Contents

1 Basic Test Results

```
Running...

Opening tar file
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

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

#include "Matrix.h"
#include <cmath>

#define DEFAULT 1

/**
 * @enum ActivationType
 * @brief Indicator of activation function.
 */
enum ActivationType
{
    Relu,
    Softmax
};

/**
 * Activation class
 */
class Activation
{
public:

    /**
     *Activation Constructor, accept activation type and activate accordingly
     * @param actType: type of activation, relu or softmax function
     */
    explicit Activation(ActivationType actType);

    /**
     * A getter for the activation type of an activation
     * @return :the activation type of the activation
     */
    ActivationType getActivationType() const
    {
        return _activationType;
    }

    /**
     * An overload to the () operator ,which applies the activation function to a matrix
     * modifying cell values
     * @param i: row index
     * @param j: column index
     * @return : the value in the i'j' cell of the matrix
     */
    Matrix operator()(const Matrix &input);

    /**
     * An overload to the () operator ,which applies the activation function to a matrix
     * copying the value for the const operator
     * @param i: row index
     * @param j: column index
     * @return : the value in the i,j element value of the matrix
     */
    Matrix operator()(const Matrix &input) const;
```

```
60
61  private:
62      ActivationType _activationType;
63  };
64
65  #endif //ACTIVATION_H
```

3 Activation.cpp

```cpp
//Activation.cpp
//yoav eshed
#include "Activation.h"


/**
 *Activation Constructor, accept activation type and activate accordingly
 * @param actType: type of activation, relu or softmax function
 */
Activation::Activation(ActivationType activationTyp
{
    _activationType = activationType;
}

/**
 * A function that preforms the Relu function on the matrix data values, by going over the matrix
 * replaces each value that smaller than zero with zero
 * @param m : the matrix that the relu function will be applied to
 * @return: A matrix with the values of m after relu function has been applied to it.
 */
Matrix reluFunc(const Matrix &m)
{
    {
        int ROWS = m.getRows();
        int COLS = m.getCols();
        Matrix result(m.getRows(), m.getCols());
        for (int i = 0; i < ROWS; i++)
        {
            for (int j = 0; j < COLS; j++)
            {
                if (m(i, j) <= 0)
                {
                    result(i, j) = 0;
                }
                else
                {
                    result(i, j) = m(i, j);
                }
            }
        }
        return result;
    }
}

/**
 * A function that preforms the Softmax function on the matrix data values, by going over the matrix
 * calculating the sum of the exponents of all the matrix values, and then multiply all the matrix
 * by 1/sum , as in the given formula
 * @param m : the matrix that the softMax function will be applied to
 * @return: A matrix with the values of m after softmax function has been applied to it.
 */
Matrix softMaxFunc(const Matrix &m)
{
    int ROWS = m.getRows();
    int COLS = m.getCols();
    float sum = 0;
    Matrix result(m.getRows(), m.getCols());
    for (int i = 0; i < ROWS; i++)
    {
```

5.1

```
60          for (int j = 0; j < COLS; j++)
61          {
62              result(i, j) = std::exp(m(i, j));
63              sum += result(i, j);
64          }
65      }
66      float scalar = (1.f / sum);
67      return scalar * result;
68  }
69
70  /**
71   * An overload to the () operator ,which applies the activation function to a matrix
72   * modifying cell values
73   * @param i: row index
74   * @param j: column index
75   * @return : the value in the i'j' cell of the matrix
76   */
77  Matrix Activation::operator()(const Matrix &input)
78  {
79      if (_activationType == Relu)
80      {
81          return reluFunc(input);
82      }
83      return softMaxFunc(input);
84  }
85
86  /**
87   * An overload to the () operator ,which applies the activation function to a matrix
88   * copying the value for the const operator
89   * @param i: row index
90   * @param j: column index
91   * @return : the value in the i,j element value of the matrix
92   */
93  Matrix Activation::operator()(const Matrix &input) const
94  {
95      if (_activationType == Relu)
96      {
97          return reluFunc(input);
98      }
99      return softMaxFunc(input);
100 }
101
```

