NMDL - Software for the implementation of deep neural networks on the NeuroMatrix® platform

NMDL
User's manual

Module®
JOINT-STOCK COMPANY
RESEARCH CENTRE

Contents

List of Figures

List of Tables

1. General information

1.1. The Purpose of Software

NMDL software module allows you to run a pretrained deep convolutional neural network on computational modules *MC121.01*, *MC127.05*, *NMStick*, *NMCard* and on the *MC127.05* module simulator . The software module consists of 2 parts. One part runs on a personal computer (host) running 64-bit Microsoft® Windows 7/10 or Linux OS, the other part starts and runs on the processor of the computing module. Communication of *MC121.01* and *NMStick* devices with the host is carried out via the USB2.0 channel, for communication of *MC127.05* and *NMCard* modules with the host, the PCIe interface is used.

To work with *NMDL*, you must first install the software to support the computational modules used in the system. No support software installation is required to work with the simulator.

NMDL performs processing of custom source images in accordance with the specified neural network model. Before processing, you need to prepare the model and image data.

The model is preliminarily prepared by a special compiler from *NMDL*. Source models can be in *ONNX* or *DarkNet* format. Not all the operations defined in *ONNX* are supported by the *NMDL* compiler. For a list of supported operations and other restrictions, see "Supported Operations" .

Images must also be pre-processed with a special image converter. Only prepared models and images can be loaded and processed on computational modules.

The library provides a C / C ++ programming interface.

Further in the text *NMDL* (in upper case) - designation of the software package, *nmdl* (in lower case) - files of the program module.

1.2. Processing modes

Input data processing (inference) is performed in accordance with the processing graph defined in the neural network model. Each of the models is processed on a computing device - a unit. Devices and modules based on *NM6407* processor - *MC121.01* and *NMStick* - have one unit. *MC127.05*, *NMCard* modules and other devices based on *NM6408* processor have four units.

On devices with the *NM6408* processor, simultaneous and independent processing of various models is possible. Such processing is illustrated in the figure 1.1 - each unit independently processes its own neural network model.

Figure 1.1. Simultaneous processing of multiple neural networks

Four units can be configured to process the same models, with each unit executing the same processing graph, and batch processing (*"batch mode"*) can be set up to achieve maximum throughput for processing data streams, such as frame streams from video cameras (see picture 1.2).

Batch processing is characterized by a high latency - the time from the start of processing to the receipt of the result.

Figure 1.2. Batch mode

For devices based on *NM6408*, the *NMDL* implements the processing mode of one model on four units with uniform data division (*"multi unit"*) (see figure 1.2). This achieves the minimum delay. Performance here is usually somewhat lower due to the overhead of organizing parallelism. For example, when performing convolutions on separated tensors, it is necessary to compensate for the processing of boundaries, perform repacking, data transit between clusters, etc.

Figure 1.3. Multi unit mode

1.3. Quick start

This section provides step-by-step instructions for getting started quickly with the demo . More complete information about *NMDL* can be found in the corresponding sections of the manual. Here is a demonstration of image processing using a neural network to classify *Squeezenet* on a board with *NM6408* processor simulator.

- Install the *NMDL* distribution as described in Installation section.
- Go to the *bin* directory in the *NMDL* installation directory (by default in Windows *"c: \ Program Files \ Module \ NMDL"*, by default in Linux *"/ opt / nmdl "*) and run the *nmdl_gui* demo program.
- Select the device using the *File - Open Board* ... menu. Make sure the simulator is selected in the dialog box.
- Select the model description using the *File - Open Description...* menu. In the file selection dialog, select the file *PATH_TO_NMDL/nmdl_ref_data/ squeezenet_imagenet/ description08.xml*.
- Select the image to be processed using the *File - Open Picture...* menu. In the file selection dialog, select the *PATH_TO_NMDL/nmdl_ref_data/squeezenet_imagenet/frame.bmp* file.
- Start processing using the *File - Run* menu. As a result of processing, a classification window will pop up with the calculated probabilities in decreasing order. The correct class for the selected image is "lakeside".

1.4. NMDL performance

The tables show the performance values (frames per second FPS) and the delay values from the start of frame processing until the result is obtained (Latency). The size of the processed image is indicated in brackets next to the network name.

Table 1.1. FPS

	MC121.01	**NMStick**	**MC127.05 и NMCard (multi unit mode)**	**MC127.05 и NMCard (batch mode)**
alexnet (227x227)	3,45	3,2	12,6	13
inception v3 (299x299)	0,63	0,6	12,8	20,3
inception v3 (512x512)	0,24	0,23	3,93	5,44
resnet 18 (224x224)	2,28	2,2	25	47
resnet 50 (224x224)	0,8	0,75	12,2	20,6
squeezenet (224x224)	8,3	8	74,4	100
u-net (512x512)*	-	-	2	2
yolo v2 tiny (416x416)	1,16	1,1	21	30,4
yolo v3 (416x416)	0,1	0,09	3,7	4,5
yolo v3 tiny (416x416)	1,44	1,38	27,3	35,3
yolo v5s (640x640)	-	-	3,3	3,4

Table 1.2. Latency (ms)

	MC121.01	**NMStick**	**MC127.05 и NMCard (multi unit mode)**	**MC127.05 и NMCard (batch mode)**
alexnet (227x227)	290	302	79	308
inception v3 (299x299)	1587	1653	78	197
inception v3 (512x512)	4166	4340	254	735
resnet 18 (224x224)	439	457	40	85
resnet 50 (224x224)	1250	1300	82	194
squeezenet (224x224)	120	125	13	40
u-net (512x512)*	-	-	500	2000
yolo v2 tiny (416x416)	862	898	47	132
yolo v3 (416x416)	10000	10416	270	889
yolo v3 tiny (416x416)	694	725	36	113
yolo v5s (640x640)	-	-	303	1176

* The model u-net has replaced the *transposed_convolution* layers with *upsampling* .

2. Software components

Neural network implementation software consists of program modules (API), utilities and manuals.

API files for developing programs using *NMDL*:

- *nmdl.dll/nmdl.so* - software module for inference a neural network. See "Description of nmdl identifiers, functions and structures" .
- *nmdl.lib* - library for early binding of programs with *NMDL* in MSVC ++ environment.
- *nmdl.h* - header file with description of API structures and functions.
- *nmdl_compiler.dll/nmdl_compiler.so* - program module - model compiler *ONNX* / *DarkNet* to internal representation. See "Description of nmdl_compiler identifiers and functions"
- *nmdl_compiler.lib* - ibrary for early binding of the model compiler module in the MSVC ++ environment.
- *nmdl_image_converter.dll/nmdl__image_converter.so* - программный модуль для подготовки обрабатываемых изображений. См. "Описание идентификаторов и функций nmdl_image_converter"
- *nmdl_compiler.h* - header file describing the structures and functions of the model compiler.
- *nmdl_image_converter.dll* / *nmdl__image_converter.so* - a program module for preparing processed images. See "Description of nmdl_image_converter identifiers and functions"
- *nmdl_image_converter.lib* - module for early binding of the MSVC++ image preparation module.
- *nmdl_image_converter.h* - header file describing structures and functions for preparing images.

Header files and early binding libraries are located in the *include* and *lib* directories of the *NMDL* directory.

Utilities:

- *nmdl_compiler_console* - command line utility for compiling models from *ONNX* and *DarkNet* formats into internal format for loading on computational modules. The *ONNX* model file usually has the extension *.onnx* . The model in *DarkNet* format is saved in two files - with the extension *.cfg* and the extension *.weights* . The prepared model for the *MC121.01* and *NMStick* devices has the extension *.nm7* . The model for *MC127.05* and *NMCard* has the extension *.nm8* . See "Compiling a Model" .
- *nmdl_nmdl_image_converter_console* - command line utility for preparing processed images. See Preparing Images .

- *nmdl_gui* - a windowing utility to demonstrate the functionality of the *NMDL*. See "Demo" .

3. Installation

3.1. Windows installation

NMDL distributed as an installation program. Only 64-bit systems are supported.

To install the distribution, run the installer executable file with administrator rights. Follow the instructions of the installation wizard.

To work with program modules, they must be included in the "visibility" of the operating system. One solution is to create an environment variable, for example, named *NMDL* , which records the path to the directory with header and binaries, and adding the created variable to the *PATH* environment variable.

To use the nmdl module in C / C ++ programs, include the #include "nmdl.h" directive in the source file and link the program with the *NMDL* library. If you are using the MSVC++ development environment, include the *nmdl.lib* file for early binding. You can create and use an environment variable *NMDL*to set the path to the files *nmdl.h* and *nmdl.lib* . In the same way, you can connect the *nmdl_compiler* and *nmdl_image_converter* modules.

3.2. Linux installation

It is distributed as a .deb package. Only 64-bit systems of the Debian family are supported.

Use the dpkg package manager to install.

For example:

```
   dpkg -i NMDL.deb
```

3.3. Additional components

Additional components are available with the software module to test and demonstrate *NMDL* .

Components are distributed in archives:

- *nmdl_ref_data_alexnet_imagenet.tar* - archive for demonstration of the *ALEXNET*.
- *nmdl_ref_data_inception_v3_imagenet.tar* -archive for demonstration of the *INCEPTION V3*.
- *nmdl_ref_data_resnet_18_imagenet.tar* -archive for demonstration of the *RESNET18*.
- *nmdl_ref_data_resnet_50_imagenet.tar* -archive for demonstration of the *RESNET50*.
- *nmdl_ref_data_squeezenet_imagenet.tar* - archive for demonstration of the *SQUEEZENET*.
- *nmdl_ref_data_unet_no_transp_conv_covid.tar* - archive for demonstration of the *U-NET*. The model has replaced the *transposed_convolution* layers with *upsampling* .

- *nmdl_ref_yolo2_tiny_pascal_voc.tar* - archive for demonstration of the *YOLONET V2 TINY*.

- *nmdl_ref_yolo3_coco.tar* - archive for demonstration of the *YOLONET V3*.

- *nmdl_ref_yolo3_tiny_coco.tar* - archive for demonstration of the *YOLONET V3 TINY*.

- *nmdl_ref_yolo5s_coco.tar* - archive for demonstration of the *YOLONET V5S*. Works only on modules with NM6408 processor - MC127.05, NMCard, NMMezzo, as well as on the simulator.

The directories contain files:

- *frame.bmp* - test image.

- *model.cfg* - model in *DARKNET* format.

- *model.onnx* - model in *ONNX* format.

- *model.weights* - model weights in *DARKNET* format.

- *description07.xml* - model description file for *MC121.01* and *NMStick* devices to run in the nmdl_gui program.

