Shallow Neural Network implemented in C++ for Handwritten Digits Classification

Alberto A. González, Octavio T. Delgado; *UASLP*

Over the last decades machine learning has become very popular since new algorithms were developed and more complex implementations of neural networks became possible, opening a new world of possibilities and solutions. Nowdays is very common to use tools and software that already have machine learning algorithms implemented, and people only need to know the necessary theory to acomplish systems that learn and do what they need. Still, when working with machine learning it is very important to understand the basics and go deeper in the algorithms so the student gets a better understanding of this mathematical tools and later he can get more involved with AI without much trouble. This is why we decided to implement a Shallow Neural Network from scratch in C++ that learns how to classify handwritten digits with 84.42% precision. We did not use any machine learning library and everything was implemented in C++.

Index Terms—**Neural Network, handwritten digits, machine learning, C++, MNIST, classification.**

I. INTRODUCTION

THE use of neural networks has become one of the main tools when trying to solve problems that cannot be solved using mathematical models and algorithms. Back in the late 50's the perceptron was introduced by Frank Rosenblatt and the era of Artificial Intelligence started to come to life. Since then, many different tools were developed and neural networks seemed to be the solution to almost all unsolved problems. Not much time went by and researchers noticed that neural networks were not perfect, in fact, they had a problem when the architecture had more than two layers. This problem prevented the neural networks to learn, basically because it was not possible to update the weights and biases. Almost thirty years had to pass for researchers to develop the backpropagation algorithm, which revived neural networks. Now, thanks to this algorithm and some other major improvements, there are Deep Neural Networks that have more than two hidden layers and can solve very complex problems.

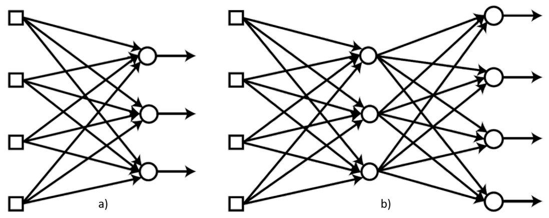


Fig. 1: a) Single-Layer Neural Network. b) Shallow Neural Network.

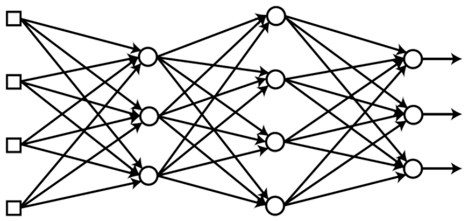


Fig. 2: Deep Neural Network.

Neural Networks are classified in two main categories: Single-Layer Neural Networks and Multi-Layer Neural Networks. As its name implies, Single-Layer NN have only one input layer and one output layer, contrary to Multi-Layer NN, which have one hidden layer or more. If a Multi-Layer NN has only one hidden layer, then it is known as a *Shallow Neural Network*, but if it has more than one, then it is consider a Deep Neural Network. We implemented a Shallow Neural Network for handwritten digits classification, which means that the architecture used is shown in Fig. 1-b).

Usually when working with machine learning Python is the preferred programming language. This is due to all the libraries already implemented that make everything very simple for the programmer and results can be obtained fast. On the contrary, C++ does not have so many machine learning libraries and the community tend to use this language for other purposes. So, why to bother implementing a neural network in C++? The answer is simple: to better understand basic neural networks. Due to the lack of libraries, we were forced to implement almost every single function needed for the neural network to work in C++. We only used two basic libraries from the STL for random numbers generation with normal Gaussian distribution and also to randomly shuffle some arrays. It is also important to mention that we used supervised training for the classification.

II. ALGORITHMS AND THEORETICAL FRAMEWORK

A. Handwritten Digits

For this problem we used the MNIST dataset which has a training set of 60,000 handwritten digits each with its corresponding label, and 10,000 different digits for testing, also with labels. All digits were normalized to fit in a 28x28 pixel image, each pixel with values from 0 to 255 (gray scale); see Fig. 3. The dataset is free and can be download as four binary files, two containing the training data and the other two containing the test data. It was necessary to implement in C++ a program (then used as a class) specifically made for reading this binary files and format the data properly for the neural network. Later we will discuss this in section IV.

Fig. 3: Handwritten digits from MNIST dataset.

B. Shallow Neural Network

We already mention that a shallow neural network was implemented for the classification. Now we explain without much detail the algorithms used for training the neural network and the necessary theory for the implementation in C++. We will not focus to much in the mathematics because the aim of this project was not the study of neural networks, but the implementation of the code in C++. Still, it is important to include this as theoretical framework.

1) Sigmoid Neurons

The neural network is made of sigmoid neurons, which are very similar to perceptrons, but modified so that small changes in their weights and bias cause only a small change in their output. A sigmoid neuron is represented in Fig. 4.

Perceptrons are represented just like sigmoid neurons and also have inputs $x_1, x_2, x_3, ..., x_n$, but the difference is that perceptrons can only have binary inputs (0 or 1),

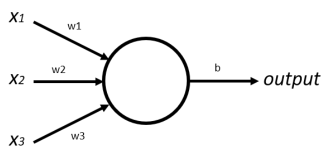


Fig. 4: Sigmoid neuron representation.