4 Dense.h

```cpp
//Dense.h
//yoav eshed
#ifndef DENSE_H
#define DENSE_H

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

/**
 * Dense Class
 */
class Dense
{
public:
    /**
     * Dense constructor
     * @param w: weight matrix
     * @param bias : bias vector
     * @param actType : activation type
     */

    Dense(Matrix &w, Matrix &bias, ActivationType actType);

    /**
     * A getter for the weights matrix
     * @return : int that indicates the matrix columns
     */
    inline const Matrix &getWeights()
    { return _weights; }

    /**
     * A getter for the bias vector
     * @return : int that indicates the matrix columns
     */
    inline const Matrix &getBias()
    { return _bias; }

    /**
     * A getter for the current layer Activation type
     * @return : int that indicates the matrix columns
     */
    inline Activation getActivation()
    { return _activation; }

    /**
     * An overload to the () operator ,which applies the activation function to the input matrix
     * modifying cell values
     * @param i: row index
     * @param j: column index
     * @return : the value in the i'j' cell of the matrix
     */
    Matrix operator()(Matrix &i 7.1 );

    /**
     * An overload to the () operator ,which applies the activation function to the input matrix
     * copying the value for the const operator
     * @param i: row index
     * @param j: column index
     * @return : the value in the i,j element value of the matrix
```

```
60          */
61          Matrix operator()(Matrix &input) const;
62
63
64      private:
65          Matrix _weights;
66          Matrix _bias;
67          Activation _activation;
68      };
69
70      #endif //DENSE_H
```

5 Dense.cpp

```cpp
//Dense.cpp
// yoav eshed
#include "Dense.h"


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

/**
 * An overload to the () operator ,which applies the activation function to the input matrix
 * modifying cell values
 * @param i: row index
 * @param j: column index
 * @return : the value in the i'j' cell of the matrix
 */
Matrix Dense::operator()(Matrix &input)
{
    Matrix result = ((_weights * input) + _bias);
    result = _activation(result);
    return result;
}

/**
 * An overload to the () operator ,which applies the activation function to the input matrix
 * copying the value for the const operator
 * @param i: row index
 * @param j: column index
 * @return : the value in the i,j element value of the matrix
 */
Matrix Dense::operator()(Matrix &input) const
{
    Matrix result = ((_weights * input) + _bias);
    result = _activation(result);
    return result;
}
```