- *description08.xml* - model description file for modules *MC127.05* , *NMCard* and a simulator to run in the nmdl_gui program.

To prepare demo data that will be processed on the computational module, you need to unpack it into the *nmdl_ref_data* directory, compile models and prepare images as described in "Compile Model" and "Prepare Images" of this manual. For ease of compilation, the archive includes scripts *prepare.cmd* and *prepare.sh*. As a result of the script's work, files will appear in each directory:

- *frame07* - image prepared for processing on *MC121.01* and *NMStick* .

- *frame08* - image prepared for processing on *MC127.05* and *NMCard* .

- *model.nm7* - compiled model for uploading to *MC121.01* and *NMStick* .

- *model.nm8* - compiled model for uploading to *MC127.05* and *NMCard* .

3.4. Preparing the MC121.01 module

When using NMDL with *MC121.01* devices, set the following jumpers before connecting the module to the PC USB connector (see figure 3.1):

- Pin 1-2 of connector X9.

- Pins 3-4 of connector X9.

- Pins 5-6 of connector X9.

- When using USB power, pins 1-2 of connector X11 (when using a power supply, open the contacts).
- Pins 1-2 of connector X18.

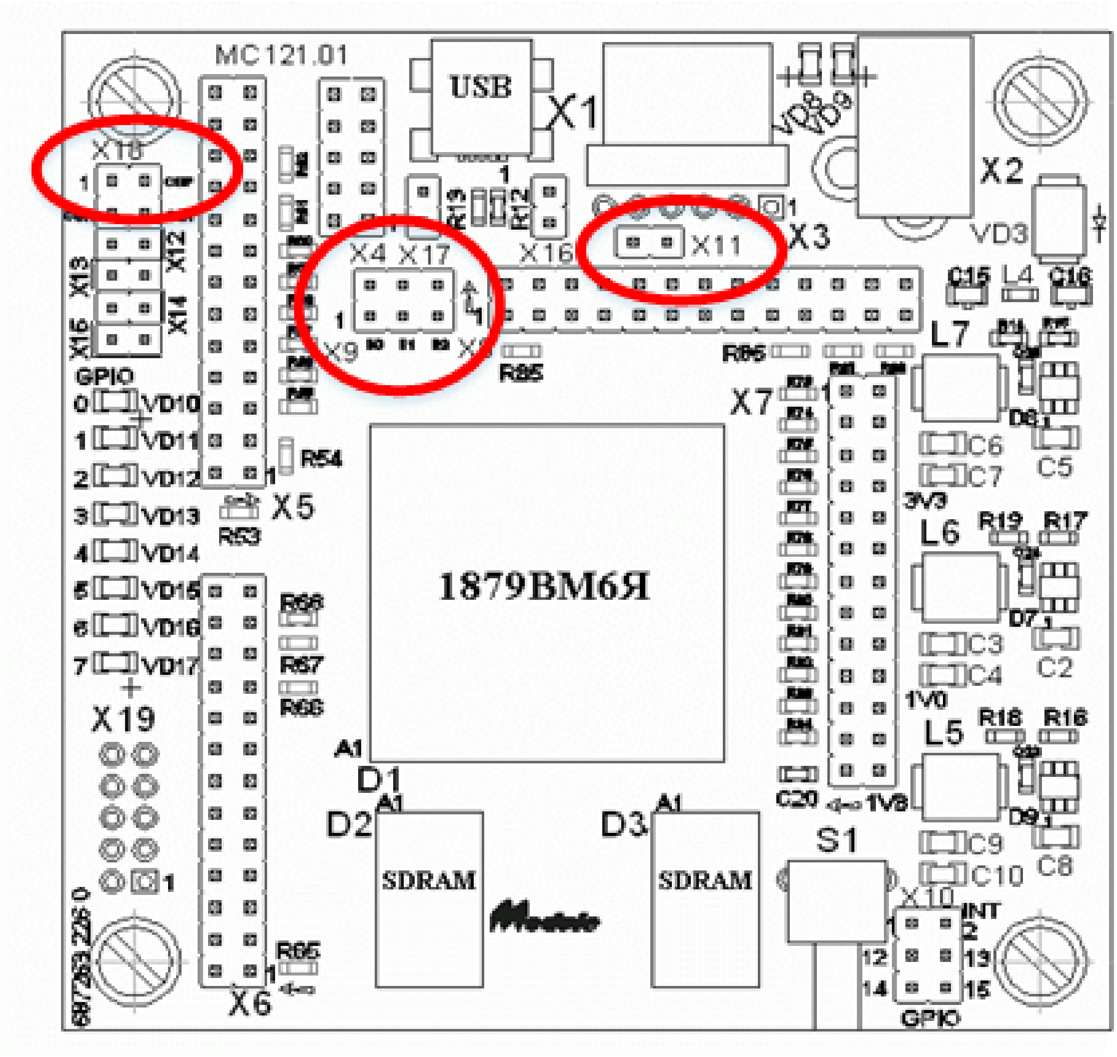


Figure 3.1. Jumpers on the MC121.01 module

3.5. Preparing the NMStick device

When using *NMDL* with *NMStick* , you need to install the device support software (supplied with the product) and connect the device to the USB port of your PC.

3.6. Preparation of MC127.05 and NMCard, NMMezzo

To work with *NMDL*, you need to install the card in a free PCIe slot and install the module support software included in the product delivery set.

4. Compiling the model

The neural network must be pre-trained using a neural network package (e.g. Microsoft® Cognitive Toolkit (CNTK)), and converted to *ONNX* or *DarkNet* .

The original *ONNX* model is usually stored in a file with the Google Protocol Buffer structure and has a **.onnx* extension.

Models for *YOLO* networks can be stored in *DarkNet* format. In this case, the model is described by two files - a file with the **.cfg* extension contains a description of the processing graph, he **.weights* file contains the weights for the convolutions.

The *ONNX* and *DarkNet* model files are the result of training the neural network and at the same time the initial data for the compiler, which, as a result of processing, creates files with *nm7* extension for *MC121.01* and *NMStick* devices, or *nm8* for *MC127.05 modules* , *NMCard* and simulator.

If the input and output tensors in the ONNX model have dynamic dimensions, then to compile the model, it is necessary to make them static.A commit can be done with the following python example. It is assumed here that the dimensions of the input tensor are dynamic. We fix them to the values [1, 3, 640, 640].The output dimensions will be calculated automatically:

```
   #Model fixation script
import onnx
from onnx.tools import update_model_dims
from onnx import helper, shape_inference

model = onnx.load('path/to/the/dynamic_dim_model.onnx')

# update model dimensions
variable_length_model = update_model_dims.update_inputs_outputs_dims(model, \
{'input1': [1, 3, 640, 640]}, {'output1': ['1', 'C','H','W'], \
'output2': ['1', 'C','H','W'], 'output3': ['1', 'C','H','W']})

# out dim will be calculated automatically
inferred_model = shape_inference.infer_shapes(variable_length_model)

# Check the model
onnx.checker.check_model(inferred_model)

# Save the ONNX model
onnx.save(inferred_model, 'static_dim_model.onnx')
```

The *nm7* and *nm8* models are binary images of compiled user models. Their structure is not documented and depends on the target device and compiler version.

The compiler is designed as a dynamically loadable module and can be embedded in a user program. You can also use the console utility nmdl_compiler_console to perform model transformations from the command line.

```
   nmdl_compiler_console BOARD_TYPE NN_TYPE
        SRC_FILENAME DST_FILENAME IS_MULTI_UNIT [WEIGHTS_FILENAME]
```

Command line arguments:

- *BOARD_TYPE* - module type. Valid values: *"MC12101"* - model conversion to nm7 format for *MC121.01* and *NMStick* , *"MC12705"* - converting the model to nm8 format for the *MC127.05*, *NMCard* modules and the simulator.

- *NN_TYPE* - the file format of the input model. Valid values are *"ONNX"* or *"DARKNET"* .

- *SRC_FILENAME* - the filename of the original model.

- *DST_FILENAME* - the filename of the output model.

- *IS_MULTI_UNIT* - the model will be processed in *"multi unit"* mode (see section "Processing modes").

- *WEIGHTS_FILENAME* - the name of file with model weights. Needed only for models in DarkNet format.

An example of compiling a squeezenet model to run on a single unit:

```
>nmdl_compiler_console MC12705 ONNX
     squeezenet.onnx squeezenet.nm8  0
```

An example of compiling a squeezenet model to run on a multi unit mode:

```
>nmdl_compiler_console MC12705 DARKNET
     yolo2t.onnx yolo2t.nm8  1  yolo2t.weights
```

4.1. Supported operations

The compiler supports the following set of operations:

1. Abs

2. Add

3. AveragePool

    - AveragePool2x2, no pad, stride=1

    - AveragePool2x2, no pad, stride=2

    - AveragePool3x3, no pad, stride=2

    - AveragePool3x3, pad, stride=2

4. BatchNormalization (provided that the operation is performed immediately after the convolution).

5. Concat (channels axis or width axis, the number of input tensors must be no more than 7)

6. Convolution

- Conv1x1, stride=1
- Conv1x1, stride=2
- Conv3x3, no pad, all strides
- Conv3x3, pad, stride=1
- Conv3x3, pad, stride=2
- Conv5x5, no pad, all strides
- Conv5x5, pad, stride=1
- Conv7x7, no pad, all strides
- Conv7x7, pad, stride=2
- Conv11x11, no pad, all strides
- Conv7x1, pad_w, stride=1
- Conv1x7, pad_h, stride=1
- Conv3x1, pad_w, stride=1
- Conv1x3, pad_h, stride=1

7. ConvTranspose (kernel 2x2, stride 2x2)
8. Div
9. GEMM
10. GlobalAveragePool
11. Leaky Relu
12. Mat Mul
13. MaxPool

    - MaxPool2x2, no pad, stride=1
    - MaxPool2x2, no pad, stride=2
    - MaxPool3x3, no pad, stride=2
    - MaxPool3x3, pad, stride=2
    - MaxPoolNxN (N - odd value), stride=1*

14. Mul
15. Pad
16. PRelu
17. Relu
18. Reshape
19. Resize* (resize nearest + resize linear, half fixel, scale 2x2)
20. Sigmoid
21. Slice
22. Sub*
23. Transpose
24. Upsample nearest*

* - for MC127.05, NMCard and NMMezzo boards only.

NMDL can work with models with one processed image, that is, one input tensor is processed.

The number of terminal nodes (output tensors) - no more than 5.

