

BEHLUL

0.9

Generated by Doxygen 1.8.15

1 BEHLUL	1
1.1 Layers	1
1.2 State Farm Distracted Driver Detection (Final Project)	1
1.2.1 How to run?	1
1.2.2 How to use Behlul in python?	2
1.2.3 Data	2
1.2.4 Data Preprocess	2
1.2.5 Model	2
1.2.6 Result	2
1.3 MNIST classifier (Milestone)	3
1.3.1 How to run?	3
1.3.2 Data	3
1.3.3 Model	3
1.3.4 Result	3
2 Class Index	5
2.1 Class List	5
3 File Index	7
3.1 File List	7
4 Class Documentation	9
4.1 Conv_Layer Class Reference	9
4.1.1 Detailed Description	10
4.1.2 Constructor & Destructor Documentation	11
4.1.2.1 Conv_Layer()	11
4.1.3 Member Function Documentation	11
4.1.3.1 back_propagation()	12
4.1.3.2 clear_output()	13
4.1.3.3 feed_forward()	14
4.1.3.4 load_filters()	14
4.1.3.5 save_filters()	15
4.1.3.6 set_input()	15
4.1.3.7 update_weights()	16
4.1.4 Member Data Documentation	16
4.1.4.1 accumulated_gradient_filters	16
4.1.4.2 depth	16
4.1.4.3 filter_size	17
4.1.4.4 filters	17
4.1.4.5 gradient_filters	17
4.1.4.6 gradients	17
4.1.4.7 height	17
4.1.4.8 input	18

4.1.4.9 num_filters	18
4.1.4.10 output	18
4.1.4.11 stride	18
4.1.4.12 width	18
4.2 Cross_Entropy Class Reference	19
4.2.1 Detailed Description	19
4.2.2 Constructor & Destructor Documentation	20
4.2.2.1 Cross_Entropy()	20
4.2.3 Member Function Documentation	20
4.2.3.1 back_propagation()	20
4.2.3.2 feed_forward()	20
4.2.4 Member Data Documentation	21
4.2.4.1 actual	21
4.2.4.2 gradients	21
4.2.4.3 loss	21
4.2.4.4 predicted	21
4.3 Dense_Layer Class Reference	22
4.3.1 Detailed Description	23
4.3.2 Constructor & Destructor Documentation	23
4.3.2.1 Dense_Layer()	23
4.3.3 Member Function Documentation	24
4.3.3.1 back_propagation()	24
4.3.3.2 clear_output()	25
4.3.3.3 feed_forward()	25
4.3.3.4 load_weights()	25
4.3.3.5 save_weights()	26
4.3.3.6 set_input()	26
4.3.3.7 update_weights()	27
4.3.4 Member Data Documentation	27
4.3.4.1 accumulated_gradient_biases	27
4.3.4.2 accumulated_gradient_weights	28
4.3.4.3 accumulated_gradients	28
4.3.4.4 biases	28
4.3.4.5 depth	28
4.3.4.6 gradient_biases	28
4.3.4.7 gradient_weights	29
4.3.4.8 gradients	29
4.3.4.9 height	29
4.3.4.10 input	29
4.3.4.11 num_outputs	29
4.3.4.12 output	30
4.3.4.13 weights	30

4.3.4.14 width	30
4.4 Max_Pool Class Reference	30
4.4.1 Detailed Description	31
4.4.2 Constructor & Destructor Documentation	32
4.4.2.1 Max_Pool()	32
4.4.3 Member Function Documentation	32
4.4.3.1 back_propagation()	32
4.4.3.2 clear_output()	33
4.4.3.3 feed_forward()	33
4.4.3.4 set_input()	34
4.4.4 Member Data Documentation	34
4.4.4.1 depth	34
4.4.4.2 filter_size	34
4.4.4.3 gradients	35
4.4.4.4 height	35
4.4.4.5 input	35
4.4.4.6 output	35
4.4.4.7 stride	35
4.4.4.8 width	36
4.5 ReLU Class Reference	36
4.5.1 Detailed Description	37
4.5.2 Constructor & Destructor Documentation	37
4.5.2.1 ReLU()	37
4.5.3 Member Function Documentation	38
4.5.3.1 back_propagation()	38
4.5.3.2 clear_output()	38
4.5.3.3 feed_forward()	38
4.5.3.4 set_input()	39
4.5.4 Member Data Documentation	39
4.5.4.1 depth	39
4.5.4.2 gradients	39
4.5.4.3 height	40
4.5.4.4 input	40
4.5.4.5 output	40
4.5.4.6 width	40
4.6 Softmax Class Reference	40
4.6.1 Detailed Description	41
4.6.2 Constructor & Destructor Documentation	42
4.6.2.1 Softmax()	42
4.6.3 Member Function Documentation	42
4.6.3.1 back_propagation()	42
4.6.3.2 feed_forward()	42

4.6.3.3 <code>set_input()</code>	43
4.6.4 Member Data Documentation	43
4.6.4.1 gradients	43
4.6.4.2 input	43
4.6.4.3 output	44
5 File Documentation	45
5.1 <code>/home/mustafa/DLCP/libdl/binds.cpp</code> File Reference	45
5.2 <code>/home/mustafa/DLCP/libdl/layers/conv_layer.cpp</code> File Reference	45
5.3 <code>/home/mustafa/DLCP/libdl/layers/conv_layer.hpp</code> File Reference	45
5.4 <code>/home/mustafa/DLCP/libdl/layers/cross_entropy.cpp</code> File Reference	46
5.5 <code>/home/mustafa/DLCP/libdl/layers/cross_entropy.hpp</code> File Reference	46
5.6 <code>/home/mustafa/DLCP/libdl/layers/dense_layer.cpp</code> File Reference	46
5.7 <code>/home/mustafa/DLCP/libdl/layers/dense_layer.hpp</code> File Reference	46
5.8 <code>/home/mustafa/DLCP/libdl/layers/max_pool.cpp</code> File Reference	47
5.9 <code>/home/mustafa/DLCP/libdl/layers/max_pool.hpp</code> File Reference	47
5.10 <code>/home/mustafa/DLCP/libdl/layers/relu.cpp</code> File Reference	47
5.10.1 Function Documentation	47
5.10.1.1 <code>derivative()</code>	47
5.11 <code>/home/mustafa/DLCP/libdl/layers/relu.hpp</code> File Reference	48
5.12 <code>/home/mustafa/DLCP/libdl/layers/softmax.cpp</code> File Reference	48
5.13 <code>/home/mustafa/DLCP/libdl/layers/softmax.hpp</code> File Reference	48
5.14 <code>/home/mustafa/DLCP/libdl/main.cpp</code> File Reference	48
5.14.1 Function Documentation	49
5.14.1.1 <code>main()</code>	49
5.15 <code>/home/mustafa/DLCP/libdl/README.md</code> File Reference	52
Index	53

Chapter 1

BEHLUL

My awesome deep learning library from scratch with C++. BEHLUL is acronym for Behlul is an Efficient High Level Useful Library. Name is inspired from the famous fictitious Turkish novel character Behlul Ziyagil.

1.1 Layers

All layers are implemented in different classes. Sizes of outputs and inputs must match. Here are the classes:

- Convolution Layer
- [ReLU](#) Layer
- Max Pool Layer
- Dense Layer
- [Softmax](#) Layer
- Cross Entropy Layer

Xor network is deprecated and dense layer is implemented all over again.

1.2 State Farm Distracted Driver Detection (Final Project)

1.2.1 How to run?

To run State Farm Distracted Driver Detection from python notebook, run those commands in the same order.

```
$ mkdir build
$ cd build
$ cmake ..
$ make
$ cd ..
$ python3 -m venv .venv
$ source .venv/bin/activate
$ pip install jupyter Pillow numpy pytest matplotlib
$ jupyter notebook
```

Open `state_farm_cnn.ipynb`. Then run the cells of the notebook in an order according to your purpose (run all, run pretrained etc.)

To run unit tests, run those commands in the same order.

```
$ cd build
$ cmake ..
$ make
$ ./unit_test_main
```

1.2.2 How to use Behlul in python?

When Behlul is compiled in previous step, it generates a file named `my_project.cpython-36m-x86_64-linux-gnu.so` in build folder. I created my notebook in root folder, so i import behlul as `import build.my_project as behlul` and here is an example conv layer created from Behlul: `conv = behlul.Conv_Layer (28, 28, 1, 5, 1, 6)`.

1.2.3 Data

I get the Driver Images data from [kaggle](#) as zip file. I extracted `state-farm-distracted-driver-detection/imgs/train` to `data/train` folder. I only used train data as my whole dataset for simplicity. I splitted the given train data to ~80% of it as train data, ~10% of it as validation data and remaining ~10% of it as test data. Pretrained weights are also kept under data folder as `data/state_conv1.out`, `data/state_conv2.out`, `data/state_dense.out`.

1.2.4 Data Preprocess

All images are 480x640. I cropped 80px from left and rights to make images square for my model. As main attraction points are almost always in the middle of the images, I didn't lose much useful information. Initially, I was going to resize images to 224x224 to run AlexNet. However, when I tried my old model from MNIST with resized 28x28 new images, I got some successful results (50% acc. for 1 epoch with 10 classes). Then, I didn't go further and stayed with my model.

1.2.5 Model

I used the same model as I used in MNIST.

