neurons in each of them, η and others.

5 System architecture

5.1 Structure

The system architecture of this project has modular design. Sources folder is divided into logical parts - classes, of which the network consists. Each directory in the `src/` folder corresponds to a specific functionality of the neural network pipeline:

```
ProjectRoot/  
src/  
  ActivationFunctions/  
    ActivationFunction.cpp ..... Activation functions implementation  
    ActivationFunction.h ..... Activation functions interface  
  Layers/  
    Layer.cpp ..... Layer logic implementation  
    Layer.h ..... Layer class definition  
  Loader/  
    MNISTLoader.cpp ..... MNIST dataset loading logic  
    MNISTLoader.h ..... MNIST loader interface  
  LossFunctions/  
    LossFunction.cpp ..... Loss function implementation  
    LossFunction.h ..... Loss function interface  
  Model/  
    Model.cpp ..... Network model logic  
    Model.h ..... Network model interface  
  Optimizer/  
    Optimizer.cpp ..... Optimization algorithm logic  
    Optimizer.h ..... Optimizer interface  
  Tests/  
    Tests.cpp ..... Unit tests implementation  
    Tests.h ..... Test interface/headers  
  Utilities/  
    FileReader.cpp ..... File reading logic  
    FileReader.h ..... File reader interface  
    FileWriter.cpp ..... File writing logic  
    FileWriter.h ..... File writer interface  
    Random.cpp ..... Random number generator logic  
    Random.h ..... RNG interface  
    Utils.h ..... Utility functions  
  main.cpp ..... Entry point of the application
```

Figure 3: Directory structure of the project source code

- **ActivationFunctions:** Implements and exposes common activation functions. Has two methods - `apply`, which applies the activation function to the vector element-wise and `derivative`, which computes the derivative. Static factory methods are used to concretize predefined activation types or create one from an enumerated `Type`.

```
1 Vector apply(const Vector &x) const;  
2 Vector derivative(const Vector &x) const;  
3  
4 static ActivationFunction ReLU();  
5 static ActivationFunction Sigmoid();  
6 static ActivationFunction Identity();  
7 static ActivationFunction Tanh();  
8  
9 static ActivationFunction create(Type type);
```

- **Layers:** Contains definitions of network layers and logic for forward and backward propagation. Stores cache for backpropagation and Adam optimizer. `forward` is used only during training and it fills the cache. And `predict` does not save anything to the cache, but simply calculates the output of the neural network, and so the `predict` should be `const`.

```

1  Vector forward(const Vector &input);
2  Vector predict(const Vector &input) const;
3
4  Vector backward(const Vector &grad_output, const Optimizer &optimizer);
5
6  void setCache(const Optimizer &opt);
7  void freeCache();

```

- **Loader:** Responsible for reading and parsing MNIST dataset files into a usable format. It opens the files with data using binary streams, reads the headers to extract metadata, reads the pixel data and the label data and stores the result in vectors. Also this file includes normalization: pixel values are in range from 0 to 255, the program divides each value by 255.0 in order to speed up the training process and make models converge more easily.

```

1  bool loadMNIST(const std::string &image_file, const std::string &label_file,
2  std::vector<Vector> &images, std::vector<int> &labels);

```

- **LossFunctions:** Provides two types of loss functions such as Mean squared error and Cross-entropy. Computes their gradients. SGD and Adam use the computed value of loss function to update weights to reduce the error.

```

1  static double mse(const Vector &y_pred, const Vector &y_true);
2  static Vector mseGrad(const Vector &y_pred, const Vector &y_true);
3
4  static double crossEntropy(const Vector &y_pred, const Vector &y_true);
5  static Vector crossEntropyGrad(const Vector &y_pred, const Vector &y_true);

```

- **Model:** Serves as the core component that defines, manages and trains the whole neural network, encapsulates its structure and logic, connects multiple layers together and coordinated their operation during the forward and backward pass. Keeps a vector of `Layer` objects and sequentially propagates the input through all layers.

```

1  Model(
2      std::initializer_list<size_t> layer_sizes,
3      std::initializer_list<ActivationFunction::Type> activations);
4
5  Vector forward(const Vector &input);
6
7  void trainStep(
8      const Vector &x, const Vector &y,
9      const std::function<Vector(const Vector &, const Vector &)> &lossGrad,
10     Optimizer &optimizer);
11
12  void train(
13      const std::vector<Vector> &xs,
14      const std::vector<Vector> &ys,
15      int epochs,
16      LossFunction loss,
17      Optimizer &optimizer);
18
19  const std::vector<Layer> &layers() const;