where sigmoid neurons can also take any value between 0 and 1. This means that 0.7621 is a valid input for a sigmoid neuron. Also, just like perceptrons, sigmoid neurons have weights and biases, but the output is not 0 or 1, it is $\sigma(w \cdot x + b)$, where σ is called the sigmoid function and is defined by:

$$\sigma(z) \equiv \frac{1}{1+e^{-z}} \tag{1}$$

Knowing this, then the output of a sigmoid neuron with inputs $x_1, x_2, x_3, ..., x_j$, weights $w_1, w_2, w_3, ..., w_j$ and bias b is defined by:

$$\frac{1}{1+e^{-\{\sum_j w_j \cdot x_j - b\}}} \tag{2}$$

2) Stochastic Gradient Descent

We use the notation x to denote a training input. As mention before, each image is a 28x28 pixel matrix, but we need to vectorize it so we have a $28 \times 28 = 784$-dimensional vector, where its elements are values representing each pixel in the image. Also, we denote the desired output of the neural network as $y = y(x)$, where y is a 10-dimensional vector. For example, suppose we have an image x containing the handwritten digit 7, then the desired output of the network will be also 7, and this is represented as $y(x) = [0, 0, 0, 0, 0, 0, 0, 1, 0, 0]^T$. Knowing this we define the cost function by:

$$C(w, b) \equiv \frac{1}{2n} \sum_x ||y(x) - a||^2 \tag{3}$$

$$a = \sigma(w \cdot x + b) \tag{4}$$

where w denotes the collection of all weights, b all the biases, n the total number of training inputs, a the vector of outputs from the network when x is input, and the sum is over all training inputs, x. We can see that the cost function is nothing else than the *mean squared error*, a very common metric and also easy to understand. Still, the MSE is probably the best choice for this problem and there are studies showing that this is the best option. Now we need to find the set of weights and biases which make the cost as small as possible, and this is achieved using the *gradient descent algorithm*.

The gradient descent algorithm is explained without much detail. We need to minimize $C(w,b)$, but for explanation purposes we will use a function $C(v)$ of two variables, which will be called v_1 and v_2. We want to find where C achieves its global minimum, but we cannot use calculus because neural networks usually have lots of variables (not just two), and this becomes to complex to solve. So, lets think of this function C as a valley, and we imagine a ball rolling down its slope. Eventually this ball will roll down to the bottom of the valley, where it should be the minimum. We use this analogy for the computation of the minimum.

Let's think about what happens when we move the ball a small amount Δv_1 in the v_1 direction and a small amount Δv_2 in the v_2 direction. Calculus tells us that C changes as follows:

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2 \qquad (5)$$

We now find a way of choosing Δv_1 and Δv_2 so as to make ΔC negative, which implies that the ball is rolling down into the valley, and not up. Now, expression (5) is rewritten as:

$$\Delta C \approx \nabla C \cdot \Delta v \qquad (6)$$

where

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T \qquad (7)$$

$$\Delta v \equiv (\Delta v_1, \Delta v_2)^T \qquad (8)$$

Expression (6) helps explain why ∇C is called the gradient vector: ∇C relates changes in v to changes in C, which is something a gradient does. Now we can choose Δv so as to make ΔC negative. We choose:

$$\Delta v = -\eta \nabla C \qquad (9)$$

where η is a small, positive parameter known as the learning rate. Now, substituting (9) in (6) we have $\Delta C \approx -\eta||\nabla C||^2$ which means C will always decrease. Finally, we define an update rule that makes C decrease until it reaches the global minimum by:

$$v \rightarrow v' = v - \eta \nabla C \qquad (10)$$

Obviously, we will not be working with a cost function of just two variables, but everything explained above works perfectly well with m variables. Now, using the expression (10) we define the update rules for the weights and biases of the neural network as follows:

$$w_k \rightarrow w'_k = w_k - \eta \frac{\partial C}{\partial w_k} \qquad (11)$$

$$b_k \rightarrow b'_k = b_k - \eta \frac{\partial C}{\partial b_k} \qquad (12)$$

Now, with these rules we can make the neural network learn, but it will take to long to compute when implemented in code. This is due because in practice we need to compute the gradients ∇C_x separately for each training input, x, and then average them: $\nabla C = \frac{1}{n}\sum_x \nabla C_x$. Unfortunately, this is very slow when dealing with too much data, and that is why we need to implement the *stochastic gradient descent*.

The stochastic gradient descent works by randomly picking out a small number m of randomly chosen training inputs. We name those random training inputs $X_1, X_2, X_3, ..., X_m$ and refer to them as a mini-batch. If m is large enough we expect that the average value of the ∇C_{x_j} will be roughly equal to the average over all ∇C_x, that is:

$$\frac{\sum_{j=1}^{m} \nabla C_{x_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C \qquad (13)$$

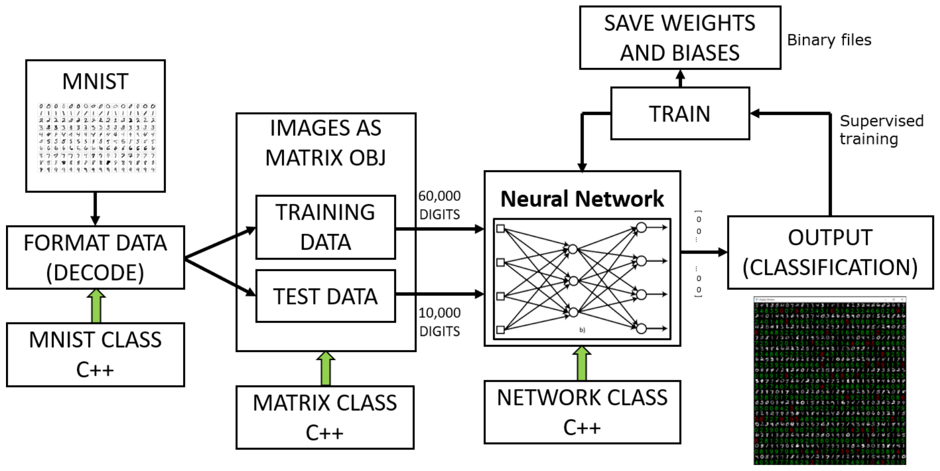


Fig. 5: Block diagram of whole implementation in C++.

$$\nabla C = \frac{1}{m}\sum_{j=1}^{m}\nabla C_{x_j} \tag{14}$$

Applying this to the neural network we have the following update rules:

$$w_k \rightarrow w'_k = w_k - \frac{\eta}{m}\sum_j \frac{\partial C_{X_j}}{\partial w_k} \tag{15}$$

$$b_l \rightarrow b'_l = b_l - \frac{\eta}{m}\sum_j \frac{\partial C_{X_j}}{\partial b_l} \tag{16}$$

where the sums are over all the training examples X_j in the current mini-batch. This will be repeated until finally the neural network is trained, with each mini-batch been part of an epoch.