- [Conv_Layer](#) 1: input: 28x28x1 filter: 5x5x1 num_filters: 6 stride: 1 output: 24x24x6
- [ReLU](#) 1: output: 24x24x6
- [Max_Pool](#) 1: input: 24x24x6 filter: 2x2x6 stride: 2 output: 12x12x6
- [Conv_Layer](#) 2: input: 12x12x6 filter: 5x5x6 num_filters: 16 stride: 1 output: 8x8x16
- [ReLU](#) 2: output: 8x8x16
- [Max_Pool](#) 2: input: 8x8x16 filter: 2x2x16 stride: 2 output: 4x4x16
- [Dense_Layer](#): input: 4x4x16 output: 1x10
- [Softmax](#)
- [Cross_Entropy](#)

1.2.6 Result

I ran the data 5 epoch and it took 30 mins to get output.

```
Training accuracy: 0.78
Validation accuracy: 0.75
Test accuracy: 0.72
```


1.3 MNIST classifier (Milestone)

1.3.1 How to run?

To run MNIST classifier from c++ main, run those commands in the same order.

```
$ mkdir build
$ cd build
$ cmake ..
$ make
$ ./run_main
```

If you don't want to train data all over again and want to use pretrained weights, press `y` and then `enter` when the program asks after the start. Otherwise, press another character `not y` and then `enter` to standard long hours training.

Normal training takes ~ 4 mins. By using pretrained weights, running train set and validation set takes ~ 40 secs. By using pretrained weights, running only validation set takes ~ 7 secs.

So I decided to use train set and validation set with pretrained weights for demo purpose. (I printed the index on every 1000 example to keep track.)

1.3.2 Data

I get the MNIST data from [kaggle](#) as csv files. I read the data from `data/train.csv` and `data/test.csv`. I splitted the given train data to 90% of it as train data and remaining 10% of it as validation data. I get the output of the test data and send it to kaggle competition. Pretrained weights are also kept under `data` folder as `data/conv1.out`, `data/conv2.out`, `data/dense.out`.

1.3.3 Model

- [Conv_Layer](#) 1: input: 28x28x1 filter: 5x5x1 num_filters: 6 stride: 1 output: 24x24x6
- [ReLU](#) 1: output: 24x24x6
- [Max_Pool](#) 1: input: 24x24x6 filter: 2x2x6 stride: 2 output: 12x12x6
- [Conv_Layer](#) 2: input: 12x12x6 filter: 5x5x6 num_filters: 16 stride: 1 output: 8x8x16
- [ReLU](#) 2: output: 8x8x16
- [Max_Pool](#) 2: input: 8x8x16 filter: 2x2x16 stride: 2 output: 4x4x16
- [Dense_Layer](#): input: 4x4x16 output: 1x10
- [Softmax](#)
- [Cross_Entropy](#)

1.3.4 Result

I ran the data 1 epoch and it took 4 mins to get output.

```
Training accuracy: 0.970026
Validation accuracy: 0.96881
Test accuracy: 0.96700
```


Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Conv_Layer	Convolutional Layer	9
Cross_Entropy	Cross Entropy Layer	19
Dense_Layer	Dense Layer	22
Max_Pool	Maximum Pooling Layer	30
ReLU	ReLU Layer	36
Softmax	Softmax Layer	40

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

/home/mustafa/DLCPP/libdl/ binds.cpp	45
/home/mustafa/DLCPP/libdl/ main.cpp	48
/home/mustafa/DLCPP/libdl/layers/ conv_layer.cpp	45
/home/mustafa/DLCPP/libdl/layers/ conv_layer.hpp	45
/home/mustafa/DLCPP/libdl/layers/ cross_entropy.cpp	46
/home/mustafa/DLCPP/libdl/layers/ cross_entropy.hpp	46
/home/mustafa/DLCPP/libdl/layers/ dense_layer.cpp	46
/home/mustafa/DLCPP/libdl/layers/ dense_layer.hpp	46
/home/mustafa/DLCPP/libdl/layers/ max_pool.cpp	47
/home/mustafa/DLCPP/libdl/layers/ max_pool.hpp	47
/home/mustafa/DLCPP/libdl/layers/ relu.cpp	47
/home/mustafa/DLCPP/libdl/layers/ relu.hpp	48
/home/mustafa/DLCPP/libdl/layers/ softmax.cpp	48
/home/mustafa/DLCPP/libdl/layers/ softmax.hpp	48

Chapter 4

Class Documentation

4.1 Conv_Layer Class Reference

Convolutional Layer.

```
#include <conv_layer.hpp>
```

Collaboration diagram for Conv_Layer:

Public Member Functions

- [Conv_Layer](#) (int [height](#), int [width](#), int [depth](#), int [filter_size](#), int [stride](#), int [num_filters](#))
Create a [Conv_Layer](#).
- void [set_input](#) (vector< MatrixXd > [input](#))
Set input.
- void [clear_output](#) ()
Clear output.
- void [feed_forward](#) (vector< MatrixXd > [input](#))
Forward pass of the Convolutional Layer.
- void [back_propagation](#) (vector< MatrixXd > upstream_gradient)
Backward pass of the Convolutional Layer.
- void [update_weights](#) (int batch_size, double learning_rate)
Update filters of the Convolutional Layer.
- void [save_filters](#) (string dir)
Save filters to local file.
- void [load_filters](#) (string dir)
Load filters from local file.

Public Attributes

- int [height](#)
height of the input
- int [width](#)
width of the input
- int [depth](#)
depth of the input
- int [filter_size](#) = 3
height and width of the square filter
- int [stride](#) = 1
stride of the filter
- int [num_filters](#) = 1
number of filters
- vector< MatrixXd > [input](#)
input of the layer
- vector< MatrixXd > [output](#)
output of the layer
- vector< MatrixXd > [gradients](#)
gradients of the layer
- vector< vector< MatrixXd > > [filters](#)
filters of the layer
- vector< vector< MatrixXd > > [gradient_filters](#)
gradients of the filters
- vector< vector< MatrixXd > > [accumulated_gradient_filters](#)
accumulated gradients of the filters

4.1.1 Detailed Description

Convolutional Layer.

This class is used as Convolutional Layer of a neural network. It gets details of its input and filter initially.

Author

Author

Mustafa Erdogan

Version

Revision

0.87

Date

Date

24/07/2019 14:16:20

Contact: mustafa.erdogan@tum.de

Definition at line 34 of file conv_layer.hpp.

4.1.2 Constructor & Destructor Documentation

4.1.2.1 Conv_Layer()

```
Conv_Layer::Conv_Layer (
    int height,
    int width,
    int depth,
    int filter_size,
    int stride,
    int num_filters )
```

Create a [Conv_Layer](#).

Parameters

<i>height</i>	height of the input
<i>width</i>	width of the input
<i>depth</i>	depth of the input
<i>filter_size</i>	height and width of the square filter
<i>stride</i>	stride of the filter
<i>num_filters</i>	number of filters

This method creates a Convolutional Layer. Initializes given values. Initializes filters with uniform random values.

Definition at line 3 of file conv_layer.cpp.

```
4
5     this->height = height;
6     this->width = width;
7     this->depth = depth;
8     this->filter_size = filter_size;
9     this->stride = stride;
10    this->num_filters = num_filters;
11
12    gradients.resize(this->depth);
13    filters.resize(this->num_filters);
14
15    for (int i = 0; i < this->num_filters; i++) {
16        for (int j = 0; j < this->depth; j++) {
17            filters[i].push_back(MatrixXd::Random(filter_size, filter_size));
18        }
19    }
20
21    accumulated_gradient_filters.resize(num_filters);
22    for (int f = 0; f < num_filters; f++)
23        for (int d = 0; d < depth; d++)
24            accumulated_gradient_filters[f].push_back(
25                MatrixXd::Zero(filter_size, filter_size));
26 }
```

4.1.3 Member Function Documentation

4.1.3.1 back_propagation()

```
void Conv_Layer::back_propagation (
    vector< MatrixXd > upstream_gradient )
```

Backward pass of the Convolutional Layer.

Parameters

<i>upstream_gradient</i>	gradients coming from the next layer
--------------------------	--------------------------------------

This function iterates backward pass of the Convolutional Layer. As input it gets the next layer's gradients.

Definition at line 63 of file conv_layer.cpp.

```

63                                     {
64     for (int d = 0; d < depth; d++) {
65         gradients[d] = MatrixXd::Zero(height, width);
66     }
67
68     for (int f = 0; f < num_filters; f++) {
69         for (int r = 0; r < output[f].rows(); r++) {
70             for (int c = 0; c < output[f].cols(); c++) {
71                 for (int d = 0; d < depth; d++) {
72                     MatrixXd tmp = MatrixXd::Zero(height, width);
73                     tmp.block(r * stride, c * stride, filter_size, filter_size) =
74                         filters[f][d];
75                     gradients[d] += upstream_gradient[f](r, c) * tmp;
76                 }
77             }
78         }
79     }
80
81     gradient_filters.clear();
82     gradient_filters.resize(num_filters);
83     for (int i = 0; i < num_filters; i++) {
84         gradient_filters[i].resize(depth);
85         for (int j = 0; j < depth; j++) {
86             gradient_filters[i][j] = MatrixXd::Zero(filter_size, filter_size);
87         }
88     }
89
90     for (int f = 0; f < num_filters; f++) {
91         for (int r = 0; r < output[f].rows(); r++) {
92             for (int c = 0; c < output[f].cols(); c++) {
93                 for (int d = 0; d < depth; d++) {
94                     MatrixXd tmp = MatrixXd::Zero(filter_size, filter_size);
95                     tmp =
96                         input[d].block(r * stride, c * stride, filter_size, filter_size);
97                     gradient_filters[f][d] += upstream_gradient[f](r, c) * tmp;
98                 }
99             }
100         }
101     }
102
103     for (int f = 0; f < num_filters; f++)
104         for (int d = 0; d < depth; d++)
105             accumulated_gradient_filters[f][d] += gradient_filters[f][d];
106 }

```

4.1.3.2 clear_output()

```
void Conv_Layer::clear_output ( )
```

Clear output.

This method clears output of the Convolutional Layer and fills with zeros.