```

- **Optimizer:** Includes two optimization algorithms - SGD and Adam. Has methods to update parameters - weights and biases.

```

1  static Optimizer SGD(double lr);
2

```

```

3  static Optimizer Adam(double lr, double beta1 = 0.9, double beta2 = 0.999,
4  double eps = 1e-8);
5
6  void update(Matrix &param, std::any &cache, const Matrix &grad) const;
7  void update(Vector &param, std::any &cache, const Vector &grad) const;
8  std::any init_cache(int rows, int cols) const;

```

- **Tests:** Unit test definitions for checking module correctness.

```

1  void runAllTests() {
2      if (testActivationFunction() == TestStatus::Error)
3          return;
4      if (testOptimizerSGD() == TestStatus::Error)
5          return;
6      if (testOptimizerAdam() == TestStatus::Error)
7          return;
8      if (testLayerForwardBackward() == TestStatus::Error)
9          return;
10
11     std::cout << "[OK] All tests passed!\n";
12 }

```

- **Utilities:** Contains file readers and writers, random number generators using EigenRand - normal and uniform, and helper functions. Here is the declaration of methods in class Random.

```

1  Matrix uniformMatrix(Index rows, Index cols, double a, double b);
2  Vector uniformVector(Index size, double a, double b);
3
4  Matrix normalMatrix(Index rows, Index cols, double mean, double stddev);
5  Vector normalVector(Index size, double mean, double stddev);

```

- **main.cpp:** The main entry point that launches training and evaluation. Here is the initialization of a Model.

```

1  Optimizer opt = Optimizer::Adam(0.001, 0.9, 0.999, 1e-8);
2
3  Model model({784, 128, 10},
4      {ActivationFunction::Type::ReLU,
5      ActivationFunction::Type::Identity});

```

5.2 Data flow and complexity considerations

Data flows through the system during a forward pass. Starting from the input, each layer takes data from its previous layer, processes it, and passes it to the next layer. For example, in a forward pass for one image: the input pixel values of size 784 (after normalization) are loaded into the first Layer, which outputs a vector of length equal to its neurons (128 in our example), that vector is going through a linear transformation and then through the activation function. Next, that goes into the second Layer which outputs 10 values (one per digit). Finally, the Loss function (outside layers) computes a value comparing those 10 outputs to the true label.

During the backpropagation, the flow of data is reversed. Firstly, the loss gradient is computed at the output layer, then is passed backward into the last Layer's backward method. That Layer uses it to update weights and also produces a gradient with respect to its inputs, which then becomes the input to the previous Layer's backward method, that modifies that gradient and passes it further back to the first Layer. The first Layer then updates its weights and computes gradients to pass to input (which would end there since input layer has no preceding layer). Throughout this backward flow, each layer is responsible for updating its own parameters using the gradients and the learning rate.

The following complexities are provided:

- N : number of training samples (e.g., 60000)

- L : number of layers
- d_l : number of neurons in layer l
- $W = \sum_{l=1}^L d_l \cdot d_{l-1}$: total number of weights
- $B = \sum_{l=1}^L d_l$: total number of biases

Step	Metric	General Case	Model (784–128–10)
Forward Pass	Time Complexity	$\mathcal{O}\left(N \cdot \sum_{l=1}^L d_l \cdot d_{l-1}\right)$	$\mathcal{O}(N \cdot (784 \cdot 128 + 128 \cdot 10))$
Backward Pass	Time Complexity	$\mathcal{O}\left(N \cdot \sum_{l=1}^L d_l \cdot d_{l-1}\right)$	$\mathcal{O}(N \cdot (784 \cdot 128 + 128 \cdot 10))$
Parameter Update (SGD)	Time Complexity	$\mathcal{O}(W + B)$	$\mathcal{O}(100490)$
Parameter Update (Adam)	Time Complexity	$\mathcal{O}(6 \cdot (W + B))$	$\mathcal{O}(600000)$
Memory Usage (SGD)	Space Complexity	$\mathcal{O}(W + B)$	$\mathcal{O}(100490)$
Memory Usage (Adam)	Space Complexity	$\mathcal{O}(3 \cdot (W + B))$	$\mathcal{O}(301470)$

6 Experimental evaluation

The program was tested in the file `main.cpp` with console output. Testing consisted of the following parts:

