Contents

1 Basic Test Results

```
Running...

Opening tar file
MlpNetwork.h
MlpNetwork.cpp
Matrix.h
Matrix.cpp
Makefile
Dense.h
Dense.cpp
Activation.h
Activation.cpp
OK
Tar extracted O.K.

Checking files...
OK
Making sure files are not empty...
OK
Compilation check...
#1
Compiling...
OK
#2
Compiling...
OK
#3
Compiling...
OK
#4
Compiling...
OK
#5
Compiling...
OK
#6
Compiling...
OK
#7
Compiling...
OK
Compilation seems OK! Check if you got warnings!


=====================
 Public test cases
=====================

=====================
Running test...
OK
Running test...
OK
Test im0 Succeeded.
=====================



========================
= Checking coding style =
========================
 ** Total Violated Rules      : 0
 ** Total Errors Occurs       : 0
 ** Total Violated Files Count: 0
```

2 Activation.h

```
//Activation.h
#ifndef ACTIVATION_H
#define ACTIVATION_H

#include "Matrix.h"

#define FLOAT_NORM 1.f
#define MIN_RELU_VAL 0
/**
 * @enum ActivationType
 * @brief Indicator of activation function.
 */
enum ActivationType
{
    Relu,
    Softmax
};

/**
 * Activation class
 */
class Activation
{
public:
    /**
     * default constructor
     */
    Activation() = default;

    /**
     * constructor for the project
     * @param actType the activationtype
     */
    explicit Activation(ActivationType actType);

    /**
     * getter for teh activationtype
     * @return  the activation type
     */
    ActivationType getActivationType();

    /**
     * overload the () function, activate the activation type function on the input matrix
     * @param input the input matrix
     * @return matrix after the () operator
     */
    Matrix operator()(const Matrix &input);

    /**
     * overload the () function, activate the activation type function on the input matrix
     * @param input the input matrix
     * @return matrix after the () operator
     */
    Matrix operator()(const Matrix &input) const;

    /**
     * default destructor
     */
    ~Activation() = default;

private:
    /**
     * data member the activation type
     */
    ActivationType activationType;

    /**
     * relu activation function
     * @param matrix  the input matrix
     * @return matrix after the activation function
     */
    static Matrix _reluActivation(const Matrix &matrix);

    /**
     * softmax activation function
     * @param matrix  the input matrix
     * @return matrix after the activation function
     */
    static Matrix _softMaxActivation(const Matrix &matrix);
};

#endif //ACTIVATION_H
```

3 Activation.cpp

```cpp
#include <cmath>
#include "Activation.h"

/**
 * constructor for the project
 * @param actType the activationtype
 */
Activation::Activation(ActivationType actType)
{
    this->activationType = actType;
}

/**
 * getter for teh activationtype
 * @return  the activation type
 */
ActivationType Activation::getActivationType()
{
    return this->activationType;
}

/**
 * overload the () function, activate the activation type function on the input matrix
 * @param input the input matrix
 * @return matrix after the () operator
 */
Matrix Activation::operator()(const Matrix &input)
{
    if (this->getActivationType() == Relu)
    {
        return _reluActivation(input);
    }
    return _softMaxActivation(input);
}

/**
 * overload the () function, activate the activation type function on the input matrix
 * @param input the input matrix
 * @return matrix after the () operator
 */
Matrix Activation::operator()(const Matrix &input) const
{
    if (this->activationType == Relu)
    {
        return _reluActivation(input);
    }
    return _softMaxActivation(input);
}

/**
 * relu activation function
 * @param matrix  the input matrix
 * @return matrix after the activation function
 */
Matrix Activation::_reluActivation(const Matrix &matrix)
{
    int size = matrix.getRows() * matrix.getCols();
    Matrix out = Matrix(matrix);
    for (int i = 0; i < size; ++i)
    {
        if (out[i] < MIN_RELU_VAL)
        {
            out[i] = MIN_RELU_VAL;
        }
    }
    return out;
}

/**
 * softmax activation function
 * @param matrix  the input matrix
 * @return matrix after the activation function
 */
Matrix Activation::_softMaxActivation(const Matrix &matrix)
{
    int size = matrix.getRows() * matrix.getCols();
    Matrix out = Matrix(matrix);
    float sum = 0;
    for (int i = 0; i < size; ++i)
    {
        out[i] = std::exp(out[i]);
        sum += out[i];
    }
    float scalar = (FLOAT_NORM / sum);
    return scalar * out;
}
```