5. Preparing images

The processed images must first be converted to a float format (tensor). This can be done using the *nmdl_image_converter_console* utility

```
nmdl_image_converter_console SRC_FILE DST_FILE W H F DR DG DB AR AG AB BT
```

Command line arguments:

- *SRC_FILE* - input file name (*bmp*, *gif*, *jpg*, *emf*, *png*, *tiff*),
- *DST_FILE* - output file name,
- *W* - image tensor width,
- *H* - image tensor height,
- *F* - the order of the color channels in the image tensor (valid values: *rgb* , *rbg* , *grb* , *gbr* , *brg* , *bgr* , intensity - one grayscale channel),
- *DR* - divisor for red pixel channel in the expression *dst = src / D + A* (float value),
- *DG* - divisor for green pixel channel in the expression *dst = src / D + A* (float value),
- *DB* - divisor for blue pixel channel in the expression *dst = src / D + A* (float value),
- *AR* - the term for red pixel channel in the expression *dst = src / D + A* (float value).
- *AG* - the term for green pixel channel in the expression *dst = src / D + A* (float value).
- *AB* - the term for blue pixel channel in the expression *dst = src / D + A* (float value).
- *BT* - the type of the module on which the processing is supposed (valid values: mc12101, mc12705).

For grayscale images, when F is intensity, only the divisor DR and the AR term are used. Other divisors (DG and DB) and terms (AG and AB) are ignored and can be set to any value.

The program scales the input image relative to the center according to the given arguments.

The prepared images are either planar (for *MC121.01* and *NMStick*) or pixelated (for *MC127.05* and *NMCard*) layout format for elements of type *float32* .

In the planar format, the planes of each channel are written to the file in turn. In this case, the buffer can be thought of as a C/C ++ style array:

```
float image[CHANNELS][HEIGHT][WIDTH];
```

That is, the fastest changing index is the index on the width of the image. The number of channels is three (RGB channels), or one for single-channel images (grayscale). In the prepared buffer, the size is aligned to the width of the image. For images with even width, the buffer size will be: _size = width * height * channels_ (float32), for images with odd width: _size = (width + 1) * height * channels_ (float32).

In the pixel format, the channels of the image pixels are written to the file sequentially. A buffer can be thought of as a C / C ++ style array:

```
  float image[HEIGHT][WIDTH][CHANNELS];
```

That is, the fastest changing index is the channel index. The number of channels is three (RGB channels), or one for single-channel images (grayscale). In the prepared buffer, the size is aligned along the image channels to the nearest even value. The size of the buffer with the prepared image for the _MC127.05_ and _NMCard_ modules will be: _size = width * height * (channels + 1)_ (float32).

6. Demo program

To demonstrate the functionality of the NMDL library, the nmdl_gui utility was developed. The tasks of the utility include:

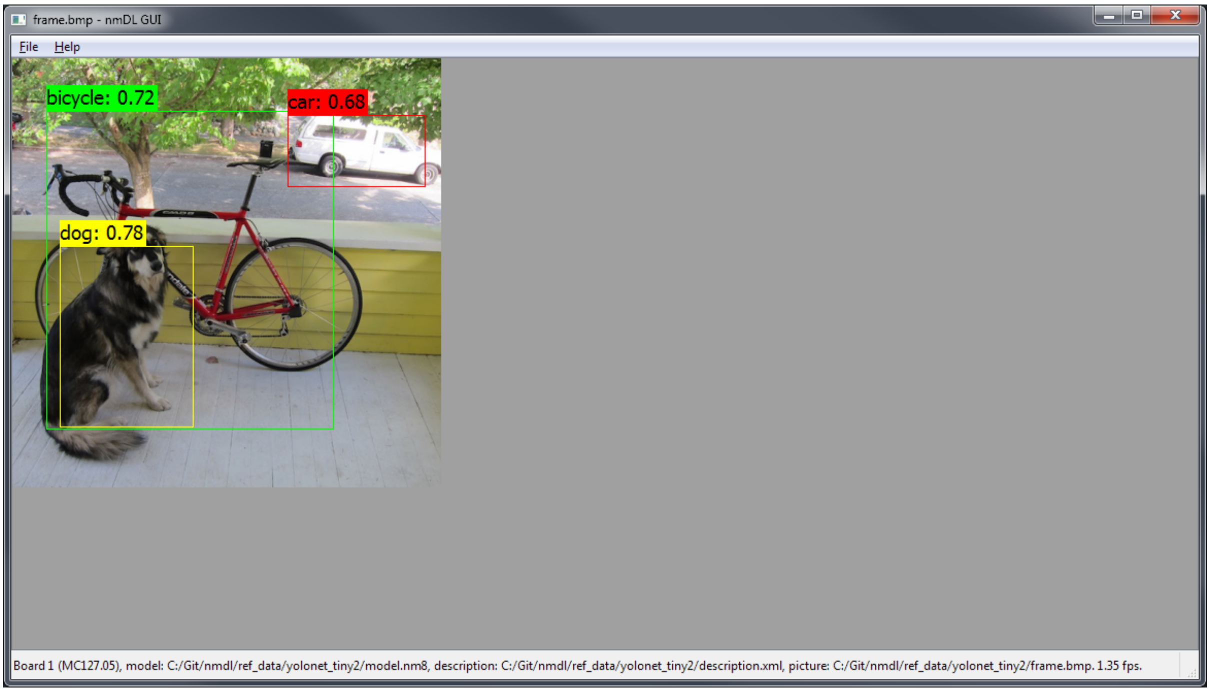
- Library initialization *NMDL*;
- Detection and identification of available accelerators (*simulator* , *MC121.01* , *MC127.05* , *NMStick* , *NMCard*);
- Loading a neural network model file (*.nm7* or *.nm8*);
- Loading the neural network description file (*xml*);
- Pre-processing and loading of images on the computer module;
- Starting processing on the computing module;
- Processing and output of results.

When the program is started, the main program window appears with menu bars, status and client area.

The menu contains controls for the program. The client area contains the image and processing results. The status bar displays the following information:

- Selected accelerator (*simulator* , *MC121.01* , *MC127.05* , *NMStick* , *NMCard*);
- Selected neural network model;
- Selected description of the neural network;
- The selected image;
- Processing speed (frames/s).

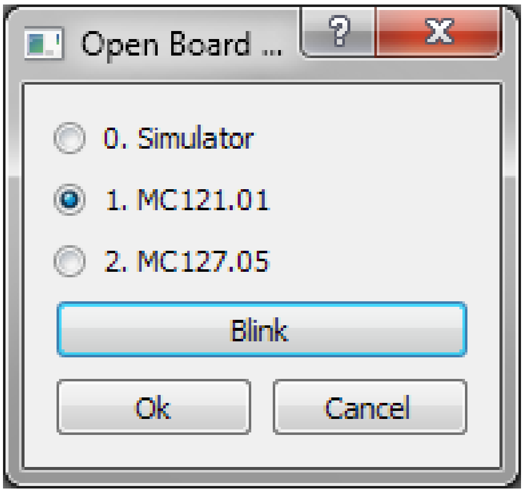
The appearance of the program during operation is shown in the figure 6.1.



Board 1 (MC127.05), model: C:/Git/nmdl/ref_data/yolonet_tiny2/model.nm8, description: C:/Git/nmdl/ref_data/yolonet_tiny2/description.xml, picture: C:/Git/nmdl/ref_data/yolonet_tiny2/frame.bmp. 1.35 fps.

Figure 6.1. The appearance of the program in the process

The choice of the module is carried out in the *Open Board* dialog box, the appearance of which is shown in the figure 6.2. The window contains a list of accelerators available in the system, selection buttons, and the ability to identify the device by means of LED indication.



○ 0. Simulator

● 1. MC121.01

○ 2. MC127.05

Figure 6.2. Module selection window appearance

The selection of the neural network description is carried out in the *Open Description* dialog box. The neural network is described in *XML* format and contains information on the required image format, as well as parameters for interpreting the output parameters.

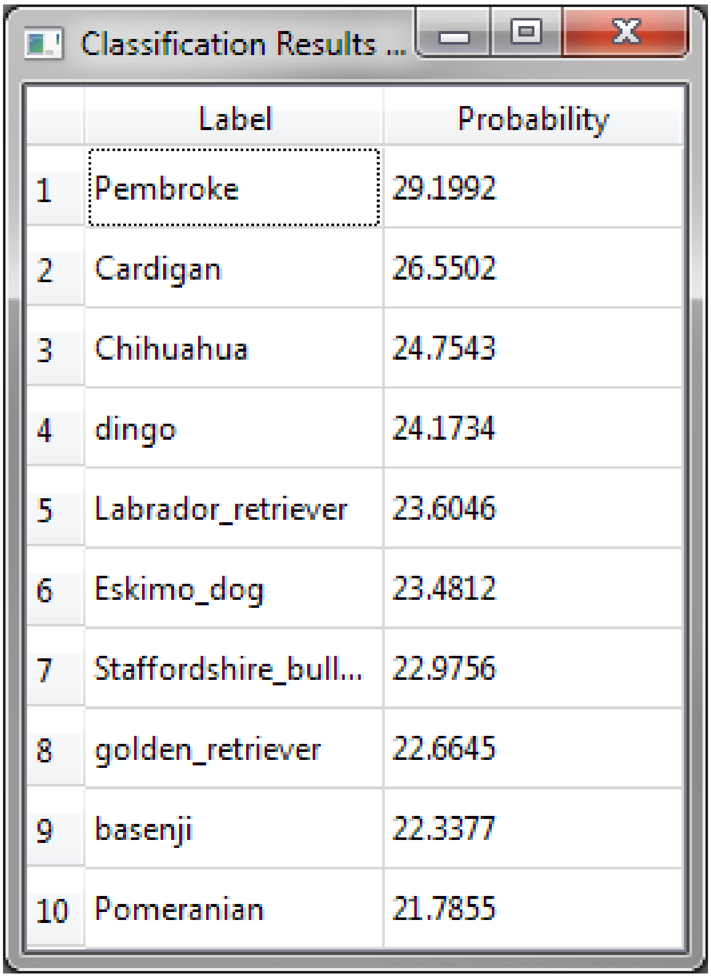
Parameter	**Description**
model	A string with the name of the neural network model file in one of the following formats: • **.nm7* - description of the model for loading on the *MC121.01* and *NMStick* modules. • **.nm8* - description of the model for loading on the *MC127.05* , *NMCard* modules or on the simulator. • **.onnx* or **.onnx* - description of the model in ONNX Protobuf format. The model can be loaded onto any module. Before loading onto a module, the model is pre-converted to **.nm7* or **.nm8* depending on the type of the module. • **.cfg* - description of the model in DARKNET format. The file with weights **.weights* must be located in the same directory and have the same name as the model description file **.cfg* . The model can be loaded onto any module. Before loading onto a module, the model is pre-converted to **.nm7* or **.nm8* depending on the type of the module. You can specify either an absolute path to the file or a path relative to the location of the XML deployment descriptor.