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

```
1  // Matrix.h
2
3  #ifndef MATRIX_H
4  #define MATRIX_H
5
6  #include <iostream>
7
8  #define FIRST_VAL 0
9  #define MIN_INDEX 1
10 #define INPUT_ERROR "ERROR: invalid input"
11 #define MIN_PROBABILITY 0.1f
12 /**
13  * @struct MatrixDims
14  * @brief Matrix dimensions container
15  */
16 typedef struct MatrixDims
17 {
18     int rows, cols;
19 } MatrixDims;
20
21 /**
22  * Matrix Class
23  */
24 class Matrix
25 {
26 public:
27     /**
28      * Matrix constructor
29      * @param rows : matrix number of rows
30      * @param cols : matrix number of cols
31      */
32     Matrix(int rows, int cols);
33
34     /**
35      * A default constructor of the matrix
36      *  constructs 1 by 1 matrix with single element 0
37      */
38     Matrix();
39
40     /**
41      * A constructor for copying a given matrix to the current one.
42      * @param m: the matrix to be copied
43      */
44     Matrix(const Matrix &m);
45
46     /**
47      * A destructor for the function
48      */
49     ~Matrix();
50
51
52     /**
53      * A getter for the matrix rows
54      * @return : int that indicates the matrix rows
55      */
56     int getRows() const;
57
58     /**
59      * A getter for the matrix columns
```

```
60          * @return : int that indicates the matrix columns
61          */
62         int getCols() const;
63
64         /**
65          * A function that takes a matrix and transforms it into a vector
66          * @return a vector in the length of (original matrix length * original matrix width)
67          */
68         Matrix vectorize();
69
70         /**
71          * A function that prints the matrix
72          */
73         void plainPrint() const;
74
75         /**
76          * An assigment operator that assigns given matrix values into the current matrix
77          * @param m: given matrix
78          * @return: the current matrix with the given matrix values
79          */
80         Matrix &operator=(const Matrix &m);
81
82
83         /**
84          * An overload to the multiplication operator, which multiplies one matrix with another
85          * @param m : a given matrix that will be multiplied by our matrix
86          * @return a matrix the is the result of the multiplication between two matrices.
87          */
88         Matrix operator*(const Matrix &m) const;
89
90         /**
91          * An overload to the multiplication operator, which multiplies the current matrix by a scalar
92          * on the right (matrix * scalar)
93          * @param c : the scalar to be multiplied by
94          * @return : the current matrix after it was multiplied by the scalar
95          */
96         Matrix operator*(const float &c) const;
97
98
99         /**
100          *  An overload to the multiplication operator, which multiplies
101          *  the current matrix by a scalar on the left (scalar * matrix)
102          * @param c : the scalar to be multiplied by
103          * @param m : a given matrix that will be multiplied by our matrix
104          * @return : the current matrix after it was multiplied by the scalar
105          */
106         friend Matrix operator*(const float &c, const Matrix &matrix);
107
108         /**
109          * An overload to the addition operator, which adds two matrices together
110          * @param m: the matrix whose values will be added to the current matrix
111          * @return: A matrix with the results of (current matrix + other matrix values) as values
112          */
113         Matrix operator+(const Matrix &m) const;
114
115         /**
116         * An overload the ++ operator , which  adds  matrix with our matrix
117         * @param other the other matrix we want to add to our
118         * @return the current matrix with the addition of the given matrix values to its own values
119         */
120         Matrix &operator+=(const Matrix &m);
121
122         /**
123          * An overload to the () operator , which returns the (i,j) cell of the current matrix,
124          * modifying cell value
125          * @param i: row index
126          * @param j: column index
127          * @return : the value in the i'j' cell of the matrix
```

```
128         */
129        float &operator()(const int &i, const int &j);
130
131        /**
132         * An overload to the () operator , which returns the (i,j) cell of the current matrix,
133         * copying the value for the const operator
134         * @param i: row index
135         * @param j: column index
136         * @return : the value in the i,j element value of the matrix
137         */
138        float operator()(const int &i, const int &j) const;
139
140        /**
141        *An overload to the [] operator , which returns the (i) cell of the matrix, data value,
142        * when i is representing the formula [i*rows+j]
143        * copying the value for the const operator
144        * @param  the matrix idx
145        * @return the ith object in teh matrix
146        */
147        float &operator[](int i);
148
149        /**
150        *An overload to the [] operator , which returns the (i) cell of the matrix, data value,
151        * when i is representing the formula [i*rows+j]
152        * copying the value for the const operator
153        * @param  the matrix idx
154        * @return the ith object in teh matrix
155        */
156        float operator[](int i) const;
157
158        /**
159         * An overload to the >> operator, that inputs the values into the matrix
160         * @param in: the input stream
161         * @param m: the matrix that the values will be loaded to
162         * @return :the input stream
163         */
164        friend std::istream &operator>>(std::istream &input, const Matrix &m);
165
166        /**
167         * An overload to the << operator, that inputs the values into the matrix
168         * @param out: the output stream
169         * @param m: the matrix which values will be the output data
170         * @return :the output stream
171         */
172        friend std::ostream &operator<<(std::ostream &output, const Matrix &m);
173
174    private:
175        MatrixDims _dims;
176        float *_data{};
177    };
178
179    #endif //MATRIX_H
```

8 Matrix.cpp

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

/**
 * Matrix constructor
 * @param rows : matrix number of rows
 * @param cols : matrix number of cols
 */
Matrix::Matrix(int rows, int cols) :
        _dims{rows = rows, cols = cols}
{
    if (rows < MIN_INDEX || cols < MIN_INDEX)
    {
        std::cerr << INPUT_ERROR << std::endl;
        exit(EXIT_FAILURE);
    }
    _data = new float[rows * cols]();
}

