N2D2 Documentation

CEA LIST

Jan 23, 2023

INTRODUCTION

CHAPTER

ONE

PRESENTATION

The N2D2 platform is a comprehensive solution for fast and accurate Deep Neural Network (DNN) simulation and full and automated DNN-based applications building. The platform integrates database construction, data pre-processing, network building, benchmarking and hardware export to various targets. It is particularly useful for DNN design and exploration, allowing simple and fast prototyping of DNN with different topologies. It is possible to define and learn multiple network topology variations and compare the performances (in terms of recognition rate and computationnal cost) automatically. Export targets include CPU, DSP and GPU with OpenMP, OpenCL, Cuda, cuDNN and TensorRT programming models as well as custom hardware IP code generation with High-Level Synthesis for FPGA and dedicated configurable DNN accelerator IP[1].

In the following, the first section describes the database handling capabilities of the tool, which can automatically generate learning, validation and testing data sets from any hand made database (for example from simple files directories). The second section briefly describes the data pre-processing capabilites built-in the tool, which does not require any external pre-processing step and can handle many data transformation, normalization and augmentation (for example using elastic distortion to improve the learning). The third section show an example of DNN building using a simple INI text configuration file. The fourth section show some examples of metrics obtained after the learning and testing to evaluate the performances of the learned DNN. Next, the fifth section introduces the DNN hardware export capabilities of the toolflow, which can automatically generate ready to use code for various targets such as embedded GPUs or full custom dedicated FPGA IP. Finally, we conclude by summarising the main features of the tool.

1.1 Database handling

The tool integrates everything needed to handle custom or hand made databases:

- Genericity: load image and sound, 1D, 2D or 3D data;
- Associate a label for each data point (useful for scene labeling for example) or a single label to each data file (one object/class per image for example), 1D or 2D labels;
- Advanced Region of Interest (ROI) handling:
  - Support arbitrary ROI shapes (circular, rectangular, polygonal or pixelwise defined);
  - Convert ROIs to data point (pixelwise) labels;
  - Extract one or multiple ROIs from an initial dataset to create as many corresponding additional data to feed the DNN;
- Native support of file directory-based databases, where each sub-directory represents a different label. Most used image file formats are supported (JPEG, PNG, PGM…);
- Possibility to add custom datafile format in the tool without any change in the code base;
- Automatic random partitionning of the database into learning, validation and testing sets.

[1] Ongoing work

1.2 Data pre-processing

Data pre-processing, such as image rescaling, normalization, filtering… is directly integrated into the toolflow, with no need for external tool or pre-processing. Each pre-processing step is called a *transformation*.

The full sequence of transformations can be specified easily in a INI text configuration file. For example:

```
; First step: convert the image to grayscale
[env.Transformation-1]
Type=ChannelExtractionTransformation
CSChannel=Gray

; Second step: rescale the image to a 29x29 size
[env.Transformation-2]
Type=RescaleTransformation
Width=29
Height=29

; Third step: apply histogram equalization to the image
[env.Transformation-3]
Type=EqualizeTransformation

; Fourth step (only during learning): apply random elastic distortions to the images to␣
↪extent the learning set
[env.OnTheFlyTransformation]
Type=DistortionTransformation
ApplyTo=LearnOnly
ElasticGaussianSize=21
ElasticSigma=6.0
ElasticScaling=20.0
Scaling=15.0
Rotation=15.0
```

Example of pre-processing transformations built-in in the tool are:

- Image color space change and color channel extraction;
- Elastic distortion;
- Histogram equalization (including CLAHE);
- Convolutional filtering of the image with custom or pre-defined kernels (Gaussian, Gabor…);
- (Random) image flipping;
- (Random) extraction of fixed-size slices in a given label (for multi-label images)
- Normalization;
- Rescaling, padding/cropping, triming;
- Image data range clipping;
- (Random) extraction of fixed-size slices.

1.3 Deep network building

The building of a deep network is straightforward and can be done withing the same INI configuration file. Several layer types are available: convolutional, pooling, fully connected, Radial-basis function (RBF) and softmax. The tool is highly modular and new layer types can be added without any change in the code base. Parameters of each layer type are modifiable, for example for the convolutional layer, one can specify the size of the convolution kernels, the stride, the number of kernels per input map and the learning parameters (learning rate, initial weights value…). For the learning, the data dynamic can be chosen between 16 bits (with NVIDIA cuDNN[2]), 32 bit and 64 bit floating point numbers.

The following example, which will serve as the use case for the rest of this presentation, shows how to build a DNN with 5 layers: one convolution layer, followed by one MAX pooling layer, followed by two fully connected layers and a softmax output layer.

```
; Specify the input data format
[env]
SizeX=24
SizeY=24
BatchSize=12

; First layer: convolutional with 3x3 kernels
[conv1]
Input=env
Type=Conv
KernelWidth=3
KernelHeight=3
NbOutputs=32
Stride=1

; Second layer: MAX pooling with pooling area 2x2
[pool1]
Input=conv1
Type=Pool
Pooling=Max
PoolWidth=2
PoolHeight=2
NbOutputs=32
Stride=2
Mapping.Size=1 ; one to one connection between convolution output maps and pooling input␣
↪maps

; Third layer: fully connected layer with 60 neurons
[fc1]
Input=pool1
Type=Fc
NbOutputs=60

; Fourth layer: fully connected with 10 neurons
[fc2]
Input=fc1
Type=Fc
NbOutputs=10
```

(continues on next page)

[2] On future GPUs

(continued from previous page)

```
; Final layer: softmax
[softmax]
Input=fc2
Type=Softmax
NbOutputs=10
WithLoss=1

[softmax.Target]
TargetValue=1.0
DefaultValue=0.0
```

The resulting DNN is shown in figure [fig:DNNExample].

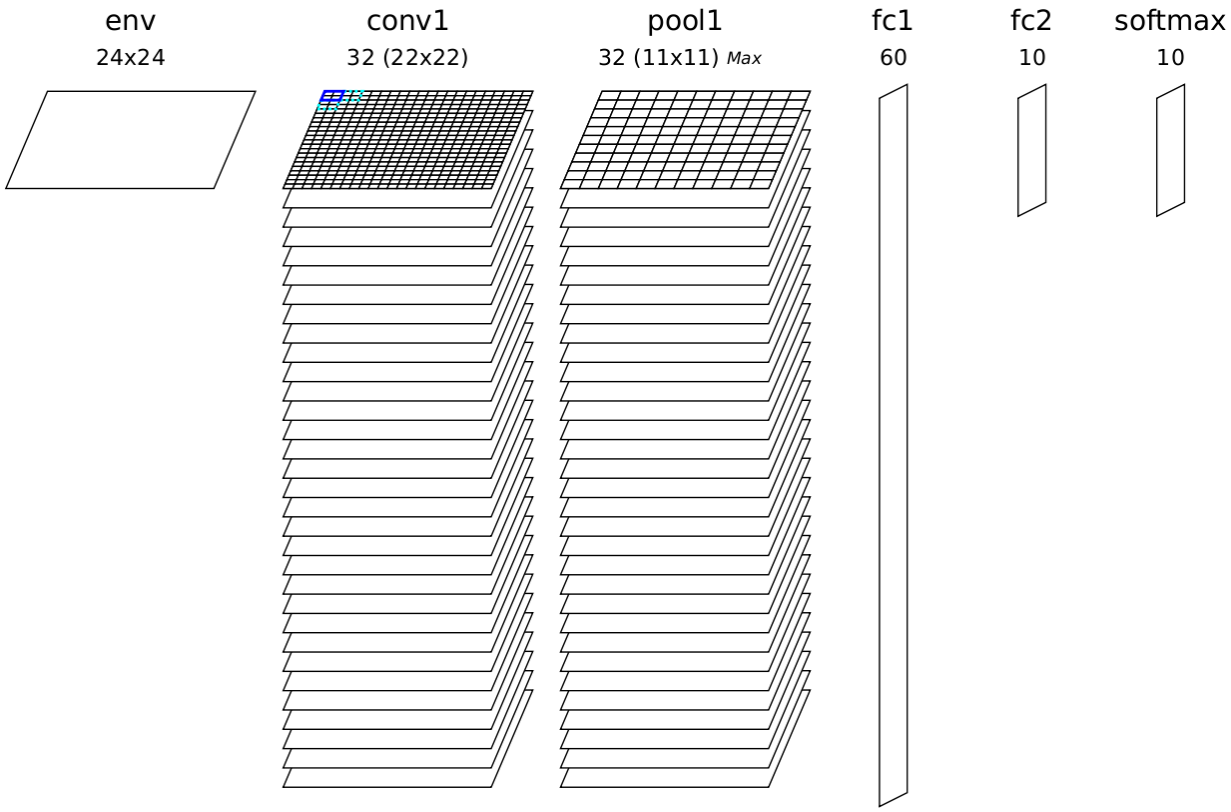


Fig. 1: Automatically generated and ready to learn DNN from the INI configuration file example.

The learning is accelerated in GPU using the NVIDIA cuDNN framework, integrated into the toolflow. Using GPU acceleration, learning times can be reduced typically by two orders of magnitude, enabling the learning of large databases within tens of minutes to a few hours instead of several days or weeks for non-GPU accelerated learning.

1.4 Performances evaluation

The software automatically outputs all the information needed for the network applicative performances analysis, such as the recognition rate and the validation score during the learning; the confusion matrix during learning, validation and test; the memory and computation requirements of the network; the output maps activity for each layer, and so on, as shown in figure [fig:metrics].

1.5 Hardware exports

Once the learned DNN recognition rate performances are satisfying, an optimized version of the network can be automatically exported for various embedded targets. An automated network computation performances benchmarking can also be performed among different targets.

The following targets are currently supported by the toolflow:

- Plain C code (no dynamic memory allocation, no floating point processing);
- C code accelerated with OpenMP;
- C code tailored for High-Level Synthesis (HLS) with Xilinx Vivado HLS;
  - Direct synthesis to FPGA, with timing and utilization after routing;
  - Possibility to constrain the maximum number of clock cycles desired to compute the whole network;
  - FPGA utilization vs number of clock cycle trade-off analysis;
- OpenCL code optimized for either CPU/DSP or GPU;
- Cuda kernels, cuDNN and TensorRT code optimized for NVIDIA GPUs.

Different automated optimizations are embedded in the exports:

- DNN weights and signal data precision reduction (down to 8 bit integers or less for custom FPGA IPs);
- Non-linear network activation functions approximations;
- Different weights discretization methods.

The exports are generated automatically and come with a Makefile and a working testbench, including the pre-processed testing dataset. Once generated, the testbench is ready to be compiled and executed on the target platform. The applicative performance (recognition rate) as well as the computing time per input data can then be directly mesured by the testbench.

The figure [fig:TargetsBenchmarking] shows an example of benchmarking results of the previous DNN on different targets (in log scale). Compared to desktop CPUs, the number of input image pixels processed per second is more than one order of magnitude higher with GPUsand at least two orders of magnitude better with synthesized DNN on FPGA.

1.6 Summary

The N2D2 platform is today a complete and production ready neural network building tool, which does not require advanced knownledges in deep learning to be used. It is tailored for fast neural network applications generation and porting with minimum overhead in terms of database creation and management, data pre-processing, networks configuration and optimized code generation, which can save months of manual porting and verification effort to a single automated step in the tool.

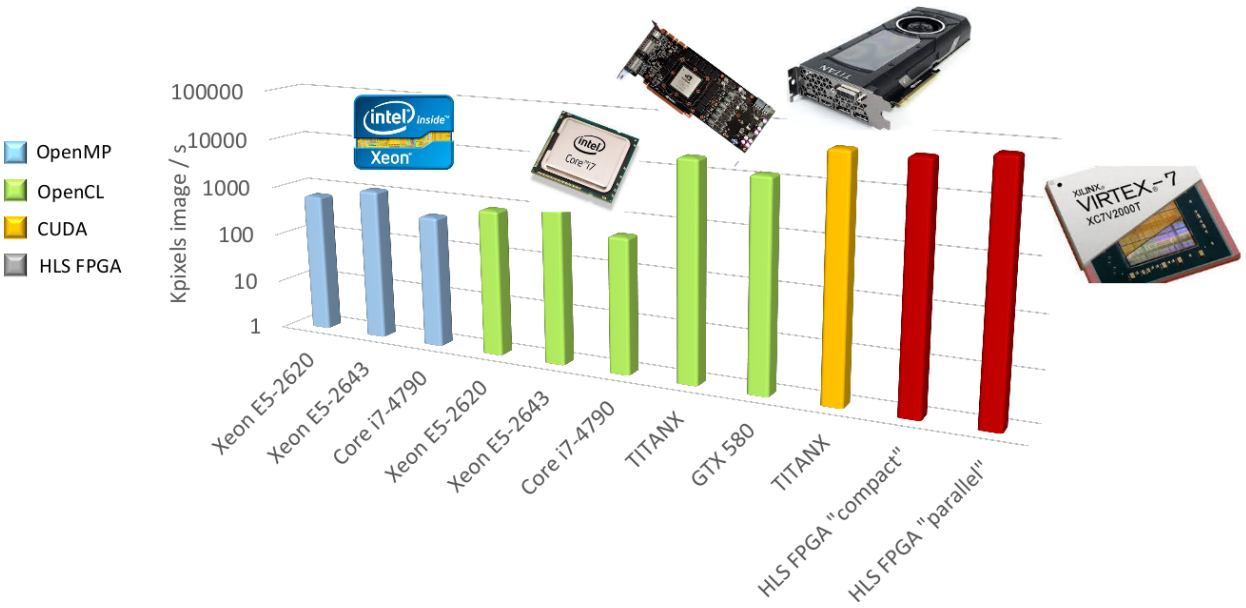


Fig. 2: Example of network benchmarking on different hardware targets.

CHAPTER

TWO

ABOUT N2D2-IP

While N2D2 is our deep learning open-source core framework, some modules referred as "N2D2-IP" in the manual, are only available through custom license agreement with CEA LIST.

If you are interested in obtaining some of these modules, please contact our business developer for more information on available licensing options:

Sandrine VARENNE (Sandrine.VARENNE@cea.fr)

In addition to N2D2-IP modules, we can also provide our expertise to design specific solutions for integrating DNN in embedded hardware systems, where power, latency, form factor and/or cost are constrained. We can target CPU/DSP/GPU CoTS hardware as well as our own PNeuro (programmable) and DNeuro (dataflow) dedicated hardware accelerator IPs for DNN on FPGA or ASIC.

CHAPTER

THREE

PERFORMING SIMULATIONS

3.1 Minimum system requirements

- Supported processors:
  - ARM Cortex A15 (tested on Tegra K1)
  - ARM Cortex A53/A57 (tested on Tegra X1)
  - Pentium-compatible PC (Pentium III, Athlon or more-recent system recommended)
- Supported operating systems:
  - Windows $\geq$ 7 or Windows Server $\geq$ 2012, 64 bits with Visual Studio $\geq$ 2015.2 (2015 Update 2)
  - GNU/Linux with GCC $\geq$ 4.4 (tested on RHEL $\geq$ 6, Debian $\geq$ 6, Ubuntu $\geq$ 14.04)
- At least 256 MB of RAM (1 GB with GPU/CUDA) for MNIST dataset processing
- At least 150 MB available hard disk space + 350 MB for MNIST dataset processing

For CUDA acceleration:

- CUDA $\geq$ 6.5 and CuDNN $\geq$ 1.0
- NVIDIA GPU with CUDA compute capability $\geq$ 3 (starting from *Kepler* micro-architecture)
- At least 512 MB GPU RAM for MNIST dataset processing

3.2 Obtaining N2D2

3.2.1 Prerequisites

Red Hat Enterprise Linux (RHEL) 6

Make sure you have the following packages installed:

- `cmake`
- `gnuplot`
- `opencv`
- `opencv-devel` (may require the `rhel-x86_64-workstation-optional-6` repository channel)

Plus, to be able to use GPU acceleration:

- Install the CUDA repository package:

```
rpm -Uhv http://developer.download.nvidia.com/compute/cuda/repos/rhel6/x86_64/cuda-repo-
↪rhel6-7.5-18.x86_64.rpm
yum clean expire-cache
yum install cuda
```

- Install cuDNN from the NVIDIA website: register to NVIDIA Developer and download the latest version of cuDNN. Simply copy the header and library files from the cuDNN archive to the corresponding directories in the CUDA installation path (by default: /usr/local/cuda/include and /usr/local/cuda/lib64, respectively).
- Make sure the CUDA library path (e.g. /usr/local/cuda/lib64) is added to the LD_LIBRARY_PATH environment variable.

Ubuntu

Make sure you have the following packages installed, if they are available on your Ubuntu version:

- `cmake`
- `gnuplot`
- `libopencv-dev`
- `libcv-dev`
- `libhighgui-dev`

Plus, to be able to use GPU acceleration:

- Install the CUDA repository package matching your distribution. For example, for Ubuntu 14.04 64 bits:

```
wget http://developer.download.nvidia.com/compute/cuda/repos/ubuntu!\color{gray}{1404}!/!
↪\color{gray}{x86\_64}!/cuda-repo-ubuntu!\color{gray}{1404}!_7.5-18_!\color{gray}{amd64}
↪!.deb
dpkg -i cuda-repo-ubuntu!\color{gray}{1404}!_7.5-18_!\color{gray}{amd64}!.deb
```

- Install the cuDNN repository package matching your distribution. For example, for Ubuntu 14.04 64 bits:

```
  wget http://developer.download.nvidia.com/compute/machine-learning/repos/ubuntu!\color
↪{gray}{1404}!/!\color{gray}{x86\_64}!/nvidia-machine-learning-repo-ubuntu!\color{gray}
↪{1404}!_4.0-2_!\color{gray}{amd64}!.deb
  dpkg -i nvidia-machine-learning-repo-ubuntu!\color{gray}{1404}!_4.0-2_!\color{gray}
↪{amd64}!.deb

Note that the cuDNN repository package is provided by NVIDIA for Ubuntu
starting from version 14.04.
```

- Update the package lists: `apt-get update`
- Install the CUDA and cuDNN required packages:

```
apt-get install cuda-core-7-5 cuda-cudart-dev-7-5 cuda-cublas-dev-7-5 cuda-curand-dev-7-
↪5 libcudnn5-dev
```

- Make sure there is a symlink to `/usr/local/cuda`:

```
ln -s /usr/local/cuda-7.5 /usr/local/cuda
```

- Make sure the CUDA library path (e.g. /usr/local/cuda/lib64) is added to the LD_LIBRARY_PATH environment variable.

Windows

On Windows 64 bits, Visual Studio ≥ 2015.2 (2015 Update 2) is required.

Make sure you have the following software installed:

- CMake (http://www.cmake.org/): download and run the Windows installer.
- `dirent.h` C++ header (https://github.com/tronkko/dirent): to be put in the Visual Studio include path.
- Gnuplot (http://www.gnuplot.info/): the bin sub-directory in the install path needs to be added to the Windows `PATH` environment variable.
- OpenCV (http://opencv.org/): download the latest 2.x version for Windows and extract it to, for example, `C:\OpenCV\`. Make sure to define the environment variable `OpenCV_DIR` to point to `C:\OpenCV\opencv\build`. Make sure to add the bin sub-directory (`C:\OpenCV\opencv\build\x64\vc12\bin`) to the Windows `PATH` environment variable.

Plus, to be able to use GPU acceleration:

- Download and install CUDA toolkit 8.0 located at https://developer.nvidia.com/compute/cuda/8.0/prod/local_installers/cuda_8.0.44_windows-exe:

```
rename cuda_8.0.44_windows-exe cuda_8.0.44_windows.exe
cuda_8.0.44_windows.exe -s compiler_8.0 cublas_8.0 cublas_dev_8.0 cudart_8.0 curand_8.0␣
↪curand_dev_8.0
```

- Update the `PATH` environment variable:

```
set PATH=%ProgramFiles%\NVIDIA GPU Computing Toolkit\CUDA\v8.0\bin;%ProgramFiles%\NVIDIA␣
↪GPU Computing Toolkit\CUDA\v8.0\libnvvp;%PATH%
```

- Download and install cuDNN 8.0 located at http://developer.download.nvidia.com/compute/redist/cudnn/v5.1/cudnn-8.0-windows7-x64-v5.1.zip (the following command assumes that you have 7-Zip installed):

```
7z x cudnn-8.0-windows7-x64-v5.1.zip
copy cuda\include\*.* ^
  "%ProgramFiles%\NVIDIA GPU Computing Toolkit\CUDA\v8.0\include\"
copy cuda\lib\x64\*.* ^
  "%ProgramFiles%\NVIDIA GPU Computing Toolkit\CUDA\v8.0\lib\x64\"
copy cuda\bin\*.* ^
  "%ProgramFiles%\NVIDIA GPU Computing Toolkit\CUDA\v8.0\bin\"
```

3.2.2 Getting the sources

Use the following command:

```
git clone --recursive git@github.com:CEA-LIST/N2D2.git
```

Note: You need the recursive option to download the pybind11 submodule.

3.2.3 Compilation

To compile the program:

```
mkdir build
cd build
cmake .. && make
```

On Windows, you may have to specify the generator, for example:

```
cmake .. -G"Visual Studio 14"
```

Then open the newly created N2D2 project in Visual Studio 2015. Select "Release" for the build target. Right click on `ALL_BUILD` item and select "Build".

3.3 Downloading training datasets

A python script located in the repository root directory allows you to select and automatically download some well-known datasets, like MNIST and GTSRB (the script requires Python 2.x with bindings for GTK 2 package):

```
./tools/install/install_dataset.py
```

By default, the datasets are downloaded in the path specified in the `N2D2_DATA` environment variable, which is the root path used by the N2D2 tool to locate the databases. If the `N2D2_DATA` variable is not set, the default value used is /local/$USER/n2d2_data/ (or /local/n2d2_data/ if the `USER` environment variable is not set) on Linux and C:\n2d2_data\ on Windows.

Please make sure you have write access to the `N2D2_DATA` path, or if not set, in the default /local/$USER/n2d2_data/ path.

3.4 Run the learning

The following command will run the learning for 600,000 image presentations/steps and log the performances of the network every 10,000 steps:

```
./n2d2 "mnist24_16c4s2_24c5s2_150_10.ini" -learn 600000 -log 10000
```

Note: you may want to check the gradient computation using the `-check` option. Note that it can be extremely long and can occasionally fail if the required precision is too high.

3.5 Test a learned network

After the learning is completed, this command evaluate the network performances on the test data set:

```
./n2d2 "mnist24_16c4s2_24c5s2_150_10.ini" -test
```

3.5.1 Interpreting the results

Recognition rate

The recognition rate and the validation score are reported during the learning in the *TargetScore_/Success_validation.png* file, as shown in figure [fig:validationScore].

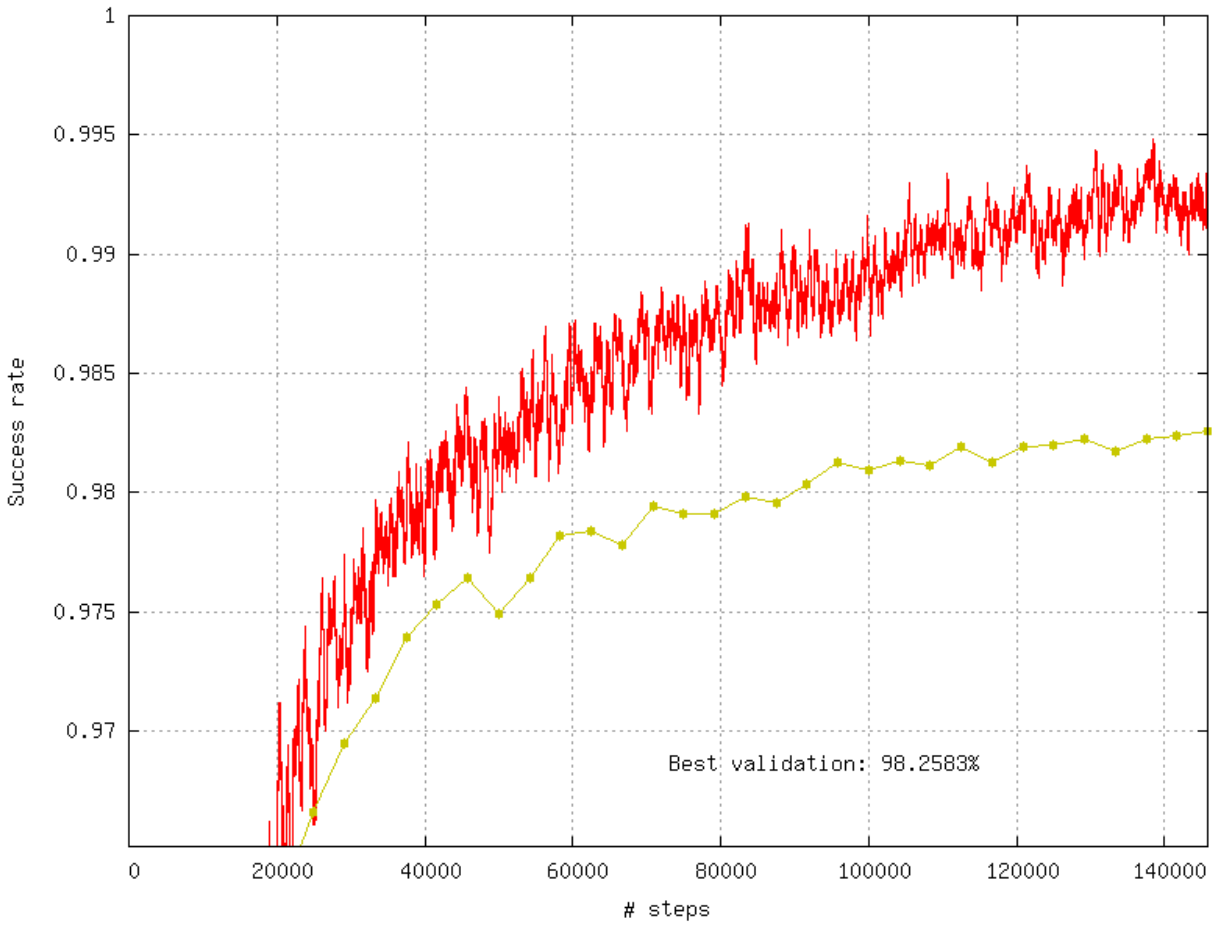


Fig. 1: Recognition rate and validation score during learning.

Confusion matrix

The software automatically outputs the confusion matrix during learning, validation and test, with an example shown in figure [fig:ConfusionMatrix]. Each row of the matrix contains the number of occurrences estimated by the network for each label, for all the data corresponding to a single actual, target label. Or equivalently, each column of the matrix contains the number of actual, target label occurrences, corresponding to the same estimated label. Idealy, the matrix should be diagonal, with no occurrence of an estimated label for a different actual label (network mistake).

The confusion matrix reports can be found in the simulation directory:

- *TargetScore_/ConfusionMatrix_learning.png*;
- *TargetScore_/ConfusionMatrix_validation.png*;
- *TargetScore_/ConfusionMatrix_test.png*.

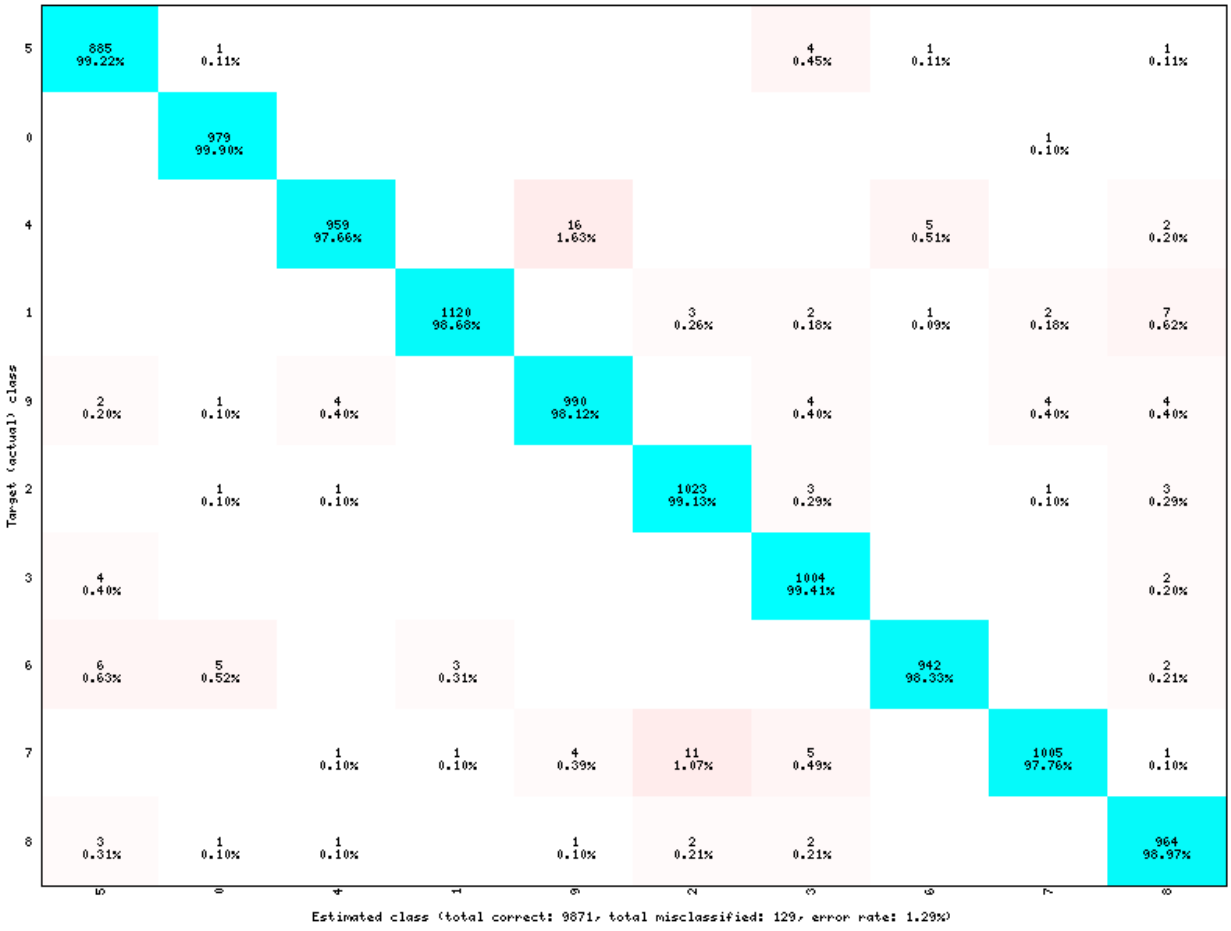


Fig. 2: Example of confusion matrix obtained after the learning.

