Contents

1 Basic Test Results

```

Running presubmission script...


Opening tar file
OK
Tar extracted O.K.
For your convenience, the MD5 checksum for your submission is 1318e5a96905e08d61d3876e7e0d726b
Checking files...
OK
Making sure files are not empty...
OK
Checking CodingStyle...
Checking file Matrix.cpp...
Problem: in Matrix.cpp, line 116 is longer than 79 characters and will be truncated.
Checking file Activation.cpp...
Checking file Dense.cpp...
Checking file MlpNetwork.cpp...
Problem: in MlpNetwork.cpp, line 6 is longer than 79 characters and will be truncated.
Checking file Matrix.h...
Problem: in Matrix.h, line [42:16], warning: parameter 'm' is const-qualified in the function declaration; const-qualificati

    Matrix dot(const Matrix m) const;

               ^~~~~~


Problem: in Matrix.h, line 35 is longer than 79 characters and will be truncated.
Checking file Activation.h...
Checking file Dense.h...
Checking file MlpNetwork.h...
Compilation check...
Compiling...

Compilation looks good!


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

Running test...
Checking functions exist and basic functionality:
Checking Matrix class:
    constructors
    operators () and []
    get_rows, get_cols
    vectorize
    dot
    norm
    transpose
    operator =
    Matrix multiplication
    Scalar multiplication on left
    Scalar multiplication on right
    Matrix addition
    Matrix addition accumulation
    plain_print: Should print 0 to 8
0 1 2 3 4 5 6 7 8
```

```
Passed: All Matrix functions exist

Checking Activation class:
    existance
    operator ()
Passed: All Activation functions exist

Checking Dense class:
    constructor
    get_weights, get_bias, get_activation
    operator ()
Passed: All Dense functions exist

Checking MlpNetwork, displaying output:
Image processed:





                                ****************
                ********************************
              ********************************
                ********************
                ************    ****
                      ******
                      ******
                        ******
                        **********
                          **********
                            ************
                                ********
                                  ********
                            ************
                        ****************
                      ****************
                  ****************
              ****************
        ******************
        **************




Mlp result: 5 at probability: 0.99821
All presubmit tests finished!


OK
*****************************************
*                  ***                  *
*          Passed all tests!!           *
*              Good Job!                *
*                  ***                  *
*****************************************
```

2 ex4/README.md

```
# ex4-shiraneyal
```

3 ex4/Activation.h

```cpp
#include "Matrix.h"

#ifndef ACTIVATION_H
#define ACTIVATION_H

// Insert Activation class here...
namespace activation {
    // relu function (given matrix, change every negative element to zero)
    Matrix relu(const Matrix& m);
    // softmax function (returns exponent vector)
    Matrix softmax(const Matrix& m);
};
#endif //ACTIVATION_H
```

4 ex4/Activation.cpp

```cpp
#include "Activation.h"

Matrix activation::relu(const Matrix& m) {
    Matrix m2(m.get_rows(), m.get_cols());
    for (int i = 0; i < m.get_rows(); i++){
        for (int j = 0; j < m.get_cols(); j++) {
            if (m(i, j) < 0) {
                m2(i, j) = 0;
            } else {
                m2(i, j) = m(i, j);
            }
        }
    }
    return m2;
}

Matrix activation::softmax(const Matrix& m) {
    Matrix m2(m.get_rows(), m.get_cols());
    float exp_sum = 0;
    for (int i = 0; i < m.get_rows(); i++){
        for (int j = 0; j < m.get_cols(); j++) {
            exp_sum += (float) std::exp(m(i, j));
            m2(i, j) = (float) std::exp(m(i, j));
        }
    }
    for (int i = 0; i < m.get_rows(); i++){
        for (int j = 0; j < m.get_cols(); j++) {
            m2(i, j) = m2(i, j) / exp_sum;
        }
    }
    return m2;
}
```