4 Dense.h

```
//
// Created by brahan on 19/12/2019.
//

#ifndef CPP_EX1_DENSE_H
#define CPP_EX1_DENSE_H

#include "Matrix.h"
#include "Activation.h"

/**
 * Dense class
 */
class Dense
{
public:
    /**
     * def constructor
     */
    Dense() = default;

    /**
     * Dense constructor
     * @param w weight matrix
     * @param bias bias matrix
     * @param actType activation type
     */
    Dense(Matrix &w, Matrix &bias, ActivationType actType);

    /**
     * def destructor
     */
    ~Dense() = default;

    /**
     * getter for the weights matrix
     * @return the weight matrix
     */
    Matrix &getWeights();

    /**
     * getter for the weights matrix
     * @return the weight matrix
     */
    Matrix getWeights() const;

    /**
     * getter for teh bias matrix
     * @return the bias matrix
     */
    Matrix &getBias();

    /**
     * getter for teh bias matrix
     * @return the bias matrix
     */
    Matrix getBias() const;


    /**
     * getter for the activation data member
     * @return the activation
     */
    Activation &getActivation();

    /**
     * getter for the activation data member
     * @return the activation
     */
    Activation getActivation() const;

    /**
     * overload the () operator
     * @param input , activate the layer on teh input matrix
     * @return matrix after the layer process
     */
    Matrix operator()(Matrix &input);

    /**
     * overload the () operator
     * @param input , activate the layer on teh input matrix
     * @return matrix after the layer process
     */
    Matrix operator()(Matrix &input) const;


private:
    Matrix _weights, _bias;
    Activation _activation;
};

#endif //CPP_EX1_DENSE_H

```

5 Dense.cpp

```cpp

#include "Dense.h"

/**
 * Dense constructor
 * @param w weight matrix
 * @param bias bias matrix
 * @param actType activation type
 */
Dense::Dense(Matrix &w, Matrix &bias, ActivationType actType) :
        _weights(w), _bias(bias), _activation(Activation(actType))
{
}

/**
 * getter for the weights matrix
 * @return the weight matrix
 */
Matrix &Dense::getWeights()
{
    return this->_weights;
}

/**
 * getter for teh bias matrix
 * @return the bias matrix
 */
Matrix &Dense::getBias()
{
    return this->_bias;
}

/**
 * getter for the activation data member
 * @return the activation
 */
Activation &Dense::getActivation()
{
    return this->_activation;
}

/**
 * getter for the weights matrix
 * @return the weight matrix
 */
Matrix Dense::getWeights() const
{
    return this->_weights;
}

/**
 * getter for teh bias matrix
 * @return the bias matrix
 */
Matrix Dense::getBias() const
{
    return this->_bias;
}

```

```
/**
 * getter for the activation data member
 * @return the activation
 */
Activation Dense::getActivation() const
{
    return this->_activation;
}

/**
 * overload the () operator
 * @param input , activate the layer on teh input matrix
 * @return matrix after the layer process
 */
Matrix Dense::operator()(Matrix &input)
{
    Matrix out = (this->_weights * input) + this->_bias;
    out = this->_activation(out);
    return out;
}

/**
 * overload the () operator
 * @param input , activate the layer on teh input matrix
 * @return matrix after the layer process
 */
Matrix Dense::operator()(Matrix &input) const
{
    Matrix out = (this->_weights * input) + this->_bias;
    out = this->_activation(out);
    return out;
}
```

6 Makefile

```
CC=g++
CXXFLAGS= -Wall -Wvla -Wextra -Werror -g -std=c++17
LDFLAGS= -lm
HEADERS= Matrix.h Activation.h Dense.h MlpNetwork.h Digit.h
OBJS= Matrix.o Activation.o Dense.o MlpNetwork.o main.o

%.o : %.c


mlpnetwork: $(OBJS)
    $(CC) $(LDFLAGS) -o $@ $^

$(OBJS) : $(HEADERS)

.PHONY: clean
clean:
    rm -rf *.o
    rm -rf mlpnetwork
```