Memory and computation requirements

The software also report the memory and computation requirements of the network, as shown in figure [fig:stats]. The corresponding report can be found in the *stats* sub-directory of the simulation.

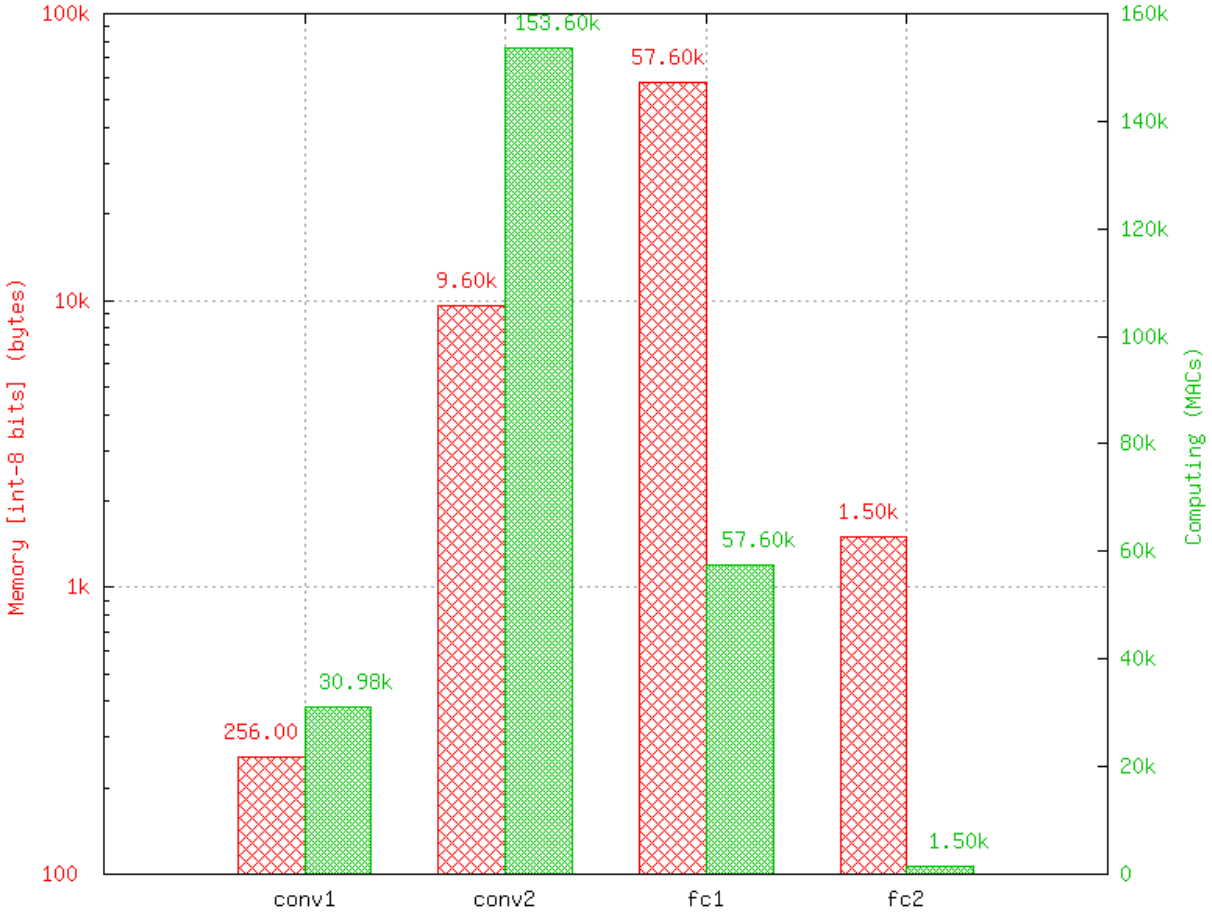


Fig. 3: Example of memory and computation requirements of the network.

Kernels and weights distribution

The synaptic weights obtained during and after the learning can be analyzed, in terms of distribution (*weights* sub-directory of the simulation) or in terms of kernels (*kernels* sub-directory of the simulation), as shown in [fig:weights].

Output maps activity

The initial output maps activity for each layer can be visualized in the *outputs_init* sub-directory of the simulation, as shown in figure [fig:outputs].

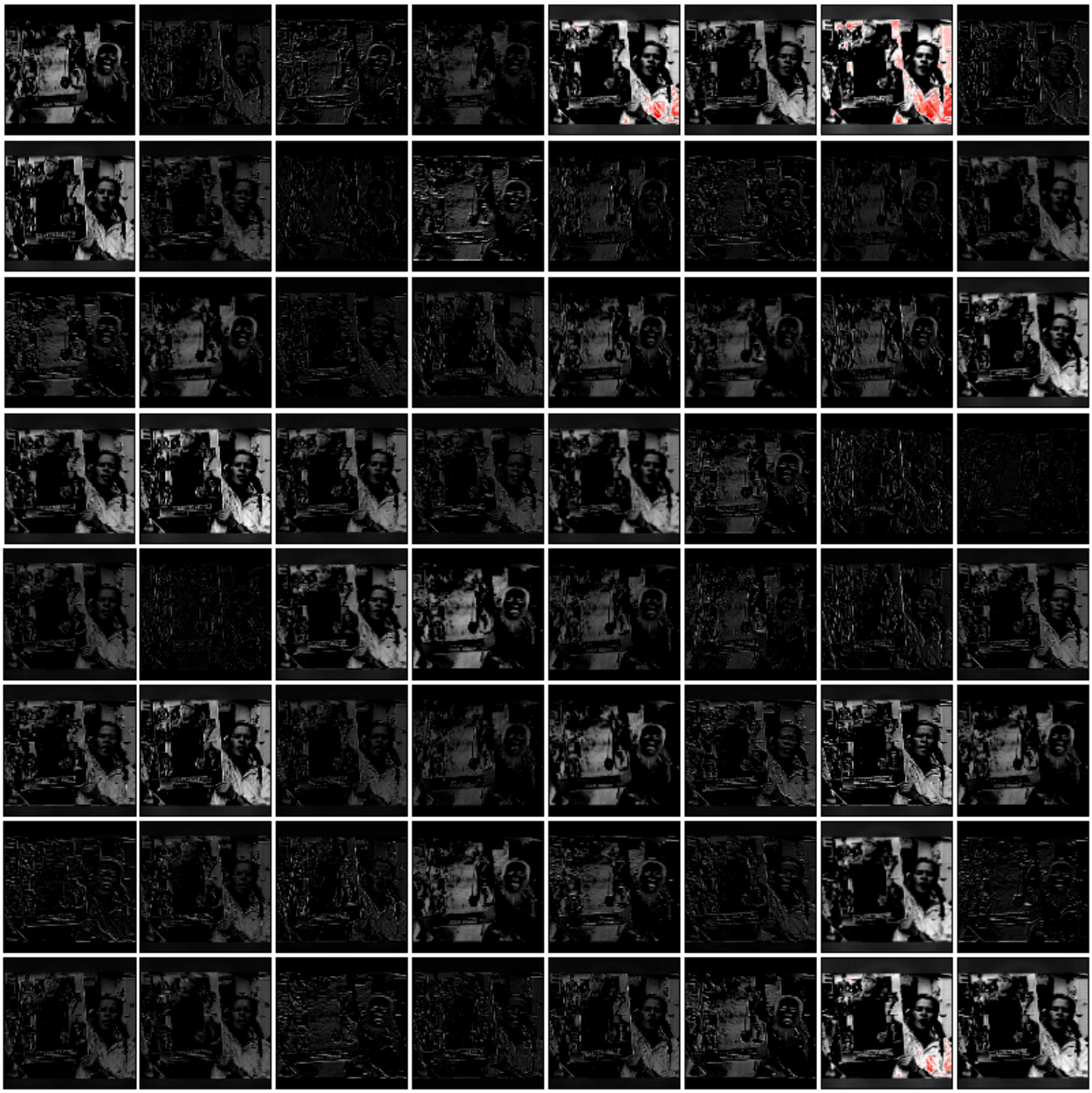


Fig. 4: Output maps activity example of the first convolutional layer of the network.

CHAPTER

FOUR

PERFORMANCE EVALUATION TOOLS

4.1 Key performance metrics

The key performance metrics are the recall and precision for each class and the average recall and precision over all the classes. These performance metrics are automatically computed by the interactive confusion matrix tool embedded in N2D2, as shown below.

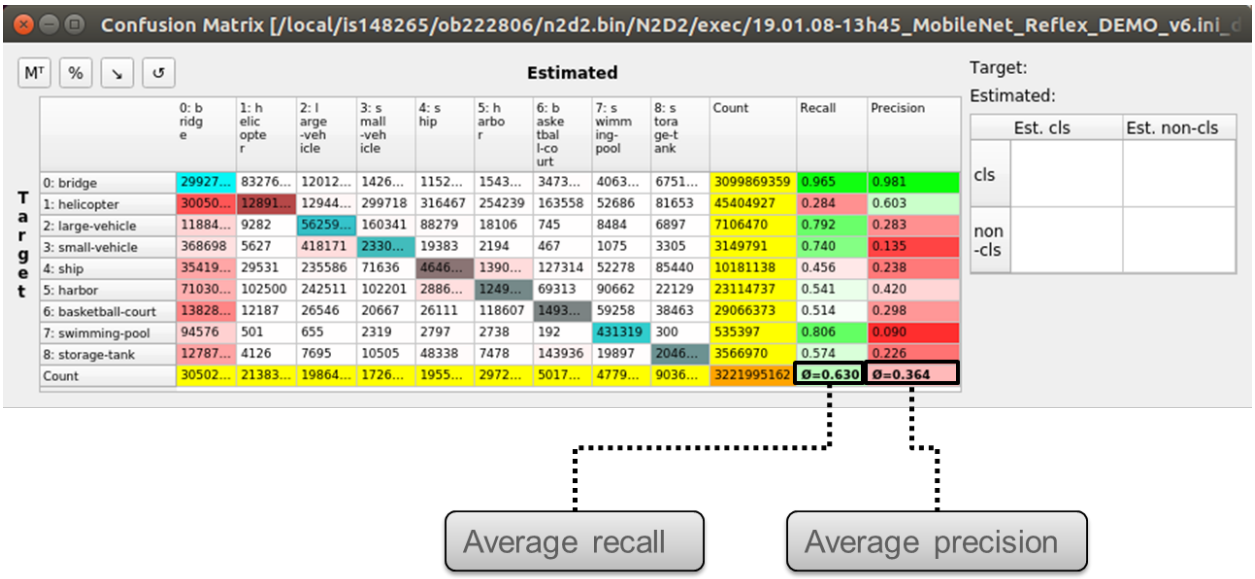


Fig. 1: Recall and precision for each class and average recall and precision over all the classes in the interactive confusion matrix tool.

The accuracy, in classification context, is not a valid metric for unbalanced classes, which is typically the case for object detection / segmentation applications, where the background represent a much larger portion of the objects or defects. Indeed, in this case, the number of true negative is easily much larger than the number of true positives, false positives and false negatives combined, leading to accuracies close to 100% even for no informative (non working) classification, as illustrated in the figure below.

During the learning, N2D2 also provides a range of metrics with the confusion matrix, which are saved in the `*.Target/ConfusionMatrix_*_score.png` file, as shown in the extract in the following figure. The metric used for the validation can be specified with the `-valid-metric` N2D2 command line argument. The default metric used is the recall (a.k.a. sensitivity).

The metric figure used for the validation and the computing of the overall score in N2D2 is a raw figure, aggregating all the classes and images. It can therefore differ from the average metric reported in the confusion matrix, which is

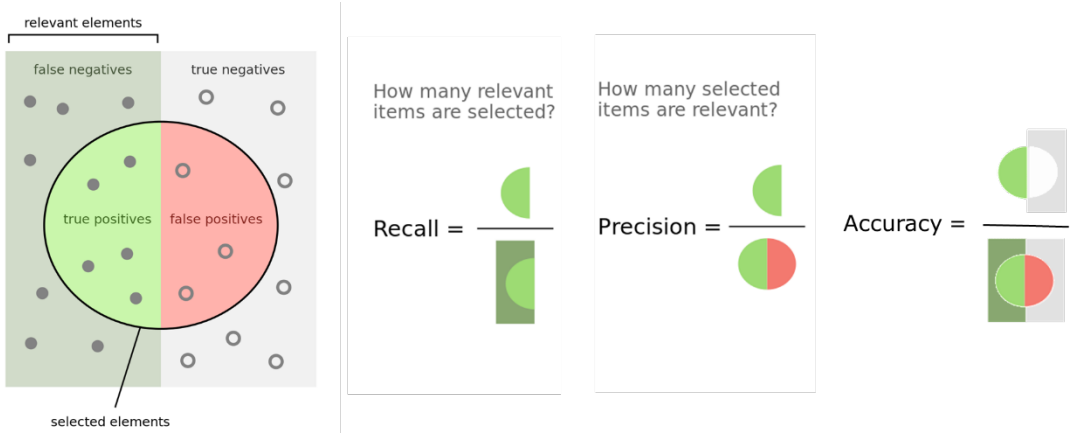


Fig. 2: Recall, Precision and Accuracy definitions.

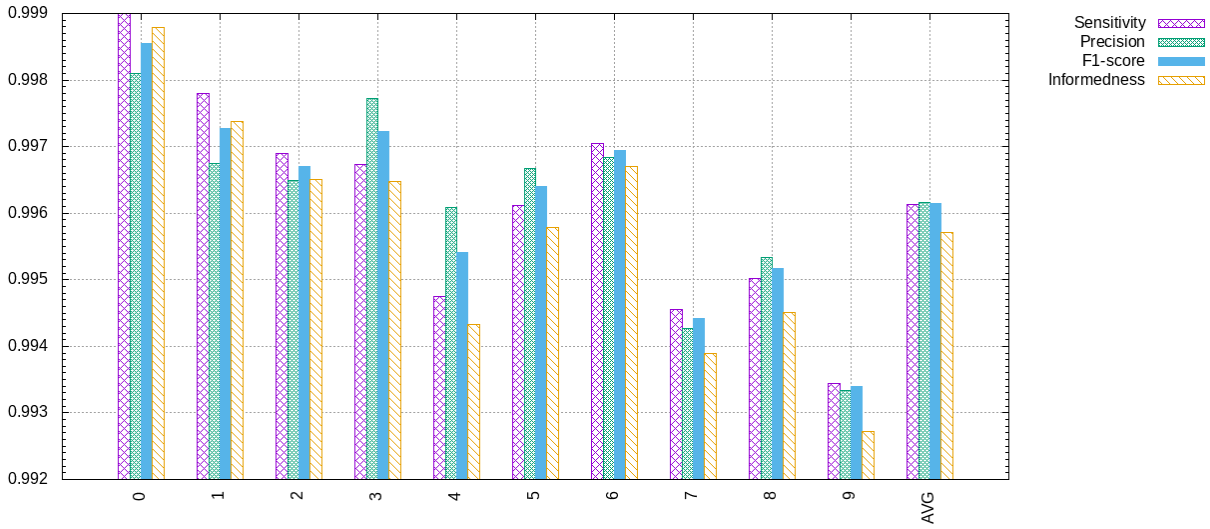


Fig. 3: Metrics associated to the confusion matrix in N2D2.

the average of the metric of each class.

4.2 Interactive Confusion Matrix Tool

N2D2-IP only: available upon request.

4.2.1 Overview

The interactive confusion matrix (main window show in next figure) tool allows you to explore, sort, combine or extract scores from large confusion matrix. Its main features are:

- Sorting;
- Transpose switch;
- Recall and precision;
- Aggregated recall and precision on selected classes;
- Percentages / total count switch;
- Reordering / ordering reset;
- for ignored area.

The tool can be run after a learning or a test in N2D2, by launching the `*.Target/ConfusionMatrix_*.py` Python script (requires Python 3.7).

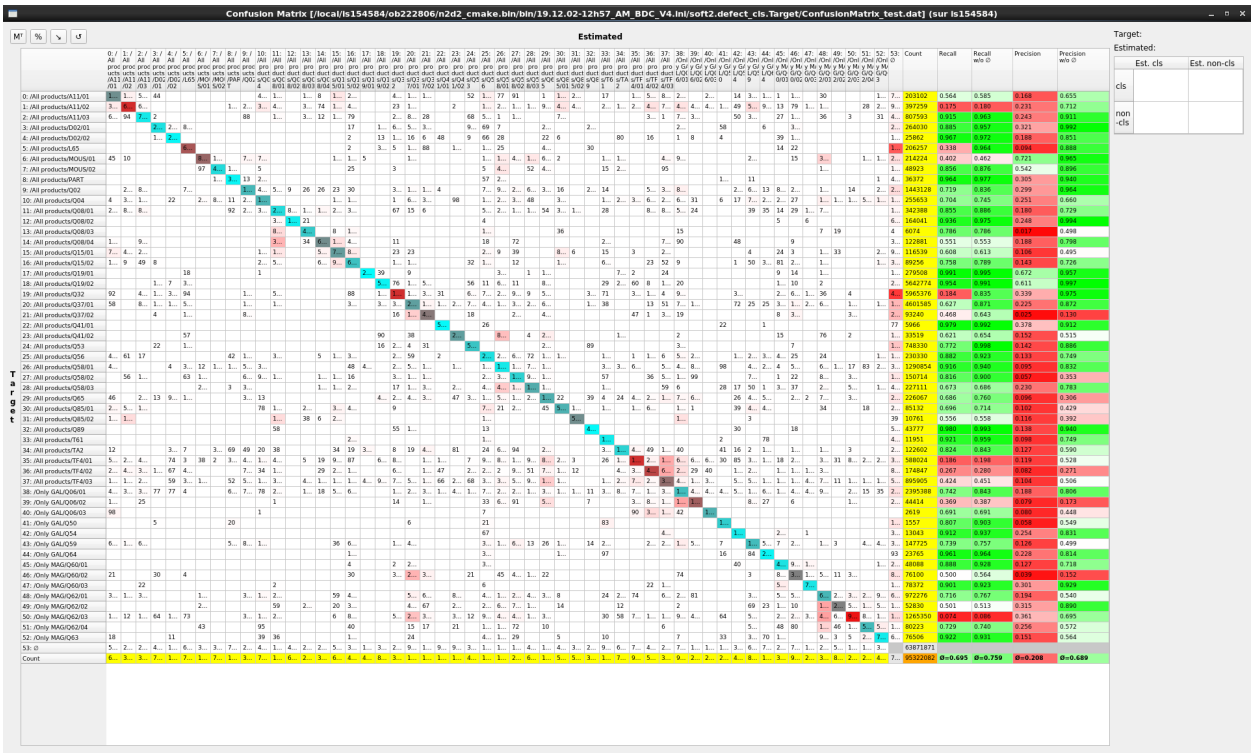


Fig. 4: Interactive confusion matrix tool main window.

4.2.2 Single class performances evaluation

Single class recall and precision score metrics are shown for each row of the confusion matrix. There are two sets of metrics:

- Recall w/o Ø and precision w/o Ø: metrics considering only the defect type confusion over the pixels annotated as defect, as show in the table below;

		Estimated class: Selected class(es)	Estimated class: Other classes	
Target class	Selected class(es)	True positive	False negative	Recall w/o Ø = Σ True positive / Σ Target selected class(es)
	Other classes	False positive	True negative	
		Precision w/o Ø = Σ True positive / Σ Estimated selected class(es)		

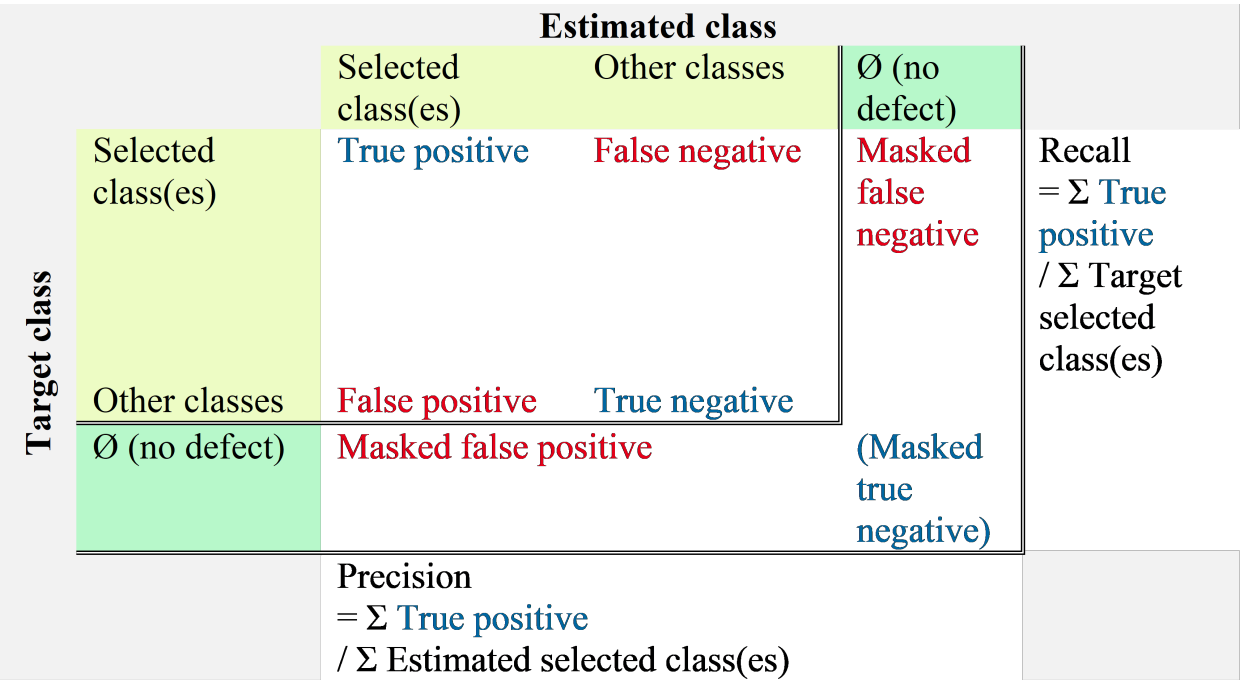
- Recall and precision: metrics including the defect/no defect confusion as well as the type confusion, as shown in the table below.

4.2.3 Classes aggregation

When selecting multiple row in the confusion table, the overall recall and precision are automatically computed for the selection by aggregating the selected values. They are displayed in the right table, as shown below.

4.2.4 Selected items table view

When double-clicking on a single cell in the confusion table, or pressing the Enter key with a selection, the list of all the images with the confusions for the selected cells is displayed. The recall and precision are computed for each (image + target class) pair, as shown below.



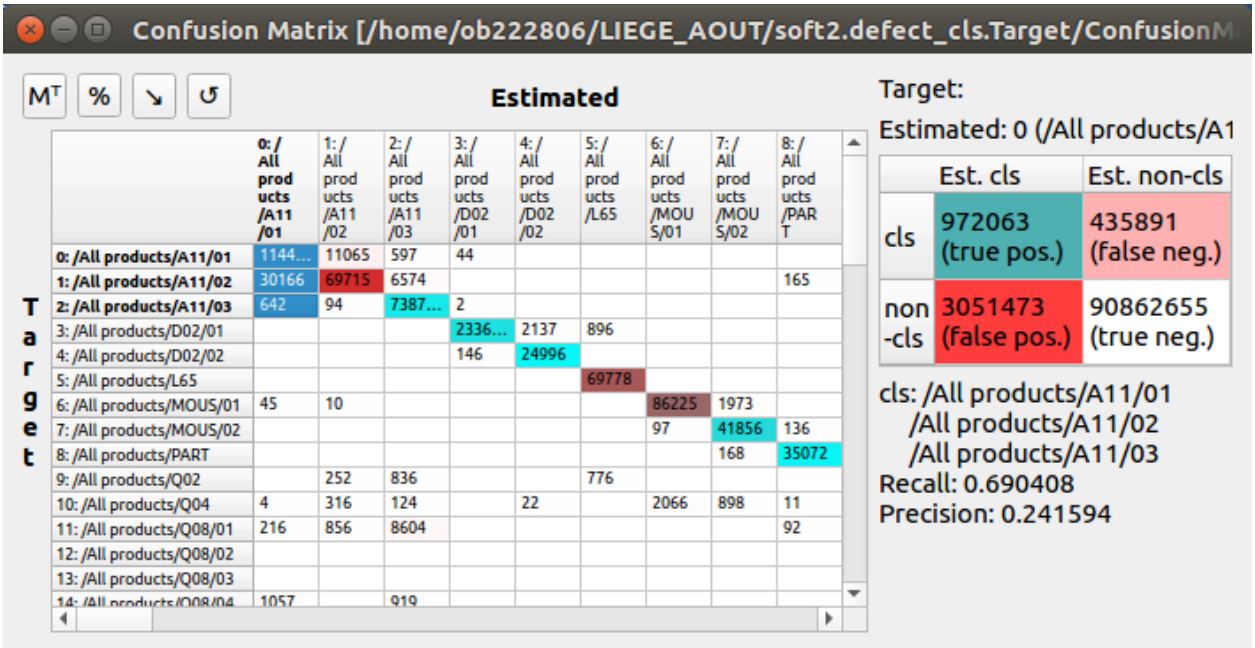
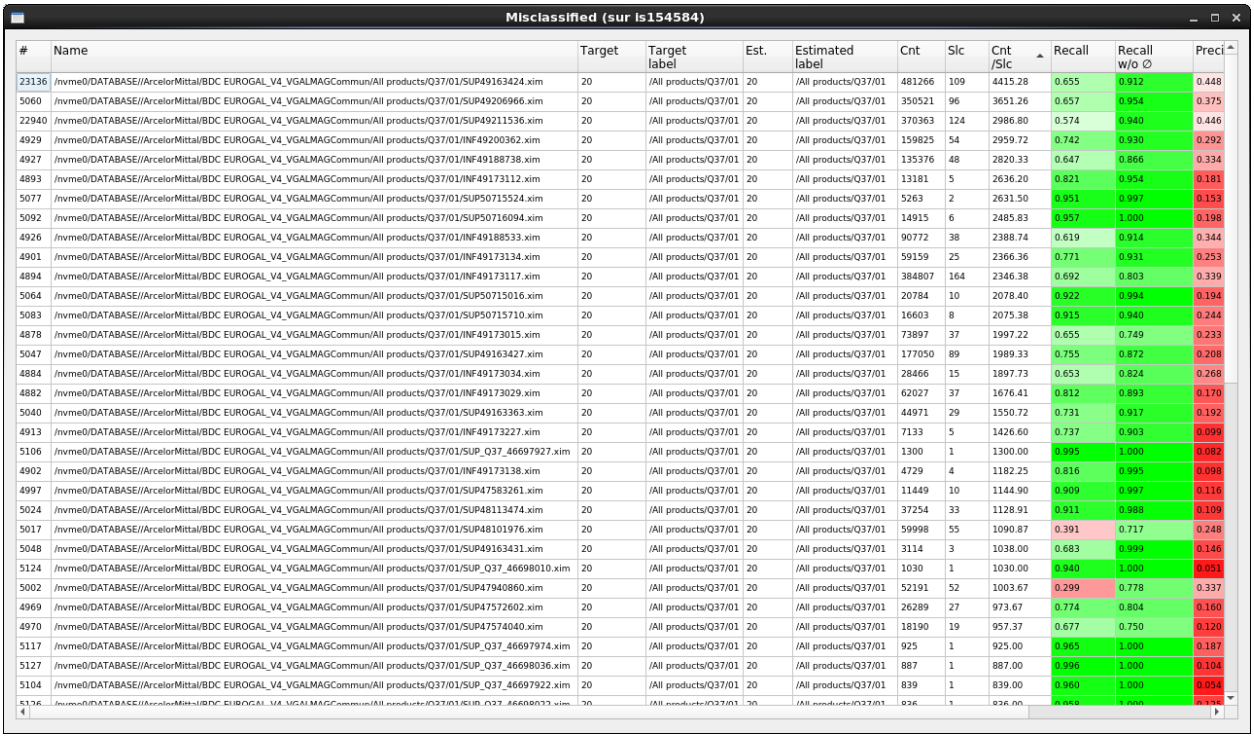


Fig. 5: Classes aggregation recall and precision.