3) Backpropagation algorithm

We tried to summarize the mathematical theory of the stochastic gradient descent above, but we will not do the same for the backpropagation algorithm. It is not an easy algorithm to understand and it would take a considerably part of this document if we try to explain it the best we can. So we decided to provide only the equations of backpropagation and the steps for the implementation in code. The reader may find a lot of information of this algorithm in [3] if interested. The backpropagation equations are:

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{17}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1} \odot \sigma'(z^l)) \tag{18}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{19}$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \tag{20}$$

where $\odot$ represents the Hadamard product (elementwise product) and δ^l_j the error in the j^{th} neuron in the l^{th} layer.

Now we show those equations in the form of an algorithm that can be implemented in any programming language:

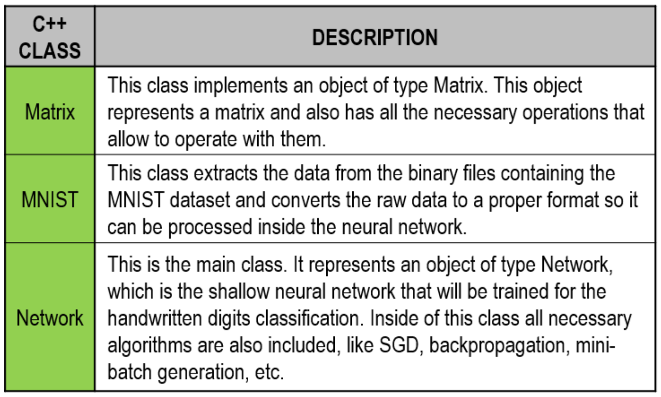
1) **Input** x**:** Set the corresponding activation a^l for the input layer.
2) **Feedforward:** For each $l = 2, 3, ..., L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$.
3) **Output error** δ^L**:** Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$.
4) **Backpropagate the error:** For each $l = L-1, L-2, ..., 2$ compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$.
5) **Output:** The gradient of the cost function is given by $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ and $\frac{\partial C}{\partial b^l_j} = \delta^l_j$.

If we analyze this algorithm, after some thinking, we can see why it is called backpropagation. We are computing the error vectors δ^l backward, starting from the final layer.

With everything said, now we have all the mathematical tools needed for the implementation of a shallow neural network in C++. We know that there is still a lot of information about neural networks to cover, but it is out of the scope of this document.

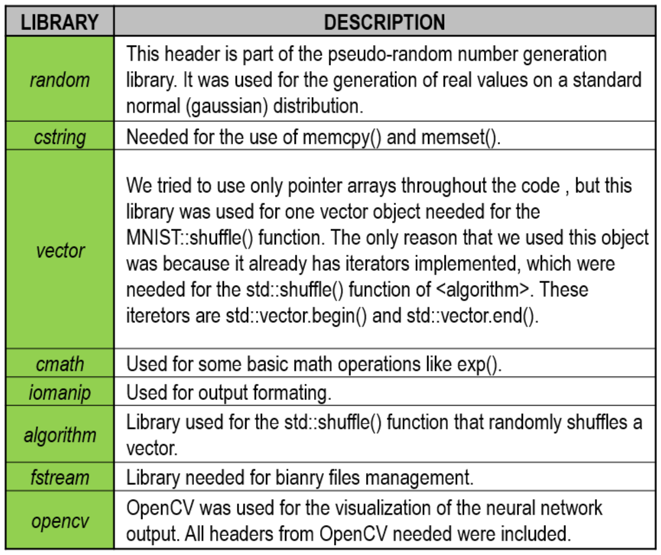
III. Software Design

The final design consists of only three classes: *Matrix*, *MNIST* and *Network*. The description of each one of them is shown in Fig. 6.



C++ CLASS	DESCRIPTION
Matrix	This class implements an object of type Matrix. This object represents a matrix and also has all the necessary operations that allow to operate with them.
MNIST	This class extracts the data from the binary files containing the MNIST dataset and converts the raw data to a proper format so it can be processed inside the neural network.
Network	This is the main class. It represents an object of type Network, which is the shallow neural network that will be trained for the handwritten digits classification. Inside of this class all necessary algorithms are also included, like SGD, backpropagation, mini-batch generation, etc.

Fig. 6: Description of all three implemented classes.



LIBRARY	DESCRIPTION
random	This header is part of the pseudo-random number generation library. It was used for the generation of real values on a standard normal (gaussian) distribution.
cstring	Needed for the use of memcpy() and memset().
vector	We tried to use only pointer arrays throughout the code , but this library was used for one vector object needed for the MNIST::shuffle() function. The only reason that we used this object was because it already has iterators implemented, which were needed for the std::shuffle() function of <algorithm>. These iteretors are std::vector.begin() and std::vector.end().
cmath	Used for some basic math operations like exp().
iomanip	Used for output formating.
algorithm	Library used for the std::shuffle() function that randomly shuffles a vector.
fstream	Library needed for bianry files management.
opencv	OpenCV was used for the visualization of the neural network output. All headers from OpenCV needed were included.