/**
 * A default constructor of the matrix
 *  constructs 1 by 1 matrix with single element with zero value
 */
Matrix::Matrix() :
        Matrix(MIN_INDEX, MIN_INDEX)
{
}

/**
 * A constructor for copying a given matrix to the current one.
 * @param m: the matrix to be copied
 */

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

/**
 * A destructor for deleting the matrix
 */
Matrix::~Matrix()
{
    delete[](_data);
}


/**
 * A getter for the matrix rows
 * @return : int that indicates the matrix rows
 */
```

```cpp
int Matrix::getRows() const
{
    return _dims.rows;
}

/**
 * A getter for the matrix cols
 * @return : int that indicates the matrix rows
 */
int Matrix::getCols() const
{
    return _dims.cols;
}

/**
 * A function that takes a matrix and transforms it into a vector
 * @return a vector in the length of (original matrix length * original matrix width)
 */
Matrix Matrix::vectorize()
{
    int newSize = _dims.rows * _dims.cols;
    _dims.rows = newSize;
    _dims.cols = 1;
    return *this;
}

/**
 * A function that prints the matrix
 */
void Matrix::plainPrint() const
{
    for (int i = 0; i < _dims.rows; ++i)
    {
        for (int j = 0; j < _dims.cols; ++j)
        {
            std::cout << ((*this)(i, j)) << " ";
        }
        std::cout << std::endl;
    }
}

/**
 * An assigment operator that assigns given matrix values into the current matrix
 * @param m: given matrix
 * @return: the current matrix with the given matrix values
 */
Matrix &Matrix::operator=(const Matrix &m)
{
    if (this != &m)
    {
        delete[] _data;
        _dims = m._dims;
        _data = new float[m._dims.rows * m._dims.cols];
        for (int i = 0; i < _dims.rows * _dims.cols; ++i)
        {
            _data[i] = m._data[i];
        }
    }
    return *this;
}

/**
 * An overload to the multiplication operator, which multiplies one matrix with another
 * @param m : a given matrix that will be multiplied by our matrix
 * @return a matrix the is the result of the multiplication between two matrices.
 */
Matrix Matrix::operator*(const Matrix &m) const
{
```

```
128     if ((m._dims.rows != _dims.cols))
129     {
130         std::cerr << INPUT_ERROR << std::endl;
131         exit(EXIT_FAILURE);
132     }
133     Matrix result = Matrix(_dims.rows, m._dims.cols);
134     for (int i = 0; i < _dims.rows; ++i)
135     {
136         for (int j = 0; j < m._dims.cols; ++j)
137         {
138             for (int k = 0; k < _dims.cols; ++k)
139             {
140                 result(i, j) += (*this)(i, k) * m(k, j);
141             }
142         }
143     }
144     return result;
145 }
146
147
148 /**
149  * An overload to the multiplication operator, which multiplies the current matrix by a scalar
150  * on the right (matrix * scalar)
151  * @param c : the scalar to be multiplied by
152  * @return : the current matrix after it was multiplied by the scalar
153  */
154 Matrix Matrix::operator*(const float &c) const
155 {
156     Matrix result = *this;
157     for (int i = 0; i < _dims.rows * _dims.cols; ++i)
158     {
159         result._data[i] *= c;
160     }
161     return result;
162 }
163
164 /**
165  *  An overload to the multiplication operator, which multiplies
166  *  the current matrix by a scalar on the left (scalar * matrix)
167  * @param c : the scalar to be multiplied by
168  * @param m : a given matrix that will be multiplied by our matrix
169  * @return : the current matrix after it was multiplied by the scalar
170  */
171 Matrix operator*(const float &c, const Matrix &m)
172 {
173     Matrix result = m;
174     for (int i = 0; i < m._dims.rows * m._dims.cols; ++i)
175     {
176         result._data[i] *= c;
177     }
178     return result;
179 }
180
181 /**
182  * An overload to the addition operator, which adds two matrices together
183  * @param m: the matrix whose values will be added to the current matrix
184  * @return: A matrix with the results of (current matrix + other matrix values) as values
185  */
186 Matrix Matrix::operator+(const Matrix &m) const
187 {
188     if ((m._dims.rows != _dims.rows) || ((m._dims.cols != _dims.cols)))
189     {
190         std::cerr << INPUT_ERROR << std::endl;
191         exit(EXIT_FAILURE);
192     }
193     Matrix result = Matrix(*this);
194     for (int i = 0; i < _dims.rows; ++i)
195     {
```