#	Name	Target	Target label	Est.	Estimated label	Cnt	Slc	Cnt /Slc	Recall	Recall w/o ∅	Preci
23136	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/SUP49163424.xim	20	/All products/Q37/01	20	/All products/Q37/01	481266	109	4415.28	0.655	0.912	0.448
5060	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/SUP49206966.xim	20	/All products/Q37/01	20	/All products/Q37/01	350521	96	3651.26	0.657	0.954	0.375
22940	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/SUP49211536.xim	20	/All products/Q37/01	20	/All products/Q37/01	370363	124	2986.80	0.574	0.940	0.446
4929	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/INF49200362.xim	20	/All products/Q37/01	20	/All products/Q37/01	159825	54	2959.72	0.742	0.930	0.292
4927	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/INF49188738.xim	20	/All products/Q37/01	20	/All products/Q37/01	135376	48	2820.33	0.647	0.866	0.334
4893	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/INF49173112.xim	20	/All products/Q37/01	20	/All products/Q37/01	13181	5	2636.20	0.821	0.954	0.181
5077	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/SUP50715524.xim	20	/All products/Q37/01	20	/All products/Q37/01	5263	2	2631.50	0.951	0.997	0.153
5092	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/SUP50716094.xim	20	/All products/Q37/01	20	/All products/Q37/01	14915	6	2485.83	0.957	1.000	0.198
4926	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/INF49188533.xim	20	/All products/Q37/01	20	/All products/Q37/01	90772	38	2388.74	0.619	0.914	0.344
4901	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/INF49173134.xim	20	/All products/Q37/01	20	/All products/Q37/01	59159	25	2366.36	0.771	0.931	0.253
4894	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/INF49173117.xim	20	/All products/Q37/01	20	/All products/Q37/01	384807	164	2346.38	0.692	0.803	0.339
5064	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/SUP50715016.xim	20	/All products/Q37/01	20	/All products/Q37/01	20784	10	2078.40	0.922	0.994	0.194
5083	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/SUP50715710.xim	20	/All products/Q37/01	20	/All products/Q37/01	16603	8	2075.38	0.915	0.940	0.244
4878	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/INF49173015.xim	20	/All products/Q37/01	20	/All products/Q37/01	73897	37	1997.22	0.655	0.749	0.233
5047	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/SUP49163427.xim	20	/All products/Q37/01	20	/All products/Q37/01	177050	89	1989.33	0.755	0.872	0.208
4884	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/INF49173034.xim	20	/All products/Q37/01	20	/All products/Q37/01	28466	15	1897.73	0.653	0.824	0.268
4882	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/INF49173029.xim	20	/All products/Q37/01	20	/All products/Q37/01	62027	37	1676.41	0.812	0.893	0.170
5040	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/SUP49163363.xim	20	/All products/Q37/01	20	/All products/Q37/01	44971	29	1550.72	0.731	0.917	0.192
4913	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/INF49173227.xim	20	/All products/Q37/01	20	/All products/Q37/01	7133	5	1426.60	0.737	0.903	0.099
5106	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/SUP_Q37_46697927.xim	20	/All products/Q37/01	20	/All products/Q37/01	1300	1	1300.00	0.995	1.000	0.082
4902	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/INF49173138.xim	20	/All products/Q37/01	20	/All products/Q37/01	4729	4	1182.25	0.816	0.995	0.098
4997	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/SUP47583261.xim	20	/All products/Q37/01	20	/All products/Q37/01	11449	10	1144.90	0.909	0.997	0.116
5024	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/SUP48113474.xim	20	/All products/Q37/01	20	/All products/Q37/01	37254	33	1128.91	0.911	0.988	0.109
5017	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/SUP48101976.xim	20	/All products/Q37/01	20	/All products/Q37/01	59998	55	1090.87	0.391	0.717	0.248
5048	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/SUP49163431.xim	20	/All products/Q37/01	20	/All products/Q37/01	3114	3	1038.00	0.683	0.999	0.146
5124	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/SUP_Q37_46698010.xim	20	/All products/Q37/01	20	/All products/Q37/01	1030	1	1030.00	0.940	1.000	0.051
5002	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/SUP47940860.xim	20	/All products/Q37/01	20	/All products/Q37/01	52191	52	1003.67	0.299	0.778	0.337
4969	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/SUP47572602.xim	20	/All products/Q37/01	20	/All products/Q37/01	26289	27	973.67	0.774	0.804	0.160
4970	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/SUP47574040.xim	20	/All products/Q37/01	20	/All products/Q37/01	18190	19	957.37	0.677	0.750	0.120
5117	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/SUP_Q37_46697974.xim	20	/All products/Q37/01	20	/All products/Q37/01	925	1	925.00	0.965	1.000	0.187
5127	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/SUP_Q37_46698036.xim	20	/All products/Q37/01	20	/All products/Q37/01	887	1	887.00	0.996	1.000	0.104
5104	/nvme0/DATABASE//ArcelorMittal/BDC EUROGAL_V4_VGALMAGCommun/All products/Q37/01/SUP_Q37_46697922.xim	20	/All products/Q37/01	20	/All products/Q37/01	839	1	839.00	0.960	1.000	0.054

Fig. 6: List of confusion for selected cells in the confusion matrix.

4.2.5 Items viewer

When double-clicking on a row in the selected items table view, the ScoreTune viewer is opened for the corresponding image, showing the estimated classes in the image, as shown below. The F1 key allows to switch between the estimated classes and the annotations in the ScoreTune viewer.

Automatic Performances Report Generation It is possible to copy selections from the confusion matrix table or the selected items table using the CTRL + c keyboard shortcut. Selections can be pasted to Microsoft Word or LibreOffice Writer keeping the tabular formatting from the interactive viewer. When copying rows from the selected items table, the corresponding images with the target and estimated annotation are copied as well for each row.

In addition to the copy and paste feature, full report can be generated automatically from the confusion matrix, using the F1 key in the main window. Parameters for the report generation can be modified directly in the Python script and are described below:

4.3 Automatic Performances Report Generation

N2D2-IP only: available upon request.

It is possible to copy selections from the confusion matrix table or the selected items table using the CTRL + c keyboard shortcut. Selections can be pasted to Microsoft Word or LibreOffice Writer keeping the tabular formatting from the interactive viewer. When copying rows from the selected items table, the corresponding images with the target and estimated annotation are copied as well for each row.

In addition to the copy and paste feature, full report can be generated automatically from the confusion matrix, using the F1 key in the main window. Parameters for the report generation can be modified directly in the Python script and are described below:

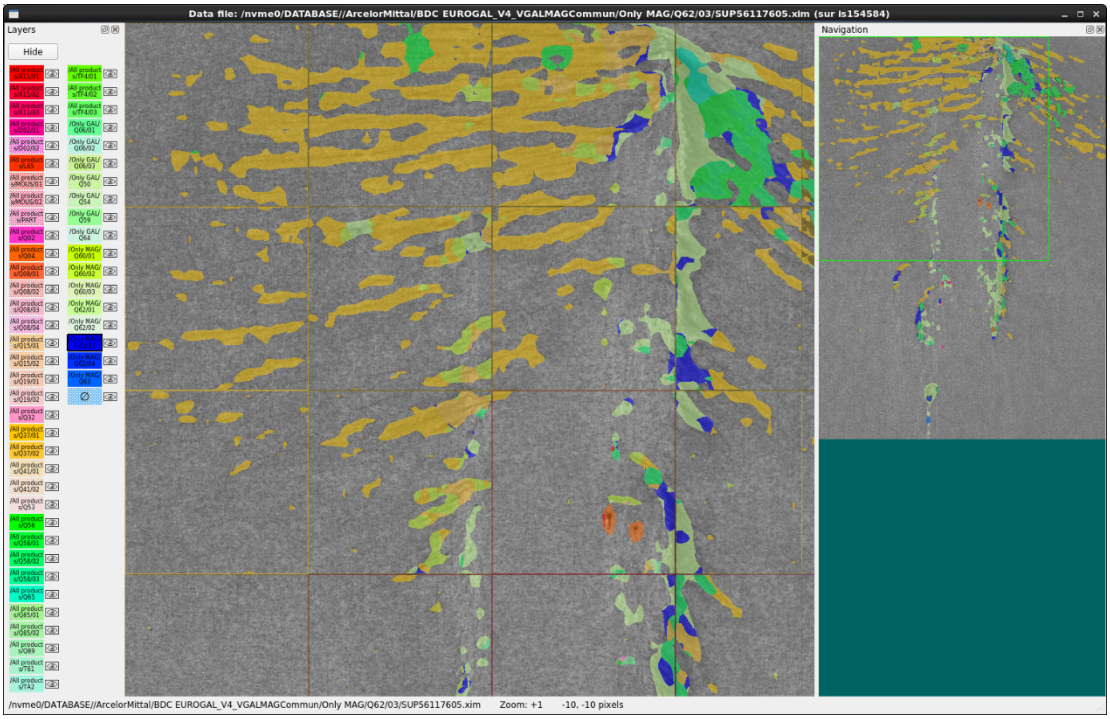


Fig. 7: Items viewer (using ScoreTune).

Parameter	Default value	Description
nbMainHits	3	Maximum number of items to include for the main hits
thresMainHits	1000.0	Threshold in count/slice for matching target and estimated items to consider a main hit
nbMainConfusions	10	Maximum number of items to include for the main confusions
thresMainConfusions	100.0	Threshold in count/slice for mismatching target and estimated items to consider a main confusion
nbMainMisses	10	Maximum number of items to include for the main misses
thresMainMisses	1000.0	Threshold in count/slice for mismatching target and Ø items to consider a main miss

CHAPTER

FIVE

TUTORIALS

5.1 Learning deep neural networks: tips and tricks

5.1.1 Choose the learning solver

Generally, you should use the SGD solver with a momemtum (typical value for the momentum: 0.9). It generalizes better, often significantly better, than adaptive methods like Adam [WilsonRoelofsStern+17].

Adaptive solvers, like Adam, may be used for fast exploration and prototyping, thanks to their fast convergence.

5.1.2 Choose the learning hyper-parameters

You can use the `-find-lr` option available in the `n2d2` executable to automatically find the best learning rate for a given neural network.

Usage example:

```
./n2d2 model.ini -find-lr 10000
```

This command starts from a very low learning rate (1.0e-6) and increase it exponentially to reach the maximum value (10.0) after 10000 steps, as shown in figure [fig:findLrRange]. The loss change during this phase is then plotted in function of the learning rate, as shown in figure [fig:findLr].

Note that in N2D2, the learning rate is automatically normalized by the global batch size ($N \times$ IterationSize) for the `SGDSolver`. A simple linear scaling rule is used, as recommanded in [GDollarG+17]. The effective learning rate α_{eff} applied for parameters update is therefore:

$$\alpha_{\text{eff}} = \frac{\alpha}{N \times \text{ IterationSize}} \text{ with } \alpha = \text{LearningRate}$$

Typical values for the `SGDSolver` are:

```
Solvers.LearningRate=0.01
Solvers.Decay=0.0001
Solvers.Momentum=0.9
```

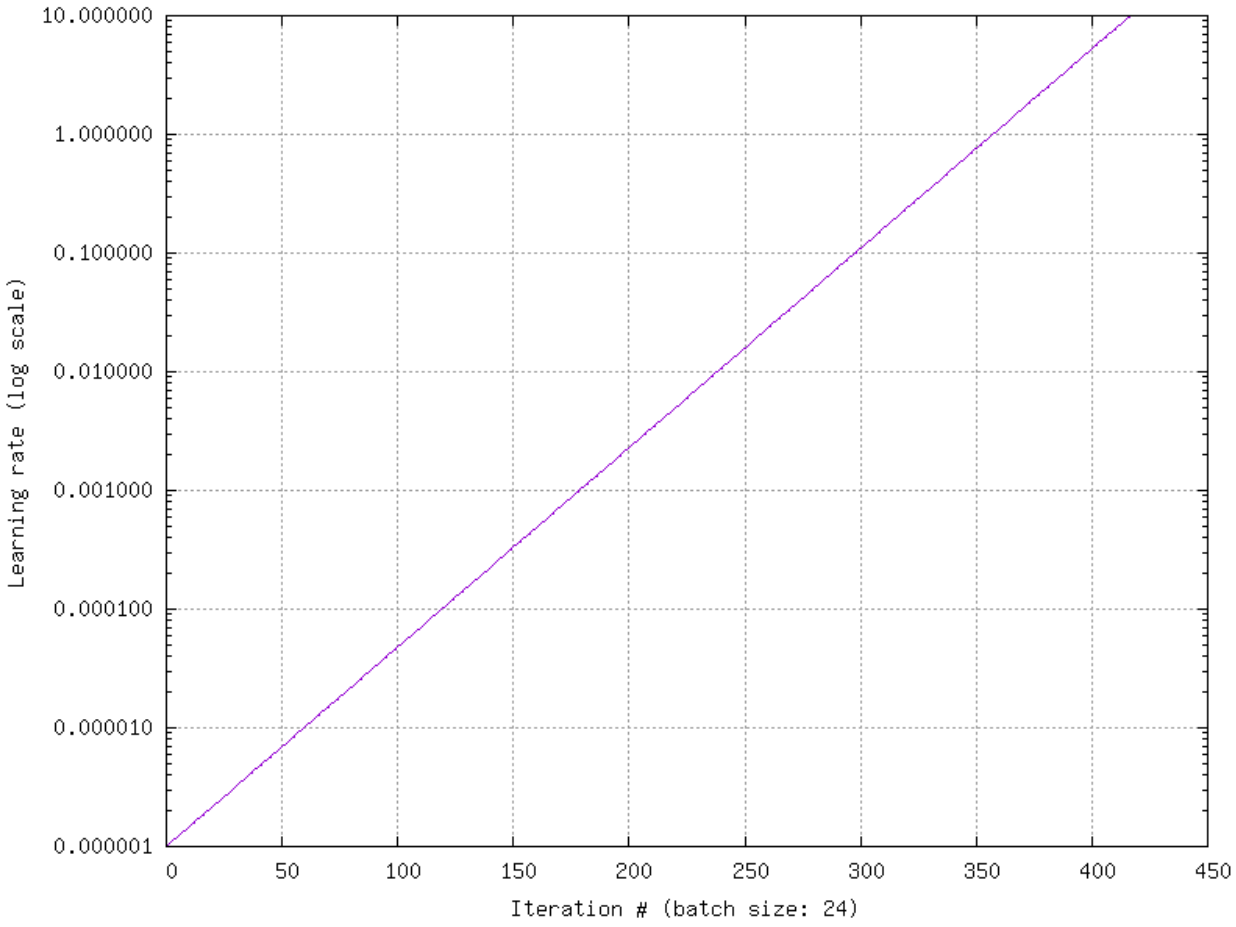


Fig. 1: Exponential increase of the learning rate over the specified number of iterations, equals to the number of steps divided by the batch size (here: 24).

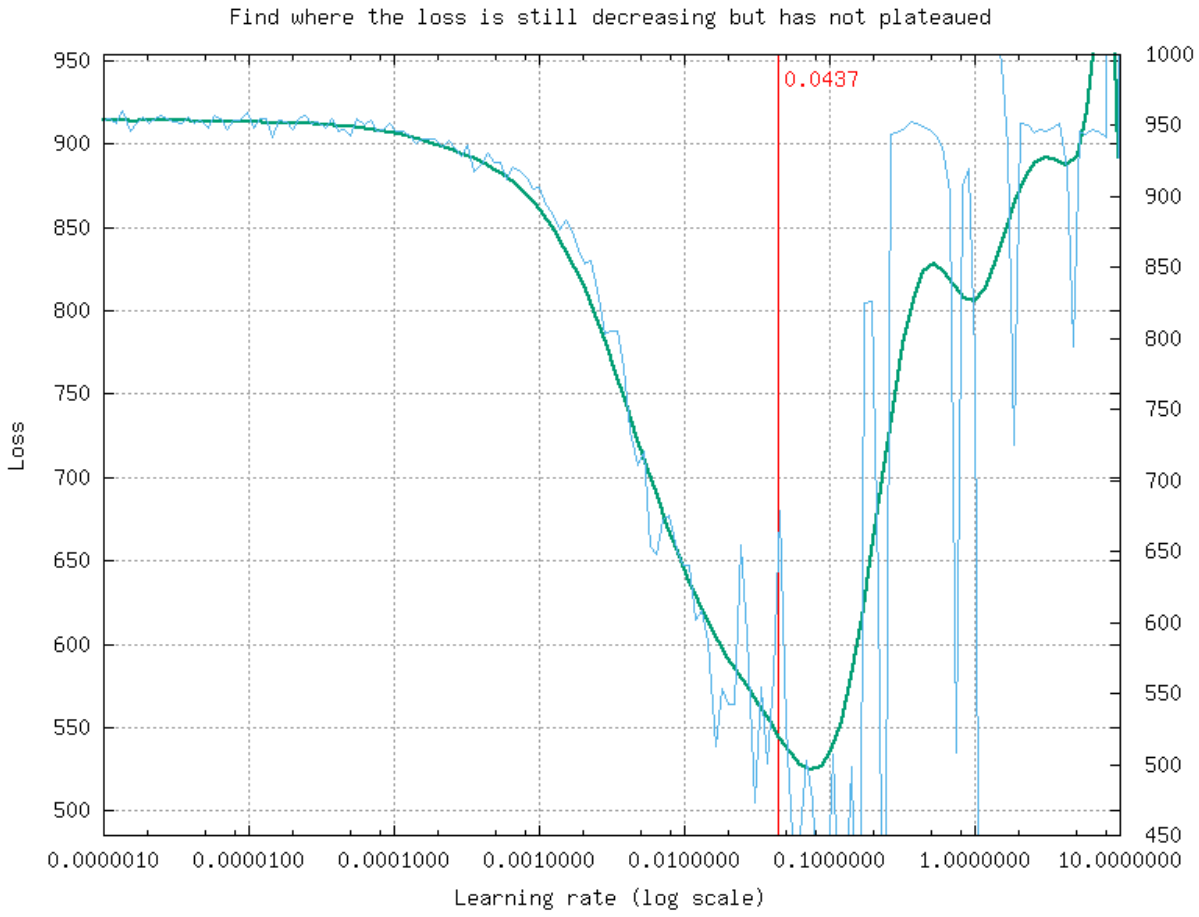


Fig. 2: Loss change as a function of the learning rate.

5.1.3 Convergence and normalization

Deep networks (> 30 layers) and especially residual networks usually don't converge without normalization. Indeed, batch normalization is almost always used. *ZeroInit* is a method that can be used to overcome this issue without normalization [ZDM19].

5.2 Building a classifier neural network

For this tutorial, we will use the classical MNIST handwritten digit dataset. A driver module already exists for this dataset, named `MNIST_IDX_Database`.

To instantiate it, just add the following lines in a new INI file:

```
[database]
Type=MNIST_IDX_Database
Validation=0.2 ; Use 20\% of the dataset for validation
```

In order to create a neural network, we first need to define its input, which is declared with a `[sp]` section (*sp* for *StimuliProvider*). In this section, we configure the size of the input and the batch size:

```
[sp]
SizeX=32
SizeY=32
BatchSize=128
```

We can also add pre-processing transformations to the *StimuliProvider*, knowing that the final data size after transformations must match the size declared in the `[sp]` section. Here, we must rescale the MNIST 28x28 images to match the 32x32 network input size.

```
[sp.Transformation_1]
Type=RescaleTransformation
Width=[sp]SizeX
Height=[sp]SizeY
```

Next, we declare the neural network layers. In this example, we reproduced the well-known LeNet network. The first layer is a 5x5 convolutional layer, with 6 channels. Since there is only one input channel, there will be only 6 convolution kernels in this layer.

```
[conv1]
Input=sp
Type=Conv
KernelWidth=5
KernelHeight=5
NbOutputs=6
```

The next layer is a 2x2 MAX pooling layer, with a stride of 2 (non-overlapping MAX pooling).

```
[pool1]
Input=conv1
Type=Pool
PoolWidth=2
PoolHeight=2
NbOutputs=[conv1]NbOutputs
```

(continues on next page)

(continued from previous page)

```
Stride=2
Pooling=Max
Mapping.Size=1 ; One to one connection between input and output channels
```

The next layer is a 5x5 convolutional layer with 16 channels.

```
[conv2]
Input=pool1
Type=Conv
KernelWidth=5
KernelHeight=5
NbOutputs=16
```

Note that in LeNet, the `[conv2]` layer is not fully connected to the pooling layer. In N2D2, a custom mapping can be defined for each input connection. The connection of n-th output map to the inputs is defined by the n-th column of the matrix below, where the rows correspond to the inputs.

```
Mapping(pool1)=\
1 0 0 0 1 1 1 0 0 1 1 1 1 0 1 1 \
1 1 0 0 0 1 1 1 0 0 1 1 1 1 0 1 \
1 1 1 0 0 0 1 1 1 0 0 1 0 1 1 1 \
0 1 1 1 0 0 1 1 1 1 0 0 1 0 1 1 \
0 0 1 1 1 0 0 1 1 1 1 0 1 1 0 1 \
0 0 0 1 1 1 0 0 1 1 1 1 0 1 1 1
```

Another MAX pooling and convolution layer follow:

```
[pool2]
Input=conv2
Type=Pool
PoolWidth=2
PoolHeight=2
NbOutputs=[conv2]NbOutputs
Stride=2
Pooling=Max
Mapping.Size=1

[conv3]
Input=pool2
Type=Conv
KernelWidth=5
KernelHeight=5
NbOutputs=120
```

The network is composed of two fully-connected layers of 84 and 10 neurons respectively:

```
[fc1]
Input=conv3
Type=Fc
NbOutputs=84

[fc2]
Input=fc1
```

(continues on next page)

(continued from previous page)

```
Type=Fc
NbOutputs=10
```

Finally, we use a softmax layer to obtain output classification probabilities and compute the loss function.

```
[softmax]
Input=fc2
Type=Softmax
NbOutputs=[fc2]NbOutputs
WithLoss=1
```

In order to tell N2D2 to compute the error and the classification score on this softmax layer, one must attach a N2D2 *Target* to this layer, with a section with the same name suffixed with `.Target`:

```
[softmax.Target]
```

By default, the activation function for the convolution and the fully-connected layers is the hyperbolic tangent. Because the `[fc2]` layer is fed to a softmax, it should not have any activation function. We can specify it by adding the following line in the `[fc2]` section:

```
[fc2]
...
ActivationFunction=Linear
```

In order to improve further the networks performances, several things can be done:

Use ReLU activation functions. In order to do so, just add the following in the `[conv1]`, `[conv2]`, `[conv3]` and `[fc1]` layer sections:

```
ActivationFunction=Rectifier
```

For the ReLU activation function to be effective, the weights must be initialized carefully, in order to avoid dead units that would be stuck in the $]-\infty, 0]$ output range before the ReLU function. In N2D2, one can use a custom `WeightsFiller` for the weights initialization. For the ReLU activation function, a popular and efficient filler is the so-called `XavierFiller` (see the [par:XavierFiller] section for more information):

```
WeightsFiller=XavierFiller
```

Use dropout layers. Dropout is highly effective to improve the network generalization capacity. Here is an example of a dropout layer inserted between the `[fc1]` and `[fc2]` layers:

```
[fc1]
...

[fc1.drop]
Input=fc1
Type=Dropout
NbOutputs=[fc1]NbOutputs

[fc2]
Input=fc1.drop ; Replaces "Input=fc1"
...
```

Tune the learning parameters. You may want to tune the learning rate and other learning parameters depending on

the learning problem at hand. In order to do so, you can add a configuration section that can be common (or not) to all the layers. Here is an example of configuration section:

```
[conv1]
...
ConfigSection=common.config

[...]
...

[common.config]
NoBias=1
WeightsSolver.LearningRate=0.05
WeightsSolver.Decay=0.0005
Solvers.LearningRatePolicy=StepDecay
Solvers.LearningRateStepSize=[sp]_EpochSize
Solvers.LearningRateDecay=0.993
Solvers.Clamping=-1.0:1.0
```

For more details on the configuration parameters for the `Solver`, see section [sec:WeightSolvers].

Add input distortion. See for example the `DistortionTransformation` (section [par:DistortionTransformation]).

The complete INI model corresponding to this tutorial can be found in *models/LeNet.ini*.

In order to use CUDA/GPU accelerated learning, the default layer model should be switched to `Frame_CUDA`. You can enable this model by adding the following line at the top of the INI file (before the first section):

```
DefaultModel=Frame_CUDA
```

5.3 Building a segmentation neural network

In this tutorial, we will learn how to do image segmentation with N2D2. As an example, we will implement a face detection and gender recognition neural network, using the IMDB-WIKI dataset.

First, we need to instanciate the IMDB-WIKI dataset built-in N2D2 driver:

```
[database]
Type=IMDBWIKI_Database
WikiSet=1 ; Use the WIKI part of the dataset
IMDBSet=0 ; Don't use the IMDB part (less accurate annotation)
Learn=0.90
Validation=0.05
DefaultLabel=background ; Label for pixels outside any ROI (default is no label, pixels␣
↪are ignored)
```

We must specify a default label for the background, because we want to learn to differenciate faces from the background (and not simply ignore the background for the learning).

The network input is then declared:

```
[sp]
SizeX=480
SizeY=360
```

(continues on next page)

(continued from previous page)

```
BatchSize=48
CompositeStimuli=1
```

In order to work with segmented data, i.e. data with bounding box annotations or pixel-wise annotations (as opposed to a single label per data), one must enable the `CompositeStimuli` option in the `[sp]` section.

We can then perform various operations on the data before feeding it to the network, like for example converting the 3-channels RGB input images to single-channel gray images:

```
[sp.Transformation-1]
Type=ChannelExtractionTransformation
CSChannel=Gray
```

We must only rescale the images to match the networks input size. This can be done using a `RescaleTransformation`, followed by a `PadCropTransformation` if one want to keep the images aspect ratio.

```
[sp.Transformation-2]
Type=RescaleTransformation
Width=[sp]SizeX
Height=[sp]SizeY
KeepAspectRatio=1 ; Keep images aspect ratio

; Required to ensure all the images are the same size
[sp.Transformation-3]
Type=PadCropTransformation
Width=[sp]SizeX
Height=[sp]SizeY
```

A common additional operation to extend the learning set is to apply random horizontal mirror to images. This can be achieved with the following `FlipTransformation`:

```
[sp.OnTheFlyTransformation-4]
Type=FlipTransformation
RandomHorizontalFlip=1
ApplyTo=LearnOnly ; Apply this transformation only on the learning set
```

Note that this is an *on-the-fly* transformation, meaning it cannot be cached and is re-executed every time even for the same stimuli. We also apply this transformation only on the learning set, with the `ApplyTo` option.

Next, the neural network can be described:

```
[conv1.1]
Input=sp
Type=Conv
...

[pool1]
...

[...]
...

[fc2]
Input=drop1
```

(continues on next page)

(continued from previous page)

```
Type=Conv
...

[drop2]
Input=fc2
Type=Dropout
NbOutputs=[fc2]NbOutputs
```

A full network description can be found in the *IMDBWIKI.ini* file in the *models* directory of N2D2. It is a fully-CNN network.

Here we will focus on the output layers required to detect the faces and classify their gender. We start from the `[drop2]` layer, which has 128 channels of size 60x45.

5.3.1 Faces detection

We want to first add an output stage for the faces detection. It is a 1x1 convolutional layer with a single 60x45 output map. For each output pixel, this layer outputs the probability that the pixel belongs to a face.

```
[fc3.face]
Input=drop2
Type=Conv
KernelWidth=1
KernelHeight=1
NbOutputs=1
Stride=1
ActivationFunction=LogisticWithLoss
WeightsFiller=XavierFiller
ConfigSection=common.config ; Same solver options that the other layers
```

In order to do so, the activation function of this layer must be of type `LogisticWithLoss`.

We must also tell N2D2 to compute the error and the classification score on this softmax layer, by attaching a N2D2 *Target* to this layer, with a section with the same name suffixed with `.Target`:

```
[fc3.face.Target]
LabelsMapping=\${N2D2_MODELS}/IMDBWIKI_target_face.dat
; Visualization parameters
NoDisplayLabel=0
LabelsHueOffset=90
```

In this *Target*, we must specify how the dataset annotations are mapped to the layer's output. This can be done in a separate file using the `LabelsMapping` parameter. Here, since the output layer has a single output per pixel, the target value can only be 0 or 1. A target value of -1 means that this output is ignored (no error back-propagated). Since the only annotations in the IMDB-WIKI dataset are faces, the mapping described in the *IMDBWIKI_target_face.dat* file is easy:

```
# background
background 0

# padding (*) is ignored (-1)
* -1
```

(continues on next page)

(continued from previous page)

```
# not background = face
default 1
```

5.3.2 Gender recognition

We can also add a second output stage for gender recognition. Like before, it would be a 1x1 convolutional layer with a single 60x45 output map. But here, for each output pixel, this layer would output the probability that the pixel represents a female face.

```
[fc3.gender]
Input=drop2
Type=Conv
KernelWidth=1
KernelHeight=1
NbOutputs=1
Stride=1
ActivationFunction=LogisticWithLoss
WeightsFiller=XavierFiller
ConfigSection=common.config
```

The output layer is therefore identical to the face's output layer, but the target mapping is different. For the target mapping, the idea is simply to ignore all pixels not belonging to a face and affect the target 0 to male pixels and the target 1 to female pixels.

```
[fc3.gender.Target]
LabelsMapping=\${N2D2_MODELS}/IMDBWIKI_target_gender.dat
; Only display gender probability for pixels detected as face pixels
MaskLabelTarget=fc3.face.Target
MaskedLabel=1
```

The content of the *IMDBWIKI_target_gender.dat* file would therefore look like:

```
# background
# ?-* (unknown gender)
# padding
default -1

# male gender
M-? 0   # unknown age
M-0 0
M-1 0
M-2 0
...
M-98 0
M-99 0

# female gender
F-? 1   # unknown age
F-0 1
F-1 1
F-2 1
```

(continues on next page)

(continued from previous page)

```
...
F-98 1
F-99 1
```

5.3.3 ROIs extraction

The next step would be to extract detected face ROIs and assign for each ROI the most probable gender. To this end, we can first set a detection threshold, in terms of probability, to select face pixels. In the following, the threshold is fixed to 75% face probability:

```
[post.Transformation-thres]
Input=fc3.face
Type=Transformation
NbOutputs=1
Transformation=ThresholdTransformation
Operation=ToZero
Threshold=0.75
```

We can then assign a target of type `TargetROIs` to this layer that will automatically create the bounding box using a segmentation algorithm.

```
[post.Transformation-thres.Target-face]
Type=TargetROIs
MinOverlap=0.33 ; Min. overlap fraction to match the ROI to an annotation
FilterMinWidth=5 ; Min. ROI width
FilterMinHeight=5 ; Min. ROI height
FilterMinAspectRatio=0.5 ; Min. ROI aspect ratio
FilterMaxAspectRatio=1.5 ; Max. ROI aspect ratio
LabelsMapping=\${N2D2_MODELS}/IMDBWIKI_target_face.dat
```

In order to assign a gender to the extracted ROIs, the above target must be modified to:

```
[post.Transformation-thres.Target-gender]
Type=TargetROIs
ROIsLabelTarget=fc3.gender.Target
MinOverlap=0.33
FilterMinWidth=5
FilterMinHeight=5
FilterMinAspectRatio=0.5
FilterMaxAspectRatio=1.5
LabelsMapping=\${N2D2_MODELS}/IMDBWIKI_target_gender.dat
```

Here, we use the `fc3.gender.Target` target to determine the most probable gender of the ROI.