5 ex4/Dense.h

```
#ifndef DENSE_H
#define DENSE_H

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

typedef Matrix (*activation_func_pointer)(const Matrix& m);
// Insert Dense class here...
class Dense {
public:
    // constructor
    Dense(const Matrix w, const Matrix b, activation_func_pointer func):
        weights(w), bias(b), activation(func) {};
    // get weights matrix
    Matrix get_weights() const {return weights;};
    // get bias matrix
    Matrix get_bias() const {return bias;};
    // get activation func
    activation_func_pointer get_activation() const {return activation;};
    // parenthesis operator
    Matrix operator() (const Matrix& m) const;
private:
    Matrix weights;
    Matrix bias;
    activation_func_pointer activation;
};

#endif //DENSE_H
```

6 ex4/Dense.cpp

```cpp
#include "Dense.h"

Matrix Dense::operator() (const Matrix& m) const {
    Matrix m1 = activation((weights * m) + bias);
    return m1;
}
```

7 ex4/Matrix.h

```cpp
// Matrix.h
#ifndef MATRIX_H
#define MATRIX_H

#include <iostream>
#include <fstream>
#include <cmath>
/**
 * @struct matrix_dims
 * @brief Matrix dimensions container. Used in MlpNetwork.h and main.cpp
 */
typedef struct matrix_dims
{
    int rows, cols;
} matrix_dims;

// Insert Matrix class here...
class Matrix {
public:
    // constructor that gets matrix dimensions and initializes zeros matrix of that size
    Matrix(int rows, int cols);
    // constructor that gets nothing and initializes a 1x1 matrix
    Matrix(): Matrix(1, 1) {};
    // constructor that gets matrix and initializes new identical one
    Matrix(const Matrix& m);
    // destructor
    ~Matrix() {
        delete[] matrix_arr;
    };

    // get amount of rows
    int get_rows() const {return dim.rows;};
    // get amount of cols
    int get_cols() const {return dim.cols;};
    // transpose a matrix (returns reference to self so that b.transpose().transpose() is legal)
    Matrix& transpose();
    // vectorize a matrix
    Matrix& vectorize();
    // print matrix elements
    void plain_print() const;
    // return multiplication matrix of current matrix with given one
    Matrix dot(const Matrix m) const;
    // return frobenius norm of matrix
    float norm() const;

    // Matrix addition
    Matrix operator+ (const Matrix& m) const;
    // Matrix assignment
    Matrix& operator= (const Matrix& m);
    // Matrix multiplication
    Matrix operator* (const Matrix& m) const;
    // Scalar multiplication from right
    Matrix operator* (float c) const;
    // Scalar multiplication from left
    friend Matrix operator* (float c, Matrix& m);
    // Matrix addition accumulation
    Matrix& operator+= (Matrix& m);
    // () indexing
    float &operator() (int i, int j);
    float operator() (int i, int j) const;
    // [] indexing
    float &operator[] (int i);
    float operator[] (int i) const;
    // pretty export of matrix
    friend std::ostream &operator<< (std::ostream& ostream, Matrix& m);
    // read file content into matrix
    friend std::ifstream &operator>> (std::ifstream& ifs, Matrix& m);
private:
    matrix_dims dim;
    float* matrix_arr;
};
#endif //MATRIX_H
```

8 ex4/Matrix.cpp

```
#include "Matrix.h"

#define LENGTH_ERROR_MESSAGE "error relating to invalid dimensions has occurred"
#define OUT_OF_RANGE_ERROR_MESSAGE "error relating to inaccessibility has occurred"
#define RUNTIME_ERROR_MESSAGE "error relating to invalid input has occurred"
#define DOUBLE_ASTERISK "**"
#define DOUBLE_SPACE "  "
#define ASTERISK_THRESHOLD 0.1

Matrix::Matrix(int rows, int cols):
dim({rows, cols}), matrix_arr(nullptr)
{
    if (rows <= 0 || cols <= 0) {
        throw std::length_error (LENGTH_ERROR_MESSAGE);
    }
    matrix_arr = new float[rows * cols]();
}

Matrix::Matrix(const Matrix& m):
dim({m.dim.rows, m.dim.cols}), matrix_arr(new float[m.dim.rows * m.dim.cols])
{
    for (int i = 0; i < dim.rows * dim.cols; i++) {
        matrix_arr[i] = m[i];
    }
}

Matrix& Matrix::transpose() {
    Matrix temp(*this);
    // swap rows and cols
    dim.rows = dim.rows + dim.cols;
    dim.cols = dim.rows - dim.cols;
    dim.rows = dim.rows - dim.cols;
    for (int i = 0; i < dim.rows; i++){
        for (int j = 0; j < dim.cols; j++) {
            (*this)(i, j) = temp(j, i);
        }
    }
    return *this;
}

Matrix& Matrix::vectorize() {
    // redefine matrix's rows and cols to be 1 column
    dim.rows = dim.rows * dim.cols;
    dim.cols = 1;
    return *this;
}

void Matrix::plain_print() const {
    for (int i = 0; i < dim.rows; i++){
        for (int j = 0; j < dim.cols; j++) {
            std::cout << matrix_arr[i * dim.cols + j] << " ";
        }
        std::cout << "\n";
    }
}

Matrix Matrix::dot(const Matrix m) const {
    // verify dimensions validity
    if (m.dim.rows != dim.rows || m.dim.cols != dim.cols) {
        throw std::length_error (LENGTH_ERROR_MESSAGE);
    }
    // create dot product matrix
    Matrix dot_matrix(dim.rows, dim.cols);
    for (int i = 0; i < dim.rows; i++){
        for (int j = 0; j < dim.cols; j++) {
            dot_matrix(i, j) = m(i, j) * matrix_arr[i * dim.cols + j];
        }
    }
    return dot_matrix;
}

float Matrix::norm() const {
    float sum = 0;
    for (int i = 0; i < dim.rows; i++){
        for (int j = 0; j < dim.cols; j++) {
            sum += matrix_arr[i * dim.cols + j] * matrix_arr[i * dim.cols + j];
        }
    }
    return sqrtf(sum);
}

Matrix Matrix::operator+ (const Matrix& m) const {
    if (m.dim.rows != dim.rows || m.dim.cols != dim.cols) {
        throw std::length_error (LENGTH_ERROR_MESSAGE);
    }
    // create addition matrix
    Matrix addition_matrix(dim.rows, dim.cols);
    for (int i = 0; i < dim.rows; i++){
        for (int j = 0; j < dim.cols; j++) {
            addition_matrix(i, j) = m(i, j) + matrix_arr[i * dim.cols + j];
        }
    }
    return addition_matrix;
}

Matrix& Matrix::operator= (const Matrix& m) {
    delete[] matrix_arr;
    dim.rows = m.dim.rows;
    dim.cols = m.dim.cols;
    matrix_arr = new float[dim.rows * dim.cols];
    for (int i = 0; i < dim.rows; i++){
        for (int j = 0; j < dim.cols; j++) {
            matrix_arr[i * dim.cols + j] = m(i, j);
        }
    }
    return *this;
}

Matrix Matrix::operator* (const Matrix& m) const {
    if (dim.cols != m.get_rows()) {
        throw std::length_error (LENGTH_ERROR_MESSAGE);
    }
    Matrix matrix_mult(dim.rows, m.get_cols());
    for (int i = 0; i < dim.rows; i++){
        for (int j = 0; j < m.get_cols(); j++) {
            // for each entry in new matrix, sum i'th row of self multiplied with j'th col of m
            for (int k = 0; k < dim.cols; k++) {
                matrix_mult(i, j) += matrix_arr[i * dim.cols + k] * m(k, j);
            }
        }
    }
    return matrix_mult;
}

Matrix Matrix::operator* (float c) const {
    Matrix scalar_mult(dim.rows, dim.cols);
    for (int i = 0; i < dim.rows; i++){
        for (int j = 0; j < dim.cols; j++) {
            scalar_mult(i, j) = c * scalar_mult(i, j);
        }
    }
    return scalar_mult;
}

Matrix operator* (float c, Matrix& m) {
    return (m * c);
};

Matrix& Matrix::operator+= (Matrix& m) {
    if (m.get_rows() != dim.rows || m.get_cols() != dim.cols) {
        throw std::length_error (LENGTH_ERROR_MESSAGE);
    }
    for (int i = 0; i < dim.rows; i++){
        for (int j = 0; j < dim.cols; j++) {
            matrix_arr[i * dim.cols + j] += m(i, j);
        }
    }
    return *this;
}

float &Matrix::operator() (int i, int j) {
    if (i >= dim.rows || i < 0 || j >= dim.cols || j < 0) {
        throw std::out_of_range (OUT_OF_RANGE_ERROR_MESSAGE);
    }
    return matrix_arr[i * dim.cols + j];
}

float Matrix::operator() (int i, int j) const {
    if (i >= dim.rows || i < 0 || j >= dim.cols || j < 0) {
        throw std::out_of_range (OUT_OF_RANGE_ERROR_MESSAGE);
    }
    return matrix_arr[i * dim.cols + j];
}

float &Matrix::operator[] (int i) {
    if (i >= dim.rows * dim.cols || i < 0 ) {
        throw std::out_of_range (OUT_OF_RANGE_ERROR_MESSAGE);
    }
    return matrix_arr[i];
}

float Matrix::operator[] (int i) const {
    if (i >= dim.rows * dim.cols || i < 0 ) {
        throw std::out_of_range (OUT_OF_RANGE_ERROR_MESSAGE);
    }
    return matrix_arr[i];
}

std::ostream &operator<< (std::ostream& ostream, Matrix& m) {
    for (int i = 0; i < m.get_rows(); i++){
        for (int j = 0; j < m.get_cols(); j++) {
            if (m(i, j) > ASTERISK_THRESHOLD) {
                ostream << DOUBLE_ASTERISK;
            } else {
                ostream << DOUBLE_SPACE;
            }
        }
        ostream << "\n";
    }
    return ostream;
}

std::ifstream &operator>> (std::ifstream& ifs, Matrix& m) {
    ifs.read((char *) m.matrix_arr,  sizeof (float) * m.get_rows() * m.get_cols());
    if (!ifs) {
        throw std::runtime_error(RUNTIME_ERROR_MESSAGE);
    }
    return ifs;
}

```

9 ex4/MlpNetwork.h

```
//MlpNetwork.h

#ifndef MLPNETWORK_H
#define MLPNETWORK_H

#include "Dense.h"

#define MLP_SIZE 4

/**
 * @struct digit
 * @brief Identified (by Mlp network) digit with
 *        the associated probability.
 * @var value - Identified digit value
 * @var probability - identification probability
 */
typedef struct digit {
    unsigned int value;
    float probability;
} digit;

const matrix_dims img_dims = {28, 28};
const matrix_dims weights_dims[] = {{128, 784},
                                    {64,  128},
                                    {20,  64},
                                    {10,  20}};
const matrix_dims bias_dims[] = {{128, 1},
                                 {64,  1},
                                 {20,  1},
                                 {10,  1}};

// Insert MlpNetwork class here...
class MlpNetwork {
public:
    // constructor
    MlpNetwork(Matrix weights[MLP_SIZE], Matrix biases[MLP_SIZE]);
    // parenthesis operator
    digit operator() (const Matrix& m) const;
private:
    Dense dense_arr[MLP_SIZE];
};
#endif // MLPNETWORK_H
```

10 ex4/MlpNetwork.cpp

```cpp
#include "MlpNetwork.h"

#define LENGTH_ERROR_MESSAGE "error relating to invalid dimensions has occurred"

MlpNetwork::MlpNetwork (Matrix weights[MLP_SIZE], Matrix biases[MLP_SIZE]):
    dense_arr{Dense(weights[0], biases[0], (activation::relu)),Dense(weights[1], biases[1], activation::relu),
            Dense(weights[2], biases[2], activation::relu),Dense(weights[3], biases[3], activation::softmax)}
 {
     if (weights[0].get_rows() != weights[1].get_cols() || weights[1].get_rows() != weights[2].get_cols() ||
        weights[2].get_rows() != weights[3].get_cols()) {
         throw std::length_error(LENGTH_ERROR_MESSAGE);
     }
}


digit MlpNetwork::operator() (const Matrix& m) const {
    Matrix output = m;
    // run all stages
    for (int i = 0; i < MLP_SIZE; i++) {
        output = dense_arr[i](output);
    }
    // find maximum probability in output
    digit max = {0,output[0]};
    for (int j = 0; j < output.get_rows(); j++) {
        if (output[j] > max.probability) {
            max = {(unsigned) j, output[j]};
        }
    }
    return max;
}
```