7 Matrix.h

```
// Matrix.h
#include <iostream>

#ifndef MATRIX_H
#define MATRIX_H
#define EMPTY_MATRIX 0
#define MIN_MATRIX_DIM 1
#define ERROR "ERROR: invalid input"
#define FAILURE 1
#define MIN_PROB 0.1f
#define FIRST_VAL 0
/**
 * @struct MatrixDims
 * @brief Matrix dimensions container
 */
typedef struct MatrixDims
{
    int rows, cols;
} MatrixDims;

/**
 * Matrix class
 */
class Matrix
{
public:
    /**
     * empty const, const 1x1 matrix, init the single ele to 0
     */
    Matrix(); // default
    /**
     * dest for Matrix object
     */
    ~Matrix();

    /**
     * const for Matrix
     * @param rows the number of rows in the matrix
     * @param cols the number of cols in the matrix
     */
    Matrix(int rows, int cols);

    /**
     * const matrix from anther matrix m
     * @param m the other matrix
     */
    Matrix(const Matrix &m); // copy constructor
    //METHODS:
    /**
     * getter for the rows
     * @return number of rows
     */
    int getRows();

    /**
     * getter for the rows
     * @return number of rows
     */
    int getRows() const;

    /**
     * gettter for the cols
     * @return number of cols
     */
    int getCols();

    /**
     * gettter for the cols
     * @return number of cols
     */
    int getCols() const;

    /**
     * Transforms a matrix into a coloumn vector Supports function calling concatenation i.e.(1)
     * Matrix m(5,4);... m.vectorize(), then m.vectorize() + b should be a valid expression
     */
    Matrix vectorize();

    /**
     * Prints matrix elements, no return value. prints space after each element (incl. last element in the row)
     * prints newline after each row (incl. last row)
     */
    void plainPrint() const;

    // OPERATORS:
    /**
 *  overload the * operator to multiply matrix and scalar
 * @param scalar the scalar we want to multiply the matrix by
 * @param matrix the matrix
 * @return scalar * matrix
 */
    friend Matrix operator*(const float &scalar, const Matrix &matrix);

    /**
 *  overload the * operator to multiply matrix and scalar
 * @param scalar the scalar we want to multiply the matrix by
 * @return scalar * matrix
 */
    Matrix operator*(const float &scalar) const;

    /**
     * overload the * operator, to multiply with other matrix
     * @param other the other matrix
     * @return the solution of 2 matrix multiplication
     */
    Matrix operator*(const Matrix &other) const;


    /**
     * overload the () operator, to get the i,j cell inside the matrix
     * @param i the row index
     * @param j the column index
     * @return the i,j element inside the matrix
     */
    float &operator()(const int &i, const int &j) const;

    /**
     * overload the () operator, to get the i,j cell inside the matrix
     * @param i the row index
     * @param j the column index
     * @return the i,j element inside the matrix
     */
    float &operator()(const int &i, const int &j);

    /**
     * overload the + operator to add our matrix with our matrix
     * @param other the other matrix we want to add to our
     * @return the addition matrix
     */
    Matrix operator+(const Matrix &other) const;

    /**
     * overload the + operator to add our matrix with our matrix
     * @param other the other matrix we want to add to our
     * @return our matrix += the other matrix values
     */
    Matrix &operator+=(const Matrix &other);

    /**
     * place the other matrix values inside ours
     * @param other the other matrix
     * @return our matrix with the other values
     */
    Matrix &operator=(const Matrix &other);

    /**
     * overload the [] operator to return the i object in the matrix
     * @param idx the matrix idx
     * @return the ith object in teh matrix
     */
    float &operator[](int idx);

    /**
     * overload the [] operator to return the i object in the matrix
     * @param idx the matrix idx
     * @return the ith object in teh matrix
     */
    float operator[](int idx) const;

/**
 * overlaod the << operator, output the matrix values
 * @param output output stream
 * @param matrix the matrix we want output its values
 * @return the output stream
 */
    friend std::ostream &operator<<(std::ostream &output, const Matrix &matrix);

/**
* overlaod the >> operator, input the matrix values
* @param input the input stream
* @param matrix the matrix we want to load values to
* @return the input stream
*/
    friend std::istream &operator>>(std::istream &input, const Matrix &matrix);

private:
    MatrixDims _matrixDims;
    float *_table{};
};

#endif //MATRIX_H
```