5.3.4 Data visualization

For each *Target* in the network, a corresponding folder is created in the simulation directory, which contains learning, validation and test confusion matrixes. The output estimation of the network for each stimulus is also generated automatically for the test dataset and can be visualized with the *./test.py* helper tool. An example is shown in figure [fig:targetvisu].

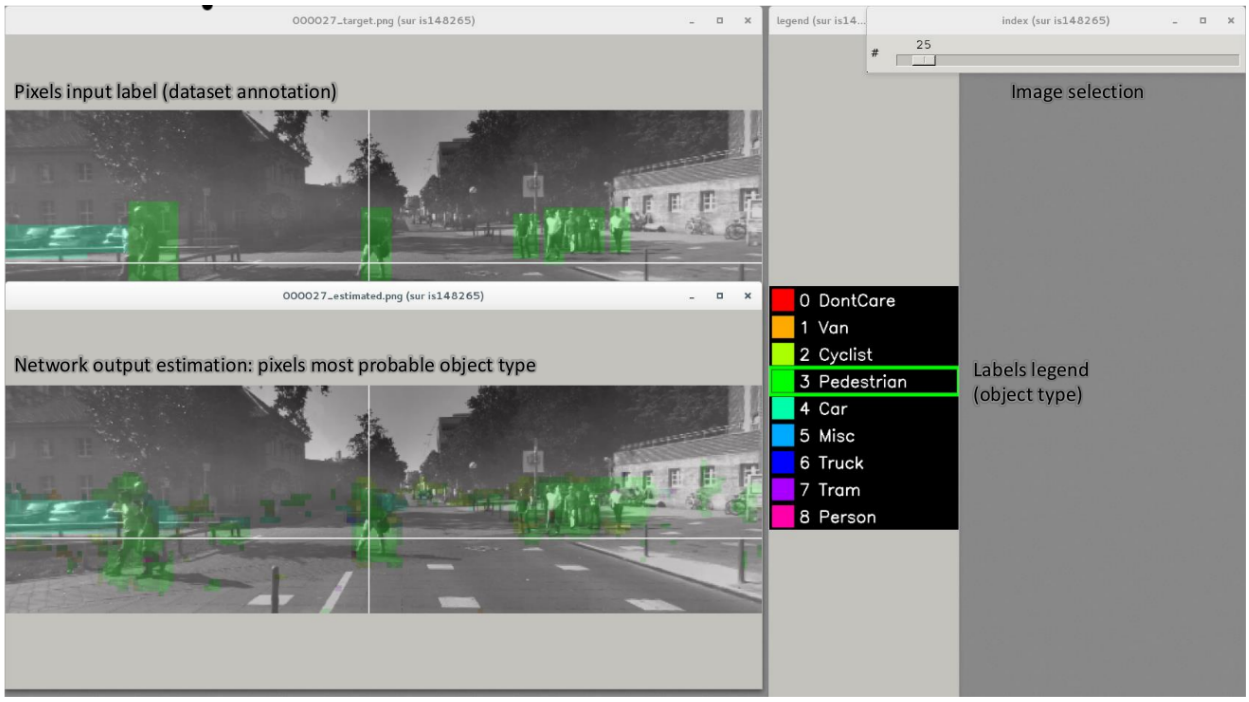


Fig. 3: Example of the target visualization helper tool.

5.4 Transcoding a learned network in spike-coding

N2D2 embeds an event-based simulator (historically known as 'Xnet') and allows to transcode a whole DNN in a spike-coding version and evaluate the resulting spiking neural network performances. In this tutorial, we will transcode the LeNet network described in section [sec:BuildingClassifierNN].

5.4.1 Render the network compatible with spike simulations

The first step is to specify that we want to use a transcode model (allowing both formal and spike simulation of the same network), by changing the `DefaultModel` to:

```
DefaultModel=Transcode_CUDA
```

In order to perform spike simulations, the input of the network must be of type *Environment*, which is a derived class of *StimuliProvider* that adds spike coding support. In the INI model file, it is therefore necessary to replace the `[sp]` section by an `[env]` section and replace all references of `sp` to `env`.

Note that these changes have at this point no impact at all on the formal coding simulations. The beginning of the INI file should be:

```
DefaultModel=!\color{red}{Transcode\_CUDA}!

; Database
[database]
Type=MNIST_IDX_Database
Validation=0.2 ; Use 20% of the dataset for validation

; Environment
[!\color{red}{env}!]
SizeX=32
SizeY=32
BatchSize=128

[env.Transformation_1]
Type=RescaleTransformation
Width=[!\color{red}{env}!]SizeX
Height=[!\color{red}{env}!]SizeY

[conv1]
Input=!\color{red}{env}!
...
```

The dropout layer has no equivalence in spike-coding inference and must be removed:

```
...
!\color{red}{\st{[fc1.drop]}}!
!\color{red}{\st{Input=fc1}}!
!\color{red}{\st{Type=Dropout}}!
!\color{red}{\st{NbOutputs=[fc1]NbOutputs}}!

[fc2]
Input=fc1!\color{red}{\st{.drop}}!
...
```

The softmax layer has no equivalence in spike-coding inference and must be removed as well. The *Target* must therefore be attached to `[fc2]`:

```
...
!\color{red}{\st{[softmax]}}!
!\color{red}{\st{Input=fc2}}!
!\color{red}{\st{Type=Softmax}}!
!\color{red}{\st{NbOutputs=[fc2]NbOutputs}}!
!\color{red}{\st{WithLoss=1}}!

!\color{red}{\st{[softmax.Target]}}!

[fc2.Target]
...
```

The network is now compatible with spike-coding simulations. However, we did not specify at this point how to translate the input stimuli data into spikes, nor the spiking neuron parameters (threshold value, leak time constant…).

5.4.2 Configure spike-coding parameters

The first step is to configure how the input stimuli data must be coded into spikes. To this end, we must attach a configuration section to the *Environment*. Here, we specify a periodic coding with random initial jitter with a minimum period of 10 ns and a maximum period of 100 us:

```
...
ConfigSection=env.config

[env.config]
; Spike-based computing
StimulusType=JitteredPeriodic
PeriodMin=1,000,000 ; unit = fs
PeriodMeanMin=10,000,000 ; unit = fs
PeriodMeanMax=100,000,000,000 ; unit = fs
PeriodRelStdDev=0.0
```

The next step is to specify the neurons parameters, that will be common to all layers and can therefore be specified in the `[common.config]` section. In N2D2, the base spike-coding layers use a Leaky Integrate-and-Fire (LIF) neuron model. By default, the leak time constant is zero, resulting to simple Integrate-and-Fire (IF) neurons.

Here we simply specify that the neurons threshold must be the unity, that the threshold is only positive and that there is no incoming synaptic delay:

```
...
; Spike-based computing
Threshold=1.0
BipolarThreshold=0
IncomingDelay=0
```

Finally, we can limit the number of spikes required for the computation of each stimulus by adding a decision delta threshold at the output layer:

```
...
ConfigSection=common.config,fc2.config

[fc2.Target]

[fc2.config]
; Spike-based computing
TerminateDelta=4
BipolarThreshold=1
```

The complete INI model corresponding to this tutorial can be found in *models/LeNet_Spike.ini*.

Here is a summary of the steps required to reproduce the whole experiment:

```
./n2d2 "\$N2D2_MODELS/LeNet.ini" -learn 6000000 -log 100000
./n2d2 "\$N2D2_MODELS/LeNet_Spike.ini" -test
```

The final recognition rate reported at the end of the spike inference should be almost identical to the formal coding network (around 99% for the LeNet network).

Various statistics are available at the end of the spike-coding simulation in the *stats_spike* folder and the *stats_spike.log* file. Looking in the *stats_spike.log* file, one can read the following line towards the end of the file:

```
Read events per virtual synapse per pattern (average): 0.654124
```

This line reports the average number of accumulation operations per synapse per input stimulus in the network. If this number if below 1.0, it means that the spiking version of the network is more efficient than its formal counterpart in terms of total number of operations!

CHAPTER

SIX

OBTAIN ONNX MODELS

6.1 Convert from PyTorch

ONNX conversion is natively supported in PyTorch with the `torch.onnx.export` function. An example of a pre-trained PyTorch model conversion to ONNX is provided in `tools/pytorch_to_onnx.py`:

```
import torch
from MobileNetV2 import mobilenet_v2

dummy_input = torch.randn(10, 3, 224, 224)
model = mobilenet_v2(pretrained=True)

input_names = [ "input" ]
output_names = [ "output" ]

torch.onnx.export(model, dummy_input, "mobilenet_v2_pytorch.onnx", verbose=True, input_
↪names=input_names, output_names=output_names)
```

6.2 Convert from TF/Keras

ONNX conversion is not natively supported by TF/Keras. Instead, a third-party tool must be used, like `keras2onnx` or `tf2onnx`. Currently, the `tf2onnx` is the most active and most maintained solution.

The `tf2onnx` tool can be used in command line, by providing a TensorFlow frozen graph (.pb).

Note: Make sure to use the option `--inputs-as-nchw` on the model input(s) because N2D2 expects NCHW inputs, but the default format in TF/Keras is NHWC. Otherwise you would typically get an error like:

```
Error: Unexpected size for ONNX input "conv2d_77_input": got 3 224 224  , but␣
↪StimuliProvider provides 224 224 3
```

The format of the exported ONNX graph from TF/Keras will depend on the execution platform (CPU or GPU). The default format is NHWC on CPU and NCHW on GPU. ONNX mandates the NCHW format for the operators, so exporting an ONNX model on CPU can result in the insertion of many `Transpose` operations in the graph before and after other operators.

```
tfmodel=mobilenet_v1_1.0_224_frozen.pb
onnxmodel=mobilenet_v1_1.0_224.onnx
```

(continues on next page)

(continued from previous page)

```
url=http://download.tensorflow.org/models/mobilenet_v1_2018_08_02/mobilenet_v1_1.0_224.
↪tgz
tgz=$(basename $url)

if [ ! -r $tgz ]; then
    wget  -q  $url
    tar zxvf $tgz
fi
python3 -m tf2onnx.convert --input $tfmodel --output $onnxmodel \
    --opset 10 --verbose \
    --inputs-as-nchw input:0 \
    --inputs input:0 \
    --outputs MobilenetV1/Predictions/Reshape_1:0
```

Example conversion scripts are provided for the Mobilenet families: `tools/mobilenet_v1_to_onnx.sh`, `tools/mobilenet_v2_to_onnx.sh` and `tools/mobilenet_v3_to_onnx.sh`.

6.3 Download pre-trained models

Many already trained ONNX models are freely available and ready to use in the ONNX Model Zoo: https://github.com/onnx/models/blob/master/README.md

CHAPTER

SEVEN

IMPORT ONNX MODELS

7.1 Preliminary steps

ONNX generators may generate complicated models, in order to take into account for example dynamic size or shape calculation, from previous operator outputs dimensions. This can be the case even when the graph is static and the dimensions are known in the ONNX model. While such model may be imported in DL frameworks using standard operators/layers, it would be vastly sub-optimal, as some part of the graph would require unnecessary dynamic allocation, and would be very hard to optimize for inference on embedded platforms.

For this reason, we do not always try to allow proper import of such graph in N2D2 as is. While some simplifications may be handled directly in N2D2, we recommend using the ONNX Simplifier tool on your ONNX model before importing it into N2D2.

7.2 With an INI file

It is possible to include an ONNX model inside a N2D2 INI file, as part of a graph. This is particularly useful to add pre-processing and post-processing to an existing ONNX model. Below is an example with the MobileNet ONNX model provided by Google:

```
$BATCH_SIZE=256

DefaultModel=Frame_CUDA

; Database
[database]
Type=ILSVRC2012_Database
RandomPartitioning=0
Learn=1.0
BackgroundClass=1  ; Necessary for Google MobileNet pre-trained models

; Environment
[sp]
SizeX=224
SizeY=224
NbChannels=3
BatchSize=${BATCH_SIZE}

[sp.Transformation-1]
Type=RescaleTransformation
```

(continues on next page)

(continued from previous page)

```
Width=256
Height=256

[sp.Transformation-2]
Type=PadCropTransformation
Width=224
Height=224

[sp.Transformation-3]
Type=ColorSpaceTransformation
ColorSpace=RGB

[sp.Transformation-4]
Type=RangeAffineTransformation
FirstOperator=Minus
FirstValue=127.5
SecondOperator=Divides
SecondValue=127.5

; Here, we insert an ONNX graph in the N2D2 flow the same way as a regular Cell
[onnx]
Input=sp
Type=ONNX
File=mobilenet_v1_1.0_224.onnx

; We can add targets to ONNX cells
[MobilenetV1/Predictions/Softmax:0.Target-Top5]
TopN=5
```

A N2D2 target must be associated to the output layer of the ONNX model in order to compute the score in N2D2.

Note: The imported ONNX layer names in N2D2 is the name of their first output (the operator "name" field is indeed optional in the ONNX standard). You can easily find the ONNX cell names after running N2D2 or by opening the ONNX graph in a graph viewer like NETRON (https://lutzroeder.github.io/netron/).

Once the INI file including the ONNX model is ready, the following command must be used to run N2D2 in test (inference) mode:

```
n2d2 MobileNet_ONNX.ini -seed 1 -w /dev/null -test
```

There required command line arguments for running INI files including ONNX model are described above:

Command line argument	Description
`-seed 1`	Initial seed, necessary for test without learning before
`-w /dev/null`	No external weight loading: trained weight values are contained in the ONNX model

7.2.1 ONNX INI section type

The table below summarizes the parameters of an ONNX INI section. To declare an ONNX section, the `Type` parameter must be equal to `ONNX`. The name of the section can be arbitrary.

Option [default value]	Description
`Type=ONNX`	ONNX section type
`File`	Path to the ONNX file
`Ignore` []	Space-separated list of ONNX operators to ignore during import
`IgnoreInputSize` [0]	If true (1), the input size specified in the ONNX model is ignored and the N2D2 `StimuliProvider` size is used
`Transpose` [0]	If true (1), the first 2 dimensions are transposed in the whole ONNX graph (1D graph are first interpreted as 2D with the second dimension equal to 1)

7.2.2 `Transpose` option usage

The `Transpose` option allows to transpose the first two dimensions of a whole graph. This can be used in practice to used transposed inputs (like a transposed image, or a transposed vector for 1D signal inputs), like shown below:

```
[sp]
Size=8000 1 1
BatchSize=${BATCH_SIZE}

; Transpose the input:
[trans]
Input=sp
Type=Transpose
NbOutputs=1
Perm=1 0 2 3
; Output dimensions are now "1 8000 1 ${BATCH_SIZE}"

[onnx]
Input=trans
Type=ONNX
Transpose=1
; The graph originally expects an input dimension of "8000"
; After "Transpose=1", the expected input dimension becomes "1 8000"
File=sound_processing_graph.onnx
```

7.3 Supported operators

The table below summarizes the currently implemented ONNX operators:

Operator	Support	Remarks
Add	✓	
AveragePool	✓	Exc. *ceil_mode* and *count_include_pad* attributes
BatchNormalization	✓	
Cast	✗	Ignored

continues on next page

Table 1 – continued from previous page

Operator	Support	Remarks
Clip	✓	Only for *min* = 0 and *max* > 0
Concat	✓	Only for layers that support it
Constant	✓	In some contexts only
Conv	✓	
Dropout	✓	Exc. *mask* output
Div	✓	With constant second operand only
Flatten	✓	Ignored (not necessary)
Gemm	✓	Only for fully-connected layers
GlobalAveragePool	✓	
GlobalMaxPool	✓	
LRN	✓	
LeakyRelu	✓	
MatMul	✓	Only for fully-connected layers
Max	✓	
MaxPool	✓	Exc. *Indices* output
Mul	✓	
Pad	✓	
Relu	✓	
Reshape	✓	Only for fixed dimensions
Resize	✗	Planned (partially)
Shape	✗	Ignored
Sigmoid	✓	
Softmax	✓	
Softplus	✓	
Squeeze	✗	Ignored
Sub	✓	
Sum	✓	
Tanh	✓	
Transpose	✓	
Upsample	✗	Planned

CHAPTER

EIGHT

TRAIN FROM ONNX MODELS

The ONNX specification does not include any training parameter. To perform a training on an imported ONNX model, it is possible to add the training elements (solvers, learning rate scheduler...) on top of an ONNX model in N2D2, in the INI file directly or using the Python API.

This is particularly useful to perform transfer learning from an existing ONNX model trained on ImageNet for example.

8.1 With an INI file

We propose in this section to apply transfer learning to a MobileNet v1 ONNX model. We assume that this model is obtained by converting the reference pre-trained model from Google using the `tools/mobilenet_v1_to_onnx.sh` tool provided in N2D2. The resulting model file name is therefore assumed to be `mobilenet_v1_1.0_224.onnx`.

8.1.1 1) Remove the original classifier

The first step to perform transfer learning is to remove the existing classifier from the ONNX model. To do so, one can simply use the `Ignore` parameter in the ONNX INI section.

```
[onnx]
Input=sp
Type=ONNX
File=mobilenet_v1_1.0_224.onnx
; Remove the last layer and the softmax for transfer learning
Ignore=Conv__252:0 MobilenetV1/Predictions/Softmax:0
```

8.1.2 2) Add a new classifier to the ONNX model

The next step is to add a new classifier (fully connected layer with a softmax) and connect it to the last layer in the ONNX model.

In order to properly handle graph dependencies, all the N2D2 layers connected to a layer embedded in an ONNX model, must take the ONNX section name (here `onnx`) as first input in the `Input` parameter. The actual inputs are then added in the comma-separated list, which can mix ONNX and N2D2 layers. In the example below, the average pooling layer from the ONNX model is connected to the `Fc` cell:

```
; Here, we add our new layers for transfer learning
[fc]
; first input MUST BE "onnx"
; for proper dependency handling
```

(continues on next page)

(continued from previous page)

```
Input=onnx,MobilenetV1/Logits/AvgPool_1a/AvgPool:0
Type=Fc
NbOutputs=100
ActivationFunction=Linear
WeightsFiller=XavierFiller
ConfigSection=common.config

[softmax]
Input=fc
Type=Softmax
NbOutputs=[fc]NbOutputs
WithLoss=1
[softmax.Target]

; Common config for static model
[common.config]
WeightsSolver.LearningRate=0.01
WeightsSolver.Momentum=0.9
WeightsSolver.Decay=0.0005
Solvers.LearningRatePolicy=StepDecay
Solvers.LearningRateStepSize=[sp]_EpochSize
Solvers.LearningRateDecay=0.993
```

As this new classifier must be trained, all the training parameter must be specified as usual for this layer.

8.1.3 3) Fine tuning (optional)

If one wants to also fine-tune the existing ONNX layers, one must set the solver configuration for the ONNX layers, using default configuration sections.

Default configuration sections applies to all the layers of the same type in the ONNX model. For example, to add default parameters to all convolution layers in the ONNX model loaded in a section of type ONNX named `onnx`, just add a section named `[onnx:Conv_def]` in the INI file. The name of the default section follows the convention `[ONNXSection:N2D2CellType_def]`.

```
; Default section for ONNX Conv from section "onnx"
; "ConfigSection", solvers and fillers can be specified here...
[onnx:Conv_def]
ConfigSection=common.config

; Default section for ONNX Fc from section "onnx"
[onnx:Fc_def]
ConfigSection=common.config

; For BatchNorm, make sure the stats won't change if there is no fine-tuning
[onnx:BatchNorm_def]
ConfigSection=bn_notrain.config
[bn_notrain.config]
MovingAverageMomentum=0.0
```

Note: Important: make sure that the BatchNorm stats does not change if the BatchNorm layer are not fine-tuned! This

can be done by setting the parameter `MovingAverageMomentum` to 0.0 for the layer than must not be fine-tuned.

It is possible to add parameters for a specific ONNX layer by adding a section with the ONNX layer named.

You can fine-tune the whole network or only some of its layers, usually the last ones. To stop the fine-tuning at a specific layer, one can simply prevent the gradient from back-propagating further. This can be achieved with the `BackPropagate=0` configuration parameter.

```
[Conv__250]
ConfigSection=common.config,notrain.config
[notrain.config]
BackPropagate=0
```

For the full configuration related to this example and more information, have a look in `models/MobileNet_v1_ONNX_transfer.ini`.

8.2 With the Python API

Coming soon.

CHAPTER

NINE

POST-TRAINING QUANTIZATION

9.1 Principle

The post-training quantization algorithm is done in 3 steps:

9.1.1 1) Weights normalization

All weights are rescaled in the range $[-1.0, 1.0]$.

Per layer normalization
: There is a single weights scaling factor, global to the layer.

Per layer and per output channel normalization
: There is a different weights scaling factor for each output channel. This allows a finer grain quantization, with a better usage of the quantized range for some output channels, at the expense of more factors to be saved in memory.

9.1.2 2) Activations normalization

Activations at each layer are rescaled in the range $[-1.0, 1.0]$ for signed outputs and $[0.0, 1.0]$ for unsigned outputs.

The **optimal quantization threshold value** of the activation output of each layer is determined using the validation dataset (or test dataset if no validation dataset is available).

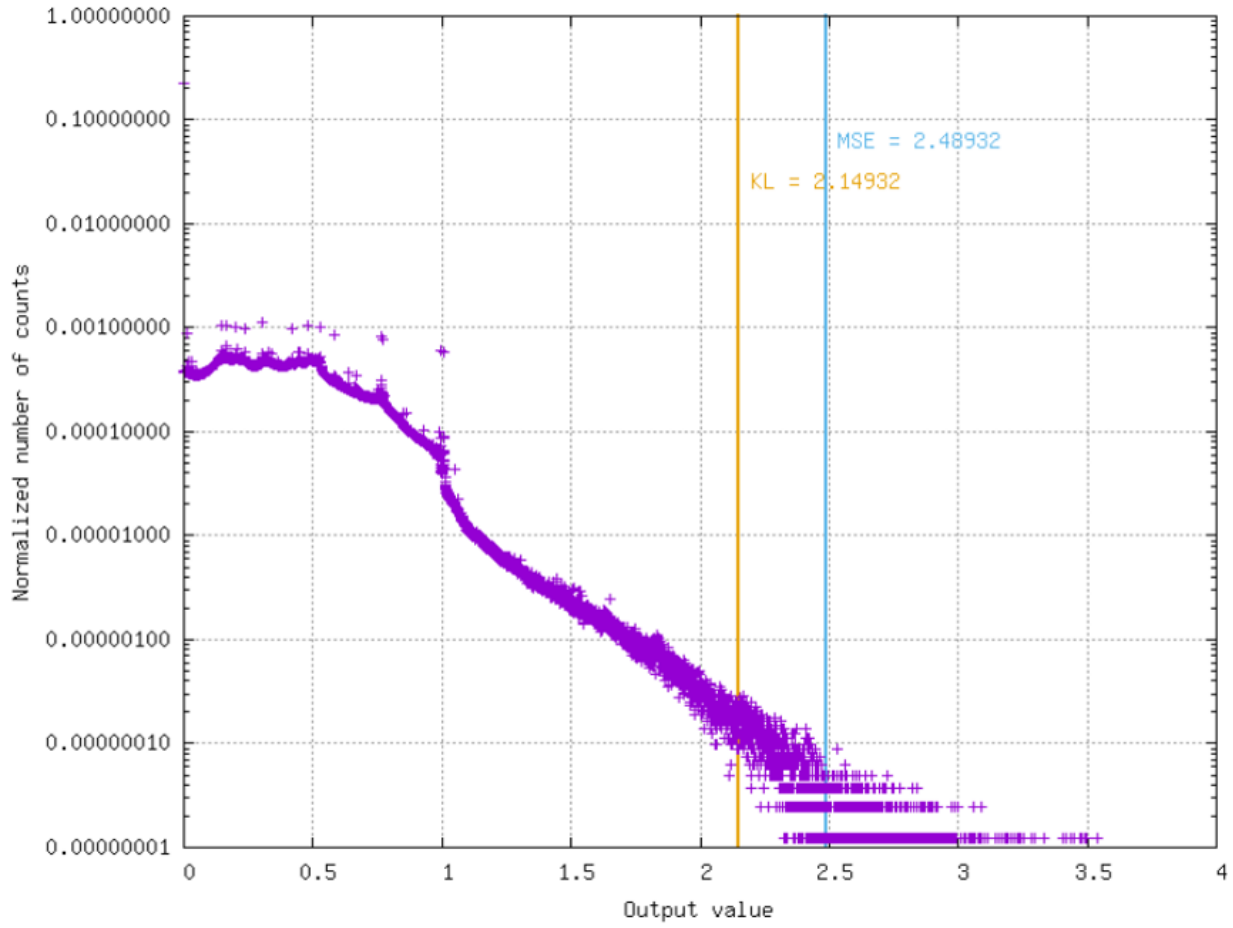
This is an iterative process: need to take into account previous layers normalizing factors.

Finding the optimal quantization threshold value of the activation output of each layer is done the following:

1) Compute histogram of activation values;
2) Find threshold that minimizes distance between original distribution and clipped quantized distribution. Two distance algorithms can be used:
   - Mean Squared Error (MSE);
   - Kullback–Leibler divergence metric (KL-divergence).

   Another, simpler method, is to just clip the values above a fixed quantile.

The obtained threshold value is therefore the activation scaling factor to be taken into account during quantization.


1.00000000
0.10000000
0.01000000
0.00100000
0.00010000
0.00001000
0.00000100
0.00000010
0.00000001
Normalized number of counts
MSE = 2.48932
KL = 2.14932
0
0.5
1
1.5
2
2.5
3
3.5
4
Output value

9.1.3 3) Quantization

Inputs, weights, biases and activations are quantized to the desired $nbbits$ precision.

Convert ranges from $[-1.0, 1.0]$ and $[0.0, 1.0]$ to $[-2^{nbbits-1} - 1, 2^{nbbits-1} - 1]$ and $[0, 2^{nbbits} - 1]$ taking into account all dependencies.

9.1.4 Additional optimization strategies

Weights clipping (optional)

Weights can be clipped using the same strategy than for the activations (finding the optimal quantization threshold using the weights histogram). However, this usually leads to worse results than no clipping.

Activation scaling factor approximation

The activation scaling factor α can be approximated the following ways:

- Fixed-point: α is approximated by $x2^{-p}$;
- Single-shift: α is approximated by 2^x;
- Double-shift: α is approximated by $2^n + 2^m$.

9.2 Usage in N2D2

All the post-training strategies described above are available in N2D2 for any export type. To apply post-training quantization during export, simply use the `-calib` command line argument.

The following parameters are available in command line:

Argument [default value]	Description
`-calib`	Number of stimuli used for the calibration (`-1` = use the full validation dataset)
`-calib-reload`	Reload and reuse the data of a previous calibration
`-wt-clipping-mode` [`None`]	Weights clipping mode on export, can be `None`, `MSE` or `KL-Divergence`
`-act-clipping-mode` [`MSE`]	Activations clipping mode on export, can be `None`, `MSE`, `KL-Divergence` or `Quantile`
`-act-rescaling-mode` [`Single-shift`]	Activations scaling mode on export, can be `Floating-point`, `Fixed-point16`, `Fixed-point32`, `Single-shift` or `Double-shift`
`-act-rescale-per-output` [0]	If true (1), rescale activation per output instead of per layer
`-act-quantile-value` [0.9999]	If activation clipping mode is `Quantile`, fraction of the values to keep without clipping

9.2.1 -act-rescaling-mode

The `-act-rescaling-mode` specifies how the activation scaling must be approximated, for values other than `Floating-point`. This allows to avoid floating-point operation altogether in the generated code, even for complex, multi-branches networks. This is particularly useful on architectures without FPU or on FPGA.

For fixed-point scaling approximation ($x2^{-p}$), two modes are available: `Fixed-point16` and `Fixed-point32`. `Fixed-point16` specifies that x must hold in at most 16-bits, whereas `Fixed-point32` allows 32-bits x. In the later case, beware that overflow can occur on 32-bits only architectures when computing the scaling multiplication before the right shift (p).

For the `Single-shift` and `Double-shift` modes, only right shifts are allowed (scaling factor < 1.0). In case of layers with scaling factor above 1.0, `Fixed-point16` is used as fallback for these layers.

9.2.2 Command line example

Command line example to run the C++ Export on a INI file containing an ONNX model:

```
n2d2 MobileNet_ONNX.ini -seed 1 -w /dev/null -export CPP -fuse -calib -1 -act-clipping-
↪mode KL-Divergence
```

9.3 Examples and results

Post-training quantization accuracy obtained with some models from the ONNX Model Zoo are reported in the table below, using `-calib 1000`:

ONNX Model Zoo model (specificities)	FP acc.	Fake 8 bits acc.	8 bits acc.
resnet18v1.onnx (`-no-unsigned -act-rescaling-mode Fixed-point`)	69.83%	68.82%	68.78%
mobilenetv2-1.0.onnx (`mobilenetv20_output_flatten0_reshape0` ignored)	70.95%	65.40%	65.40%
mobilenetv2-1.0.onnx (`mobilenetv20_output_flatten0_reshape0` ignored `-act-rescaling-mode Fixed-point`)		66.67%	66.70%
squeezenet/model.onnx (`-no-unsigned -act-rescaling-mode Floating-point`)	57.58%	57.11%	54.98%

- *FP acc.* is the floating point accuracy obtained before post-training quantization on the model imported in ONNX;
- *Fake 8 bits acc.* is the accuracy obtained after post-training quantization in N2D2, in fake-quantized mode (the numbers are quantized but the representation is still floating point);
- *8 bits acc.* is the accuracy obtained after post-training quantization in the N2D2 reference C++ export, in actual 8 bits representation.