Parameter	**Description**
format	The pixel format to which the pixels in the input image will be converted before processing. Can take values: • rgb • rbg • grb • gbr • brg • bgr This value corresponds to the order of the input tensor channels.
divider	The scaling factor for each color component of a pixel in the input image. Used when converting an image to an input tensor (see "Preparing Images").
adder	Displacement factor for each color component of a pixel in the input image. Used when converting an image to an input tensor (see "Preparing Images").
neural_network	Selected neural network Can take values: • *classifier* - neural network - a classifier, for example *ALEXNET* , *RESNET* , *SQUEEZENET* . • *unet* - neural network of the U-Net family. • *yolo2* - neural network of the YOLO V2 family. • *yolo3* - neural network of the YOLO V3 family. • *yolo5* - neural network of the YOLO V5 family. The result of the work of the classifier is a vector of probabilities that the image belongs to certain classes. In the demo, the classes and probabilities of detection are displayed in a pop-up window. As a result of processing, YOLO networks provide information about several detected objects and their location in the original image. In the demo program, the detected objects are indicated directly on the original image in the enclosing rectangles.
yolo_anchors	Parameters for initial initialization of detected rectangles (only for *YOLO2* and *YOLO3* networks).
yolo_confidence_threshold	Threshold for displaying detection results (only for *YOLO2* and *YOLO3* networks).

Parameter	Description
yolo_iou_threshold	Overlapping detection rectangle threshold (only for *YOLO2* and *YOLO3* networks)
labels	List of strings with class names that will be displayed in the processing result window.

The *Open Picture* dialog box allows you to open one or more images for processing.

When all the data is available, the *Run* button becomes available and starts processing. The launch can be initiated using the *"Ctrl + R"* combination. Pre-processing is carried out for each selected image in accordance with the given description of the neural network for further processing on the accelerator. As a result of processing, the output structure with N tensors is filled. Depending on the neural network, the output structure is interpreted in different ways. For classification networks (such as *SqueezeNet*), an additional window with classes and probability appears. The appearance of this window is shown in the figure 6.3.



	Label	Probability
1	Pembroke	29.1992
2	Cardigan	26.5502
3	Chihuahua	24.7543
4	dingo	24.1734
5	Labrador_retriever	23.6046
6	Eskimo_dog	23.4812
7	Staffordshire_bull...	22.9756
8	golden_retriever	22.6645
9	basenji	22.3377
10	Pomeranian	21.7855

Figure 6.3. Classification results window appearance

For networks that perform object detection (such as *YOLO*), the result is shown directly in the original image. After the accelerator is finished, the rectangles are superimposed on the detected objects, the class name and the detection probability are given. An example of overlaying the result on an image is shown in the figure. 6.4.

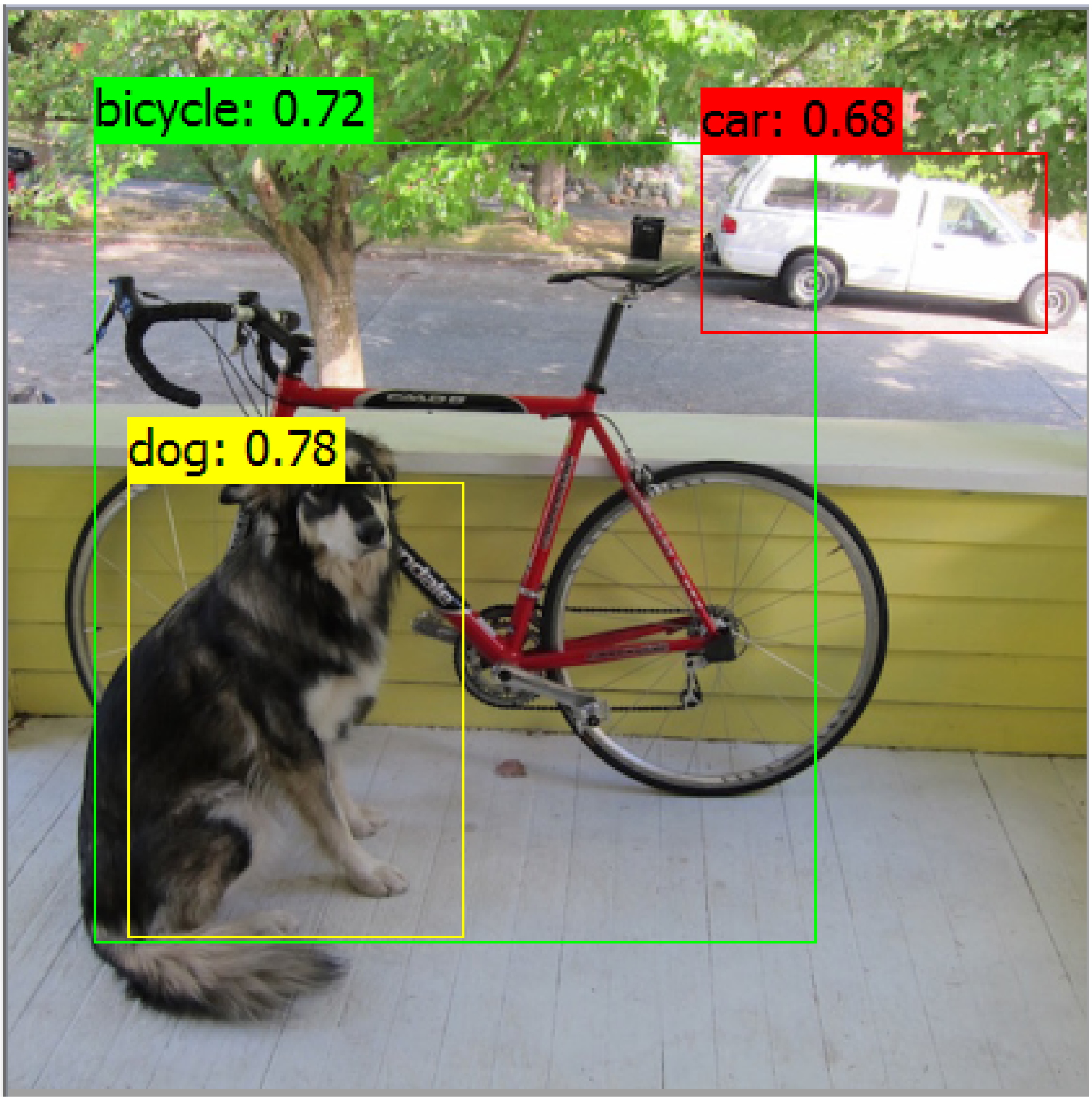


Figure 6.4. Displaying detection results

The program allows you to process a sequence of images. To do this, select several files. After that, when starting processing (*Run*), one image will be processed. When restarting - the second, etc. You can also start processing in automatic mode (*Run Auto*). Press the *spacebar* key to stop automatic processing.

The status bar is used to display additional information. The line displays the name of the selected device, the neural network description file name, the image filename, as well as the fps value measured by the program, taking into account the frame transfer and the result obtained. The processing speed without taking into account the transfer time is displayed in brackets.

7. An example of using NMDL

The example demonstrates the use of the *NMDL* library, software modules for model compilation and image preparation. The example consists of a source file *example1.cpp* and a build script *CMakeLists.txt* in the *"examples/example1"* directory of the installation directory.

It is assumed that the source data for processing is in the *"nmdl_ref_data/squeezenet"* folder in the installation directory, this means that the example demonstrates the processing of the neural network *squeezenet* . The initial data are:

- Neural network model in ONNX format - file *"nmdl_ref_data/squeezenet/model.onnx"*
- Processed image in BMP format - file *"nmdl_ref_data/squeezenet/frame.bmp"*