Fig. 7: Library's description.

As we already mentioned, we tried to avoid the use of libraries for the neural network implementation, but still it was necessary to use some external libraries. These libraries are shown in Fig. 7. Common libraries were omitted, like *iostream*, and *string*.

A block diagram of the system is shown in Fig. 5 for a better understanding of what was done. Basically, we use the MNIST class to extract the dataset from the binary files and then format the images and labels with the help of Matrix class, which also has the operators needed later in the neural network for the training and feedforward section. This images and labels are used as inputs for the training process (supervised learning) and finally the network outputs the desired classification.

Finally, we show in Fig. 8 an UML diagram containing the three classes and its relationship between them. As we can see, no inheritance was used and only composition was necessary.

IV. IMPLEMENTATION

It is not possible to explain here all the code because the document would be too long, that is why we only explain the most important part of the whole implementation. We start with the Matrix class, then the MNIST class and finally the Network class.

A. Matrix Class

This class represents a matrix object and also implements all necessary operators. It has three constructors and its destructor because it uses dynamic memory allocation. Below is the header file of this class:

```cpp
class Matrix {
  int n;       //number of rows
  int m;       //number of columns
  float* matrix;  //Matrix elements
public:
  Matrix();
  Matrix(int, int);
  Matrix(const Matrix& mtx);
  ~Matrix();
  Matrix operator+(const Matrix& mtx);
  Matrix operator-(const Matrix& mtx);
  Matrix operator*(const Matrix& mtx);
  Matrix operator^(const Matrix& mtx);
  Matrix& operator=(const Matrix& rhs);
  Matrix operator~();
  float& operator[](int);
  friend Matrix operator*(const Matrix& mtx,
    const float& scalar);
  friend Matrix operator*(const float& scalar,
     const Matrix& mtx);
  friend std::ostream& operator<<(std::ostream
    & os, const Matrix& mtx);
  int getSize() const;
  int getRows();
  int getColumns();
  void zeros();
  void saveMatrix(std::string name);
  void readMatrix(std::string name);
};
```

Listing 1: Matrix.h

This class is very easy to understand and the code is self-explanatory when reading directly on the IDE. We can see on the header file that all operations have its own overloaded operator. Also, this class is in charge of reading and saving the weights and biases already