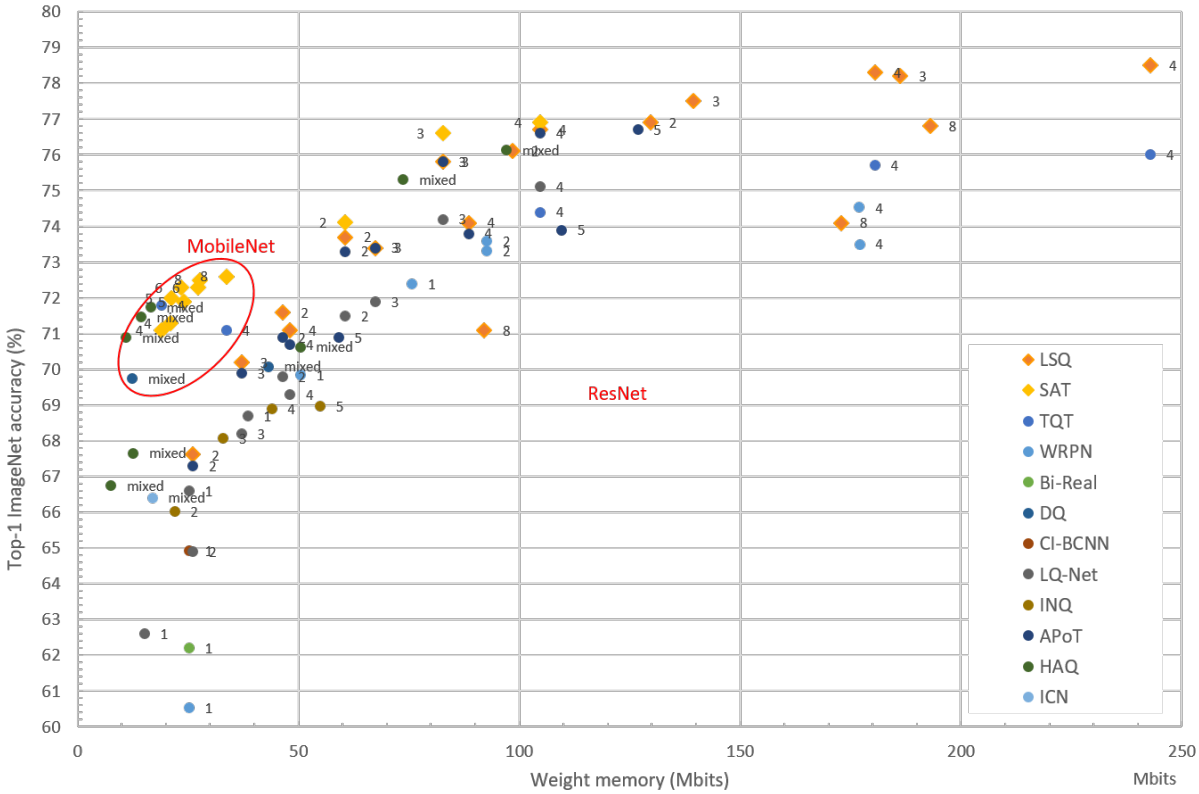
CHAPTER

TEN

QUANTIZATION-AWARE TRAINING

10.1 Getting Started

N2D2 provides a complete design environement for a super wide range of quantization modes. Theses modes are implemented as a set of integrated highly modular blocks. N2D2 implements a per layer quantization scheme that can be different at each level of the neural network. This high granularity enables to search for the best implementation depending on the hardware constraints. Moreover to achieve the best performances, N2D2 implements the latest quantization methods currently at the best of the state-of-the-art, summarized in the figure below. Each dot represents one DNN (from the MobileNet or ResNet family), quantized with the number of bits indicated beside.



The user can leverage the high modularity of our super set of quantizer blocks and simply choose the method that best fits with the initial requirements, computation resources and time to market strategy. For example to implement the `LSQ` method, one just need a limited number of training epochs to quantize a model while implementing the `SAT` method requires a higher number of training epochs but gives today the best quantization performance. In addition, the final

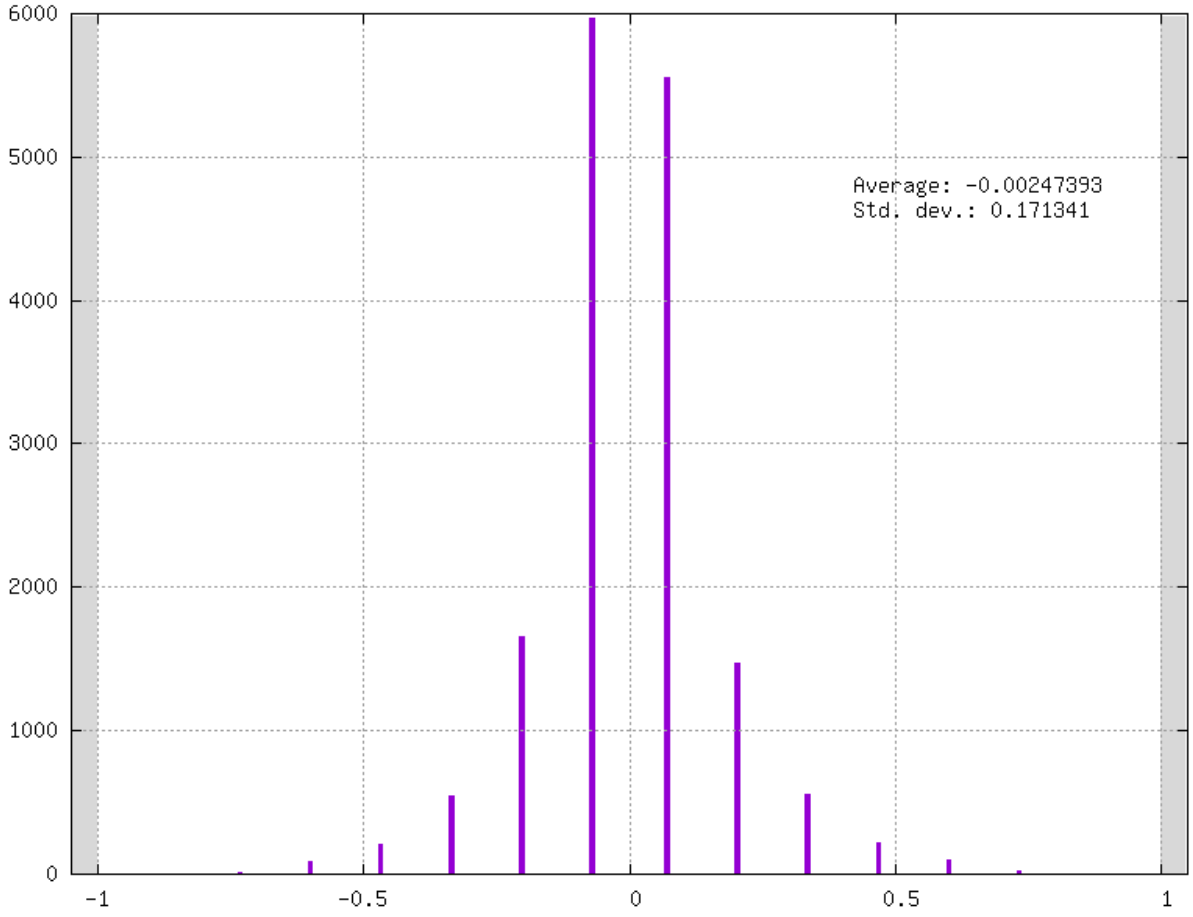
objectives can be expressed in terms of different user requirements, depending on the compression capability of the targeted hardware. Depending on these different objectives we can consider different quantization schemes:

Weights-Only Quantization

In this quantization scheme only weights are discretized to fit in a limited set of possible states. Activations are not impacted. Let's say we want to evaluate the performances of our model with 3 bits weights for convolutions layers. N2D2 natively provides the possibility to add a quantizer module, no need to import a new package or to modify any source code. We then just need to specify `QWeight` type and `QWeight.Range` for step level discretization.

```
...
QWeight=SAT ; Quantization Method can be ``LSQ`` or ``SAT``
QWeight.Range=15 ; Range is set to ``15`` step level, can be represented as a 4-bits word
...
```

Example of fake-quantized weights on 4-bits / 15 levels:



Mixed Weights-Activations Quantization

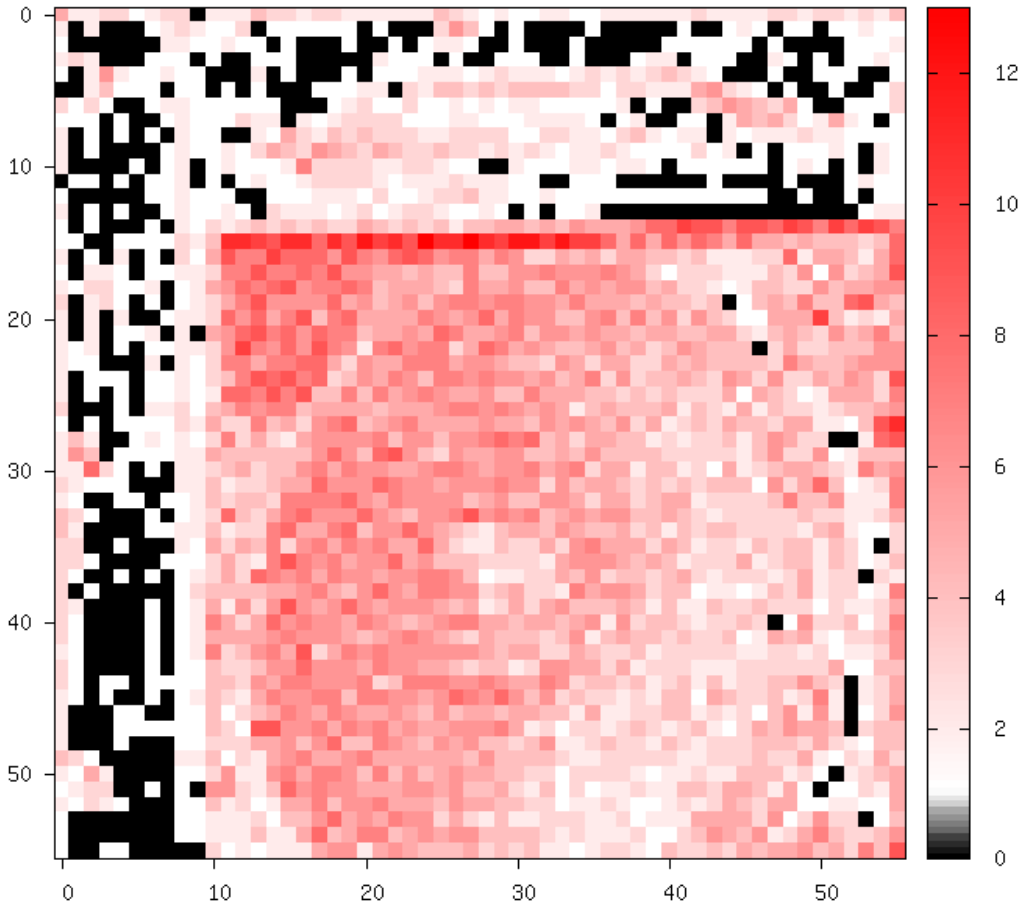
In this quantization scheme both activations and weights are quantized at different possible step levels. For layers that have a non-linear activation function and learnable parameters, such as `Fc` and `Conv`, we first specify `QWeight` in the same way as Weights-Only quantization mode.

Let's say now that we want to evaluate the performances of our model with activations quantized to 3-bits. In a similar manner, as for `QWeight` quantizer we specify the activation quantizer `QAct` for all layers that have a non-linear activation function. Where the method itself, here `QAct=SAT` ensures the non-linearity of the activation

function.

```
...
ActivationFunction=Linear
QAct=SAT ; Quantization Method can be ``LSQ`` or ``SAT``
QAct.Range=7 ; Range is set to ``7`` step level, can be represented as a 3-bits word
...
```

Example of an activation feature map quantized in 4-bits / 15 levels:

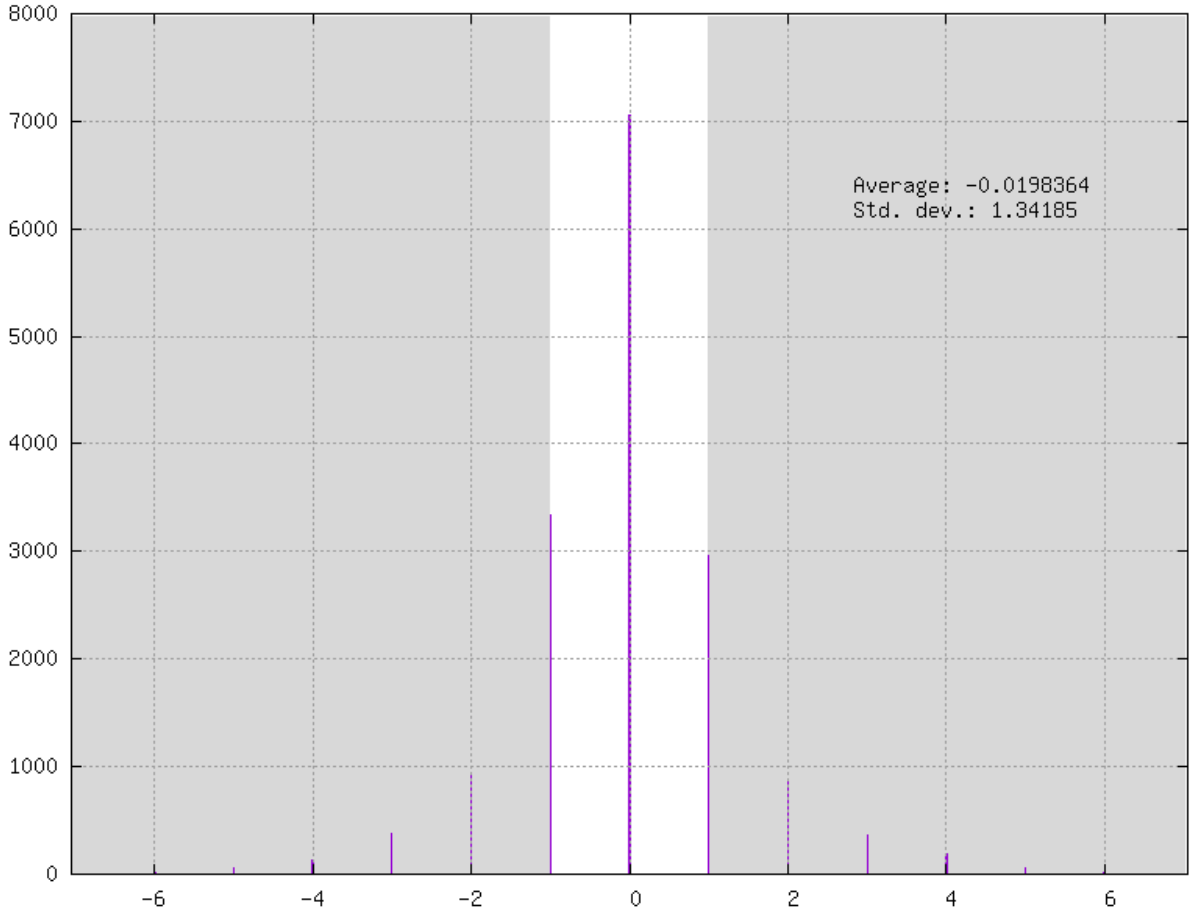


Integer-Only Quantization

Activations and weights are only represented as Integer during the learning phase, it's one step beyond classical fake quantization !! In practice, taking advantage of weight-only quantization scheme or fake quantization is clearly not obvious on hardware components. The Integer-Only quantization mode is made to fill this void and enable to exploit QAT independently of the targeted hardware architecture. Most common programmable architectures like CPU, GPU, DSP can implement it without additional burden. In addition, hardware implementation like HLS or RTL description natively support low-precision integer operators. In this mode, we replace the default quantization mode of the weights as follows :

```
...
QWeight.Mode=Integer ; Can be ``Default`` (fake-quantization) mode or ``Integer``(true␣
↪integer) mode
...
```

Example of full integer weights on 4-bits / 15 levels:



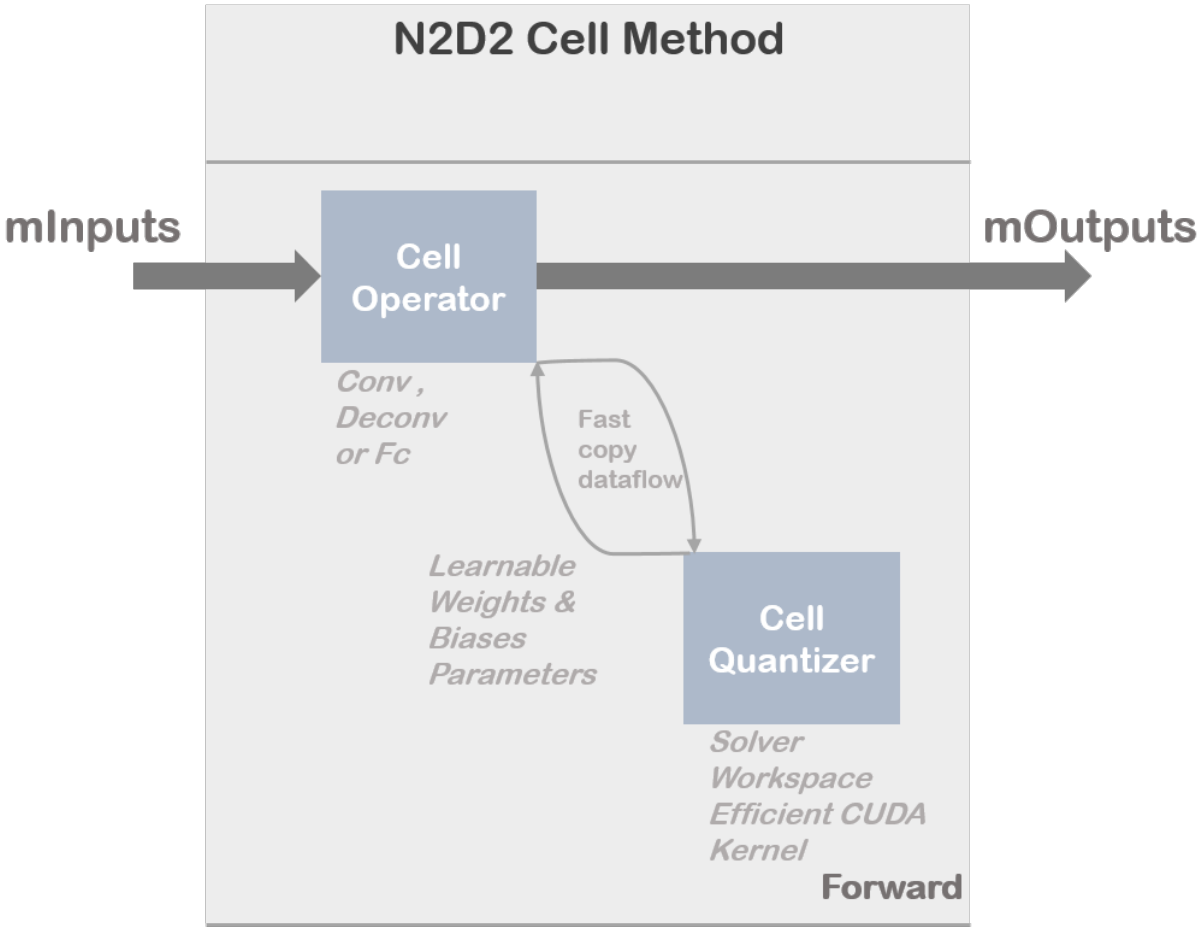
10.2 Cell Quantizer Definition

N2D2 implements a cell quantizer block for discretizing weights and biases at training time. This cell quantizer block is totally transparent for the user. The quantization phase of the learnable parameters requires intensive operation to adapt the distribution of the full-precision weights and to adapt the gradient. In addition the implementation can become highly memory greedy which can be a problem to train a complex model on a single GPU without specific treatment (gradient accumulation, etc..). That is why N2D2 merged different operations under dedicated CUDA kernels or CPU kernels allowing efficient utilization of available computation resources.

Overview of the cell quantizer implementation :

The common set of parameters for any kind of Cell Quantizer.

Option [default value]	Description
`QWeight`	Quantization method can be `SAT` or `LSQ`.
`QWeight.Range` [`255`]	Range of Quantization, can be `1` for binary, `255` for 8-bits etc..
`QWeight.Solver` [`SGD`]	Type of the Solver for learnable quantization parameters, can be `SGD` or `ADAM`
`QWeight.Mode` [`Default`]	Type of quantization Mode, can be `Default` or `Integer`


N2D2 Cell Method
mInputs
Cell Operator
mOutputs
Conv , Deconv or Fc
Fast copy dataflow
Learnable Weights & Biases Parameters
Cell Quantizer
Solver Workspace Efficient CUDA Kernel
Forward

10.2.1 LSQ

The Learned Step size Quantization method is tailored to learn the optimal quantization step size parameters in parallel with the network weights. As described in [BLN+20], LSQ tries to estimate and scale the task loss gradient at each weight and activations layer's quantizer step size, such that it can be learned in conjunction with other network parameters. This method can be initialized using weights from a pre-trained full precision model.

Option [default value]	Description
`QWeight.StepSize` [`100`]	Initial value of the learnable StepSize parameter
`QWeight.StepOptInitStepSize` [`true`]	If `true` initialize StepSize along first batch variance

10.2.2 SAT

Scale-Adjusted Training : [JYL19] method is one of the most promising solutions. The authors proposed SAT as a simple yet effective technique with which the rules of efficient training are maintained so that performance can be boosted and low-precision models can even surpass their full-precision counterparts in some cases. This method exploits DoReFa scheme for the weights quantization.

Option [default value]	Description
`QWeight.ApplyQuantization` [`true`]	Use `true` to enable quantization, if `false` parameters will be clamped between [-1.0,1.0]
`QWeight.ApplyScaling` [`false`]	Use `true` to scale the parameters as described in the SAT paper

Example of clamped weights when `QWeight.ApplyQuantization=false`:

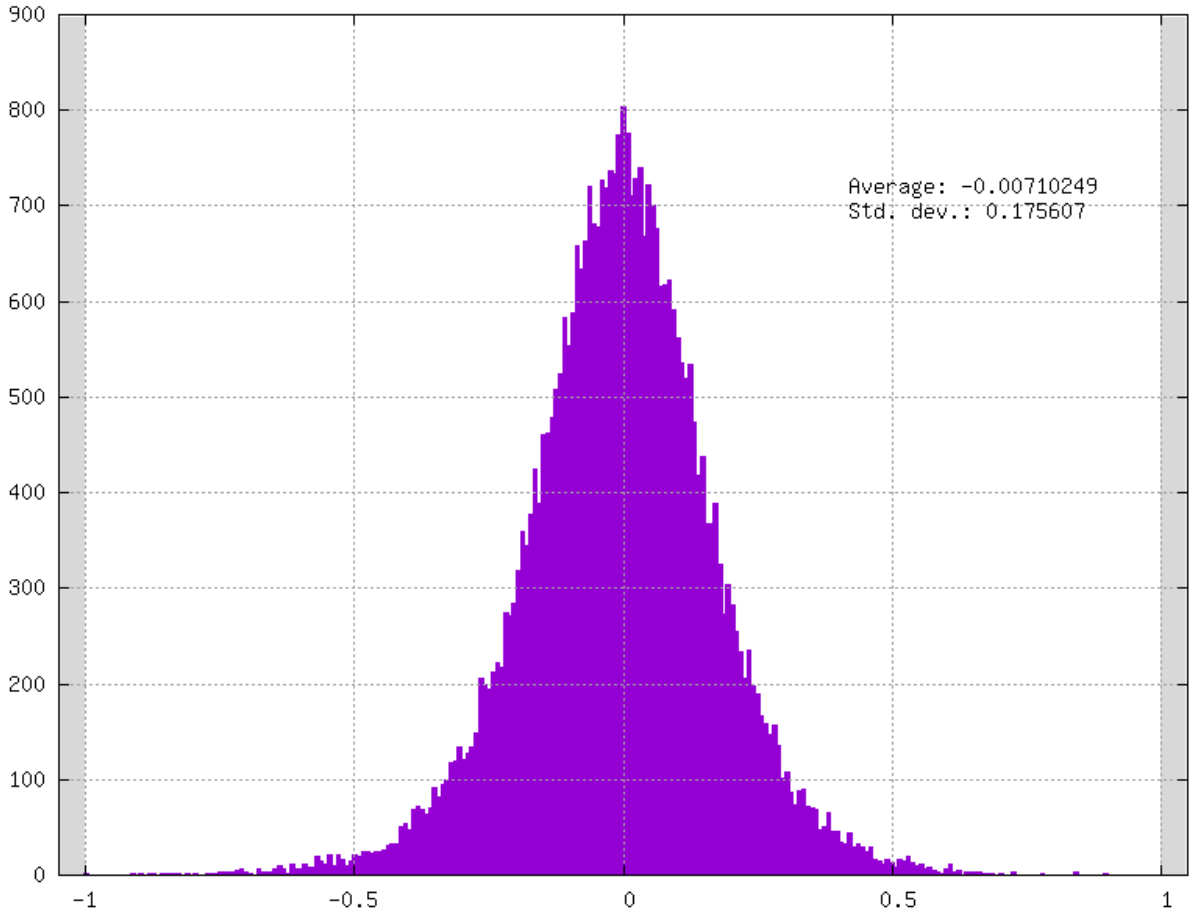
10.3 Activation Quantizer Definition

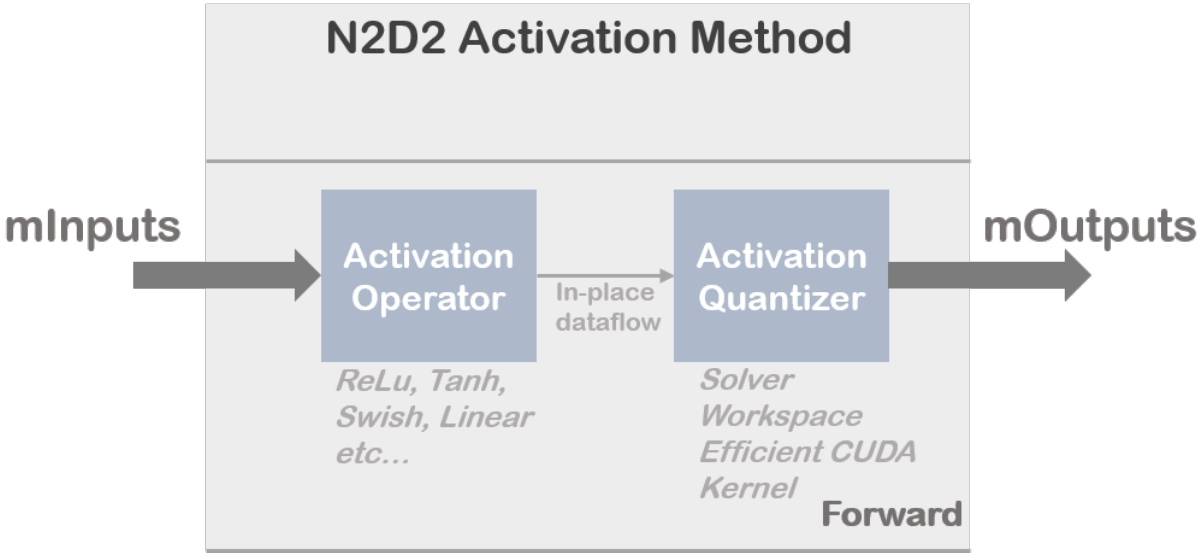
N2D2 implements an activation quantizer block to discretize activation at training time. Activation quantizer block is totally transparent for the user. Quantization phase of the activation requires intensive operation to learn parameters that will rescale the histogram of full-precision activation at training time. In addition the implementation can become highly memory greedy which can be a problem to train a complex model on a single GPU without specific treatment (gradient accumulation etc..). That why N2D2 merged different operations under dedicated CUDA kernels or CPU kernels allowing efficient utilization of available computing resources.

Overview of the activation quantizer implementation:

The common set of parameters for any kind of Activation Quantizer.

Option [default value]	Description
`QAct`	Quantization method can be `SAT` or `LSQ`.
`QAct.Range` [`255`]	Range of Quantization, can be `1` for binary, `255` for 8-bits etc..
`QAct.Solver` [`SGD`]	Type of the Solver for learnable quantization parameters, can be `SGD` or `ADAM`


Average: -0.00710249
Std. dev.: 0.175607
900
800
700
600
500
400
300
200
100
0
-1
-0.5
0
0.5
1


N2D2 Activation Method
mInputs
Activation Operator
In-place dataflow
Activation Quantizer
mOutputs
ReLu, Tanh, Swish, Linear etc...
Solver Workspace Efficient CUDA Kernel
Forward

10.3.1 LSQ

The Learned Step size Quantization method is tailored to learn the optimum quantization stepsize parameters in parallel to the network's weights. As described in [BLN+20], LSQ tries to estimate and scale the task loss gradient at each weight and activations layer's quantizer step size, such that it can be learned in conjunction with other network parameters. This method can be initialized using weights from a pre-trained full precision model.

Option [default value]	Description
`QAct.StepSize` [`100`]	Initial value of the learnable StepSize parameter
`QAct.StepOptInitStepSize` [`true`]	If `true` initialize StepSize following first batch variance

10.3.2 SAT

Scale-Adjusted Training : [JYL19] is one of the most promising solutions. The authors proposed SAT as a simple yet effective technique for which the rules of efficient training are maintained so that performance can be boosted and low-precision models can even surpass their full-precision counterparts in some cases. This method exploits a CG-PACT scheme for the activations quantization which is a boosted version of PACT for low precision quantization.

Option [default value]	Description
`QAct.Alpha` [`8.0`]	Initial value of the learnable alpha parameter

10.4 Layer compatibility table

Here we describe the compatibility table as a function of the quantization mode. The column `Cell` indicates layers that have a full support to quantize their learnable parameters during the training phase. The column `Activation` indicates layers that can support an activation quantizer to their output feature map. An additional column `Integer Core` indicates layers that can be represented without any full-precision operators at inference time. Of course it is necessary that their input comes from quantized activations.