```
196         for (int j = 0; j < _dims.cols; ++j)
197         {
198             result(i, j) += m(i, j);
199         }
200     }
201     return result;
202 }
203 
204 /**
205 * An overload the += operator , which  adds  matrix with our matrix
206 * @param other the other matrix we want to add to our
207 * @return the current matrix with the addition of the given matrix values to its own values
208 */
209 Matrix &Matrix::operator+=(const Matrix &m)
210 {
211     if (_dims.rows != m._dims.rows || _dims.cols != m._dims.cols)
212     {
213         std::cerr << "INPUT_ERROR" << std::endl;
214         exit(EXIT_FAILURE);
215     }
216     for (int i = 0; i < _dims.rows; ++i)
217     {
218         for (int j = 0; j < _dims.cols; ++j)
219         {
220             (*this)(i, j) += m(i, j);
221         }
222     }
223     return *this;
224 }
225 
226 /**
227  * An overload to the () operator , which returns the (i,j) cell of the current matrix,
228  * modifying cell value
229  * @param i: row index
230  * @param j: column index
231  * @return  the value in the i'j' cell of the matrix
232  */
233 float &Matrix::operator()(const int &i, const int &j)
234 {
235     if ((_dims.rows <= i) || ((_dims.cols <= j)) || i < FIRST_VAL || j < FIRST_VAL)
236     {
237         std::cerr << INPUT_ERROR << std::endl;
238         exit(EXIT_FAILURE);
239     }
240     return _data[i * _dims.cols + j];
241 }
242 
243 /**
244  * An overload to the () operator , which returns the (i,j) cell of the current matrix,
245  * copying the value for the const operator
246  * @param i: row index
247  * @param j: column index
248  * @return : the value in the i,j element value of the matrix
249  */
250 float Matrix::operator()(const int &i, const int &j) const
251 {
252     if ((_dims.rows <= i) || ((_dims.cols <= j)) || i < FIRST_VAL || j < FIRST_VAL)
253     {
254         std::cerr << INPUT_ERROR << std::endl;
255         exit(EXIT_FAILURE);
256     }
257     return _data[i * _dims.cols + j];
258 }
259 
260 /**
261  *An overload to the [] operator , which returns the (i) cell of the matrix, data value,
262  * when i is representing the formula [i*rows+j]
263  * modifying cell value
```

```
264  * @param  the matrix idx
265  * @return the ith object in teh matrix
266  */
267 float &Matrix::operator[](int i)
268 {
269     if (_dims.rows * _dims.cols <= i || i < FIRST_VAL)
270     {
271         std::cerr << INPUT_ERROR << std::endl;
272         exit(EXIT_FAILURE);
273     }
274     return _data[i];
275 }
276
277 /**
278  *An overload to the [] operator , which returns the (i) cell of the matrix, data value,
279  * when i is representing the formula [i*rows+j]
280  * copying the value for the const operator
281  * @param  the matrix idx
282  * @return the ith object in teh matrix
283  */
284 float Matrix::operator[](int i) const
285 {
286     if (_dims.rows * _dims.cols <= i || i < FIRST_VAL)
287     {
288         std::cerr << INPUT_ERROR << std::endl;
289         exit(EXIT_FAILURE);
290     }
291     return _data[i];
292 }
293
294 /**
295  * An overload to the >> operator, that inputs the values into the matrix
296  * @param in: the input stream
297  * @param m: the matrix that the values will be loaded to
298  * @return :the input stream
299  */
300 std::istream &operator>>(std::istream &input, const Matrix &m)
301 {
302     int i = 0;
303     int size = m._dims.rows * m._dims.cols + 1;
304     while (input.good())
305     {
306         if (i == size)
307         {
308             std::cerr << INPUT_ERROR << std::endl;
309             exit(EXIT_FAILURE);
310         }
311         input.read(reinterpret_cast<char *>(&m._data[i]), sizeof(float));
312         i++;
313     }
314     if (i != size)
315     {
316         std::cerr << INPUT_ERROR << std::endl;
317         exit(EXIT_FAILURE);
318     }
319     return input;
320 }
321
322 /**
323  * An overload to the << operator, that inputs the values into the matrix
324  * @param out: the output stream
325  * @param m: the matrix which values will be the output data
326  * @return :the output stream
327  */
328 std::ostream &operator<<(std::ostream &output, const Matrix &m)
329 {
330     for (int i = 0; i < m._dims.rows; ++i)
331     {
```