1. **Run unit tests:** Before the training starts, the program calls `test::runAllTests()` to verify the correctness of all internal components. If all unit tests are passed, the output is: `[OK] All tests passed !`, otherwise the program will output the local bug, which occurred on a unit test.
2. **Data loading:** Using `loadMNIST()`, the program loads the data from MNIST dataset containing 60,000 images of size 28 x 28 pixels each, they are flattened to vectors of size 784 and labels are converted into one-hot encoded vectors of size 10. The resulting vectors are used as training targets.
3. **Model setup:** The tested model consists of 3 layers: one input with 784 neurons, one hidden with 128 neurons and one output with 10 neurons. The hidden layer uses **ReLU** activation function and the output layer uses **Identity** activation function. Optimization algorithm is **Adam** with default hyperparameters ($lr = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 1e-8$). MSE was chosen as a loss function.
4. **Training:** The model trained 1 epoch using the entire training set and for each sample the model performed forward and backward passes through `trainStep()`. The loss is computed and reported. A progress bar is rendered in the console with current loss and completion percentage.
5. **After training:** The model is evaluated on a MNIST test set consisting of 10,000 images. For each of them, the output vector is computed using a forward pass. Then the number of correct predictions is counted and the test accuracy is computed as a percentage.

In order to test the program, write in terminal the following commands:

- Open the folder with the project.

```
1 cd /path/to/Neural_Networks_Course_Project1
```

- Build the project.

```
1 mkdir build
2 cd build
3 cmake ..
4 make
```

- Run the program.

```
1 ./neural_net
```

After the execution, the resulting output will be:

```
1 [OK] All tests passed!
2 Train size: 60000
3
4 === Training (1 epoch) ===
5 [=====] 100% (60000/60000) L:0.0343
6
7 === Training finished ===
8 Test size: 10000
9
10 Test Accuracy: 88.7600%
```

In order to track the changes in the average value of the loss function, it was tracked in the training cycle in the `main.cpp` file and written in the `loss file`:

```
1 loss_file << (i + 1) << ", " << avgLoss << "\n";
```

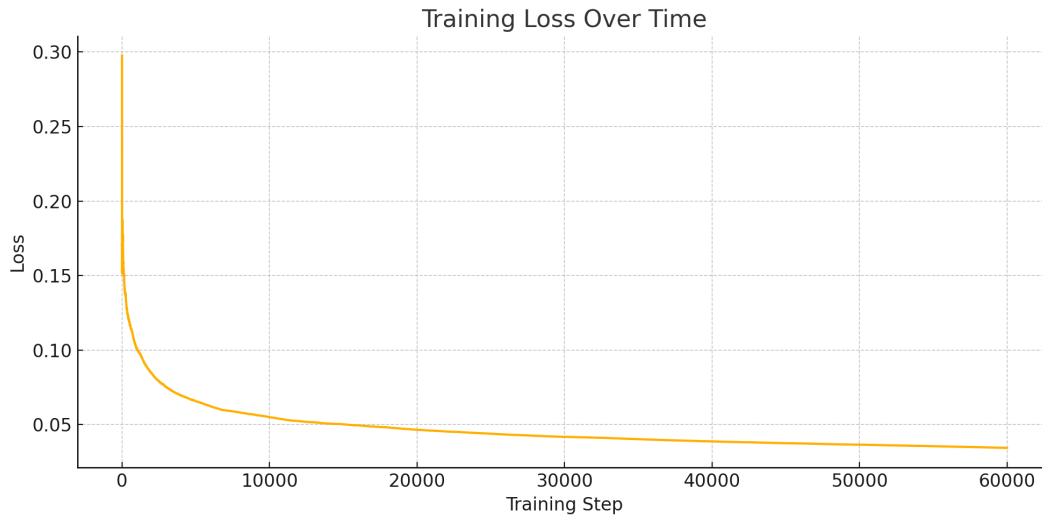



Figure 4: Loss vs Step plot during training

Also, a *confusion matrix* was implemented in order to track, what digits does the neural network confuse most. Element $[i][j]$ shows how much the digit i was confused with the digit j . In `main.cpp` it was implemented through writing into the file:

```
1 cm_file << truth << "," << predicted << "\n";
```