Layer compatibility table	Quantization Mode		
	Cell (parameters)	Activation	Integer Core
Activation		✓	✓
Anchor		✓	✗
BatchNorm*	✓	✓	✓
Conv	✓	✓	✓
Deconv	✓	✓	✓
ElemWise		✓	✓
Fc	✓	✓	✓
FMP		✓	✗
LRN	✗	✗	✗
LSTM	✗	✗	✗
ObjectDet		✓	✗
Padding		✓	✓
Pool		✓	✓
Proposal		✓	✗
Reshape		✓	✓
Resize		✓	✓
ROIPooling		✓	✗
RP		✓	✗
Scaling		✓	✓
Softmax		✓	✗
Threshold		✓	✓
Transformation		✓	✗
Transpose		✓	✓
Unpool		✓	✗

BatchNorm Cell parameters are not directly quantized during the training phase. N2D2 provides a unique approach to absorb its trained parameters as an integer within the only-integer representation of the network during a fusion phase. This method is guaranteed without any loss of applicative performances.

10.5 Tutorial

10.5.1 ONNX model : ResNet-18 Example - INI File

In this example we show how to quantize the `resnet-18-v1` ONNX model with 4-bits weights and 4-bits activations using the `SAT` quantization method. We start from the `resnet18v1.onnx` file that you can pick-up at https://s3.amazonaws.com/onnx-model-zoo/resnet/resnet18v1/resnet18v1.onnx . You can also download it from the N2D2 script `N2D2/tools/install_onnx_models.py` that will automatically install a set of pre-trained ONNX models under your `N2D2_MODELS` system path.

Moreover you can start from `.ini` located at `N2D2/models/ONNX/resnet-18-v1-onnx.ini` and directly modify it or you can create an empty `resnet18-v1.ini` file in your simulation folder and to copy/paste all the following `ini` inistruction in it.

Also in this example you will need to know the ONNX cell names of your graph. We recommend you to opening the ONNX graph in a graph viewer like NETRON (https://lutzroeder.github.io/netron/).

In this example we focus to demonstrate how to apply `SAT` quantization procedure in the `resnet-18-v1` ONNX model. The first step of the procedure consists to learn `resnet-18-v1` on `ImageNet` database with clamped weights.

First of all we instantiate driver dataset and pre-processing / data augmentation function:

```
DefaultModel=Frame_CUDA
;ImageNet dataset
[database]
Type=ILSVRC2012_Database
RandomPartitioning=1
Learn=1.0

;Standard image resolution for ImageNet, batchsize=128
[sp]
SizeX=224
SizeY=224
NbChannels=3
BatchSize=128

[sp.Transformation-1]
Type=ColorSpaceTransformation
ColorSpace=RGB

[sp.Transformation-2]
Type=RangeAffineTransformation
FirstOperator=Divides
FirstValue=255.0

[sp.Transformation-3]
Type=RandomResizeCropTransformation
Width=224
Height=224
ScaleMin=0.2
ScaleMax=1.0
RatioMin=0.75
RatioMax=1.33
ApplyTo=LearnOnly

[sp.Transformation-4]
Type=RescaleTransformation
Width=256
Height=256
KeepAspectRatio=1
ResizeToFit=0
ApplyTo=NoLearn

[sp.Transformation-5]
Type=PadCropTransformation
Width=[sp.Transformation-4]Width
Height=[sp.Transformation-4]Height
ApplyTo=NoLearn

[sp.Transformation-6]
Type=SliceExtractionTransformation
Width=[sp]SizeX
Height=[sp]SizeY
OffsetX=16
OffsetY=16
```

(continues on next page)

(continued from previous page)

```
ApplyTo=NoLearn

[sp.OnTheFlyTransformation-7]
Type=FlipTransformation
ApplyTo=LearnOnly
RandomHorizontalFlip=1
```

Now that dataset driver and pre-processing are well defined we can now focus on the neural network configuration. In our example we decide to quantize all convolutions and fully-connected layers. A base block common to all convolution layers can be defined in the *.ini* file. This specific base-block uses `onnx:Conv_def` that will overwrite the native definition of all convolution layers defined in the ONNX file. This base block is used to set quantization parameters, like weights bits range, the scaling mode and the quantization mode, and also solver configuration.

```
[onnx:Conv_def]
QWeight=SAT
QWeight.ApplyScaling=0  ; No scaling needed because each conv is followed by batch-
↪normalization layers
QWeight.ApplyQuantization=0 ; Only clamp mode for the 1st step
WeightsFiller=XavierFiller ; Specific filler for SAT method
WeightsFiller.VarianceNorm=FanOut ; Specific filler for SAT method
WeightsFiller.Scaling=1.0 ; Specific filler for SAT method
ConfigSection=conv.config ; Config for conv parameters

[conv.config]
NoBias=1 ; No bias needed because each conv is followed by batch-normalization layers
Solvers.LearningRatePolicy=CosineDecay ; Can be different Policy following your problem,␣
↪recommended with SAT method
Solvers.LearningRate=0.05 ; Typical value for batchsize=256 with SAT method
Solvers.Momentum=0.9 ; Typical value for batchsize=256 with SAT method
Solvers.Decay=0.00004 ; Typical value for batchsize=256 with SAT method
Solvers.MaxIterations=192175050; For 150-epoch on ImageNet 1 epoch = 1281167 samples,␣
↪150 epoch = 1281167*150 samples
Solvers.IterationSize=2 ;Our physical batch size is set to 128, iteration size is set to␣
↪2 because we want a batchsize of 256
```

A base block common to all Fully-Connected layers can be defined in the *.ini* file. This specific base-block uses `onnx:Fc_def` that will overwrite the native definition of all fully-connected layers defined in the ONNX file. This base block is used to set quantization parameters, like weights bits range, the scaling mode and the quantization mode, and also solver configuration.

```
[onnx:Fc_def]
QWeight=SAT
QWeight.ApplyScaling=1  ; Scaling needed for Full-Connected
QWeight.ApplyQuantization=0 ; Only clamp mode for the 1st step
WeightsFiller=XavierFiller ; Specific filler for SAT method
WeightsFiller.VarianceNorm=FanOut ; Specific filler for SAT method
WeightsFiller.Scaling=1.0 ; Specific filler for SAT method
ConfigSection=fc.config ; Config for conv parameters

[fc.config]
NoBias=0 ; Bias needed for fully-connected
Solvers.LearningRatePolicy=CosineDecay ; Can be different Policy following your problem,␣
```

(continues on next page)

(continued from previous page)

```
↪recommended with SAT method
Solvers.LearningRate=0.05 ; Typical value for batchsize=256 with SAT method
Solvers.Momentum=0.9 ; Typical value for batchsize=256 with SAT method
Solvers.Decay=0.00004 ; Typical value for batchsize=256 with SAT method
Solvers.MaxIterations=192175050; For 150-epoch on ImageNet 1 epoch = 1281167 samples,␣
↪150 epoch = 1281167*150 samples
Solvers.IterationSize=2 ;Our physical batch size is set to 128, iteration size is set to␣
↪2 because we want a batch size of 256
```

A base block common to all Batch-Normalization layers can be defined in the *.ini* file. This specific base-block uses `onnx:Batchnorm_def` that will overwrites the native definition of all the batch-normalization defined in the ONNX file. We simply defined here hyper-parameters of batch-normalization layers.

```
[onnx:BatchNorm_def]
ConfigSection=bn_train.config

[bn_train.config]
Solvers.LearningRatePolicy=CosineDecay ; Can be different Policy following your problem,␣
↪recommended with SAT method
Solvers.LearningRate=0.05 ; Typical value for batchsize=256 with SAT method
Solvers.Momentum=0.9 ; Typical value for batchsize=256 with SAT method
Solvers.Decay=0.00004 ; Typical value for batchsize=256 with SAT method
Solvers.MaxIterations=192175050; For 150-epoch on ImageNet 1 epoch = 1281167 samples,␣
↪150 epoch = 1281167*150 samples
Solvers.IterationSize=2 ;Our physical batchsize is set to 128, iterationsize is set to 2␣
↪because we want a batchsize of 256
```

Then we described the `resnet-18-v1` topology directly from the ONNX file that you previously installed in your simulation folder :

```
[onnx]
Input=sp
Type=ONNX
File=resnet18v1.onnx
ONNX_init=0 ; For SAT method we need to initialize from clamped weights or dedicated␣
↪filler

[soft1]
Input=resnetv15_dense0_fwd
Type=Softmax
NbOutputs=1000
WithLoss=1

[soft1.Target]
```

Now that you set your `resnet18-v1.ini` file in your simulation folder you juste have to run the learning phase to clamp the weights with the command:

```
./n2d2 resnet18-v1.ini -learn-epoch 150 -valid-metric Precision
```

This command will run the learning phase over 150 epochs with the `Imagenet` dataset. The final test accuracy must reach at least 70%.

Next, you have to save parameters of the weights folder to the other location, for example *weights_clamped* folder.

Congratulations! Your `resnet-18-v1` model have clamped weights now ! You can check the results in your *weights_clamped* folder. Now that your `resnet-18-v1` model provides clamped weights you can play with it and try different quantization mode.

In addition, if you want to quantized also the `resnet-18-v1` activations you need to create a specific base-block in your `resnet-18-v1.ini` file in that way :

```
[ReluQ_def]
ActivationFunction=Linear ; No more need Relu because SAT quantizer integrates it's own␣
↪non-linear activation
QAct=SAT ; SAT quantization method
QAct.Range=15 ; Range=15 for 4-bits quantization model
QActSolver=SGD ; Specify SGD solver for learned alpha parameter
QActSolver.LearningRatePolicy=CosineDecay ; Can be different Policy following your␣
↪problem, recommended with SAT method
QActSolver.LearningRate=0.05 ; Typical value for batchsize=256 with SAT method
QActSolver.Momentum=0.9 ; Typical value for batchsize=256 with SAT method
QActSolver.Decay=0.00004 ; Typical value for batchsize=256 with SAT method
QActSolver.MaxIterations=192175050; For 150-epoch on ImageNet 1 epoch = 1281167 samples,␣
↪150 epoch = 1281167*150 samples
QActSolver.IterationSize=2 ;Our physical batch size is set to 128, iteration size is set␣
↪to 2 because we want a batchsize of 256
```

This base-block will be used to overwrites all the `rectifier` activation function of the ONNX model. To identify the name of the different activation function you can use the netron tool:

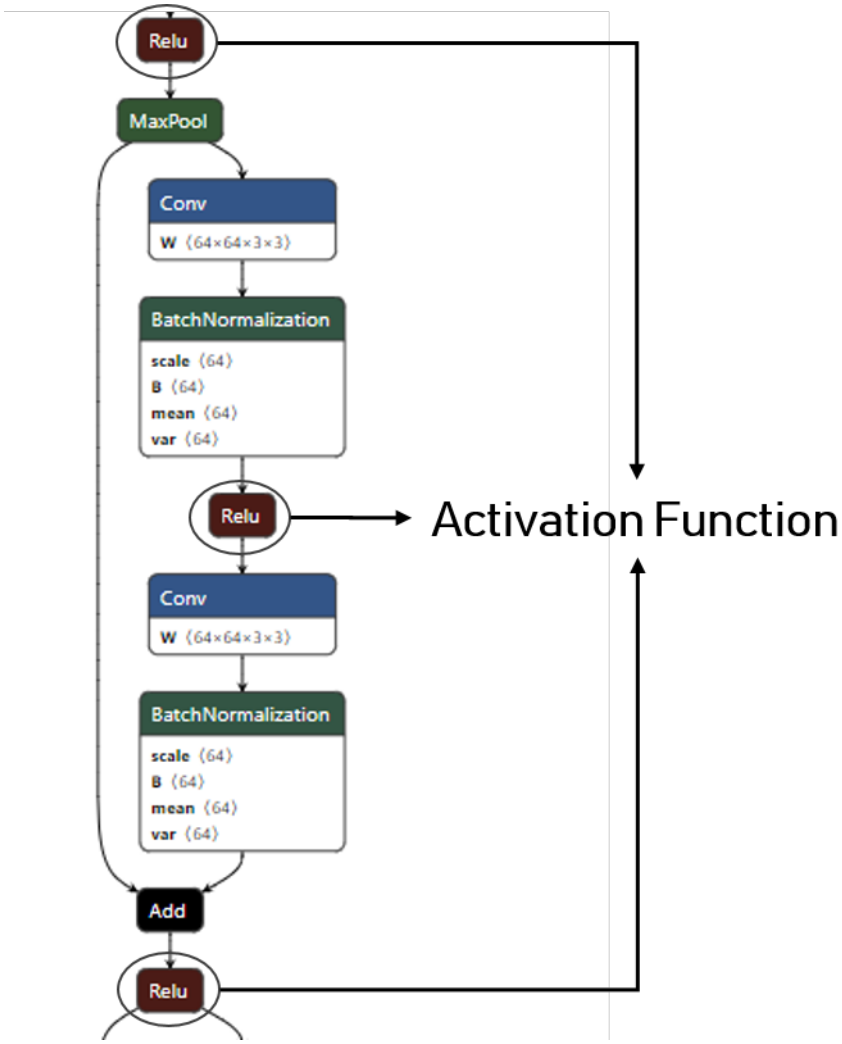
We then overrides all the activation function of the model by our previously described activation quantizer:

```
[resnetv15_relu0_fwd]ReluQ_def
[resnetv15_stage1_relu0_fwd]ReluQ_def
[resnetv15_stage1_activation0]ReluQ_def
[resnetv15_stage1_relu1_fwd]ReluQ_def
[resnetv15_stage1_activation1]ReluQ_def
[resnetv15_stage2_relu0_fwd]ReluQ_def
[resnetv15_stage2_activation0]ReluQ_def
[resnetv15_stage2_relu1_fwd]ReluQ_def
[resnetv15_stage2_activation1]ReluQ_def
[resnetv15_stage3_relu0_fwd]ReluQ_def
[resnetv15_stage3_activation0]ReluQ_def
[resnetv15_stage3_relu1_fwd]ReluQ_def
[resnetv15_stage3_activation1]ReluQ_def
[resnetv15_stage4_relu0_fwd]ReluQ_def
[resnetv15_stage4_activation0]ReluQ_def
[resnetv15_stage4_relu1_fwd]ReluQ_def
[resnetv15_stage4_activation1]ReluQ_def
```

Now that activations quantization mode is set we focuses on the weights parameters quantization. For example to quantize weights also in a 4 bits range, you should set the parameters convolution base-block in that way:

```
[onnx:Conv_def]
...
QWeight.ApplyQuantization=1 ; Set to 1 for quantization mode
QWeight.Range=15 ;  Conv is now quantized in 4-bits range (2^4 - 1)
...
```

In a same manner, you can modify the fully-connected base-block in that way :


Relu
MaxPool
Conv
W (64×64×3×3)
BatchNormalization
scale (64)
B (64)
mean (64)
var (64)
Relu
Activation Function
Conv
W (64×64×3×3)
BatchNormalization
scale (64)
B (64)
mean (64)
var (64)
Add
Relu

```
[onnx:Fc_def]
...
QWeight.ApplyQuantization=1 ; Set to 1 for quantization mode
QWeight.Range=15 ;  Fc is now quantized in 4-bits range (2^4 - 1)
...
```

As a common practice in quantization aware training the first and last layers are quantized in 8-bits. In ResNet-18 the first layer is a convolution layer, we have to specify that to the first layer.

We first start to identify the name of the first layer under the netron environement:

We then overrides the range of the first convolution layer of the `resnet18v1.onnx` model:

```
[resnetv15_conv0_fwd]onnx:Conv_def
QWeight.Range=255 ;resnetv15_conv0_fwd is now quantized in 8-bits range (2^8 - 1)
```

In a same way we overrides the range of the last fully-connected layer in 8-bits :

```
[resnetv15_dense0_fwd]onnx:Fc_def
QWeight.Range=255 ;resnetv15_dense0_fwd is now quantized in 8-bits range (2^8 - 1)
```

Now that your modified `resnet-18-v1.ini` file is ready just have to run a learning phase with the same hyperparameters by using transfer learning method from the previously clamped weights with this command:

```
./n2d2 resnet-18-v1.ini -learn-epoch 150 -w weights_clamped -valid-metric Precision
```

This command will run the learning phase over 150 epochs with the `Imagenet` dataset. The final test accuracy must reach at least 70%.

Congratulations! Your `resnet-18-v1` model have now it's weights parameters and activations quantized in a 4-bits way !

10.5.2 ONNX model : ResNet-18 Example - Python

In this example, we will do the same as in the previous section showcasing the python API.

You can find the complete scrip for this tutorial here `resnet18v1 quantization example`.

Firstly, you need to retrieved the `resnet18v1.onnx` file that you can pick-up at https://s3.amazonaws.com/onnx-model-zoo/resnet/resnet18v1/resnet18v1.onnx. Or with the N2D2 script `N2D2/tools/install_onnx_models.py` that will automatically install a set of pre-trained ONNX models under your `N2D2_MODELS` system path.

Once this is done, you can create a data provider for the dataset `ILSVRC2012`.

```
print("Create database")
database = n2d2.database.ILSVRC2012(learn=1.0, random_partitioning=True)
database.load(args.data_path, label_path=args.label_path)
print(database)
print("Create provider")
provider = n2d2.provider.DataProvider(database=database, size=[224, 224, 3], batch_
↪size=batch_size)
print(provider)
```

We will then do some pre-processing to the data-set.

We use the `n2d2.transform.Composite` to have a compact syntax and avoid multiple call to the method `add_transformation`.

```
print("Adding transformations")
transformations = n2d2.transform.Composite([
    n2d2.transform.ColorSpace("RGB"),
    n2d2.transform.RangeAffine("Divides", 255.0),
    n2d2.transform.RandomResizeCrop(224, 224, scale_min=0.2, scale_max=1.0, ratio_min=0.
↪75,
                                    ratio_max=1.33, apply_to="LearnOnly"),
    n2d2.transform.Rescale(256, 256, keep_aspect_ratio=True, resize_to_fit=False,
                           apply_to="NoLearn"),
    n2d2.transform.PadCrop(256, 256, apply_to="NoLearn"),
    n2d2.transform.SliceExtraction(224, 224, offset_x=16, offset_y=16, apply_to="NoLearn
↪"),
])

print(transformations)

flip_trans = n2d2.transform.Flip(apply_to="LearnOnly", random_horizontal_flip=True)

provider.add_transformation(transformations)
provider.add_on_the_fly_transformation(flip_trans)
print(provider)
```

Once this is done, we can import the `resnet-18-v1` ONNX model using `n2d2.cells.DeepNetCell`.

```
model = n2d2.cells.DeepNetCell.load_from_ONNX(provider, path_to_ONNX)
```

Once the ONNX model is loaded, we will change the configuration of the `n2d2.cells.Conv`, `n2d2.cells.Fc` and `n2d2.cells.BatchNorm2d` layers. To do so, we will iterate through the layer of our model and check the type of the layer. Then we will apply the wanted configuration for each cells.

```
print("Updating cells ...")

for cell in model:
    ### Updating Conv Cells ###
    if isinstance(cell, n2d2.cells.Conv):
        # You need to replace weights filler before adding the quantizer.
        cell.set_weights_filler(
            n2d2.filler.Xavier(
                variance_norm="FanOut",
                scaling=1.0,
        ), refill=True)

        if cell.has_bias():
            cell.refill_bias()
        cell.quantizer = SATCell(
                apply_scaling=False,
                apply_quantization=False
        )

        cell.set_solver_parameter("learning_rate_policy", "CosineDecay")
        cell.set_solver_parameter("learning_rate", 0.05)
        cell.set_solver_parameter("momentum", 0.9)
        cell.set_solver_parameter("decay", 0.00004)
        cell.set_solver_parameter("max_iterations", 192175050)
        cell.set_solver_parameter("iteration_size", 2)

    ### Updating Fc Cells ###
    if isinstance(cell, n2d2.cells.Fc):
        cell.set_weights_filler(
            n2d2.filler.Xavier(
                variance_norm="FanOut",
                scaling=1.0,
        ), refill=True)
        cell.set_bias_filler(
            n2d2.filler.Constant(
                value=0.0,
        ), refill=True)


        cell.quantizer = SATCell(
                apply_scaling=False,
                apply_quantization=False
        )
        cell.set_solver_parameter("learning_rate_policy", "CosineDecay")
        cell.set_solver_parameter("learning_rate", 0.05)
        cell.set_solver_parameter("momentum", 0.9)
        cell.set_solver_parameter("decay", 0.00004)
        cell.set_solver_parameter("max_iterations", 192175050)
        cell.set_solver_parameter("iteration_size", 2)

    ### Updating BatchNorm Cells ###
    if isinstance(cell, n2d2.cells.BatchNorm2d):
        cell.set_solver_parameter("learning_rate_policy", "CosineDecay")
```

(continues on next page)

(continued from previous page)

```
        cell.set_solver_parameter("learning_rate", 0.05)
        cell.set_solver_parameter("momentum", 0.9)
        cell.set_solver_parameter("decay", 0.00004)
        cell.set_solver_parameter("max_iterations", 192175050)
        cell.set_solver_parameter("iteration_size", 2)
  print("AFTER MODIFICATION :")
  print(model)
```

Once this is done, we will do a regular training loop and save weights every time we met a new best *precision* during the validation phase. The clamped weights will be saved in a folder *resnet_weights_clamped*.

```
softmax = n2d2.cells.Softmax(with_loss=True)

loss_function =  n2d2.target.Score(provider)
max_precision = -1
print("\n### Training ###")
for epoch in range(nb_epochs):
    provider.set_partition("Learn")
    model.learn()

    print("\n# Train Epoch: " + str(epoch) + " #")

    for i in range(math.ceil(database.get_nb_stimuli('Learn') / batch_size)):
        x = provider.read_random_batch()
        x = model(x)
        x = softmax(x)
        x = loss_function(x)

        x.back_propagate()
        x.update()

        print("Example: " + str(i * batch_size) + ", loss: "
              + "{0:.3f}".format(x[0]), end='\r')

    print("\n### Validation ###")

    loss_function.clear_success()

    provider.set_partition('Validation')
    model.test()

    for i in range(math.ceil(database.get_nb_stimuli('Validation') / batch_size)):
        batch_idx = i * batch_size

        x = provider.read_batch(batch_idx)
        x = model(x)
        x = softmax(x)
        x = loss_function(x)

        print("Validate example: " + str(i * batch_size) + ", val success: "
              + "{0:.2f}".format(100 * loss_function.get_average_score(metric="Precision
↪")) + "%", end='\r')
```

(continues on next page)

(continued from previous page)

```
print("\nPloting the network ...")
x.get_deepnet().draw_graph("./resnet18v1_clamped")
x.get_deepnet().log_stats("./resnet18v1_clamped_stats")
print("Saving weights !")
model.get_embedded_deepnet().export_network_free_parameters("resnet_weights_clamped")
```

Your *resnet-18-v1* model now have clamped weights !

Now we will change the `quantizer` objects to quantize the network et 4 bits (range=15).

```
print("Updating cells")

for cell in model:
    ### Updating Rectifier ###
    if isinstance(cell.activation, n2d2.activation.Rectifier):
        cell.activation = n2d2.activation.Linear(
                quantizer=SATAct(
                    range=15,
                    solver=n2d2.solver.SGD(
                        learning_rate_policy = "CosineDecay",
                        learning_rate=0.05,
                        momentum=0.9,
                        decay=0.00004,
                        max_iterations=115305030
        )))

    if isinstance(cell, (n2d2.cells.Conv, n2d2.cells.Fc)):
        cell.quantizer.set_quantization(True)
        cell.quantizer.set_range(15)

# The first and last cell are in 8 bits precision !
model["resnetv15_conv0_fwd"].quantizer.set_range(255)
model["resnetv15_dense0_fwd"].quantizer.set_range(255)
```

Once the `quantizer` objects have been updated we can run a new training loop to learn the quantized wieghts and activations.

```
print("\n### Training ###")
for epoch in range(nb_epochs):

    provider.set_partition("Learn")
    model.learn()

    print("\n# Train Epoch: " + str(epoch) + " #")

    for i in range(math.ceil(database.get_nb_stimuli('Learn') / batch_size)):
        x = provider.read_random_batch()
        x = model(x)
        x = softmax(x)
        x = loss_function(x)

        x.back_propagate()
```

(continues on next page)

(continued from previous page)

```
        x.update()

        print("Example: " + str(i * batch_size) + ", loss: "
              + "{0:.3f}".format(x[0]), end='\r')

    print("\n### Validation ###")

    loss_function.clear_success()

    provider.set_partition('Validation')
    model.test()

    for i in range(math.ceil(database.get_nb_stimuli('Validation') / batch_size)):
        batch_idx = i * batch_size

        x = provider.read_batch(batch_idx)
        x = model(x)
        x = softmax(x)
        x = loss_function(x)

        print("Validate example: " + str(i * batch_size) + ", val success: "
              + "{0:.2f}".format(100 * loss_function.get_average_score(metric="Precision
↪")) + "%", end='\r')

x.get_deepnet().draw_graph("./resnet18v1_quant")
x.get_deepnet().log_stats("./resnet18v1_quant_stats")
model.get_embedded_deepnet().export_network_free_parameters("resnet_weights_SAT")
```

You can look at your quantized weights in the newly created `resnet_weights_SAT` folder.

10.5.3 Hand-Made model : LeNet Example - INI File

One can apply the SAT quantization methodology on the chosen deep neural network by adding the right parameters to the `.ini` file. Here we show how to configure the `.ini` file to correctly apply the SAT quantization. In this example we decide to apply the SAT quantization procedure in a hand-made LeNet model. The first step of the procedure consists to learn `LeNet` on `MNIST` database with clamped weights.

We recommend you to create an empty `LeNet.ini` file in your simulation folder and to copy/paste all following `ini` block inside.

First of all we start to described `MNIST` driver dataset and pre-processing use for data augmentation at training and test phase:

```
; Frame_CUDA for GPU and Frame for CPU
DefaultModel=Frame_CUDA

; MNIST Driver Database Instantiation
[database]
Type=MNIST_IDX_Database
RandomPartitioning=1

; Environment Description , batch=256
```

(continues on next page)

(continued from previous page)

```
[env]
SizeX=32
SizeY=32
BatchSize=256

[env.Transformation_0]
Type=RescaleTransformation
Width=32
Height=32
```

In our example we decide to quantize all convolutions and fully-connected layers. A base block common to all convolution layers can be defined in the *.ini* file. This base block is used to set quantization parameters, like weights bits range, the scaling mode and the quantization mode, and also solver configuration.

```
[Conv_def]
Type=Conv
ActivationFunction=Linear
QWeight=SAT
QWeight.ApplyScaling=0 ; No scaling needed because each conv is followed by batch-
↪normalization layers
QWeight.ApplyQuantization=0 ; Only clamp mode for the 1st step
ConfigSection=common.config

[common.config]
NoBias=1
Solvers.LearningRate=0.05
Solvers.LearningRatePolicy=None
Solvers.Momentum=0.0
Solvers.Decay=0.0
```

A base block common to all Full-Connected layers can be defined in the *.ini* file. This base block is used to set quantization parameters, like weights bits range, the scaling mode and the quantization mode, and also solver configuration.

```
[Fc_def]
Type=Fc
ActivationFunction=Linear
QWeight=SAT
QWeight.ApplyScaling=1 ; Scaling needed because for Full-Conncted
QWeight.ApplyQuantization=0 ; Only clamp mode for the 1st step
ConfigSection=common.config
```

A base block common to all Batch-Normalization layers can be defined in the *.ini* file. This base block is used to set quantization activations, like activations bits range, the quantization mode, and also solver configuration. In this first step batch-normalization activation are not quantized yet. We simply defined a typical batch-normalization layer with `Rectifier` as non-linear activation function.

```
[Bn_def]
Type=BatchNorm
ActivationFunction=Rectifier
ConfigSection=bn.config

[bn.config]
Solvers.LearningRate=0.05
```

(continues on next page)

(continued from previous page)

```
Solvers.LearningRatePolicy=None
Solvers.Momentum=0.0
Solvers.Decay=0.0
```

Finally we described the full backbone of `LeNet` topology:

```
[conv1] Conv_def
Input=env
KernelWidth=5
KernelHeight=5
NbOutputs=6

[bn1] Bn_def
Input=conv1
NbOutputs=[conv1]NbOutputs

; Non-overlapping max pooling P2
[pool1]
Input=bn1
Type=Pool
PoolWidth=2
PoolHeight=2
NbOutputs=6
Stride=2
Pooling=Max
Mapping.Size=1

[conv2] Conv_def
Input=pool1
KernelWidth=5
KernelHeight=5
NbOutputs=16
[bn2] Bn_def
Input=conv2
NbOutputs=[conv2]NbOutputs

[pool2]
Input=bn2
Type=Pool
PoolWidth=2
PoolHeight=2
NbOutputs=16
Stride=2
Pooling=Max
Mapping.Size=1

[conv3] Conv_def
Input=pool2
KernelWidth=5
KernelHeight=5
NbOutputs=120
```

(continues on next page)

(continued from previous page)

```
[bn3]Bn_def
Input=conv3
NbOutputs=[conv3]NbOutputs

[conv3.drop]
Input=bn3
Type=Dropout
NbOutputs=[conv3]NbOutputs

[fc1] Fc_def
Input=conv3.drop
NbOutputs=84

[fc1.drop]
Input=fc1
Type=Dropout
NbOutputs=[fc1]NbOutputs

[fc2] Fc_def
Input=fc1.drop
ActivationFunction=Linear
NbOutputs=10

[softmax]
Input=fc2
Type=Softmax
NbOutputs=10
WithLoss=1

[softmax.Target]
```

Now that you have your ready `LeNet.ini` file in your simulation folder you juste have to run the learning phase to clamp the weights with the command:

```
./n2d2 LeNet.ini -learn-epoch 100
```

This command will run the learning phase over 100 epochs with the MNIST dataset. The final test accuracy must reach at least 98.9%:

```
Final recognition rate: 98.95%    (error rate: 1.05%)
Sensitivity: 98.94% / Specificity: 99.88% / Precision: 98.94%
Accuracy: 99.79% / F1-score: 98.94% / Informedness: 98.82%
```

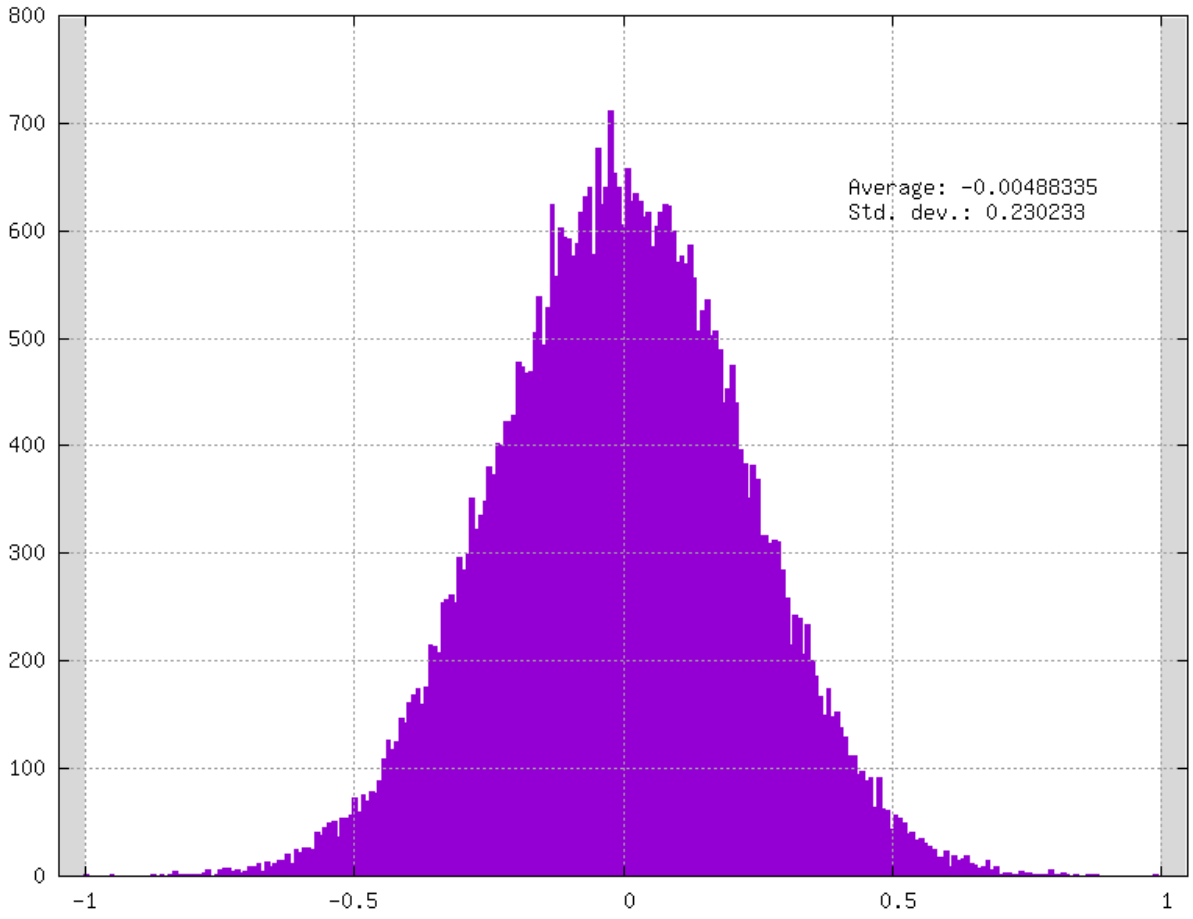
Next, you have to save parameters of the weights folder to the other location, for example *weights_clamped* folder.