The example shows the calls to the functions of the libraries in the order necessary for correct operation.

```
#include <array>
#include <fstream>
#include <iostream>
#include <string>
#include <unordered_map>
#include <vector>
#include "nmdl.h"
#include "nmdl_compiler.h"
#include "nmdl_image_converter.h"

//#define _USE_DARKNET_

namespace {

auto Call(NMDL_COMPILER_RESULT result, const std::string &function_name) {
    static std::unordered_map<NMDL_COMPILER_RESULT, std::string> map = {
        {NMDL_COMPILER_RESULT_OK,                      "OK"},
        {NMDL_COMPILER_RESULT_MEMORY_ALLOCATION_ERROR, "MEMORY_ALLOCATION_ERROR"},
        {NMDL_COMPILER_RESULT_MODEL_LOADING_ERROR,     "MODEL_LOADING_ERROR"},
        {NMDL_COMPILER_RESULT_INVALID_PARAMETER,       "INVALID_PARAMETER"},
        {NMDL_COMPILER_RESULT_INVALID_MODEL,           "INVALID_MODEL"},
        {NMDL_COMPILER_RESULT_UNSUPPORTED_OPERATION,   "UNSUPPORTED_OPERATION"}
    };
    if(result != NMDL_COMPILER_RESULT_OK) {
        throw std::runtime_error(function_name + ": " + map[result] + ": " +
            NMDL_COMPILER_GetLastError());
    }
    return NMDL_RESULT_OK;
}

auto Call(NMDL_RESULT result, const std::string &function_name) {
    static std::unordered_map<NMDL_RESULT, std::string> map = {
        {NMDL_RESULT_OK,                            "OK"},
        {NMDL_RESULT_INVALID_FUNC_PARAMETER,        "INVALID_FUNC_PARAMETER"},
        {NMDL_RESULT_NO_LOAD_LIBRARY,               "NO_LOAD_LIBRARY"},
        {NMDL_RESULT_NO_BOARD,                      "NO_BOARD"},
        {NMDL_RESULT_BOARD_RESET_ERROR,             "BOARD_RESET_ERROR"},
        {NMDL_RESULT_INIT_CODE_LOADING_ERROR,       "INIT_CODE_LOADING_ERROR"},
        {NMDL_RESULT_CORE_HANDLE_RETRIEVAL_ERROR,   "CORE_HANDLE_RETRIEVAL_ERROR"},
        {NMDL_RESULT_FILE_LOADING_ERROR,            "FILE_LOADING_ERROR"},
        {NMDL_RESULT_MEMORY_WRITE_ERROR,            "MEMORY_WRITE_ERROR"},
        {NMDL_RESULT_MEMORY_READ_ERROR,             "MEMORY_READ_ERROR"},
        {NMDL_RESULT_MEMORY_ALLOCATION_ERROR,       "MEMORY_ALLOCATION_ERROR"},
        {NMDL_RESULT_MODEL_LOADING_ERROR,           "MODEL_LOADING_ERROR"},
        {NMDL_RESULT_INVALID_MODEL,                 "INVALID_MODEL"},
        {NMDL_RESULT_BOARD_SYNC_ERROR,              "BOARD_SYNC_ERROR"},
        {NMDL_RESULT_BOARD_MEMORY_ALLOCATION_ERROR, "BOARD_MEMORY_ALLOCATION_ERROR"},
        {NMDL_RESULT_NN_CREATION_ERROR,             "NN_CREATION_ERROR"},
        {NMDL_RESULT_NN_LOADING_ERROR,              "NN_LOADING_ERROR"},
        {NMDL_RESULT_NN_INFO_RETRIEVAL_ERROR,       "NN_INFO_RETRIEVAL_ERROR"},
        {NMDL_RESULT_MODEL_IS_TOO_BIG,              "MODEL_IS_TOO_BIG"},
        {NMDL_RESULT_NOT_INITIALIZED,               "NOT_INITIALIZED"},
        {NMDL_RESULT_BUSY,                          "BUSY"},
        {NMDL_RESULT_UNKNOWN_ERROR,                 "UNKNOWN_ERROR"}
    };
    if(result != NMDL_RESULT_OK) {
        throw std::runtime_error(function_name + ": " + map[result]);
    }
    return NMDL_RESULT_OK;
}

template <typename T>
auto ReadFile(const std::string &filename) {
    std::ifstream ifs(filename, std::ios::binary | std::ios::ate);
    if(!ifs.is_open()) {
        throw std::runtime_error("Unable to open input file: " + filename);
    }
    auto fsize = static_cast<std::size_t>(ifs.tellg());
    ifs.seekg(0);
    std::vector<T> data(fsize / sizeof(T));
    ifs.read(reinterpret_cast<char*>(data.data()), data.size() * sizeof(T));
    return data;
}

void ShowNMDLVersion() {
    std::uint32_t major = 0;
    std::uint32_t minor = 0;
    std::uint32_t patch = 0;
    Call(NMDL_GetLibVersion(&major, &minor, &patch), "GetLibVersion");
    std::cout << "Lib version: " << major << "." << minor
            << "." << patch << std::endl;
}

void CheckBoard(std::uint32_t required_board_type) {
    std::uint32_t boards;
    std::uint32_t board_number = -1;
    Call(NMDL_GetBoardCount(required_board_type, &boards), "GetBoardCount");
    std::cout << "Detected boards: " << boards << std::endl;
    if(!boards) {
        throw std::runtime_error("Board not found");
    }
}

#ifdef _USE_DARKNET_
auto CompileModel(const std::string &config_filename,
        const std::string &weights_filename,
        std::uint32_t board_type,
        bool is_multi_unit) {
    float *nm_model = nullptr;
    std::uint32_t nm_model_floats = 0u;
    auto config = ReadFile<char>(config_filename);
    auto weights = ReadFile<char>(weights_filename);
    Call(NMDL_COMPILER_CompileDarkNet(is_multi_unit, board_type,
            config.data(), config.size(), weights.data(), weights.size(),
            &nm_model, &nm_model_floats), "CompileONNX");
    std::vector<float> result(nm_model, nm_model + nm_model_floats);
    NMDL_COMPILER_FreeModel(board_type, nm_model);
    return result;
}
#else
auto CompileModel(const std::string &model_filename, std::uint32_t board_type,
        bool is_multi_unit) {
    float *nm_model = nullptr;
    std::uint32_t nm_model_floats = 0u;
    auto model = ReadFile<char>(model_filename);
    Call(NMDL_COMPILER_CompileONNX(is_multi_unit, board_type, model.data(),
            model.size(), &nm_model, &nm_model_floats), "CompileONNX");
    std::vector<float> result(nm_model, nm_model + nm_model_floats);
    NMDL_COMPILER_FreeModel(board_type, nm_model);
    return result;
}
#endif

auto GetModelInformation(NMDL_HANDLE nmdl, std::uint32_t unit_num) {
    NMDL_ModelInfo model_info;
    Call(NMDL_GetModelInfo(nmdl, unit_num, &model_info), "GetModelInfo");
    std::cout << "Input tensor number: " << model_info.input_tensor_num << std::endl;
    for(std::size_t i = 0; i < model_info.input_tensor_num; ++i) {
        std::cout << "Input tensor " << i << ": " <<
            model_info.input_tensors[i].width << ", " <<
            model_info.input_tensors[i].height << ", " <<
            model_info.input_tensors[i].depth <<
            std::endl;
    }
    std::cout << "Output tensor number: " << model_info.output_tensor_num << std::endl;
    for(std::size_t i = 0; i < model_info.output_tensor_num; ++i) {
        std::cout << "Output tensor " << i << ": " <<
            model_info.output_tensors[i].width << ", " <<
            model_info.output_tensors[i].height << ", " <<
            model_info.output_tensors[i].depth <<
            std::endl;
    }
    return model_info;
}

auto PrepareInput(const std::string &filename, std::uint32_t width,
        std::uint32_t height, std::uint32_t board_type,
        std::uint32_t color_format, const float rgb_divider[3],
        const float rgb_adder[3]) {
    auto bmp_frame = ReadFile<char>(filename);
    std::vector<float> input(NMDL_IMAGE_CONVERTER_RequiredSize(
            width, height, color_format, board_type));
    if(NMDL_IMAGE_CONVERTER_Convert(bmp_frame.data(), input.data(), bmp_frame.size(),
            width, height, color_format, rgb_divider, rgb_adder, board_type)) {
        throw std::runtime_error("Image conversion error");
    }
    return input;
}

void WaitForOutput(NMDL_HANDLE nmdl, std::uint32_t unit_num, float *outputs[]) {
    std::uint32_t status = NMDL_PROCESS_FRAME_STATUS_INCOMPLETE;
    while(status == NMDL_PROCESS_FRAME_STATUS_INCOMPLETE) {
        NMDL_GetStatus(nmdl, unit_num, &status);
    };
    double fps;

    Call(NMDL_GetOutput(nmdl, unit_num, outputs, &fps), "GetOutput");
    std::cout << "First four result values:" << std::endl;
    for(std::size_t i = 0; i < 4; ++i) {
        std::cout << outputs[0][i] << std::endl;
    }
    std::cout << "FPS:" << fps << std::endl;
}

}

int main() {
    const std::uint32_t BOARD_TYPE = NMDL_BOARD_TYPE_SIMULATOR;
    //const uint32_t BOARD_TYPE = NMDL_BOARD_TYPE_MC12705;
    //const uint32_t BOARD_TYPE = NMDL_BOARD_TYPE_MC12101;
    const std::uint32_t COMPILER_BOARD_TYPE =
        BOARD_TYPE == NMDL_BOARD_TYPE_MC12101 ?
        NMDL_COMPILER_BOARD_TYPE_MC12101 :
        NMDL_COMPILER_BOARD_TYPE_MC12705;
    const std::uint32_t IMAGE_CONVERTER_BOARD_TYPE =
        BOARD_TYPE == NMDL_BOARD_TYPE_MC12101 ?
        NMDL_IMAGE_CONVERTER_BOARD_TYPE_MC12101 :
        NMDL_IMAGE_CONVERTER_BOARD_TYPE_MC12705;
#ifdef _USE_DARKNET_
```

```
    const std::string DARKNET_CONFIG_FILENAME =
            "../nmdl_ref_data/yolo_v3_tiny_coco/model.cfg";
    const std::string DARKNET_WEIGHTS_FILENAME =
            "../nmdl_ref_data/yolo_v3_tiny_coco/model.weights";
    const std::string BMP_FRAME_FILENAME =
            "../nmdl_ref_data/yolo_v3_tiny_coco/frame.bmp";
    const NMDL_IMAGE_CONVERTER_COLOR_FORMAT IMAGE_CONVERTER_COLOR_FORMAT =
        NMDL_IMAGE_CONVERTER_COLOR_FORMAT_RGB;
    const float NM_FRAME_RGB_DIVIDER[3] = {255.0f, 255.0f, 255.0f};
    const float NM_FRAME_RGB_ADDER[3] = {0.0f, 0.0f, 0.0f};
#else
    const std::string ONNX_MODEL_FILENAME =
            "../nmdl_ref_data/squeezenet_imagenet/model.onnx";
    const std::string BMP_FRAME_FILENAME =
            "../nmdl_ref_data/squeezenet_imagenet/frame.bmp";
    const NMDL_IMAGE_CONVERTER_COLOR_FORMAT IMAGE_CONVERTER_COLOR_FORMAT =
        NMDL_IMAGE_CONVERTER_COLOR_FORMAT_BGR;
    const float NM_FRAME_RGB_DIVIDER[3] = {1.0f, 1.0f, 1.0f};
    const float NM_FRAME_RGB_ADDER[3] = {0.0f, 0.0f, 0.0f};
#endif
    const std::size_t BATCHES = 4;
    const std::size_t FRAMES = 5;

    NMDL_HANDLE nmdl = 0;

    try {
        std::cout << "Query library version..." << std::endl;
        ShowNMDLVersion();

        std::cout << "Board detection... " << std::endl;
        CheckBoard(BOARD_TYPE);

        std::cout << "NMDL initialization... " << std::endl;
        Call(NMDL_Create(&nmdl), "Create");

        std::cout << "Use multi unit... " << std::endl;

        std::cout << "Compile model... " << std::endl;
#ifdef _USE_DARKNET_
        auto model = CompileModel(DARKNET_CONFIG_FILENAME, DARKNET_WEIGHTS_FILENAME,
                COMPILER_BOARD_TYPE, true);
#else
        auto model = CompileModel(ONNX_MODEL_FILENAME, COMPILER_BOARD_TYPE, true);
#endif

        std::array<const float*, NMDL_MAX_UNITS> models = {model.data()};
        std::array<std::uint32_t, NMDL_MAX_UNITS> model_floats =
                {static_cast<std::uint32_t>(model.size())};
        Call(NMDL_Initialize(nmdl, BOARD_TYPE, 0, 0, models.data(),
                model_floats.data()), "Initialize");

        std::cout << "Get model information... " << std::endl;
        auto model_info = GetModelInformation(nmdl, 0);

        std::cout << "Prepare inputs... " << std::endl;
        auto input = PrepareInput(BMP_FRAME_FILENAME, model_info.input_tensors[0].width,
                model_info.input_tensors[0].height, IMAGE_CONVERTER_BOARD_TYPE,
                IMAGE_CONVERTER_COLOR_FORMAT, NM_FRAME_RGB_DIVIDER, NM_FRAME_RGB_ADDER);
        std::array<const float*, 1> inputs = {input.data()};

        std::cout << "Reserve outputs... " << std::endl;
        std::vector<std::vector<float>> output_tensors(model_info.output_tensor_num);
        std::vector<float*> outputs(model_info.output_tensor_num);
        for(std::size_t i = 0; i < model_info.output_tensor_num; ++i) {
            output_tensors[i].resize(static_cast<std::size_t>(
                    model_info.output_tensors[i].width) *
                    model_info.output_tensors[i].height *
                    model_info.output_tensors[i].depth);
            outputs[i] = output_tensors[i].data();
        }

261         std::cout << "Process inputs... " << std::endl;
262         for(std::size_t i = 0; i < FRAMES; ++i) {
263             Call(NMDL_Process(nmdl, 0, inputs.data()), "Process");
264             WaitForOutput(nmdl, 0, outputs.data());
265         }
266         NMDL_Release(nmdl);
267
268         std::cout << "Process batch... " << std::endl;
269
270         std::cout << "Compile model... " << std::endl;
271 #ifdef _USE_DARKNET_
272         model = CompileModel(DARKNET_CONFIG_FILENAME, DARKNET_WEIGHTS_FILENAME,
273                 COMPILER_BOARD_TYPE, false);
274 #else
275         model = CompileModel(ONNX_MODEL_FILENAME, COMPILER_BOARD_TYPE, false);
276 #endif
277         models = {model.data(), model.data(), model.data(), model.data()};
278         model_floats = {
279                 static_cast<std::uint32_t>(model.size()),
280                 static_cast<std::uint32_t>(model.size()),
281                 static_cast<std::uint32_t>(model.size()),
282                 static_cast<std::uint32_t>(model.size())};
283         Call(NMDL_Initialize(nmdl, BOARD_TYPE, 0, 0, models.data(),
284                 model_floats.data()), "Initialize");
285
286         std::uint32_t cnt_in = 0;
287         std::uint32_t cnt_out = 0;
288         for(auto i = 0u; i < BATCHES; ++i) {
289             Call(NMDL_Process(nmdl, (cnt_in++) % BATCHES, inputs.data()),
290                     "ProcessFrame");
291         }
292         for(auto i = BATCHES; i < FRAMES; ++i) {
293             WaitForOutput(nmdl, (cnt_out++) % BATCHES, outputs.data());
294             Call(NMDL_Process(nmdl, (cnt_in++) % BATCHES, inputs.data()),
295                     "ProcessFrame");
296         }
297         for(auto i = 0u; i < BATCHES; ++i) {
298             WaitForOutput(nmdl, (cnt_out++) % BATCHES, outputs.data());
299         }
300     }
301     catch (std::exception& e) {
302         std::cerr << e.what() << std::endl;
303     }
304     NMDL_Release(nmdl);
305     NMDL_Destroy(nmdl);
306
307     return 0;
308 }
```