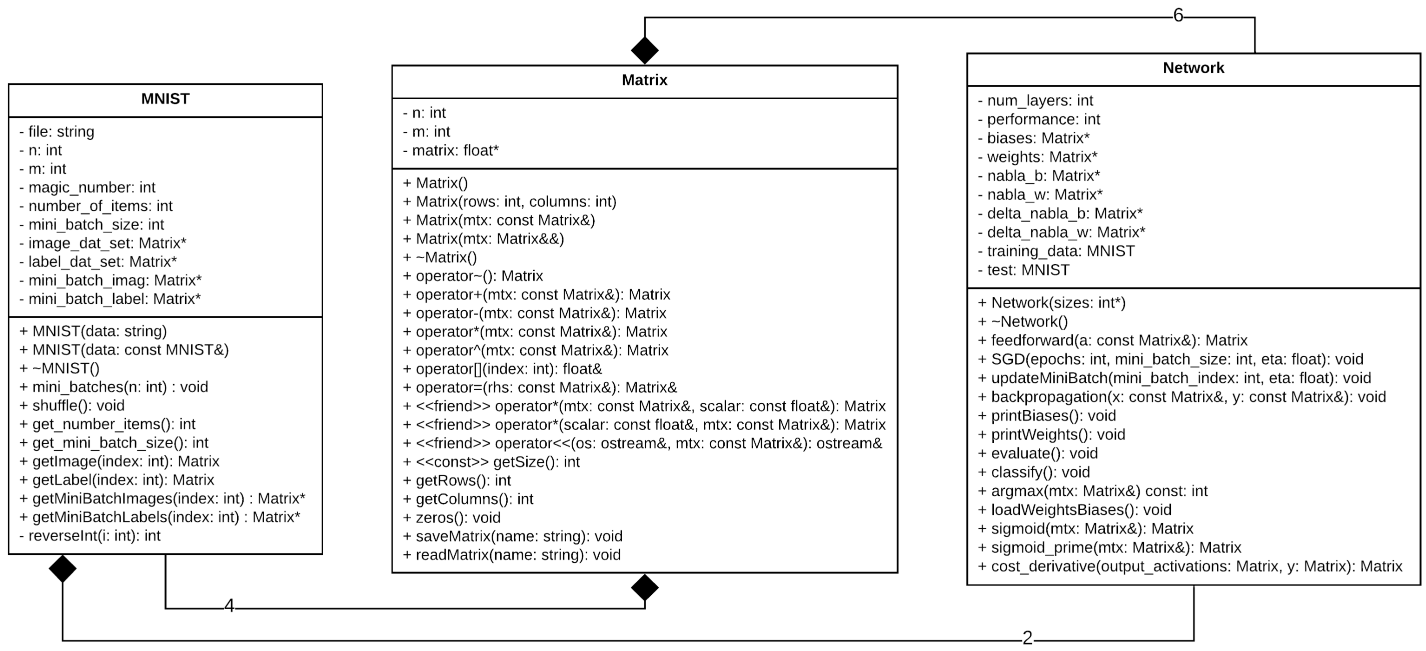

MNIST
- file: string
- n: int
- m: int
- magic_number: int
- number_of_items: int
- mini_batch_size: int
- image_dat_set: Matrix*
- label_dat_set: Matrix*
- mini_batch_imag: Matrix*
- mini_batch_label: Matrix*
+ MNIST(data: string)
+ MNIST(data: const MNIST&)
+ ~MNIST()
+ mini_batches(n: int) : void
+ shuffle(): void
+ get_number_items(): int
+ get_mini_batch_size(): int
+ getImage(index: int): Matrix
+ getLabel(index: int): Matrix
+ getMiniBatchImages(index: int) : Matrix*
+ getMiniBatchLabels(index: int) : Matrix*
- reverseInt(i: int): int
Matrix
- n: int
- m: int
- matrix: float*
+ Matrix()
+ Matrix(rows: int, columns: int)
+ Matrix(mtx: const Matrix&)
+ Matrix(mtx: Matrix&&)
+ ~Matrix()
+ operator~(): Matrix
+ operator+(mtx: const Matrix&): Matrix
+ operator-(mtx: const Matrix&): Matrix
+ operator*(mtx: const Matrix&): Matrix
+ operator^(mtx: const Matrix&): Matrix
+ operator[](index: int): float&
+ operator=(rhs: const Matrix&): Matrix&
+ <<friend>> operator*(mtx: const Matrix&, scalar: const float&): Matrix
+ <<friend>> operator*(scalar: const float&, mtx: const Matrix&): Matrix
+ <<friend>> operator<<(os: ostream&, mtx: const Matrix&): ostream&
+ <<const>> getSize(): int
+ getRows(): int
+ getColumns(): int
+ zeros(): void
+ saveMatrix(name: string): void
+ readMatrix(name: string): void
Network
- num_layers: int
- performance: int
- biases: Matrix*
- weights: Matrix*
- nabla_b: Matrix*
- nabla_w: Matrix*
- delta_nabla_b: Matrix*
- delta_nabla_w: Matrix*
- training_data: MNIST
- test: MNIST
+ Network(sizes: int*)
+ ~Network()
+ feedforward(a: const Matrix&): Matrix
+ SGD(epochs: int, mini_batch_size: int, eta: float): void
+ updateMiniBatch(mini_batch_index: int, eta: float): void
+ backpropagation(x: const Matrix&, y: const Matrix&): void
+ printBiases(): void
+ printWeights(): void
+ evaluate(): void
+ classify(): void
+ argmax(mtx: Matrix&) const: int
+ loadWeightsBiases(): void
+ sigmoid(mtx: Matrix&): Matrix
+ sigmoid_prime(mtx: Matrix&): Matrix
+ cost_derivative(output_activations: Matrix, y: Matrix): Matrix
6
4
2

Fig. 8: UML diagram of C++ program.

computed for the neural network in the training stage. All files used in this implementation are binary files. We will not show here the *Matrix.cpp* file because it is easy to follow along. The only operation that is important to mention is the Hadamard Product, which is implemented as the operator $\wedge$ and performs the elementwise product of two vectors, needed in the backpropagation algorithm.

B. *MNIST class*

This class is very important because it is in charge of reading the MNIST dataset and format it properly so the neural network can work with it. We explain what we consider necessary for the understanding of this class, and show the header file as well as some code fragments from the *MNIST.cpp* file. Below is the header file:

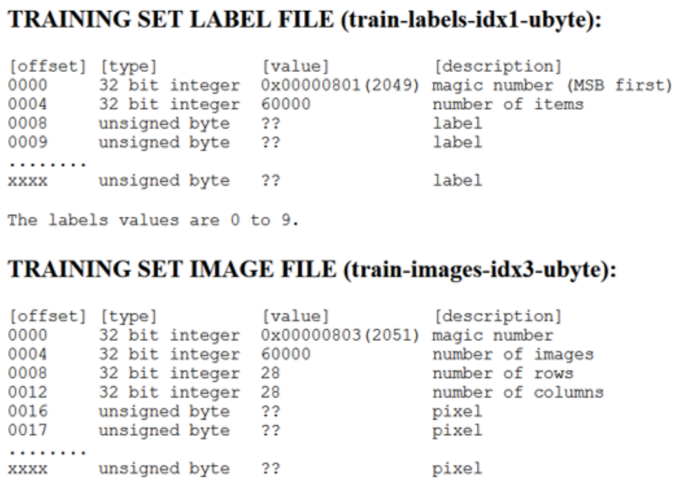
```
#include <fstream>
#include <string>
#include "Matrix.h"
#pragma once

using std::string;
using std::ifstream;

class MNIST {
  string file;  //Flag that indicates file
  int n;       //Number of image rows (28)
  int m;       //Number of image columns (28)
  int magic_number;    //2049 or 2051
  int number_of_items;   //60,000 or 10,000
  int mini_batch_size;   //User's decision
  Matrix* image_dat_set; //Images from MNIST
  Matrix* label_dat_set; //Labels from MNIST
  Matrix* mini_batch_imag; //Mini-batch images
  Matrix* mini_batch_label;//Mini-batch labels
  int reverseInt(int i); //Raw bytes to int

public:
  MNIST(string data);
  MNIST(const MNIST& data);
  ~MNIST();
  void mini_batches(int n);
  void shuffle();
  int get_number_items();
  int get_mini_batch_size();
  Matrix getImage(int index);
  Matrix getLabel(int index);
  Matrix* getMiniBatchImages(int index);
  Matrix* getMiniBatchLabels(int index);
};
```

Listing 2: MNIST.h

We start by explaining how the MNIST dataset is provided. All data can be downloaded as four binary files, specifically named as:

- **train-images-idx3-ubyte.gz:** training set images (9912422 bytes)
- **train-labels-idx1-ubyte.gz:** training set labels (28881 bytes)
- **t10k-images-idx3-ubyte.gz:** test set images (1648877 bytes)
- **t10k-labels-idx1-ubyte.gz:** test set labels (4542 bytes)

which then need to be read by a program that can decode and format the raw data properly. The MNIST class is this program. In Fig. 9 we show how the data is stored in the binary files, and with that information the MNIST class was developed.