```
332             for (int j = 0; j < m._dims.cols; ++j)
333             {
334                 if (m(i, j) <= MIN_PROBABILITY)
335                 {
336                     output << "  ";
337                 }
338                 else
339                 {
340                     output << "**";
341                 }
342             }
343             output << std::endl;
344         }
345         return output;
346     }
347
348
```

9 MlpNetwork.h

```
1  //MlpNetwork.h
2  // yoav eshed
3  #ifndef MLPNETWORK_H
4  #define MLPNETWORK_H
5
6  #include "Matrix.h"
7  #include "Dense.h"
8  #include "Digit.h"
9
10 #define MLP_SIZE 4
11 #define FIRST 0
12 #define SECOND 1
13 #define THIRD 2
14 #define FOURTH 3
15
16 const MatrixDims imgDims = {28, 28};
17 const MatrixDims weightsDims[] = {{128, 784},
18                                   {64,  128},
19                                   {20,  64},
20                                   {10,  20}};
21 const MatrixDims biasDims[] = {{128, 1},
22                                {64,  1},
23                                {20,  1},
24                                {10,  1}};
25 /**
26  * Mlp network class
27  */
28 class MlpNetwork
29 {
30
31 public:
32     /**
33      * A Mlp network constructor
34      * @param weights : weights array
35      * @param biases : biases array
36      * @param layer1 : first layer of the Mlp network
37      * @param layer2 : second layer of the Mlp network
38      * @param layer3 : third layer of the Mlp network
39      * @param layer4 : fourth layer of the Mlp network
40      */
41     MlpNetwork(Matrix *weights, Matrix *biases);
42
43 /**
44      * An overload to the () operator, activates the process on the input matrix and creates the
45      * mlp network
46      * @param matrix the input matrix
47      * @return A digit with the wanted value and probability
48      */
49     Digit operator()(const Matrix &m);
50
51 private:
52     Dense _layer1;
53     Dense _layer2;
54     Dense _layer3;
55     Dense _layer4;
56 };
57
58 #endif // MLPNETWORK_H
```

10 MlpNetwork.cpp

```cpp
//MlpNetwork.cpp
//yoav
#include "MlpNetwork.h"

/**
 * An Mlp network constructor
 * @param weights :weights matrix
 * @param biases :biases matrix
 */
MlpNetwork::MlpNetwork(Matrix *weights, Matrix *biases) :
        _layer1(Dense(weights[FIRST], biases[FIRST], Relu)),
        _layer2(Dense(weights[SECOND], biases[SECOND], Relu)),
        _layer3(Dense(weights[THIRD], biases[THIRD], Relu)),
        _layer4(Dense(weights[FOURTH], biases[FOURTH], Softmax))
{
}


/**
 * An overload to the () operator, activates the process on the input matrix and creates the
 * mlp network
 * @param matrix the input matrix
 * @return A digit with the wanted value and probability
 */
Digit MlpNetwork::operator()(const Matrix &m)
{
    Matrix result = Matrix(m);
    result = _layer1(result);
    result = _layer2(result);
    result = _layer3(result);
    result = _layer4(result);
    int index = 0;
    float probability = 0.0;
    for (int i = 0; i < result.getRows(); ++i)
    {
        if (result[i] > probability)
        {
            probability = result[i];
            index = i;
        }
    }
    Digit digit;
    digit.value = index;
    digit.probability = probability;
    return digit;
}


```

Index of comments