8 Matrix.cpp

```
#include <iostream>
#include <cstring>
#include "Matrix.h"

/**
 * const for Matrix
 * @param rows the number of rows in the matrix
 * @param cols the number of cols in the matrix
 */
Matrix::Matrix(int rows, int cols) :
        _matrixDims{rows = rows, cols = cols}
{
    if (rows < MIN_MATRIX_DIM || cols < MIN_MATRIX_DIM)
    {
        std::cerr << ERROR << std::endl;
        exit(FAILURE);
    }
    else
    {
        this->_table = new float[rows * cols]();
    }
}

/**
 * empty const, const 1x1 matrix, init the single ele to 0
 */
Matrix::Matrix() :
        Matrix(MIN_MATRIX_DIM, MIN_MATRIX_DIM)
{
}

/**
 * const matrix from anther matrix m
 * @param m the other matrix
 */
Matrix::Matrix(const Matrix &other) :
        _matrixDims{other._matrixDims.rows, other._matrixDims.cols}
{
    delete[](this->_table); // delete the current info
    this->_table = new float[other._matrixDims.rows * other._matrixDims.cols];
    for (int i = 0; i < _matrixDims.rows * _matrixDims.cols; ++i)
    {
        this->_table[i] = other._table[i];
    }
}

/**
 * dest for Matrix object
 */
Matrix::~Matrix()
{
    _matrixDims.cols = EMPTY_MATRIX;
    _matrixDims.rows = EMPTY_MATRIX;
    delete[](this->_table);
}

/**
 * getter for the rows
 * @return number of rows
 */
int Matrix::getRows()
{
    return this->_matrixDims.rows;
}

/**
 * gettter for the cols
 * @return number of cols
 */
int Matrix::getCols()
{
    return this->_matrixDims.cols;
}

/**
 * getter for the rows
 * @return number of rows
 */
int Matrix::getRows() const
{
    return this->_matrixDims.rows;
}

/**
 * gettter for the cols
 * @return number of cols
 */
int Matrix::getCols() const
{
    return this->_matrixDims.cols;
}

/**
 * Transforms a matrix into a coloumn vector Supports function calling concatenation i.e.(1)
 * Matrix m(5,4);... m.vectorize(), then m.vectorize() + b should be a valid expression
 */
Matrix Matrix::vectorize()
{
    int size = this->_matrixDims.rows * this->_matrixDims.cols;
    this->_matrixDims.rows = size;
    this->_matrixDims.cols = MIN_MATRIX_DIM;
    return *this;
}

/**
 * Prints matrix elements, no return value. prints space after each element (incl. last element in the row)
 * prints newline after each row (incl. last row)
 */
void Matrix::plainPrint() const
{
    for (int i = 0; i < this->_matrixDims.rows; ++i)
    {
        for (int j = 0; j < this->_matrixDims.cols; ++j)
        {
            std::cout << this->_table[i * this->_matrixDims.cols + j];
            std::cout << " ";
        }
        std::cout << std::endl;
    }
}

/**
*  overload the * operator to multiply matrix and scalar
* @param scalar the scalar we want to multiply the matrix by
* @param matrix the matrix
* @return scalar * matrix
*/
```

```
Matrix operator*(const float &scalar, const Matrix &matrix)
{
    Matrix output = Matrix(matrix);
    for (int i = 0; i < matrix.getRows() * matrix.getCols(); ++i)
    {
        output[i] *= scalar;
    }
    return output;
}

/**
*  overload the * operator to multiply matrix and scalar
* @param scalar the scalar we want to multiply the matrix by
* @return scalar * matrix
*/
Matrix Matrix::operator*(const float &scalar) const
{
    Matrix output = Matrix(*this);
    for (int i = 0; i < this->getRows() * this->getCols(); ++i)
    {
        output[i] *= scalar;
    }
    return output;
}

/**
 * overload the * operator, to multiply with other matrix
 * @param other the other matrix
 * @return the solution of 2 matrix multiplication
 */
Matrix Matrix::operator*(const Matrix &other) const
{
    if ((other._matrixDims.rows != this->_matrixDims.cols))
    {
        std::cerr << ERROR << std::endl;
        exit(FAILURE);
    }
    else
    {
        auto ans = Matrix(_matrixDims.rows, other._matrixDims.cols); // all zeros
        for (int i = 0; i < _matrixDims.rows; ++i)
        {
            for (int j = 0; j < other._matrixDims.cols; ++j)
            {
                for (int k = 0; k < _matrixDims.cols; ++k)
                {
                    float temp = _table[i * _matrixDims.cols + k];
                    ans._table[i * other._matrixDims.cols + j] += temp * other(k, j);
                }
            }
        }
        return ans;
    }
}

/**
 * overload the + operator to add our matrix with our matrix
 * @param other the other matrix we want to add to our
 * @return the addition matrix
 */
Matrix Matrix::operator+(const Matrix &other) const
{
    if ((other._matrixDims.rows != this->_matrixDims.rows) || ((other._matrixDims.cols != this->_matrixDims.cols)))