The following steps are performed here:

1 .. 9: Connecting header files. *"nmdl.h"* - description of the neural network processing library, *"nmdl_compiler.h"* - description of the library for compiling models, *"nmdl_image_converter.h"* is a description of the library for preparing images.

15 .. 29: A wrapper function for calls to model compilation library functions. In case of an error, it generates an error message and raises an exception.

31 .. 60: A wrapper function for calls to functions of the neural network processing library. In case of an error, it generates an error message and raises an exception.

62 .. 73: Generic function for reading data from a file into a vector.

75 .. 82: Function for outputting the NMDL version NMDL_GetLibVersion.

84 .. *92*: Function for checking the presence of a module of a given type. In fact, a request is made for the number of detected modules of a given type - NMDL_GetBoardCount .

94 .. *122*: Source model compilation function. The function is called NMDL_COMPILER_CompileONNX . This is where memory is allocated inside the compilation function, so after work the allocated memory is freed by calling NMDL_COMPILER_FreeModel , and the compilation result is copied into the returned float vector. In the target program, you can pre-compile the model using the *nmdl_compiler_console* utility like this, as described in "Compiling the Model" .

124 .. *144*: Function for receiving and displaying information about the parameters of the input/ output tensors. The call is NMDL_GetModelInfo .

146 .. *158*: Frame preparation function. This is where you read an image from a file, decode it and preprocess it. (NMDL_IMAGE_CONVERTER_Convert). For a description of preprocessing, see Preparing Images .

160 .. *173*: Frame processing wait function. Takes the number of the cluster on which processing is performed (*batch_num*) and vector buffer for storing the processing result. The function is blocked in the processing status request cycle (NMDL_GetStatus). After receiving the *NMDL_PROCESS_FRAME_STATUS_FREE* status, the result is copied by calling NMDL_GetOutput..

189 .. *209*: File names with the original model and image are specified. Parameters for preparing the image are set. For the original model, you need to prepare an image with pixels in the blue-green-red format. Each pixel is modified with the formula Y = X / 1.0 - 114. This transformation is required to process the *squeezenet* model. For other models, it is necessary to use the corresponding parameters that are determined during model creation and are the initial data bound to a specific model. For more information about preparing images, see "Preparing Images" .

210: Sets the number of clusters for batch processing.

211: The number of frames to be processed is set. In the example, one frame is processed multiple times.

Further, in the main function, the described auxiliary functions are sequentially called.

223: Initializing NMDL. By calling the NMDL_Create function, the internal structures of the NMDL library are initialized.

225 .. *266*: An example of sequential processing of frames in the mode of dividing data into clusters *"multi unit"* (see the section "Processing modes").

229 .. *232*: model compilation.

235 .. *239*: initialization. The NMDL_Initialize function loads the compiled model into the selected accelerator.

245: input tensor formation.

250 .. *259*: reserving buffers for output tensors.

262 .. *265*: processing and getting results.

268 .. *299*: An example of sequential processing of frames in batch processing mode (see the section *"batch mode"* "Processing modes").

304, 305: Completion releases the allocated resources (NMDL_Release and NMDL_Destroy).

8. Description of identifiers, functions and structures of NMDL

8.1. Identifiers and structures

8.1.1. NMDL_BOARD_TYPE

Module types.

```
  typedef enum tagNMDL_BOARD_TYPE {
    NMDL_BOARD_TYPE_SIMULATOR,
    NMDL_BOARD_TYPE_MC12101,
    NMDL_BOARD_TYPE_MC12705,
    NMDL_BOARD_TYPE_NMSTICK,
    NMDL_BOARD_TYPE_NMCARD,
    NMDL_BOARD_TYPE_NMMEZZO
} NMDL_BOARD_TYPE;
```

- *NMDL_BOARD_TYPE_SIMULATOR* - *MC127.05* simulator.
- *NMDL_BOARD_TYPE_MC12101* - *MC121.01* - board with NM6407 processor.
- *NMDL_BOARD_TYPE_MC12705* - *MC127.05*. Is a server board with NM6408 processor connected to standard PCI-E ports on the motherboard.
- *NMDL_BOARD_TYPE_NMSTICK* - *NMStick* - special board with NM6407 processor in USB Flash drive form factor.
- *NMDL_BOARD_TYPE_NMCARD* - *NMCard*. It is a special board with NM6408 processor connected via the PCIe slot to the PC motherboard.
- *NMDL_BOARD_TYPE_NMMEZZO* - *NMMezzo*. It is a special board with NM6408 processor connected via the PCIe bus to the user's carrier board.

8.1.2. NMDL_ModelInfo

Structure with information about the model.

```
  typedef struct tagNMDL_ModelInfo {
    unsigned int input_tensor_num;
    NMDL_Tensor input_tensors[NMDL_MAX_INPUT_TENSORS];
    unsigned int output_tensor_num;
    NMDL_Tensor output_tensors[NMDL_MAX_OUTPUT_TENSORS];
} NMDL_ModelInfo;
```

- *output_tensor_num* -number of input tensors,
- *output_tensors* – input tensors.
- *output_tensor_num* -number of output tensors,

- *output_tensors* – output tensors.

NMDL_MAX_INPUT_TENSORS - maximum number of input tensors.

NMDL_MAX_OUTPUT_TENSORS - maximum number of output tensors.

8.1.3. NMDL_PROCESS_FRAME_STATUS

Frame processing status identifier.

```
  typedef enum tagNMDL_PROCESS_FRAME_STATUS {
    NMDL_PROCESS_FRAME_STATUS_FREE,
    NMDL_PROCESS_FRAME_STATUS_INCOMPLETE
} NMDL_PROCESS_FRAME_STATUS;
```

- *NMDL_PROCESS_FRAME_STATUS_FREE* - the frame is not processed,
- *NMDL_PROCESS_FRAME_STATUS_INCOMPLETE* - the frame is being processed.

8.1.4. NMDL_RESULT

Returned result.

```
  typedef enum tagNMDL_RESULT {
    NMDL_RESULT_OK,
    NMDL_RESULT_INVALID_FUNC_PARAMETER,
    NMDL_RESULT_NO_BOARD,
    NMDL_RESULT_BOARD_RESET_ERROR,
    NMDL_RESULT_INIT_CODE_LOADING_ERROR,
    NMDL_RESULT_CORE_HANDLE_RETRIEVAL_ERROR,
    NMDL_RESULT_FILE_LOADING_ERROR,
    NMDL_RESULT_MEMORY_WRITE_ERROR,
    NMDL_RESULT_MEMORY_READ_ERROR,
    NMDL_RESULT_MEMORY_ALLOCATION_ERROR,
    NMDL_RESULT_MODEL_LOADING_ERROR,
    NMDL_RESULT_INVALID_MODEL,
    NMDL_RESULT_BOARD_SYNC_ERROR,
    NMDL_RESULT_BOARD_MEMORY_ALLOCATION_ERROR,
    NMDL_RESULT_NN_CREATION_ERROR,
    NMDL_RESULT_NN_LOADING_ERROR,
    NMDL_RESULT_NN_INFO_RETRIEVAL_ERROR,
    NMDL_RESULT_MODEL_IS_TOO_BIG,
    NMDL_RESULT_NOT_INITIALIZED,
    NMDL_RESULT_INCOMPLETE,
    NMDL_RESULT_UNKNOWN_ERROR
} NMDL_RESULT;
```

- *NMDL_RESULT_OK* - no errors,
- *NMDL_RESULT_INVALID_FUNC_PARAMETER* - invalid parameter,
- *NMDL_RESULT_NO_BOARD* – no board,
- *NMDL_RESULT_BOARD_RESET_ERROR* – board reset error,
- *NMDL_RESULT_INIT_CODE_LOADING_ERROR* – initialization code loading error,

- *NMDL_RESULT_CORE_HANDLE_RETRIEVAL_ERROR* – error of obtaining the identifier of the board,
- *NMDL_RESULT_FILE_LOADING_ERROR* – program loading error,
- *NMDL_RESULT_MEMORY_WRITE_ERROR* – write memory error,
- *NMDL_RESULT_MEMORY_READ_ERROR* – read memory error,
- *NMDL_RESULT_MEMORY_ALLOCATION_ERROR* – memory allocation error,
- *NMDL_RESULT_MODEL_LOADING_ERROR* – neural network model loading error,
- *NMDL_RESULT_INVALID_MODEL* – wrong neural network model,
- *NMDL_RESULT_BOARD_SYNC_ERROR* – synchronization error,
- *NMDL_RESULT_BOARD_MEMORY_ALLOCATION_ERROR* – allocation memory error on the board,
- *NMDL_RESULT_NN_CREATION_ERROR* – model creation error on the board,
- *NMDL_RESULT_NN_LOADING_ERROR* – neural network model loading error,
- *NMDL_RESULT_NN_INFO_RETRIEVAL_ERROR* – neural network model information obtaining error,
- *NMDL_RESULT_MODEL_IS_TOO_BIG* – neural network model is too big,
- *NMDL_RESULT_NOT_INITIALIZED* – NMDL is not initialized,
- *NMDL_RESULT_INCOMPLETE* – frame is processing,
- *NMDL_RESULT_UNKNOWN_ERROR* – unknown error.