Definition at line 30 of file conv_layer.cpp.

```

30                                     {
31     output.clear();
32     for (int i = 0; i < this->num_filters; i++) {
33         output.push_back(MatrixXd::Zero((height - filter_size) / stride + 1,
34                                         (width - filter_size) / stride + 1));
35     }
36 }

```

4.1.3.3 feed_forward()

```
void Conv_Layer::feed_forward (
    vector< MatrixXd > input )
```

Forward pass of the Convolutional Layer.

Parameters

<i>input</i>	input of the Convolutional Layer
--------------	----------------------------------

This function iterates forward pass of the Convolutional Layer. As input it gets the previous layer's output in middle layers or image in starting layer. The output filled is used later from the next layer.

Definition at line 38 of file conv_layer.cpp.

```
38                                     {
39     if ((height - filter_size) % stride != 0) {
40         cout << "Filter dimension and stride is not valid height" << endl;
41         return;
42     }
43     if ((width - filter_size) % stride != 0) {
44         cout << "Filter dimension and stride is not valid for width" << endl;
45         return;
46     }
47     this->set_input(input);
48     this->clear_output();
49
50     for (int f = 0; f < num_filters; f++) {
51         for (int i = 0, r = 0; i < height - filter_size + 1; i += stride, r++) {
52             for (int j = 0, c = 0; j < width - filter_size + 1; j += stride, c++) {
53                 for (int d = 0; d < depth; d++) {
54                     MatrixXd sub_matrix = input[d].block(i, j, filter_size, filter_size);
55                     output[f](r, c) += filters[f][d].cwiseProduct(sub_matrix).sum();
56                 }
57             }
58         }
59     }
60 }
61 }
```

Here is the call graph for this function:

4.1.3.4 load_filters()

```
void Conv_Layer::load_filters (
    string dir )
```

Load filters from local file.

Parameters

<i>dir</i>	file path to load filters
------------	---------------------------

This function loads filters of the Convolutional Layer to use in pretrained demo.

Definition at line 136 of file conv_layer.cpp.

```
136                                     {
137     ifstream fin(filters_file);
138     int num_filt, num_depth, num_row, num_col;
139     fin >> num_filt;
140     this->filters.clear();
141     this->filters.resize(num_filt);
142     for (int f = 0; f < num_filt; f++) {
```

```

143     fin » num_depth;
144     this->filters[f].resize(num_depth);
145     for (int d = 0; d < num_depth; d++) {
146         fin » num_row » num_col;
147         MatrixXd temp_filter(num_row, num_col);
148         for (int i = 0; i < num_row; i++) {
149             for (int j = 0; j < num_col; j++) {
150                 fin » temp_filter(i, j);
151             }
152         }
153         this->filters[f][d] = temp_filter;
154     }
155 }
156 fin.close();
157 }

```

4.1.3.5 save_filters()

```

void Conv_Layer::save_filters (
    string dir )

```

Save filters to local file.

Parameters

<i>dir</i>	file path to save filters
------------	---------------------------

This function saves filters of the Convolutional Layer to use later in pretrained demo.

Definition at line 122 of file conv_layer.cpp.

```

122     {
123         ofstream fout(filters_file);
124         fout << this->filters.size() << endl;
125         for (int f = 0; f < this->filters.size(); f++) {
126             fout << this->filters[f].size() << endl;
127             for (int d = 0; d < this->filters[f].size(); d++) {
128                 fout << this->filters[f][d].rows() << " " << this->filters[f][d].cols()
129                     << endl;
130                 fout << this->filters[f][d] << endl;
131             }
132         }
133         fout.close();
134     }

```

4.1.3.6 set_input()

```

void Conv_Layer::set_input (
    vector< MatrixXd > input )

```

Set input.

Parameters

<i>input</i>	input of the Convolutional Layer
--------------	----------------------------------

This method sets input of the Convolutional Layer

Definition at line 28 of file conv_layer.cpp.

```
28 { this->input = input; }
```

4.1.3.7 update_weights()

```
void Conv_Layer::update_weights (
    int batch_size,
    double learning_rate )
```

Update filters of the Convolutional Layer.

Parameters

<i>batch_size</i>	batch size
<i>learning_rate</i>	learning rate

This function updates filters of the Convolutional Layer.

Definition at line 108 of file conv_layer.cpp.

```
108                                     {
109     for (int f = 0; f < num_filters; f++)
110         for (int d = 0; d < depth; d++)
111             filters[f][d] -=
112                 learning_rate * (accumulated_gradient_filters[f][d] / batch_size);
113
114     accumulated_gradient_filters.clear();
115     accumulated_gradient_filters.resize(num_filters);
116     for (int f = 0; f < num_filters; f++)
117         for (int d = 0; d < depth; d++)
118             accumulated_gradient_filters[f].push_back(
119                 MatrixXd::Zero(filter_size, filter_size));
120 }
```

4.1.4 Member Data Documentation

4.1.4.1 accumulated_gradient_filters

```
vector<vector<MatrixXd> > Conv_Layer::accumulated_gradient_filters
```

accumulated gradients of the filters

Definition at line 63 of file conv_layer.hpp.

4.1.4.2 depth

```
int Conv_Layer::depth
```

depth of the input

Definition at line 52 of file conv_layer.hpp.

4.1.4.3 filter_size

```
int Conv_Layer::filter_size = 3
```

height and width of the square filter

Definition at line 53 of file conv_layer.hpp.

4.1.4.4 filters

```
vector<vector<MatrixXd> > Conv_Layer::filters
```

filters of the layer

Definition at line 60 of file conv_layer.hpp.

4.1.4.5 gradient_filters

```
vector<vector<MatrixXd> > Conv_Layer::gradient_filters
```

gradients of the filters

Definition at line 61 of file conv_layer.hpp.

4.1.4.6 gradients

```
vector<MatrixXd> Conv_Layer::gradients
```

gradients of the layer

Definition at line 59 of file conv_layer.hpp.

4.1.4.7 height

```
int Conv_Layer::height
```

height of the input

Definition at line 50 of file conv_layer.hpp.

4.1.4.8 input

```
vector<MatrixXd> Conv_Layer::input
```

input of the layer

Definition at line 57 of file conv_layer.hpp.

4.1.4.9 num_filters

```
int Conv_Layer::num_filters = 1
```

number of filters

Definition at line 55 of file conv_layer.hpp.

4.1.4.10 output

```
vector<MatrixXd> Conv_Layer::output
```

output of the layer

Definition at line 58 of file conv_layer.hpp.

4.1.4.11 stride

```
int Conv_Layer::stride = 1
```

stride of the filter

Definition at line 54 of file conv_layer.hpp.

4.1.4.12 width

```
int Conv_Layer::width
```

width of the input

Definition at line 51 of file conv_layer.hpp.

The documentation for this class was generated from the following files:

- [/home/mustafa/DLCP/libdl/layers/conv_layer.hpp](#)
- [/home/mustafa/DLCP/libdl/layers/conv_layer.cpp](#)

4.2 Cross_Entropy Class Reference

Cross Entropy Layer.

```
#include <cross_entropy.hpp>
```

Collaboration diagram for Cross_Entropy:

Public Member Functions

- [Cross_Entropy](#) (int value)
Create a [Cross_Entropy](#).
- void [feed_forward](#) (VectorXd [predicted](#), VectorXd [actual](#))
Forward pass of the Cross Entropy Layer.
- void [back_propagation](#) ()
Backward pass of the Cross Entropy Layer.

Public Attributes

- double [loss](#)
- VectorXd [predicted](#)
predicted labels
- VectorXd [actual](#)
actual labels
- VectorXd [gradients](#)
gradients of the layer

4.2.1 Detailed Description

Cross Entropy Layer.

This class is used as Cross Entropy Layer Layer of a neural network.

Author

Author

Mustafa Erdogan

Version

Revision

0.87

Date

Date

24/07/2019 14:16:20

Contact: mustafa.erdogan@tum.de

Definition at line 34 of file cross_entropy.hpp.

4.2.2 Constructor & Destructor Documentation

4.2.2.1 Cross_Entropy()

```
Cross_Entropy::Cross_Entropy (
    int value ) [inline]
```

Create a [Cross_Entropy](#).

Parameters

<i>value</i>	This method creates a Cross Entropy Layer.
--------------	--

Definition at line 42 of file cross_entropy.hpp.

```
42 {};
```

4.2.3 Member Function Documentation

4.2.3.1 back_propagation()

```
void Cross_Entropy::back_propagation ( )
```

Backward pass of the Cross Entropy Layer.

This function iterates backward pass of the Cross Entropy Layer. Computes gradients with derivative of cross entropy.

Definition at line 9 of file cross_entropy.cpp.

```
9      {
10  gradients = -(actual.array() * (1 / predicted.array())) .matrix();
11 }
```

4.2.3.2 feed_forward()

```
void Cross_Entropy::feed_forward (
    VectorXd predicted,
    VectorXd actual )
```

Forward pass of the Cross Entropy Layer.

Parameters

<i>predicted</i>	predicted labels
<i>actual</i>	actual labels

This function iterates forward pass of the Cross Entropy Layer. Compares predicted and actual labels and computes the loss with cross entropy.

Definition at line 3 of file cross_entropy.cpp.

```
3
4   this->predicted = predicted;
5   this->actual = actual;
6   this->loss = -actual.dot(predicted.array().log().matrix());
7 }
```

4.2.4 Member Data Documentation

4.2.4.1 actual

`VectorXd Cross_Entropy::actual`

actual labels

Definition at line 46 of file cross_entropy.hpp.

4.2.4.2 gradients

`VectorXd Cross_Entropy::gradients`

gradients of the layer

Definition at line 47 of file cross_entropy.hpp.

4.2.4.3 loss

`double Cross_Entropy::loss`

Definition at line 42 of file cross_entropy.hpp.