Congratulations! Your `LeNet` model have clamped weights now ! You can check the results in your *weights_clamped* folder, for example check your *conv3_weights_quant.distrib.png* file :

Now that your `LeNet` model provides clamped weights you can play with it and try different quantization mode. Moreover, if you want to quantized also the `LeNet` activations you have to modify the batch-normalization base-block from your `LeNet.ini` file in that way :

```
[Bn_def]
Type=BatchNorm
```

(continues on next page)


800
700
600
500
400
300
200
100
0
-1
-0.5
0
0.5
1
Average: -0.00488335
Std. dev.: 0.230233

(continued from previous page)

```
ActivationFunction=Linear ; Replace by linear: SAT quantizer directly apply non-linear␣
↪activation
QAct=SAT
QAct.Alpha=6.0
QAct.Range=15 ; ->15 for 4-bits range (2^4 - 1)
QActSolver=SGD
QActSolver.LearningRate=0.05
QActSolver.LearningRatePolicy=None
QActSolver.Momentum=0.0
QActSolver.Decay=0.0
ConfigSection=bn.config
```

For example to quantize weights also in a 4 bits range, these parameters from the convolution base-block must be modified in that way:

```
[Conv_def]
Type=Conv
ActivationFunction=Linear
QWeight=SAT
QWeight.ApplyScaling=0
QWeight.ApplyQuantization=1 ; ApplyQuantization is now set to 1
QWeight.Range=15 ; Conv is now quantized in 4-bits range (2^4 - 1)
ConfigSection=common.config
```

In the same way, you have to modify the fully-connected base-block:

```
[Fc_def]
Type=Fc
ActivationFunction=Linear
QWeight=SAT
QWeight.ApplyScaling=1
QWeight.ApplyQuantization=1 ; ApplyQuantization is now set to 1
QWeight.Range=15 ; FC is now quantized in 4-bits range (2^4 - 1)
ConfigSection=common.config
```

As a common practice, the first and last layer are kept with 8-bits range weights parameters. To do that, the first *conv1* layer of the `LeNet` backbone must be modified in that way:

```
[conv1] Conv_def
Input=env
KernelWidth=5
KernelHeight=5
NbOutputs=6
QWeight.Range=255 ; conv1 is now quantized in 8-bits range (2^8 - 1)
```

And the last layer *fc2* of the `LeNet` must be modified in that way:

```
[fc2] Fc_def
Input=fc1.drop
ActivationFunction=Linear
NbOutputs=10
QWeight.Range=255 ; FC is now quantized in 8-bits range (2^8 - 1)
```

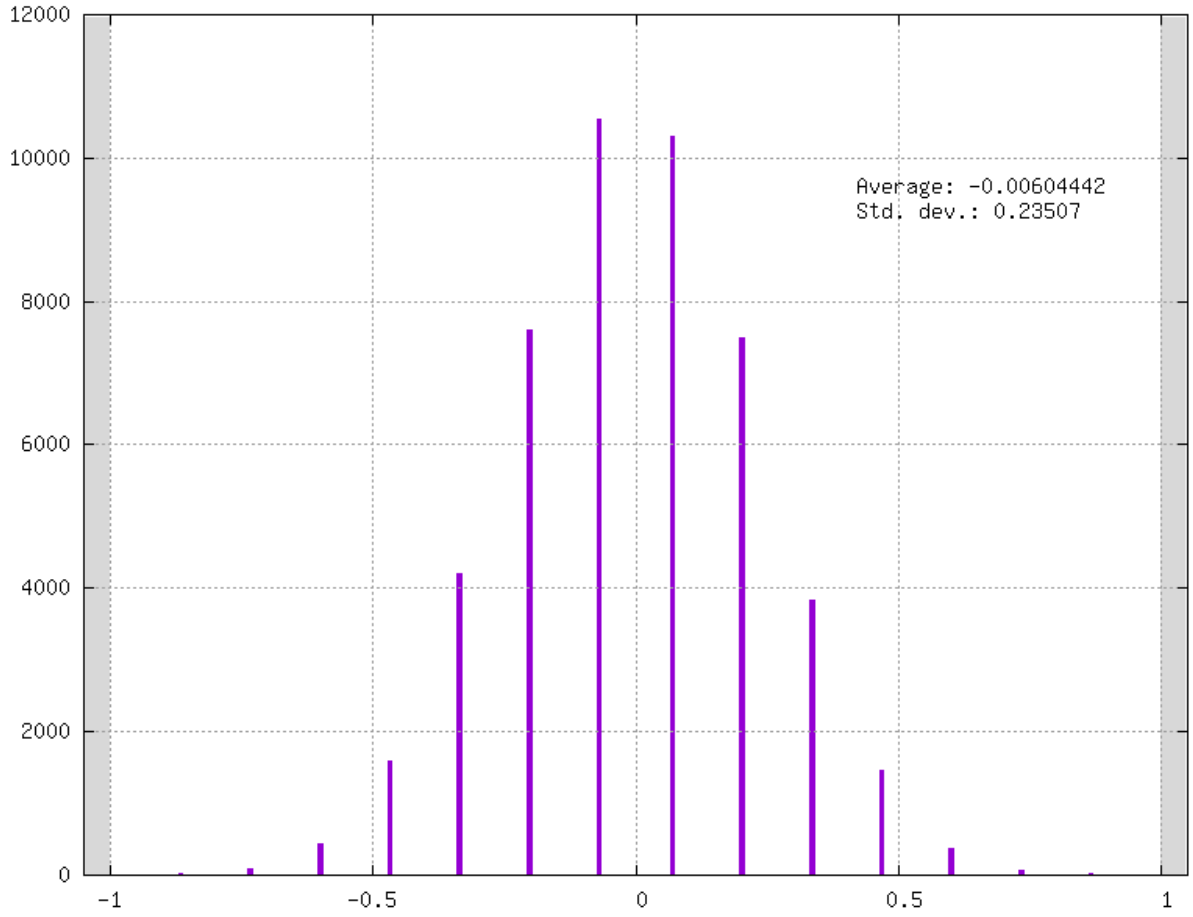
Now that your modified `LeNet.ini` file is ready just have to run a learning phase with the same hyperparameters by using transfer learning method from the previously clamped weights with this command:

```
./n2d2 LeNet.ini -learn-epoch 100 -w weights_clamped
```

The final test accuracy should be close to 99%:

```
Final recognition rate: 99.18%    (error rate: 0.82%)
  Sensitivity: 99.173293% / Specificity: 99.90895% / Precision: 99.172422%
  Accuracy: 99.836% / F1-score: 99.172195% / Informedness: 99.082242%
```

Congratulations! Your `LeNet` model is now fully-quantized ! You can check the results in your *weights* folder, for example check your *conv3_weights_quant.distrib.png* file :

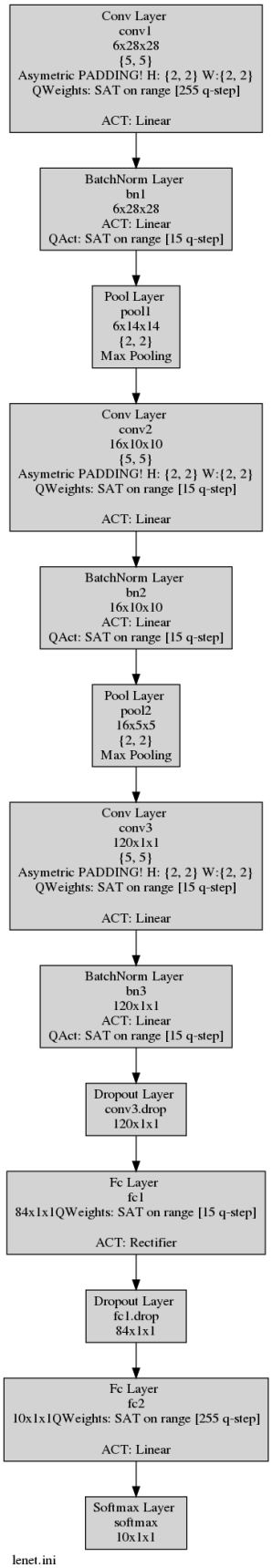


In addition you can have your model graph view that integrates the quantization information. This graph is automatically generated at the learning phase or at the test phase. In this example this graph is generated under the name `LeNet.ini.png`.

As you can see in the following figure, the batch-normalization layers are present (and essential) in your quantized model:

Obviously, no one wants batch-normalization layers in it's quantized model. We answer this problem with our internal tool named *DeepNetQAT*. This tool allowed us to fused batch normalization parameters within the scaling, clipping and biases parameters of our quantized models under the `SAT` method.

You can fuse the batch normalization parameters of your model with this command :


Conv Layer
conv1
6x28x28
{5, 5}
Asymetric PADDING! H: {2, 2} W:{2, 2}
QWeights: SAT on range [255 q-step]
ACT: Linear
BatchNorm Layer
bn1
6x28x28
ACT: Linear
QAct: SAT on range [15 q-step]
Pool Layer
pool1
6x14x14
{2, 2}
Max Pooling
Conv Layer
conv2
16x10x10
{5, 5}
Asymetric PADDING! H: {2, 2} W:{2, 2}
QWeights: SAT on range [15 q-step]
ACT: Linear
BatchNorm Layer
bn2
16x10x10
ACT: Linear
QAct: SAT on range [15 q-step]
Pool Layer
pool2
16x5x5
{2, 2}
Max Pooling
Conv Layer
conv3
120x1x1
{5, 5}
Asymetric PADDING! H: {2, 2} W:{2, 2}
QWeights: SAT on range [15 q-step]
ACT: Linear
BatchNorm Layer
bn3
120x1x1
ACT: Linear
QAct: SAT on range [15 q-step]
Dropout Layer
conv3.drop
120x1x1
Fc Layer
fc1
84x1x1QWeights: SAT on range [15 q-step]
ACT: Rectifier
Dropout Layer
fc1.drop
84x1x1
Fc Layer
fc2
10x1x1QWeights: SAT on range [255 q-step]
ACT: Linear
Softmax Layer
softmax
10x1x1

lenet.ini

```
./n2d2 LeNet.ini -test -qat-sat -w weights
```

Results must be exactly the same than with batch-normalization. Moreover quantizer modules have been entirely removed from your model ! You can check the results in the newly generated `LeNet.ini.png` graph :

Moreover you can find your quantized weights and biases under the folder `weights_quantized`.

10.5.4 Hand-Made model : LeNet Example - Python

Part 1 : Learn with clamped weights

In this section, we will see how to apply the `SAT` quantization methodology using the python API. We will apply the SAT quantization procedure in a handmade LeNet model.

You can get the script used in this example by clicking here : `LeNet quantization example`.

The first step is to learn `LeNet` on `MNIST` database with clamped weights.

Let's start by importing the folowing libraries and setting some global variables :

```
import n2d2
import n2d2_ip
from n2d2.cells.nn import Dropout, Fc, Conv, Pool2d, BatchNorm2d
import math

nb_epochs = 100
batch_size = 256
n2d2.global_variables.cuda_device = 2
n2d2.global_variables.default_model = "Frame_CUDA"
```

Let's create a `database` driver for MNIST, a `dataprovider` and apply `transformation` to the data.

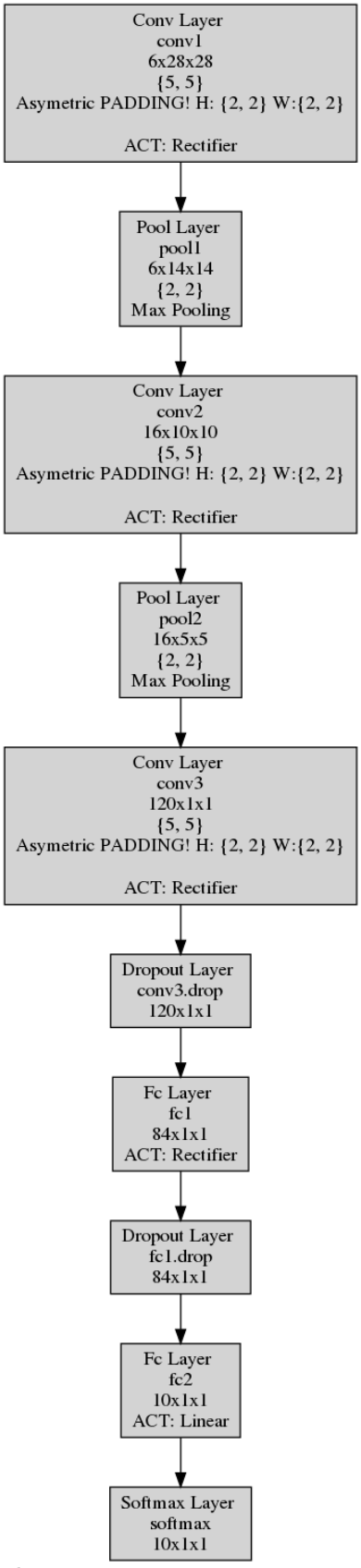
```
print("\n### Create database ###")
database = n2d2.database.MNIST(data_path=data_path, validation=0.1)
print(database)
print("\n### Create Provider ###")
provider = n2d2.provider.DataProvider(database, [32, 32, 1], batch_size=batch_size)
provider.add_transformation(n2d2.transform.Rescale(width=32, height=32))
print(provider)
```

In our example we decided to quantize every convolutions and fully-connected layers. We will use the object `n2d2.ConfigSection` to provide common parameters to the cells.

Note: We need to use a function that will generate a new config section object to avoid giving the same objects to the one we are configuring. If we defined `conv_conf` as the `solver_conf` every `Conv cells` would have the same solver and quantizer object !

```
solver_conf = n2d2.ConfigSection(
    learning_rate=0.05,
    learning_rate_policy="None",
    momentum=0.0,
    decay=0.0,
)
```

(continues on next page)


Conv Layer
conv1
6x28x28
{5, 5}
Asymetric PADDING! H: {2, 2} W:{2, 2}
ACT: Rectifier
Pool Layer
pool1
6x14x14
{2, 2}
Max Pooling
Conv Layer
conv2
16x10x10
{5, 5}
Asymetric PADDING! H: {2, 2} W:{2, 2}
ACT: Rectifier
Pool Layer
pool2
16x5x5
{2, 2}
Max Pooling
Conv Layer
conv3
120x1x1
{5, 5}
Asymetric PADDING! H: {2, 2} W:{2, 2}
ACT: Rectifier
Dropout Layer
conv3.drop
120x1x1
Fc Layer
fc1
84x1x1
ACT: Rectifier
Dropout Layer
fc1.drop
84x1x1
Fc Layer
fc2
10x1x1
ACT: Linear
Softmax Layer
softmax
10x1x1

lenet.ini

(continued from previous page)

```
def conv_conf():
    return n2d2.ConfigSection(
        activation=n2d2.activation.Linear(),
        no_bias=True,
        weights_solver=n2d2.solver.SGD(**solver_conf),
        bias_solver=n2d2.solver.SGD(**solver_conf),
        quantizer=n2d2_ip.quantizer.SATCell(
            apply_scaling=False, # No scaling needed because each conv is followed by␣
↪batch-normalization layers
            apply_quantization=False, # Only clamp mode for the 1st step
    ),)
def fc_conf():
    return n2d2.ConfigSection(
        activation=n2d2.activation.Linear(),
        no_bias=True,
        weights_solver=n2d2.solver.SGD(**solver_conf),
        bias_solver=n2d2.solver.SGD(**solver_conf),
        quantizer=n2d2_ip.quantizer.SATCell(
            apply_scaling=True, # Scaling needed for Full-Connected
            apply_quantization=False, # Only clamp mode for the 1st step
        ),
    )
def bn_conf():
    return n2d2.ConfigSection(
        activation=n2d2.activation.Rectifier(),
        scale_solver=n2d2.solver.SGD(**solver_conf),
        bias_solver=n2d2.solver.SGD(**solver_conf),
    )
```

Once we have defined the global parameters for each cell, we can define our `LeNet` model.

```
print("\n### Loading Model ###")
model = n2d2.cells.Sequence([
    Conv(1, 6, kernel_dims=[5, 5], **conv_conf()),
    BatchNorm2d(6, **bn_conf()),
    Pool2d(pool_dims=[2, 2], stride_dims=[2, 2], pooling="Max"),
    Conv(6, 16, [5, 5], **conv_conf()),
    BatchNorm2d(16, **bn_conf()),
    Pool2d(pool_dims=[2, 2], stride_dims=[2, 2], pooling="Max"),
    Conv(16, 120, [5, 5], **conv_conf()),
    Dropout(name="Conv3.Dropout"),
    BatchNorm2d(120, **bn_conf()),
    Fc(120, 84, **fc_conf()),
    Dropout(name="Fc1.Dropout"),
    Fc(84, 10, **fc_conf()),
])
print(model)

softmax = n2d2.cells.Softmax(with_loss=True)

loss_function =  n2d2.target.Score(provider)
```

The model defined, we can train it with a classic training loop :

```
print("\n### Training ###")
for epoch in range(nb_epochs):

    provider.set_partition("Learn")
    model.learn()

    print("\n# Train Epoch: " + str(epoch) + " #")

    for i in range(math.ceil(database.get_nb_stimuli('Learn')/batch_size)):

        x = provider.read_random_batch()
        x = model(x)
        x = softmax(x)
        x = loss_function(x)
        x.back_propagate()
        x.update()

        print("Example: " + str(i * batch_size) + ", loss: "
              + "{0:.3f}".format(x[0]), end='\r')


    print("\n### Validation ###")

    loss_function.clear_success()

    provider.set_partition('Validation')
    model.test()

    for i in range(math.ceil(database.get_nb_stimuli('Validation') / batch_size)):
        batch_idx = i * batch_size

        x = provider.read_batch(batch_idx)
        x = model(x)
        x = softmax(x)
        x = loss_function(x)

        print("Validate example: " + str(i * batch_size) + ", val success: "
              + "{0:.2f}".format(100 * loss_function.get_average_success()) + "%", end='\
↪r')


print("\n\n### Testing ###")

provider.set_partition('Test')
model.test()

for i in range(math.ceil(provider.get_database().get_nb_stimuli('Test')/batch_size)):
    batch_idx = i*batch_size

    x = provider.read_batch(batch_idx)
    x = model(x)
    x = softmax(x)
    x = loss_function(x)
```

(continues on next page)

(continued from previous page)

```
    print("Example: " + str(i * batch_size) + ", test success: "
          + "{0:.2f}".format(100 * loss_function.get_average_success()) + "%", end='\r')

print("\n")
```

Then, we can export the weights we have learned in order to use them for the second step.

```
### Exporting weights ###
x.get_deepnet().export_network_free_parameters("./weights_clamped")
```

If you check the generated file : *conv3_weights_quant.distrib.png* you should see the *clamped weights*.

Part 2 : Quantized LeNet with SAT

Now that we have learned clamped weights, we will quantize our network.

You can get the script used in this example by clicking here : `LeNet quantization example`.

To do so, we will create a second script. We can begin by importing the `MNIST` database and create a `dataprovider` just like in the previous section.

Then we will copy the `n2d2.ConfigSection` from the previous section and add a quantizer argument.

```
solver_conf = n2d2.ConfigSection(
    learning_rate=0.05,
    learning_rate_policy="None",
    momentum=0.0,
    decay=0.0,
)
def conv_conf():
    return n2d2.ConfigSection(
        activation=n2d2.activation.Linear(),
        no_bias=True,
        weights_solver=n2d2.solver.SGD(**solver_conf),
        bias_solver=n2d2.solver.SGD(**solver_conf),
        quantizer=n2d2_ip.quantizer.SATCell(
            apply_scaling=False,
            apply_quantization=True, # ApplyQuantization is now set to True
            range=15, # Conv is now quantized in 4-bits range (2^4 - 1)
    ))
def fc_conf():
    return n2d2.ConfigSection(
        activation=n2d2.activation.Linear(),
        no_bias=True,
        weights_solver=n2d2.solver.SGD(**solver_conf),
        bias_solver=n2d2.solver.SGD(**solver_conf),
        quantizer=n2d2_ip.quantizer.SATCell(
            apply_scaling=True,
            apply_quantization=True, # ApplyQuantization is now set to True
            range=15, # Fc is now quantized in 4-bits range (2^4 - 1)
    ))
def bn_conf():
```

(continues on next page)

(continued from previous page)

```
  return n2d2.ConfigSection(
      activation=n2d2.activation.Linear(
          quantizer=n2d2_ip.quantizer.SATAct(
              alpha=6.0,
              range=15, # -> 15 for 4-bits range (2^4-1)
      )),
      scale_solver=n2d2.solver.SGD(**solver_conf),
      bias_solver=n2d2.solver.SGD(**solver_conf),
  )
```

The configuration done, we will defined our new network.

Note: The first `Convolution` and last `Fully Connected` layer have differents parameters because we will quantize them in 8-bits instead of 4-bit as it is a common practice.

```
### Creating model ###
print("\n### Loading Model ###")
model = n2d2.cells.Sequence([
    Conv(1, 6, kernel_dims=[5, 5],
        activation=n2d2.activation.Linear(),
        no_bias=True,
        weights_solver=n2d2.solver.SGD(**solver_conf),
        bias_solver=n2d2.solver.SGD(**solver_conf),
        quantizer=n2d2_ip.quantizer.SATCell(
            apply_scaling=False,
            apply_quantization=True, # ApplyQuantization is now set to True
            range=255, # Conv_0 is now quantized in 8-bits range (2^8 - 1)
    )),
    BatchNorm2d(6, **bn_conf()),
    Pool2d(pool_dims=[2, 2], stride_dims=[2, 2], pooling="Max"),
    Conv(6, 16, [5, 5], **conv_conf()),
    BatchNorm2d(16, **bn_conf()),
    Pool2d(pool_dims=[2, 2], stride_dims=[2, 2], pooling="Max"),
    Conv(16, 120, [5, 5], **conv_conf()),
    Dropout(name="Conv3.Dropout"),
    BatchNorm2d(120, **bn_conf()),
    Fc(120, 84, **fc_conf()),
    Dropout(name="Fc1.Dropout"),
    Fc(84, 10,
        activation=n2d2.activation.Linear(),
        no_bias=True,
        weights_solver=n2d2.solver.SGD(**solver_conf),
        bias_solver=n2d2.solver.SGD(**solver_conf),
        quantizer=n2d2_ip.quantizer.SATCell(
            apply_scaling=True,
            apply_quantization=True, # ApplyQuantization is now set to True
            range=255, # Fc_1 is now quantized in 8-bits range (2^8 - 1)
    )),
])
print(model)
```

The model created we can import the learned parameter.

```
# Importing the clamped weights
model.import_free_parameters("./weights_clamped", ignore_not_exists=True)
```

The model is now ready for a training (you can use the training loop presented in the previous section).

The training done, you can save the new quantized weights with the following line :

```
### Exporting weights ###
x.get_deepnet().export_network_free_parameters("./new_weights")
```

If you check the generated file : *conv3_weights_quant.distrib.png* you should see the *quantize weights*.

You can fuse `BatchNorm` and `Conv` layers by using the following line :

```
### Fuse ###
n2d2_ip.quantizer.fuse_qat(x.get_deepnet(), provider, "NONE")
x.get_deepnet().draw_graph("./lenet_quant.py")
```

You can check the generated file : *lenet_quant.py.png* which should looks like the fig *QAT without Batchnorm*.

10.6 Results

10.6.1 Training Time Performances

Quantization-aware training induces intensive operations at training phase. Forward and backward phases require a lot of additional arithmetic operations compared to the standard floating-point training. The cost of operations involved in quantization-aware training method directly impacts the training time of a model.

To mitigate this loss at training time, that can be a huge handicap to quantize your own model, N2D2 implements CUDA kernels to efficiently perform these additional operations.

Here we estimate the training time per epoch for several well-known models on `ImageNet` and `CIFAR-100` datasets. These data are shared for information purpose, to give you a realistic idea of the necessary time required to quantize your model. It relies on a lot of parameters like the dimension of your input data, the size of your dataset, pre-processing, your server/computer set-up installation, etc…

ResNet-18 Per Epoch Training Time		
Quantization Method - Database	GPU Configuration	
	`A100` x1	`2080 RTX Ti` x1
`SAT - ImageNet`	15 min	40 min
`SAT - CIFAR100`	20 sec	1:15 min
`LSQ - ImageNet`	15 min	55 min

MobileNet-v1 Per Epoch Training Time		
Quantization Method - Database	GPU Configuration	
	`A100` x1	`2080 RTX Ti` x1
`SAT - ImageNet`	25 min	45 min
`SAT - CIFAR100`	30 sec	1:30 min

MobileNet-v2 Per Epoch Training Time		
Quantization Method - Database	GPU Configuration	
	A100 x1	2080 RTX Ti x1
SAT - ImageNet	30 min	62 min
SAT - CIFAR100	1:15 min	2:10 min
LSQ - ImageNet	33 min	xx min

Inception-v1 Per Epoch Training Time		
Quantization Method - Database	GPU Configuration	
	A100 x1	2080 RTX Ti x1
SAT - ImageNet	40 min	80 min
SAT - CIFAR100	35 sec	2:20 min
LSQ - ImageNet	25 min	xx min

These performances indicators have been realized with typical Float32 datatype. Even if most of the operations used in the quantizations methods provides support for Float16 (half-precision) datatypes we recommend to not use it. In our experiments we observes performances differences compared to the Float32 datatype mode. These differences comes from gradient instability when datatype is reduced to Float16.