8.1.5. NMDL_Tensor

Structure describing a tensor.

```
  typedef struct tagNMDL_Tensor {
    unsigned int width;
    unsigned int height;
    unsigned int depth;
} NMDL_Tensor;
```

- *width* - tensor width,
- *height* - tensor height,
- *depth* – tensor depth.

8.2. Functions

8.2.1. NMDL_Blink

LED indication for module identification.

```
  NMDL_RESULT NMDL_Blink(
    unsigned int board_type,
    unsigned int board_number
);
```

- *board_type* - [in] type of module on which the LED indication procedure is called. One of the enumeration values NMDL_BOARD_TYPE .
- *board_number* - [in] the serial number of the module on which the LED indication procedure is called.

8.2.2. NMDL_Create

Instantiate the NMDL and get the *NMDL* instance id.

```
  NMDL_RESULT NMDL_Create(
    NMDL_HANDLE *nmdl
);
```

- *nmdl* - [out] instance id*NMDL*.

After working with the NMDL instance, you need to release the allocated resources by calling NMDL_Destroy.

8.2.3. NMDL_Destroy

Deleting an NMDL Instance.

```
  void NMDL_Destroy(
    NMDL_HANDLE nmdl
);
```

- *nmdl* - [in] instance id *NMDL*.

The function is called to release resources allocated by NMDL_Create and NMDL_Initialize calls.

8.2.4. NMDL_GetBoardCount

Request for the number of detected modules of a given type.

```
  NMDL_RESULT NMDL_GetBoardCount(
    unsigned int board_type,
    unsigned int *boards
);
```

- *board_type* - [in] the type of modules being polled. One of the enumeration values NMDL_BOARD_TYPE .

- *boards* - [out] the number of modules found.

For MC127.05 simulator (type *NMDL_BOARD_TYPE_SIMULATOR*) one module is always detected.

8.2.5. NMDL_GetLibVersion

Library version query *NMDL*.

```
  NMDL_RESULT NMDL_GetLibVersion(
    unsigned int *major,
    unsigned int *minor,
    unsigned int *patch
);
```

- *major* - [out] major version number.

- *minor* - [out] minor version number.

- *patch* - [out] patch version number.

8.2.6. NMDL_GetModelInfo

Request information about the model.

```
  NMDL_RESULT NMDL_GetModelInfo(
    NMDL_HANDLE nmdl,
    unsigned int unit_num,
    NMDL_ModelInfo *model_info
);
```

- *nmdl* - [in] descriptor *NMDL*.

- *unit_num* - [in] unit number for which the processing result is requested. Relevant only when working with *MC127.05* modules, *NMCard*, *NMMezzo* or a simulator. When working with *MC121.01* and *NMStick* , set 0.

- *model_info* - [out] model information.

8.2.7. NMDL_GetOutput

Request for the processing result.

```
  NMDL_RESULT NMDL_GetOutput(
    NMDL_HANDLE nmdl,
    unsigned int unit_num,
    float *outputs[],
    double *fps
);
```

- *nmdl* - [in] instance id *NMDL*.
- *unit_num* - [in] unit number for which the processing result is requested. Relevant only when working with *MC127.05* modules, *NMCard* or a simulator. When working with *MC121.01* and *NMStick* , set 0.
- *outputs* - [out] output tensors buffer. The output tensors are placed one after another in the order of their appearance in the processing graph, that is, ordered by levels in the graph. Tensor parameters are defined in the NMDL_ModelInfo structure.
- *fps* - [out] processing performance (frames per second). May be 0.

8.2.8. NMDL_GetStatus

Request processing status. The function is called to check that the frame has been processed.

```
  NMDL_RESULT NMDL_GetStatus(
    NMDL_HANDLE nmdl,
    unsigned int unit_num,
    unsigned int *status
);
```

- *nmdl* - [in] instance id *NMDL*
- *unit_num* - [in] cluster number for which the processing status is requested. Relevant only when working with *MC127.05* modules, *NMCard* or a simulator. When working with *MC121.01* and *NMStick* , set 0.
- *status* - [out] the state of the cluster at the time the function is called. One of the enumeration values NMDL_PROCESS_FRAME_STATUS .

An example of using *NMDL_GetStatus* to polling. Blocking function to wait for processing to finish:

```
  auto WaitForOutput(NMDL_HANDLE nmdl, std::uint32_t unit_num, float *output[]) {
    std::uint32_t status = NMDL_PROCESS_FRAME_STATUS_INCOMPLETE;
    while(status != NMDL_PROCESS_FRAME_STATUS_FREE) {
        NMDL_GetStatus(nmdl, unit_num, &status);
    };
    double fps;
    NMDL_GetOutput(nmdl, unit_num, output, &fps);
    return fps;
}
```

8.2.9. NMDL_Initialize

NMDL initialization.

```
  NMDL_RESULT NMDL_Initialize(
    NMDL_HANDLE nmdl,
    unsigned int board_type,
    unsigned int board_number,
    unsigned int proc_number,
    const float *model[NMDL_MAX_UNITS],
    const unsigned int model_floats[NMDL_MAX_UNITS]
);
```

- *nmdl* - [in] instance id *NMDL* .
- *board_type* - [in] the type of the module being initialized. One of the enumeration values NMDL_BOARD_TYPE .
- *board_number* - [in] the sequence number of the module being initialized.
- *proc_number* - [in] the sequence number of the processor being initialized. All uniprocessor modules and simulator must be set to 0.
- *model* - [in] pointer to the buffer containing the model in the format *nm7* (for *MC121.01* and *NMStick*) or *nm8* (for *MC127.05* , *NMCard* and simulator).
- *model_floats* - [in] the size of the model in real numbers with single precision.
- *use_batch_mode* - [in] flag for using batch processing mode. Relevant only for *MC127.05* modules, *NMCard* or a simulator. When working with *MC121.01* and *NMStick* , set 0.

For devices based on the NM6407 processor (*MC121.01* and *NMStick*), there is one unit for the neural network inference, so only one model needs to be specified here. For example:

```
  // NMDL_HANDLE nmdl - library descriptor created in NMDL_Create.
// std::vector<float> model - compiled model data.
std::array<const float*, NMDL_MAX_UNITS> models = {
 model.data()
};
std::array<std::uint32_t, NMDL_MAX_UNITS> model_floats = {
 static_cast<std::uint32_t>(model.size())
};
NMDL_Initialize(nmdl, NMDL_BOARD_TYPE_MC12101, 0, models.data(), model_floats.data());
```

For devices based on *NM6408* processor - MC127.05, NMCard, NMMezzo and simulator - you can specify several models, since up to four units can be used here. For more information about processing modes, see Processing Modes. For example:

```
  // NMDL_HANDLE nmdl - library descriptor created in NMDL_Create.
// std::vector<float> model_0 - unit 0 compiled model data.
// std::vector<float> model_1 - unit 1 compiled model data.
// std::vector<float> model_2 - unit 2 compiled model data.
// std::vector<float> model_3 - unit 3 compiled model data.
std::array<const float*, NMDL_MAX_UNITS> models = {
 model_0.data(), model_1.data(), model_2.data(), model_3.data()
};
std::array<std::uint32_t, NMDL_MAX_UNITS> model_floats = {
 static_cast<std::uint32_t>(model_0.size()),
 static_cast<std::uint32_t>(model_1.size()),
 static_cast<std::uint32_t>(model_2.size()),
 static_cast<std::uint32_t>(model_3.size())
};
NMDL_Initialize(nmdl, NMDL_BOARD_TYPE_NMCARD, 0, models.data(), model_floats.data());
```

If the model is compiled for devices with *NM6408* processor to run in *"multi unit"* mode, then you need to set only one model - this model contains data for initializing all four units. For more information about processing modes, see Processing Modes. For example:

```
   // NMDL_HANDLE nmdl - library descriptor created in NMDL_Create.
// std::vector<float> model - compiled model data  to run in "multi unit" mode.
std::array<const float*, NMDL_MAX_UNITS> models = {
 model.data()
};
std::array<std::uint32_t, NMDL_MAX_UNITS> model_floats = {
 static_cast<std::uint32_t>(model.size())
};
NMDL_Initialize(nmdl, NMDL_BOARD_TYPE_NMCARD, 0, models.data(), model_floats.data());
```

After working with NMDL functions, you need to release the allocated resources by calling the deinitialization function NMDL_Release .

8.2.10. NMDL_Process

Input tensors processing.

```
   NMDL_RESULT NMDL_Process(
     NMDL_HANDLE nmdl,
     unsigned int unit_num,
     const float *frame[]
);
```

- *nmdl* - [in] instance id *NMDL*
- *unit_num* - [in] unit number on which processing is started. Relevant only when working with *MC127.05* modules, *NMCard* or a simulator in independent processing mode on units. When working with *MC121.01* and *NMStick* , or in multi unit processing mode, set 0.
- *frame* - [in] an array containing pointers to buffers with input tensors.