TRAINING SET LABEL FILE (train-labels-idx1-ubyte):

```
[offset] [type]          [value]          [description]
0000     32 bit integer  0x00000801(2049) magic number (MSB first)
0004     32 bit integer  60000            number of items
0008     unsigned byte   ??               label
0009     unsigned byte   ??               label
........
xxxx     unsigned byte   ??               label

The labels values are 0 to 9.
```

TRAINING SET IMAGE FILE (train-images-idx3-ubyte):

```
[offset] [type]          [value]          [description]
0000     32 bit integer  0x00000803(2051) magic number
0004     32 bit integer  60000            number of images
0008     32 bit integer  28               number of rows
0012     32 bit integer  28               number of columns
0016     unsigned byte   ??               pixel
0017     unsigned byte   ??               pixel
........
xxxx     unsigned byte   ??               pixel
```

Fig. 9: Binary files format of MNIST dataset.

We can see that the first thing the decoder must read is a 32-bit integer that represents a magic number, which can be 2049 (labels) or 2051 (images). Then comes the number of items or images in the file, which can be $60,000$ (training) or $10,000$ (test). Once we have extracted this information, then what comes next can be either a label or a pixel. If it is a label, then the program must read an *unsigned byte*, but if not, first it needs to get the image size in pixels (28x28) and after that, finally read each pixel in sets of 784. Pixel values go from 0 to 255 and labels from 0 to 9. More information can be found in [4] if needed.

Knowing how the data is stored in the files, then it is easy to understand the code. We will not show here all the *MNIST.cpp* file because it is too long, but it is worth mentioning that it is necessary to convert the raw bytes to integers, and this is done with the *MNIST::reverseInt(int i)* function, shown below. Bitwise operators were used.

```
//Convert raw bytes to int
int MNIST::reverseInt(int i) {
    unsigned char ch1, ch2, ch3, ch4;
    ch1 = i & 255;
    ch2 = (i >> 8) & 255;
    ch3 = (i >> 16) & 255;
    ch4 = (i >> 24) & 255;
    return((int)ch1 << 24) + ((int)ch2 << 16)
    + ((int)ch3 << 8) + ch4;
}
```

Listing 3: MNIST::reverseInt(int i)

When the dataset is stored in RAM, the MNIST class formats the images and labels as follows:

$$Digit = \begin{bmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,28} \\ p_{2,1} & p_{2,2} & \cdots & p_{2,28} \\ \vdots & \vdots & \ddots & \vdots \\ p_{28,1} & p_{28,2} & \cdots & p_{28,28} \end{bmatrix} \rightarrow \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_{784} \end{bmatrix}$$

$$Label = n \rightarrow \begin{bmatrix} l_1 \\ l_2 \\ \vdots \\ l_{10} \end{bmatrix} \qquad n = 0, 1, ..., 9$$

As shown above, the 28x28 pixels images are stored as a 784-column vector. Also, labels are not stored as integers, but as a 10-column vector with all its elements been zero except that one in the position that corresponds with the label. For example, if label is 3, then it will be stored in the program as $l = [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]^T$. This format is compatible with the neural network.

There are two other functions worth mentioning: *MNIST::shuffle()* and *MNIST::mini_batches(int n)*. The first one is used to randomly shuffle all the images and labels from the MNIST dataset. Of course this shuffle maintains all images with their corresponding labels. The other function divides the dataset into individual mini-batches, i.e. smaller vectors containing each one m number of images with their corresponding label. For example, if a mini-batch is of size $m = 100$ and we have $60,000$ images, then we end up having 600 vectors each one containing a 100 images. Same applies for the labels.

C. *Network class*

This is the most important class of all three. This class is the actual neural network and contains all the algorithms and functions needed for the training and testing of the classifier. The header file is shown below:

```
#include "Matrix.h"
#include "MNIST.h"
#pragma once


class Network{
  int num_layers;    //Layers (3)
  int performance;   //Correctly classified
  Matrix* biases;    //All biases
  Matrix* weights;   //All weights
  Matrix* nabla_b;   //Bias gradient
  Matrix* nabla_w;   //weight gradient
  Matrix* delta_nabla_b;   //BP bias delta
  Matrix* delta_nabla_w;   //BP weight delta
  MNIST training_data;   //MNIST training data
  MNIST test;         //MNIST test data

public:
  Network(int* sizes);
  ~Network();

  Matrix feedforward(const Matrix& a);
  void SGD(int epochs, int mini_batch_size,
    float eta);
  void updateMiniBatch(int mini_batch_index,
    float eta);
  void backpropagation(const Matrix& x, const
    Matrix& y);
  void printBiases();
  void printWeights();
  void evaluate();
  void classify();
  int argmax(Matrix& mtx) const;
  void loadWeightsBiases();
  Matrix sigmoid( Matrix& mtx);
  Matrix sigmoid_prime( Matrix& mtx);
  Matrix cost_derivative(Matrix
    output_activations, Matrix y);
};
```

Listing 4: Network.h

We start by defining our neural network. We already mentioned it is a shallow neural network, but now we specify that it has 784 neurons in the input layer, 30 hidden neurons, and 10 neurons in the output layer (one for each digit). This information is provided in the constructor via a vector of type int:

$$int \;\; size\_layers[] = \{784, 30, 10\};$$

```
Network::Network(int* size_layers)
```

With these parameters we know from theory that we need two weight matrices of size 30×784 and 10×30, and two biases vectors of size 30×1 and 10×1:

$$w_1 = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,784} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,784} \\ \vdots & \vdots & \ddots & \vdots \\ w_{30,1} & w_{30,2} & \cdots & w_{30,784} \end{bmatrix} \quad b_1 = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{30} \end{bmatrix}$$

$$w_2 = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,30} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,30} \\ \vdots & \vdots & \ddots & \vdots \\ w_{10,1} & w_{10,2} & \cdots & w_{10,30} \end{bmatrix} \quad b_2 = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{10} \end{bmatrix}$$

All initial values of these matrices are random numbers generated using the *random* library, and have a normal (gaussian) distribution with mean 0.0 and standard deviation of 1.0.

Now let's explain the feedforward function. This function receives an input a (initially x, which is a digit) and returns the corresponding output of the neural network. All this function does is to apply equation (4) for each layer. If the weights and biases are already obtained from the training stage, then this simple function should be able to correctly classify all ten digits. Unfortunately, this is not the case, and what we need to achieve is to obtain these values of weights and biases training the neural network. This is done using the stochastic gradient descent and backpropagation algorithms.

```cpp
//Feedforward
Matrix Network::feedforward(const Matrix& a) {
    Matrix output{a};

    output = weights[0] * output + biases[0];
    output = sigmoid(output);
    output = weights[1] * output + biases[1];
    output = sigmoid(output);

    return output;
}
```

Listing 5: Feedforward function implemented in C++.

The training stage was implemented as three individual functions:
SGD(int epochs, int mini_batch_size, float eta)
updateMiniBatch(int mini_batch_index, float eta)
backpropagation(const Matrix& x, const Matrix& y)

We start with the SGD function, which is shown below:

```cpp
//SGD algorithm
void Network::SGD( int epochs, int
   mini_batch_size, float eta) {
    for (size_t i{ 0 }; i < epochs; i++) {
        training_data.shuffle();
        training_data.mini_batches(
   mini_batch_size);

        for (size_t j{ 0 }; j < training_data.
   get_number_items() / mini_batch_size; j++)
            updateMiniBatch(j, eta);

        cout << "Epoch [" << i + 1 << "]
   complete." << endl;
        evaluate();
    }
}
```

Listing 6: Network::SGD(-) function.

This function implements the first stage of the training algorithms. It receives the number of epochs, the mini-batch size and the learning rate (eta) from the user. Once it has this parameters, the function will compute the stochastic gradient descent for each epoch. We can see that this function calls *training_data.shuffle()* because each mini-batch must be randomly selected as explained in section II. It also calls *training_data.mini_batches(mini_batch_size)*, which generates all the mini-batches needed for the computation. Once it has all set, it calls *updateMiniBatch()* for the next stage. So, basically, this function only sets everything needed for the stochastic gradient descent algorithm and backpropagation.

Then comes the *Network::updateMiniBatch(int mini_batch_index, float eta)* function. This function sets all weights and biases gradients to zero for each epoch. After that, it gets one mini-batch at a time which was previously generated and feeds it to the backpropagation algorithm. Backpropagation computes the errors and finally this function updates all weights and biases. We can see in this function the equations (15) and (16) implemented on lines 24 to 27.

```cpp
1  //Update mini batch
2  void Network::updateMiniBatch(int
    mini_batch_index, float eta) {
3      nabla_b[0].zeros();
4      nabla_b[1].zeros();
5      nabla_w[0].zeros();
6      nabla_w[1].zeros();
7
8      for (size_t i{ 0 }; i < training_data.
    get_mini_batch_size(); i++) {
9          Matrix* images;
10         Matrix* labels;
11         images = training_data.
    getMiniBatchImages(mini_batch_index);
12         labels = training_data.
    getMiniBatchLabels(mini_batch_index);
13         backpropagation(images[i], labels[i]);
14
15         nabla_b[0] = nabla_b[0] +
    delta_nabla_b[0];
16         nabla_w[0] = nabla_w[0] +
    delta_nabla_w[0];
17         nabla_b[1] = nabla_b[1] +
    delta_nabla_b[1];
18         nabla_w[1] = nabla_w[1] +
    delta_nabla_w[1];
19
20         delete[] images;
21         delete[] labels;
22     }
23
24     weights[0] = weights[0] - (nabla_w[0] * (
    eta / training_data.get_mini_batch_size())
    );
25     biases[0]  = biases[0] - (nabla_b[0] * (
    eta / training_data.get_mini_batch_size())
    );
26     weights[1] = weights[1] - (nabla_w[1] * (
    eta / training_data.get_mini_batch_size())
    );
27     biases[1]  = biases[1] - (nabla_b[1] * (
    eta / training_data.get_mini_batch_size())
    );
}
```

Listing 7: Network::updateMiniBatch(-) function.

Finally we have the backpropagation algorithm implemented as *Network::backpropagation(const Matrix& x, const Matrix& y)* . This function is divided into two sections: feedforward and backward pass. The feedforward section computes the output of the neural network and saves all intermediate stages needed later in the backward pass. The backward pass computes the error and propagates it back to the first layer of the neural network.

```
//Backpropagation
void Network::backpropagation(const Matrix& x, const Matrix& y) {

    //feedforward
    Matrix activation{ x };
    Matrix* activations = new Matrix[num_layers];
    activations[0] = x; //list to store all the activations, layer by layer
    Matrix* zs = new Matrix[2]; //list to store all the z vectors, layer by layer
    Matrix z{};

    z = (weights[0] * activation) + biases[0];
    zs[0] = z;
    activation = sigmoid(z);
    activations[1] = activation;

    z = (weights[1] * activation) + biases[1];
    zs[1] = z;
    activation = sigmoid(z);
    activations[2] = activation;

    //Backward pass
    Matrix delta = cost_derivative(activations[2], y) ^ sigmoid_prime(zs[1]);
    delta_nabla_b[1] = delta;
    delta_nabla_w[1] = delta * ~activations[1];

    z = zs[0];
    Matrix sp{ sigmoid_prime(z) };
    delta = (~weights[1] * delta) ^ sp;
    delta_nabla_b[0] = delta;
    delta_nabla_w[0] = delta * ~activations[0];

    delete[] activations;
    delete[] zs;
}
```

Listing 8: Network::backpropagation(-) function.