4.2.4.4 predicted

`VectorXd Cross_Entropy::predicted`

predicted labels

Definition at line 45 of file cross_entropy.hpp.

The documentation for this class was generated from the following files:

- [/home/mustafa/DLCP/libdl/layers/cross_entropy.hpp](#)
- [/home/mustafa/DLCP/libdl/layers/cross_entropy.cpp](#)

4.3 Dense_Layer Class Reference

Dense Layer.

```
#include <dense_layer.hpp>
```

Collaboration diagram for Dense_Layer:

Public Member Functions

- [Dense_Layer](#) (int [height](#), int [width](#), int [depth](#), int [num_outputs](#))
Create a [Dense_Layer](#).
- void [set_input](#) (vector< MatrixXd > [input](#))
Set input.
- void [clear_output](#) ()
Clear output.
- void [feed_forward](#) (vector< MatrixXd > [input](#))
Forward pass of the Dense Layer.
- void [back_propagation](#) (VectorXd upstream_gradient)
Backward pass of the Dense Layer.
- void [update_weights](#) (int batch_size, double learning_rate)
Update weights of the Dense Layer.
- void [save_weights](#) (string dir)
Save weights to local file.
- void [load_weights](#) (string dir)
Load weights from local file.

Public Attributes

- int [height](#)
height of the input
- int [width](#)
width of the input
- int [depth](#)
depth of the input
- int [num_outputs](#)
number of output classes
- VectorXd [output](#)
output of the layer
- vector< MatrixXd > [input](#)
input of the layer
- MatrixXd [weights](#)
weights of the layer
- VectorXd [biases](#)
biases of the layer
- vector< MatrixXd > [gradients](#)
gradients of the layer
- MatrixXd [gradient_weights](#)
gradients of the weights

- VectorXd [gradient_biases](#)
gradients of the biases
- vector< MatrixXd > [accumulated_gradients](#)
accumulated gradients of the layer
- MatrixXd [accumulated_gradient_weights](#)
accumulated gradients of the weights
- VectorXd [accumulated_gradient_biases](#)
accumulated gradients of the biases

4.3.1 Detailed Description

Dense Layer.

This class is used as Dense Layer of a neural network. It gets details of its input and output.

Author

Author

Mustafa Erdogan

Version

Revision

0.87

Date

Date

24/07/2019 14:16:20

Contact: mustafa.erdogan@tum.de

Definition at line 37 of file dense_layer.hpp.

4.3.2 Constructor & Destructor Documentation

4.3.2.1 Dense_Layer()

```
Dense_Layer::Dense_Layer (
    int height,
    int width,
    int depth,
    int num_outputs )
```

Create a [Dense_Layer](#).

Parameters

<i>height</i>	height of the input
<i>width</i>	width of the input
<i>depth</i>	depth of the input
<i>num_outputs</i>	number of output classes

This method creates a Dense Layer. Initializes given values. Initializes weights with uniform random values.

Definition at line 3 of file dense_layer.cpp.

```

3
4  this->height = height;
5  this->width = width;
6  this->depth = depth;
7  this->num_outputs = num_outputs;
8
9  weights = MatrixXd::Random(num_outputs, height * width * depth);
10
11  biases = VectorXd::Zero(num_outputs);
12
13  gradients.resize(depth);
14  accumulated_gradients.resize(depth);
15  for (int d = 0; d < depth; d++)
16      accumulated_gradients[d] = MatrixXd::Zero(height, width);
17  accumulated_gradient_weights =
18      MatrixXd::Zero(num_outputs, height * width * depth);
19  accumulated_gradient_biases = VectorXd::Zero(num_outputs);
20 }
```

4.3.3 Member Function Documentation

4.3.3.1 back_propagation()

```
void Dense_Layer::back_propagation (
    VectorXd upstream_gradient )
```

Backward pass of the Dense Layer.

Parameters

<i>upstream_gradient</i>	gradients coming from the next layer
--------------------------	--------------------------------------

This function iterates backward pass of the Dense Layer. As input it gets the next layer's gradients.

Definition at line 34 of file dense_layer.cpp.

```

34
35  VectorXd gradients_vec = VectorXd::Zero(height * width * depth);
36  for (int i = 0; i < height * width * depth; i++)
37      gradients_vec[i] = weights.col(i).dot(upstream_gradient);
38
39  for (int d = 0; d < depth; d++) {
40      gradients[d] = MatrixXd::Zero(height, width);
41  }
42  for (int d = 0; d < depth; d++) {
43      // Vector to matrix
44      Map<MatrixXd> mat(
45          gradients_vec.segment(d * height * width, height * width).data(),
46          height, width);
47      gradients[d] = mat;
48      accumulated_gradients[d] += gradients[d];
```

```

49 }
50
51 gradient_weights = MatrixXd::Zero(weights.rows(), weights.cols());
52 VectorXd flat_input(height * width * depth);
53 for (int d = 0; d < depth; d++) {
54     // Matrix to vector
55     Map<RowVectorXd> flat(input[d].data(), input[d].size());
56     flat_input.segment(d * flat.size(), flat.size()) = flat;
57 }
58 for (int r = 0; r < gradient_weights.rows(); r++) {
59     gradient_weights.row(r) = flat_input.transpose() * upstream_gradient(r);
60 }
61
62 accumulated_gradient_weights += gradient_weights;
63 gradient_biases = upstream_gradient;
64 accumulated_gradient_biases += gradient_biases;
65 }

```

4.3.3.2 clear_output()

```
void Dense_Layer::clear_output ( )
```

Clear output.

This method clears output of the Dense Layer and fills with zeros.

4.3.3.3 feed_forward()

```
void Dense_Layer::feed_forward (
    vector< MatrixXd > input )
```

Forward pass of the Dense Layer.

Parameters

<i>input</i>	input of the Dense Layer
--------------	--------------------------

This function iterates forward pass of the Dense Layer. As input it gets the previous layer's output and convert to a dense input, then uses the input. The output filled is used later from the next layer.

Definition at line 23 of file dense_layer.cpp.

```

23 {
24     this->set_input(input);
25     VectorXd flat_input(height * width * depth);
26     for (int d = 0; d < depth; d++) {
27         // Matrix to vector
28         Map<RowVectorXd> flat(input[d].data(), input[d].size());
29         flat_input.segment(d * flat.size(), flat.size()) = flat;
30     }
31     output = (weights * flat_input) + biases;
32 }

```

Here is the call graph for this function:

4.3.3.4 load_weights()

```
void Dense_Layer::load_weights (
    string dir )
```

Load weights from local file.

Parameters

<i>dir</i>	file path to load weights
------------	---------------------------

This function loads weights of the Convolutional Layer to use in pretrained demo.

Definition at line 89 of file dense_layer.cpp.

```

89                                     {
90     ifstream fin(weights_file);
91     int num_row, num_col, num_size;
92     fin >> num_row >> num_col;
93     MatrixXd temp_weights(num_row, num_col);
94     for (int i = 0; i < num_row; i++) {
95         for (int j = 0; j < num_col; j++) {
96             fin >> temp_weights(i, j);
97         }
98     }
99     this->weights = temp_weights;
100     fin >> num_size;
101     VectorXd temp_biases(num_size);
102     for (int i = 0; i < num_size; i++) {
103         fin >> temp_biases(i);
104     }
105     this->biases = temp_biases;
106     fin.close();
107 }
```

4.3.3.5 save_weights()

```

void Dense_Layer::save_weights (
    string dir )
```

Save weights to local file.

Parameters

<i>dir</i>	file path to save weights
------------	---------------------------

This function saves weights of the Convolutional Layer to use later in pretrained demo.

Definition at line 80 of file dense_layer.cpp.

```

80                                     {
81     ofstream fout(weights_file);
82     fout << this->weights.rows() << " " << this->weights.cols() << endl;
83     fout << this->weights << endl;
84     fout << this->biases.size() << endl;
85     fout << this->biases << endl;
86     fout.close();
87 }
```

4.3.3.6 set_input()

```

void Dense_Layer::set_input (
    vector< MatrixXd > input )
```

Set input.

Parameters

<i>input</i>	input of the Dense Layer
--------------	--------------------------

This method sets input of the Dense Layer

Definition at line 21 of file dense_layer.cpp.

```
21 { this->input = input; }
```

4.3.3.7 update_weights()

```
void Dense_Layer::update_weights (
    int batch_size,
    double learning_rate )
```

Update weights of the Dense Layer.

Parameters

<i>batch_size</i>	batch size
<i>learning_rate</i>	learning rate

This function updates weights of the Dense Layer.

Definition at line 67 of file dense_layer.cpp.

```
67 {
68     weights =
69     weights - learning_rate * (accumulated_gradient_weights / batch_size);
70     biases = biases - learning_rate * (accumulated_gradient_biases / batch_size);
71
72     accumulated_gradients.resize(depth);
73     for (int d = 0; d < depth; d++)
74         accumulated_gradients.push_back(MatrixXd::Zero(height, width));
75     accumulated_gradient_weights =
76         MatrixXd::Zero(num_outputs, height * width * depth);
77     accumulated_gradient_biases = VectorXd::Zero(num_outputs);
78 }
```

4.3.4 Member Data Documentation

4.3.4.1 accumulated_gradient_biases

VectorXd Dense_Layer::accumulated_gradient_biases

accumulated gradients of the biases

Definition at line 68 of file dense_layer.hpp.

4.3.4.2 accumulated_gradient_weights

```
MatrixXd Dense_Layer::accumulated_gradient_weights
```

accumulated gradients of the weights

Definition at line 67 of file dense_layer.hpp.

4.3.4.3 accumulated_gradients

```
vector<MatrixXd> Dense_Layer::accumulated_gradients
```

accumulated gradients of the layer

Definition at line 65 of file dense_layer.hpp.

4.3.4.4 biases

```
VectorXd Dense_Layer::biases
```

biases of the layer

Definition at line 58 of file dense_layer.hpp.