The number and geometry of the input tensors must match the loaded neural network model. Call example:

```
   // NMDL_HANDLE nmdl - library descriptor created in NMDL_Create.
// std::vector<float> input_tensor_0 - first input tensor data.
// std::vector<float> input_tensor_1 - second input tensor data.
std::array<const float*, 2> input_tensors = {
 input_tensor_0.data(), input_tensor_1.data()
};
NMDL_Process(nmdl, 0, input_tensors.data());
```

8.2.11. NMDL_Release

Deinitialize NMDL.

```
   void NMDL_Release(
     NMDL_HANDLE nmdl
);
```

- *nmdl* - [in] instance id *NMDL*

9. Description of identifiers and functions nmdl_compiler

9.1. Identifiers

9.1.1. NMDL_COMPILER_BOARD_TYPE

Module types.

```
  typedef enum tagNMDL_COMPILER_BOARD_TYPE {
    NMDL_COMPILER_BOARD_TYPE_MC12101,
    NMDL_COMPILER_BOARD_TYPE_MC12705
} NMDL_COMPILER_BOARD_TYPE;
```

- *NMDL_COMPILER_BOARD_TYPE_MC12101* - modules with *NM6407* processor - *MC121.01* и *NMStick*.

- *NMDL_COMPILER_BOARD_TYPE_MC12705* - modules with *NM6408* processor - *MC127.05*, *NMCard*, *NMMezzo* and simulator.

9.1.2. NMDL_COMPILER_RESULT

Returned result.

```
  typedef enum tagNMDL_COMPILER_RESULT {
    NMDL_COMPILER_RESULT_OK,
    NMDL_COMPILER_RESULT_MEMORY_ALLOCATION_ERROR,
    NMDL_COMPILER_RESULT_MODEL_LOADING_ERROR,
    NMDL_COMPILER_RESULT_INVALID_PARAMETER,
    NMDL_COMPILER_RESULT_INVALID_MODEL,
    NMDL_COMPILER_RESULT_UNSUPPORTED_OPERATION
} NMDL_COMPILER_RESULT;
```

- *NMDL_COMPILER_RESULT_OK* - no errors,

- *NMDL_COMPILER_RESULT_MEMORY_ALLOCATION_ERROR* – memory allocation error,

- *NMDL_COMPILER_RESULT_MODEL_LOADING_ERROR* – error loading neural network model,

- *NMDL_COMPILER_RESULT_INVALID_PARAMETER* - invalid parameter,

- *NMDL_COMPILER_RESULT_INVALID_MODEL* – error in the model,

- *NMDL_COMPILER_RESULT_UNSUPPORTED_OPERATION* – the model contains an unsupported operation.

9.2. Functions

9.2.1. NMDL_COMPILER_CompileDarkNet

Compilation of the original model in DarkNet format.

```
  NMDL_COMPILER_RESULT NMDL_COMPILER_CompileDarkNet(
    unsigned int is_multi_unit,
    unsigned int board,
    const char* src_model,
    unsigned int src_model_size,
    const char* src_weights,
    unsigned int src_weights_size,
    float** dst_model,
    unsigned int* dst_model_floats
);
```

- *is_multi_unit* - [in] flag for using *"multi unit"* processing mode (see section Processing modes). 0 - not used, 1 - used. Parameter relevant only for board type modules = NMDL_COMPILER_BOARD_TYPE_MC12705. For board = NMDL_COMPILER_BOARD_TYPE_MC12101 the parameter is ignored and can take any value.
- *board* - [in] module type identifier for which the model is compiled. One of the enumeration values NMDL_COMPILER_BOARD_TYPE.
- *src_model* - [in] buffer of the original model in the *DarkNet* format, previously read from the *.cfg* file.
- *src_model_size* - [in] the size of the original model buffer in bytes.
- *src_weights* - [in] coefficient buffer previously read from *.weights* .
- *src_weights_size* - [in]size of the coefficient buffer in bytes.
- *dst_model* - [out] compiled model buffer. Memory allocation occurs in a function.
- *dst_model_floats* - [out] compiled model buffer size in *float32* .

9.2.2. NMDL_COMPILER_CompileONNX

Compilation of the source model in ONNX format.

```
  NMDL_COMPILER_RESULT NMDL_COMPILER_CompileONNX(
    unsigned int is_multi_unit,
    unsigned int board,
    const char* src_model,
    unsigned int src_model_size,
    float** dst_model,
    unsigned int* dst_model_floats
);
```

- *is_multi_unit* - [in] flag for using *"multi unit"* processing mode (see section Processing modes). 0 - not used, 1 - used. Parameter relevant only for board type modules = NMDL_COMPILER_BOARD_TYPE_MC12705. For board = NMDL_COMPILER_BOARD_TYPE_MC12101 the parameter is ignored and can take any value.

- *board* - [in] module type identifier for which the model is compiled. One of the enumeration values NMDL_COMPILER_BOARD_TYPE .
- *src_model* - [in] the buffer of the original model in the *ONNX* format, previously read from the file *.onnx* .
- *src_model_size* - [in] the size of the original model buffer in bytes.
- *dst_model* - [out] compiled model buffer. Memory allocation occurs in a function.
- *dst_model_floats* - [out] compiled model buffer size in *float32* .

9.2.3. NMDL_COMPILER_FreeModel

Freeing compile-time allocated memory.

```
    NMDL_COMPILER_RESULT NMDL_COMPILER_FreeModel(
      unsinged int board,
      char* dst_model
);
```

- *board* - [in] module type identifier for which the model was compiled. One of the enumeration values NMDL_COMPILER_BOARD_TYPE .
- *dst_model* - [in]freed memory buffer. Buffer memory was allocated by calling NMDL_COMPILER_CompileDarkNet or NMDL_COMPILER_CompileONNX.

9.2.4. NMDL_COMPILER_GetLastError

Returns a constant string describing the last error.

```
    const char *NMDL_COMPILER_GetLastError();
```

10. Description of identifiers and functions nmdl_image_converter

10.1. Identifiers and structures

10.1.1. NMDL_IMAGE_CONVERTER_BOARD_TYPE

Types of modules.

```
   typedef enum tagNMDL_IMAGE_CONVERTER_BOARD_TYPE {
    NMDL_IMAGE_CONVERTER_BOARD_TYPE_MC12101,
    NMDL_IMAGE_CONVERTER_BOARD_TYPE_MC12705
} NMDL_IMAGE_CONVERTER_BOARD_TYPE;
```

- *NMDL_IMAGE_CONVERTER_BOARD_TYPE_MC12101* - modules *MC121.01* и *NMStick*.

- *NMDL_IMAGE_CONVERTER_BOARD_TYPE_MC12705* - modules *MC127.05*, *NMCard* and simulator.

10.1.2. NMDL_IMAGE_CONVERTER_COLOR_FORMAT

Pixel format identifiers. Describes the ordering of RGB color components in a pixel.

```
   typedef enum tagNMDL_IMAGE_CONVERTER_COLOR_FORMAT {
    NMDL_IMAGE_CONVERTER_COLOR_FORMAT_RGB,
    NMDL_IMAGE_CONVERTER_COLOR_FORMAT_RBG,
    NMDL_IMAGE_CONVERTER_COLOR_FORMAT_GRB,
    NMDL_IMAGE_CONVERTER_COLOR_FORMAT_GBR,
    NMDL_IMAGE_CONVERTER_COLOR_FORMAT_BRG,
    NMDL_IMAGE_CONVERTER_COLOR_FORMAT_BGR,
    NMDL_IMAGE_CONVERTER_COLOR_FORMAT_INTENSITY,
} NMDL_IMAGE_CONVERTER_COLOR_FORMAT;
```

10.2. Functions

10.2.1. NMDL_IMAGE_CONVERTER_Convert

Image preparation.

```
   int NMDL_IMAGE_CONVERTER_Convert(
    const char* src,
    float* dst,
    unsigned int src_size,
    unsigned int dst_width,
    unsigned int dst_height,
    unsigned int dst_color_format,
    const float rgb_divider[3],
    const float rgb_adder[3],
    unsigned int board_type
);
```

- *src* - [in]buffer with the original image. The contents of the buffer correspond to the contents of the image file. Images can be in .bmp, .gif, .jpg and .png formats.

- *dst* - [out] buffer for the prepared image. The buffer memory must be pre-allocated. The buffer size must not be less than the value returned by the NMDL_IMAGE_CONVERTER_RequiredSize function .

- *src_size* - [in] the size of the source image buffer in bytes.

- *dst_width* - [in] width of the prepared image.

- *dst_height* - [in] the height of the prepared image.

- *dst_color_format* - [in] pixel format identifier of the prepared image. One of the enumeration values NMDL_IMAGE_CONVERTER_COLOR_FORMAT .

- *rgb_divider* - [in] divider in expression dst = src / divider + adder. The above operation is performed on each channel of each pixel of the original image. rgb_divider[0] - red channel divider, rgb_divider[1] - green channel divider, rgb_divider[2] - blue channel divider.

- *rgb_adder* - [in] term in expression dst = src / divider + adder. The above operation is performed on channels of each pixel of the original image. rgb_adder[0] - red channel adder, rgb_adder[1] - green channel adder, rgb_adder[2] - blue channel adder.

- *board_type* - [in] the identifier of the type of computing module on which the processing is supposed One of the enumeration values NMDL_IMAGE_CONVERTER_BOARD_TYPE .

For grayscale images, when dst_color_format = NMDL_IMAGE_CONVERTER_COLOR_FORMAT_INTENSITY, only the divisor rgb_divider[0] and the rgb_adder[0] term are used. Other divisors (rgb_divider[1] and rgb_divider[2]) and terms (rgb_adder[1] and rgb_adder[2]) are ignored and can be set to any value.

Returned value: 0 - normal completion, -1 - error.

10.2.2. NMDL_IMAGE_CONVERTER_RequiredSize

Returns the size of the buffer in *float32* elements to hold the prepared image.

```
  int NMDL_IMAGE_CONVERTER_RequiredSize(
    unsigned int dst_width,
    unsigned int dst_height,
    unsigned int dst_color_format,
    unsigned int board_type
);
```

- *dst_width* - [in] width of the prepared image.

- *dst_height* - [in] the height of the prepared image.

- *dst_color_format* - [in] pixel format identifier of the prepared image. One of the enumeration values NMDL_IMAGE_CONVERTER_COLOR_FORMAT .

- *board_type* - [in] the identifier of the type of computing module on which the processing is supposed One of the enumeration values NMDL_IMAGE_CONVERTER_BOARD_TYPE .

Returned value: 0 - normal completion, -1 - error.

Research Centre Module
Box: 166, Moscow, 125190, Russia
Tel: +7 (499) 152-9698
Fax: +7 (499) 152-4661
E-Mail: sales@module.ru
WWW: http://www.module.ru

RC Module reserves the right to make changes without further notices to product herein to improve reliability, function or design. RC Module shall not be liable for any loss or damage arising from the use of any information in this overview or any error or omission in such information, or any incorrect use of the product.

Printed in Russia Data of issue: