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

```
1  //Activation.h
2  #ifndef ACTIVATION_H
3  #define ACTIVATION_H
4
5  #include "Matrix.h"
6
7  #define FLOAT_NORM 1.f
8  #define MIN_RELU_VAL 0
9  /**
10  * @enum ActivationType
11  * @brief Indicator of activation function.
12  */
13 enum ActivationType
14 {
15     Relu,
16     Softmax
17 };
18
19 /**
20  * Activation class
21  */
22 class Activation
23 {
24 public:
25     /**
26      * default constructor
27      */
28     Activation() = default;
29
30     /**
31      * constructor for the project
32      * @param actType the activationtype
33      */
34     explicit Activation(ActivationType actType);
35
36     /**
37      * getter for teh activationtype
38      * @return  the activation type
39      */
40     ActivationType getActivationType();
41
42     /**
43      * overload the () function, activate the activation type function on the input matrix
44      * @param input the input matrix
45      * @return matrix after the () operator
46      */
47     Matrix operator()(const Matrix &input);
48
49     /**
50      * overload the () function, activate the activation type function on the input matrix
51      * @param input the input matrix
52      * @return matrix after the () operator
53      */
54     Matrix operator()(const Matrix &input) const;
55
56     /**
57      * default destructor
58      */
59     ~Activation() = default;
```

```
60
61  private:
62      /**
63       * data member the activation type
64       */
65      ActivationType activationType;
66
67      /**
68       * relu activation function
69       * @param matrix  the input matrix
70       * @return matrix after the activation function
71       */
72      static Matrix _reluActivation(const Matrix &matrix);
73
74      /**
75       * softmax activation function
76       * @param matrix  the input matrix
77       * @return matrix after the activation function
78       */
79      static Matrix _softMaxActivation(const Matrix &matrix);
80  };
81
82  #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)
```

```
60      {
61          if (out[i] < MIN_RELU_VAL)
62          {
63              out[i] = MIN_RELU_VAL;
64          }
65      }
66      return out;
67  }
68
69  /**
70   * softmax activation function
71   * @param matrix  the input matrix
72   * @return matrix after the activation function
73   */
74  Matrix Activation::_softMaxActivation(const Matrix &matrix)
75  {
76      int size = matrix.getRows() * matrix.getCols();
77      Matrix out = Matrix(matrix);
78      float sum = 0;
79      for (int i = 0; i < size; ++i)
80      {
81          out[i] = std::exp(out[i]);
82          sum += out[i];
83      }
84      float scalar = (FLOAT_NORM / sum);
85      return scalar * out;
86  }
87
88
89
```

4 Dense.h

```cpp
//
// 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;


```

```
60      /**
61       * getter for the activation data member
62       * @return the activation
63       */
64      Activation &getActivation();
65
66      /**
67       * getter for the activation data member
68       * @return the activation
69       */
70      Activation getActivation() const;
71
72      /**
73       * overload the () operator
74       * @param input , activate the layer on teh input matrix
75       * @return matrix after the layer process
76       */
77      Matrix operator()(Matrix &input);
78
79      /**
80       * overload the () operator
81       * @param input , activate the layer on teh input matrix
82       * @return matrix after the layer process
83       */
84      Matrix operator()(Matrix &input) const;
85
86
87  private:
88      Matrix _weights, _bias;
89      Activation _activation;
90  };
91
92  #endif //CPP_EX1_DENSE_H
93
```

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;
}

```

```
60  /**
61   * getter for the activation data member
62   * @return the activation
63   */
64  Activation Dense::getActivation() const
65  {
66      return this->_activation;
67  }
68
69  /**
70   * overload the () operator
71   * @param input , activate the layer on teh input matrix
72   * @return matrix after the layer process
73   */
74  Matrix Dense::operator()(Matrix &input)
75  {
76      Matrix out = (this->_weights * input) + this->_bias;
77      out = this->_activation(out);
78      return out;
79  }
80
81  /**
82   * overload the () operator
83   * @param input , activate the layer on teh input matrix
84   * @return matrix after the layer process
85   */
86  Matrix Dense::operator()(Matrix &input) const
87  {
88      Matrix out = (this->_weights * input) + this->_bias;
89      out = this->_activation(out);
90      return out;
91  }
```

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

```cpp
// 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;
```

```
60
61      /**
62       * gettter for the cols
63       * @return number of cols
64       */
65      int getCols();
66
67      /**
68       * gettter for the cols
69       * @return number of cols
70       */
71      int getCols() const;
72
73      /**
74       * Transforms a matrix into a coloumn vector Supports function calling concatenation i.e.(1)
75       * Matrix m(5,4);... m.vectorize(), then m.vectorize() + b should be a valid expression
76       */
77      Matrix vectorize();
78
79      /**
80       * Prints matrix elements, no return value. prints space after each element (incl. last element in the row)
81       * prints newline after each row (incl. last row)
82       */
83      void plainPrint() const;
84
85      // OPERATORS:
86      /**
87   *  overload the * operator to multiply matrix and scalar
88   * @param scalar the scalar we want to multiply the matrix by
89   * @param matrix the matrix
90   * @return scalar * matrix
91   */
92      friend Matrix operator*(const float &scalar, const Matrix &matrix);
93
94      /**
95   *  overload the * operator to multiply matrix and scalar
96   * @param scalar the scalar we want to multiply the matrix by
97   * @return scalar * matrix
98   */
99      Matrix operator*(const float &scalar) const;
100
101     /**
102      * overload the * operator, to multiply with other matrix
103      * @param other the other matrix
104      * @return the solution of 2 matrix multiplication
105      */
106     Matrix operator*(const Matrix &other) const;
107
108
109     /**
110      * overload the () operator, to get the i,j cell inside the matrix
111      * @param i the row index
112      * @param j the column index
113      * @return the i,j element inside the matrix
114      */
115     float &operator()(const int &i, const int &j) const;
116
117     /**
118      * overload the () operator, to get the i,j cell inside the matrix
119      * @param i the row index
120      * @param j the column index
121      * @return the i,j element inside the matrix
122      */
123     float &operator()(const int &i, const int &j);
124
125     /**
126      * overload the + operator to add our matrix with our matrix
127      * @param other the other matrix we want to add to our
```

```
128         * @return the addition matrix
129         */
130        Matrix operator+(const Matrix &other) const;
131
132        /**
133         * overload the + operator to add our matrix with our matrix
134         * @param other the other matrix we want to add to our
135         * @return our matrix += the other matrix values
136         */
137        Matrix &operator+=(const Matrix &other);
138
139        /**
140         * place the other matrix values inside ours
141         * @param other the other matrix
142         * @return our matrix with the other values
143         */
144        Matrix &operator=(const Matrix &other);
145
146        /**
147         * overload the [] operator to return the i object in the matrix
148         * @param idx the matrix idx
149         * @return the ith object in teh matrix
150         */
151        float &operator[](int idx);
152
153        /**
154         * overload the [] operator to return the i object in the matrix
155         * @param idx the matrix idx
156         * @return the ith object in teh matrix
157         */
158        float operator[](int idx) const;
159
160    /**
161     * overlaod the << operator, output the matrix values
162     * @param output output stream
163     * @param matrix the matrix we want output its values
164     * @return the output stream
165     */
166        friend std::ostream &operator<<(std::ostream &output, const Matrix &matrix);
167
168    /**
169    * overlaod the >> operator, input the matrix values
170    * @param input the input stream
171    * @param matrix the matrix we want to load values to
172    * @return the input stream
173    */
174        friend std::istream &operator>>(std::istream &input, const Matrix &matrix);
175
176    private:
177        MatrixDims _matrixDims;
178        float *_table{};
179    };
180
181    #endif //MATRIX_H
```

8 Matrix.cpp

```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
```

```
60   */
61  int Matrix::getRows()
62  {
63      return this->_matrixDims.rows;
64  }
65
66  /**
67   * gettter for the cols
68   * @return number of cols
69   */
70  int Matrix::getCols()
71  {
72      return this->_matrixDims.cols;
73  }
74
75  /**
76   * getter for the rows
77   * @return number of rows
78   */
79  int Matrix::getRows() const
80  {
81      return this->_matrixDims.rows;
82  }
83
84  /**
85   * gettter for the cols
86   * @return number of cols
87   */
88  int Matrix::getCols() const
89  {
90      return this->_matrixDims.cols;
91  }
92
93  /**
94   * Transforms a matrix into a coloumn vector Supports function calling concatenation i.e.(1)
95   * Matrix m(5,4);... m.vectorize(), then m.vectorize() + b should be a valid expression
96   */
97  Matrix Matrix::vectorize()
98  {
99      int size = this->_matrixDims.rows * this->_matrixDims.cols;
100     this->_matrixDims.rows = size;
101     this->_matrixDims.cols = MIN_MATRIX_DIM;
102     return *this;
103 }
104
105 /**
106  * Prints matrix elements, no return value. prints space after each element (incl. last element in the row)
107  * prints newline after each row (incl. last row)
108  */
109 void Matrix::plainPrint() const
110 {
111     for (int i = 0; i < this->_matrixDims.rows; ++i)
112     {
113         for (int j = 0; j < this->_matrixDims.cols; ++j)
114         {
115             std::cout << this->_table[i * this->_matrixDims.cols + j];
116             std::cout << " ";
117         }
118         std::cout << std::endl;
119     }
120 }
121
122 /**
123 *  overload the * operator to multiply matrix and scalar
124 * @param scalar the scalar we want to multiply the matrix by
125 * @param matrix the matrix
126 * @return scalar * matrix
127 */
```

```cpp
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
```

```
196     {
197         auto ans = Matrix(*this);
198         for (int i = 0; i < _matrixDims.rows; ++i)
199         {
200             for (int j = 0; j < _matrixDims.cols; ++j)
201             {
202                 ans._table[i * _matrixDims.cols + j] += other(i, j);
203             }
204         }
205         return ans;
206     }
207 }
208 
209 /**
210  * overload the + operator to add our matrix with our matrix
211  * @param other the other matrix we want to add to our
212  * @return our matrix += the other matrix values
213  */
214 Matrix &Matrix::operator+=(const Matrix &other)
215 {
216     if ((other._matrixDims.rows != this->_matrixDims.rows) || ((other._matrixDims.cols != this->_matrixDims.cols)))
217     {
218         std::cerr << ERROR << std::endl;
219         exit(FAILURE);
220     }
221     else
222     {
223         for (int i = 0; i < this->_matrixDims.rows; ++i)
224         {
225             for (int j = 0; j < this->_matrixDims.cols; ++j)
226             {
227                 this->_table[i * this->_matrixDims.cols + j] += other._table[i * other._matrixDims.cols + j];
228             }
229         }
230         return *this;
231     }
232 }
233 
234 /**
235  * place the other matrix values inside ours
236  * @param other the other matrix
237  * @return our matrix with the other values
238  */
239 Matrix &Matrix::operator=(const Matrix &other)
240 {
241     if (this == &other)
242     {
243         return *this; // so we wont delete our own table
244     }
245     delete[](this->_table); // delete the current info
246     this->_matrixDims = other._matrixDims; // update the matrix dimensions
247     this->_table = new float[other._matrixDims.rows * other._matrixDims.cols];
248     std::memcpy(this->_table, other._table, _matrixDims.rows * _matrixDims.cols * sizeof(float));
249     return *this;
250 }
251 
252 /**
253  * overload the () operator, to get the i,j cell inside the matrix
254  * @param i the row index
255  * @param j the column index
256  * @return the i,j element inside the matrix
257  */
258 float &Matrix::operator()(const int &i, const int &j) const
259 {
260     if ((this->_matrixDims.rows <= i) || ((this->_matrixDims.cols <= j)) || i < FIRST_VAL || j < FIRST_VAL)
261     {
262         std::cerr << ERROR << std::endl;
263         exit(FAILURE);
```

```
264         }
265         else
266         {
267             return this->_table[i * this->_matrixDims.cols + j];
268 
269         }
270     }
271 
272     /**
273      * overload the () operator, to get the i,j cell inside the matrix
274      * @param i the row index
275      * @param j the column index
276      * @return the i,j element inside the matrix
277      */
278     float &Matrix::operator()(const int &i, const int &j)
279     {
280         if ((this->_matrixDims.rows <= i) || ((this->_matrixDims.cols <= j)) || i < FIRST_VAL || j < FIRST_VAL)
281         {
282             std::cerr << ERROR << std::endl;
283             exit(FAILURE);
284         }
285         else
286         {
287             return this->_table[i * this->_matrixDims.cols + j];
288         }
289     }
290 
291     /**
292      * overload the [] operator to return the i object in the matrix
293      * @param idx the matrix idx
294      * @return the ith object in teh matrix
295      */
296     float &Matrix::operator[](int idx)
297     {
298         if (this->_matrixDims.rows * this->_matrixDims.cols <= idx || idx < FIRST_VAL)
299         {
300             std::cerr << ERROR << std::endl;
301             exit(FAILURE);
302         }
303         else
304         {
305             return this->_table[idx];
306         }
307     }
308 
309     /**
310      * overload the [] operator to return the i object in the matrix
311      * @param idx the matrix idx
312      * @return the ith object in teh matrix
313      */
314     float Matrix::operator[](int idx) const
315     {
316         if (this->_matrixDims.rows * this->_matrixDims.cols <= idx || idx < FIRST_VAL)
317         {
318             std::cerr << ERROR << std::endl;
319             exit(FAILURE);
320         }
321         else
322         {
323             return this->_table[idx];
324         }
325     }
326 
327     /**
328      * overlaod the << operator, output the matrix values
329      * @param output output stream
330      * @param matrix the matrix we want output its values
331      * @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

```
1  //MlpNetwork.h
2
3  #ifndef MLPNETWORK_H
4  #define MLPNETWORK_H
5
6  #include "Matrix.h"
7  #include "Digit.h"
8  #include "Dense.h"
9
10 #define MLP_SIZE 4
11 #define FIRST_LAYER 0
12 #define SEC_LAYER 1
13 #define THIRD_LAYER 2
14 #define FOURTH_LAYER 3
15 #define NO_PROB 0.0
16 #define NO_IDX 0
17
18 const MatrixDims imgDims = {28, 28};
19 const MatrixDims weightsDims[] = {{128, 784},
20                                   {64,  128},
21                                   {20,  64},
22                                   {10,  20}};
23 const MatrixDims biasDims[] = {{128, 1},
24                                {64,  1},
25                                {20,  1},
26                                {10,  1}};
27
28 /**
29  * MlpNetwork
30  */
31 class MlpNetwork
32 {
33 public:
34     /**
35      * def constructor
36      */
37     MlpNetwork() = default; // default const
38     /**
39      * def destructor
40      */
41     ~MlpNetwork() = default; // default dest
42     /**
43      * mlp constructor
44      * @param weights the weights
45      * @param biases the biases
46      */
47     MlpNetwork(Matrix *weights, Matrix *biases);
48
49     /**
50      * overload () operator, activate the mlp process on the input matrix
51      * @param matrix the input matrix
52      * @return matrix after the mlp process
53      */
54     Digit operator()(const Matrix &matrix);
55
56
57 private:
58     Dense _d1, _d2, _d3, _d4;
59 };
```

```
60
61  #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;
}
```