    {
        std::cerr << ERROR << std::endl;
        exit(FAILURE);
    }
    else
    {
        auto ans = Matrix(*this);
        for (int i = 0; i < _matrixDims.rows; ++i)
        {
            for (int j = 0; j < _matrixDims.cols; ++j)
            {
                ans._table[i * _matrixDims.cols + j] += other(i, j);
            }
        }
        return ans;
    }
}

/**
 * overload the + operator to add our matrix with our matrix
 * @param other the other matrix we want to add to our
 * @return our matrix += the other matrix values
 */
Matrix &Matrix::operator+=(const Matrix &other)
{
    if ((other._matrixDims.rows != this->_matrixDims.rows) || ((other._matrixDims.cols != this->_matrixDims.cols)))
    {
        std::cerr << ERROR << std::endl;
        exit(FAILURE);
    }
    else
    {
        for (int i = 0; i < this->_matrixDims.rows; ++i)
        {
            for (int j = 0; j < this->_matrixDims.cols; ++j)
            {
                this->_table[i * this->_matrixDims.cols + j] += other._table[i * other._matrixDims.cols + j];
            }
        }
        return *this;
    }
}

/**
 * place the other matrix values inside ours
 * @param other the other matrix
 * @return our matrix with the other values
 */
Matrix &Matrix::operator=(const Matrix &other)
{
    if (this == &other)
    {
        return *this; // so we wont delete our own table
    }
    delete[](this->_table); // delete the current info
    this->_matrixDims = other._matrixDims; // update the matrix dimensions
    this->_table = new float[other._matrixDims.rows * other._matrixDims.cols];
    std::memcpy(this->_table, other._table, _matrixDims.rows * _matrixDims.cols * sizeof(float));
    return *this;
}

/**
 * overload the () operator, to get the i,j cell inside the matrix
 * @param i the row index
 * @param j the column index
 * @return the i,j element inside the matrix
 */
float &Matrix::operator()(const int &i, const int &j) const
{
    if ((this->_matrixDims.rows <= i) || ((this->_matrixDims.cols <= j)) || i < FIRST_VAL || j < FIRST_VAL)
    {
        std::cerr << ERROR << std::endl;
        exit(FAILURE);
    }
    else
    {
        return this->_table[i * this->_matrixDims.cols + j];

    }
}

/**
 * overload the () operator, to get the i,j cell inside the matrix
 * @param i the row index
 * @param j the column index
 * @return the i,j element inside the matrix
 */
float &Matrix::operator()(const int &i, const int &j)
{
    if ((this->_matrixDims.rows <= i) || ((this->_matrixDims.cols <= j)) || i < FIRST_VAL || j < FIRST_VAL)
    {
        std::cerr << ERROR << std::endl;
        exit(FAILURE);
    }
    else
    {
        return this->_table[i * this->_matrixDims.cols + j];
    }
}

/**
 * overload the [] operator to return the i object in the matrix
 * @param idx the matrix idx
 * @return the ith object in teh matrix
 */
float &Matrix::operator[](int idx)
{
    if (this->_matrixDims.rows * this->_matrixDims.cols <= idx || idx < FIRST_VAL)
    {
        std::cerr << ERROR << std::endl;
        exit(FAILURE);
    }
    else
    {
        return this->_table[idx];
    }
}

/**
 * overload the [] operator to return the i object in the matrix
 * @param idx the matrix idx
 * @return the ith object in teh matrix
 */
float Matrix::operator[](int idx) const
{
    if (this->_matrixDims.rows * this->_matrixDims.cols <= idx || idx < FIRST_VAL)
    {
        std::cerr << ERROR << std::endl;
        exit(FAILURE);
    }
    else
    {
        return this->_table[idx];
    }
}

/**
 * overlaod the << operator, output the matrix values
 * @param output output stream
 * @param matrix the matrix we want output its values
 * @return the output stream
 */
std::ostream &operator<<(std::ostream &output, const Matrix &matrix)
{
    for (int i = 0; i < matrix._matrixDims.rows; ++i)
    {
        for (int j = 0; j < matrix._matrixDims.cols; ++j)
        {
            if (matrix(i, j) <= MIN_PROB)
            {
                output << "  ";
            }
            else
            {
                output << "**";
            }
        }
        output << std::endl;
    }
    return output;
}

/**
* overlaod the >> operator, input the matrix values
* @param input the input stream
* @param matrix the matrix we want to load values to
* @return the input stream
*/
std::istream &operator>>(std::istream &input, const Matrix &matrix)
{
    int idx = 0;
    while (input.good())
    {
        input.read(reinterpret_cast<char *>(&matrix._table[idx]), sizeof(float));
        idx++;
    }
    return input;
}


```

9 MlpNetwork.h

```cpp
//MlpNetwork.h

#ifndef MLPNETWORK_H
#define MLPNETWORK_H

#include "Matrix.h"
#include "Digit.h"
#include "Dense.h"

#define MLP_SIZE 4
#define FIRST_LAYER 0
#define SEC_LAYER 1
#define THIRD_LAYER 2
#define FOURTH_LAYER 3
#define NO_PROB 0.0
#define NO_IDX 0

const MatrixDims imgDims = {28, 28};
const MatrixDims weightsDims[] = {{128, 784},
                                  {64,  128},
                                  {20,  64},
                                  {10,  20}};
const MatrixDims biasDims[] = {{128, 1},
                               {64,  1},
                               {20,  1},
                               {10,  1}};

/**
 * MlpNetwork
 */
class MlpNetwork
{
public:
    /**
     * def constructor
     */
    MlpNetwork() = default; // default const
    /**
     * def destructor
     */
    ~MlpNetwork() = default; // default dest
    /**
     * mlp constructor
     * @param weights the weights
     * @param biases the biases
     */
    MlpNetwork(Matrix *weights, Matrix *biases);

    /**
     * overload () operator, activate the mlp process on the input matrix
     * @param matrix the input matrix
     * @return matrix after the mlp process
     */
    Digit operator()(const Matrix &matrix);


private:
    Dense _d1, _d2, _d3, _d4;
};
```

```

#endif // MLPNETWORK_H
```

10 MlpNetwork.cpp

```cpp
#include "MlpNetwork.h"

/**
* mlp constructor
* @param weights the weights
* @param biases the biases
*/
MlpNetwork::MlpNetwork(Matrix *weights, Matrix *biases)
{
    this->_d1 = Dense(weights[FIRST_LAYER], biases[FIRST_LAYER], Relu);
    this->_d2 = Dense(weights[SEC_LAYER], biases[SEC_LAYER], Relu);
    this->_d3 = Dense(weights[THIRD_LAYER], biases[THIRD_LAYER], Relu);
    this->_d4 = Dense(weights[FOURTH_LAYER], biases[FOURTH_LAYER], Softmax);
}

/**
* overload () operator, activate the mlp process on the input matrix
* @param matrix the input matrix
* @return matrix after the mlp process
*/
Digit MlpNetwork::operator()(const Matrix &matrix)
{
    Matrix m = Matrix(matrix);
    m = _d1(m);
    m = _d2(m);
    m = _d3(m);
    m = _d4(m);
    float prob = NO_PROB;
    int idx = NO_IDX;
    for (int i = 0; i < m.getRows(); ++i)
    {
        if (m[i] > prob)
        {
            prob = m[i];
            idx = i;
        }
    }
    Digit digit;
    digit.probability = prob;
    digit.value = idx;
    return digit;
}
```