4.3.4.5 depth

```
int Dense_Layer::depth
```

depth of the input

Definition at line 52 of file dense_layer.hpp.

4.3.4.6 gradient_biases

```
VectorXd Dense_Layer::gradient_biases
```

gradients of the biases

Definition at line 62 of file dense_layer.hpp.

4.3.4.7 gradient_weights

```
MatrixXd Dense_Layer::gradient_weights
```

gradients of the weights

Definition at line 61 of file dense_layer.hpp.

4.3.4.8 gradients

```
vector<MatrixXd> Dense_Layer::gradients
```

gradients of the layer

Definition at line 60 of file dense_layer.hpp.

4.3.4.9 height

```
int Dense_Layer::height
```

height of the input

Definition at line 50 of file dense_layer.hpp.

4.3.4.10 input

```
vector<MatrixXd> Dense_Layer::input
```

input of the layer

Definition at line 56 of file dense_layer.hpp.

4.3.4.11 num_outputs

```
int Dense_Layer::num_outputs
```

number of output classes

Definition at line 53 of file dense_layer.hpp.

4.3.4.12 output

```
VectorXd Dense_Layer::output
```

output of the layer

Definition at line 55 of file dense_layer.hpp.

4.3.4.13 weights

```
MatrixXd Dense_Layer::weights
```

weights of the layer

Definition at line 57 of file dense_layer.hpp.

4.3.4.14 width

```
int Dense_Layer::width
```

width of the input

Definition at line 51 of file dense_layer.hpp.

The documentation for this class was generated from the following files:

- /home/mustafa/DLCPP/libdl/layers/dense_layer.hpp
- /home/mustafa/DLCPP/libdl/layers/dense_layer.cpp

4.4 Max_Pool Class Reference

Maximum Pooling Layer.

```
#include <max_pool.hpp>
```

Collaboration diagram for Max_Pool:

Public Member Functions

- [Max_Pool](#) (int [height](#), int [width](#), int [depth](#), int [filter_size](#), int [stride](#))
Create a [Max_Pool](#).
- void [set_input](#) (vector< MatrixXd > [input](#))
Set input.
- void [clear_output](#) ()
Clear output.
- void [feed_forward](#) (vector< MatrixXd > [input](#))
Forward pass of the Maximum Pooling Layer.
- void [back_propagation](#) (vector< MatrixXd > [upstream_gradient](#))
Backward pass of the Maximum Pooling Layer.

Public Attributes

- int [height](#)
height of the input
- int [width](#)
width of the input
- int [depth](#)
depth of the input
- int [filter_size](#) = 2
height and width of the square filter
- int [stride](#) = 2
stride of the filter
- vector< MatrixXd > [input](#)
input of the layer
- vector< MatrixXd > [output](#)
output of the layer
- vector< MatrixXd > [gradients](#)
gradients of the layer

4.4.1 Detailed Description

Maximum Pooling Layer.

This class is used as Maximum Pooling Layer of a neural network. It gets details of its input and filter initially.

Author

Author

Mustafa Erdogan

Version

Revision

0.87

Date

Date

24/07/2019 14:16:20

Contact: mustafa.erdogan@tum.de

Definition at line 33 of file max_pool.hpp.

4.4.2 Constructor & Destructor Documentation

4.4.2.1 Max_Pool()

```
Max_Pool::Max_Pool (
    int height,
    int width,
    int depth,
    int filter_size,
    int stride )
```

Create a [Max_Pool](#).

Parameters

<i>height</i>	height of the input
<i>width</i>	width of the input
<i>depth</i>	depth of the input
<i>filter_size</i>	height and width of the square filter
<i>stride</i>	stride of the filter

This method creates a Maximum Pooling Layer. Initializes given values.

Definition at line 3 of file max_pool.cpp.

```
4      {
5  this->height = height;
6  this->width = width;
7  this->depth = depth;
8  this->filter_size = filter_size;
9  this->stride = stride;
10
11  gradients.resize(this->depth);
12 }
```

4.4.3 Member Function Documentation

4.4.3.1 back_propagation()

```
void Max_Pool::back_propagation (
    vector< MatrixXd > upstream_gradient )
```

Backward pass of the Maximum Pooling Layer.

Parameters

<i>upstream_gradient</i>	gradients coming from the next layer
--------------------------	--------------------------------------

This function iterates backward pass of the Maximum Pooling Layer. As input it gets the next layer's gradients.

Definition at line 47 of file max_pool.cpp.

```

47                                     {
48     for (int d = 0; d < depth; d++) {
49         gradients[d] = MatrixXd::Zero(height, width);
50         for (int i = 0; i + filter_size <= height; i += stride) {
51             for (int j = 0; j + filter_size <= width; j += stride) {
52                 MatrixXd tmp = MatrixXd::Zero(filter_size, filter_size);
53                 int r, c;
54                 input[d].block(i, j, filter_size, filter_size).maxCoeff(&r, &c);
55                 tmp(r, c) = upstream_gradient[d](i / stride, j / stride);
56                 gradients[d].block(i, j, filter_size, filter_size) += tmp;
57             }
58         }
59     }
60 }
```

4.4.3.2 clear_output()

```
void Max_Pool::clear_output ( )
```

Clear output.

This method clears output of the Maximum Pooling Layer and fills with zeros.

Definition at line 16 of file max_pool.cpp.

```

16     {
17     output.clear();
18     for (int i = 0; i < this->depth; i++) {
19         output.push_back(MatrixXd::Zero((height - filter_size) / stride + 1,
20                                         (width - filter_size) / stride + 1));
21     }
22 }
```

4.4.3.3 feed_forward()

```
void Max_Pool::feed_forward (
    vector< MatrixXd > input )
```

Forward pass of the Maximum Pooling Layer.

Parameters

<i>input</i>	input of the Maximum Pooling Layer
--------------	------------------------------------

This function iterates forward pass of the Maximum Pooling Layer. As input it gets the previous layer's output. The output filled is used later from the next layer.

Definition at line 24 of file max_pool.cpp.

```

24                                     {
25     if ((height - filter_size) % stride != 0) {
26         cout << "Filter dimension and stride is not valid height" << endl;
27         return;
28     }
29     if ((width - filter_size) % stride != 0) {
30         cout << "Filter dimension and stride is not valid for width" << endl;
31         return;
32     }
33
34     this->set_input(input);
35     this->clear_output();
```

```

36
37     for (int d = 0; d < depth; d++) {
38         for (int i = 0, r = 0; i < height - filter_size + 1; i += stride, r++) {
39             for (int j = 0, c = 0; j < width - filter_size + 1; j += stride, c++) {
40                 MatrixXd sub_matrix = input[d].block(i, j, filter_size, filter_size);
41                 output[d](r, c) = sub_matrix.maxCoeff();
42             }
43         }
44     }
45 }

```

Here is the call graph for this function:

4.4.3.4 set_input()

```

void Max_Pool::set_input (
    vector< MatrixXd > input )

```

Set input.

Parameters

<i>input</i>	input of the Maximum Pooling Layer
--------------	------------------------------------

This method sets input of the Maximum Pooling Layer

Definition at line 14 of file max_pool.cpp.

```
14 { this->input = input; }
```

4.4.4 Member Data Documentation

4.4.4.1 depth

```
int Max_Pool::depth
```

depth of the input

Definition at line 48 of file max_pool.hpp.

4.4.4.2 filter_size

```
int Max_Pool::filter_size = 2
```

height and width of the square filter

Definition at line 49 of file max_pool.hpp.

4.4.4.3 gradients

```
vector<MatrixXd> Max_Pool::gradients
```

gradients of the layer

Definition at line 54 of file max_pool.hpp.

4.4.4.4 height

```
int Max_Pool::height
```

height of the input

Definition at line 46 of file max_pool.hpp.

4.4.4.5 input

```
vector<MatrixXd> Max_Pool::input
```

input of the layer

Definition at line 52 of file max_pool.hpp.

4.4.4.6 output

```
vector<MatrixXd> Max_Pool::output
```

output of the layer

Definition at line 53 of file max_pool.hpp.

4.4.4.7 stride

```
int Max_Pool::stride = 2
```

stride of the filter

Definition at line 50 of file max_pool.hpp.

4.4.4.8 width

```
int Max_Pool::width
```

width of the input

Definition at line 47 of file max_pool.hpp.

The documentation for this class was generated from the following files:

- /home/mustafa/DLCPP/libdl/layers/[max_pool.hpp](#)
- /home/mustafa/DLCPP/libdl/layers/[max_pool.cpp](#)

4.5 ReLU Class Reference

[ReLU](#) Layer.

```
#include <relu.hpp>
```

Collaboration diagram for ReLU:

Public Member Functions

- [ReLU](#) (int [height](#), int [width](#), int [depth](#))
Create a [ReLU](#).
- void [set_input](#) (vector< MatrixXd > [input](#))
Set input.
- void [clear_output](#) ()
Clear output.
- void [feed_forward](#) (vector< MatrixXd > [input](#))
Forward pass of the [ReLU](#) Layer.
- void [back_propagation](#) (vector< MatrixXd > upstream_gradient)
Backward pass of the [ReLU](#) Layer.

Public Attributes

- int [height](#)
height of the input
- int [width](#)
width of the input
- int [depth](#)
depth of the input
- vector< MatrixXd > [input](#)
input of the layer
- vector< MatrixXd > [output](#)
output of the layer
- vector< MatrixXd > [gradients](#)
gradients of the layer

4.5.1 Detailed Description

[ReLU](#) Layer.

This class is used as [ReLU\(Rectified Linear Unit\)](#) Layer of a neural network. It gets details of its input initially.

Author

Author

Mustafa Erdogan

Version

Revision

0.87

Date

Date

24/07/2019 14:16:20

Contact: mustafa.erdogan@tum.de

Definition at line 33 of file relu.hpp.

4.5.2 Constructor & Destructor Documentation

4.5.2.1 ReLU()

```
ReLU::ReLU (
    int height,
    int width,
    int depth )
```

Create a [ReLU](#).

Parameters

<i>height</i>	height of the input
<i>width</i>	width of the input
<i>depth</i>	depth of the input

Generated by Doxygen

This method creates a [ReLU](#) Layer. Initializes given values.

Definition at line 5 of file relu.cpp.

```
5                                     {
6   this->height = height;
7   this->width  = width;
8   this->depth  = depth;
9 }
```

4.5.3 Member Function Documentation

4.5.3.1 back_propagation()

```
void ReLU::back_propagation (
    vector< MatrixXd > upstream_gradient )
```

Backward pass of the [ReLU](#) Layer.

Parameters

<i>upstream_gradient</i>	gradients coming from the next layer
--------------------------	--------------------------------------

This function iterates backward pass of the [ReLU](#) Layer. As input it gets the next layer's gradients.

Definition at line 26 of file relu.cpp.

```
26                                     {
27   gradients = input;
28   for (int d = 0; d < depth; d++) {
29     gradients[d] =
30       gradients[d].unaryExpr([](double x) { return derivative(x); });
31     gradients[d] =
32       (gradients[d].array() * upstream_gradient[d].array()).matrix();
33   }
34 }
```

Here is the call graph for this function:

4.5.3.2 clear_output()

```
void ReLU::clear_output ( )
```

Clear output.

This method clears output of the [ReLU](#) Layer and fills with zeros.

Definition at line 12 of file relu.cpp.

```
12                                     {
13   output.clear();
14   for (int i = 0; i < this->depth; i++) {
15     output.push_back(MatrixXd::Zero(height, width));
16   }
17 }
```

4.5.3.3 feed_forward()

```
void ReLU::feed_forward (
    vector< MatrixXd > input )
```

Forward pass of the [ReLU](#) Layer.

Parameters

<i>input</i>	input of the ReLU Layer
--------------	---

This function iterates forward pass of the [ReLU](#) Layer. As input it gets the previous layer's output. The output filled is used later from the next layer.

Definition at line 19 of file relu.cpp.

```
19 {
20     this->set_input(input);
21     this->clear_output();
22     for (int d = 0; d < depth; d++)
23         output[d] = input[d].cwiseMax(output[d]);
24 }
```

Here is the call graph for this function:

4.5.3.4 set_input()

```
void ReLU::set_input (
    vector< MatrixXd > input )
```

Set input.

Parameters

<i>input</i>	input of the ReLU Layer
--------------	---

This method sets input of the [ReLU](#) Layer

Definition at line 10 of file relu.cpp.

```
10 { this->input = input; }
```

4.5.4 Member Data Documentation

4.5.4.1 depth

```
int ReLU::depth
```

depth of the input

Definition at line 46 of file relu.hpp.

4.5.4.2 gradients

```
vector<MatrixXd> ReLU::gradients
```

gradients of the layer

Definition at line 50 of file relu.hpp.

4.5.4.3 height

```
int ReLU::height
```

height of the input

Definition at line 44 of file relu.hpp.

4.5.4.4 input

```
vector<MatrixXd> ReLU::input
```

input of the layer

Definition at line 48 of file relu.hpp.

4.5.4.5 output

```
vector<MatrixXd> ReLU::output
```

output of the layer

Definition at line 49 of file relu.hpp.

4.5.4.6 width

```
int ReLU::width
```

width of the input

Definition at line 45 of file relu.hpp.

The documentation for this class was generated from the following files:

- [/home/mustafa/DLCP/ libdl/ layers/relu.hpp](#)
- [/home/mustafa/DLCP/ libdl/ layers/relu.cpp](#)

4.6 Softmax Class Reference

[Softmax](#) Layer.

```
#include <softmax.hpp>
```

Collaboration diagram for Softmax:

Public Member Functions

- [Softmax](#) (int value)
Create a [Softmax](#).
- void [set_input](#) (VectorXd input)
Set input.
- void [feed_forward](#) (VectorXd input)
Forward pass of the [Softmax](#) Layer.
- void [back_propagation](#) (VectorXd upstream_gradient)
Backward pass of the [Softmax](#) Layer.

Public Attributes

- VectorXd [input](#)
input of the layer
- VectorXd [output](#)
output of the layer
- VectorXd [gradients](#)
gradients of the layer

4.6.1 Detailed Description

[Softmax](#) Layer.

This class is used as [Softmax](#) Layer of a neural network.

Author

Author

Mustafa Erdogan

Version

Revision

0.87

Date

Date

24/07/2019 14:16:20

Contact: mustafa.erdogan@tum.de

Definition at line 33 of file softmax.hpp.

4.6.2 Constructor & Destructor Documentation

4.6.2.1 Softmax()

```
Softmax::Softmax (
    int value ) [inline]
```

Create a [Softmax](#).

Parameters

<i>value</i>	This method creates a Softmax Layer.
--------------	--

Definition at line 41 of file softmax.hpp.
 41 {};

4.6.3 Member Function Documentation

4.6.3.1 back_propagation()

```
void Softmax::back_propagation (
    VectorXd upstream_gradient )
```

Backward pass of the [Softmax](#) Layer.

Parameters

<i>upstream_gradient</i>	gradients coming from the next layer
--------------------------	--------------------------------------

This function iterates backward pass of the [Softmax](#) Layer. As input it gets the next layer's gradients.

Definition at line 9 of file softmax.cpp.

```
9 {
10     double sub = upstream_gradient.dot(output);
11     gradients =
12         ((upstream_gradient.array() - sub).array() * output.array()).matrix();
13 }
```

4.6.3.2 feed_forward()

```
void Softmax::feed_forward (
    VectorXd input )
```

Forward pass of the [Softmax](#) Layer.

Parameters

<i>input</i>	input of the Softmax Layer
--------------	--

This function iterates forward pass of the [Softmax](#) Layer. As input it gets the previous layer's output. The output filled is used later from the next layer. Computes probabilities of the labels using softmax function.

Definition at line 4 of file softmax.cpp.

```

4                                     {
5   this->set_input(input);
6   double sum_exp = (input.array() - input.maxCoeff()).exp().sum();
7   output = (input.array() - input.maxCoeff()).exp() / sum_exp;
8 }
```

Here is the call graph for this function:

4.6.3.3 set_input()

```

void Softmax::set_input (
    VectorXd input )
```

Set input.

Parameters

<i>input</i>	input of the Softmax Layer
--------------	--

This method sets input of the [Softmax](#) Layer

Definition at line 2 of file softmax.cpp.

```

2 { this->input = input; }
```

4.6.4 Member Data Documentation

4.6.4.1 gradients

```
VectorXd Softmax::gradients
```

gradients of the layer

Definition at line 45 of file softmax.hpp.

4.6.4.2 input

```
VectorXd Softmax::input
```

input of the layer

Definition at line 41 of file softmax.hpp.

4.6.4.3 output

`VectorXd Softmax::output`

output of the layer

Definition at line 44 of file softmax.hpp.

The documentation for this class was generated from the following files:

- [/home/mustafa/DLCPP/libdl/layers/softmax.hpp](#)
- [/home/mustafa/DLCPP/libdl/layers/softmax.cpp](#)

Chapter 5

File Documentation

5.1 /home/mustafa/DLCP/lbdl/binds.cpp File Reference

```
#include "conv_layer.hpp"
#include "cross_entropy.hpp"
#include "dense_layer.hpp"
#include "max_pool.hpp"
#include "mnist.hpp"
#include "relu.hpp"
#include "softmax.hpp"
#include <algorithm>
#include <eigen3/Eigen/Dense>
#include <fstream>
#include <iostream>
#include <math.h>
#include <pybind11/eigen.h>
#include <pybind11/pybind11.h>
#include <pybind11/stl.h>
#include <stdlib.h>
#include <vector>
```

Include dependency graph for binds.cpp:

5.2 /home/mustafa/DLCP/lbdl/layers/conv_layer.cpp File Reference

```
#include "conv_layer.hpp"
```

Include dependency graph for conv_layer.cpp:

5.3 /home/mustafa/DLCP/lbdl/layers/conv_layer.hpp File Reference

```
#include <algorithm>
#include <eigen3/Eigen/Dense>
#include <fstream>
#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <vector>
```

Include dependency graph for conv_layer.hpp: This graph shows which files directly or indirectly include this file:

Classes

- class [Conv_Layer](#)
Convolutional Layer.

5.4 /home/mustafa/DLCP/layers/cross_entropy.cpp File Reference

```
#include "cross_entropy.hpp"
```

Include dependency graph for cross_entropy.cpp:

5.5 /home/mustafa/DLCP/layers/cross_entropy.hpp File Reference

```
#include <algorithm>
#include <eigen3/Eigen/Dense>
#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <vector>
```

Include dependency graph for cross_entropy.hpp: This graph shows which files directly or indirectly include this file:

Classes

- class [Cross_Entropy](#)
Cross Entropy Layer.

5.6 /home/mustafa/DLCP/layers/dense_layer.cpp File Reference

```
#include "dense_layer.hpp"
```

Include dependency graph for dense_layer.cpp:

5.7 /home/mustafa/DLCP/layers/dense_layer.hpp File Reference

```
#include <algorithm>
#include <eigen3/Eigen/Dense>
#include <fstream>
#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <vector>
```

Include dependency graph for dense_layer.hpp: This graph shows which files directly or indirectly include this file:

Classes

- class [Dense_Layer](#)
Dense Layer.

5.8 /home/mustafa/DLCP/ libdl/ layers/ max_ pool. cpp File Reference

```
#include "max_pool.hpp"
```

Include dependency graph for max_pool.cpp:

5.9 /home/mustafa/DLCP/ libdl/ layers/ max_ pool. hpp File Reference

```
#include <algorithm>
#include <eigen3/Eigen/Dense>
#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <vector>
```

Include dependency graph for max_pool.hpp: This graph shows which files directly or indirectly include this file:

Classes

- class [Max_Pool](#)
Maximum Pooling Layer.

5.10 /home/mustafa/DLCP/ libdl/ layers/ relu. cpp File Reference

```
#include "relu.hpp"
```

Include dependency graph for relu.cpp:

Functions

- double [derivative](#) (double x)

5.10.1 Function Documentation

5.10.1.1 derivative()

```
double derivative (
    double x )
```

Definition at line 3 of file relu.cpp.

```
3 { return x > 0 ? 1 : 0; }
```

5.11 /home/mustafa/DLCP/libdl/layers/relu.hpp File Reference

```
#include <algorithm>
#include <eigen3/Eigen/Dense>
#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <vector>
```

Include dependency graph for relu.hpp: This graph shows which files directly or indirectly include this file:

Classes

- class [ReLU](#)
ReLU Layer.

5.12 /home/mustafa/DLCP/libdl/layers/softmax.cpp File Reference

```
#include "softmax.hpp"
```

Include dependency graph for softmax.cpp:

5.13 /home/mustafa/DLCP/libdl/layers/softmax.hpp File Reference

```
#include <algorithm>
#include <eigen3/Eigen/Dense>
#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <vector>
```

Include dependency graph for softmax.hpp: This graph shows which files directly or indirectly include this file:

Classes

- class [Softmax](#)
Softmax Layer.

5.14 /home/mustafa/DLCP/libdl/main.cpp File Reference

```
#include "conv_layer.hpp"
#include "cross_entropy.hpp"
#include "dense_layer.hpp"
#include "max_pool.hpp"
#include "mnist.hpp"
#include "relu.hpp"
#include "softmax.hpp"
#include <eigen3/Eigen/Dense>
#include <fstream>
#include <iostream>
#include <vector>
```

Include dependency graph for main.cpp:

Functions

- `int main()`

5.14.1 Function Documentation

5.14.1.1 main()

`int main ()`

Definition at line 15 of file main.cpp.

```

15     {
16
17     MNIST *mn = new MNIST("../data");
18
19     vector<vector<MatrixXd>> train_data = mn->train_data;
20     vector<vector<MatrixXd>> validation_data = mn->validation_data;
21     vector<vector<MatrixXd>> test_data = mn->test_data;
22
23     vector<VectorXd> train_labels = mn->train_labels;
24     vector<VectorXd> validation_labels = mn->validation_labels;
25
26     cout << "Data loaded." << endl;
27
28     int TRAIN_DATA_SIZE = train_data.size();
29     int VALIDATION_DATA_SIZE = validation_data.size();
30     int TEST_DATA_SIZE = test_data.size();
31
32     int EPOCHS = 1; // 5;
33     int BATCH_SIZE = 10;
34     int BATCHES = TRAIN_DATA_SIZE / BATCH_SIZE;
35
36     double LEARNING_RATE = 0.05;
37
38     // input: 28x28x1 filter: 5x5x1 stride: 1 output: 24x24x6
39     Conv_Layer conv1(28, 28, 1, 5, 1, 6);
40     // output: 24x24x6
41     ReLU relu1(24, 24, 6);
42     // input: 24x24x6 filter: 2x2x6 stride: 2 output: 12x12x6
43     Max_Pool pool1(24, 24, 6, 2, 2);
44     // input: 12x12x6 filter: 5x5x6 stride: 1 output: 8x8x16
45     Conv_Layer conv2(12, 12, 6, 5, 1, 16);
46     // output: 8x8x16
47     ReLU relu2(8, 8, 16);
48     // input: 8x8x16 filter: 2x2x16 stride: 2 output: 4x4x16
49     Max_Pool pool2(8, 8, 16, 2, 2);
50     // input: 4x4x16 output: 1x10
51     Dense_Layer dense(4, 4, 16, 10);
52     Softmax soft(0);
53     Cross_Entropy entropy(0);
54
55     double cumulative_loss = 0.0;
56
57     cout << "TRAIN DATA SIZE: " << TRAIN_DATA_SIZE << endl;
58     cout << "VALIDATION DATA SIZE: " << VALIDATION_DATA_SIZE << endl;
59     cout << "TEST DATA SIZE: " << TEST_DATA_SIZE << endl;
60     cout << "EPOCHS: " << EPOCHS << endl;
61     cout << "BATCHES: " << BATCHES << endl;
62
63     char selection;
64     cout << "Press \'y\' to use pretrained weights, other chars to train from "
65           << "scratch..."
66           << endl;
67     cin >> selection;
68
69     if (selection == 'y' || selection == 'Y') {
70         conv1.load_filters("../data/conv1.out");
71         conv2.load_filters("../data/conv2.out");
72         dense.load_weights("../data/dense.out");
73         double true_positive = 0.0;
74         // Training accuracy
75         for (int i = 0; i < TRAIN_DATA_SIZE; i++) {
76             if (i % 1000 == 0) {
77                 cout << "Training: " << i << endl;

```

```

78     }
79     // Forward pass
80     conv1.feed_forward(train_data[i]);
81     relu1.feed_forward(conv1.output);
82     pool1.feed_forward(relu1.output);
83     conv2.feed_forward(pool1.output);
84     relu2.feed_forward(conv2.output);
85     pool2.feed_forward(relu2.output);
86     dense.feed_forward(pool2.output);
87     dense.output /= 100;
88     soft.feed_forward(dense.output);
89
90     int actual_index, pred_index;
91     train_labels[i].maxCoeff(&actual_index);
92     soft.output.maxCoeff(&pred_index);
93     if (actual_index == pred_index)
94         true_positive += 1.0;
95 }
96 cout << "true_positive: " << true_positive << endl;
97 cout << "Training accuracy: " << true_positive / TRAIN_DATA_SIZE << endl;
98 // Validation accuracy
99 cumulative_loss = 0.0;
100 true_positive = 0.0;
101 for (int i = 0; i < VALIDATION_DATA_SIZE; i++) {
102     if (i % 1000 == 0) {
103         cout << "Validating: " << i << endl;
104     }
105     // Forward pass
106     conv1.feed_forward(validation_data[i]);
107     relu1.feed_forward(conv1.output);
108     pool1.feed_forward(relu1.output);
109     conv2.feed_forward(pool1.output);
110     relu2.feed_forward(conv2.output);
111     pool2.feed_forward(relu2.output);
112     dense.feed_forward(pool2.output);
113     dense.output /= 100;
114     soft.feed_forward(dense.output);
115     entropy.feed_forward(soft.output, validation_labels[i]);
116     cumulative_loss += entropy.loss;
117
118     int actual_index, pred_index;
119     validation_labels[i].maxCoeff(&actual_index);
120     soft.output.maxCoeff(&pred_index);
121     if (actual_index == pred_index)
122         true_positive += 1.0;
123 }
124 cout << "true_positive: " << true_positive << endl;
125 cout << "Validation set accuracy: " << true_positive / VALIDATION_DATA_SIZE
126     << endl;
127 return 0;
128 }
129
130 for (int epoch = 0; epoch < EPOCHS; epoch++) {
131     for (int b = 0; b < BATCHES; b++) {
132         if (b % 100 == 0) {
133             cout << "epoch: " << epoch << " batch: " << b << endl;
134         }
135         // Select uniform random indices
136         VectorXd batch = VectorXd::Random(BATCH_SIZE) / 2;
137         batch = (batch.array() + 0.5).matrix();
138         batch *= (TRAIN_DATA_SIZE - 1);
139
140         for (int i = 0; i < BATCH_SIZE; i++) {
141             // Forward pass
142             conv1.feed_forward(train_data[batch[i]]);
143             relu1.feed_forward(conv1.output);
144             pool1.feed_forward(relu1.output);
145             conv2.feed_forward(pool1.output);
146             relu2.feed_forward(conv2.output);
147             pool2.feed_forward(relu2.output);
148             dense.feed_forward(pool2.output);
149             dense.output /= 100;
150             soft.feed_forward(dense.output);
151             entropy.feed_forward(soft.output, train_labels[batch[i]]);
152             cumulative_loss += entropy.loss;
153
154             // Backward pass
155             entropy.back_propagation();
156             soft.back_propagation(entropy.gradients);
157             dense.back_propagation(soft.gradients);
158             pool2.back_propagation(dense.gradients);
159             relu2.back_propagation(pool2.gradients);
160             conv2.back_propagation(relu2.gradients);
161             pool1.back_propagation(conv2.gradients);
162             relu1.back_propagation(pool1.gradients);
163             conv1.back_propagation(relu1.gradients);
164         }