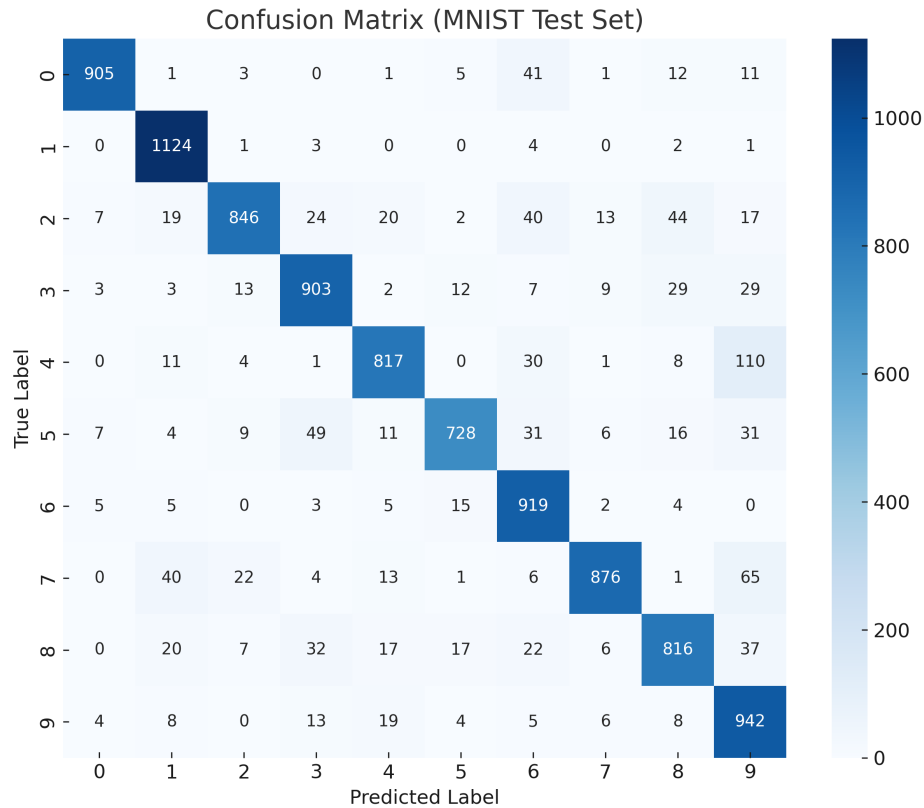


Figure 5: Confusion matrix for the MNIST test set.

7 Conclusion

In this course project a fully-connected neural network was implemented from scratch in C++. The work combined features of machine learning theory with practical software development, which resulted in a working library of a neural network capable of learning and improving accuracy.

The main **achievements** of this project are:

1. Implementation of forward and backward propagation algorithms manually using Eigen for computations.
2. Implementation of several types of Activation functions, Loss functions and Optimizers.
3. Testing and demonstrating the network's effectiveness on a MNIST dataset, achieving almost 90% accuracy after just one epoch of learning.
4. Using modern features of C++, Eigen and EigenRand to increase the performance of the code.

The main **challenges** of this project are:

1. Debugging the backpropagation implementation - a small mistake in the formula of the Adam algorithm led to the absence of learning - the accuracy tested was 7.5%.
2. Performance tuning - a console output in the training loop was slowing things down significantly.

The **future work** on the project includes integrating a graphical user interface (GUI), adding Softmax activation function to combine in with the Cross-entropy loss function, and adding Dropout layer for regularization to avoid overfitting in the future.

Beyond the quantitative results, this project provided a deeper insight into an inner construction of a neural network. By reconstructing each step (from computing weighted sums to propagating gradients), the understanding of the learning process was reinforced. The exercise of coding everything from scratch also highlighted the importance of various design choices — for instance, how the choice of activation function or the setting of a learning rate can drastically impact training outcomes.

The completion of this project provides a great basis for further exploration. The essential components of the complete system are built, opening the possibility of tackling more complex network architectures or incorporate this network as part of more extensive systems.

References

- [1] Eigen. Eigen: A c++ template library for linear algebra. 2024.
- [2] EigenRand. Eigenrand : The fastest c++11-compatible random distribution generator for eigen. 2024.
- [3] Fast Artificial Neural Network. Unveiling the world of fast artificial neural networks. <https://leenissen.dk/>, 2025. Accessed: 2025-05-01.
- [4] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [5] PyTorch. An open source machine learning framework. <https://pytorch.org/>, 2025. Accessed: 2025-05-01.
- [6] Suvorova Alexandra. Neural networks from scratch. https://github.com/Lunciare/Neural_Networks_Course_Project1, 2025.
- [7] TensorFlow. An end-to-end platform for machine learning. <https://www.tensorflow.org/>, 2025. Accessed: 2025-05-01.
- [8] Yandex Practicum. Первое знакомство с полносвязными нейросетями. <https://education.yandex.ru/handbook/ml/article/pervoe-znakomstvo-s-polnosvyaznymi-nejrosetyami>, 2024. Accessed: 2025-05-01.
- [9] Adamantios Zaras, Nikolaos Passalis, and Anastasios Tefas. Neural networks and backpropagation. In *Deep learning for robot perception and cognition*, pages 17–34. Elsevier, 2022.