This class also implements more functions needed for the training, but are easier to understand and the reader can interpret the code without much problem.

D. Main program

This is a very simple C++ program that shows a menu so the user can decide if he wants to train the neural network or just classify the digits.

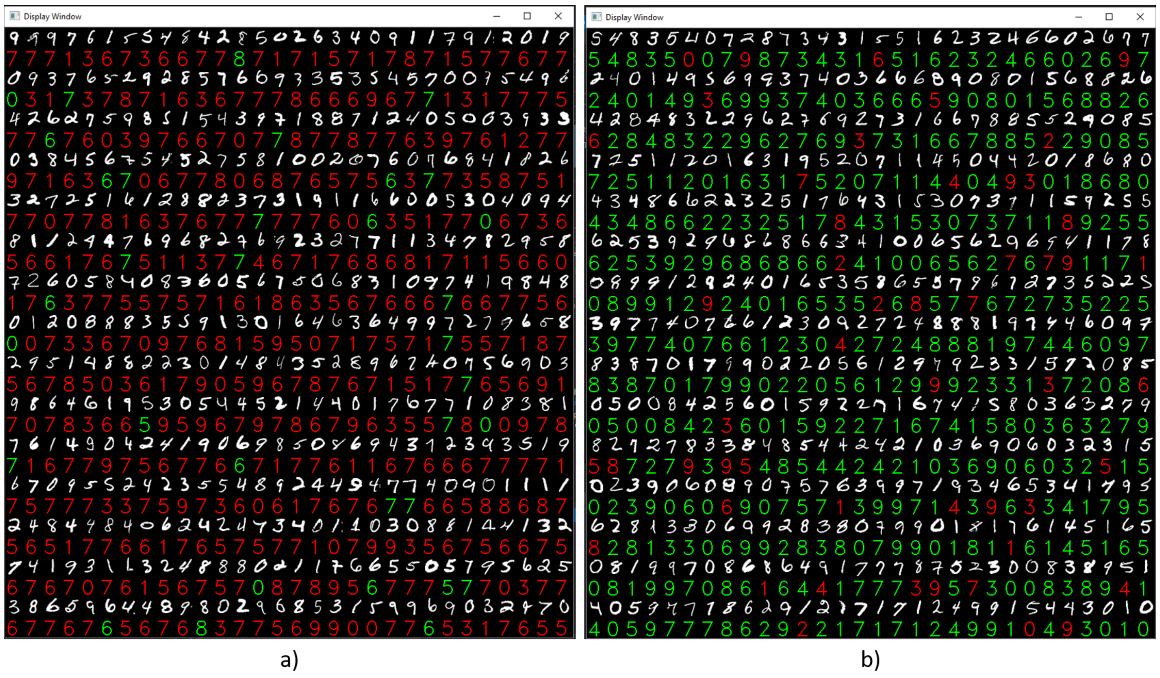
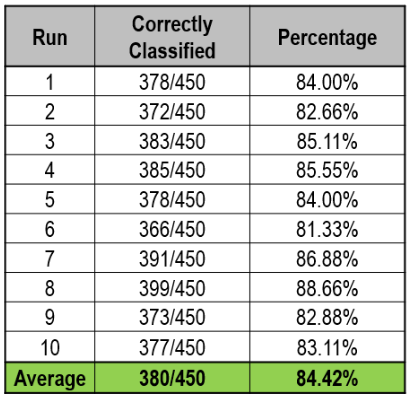


Fig. 10: a) Output of neural network without training. Just 7.55% were randomly correctly classified. b) Output from neural network after training. 88.66% of digits were correctly classified.

V. RESULTS

Once the neural network was trained, ten different classifications of 450 digits were made and the average was calculated. These results are shown in the table from Fig. 11. Approximately 380 out of 450 digits were correctly classified, which represents the 84.42% (see Fig. 10). For purposes of comparison, the neural network was used without previously been trained, and just 7.55% of all digits were correctly classified, but this was all random decisions. See Fig. 10-a.



Run	Correctly Classified	Percentage
1	378/450	84.00%
2	372/450	82.66%
3	383/450	85.11%
4	385/450	85.55%
5	378/450	84.00%
6	366/450	81.33%
7	391/450	86.88%
8	399/450	88.66%
9	373/450	82.88%
10	377/450	83.11%
Average	**380/450**	**84.42%**

Fig. 11: Results obtained from the classification.

VI. CONCLUSIONS

The classification of handwritten digits was satisfactorily achieved using a shallow neural network implemented in C++ with 15.58% of error. We know that there are better classifications out there and we still can improve these results using the same neural network, but the time of develop was longer than we expected and it was not possible to enhance this training. Although we know that having a classification with 15.58% error is not an optimal classifier, still it is considerably good for our purposes of learning.

Also, the fact that this classifier was implemented on C++ gave us a lot of insight on machine learning because we needed to implement every single function of the neural network, which forced us to understand the algorithms and not only use them out of libraries already having everything done.

REFERENCES

[1] Sergios Theodoridis, *Pattern Recognition*. Fourth Edition, Elsevier, 2009.

[2] Sergios Theodoridis, *Introduction to Pattern Recognition: A MATLAB Approach*. Academic Press, 2010.

[3] Nielsen Michael, *Neural Networks and Deep Learning*. Free online book.

[4] Yann LeCun, Corinna Cortes, Christopher J.C. Burges, *The MNIST database of handwritten digits*. Free online dataset.