```



```

165     // Update params
166     dense.update_weights(BATCH_SIZE, LEARNING_RATE);
167     conv1.update_weights(BATCH_SIZE, LEARNING_RATE);
168     conv2.update_weights(BATCH_SIZE, LEARNING_RATE);
169 }
170 // Training accuracy
171 double true_positive = 0.0;
172 for (int i = 0; i < TRAIN_DATA_SIZE; i++) {
173     if (i % 1000 == 0) {
174         cout << "Training: " << i << endl;
175     }
176     // Forward pass
177     conv1.feed_forward(train_data[i]);
178     relu1.feed_forward(conv1.output);
179     pool1.feed_forward(relu1.output);
180     conv2.feed_forward(pool1.output);
181     relu2.feed_forward(conv2.output);
182     pool2.feed_forward(relu2.output);
183     dense.feed_forward(pool2.output);
184     dense.output /= 100;
185     soft.feed_forward(dense.output);
186
187     int actual_index, pred_index;
188     train_labels[i].maxCoeff(&actual_index);
189     soft.output.maxCoeff(&pred_index);
190     if (actual_index == pred_index)
191         true_positive += 1.0;
192 }
193 cout << "true_positive: " << true_positive << endl;
194 cout << "Training accuracy: " << true_positive / TRAIN_DATA_SIZE << endl;
195
196 // Validation accuracy
197 cumulative_loss = 0.0;
198 true_positive = 0.0;
199 for (int i = 0; i < VALIDATION_DATA_SIZE; i++) {
200     if (i % 1000 == 0) {
201         cout << "Validating: " << i << endl;
202     }
203     // Forward pass
204     conv1.feed_forward(validation_data[i]);
205     relu1.feed_forward(conv1.output);
206     pool1.feed_forward(relu1.output);
207     conv2.feed_forward(pool1.output);
208     relu2.feed_forward(conv2.output);
209     pool2.feed_forward(relu2.output);
210     dense.feed_forward(pool2.output);
211     dense.output /= 100;
212     soft.feed_forward(dense.output);
213     entropy.feed_forward(soft.output, validation_labels[i]);
214     cumulative_loss += entropy.loss;
215
216     int actual_index, pred_index;
217     validation_labels[i].maxCoeff(&actual_index);
218     soft.output.maxCoeff(&pred_index);
219     if (actual_index == pred_index)
220         true_positive += 1.0;
221 }
222 cout << "true_positive: " << true_positive << endl;
223 cout << "Validation accuracy: " << true_positive / VALIDATION_DATA_SIZE
224     << endl;
225 }
226
227 ofstream fout("results.csv");
228 fout << "ImageId,Label" << endl;
229 for (int i = 0; i < TEST_DATA_SIZE; i++) {
230     if (i % 1000 == 0) {
231         cout << "Testing: " << i << endl;
232     }
233     // Forward pass
234     conv1.feed_forward(test_data[i]);
235     relu1.feed_forward(conv1.output);
236     pool1.feed_forward(relu1.output);
237     conv2.feed_forward(pool1.output);
238     relu2.feed_forward(conv2.output);
239     pool2.feed_forward(relu2.output);
240     dense.feed_forward(pool2.output);
241     dense.output /= 100;
242     soft.feed_forward(dense.output);
243
244     int pred_index;
245     soft.output.maxCoeff(&pred_index);
246     fout << to_string(i + 1) << "," << to_string(pred_index) << endl;
247 }
248 fout.close();
249
250 conv1.save_filters("../data/conv1.out");
251 conv2.save_filters("../data/conv2.out");

```

```
252     dense.save_weights("../data/dense.out");  
253 }
```

Here is the call graph for this function:

5.15 /home/mustafa/DLCP/lbdl/README.md File Reference

Index

/home/mustafa/DLCP/libdl/README.md, [52](#)
/home/mustafa/DLCP/libdl/binds.cpp, [45](#)
/home/mustafa/DLCP/libdl/layers/conv_layer.cpp, [45](#)
/home/mustafa/DLCP/libdl/layers/conv_layer.hpp, [45](#)
/home/mustafa/DLCP/libdl/layers/cross_entropy.cpp, [46](#)
/home/mustafa/DLCP/libdl/layers/cross_entropy.hpp, [46](#)
/home/mustafa/DLCP/libdl/layers/dense_layer.cpp, [46](#)
/home/mustafa/DLCP/libdl/layers/dense_layer.hpp, [46](#)
/home/mustafa/DLCP/libdl/layers/max_pool.cpp, [47](#)
/home/mustafa/DLCP/libdl/layers/max_pool.hpp, [47](#)
/home/mustafa/DLCP/libdl/layers/relu.cpp, [47](#)
/home/mustafa/DLCP/libdl/layers/relu.hpp, [48](#)
/home/mustafa/DLCP/libdl/layers/softmax.cpp, [48](#)
/home/mustafa/DLCP/libdl/layers/softmax.hpp, [48](#)
/home/mustafa/DLCP/libdl/main.cpp, [48](#)

accumulated_gradient_biases
 Dense_Layer, [27](#)
accumulated_gradient_filters
 Conv_Layer, [16](#)
accumulated_gradient_weights
 Dense_Layer, [27](#)
accumulated_gradients
 Dense_Layer, [28](#)
actual
 Cross_Entropy, [21](#)

back_propagation
 Conv_Layer, [11](#)
 Cross_Entropy, [20](#)
 Dense_Layer, [24](#)
 Max_Pool, [32](#)
 ReLU, [38](#)
 Softmax, [42](#)

biases
 Dense_Layer, [28](#)

clear_output
 Conv_Layer, [13](#)
 Dense_Layer, [25](#)
 Max_Pool, [33](#)
 ReLU, [38](#)

Conv_Layer, [9](#)
 accumulated_gradient_filters, [16](#)
 back_propagation, [11](#)
 clear_output, [13](#)
 Conv_Layer, [11](#)
 depth, [16](#)

feed_forward, [13](#)
filter_size, [16](#)
filters, [17](#)
gradient_filters, [17](#)
gradients, [17](#)
height, [17](#)
input, [17](#)
load_filters, [14](#)
num_filters, [18](#)
output, [18](#)
save_filters, [15](#)
set_input, [15](#)
stride, [18](#)
update_weights, [16](#)
width, [18](#)

Cross_Entropy, [19](#)
 actual, [21](#)
 back_propagation, [20](#)
 Cross_Entropy, [20](#)
 feed_forward, [20](#)
 gradients, [21](#)
 loss, [21](#)
 predicted, [21](#)

Dense_Layer, [22](#)
 accumulated_gradient_biases, [27](#)
 accumulated_gradient_weights, [27](#)
 accumulated_gradients, [28](#)
 back_propagation, [24](#)
 biases, [28](#)
 clear_output, [25](#)
 Dense_Layer, [23](#)
 depth, [28](#)
 feed_forward, [25](#)
 gradient_biases, [28](#)
 gradient_weights, [28](#)
 gradients, [29](#)
 height, [29](#)
 input, [29](#)
 load_weights, [25](#)
 num_outputs, [29](#)
 output, [29](#)
 save_weights, [26](#)
 set_input, [26](#)
 update_weights, [27](#)
 weights, [30](#)
 width, [30](#)

depth
 Conv_Layer, [16](#)
 Dense_Layer, [28](#)

- Max_Pool, [34](#)
 - ReLU, [39](#)
- derivative
 - relu.cpp, [47](#)
- feed_forward
 - Conv_Layer, [13](#)
 - Cross_Entropy, [20](#)
 - Dense_Layer, [25](#)
 - Max_Pool, [33](#)
 - ReLU, [38](#)
 - Softmax, [42](#)
- filter_size
 - Conv_Layer, [16](#)
 - Max_Pool, [34](#)
- filters
 - Conv_Layer, [17](#)
- gradient_biases
 - Dense_Layer, [28](#)
- gradient_filters
 - Conv_Layer, [17](#)
- gradient_weights
 - Dense_Layer, [28](#)
- gradients
 - Conv_Layer, [17](#)
 - Cross_Entropy, [21](#)
 - Dense_Layer, [29](#)
 - Max_Pool, [34](#)
 - ReLU, [39](#)
 - Softmax, [43](#)
- height
 - Conv_Layer, [17](#)
 - Dense_Layer, [29](#)
 - Max_Pool, [35](#)
 - ReLU, [39](#)
- input
 - Conv_Layer, [17](#)
 - Dense_Layer, [29](#)
 - Max_Pool, [35](#)
 - ReLU, [40](#)
 - Softmax, [43](#)
- load_filters
 - Conv_Layer, [14](#)
- load_weights
 - Dense_Layer, [25](#)
- loss
 - Cross_Entropy, [21](#)
- main
 - main.cpp, [49](#)
- main.cpp
 - main, [49](#)
- Max_Pool, [30](#)
 - back_propagation, [32](#)
 - clear_output, [33](#)
 - depth, [34](#)
 - feed_forward, [33](#)
 - filter_size, [34](#)
 - gradients, [34](#)
 - height, [35](#)
 - input, [35](#)
 - Max_Pool, [32](#)
 - output, [35](#)
 - set_input, [34](#)
 - stride, [35](#)
 - width, [35](#)
- num_filters
 - Conv_Layer, [18](#)
- num_outputs
 - Dense_Layer, [29](#)
- output
 - Conv_Layer, [18](#)
 - Dense_Layer, [29](#)
 - Max_Pool, [35](#)
 - ReLU, [40](#)
 - Softmax, [43](#)
- predicted
 - Cross_Entropy, [21](#)
- ReLU, [36](#)
 - back_propagation, [38](#)
 - clear_output, [38](#)
 - depth, [39](#)
 - feed_forward, [38](#)
 - gradients, [39](#)
 - height, [39](#)
 - input, [40](#)
 - output, [40](#)
 - ReLU, [37](#)
 - set_input, [39](#)
 - width, [40](#)
- relu.cpp
 - derivative, [47](#)
- save_filters
 - Conv_Layer, [15](#)
- save_weights
 - Dense_Layer, [26](#)
- set_input
 - Conv_Layer, [15](#)
 - Dense_Layer, [26](#)
 - Max_Pool, [34](#)
 - ReLU, [39](#)
 - Softmax, [43](#)
- Softmax, [40](#)
 - back_propagation, [42](#)
 - feed_forward, [42](#)
 - gradients, [43](#)
 - input, [43](#)
 - output, [43](#)
 - set_input, [43](#)
 - Softmax, [42](#)

stride

Conv_Layer, [18](#)Max_Pool, [35](#)

update_weights

Conv_Layer, [16](#)Dense_Layer, [27](#)

weights

Dense_Layer, [30](#)

width

Conv_Layer, [18](#)Dense_Layer, [30](#)Max_Pool, [35](#)ReLU, [40](#)