10.6.2 MobileNet-v1

Results obtained with the SAT method (~150 epochs) under the integer only mode :

MobileNet-v1 - SAT ImageNet Performances - Integer ONLY					
Top-1 Precision	Quantization Range (bits)		Parameters	Memory	Alpha
	Weights	Activations			
72.60 %	8	8	4 209 088	4.2 MB	1.0
71.50 %	4	8	4 209 088	2.6 MB	1.0
65.00 %	2	8	4 209 088	1.8 MB	1.0
60.15 %	1	8	4 209 088	1.4 MB	1.0
70.90 %	4	4	4 209 088	2.6 MB	1.0
64.60 %	3	3	4 209 088	2.2 MB	1.0
57.00 %	2	2	4 209 088	1.8 MB	1.0
69.00 %	8	8	3 156 816	2.6 MB	0.75
69.00 %	4	8	3 156 816	1.6 MB	0.75
65.60 %	3	8	3 156 816	1.4 MB	0.75
58.70 %	2	8	3 156 816	1.2 MB	0.75
53.80 %	1	8	3 156 816	0.9 MB	0.75
64.70 %	8	8	1 319 648	1.3 MB	0.5
63.40 %	4	8	1 319 648	0.9 MB	0.5
51.70 %	2	8	1 319 648	0.7 MB	0.5
44.00 %	1	8	1 319 648	0.6 MB	0.5
63.70 %	4	4	1 319 648	0.9 MB	0.5
54.80 %	3	3	1 319 648	0.8 MB	0.5
42.80 %	2	2	1 319 648	0.7 MB	0.5
55.01 %	8	8	463 600	0.4 MB	0.25
50.02 %	4	8	463 600	0.3 MB	0.25
46.80 %	3	8	463 600	0.3 MB	0.25
48.80 %	4	4	463 600	0.3 MB	0.25

10.6.3 MobileNet-v2

Results obtained with the SAT method (~150 epochs) under the integer only mode :

MobileNet-v2 - SAT ImageNet Performances - Integer ONLY					
Top-1 Precision	Quantization Range (bits)		Parameters	Memory	Alpha
	Weights	Activations			
72.5 %	8	8	3 214 048	3.2 MB	1.0
58.59 %	1	8	3 214 048	1.3 MB	1.0
70.93 %	4	4	3 214 048	2.1 MB	1.0

Results obtained with the LSQ method on 1 epoch :

MobileNet-v2 - LSQ ImageNet Performances - 1-Epoch					
Top-1 Precision	Quantization Range (bits)		Parameters	Memory	Alpha
	Weights	Activations			
70.1 %	8	8	3 214 048	3.2 MB	1.0

10.6.4 ResNet

Results obtained with the SAT method (~150 epochs) under the integer only mode :

ResNet - SAT ImageNet Performances - Integer ONLY					
Top-1 Precision	Quantization Range (bits)		Parameters	Memory	Depth
	Weights	Activations			
70.80 %	8	8	11 506 880	11.5 MB	18
67.6 %	1	8	11 506 880	1.9 MB	18
70.00 %	4	4	11 506 880	6.0 MB	18

Results obtained with the LSQ method on 1 epoch :

ResNet - LSQ ImageNet Performances - 1-Epoch					
Top-1 Precision	Quantization Range (bits)		Parameters	Memory	Depth
	Weights	Activations			
70.20 %	8	8	11 506 880	11.5 MB	18

10.6.5 Inception-v1

Results obtained with the SAT method (~150 epochs) under the integer only mode :

Inception-v1 - SAT ImageNet Performances - Integer ONLY				
Top-1 Precision	Quantization Range (bits)		Parameters	Memory
	Weights	Activations		
73.60 %	8	8	6 600 006	6.6 MB
68.60 %	1	8	6 600 006	1.7 MB
72.30 %	4	4	6 600 006	3.8 MB
68.50 %	1	4	6 600 006	1.7 MB
67.50 %	1	3	6 600 006	1.7 MB
63.30 %	1	2	6 600 006	1.7 MB
47.36 %	1	1	6 600 006	1.7 MB

Results obtained with the LSQ method on 1 epoch :

Inception-v1 - LSQ ImageNet Performances - 1-Epoch				
Top-1 Precision	Quantization Range (bits)		Parameters	Memory
	Weights	Activations		
72.60 %	8	8	6 600 006	6.6 MB

CHAPTER

ELEVEN

PRUNING

11.1 Getting Started

N2D2 provides a pruning module to perform pruning operations on your model in order to reduce its memory footprint. The module works like the QAT module i.e. it is possible to carry out trainings with pruned weights in order to improve the performance of the network. Only weights can be pruned so far.

11.2 Example with Python

Example of code to use the `n2d2.quantizer.PruneCell` in your scripts:

```
for cell in model:
    ### Add Pruning ###
    if isinstance(cell, n2d2.cells.Conv) or isinstance(cell, n2d2.cells.Fc):
        cell.quantizer = n2d2.quantizer.PruneCell(prune_mode="Static", threshold=0.3,␣
↪prune_filler="IterNonStruct")
```

Some explanations with the differents options of the `n2d2.quantizer.PruneCell` :

11.2.1 Pruning mode

3 modes are possible:

- Identity: no pruning is applied to the cell
- Static: all weights of the cell are pruned to the requested `threshold` at initialization
- Gradual: the weights are pruned to the `start` threshold at initialization and at each update of the current threshold, it is increased by `gamma` until it reaches `threshold`. By default, the update is performed at the end of each epoch (possible to change it with `stepsize`)

Warning: if you use `stepsize`, please indicate the number of steps and not the number of epochs. For example, to update each two epochs, write:

```
n2d2.quantizer.PruneCell(prune_mode="Gradual", threshold=0.3, stepsize=2*DATASET_SIZE)
```

Where `DATASET_SIZE` is the size of the dataset you are using.

11.2.2 Pruning filler

2 fillers are available to fill the masks:

- Random: The masks are filled randomly
- IterNonStruct: all weights below than the `delta` factor are pruned. If this is not enough to reach `threshold`, all the weights below 2 "delta" are pruned and so on...

Important: With `n2d2.quantizer.PruneCell`, `quant_mode` and `range` are not used.

11.3 Example with INI file

The common set of parameters for any kind of Prune Quantizer.

Option [default value]	Description
`QWeight`	Quantization / Pruning method, choose `Prune` to activate the Pruning mode.
`QWeight.PruningMode` [`Identity`]	Pruning mode, can be `Identity`, `Static` or `Gradual`
`QWeight.PruningFiller` [`Random`]	Pruning filler for the weights, can be `Random`, `IterNonStruct` or `None`
`QWeight.Threshold` [`0.2`]	Weight threshold to be pruned, 0.2 means 20% for example
`QWeight.Delta` [`0.001`]	Factor for iterative pruning, use it with `IterNonStruct` pruning filler
`QWeight.StartThreshold` [`0.1`]	Starting threshold, use it with `Gradual` pruning mode
`QWeight.StepSizeThreshold` [`0`]	Step size for the threshold update, use it with `Gradual` pruning mode
`QWeight.GammaThreshold` [`0.05`]	Value to add to current threshold during its update, use it with `Gradual` pruning mode

Example of code to use the *Prune Quantizer* in your scripts:

```
[conv1]
Input=sp
Type=Conv
KernelDims=5 5
NbOutputs=6
ActivationFunction=Rectifier
WeightsFiller=HeFiller
ConfigSection=common.config
QWeight=Prune
QWeight.PruningMode=Static
QWeight.PruningFiller=IterNonStruct
QWeight.Threshold=0.3
QWeight.StartThreshold=0.1
QWeight.GammaThreshold=0.1
```

All explanations in relation to the parameters of Prune Quantizer are provided in the python section of this page.

CHAPTER

TWELVE

EXPORT: C++

Export type: `CPP`
C++ export using OpenMP.

```
n2d2 MobileNet_ONNX.ini -seed 1 -w /dev/null -export CPP
```

12.1 Principle

The C++ export is the reference N2D2 export, which implements all the export features available in N2D2, like post-training quantization or quantization aware training.

Summary of the main features of a C++ export:

- Standalone C++11 compliant project;
  - No C++ exception (often disabled on embedded code);
  - No <stream> library (which is memory bloated).
- Fully templated compute kernels;
- Fully inlined compute kernels;
- No dynamic memory allocation;
- Memory alignment support;
- OpenMP parallelism.

Data order
The input of a layer is ordered by Height-Width-Channels (HWC) and the weights of the kernel for a convolution by Output-Height-Width-Channels (OHWC). This order allows us to do read *kernel_width* × *nb_channels* inputs and weights sequentially in memory to do the necessary MACs.

Templated layer parameters
The current export uses C++ templates heavily, most of the parameters of the layers are passed as template parameters. This allows the compiler to better optimize the code and make it easier to unroll the loops. It comes at the cost of a larger compiled binary.

Force inline
Most of the methods are forced to be inlined. As previously this increases the binary size to provide a faster inference.

Loop boundaries
The boundaries of the loops are fixed at compile time through the template parameters. If some steps in a loop must be skipped an if and continue are used inside the loop. It results in better results than having variable loop boundaries.

12.1.1 Graph optimizations

- Weights are equalized between layers when possible;
- `BatchNorm` is automatically fused with the preceding `Conv` or `Fc` when possible;
- `Padding` layers are fused with `Conv` when possible;
- `Dropout` layers are removed.

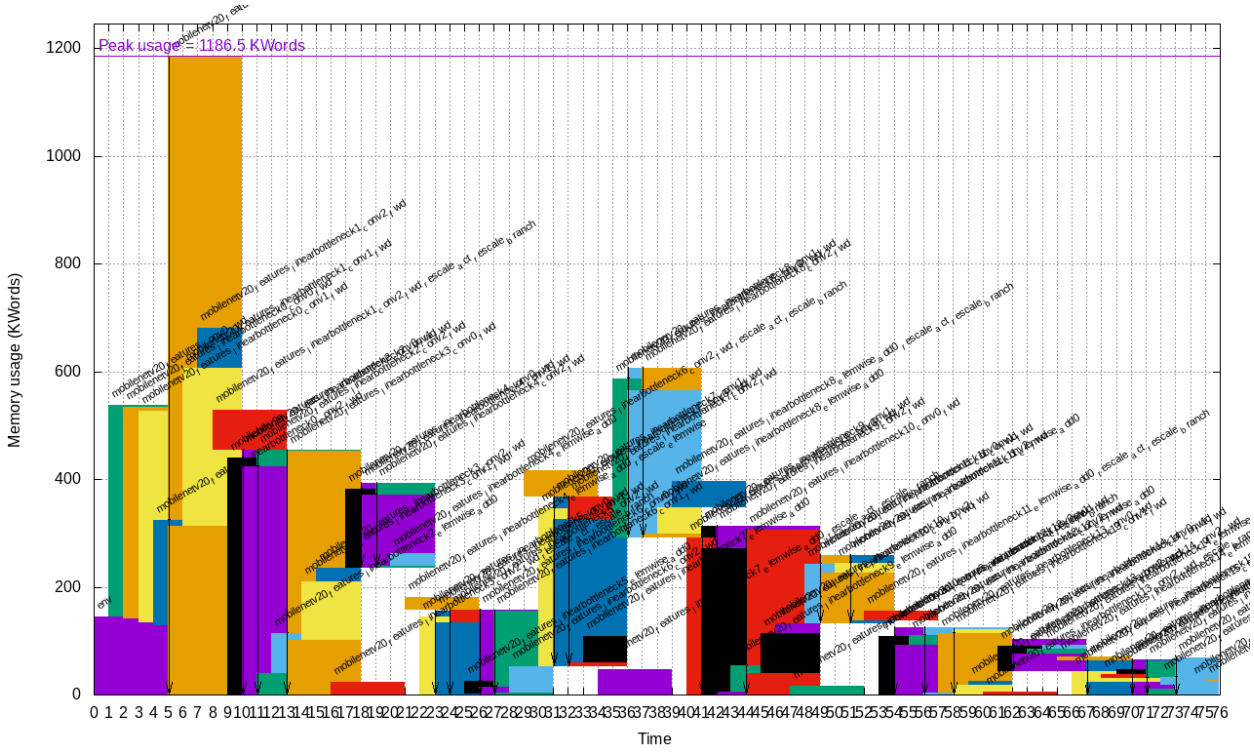
12.1.2 Memory optimizations

In the C++ export, all the memory is allocated statically at compilation time.

The following memory optimization features are integrated in this export:

- **Strided buffers:** concatenation can be done directly in memory, no memory copy is needed;
- **Memory wrapping:** memory buffers are re-used when possible (memory wrapping or in-place).

For example, the memory mapping of each layer in a global memory space for MobileNet v2 is shown below (generated automatically during an export):



In this example, the largest layer memory buffer overlaps with the memory buffer of the preceding layer thanks to the `OptimizeBufferMemory` option (see the next section).

12.1.3 Export parameters

Extra parameters can be passed during export using the `-export-parameters params.ini` command line argument. The parameters must be saved in an INI-like file.

List of available parameters:

Argument [default value]	Description
`IncludeInputInBuffer` [1]	If true (1), include the input buffer in the memory mapping
`OptimizeBufferMemory` [1]	If true (1), try to re-use memory spaces using wrapping and in-place
`OptimizeNoBranchConcat` [1]	If true (1), concatenation is done directly in memory when possible
`MemoryAlignment` [1]	Default memory alignment (in bytes)
`MemoryManagerStrategy` [`OptimizeMaxLifetimeMaxSizeFirst`]	Optimization strategy for static memory allocation

12.2 Example

```
n2d2 MobileNet_ONNX.ini -seed 1 -w weights_validation -export CPP -fuse -nbbits 8 -calib␣
↪-1 -db-export 100 -test
```

This command generates a C++ project in the sub-directory `export_CPP_int8`. This project is ready to be compiled with a `Makefile`.

Note: The `Softmax` layer is not supported by the C++ export and must be removed before export, as it is not compatible with low precision integer computing.

CHAPTER

THIRTEEN

EXPORT: C++/STM32

Export type: `CPP_STM32`
C++ export for STM32.

```
n2d2 MobileNet_ONNX.ini -seed 1 -w /dev/null -export CPP_STM32
```

13.1 Principle

This export inherit the properties and optimizations from the C++ export, but includes optimized kernels for the Cortex-M4 and the Cortex-M7. Please refer to the *Export: C++* for the available export parameters.

SIMD
The `SMLAD` intrinsic is used to do two 16-bit signed integers multiplications with accumulation. To extend the 8-bit data to the necessary 16-bit, the `XTB16` intrinsic is used.

Loop unrolling
The unrolling of the loops can be done with `#pragma GCC unroll NB_ITERATIONS` but it does not always perform as well as expected. Some loops are manually unrolled instead using C++ templates. This increases the size of the compiled binary further but it provides a faster inference.

Usage of intrinsics
Intrinsics provided by ARM are preferred to normal library methods calls when possible. For example the `SSAT` and `USAT` intrinsics are used to clamp the output value resulting in better results than a naive call to the std::clamp method.

13.2 Usage

```
n2d2 MobileNet_ONNX.ini -seed 1 -w weights_validation -export CPP_STM32 -fuse -nbbits 8 -
↪calib -1 -db-export 100 -test
```

This command generates a C++ project in the sub-directory `export_CPP_STM32_int8`. This project is ready to be cross-compiled with a `Makefile`, using the *GNU Arm Embedded Toolchain* (which provides the `arm-none-eabi-gcc` compiler).

`make`
To cross-compile the project using the *GNU Arm Embedded Toolchain*. An ELF binary file is generated in `bin/n2d2_stm32.elf`.

`make flash`
To flash the board using OpenOCD with the previously generated `bin/n2d2_stm32.elf` binary. In the provided Makefile, the default OpenOCD location is `/usr/local/bin/openocd` and the default script is

`stm32h7x3i_eval.cfg`, for the STM32H7x3I evaluation board family. These can be changed in the first lines of the Makefile.

CHAPTER

FOURTEEN

EXPORT: TENSORRT

Export type: `CPP_TensorRT`

C++ export using TensorRT.

```
n2d2 MobileNet_ONNX.ini -seed 1 -w /dev/null -export CPP_TensorRT -nbbits -32
```

Warning: The calibration for this export is done using the tools provided by NVIDIA. For this reason, you cannot calibrate when exporting your network. You need to use the export to calibrate your network.

14.1 Informations

In order to exploit TensorRT optimizations, the N2D2 Framework provide a code generator linked with a C++/Python API that gives access to TensorRT methods, I/O handling and specifics control. The generated code is provided as a standalone code with its own compilation environment under a Makefile format. Moreover a benchmark environment with stimuli from the test dataset is given to evaluates execution time performances of your model.

This allow a low level of dependency, only TensorRT, CUDA, cuDNN, cuBLAS and GCC are needed. We recommended you to ensure the correct compatibility of your installation by referring to the TensorRT archive page: https://docs.nvidia.com/deeplearning/tensorrt/archives/index.html Follow the support matrix section of your TensorRT version, notice that TensorRT export have been tested from TensorRT 2.1 to TensorRT 8.2.3 versions.

The TensorRT library includes implementation for the most common deep learning layers, but strong limitations are known depending of the TensorRT version. For example, TensorRT provide a support to the well-known resize layer since version 6.0.1. This layer is widely use for decoder, segmentation and detector tasks. For inferior version a support have been integrated under a plugin layer. The TensorRT plugin layers allow the application to implement not supported layers. You can find additional informations about how to implements new plugin layers here : https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html#add_custom_layer

The plugin layers that N2D2 TensorRT generator implements are available in the folder `export/CPP_TensorRT/include/plugins/`. These layers are used by the N2D2 TensorRT generator when TensorRT doesn't provide support to a requested layers.

14.2 Export parameters

Extra parameters can be passed during export using the `-export-parameters params.ini` command line argument. The parameters must be saved in an INI-like file.

List of available parameters:

Argument [default value]	Description
`GenStimuliCalib` [1]	If true (1), generate calibration files, necessary for 8-bits precision. Beware that calibration files may take a lot of disk space!

14.3 Benchmark your TensorRT Model - C++ Benchmark

The TensorRT export is given with a C++ benchmark ready to be used. The benchmark program is able to evaluates the applicative performances of your model on the test dataset exported under the `stimuli` folder at export time. A per-layer execution time analysis is also performed to evaluates your mode latency and identify potential bottleneck. Moreover different numerical precision supporter by NVIDIA GPU can be evaluates in order to assess potential acceleration factor and eventual applicative performances losses.

When numerical precision is set to `8` bits for benchmark, the program will use the calibration files exported under the `batches_calib` folder at export time. The calibration files also corresponds to the test stimuli pre-processes for the `IInt8EntropyCalibrator2` that implement TensorRT. You can find more informations about the INT8 calibration procedure with TensorRT here : https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html#optimizing_int8_c

The command to compile and execute the C++ TensorRT Benchmark under a `FP32` precision is :

```
make
cd export_CPP_TensorRT_float32/
./bin/n2d2_tensorRT_test -nbbits -32
```

To launch the Benchmark in `FP16` (half precision) use this command :

```
./bin/n2d2_tensorRT_test -nbbits -16
```

To launch the Benchmark in `INT8` use this command :

```
./bin/n2d2_tensorRT_test -nbbits 8
```

List of the program option related to the TensorRT C++ benchmark:

Option [default value]	Description
`-batch` [1]	Size of the batch to use
`-dev` [0]	CUDA Device ID selection
`-stimulus` [NULL]	Path to a specific input stimulus to test. For example: -stimulus $/stimulus/env0000.pgm$ command will test the file env0000.pgm of the stimulus folder.
`-prof`	Activates the layer wise profiling mechanism. This option can decrease execution time performance.
`-iter-build` [1]	Sets the number of minimization build iterations done by the tensorRT builder to find the best layer tactics.
`-nbbits` [-32]	Number of bits used for computation. Value -32 for Full FP32 bits configuration, -16 for Half FP16 bits configuration and 8 for INT8 bits configuration. When running INT8 mode for the first time, the TensorRT calibration process can be very long. Once generated the generated calibration table will be automatically reused. Supported compute mode in function of the compute capability are provided here: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities .
`-calib-cache`	Path and name to the calibration file generated by TensorRT calibrator when precision is INT8. Must be compatible with the TensorRT Entropy Calibrator version used to calibrate.
`-calib-folder` [batches_calib]	Path to the calibration data samples. This is mandatory when precision is set to INT8 and if no calibration file cache is load.

14.3.1 Analyse the execution performances of your TensorRT Model (FP32)

Here is a small example that described how to report the per-layer analysis on execution time.

Launch the Benchmark with the `-prof` argument :

```
./bin/n2d2_tensorRT_test -prof
```

At the end of the execution the performances analysis is displayed in your screen :

```
(19%)  ****************************************** CONV1 + CONV1_ACTIVATION: 0.0219467 ms
(05%)  ************ POOL1: 0.00675573 ms
(13%)  ***************************** CONV2 + CONV2_ACTIVATION: 0.0159089 ms
(05%)  ************ POOL2: 0.00616047 ms
(14%)  ******************************* CONV3 + CONV3_ACTIVATION: 0.0159713 ms
(19%)  ****************************************** FC1 + FC1_ACTIVATION: 0.0222242 ms
(13%)  ***************************** FC2: 0.0149013 ms
(08%)  ****************** SOFTMAX: 0.0100633 ms
Average profiled tensorRT process time per stimulus = 0.113932 ms
```

You can evaluates impact of the performances for various batch size and the different numerical precision supported.

14.4 Deploy your TensorRT Model in Application

The TensorRT export is provided with a C++ and a python interface. The python interface is accessible through a wrapper to the C++ API method and linked with the libboost-python librabry.

You can integrates your model in your application environment as a library thanks to this API.

The command to compile the TensorRT export as a C++ library is :

```
make WRAPPER_CPP=1
```

The library of your TensorRT model is provided under the name `libn2d2_tensorRT_inference.so` locate at `bin/` folder.

The command to compile your TensorRT export as a Python3.6m library is :

```
make WRAPPER_PYTHON=3.6m
```

The python library of your TensorRT model is then provided under the name `N2D2.so` locate at `bin/` folder.

Method accessible through C++ or Python API are listed and detailled here:

Return Type	C++ API	Python API	Args Type	ArgsName(DefaultValue)	Description	Comments
	N2D2::Network()	N2D2.Network()			TensorRT DNN object creation	
void	setMaxBatchSize	setMaxBatchSize	size_t	batchsize(1)	Maximum batchsize for setting the internal tensorrt graph memory usage limit	Use before run initialize()
void	setDeviceID	setDeviceID	size_t	device(0)	Device ID on which run the TensorRT model	Use before run initialize()
void	setPrecision	setPrecision	int	precision(-32)	Numerical Precision to use: -32 for float, -16 for half float, 8 for int8	Use before run initialize()
void	useDLA	useDLA	bool	useDla(False)	If True, use the first DLA core for every possible layers	Use before run initialize()
void	setMaxWorkSpaceSize	setMaxWorkSpaceSize	int64_t	maxWorkSpaceSize(1073741824)	Size of the workspace, influence the optimisations done by NVIDIA	Use before run initialize()
void	setDetectorThresholds	setDetectorThresholds	float*, uint	thresholds, lengthThreshold	Set the confidences thresholds of a detector output. Bypass the internal thresholds from the exported model	Use before run initialize()
void	setDetectorNMS	setDetectorNMS	double	thresholdNms	Set the threshold for non-maxima suppression range (from 0.0 to 1.0) of a detector output. Bypass the internal thresholds from the exported model	Use before run initialize()
void	setInferenceEngine	setInferenceEngine	string	enginePath	Path of a serialized and optimized TensorRT plan file. The serialized plan file are not portable across platforms or TensorRT versions and are specific to the exact GPU model they were built on	Use before run initial-

CHAPTER

FIFTEEN

EXPORT: DNEURO

N2D2-IP only: available upon request.

Export type: `DNeuro_V2`
DNeuro RTL export for FPGA.

```
n2d2 MobileNet_ONNX.ini -seed 1 -w /dev/null -export DNeuro_V2
```

15.1 Introduction

DNeuro is a synthetizable dataflow architecture, optimized for deep convolutional neural networks (CNN). It allows a fine grain allocation control of the DSP and memory resources, for each layer in a network. Globally, the FPGA resource usage can be maximized for a given network topology in order to minimize its latency.

The main features of the DNeuro are:

- Data flow architecture requiring few memory (potentially **no DDR**);
- Very high use rate of the DSP per cycle (> 90%);
- Configurable precision (integers from 2 to 16 bits, typically 8 bits);
- Up to 4 MAC/DSP operations per cycle.

The DNeuro is composed of specialized computing blocs, corresponding to specific type and configuration of layers (convolution, max pooling…), that can be chained to form a full neural network. The bloc allocation and chaining is done automatically with N2D2.

15.1.1 Interface

The DNeuro interface is extremely simple and behaves like a pipeline/FIFO.

An example of the top-level DNeuro RTL entity is described below, for one input channel and 3 output channels:

```
-- Input size: 1*640*480
-- Output size: 3*80*60
entity network is
generic (
    constant G_BATCH_SIZE: positive := 1;
    constant G_FIFO_DEPTH: positive := 1;
    constant G_DATA_LENGTH: positive := 8;
    constant G_ACC_S_LENGTH: positive := 18;
    constant G_NB_OUTPUTS_INST_N_1_ENV: positive  := 1;
```

(continues on next page)

(continued from previous page)

```
    constant G_NB_OUTPUTS_MERG_N_1_ENV: positive  := 1
);
port (
    clk             : in std_logic;
    rstn            : in std_logic;
    i_data          : in std_logic_vector ((G_DATA_LENGTH*G_BATCH_SIZE)-1 downto 0);
    i_valid_data    : in std_logic;
    o_en            : out std_logic;
    o_data          : out std_logic_vector ((3*G_DATA_LENGTH*G_BATCH_SIZE)-1 downto 0);
    o_valid_data    : out std_logic;
    i_en            : in std_logic
);
end network;
```

15.1.2 Supported layers

Layer type	Support	Comments
Dropout	n.a.	removed during export
Fc	✓	implemented with Conv during export
InnerProduct → see Fc		
Transformation	✗	
BatchNorm	n.a.	merged with Conv during export with `-fuse` option
Conv	✓	
Concat → implicit for Conv/Deconv/Pool/Fc		
Deconv	✗	
ElemWise	✓	*Sum* operation only
EltWise → see ElemWise		
Flatten → implicit to Fc/Rbf		
LRN	✗	
Maxout → see Pool		
Padding	✓	merged with Conv/Pool during export
Pool	✓	*Max* operation only
Resize	✓	*NearestNeighbor* mode only
Softmax	✗	
SortLabel → see .Target*		
Unpool	✗	
Upscale → see Resize		
.Target*	✓	top-1 sorting

Activation type	Support	Specificities
Linear	✓	saturated arithmetic
Logistic	✓	saturation approximation, configurable zero, up to two configurable thresholds
ReLU → see Rectifier		
bReLU → see Rectifier		
Rectifier	✓	saturated arithmetic (positive values)
Saturation	✓	
Softplus	✗	
Tanh	✗	

15.2 Usage

15.2.1 Simulation

When a network is exported, test vectors are exported automatically too, if the `-db-export` command line option value is > 0 (by default, the full test set is exported). All the test vectors are exported for the C++ emulator, while only the first image is pre-loaded as a test vector for the RTL simulation in the `RTL/NETWORK/TB/network_tb.vhd` file. This testbench is configured with a clock frequency of 100MHz (regardless of the `EstimationFrequency` export parameter). The testbench reads 3 times the same (first) image and outputs the results in the `out_file/out.txt` file, located in `RTL/NETWORK/simu/VsimTOOL` for ModelSim.

```
cd RTL/NETWORK/simu
make vsim
```

15.2.2 C++ emulation

The DNeuro export comes with a C++ bit-accurate emulator.

By default, the provided emulator will use the same parameters as the ones defined in the export. For testing purposes it is possible to change the accumulation size by defining the `ACC_NB_BITS` variable.

```
cd EMULATOR
CXXFLAGS="-DACC_NB_BITS=18" make
./dneuro_v2_emulator
```

When running the emulator, all the exported images are evaluated by default, and a global score is computed from individual images good or bad classifications. It is possible to evaluate a single image with the following command line argument:

```
./dneuro_v2_emulator -stimulus stimuli/env00.ppm
```

The `stimuli/env00.ppm` is an already pre-processed image automatically exported by N2D2 and ready to be feed at the input of the neural network. The emulator generates for each network's layer an output file *layer_name_output.txt* containing the output tensor values of the layer, as expected for the DNeuro IP.

15.2.3 Synthesis

To generate a project ready for synthesis in Vivado or Quartus, use the scripts provided in `RTL/NETWORK/simu/PythonTOOL`. To generate a Vivado project, run:

```
cd RTL/NETWORK/simu
python PythonTOOL/vivadoGenerate.py
```

This script creates a new project in `RTL/NETWORK/simu/VivadoTOOL/project_export_DNeuro`. Do not forget to change the default project's part.

Warning: Do not create a project and add the sources manually, as the sources organization into libraries will not be setup properly: the sources in the directories `CONV_COMMON`, `CONV_Tn_Oy_CHy_K1_Sy_P1` and `CONV_Tn_Oy_CHy_K1_Sy_Pn` must be placed in libraries of the same name!

15.2.4 Export parameters

Extra parameters can be passed during export using the `-export-parameters params.ini` command line argument. The parameters must be saved in an INI-like file.

List of general available parameters:

Argument [default value]	Description
`NbDSPs`	Set the maximum number of DSPs that the network can use on the FPGA
`NbMemoryBytes`	Set the maximum memory, in bytes, that the network can use on the FPGA
`Network` [network]	Name of the top-level HDL entity
`EstimationFrequency` [200]	Frequency used for the FPS estimation given by the export
`AccumulationNbBits` [2.DATA_LENGTH+4]	Number of bits to use for the accumulation

Output map class conversion to RGB settings:

Argument [default value]	Description
`OutputMapToRGB` [0]	If true (1), add an extra layer at the end of the network that converts the output of the network to an RGB output
`OutputMapToRGBBackgroundClass` []	When `OutputMapToRGB` is 1, set the class that is used for background objects. The overlay color for this class will be transparent
`OutputMapToRGBColorMasks` []	When `OutputMapToRGB` is 1, list of colors to use for the classes
`OutputMapToRGBBinaryThresholdUpper` [0]	Upper threshold for binary outputs
`OutputMapToRGBBinaryThresholdLower` [0]	Lower threshold for binary outputs

Internal per layer settings (for debug purpose only!):

Argument [default value]	Description
`RTLType` []	Specific name of the RTL library module to use for this layer
`NbChannelsInstantiation` []	Specific number of channels to instantiate
`NbOutputsInstantiation` []	Specific number of outputs to instantiate
`KernelHeightInstantiation` []	Specific number of kernel height to instantiate
`KernelWidthInstantiation` []	Specific number of kernel width to instantiate

15.2.5 FPGA compatibility tables

Legend:

- should be OK for the standard 224x224 input, but depends on the resolution;
- should be OK for the standard 224x224 input using also the UltraRAM, but depends on the resolution (Xilinx FPGA only);
- M20K memory may be insufficient depending on the resolution;
- there is a better equivalent neural network (see on the same column);
  - using an alternative neural network is possible with a small accuracy loss.

Arria 10

Neural networks compatibility table with DNeuro, in terms of memory requirement.

Arria 10	**GX/SX 160**	**GX/SX 220**	**GX/SX 270**	**GX/SX 320**	**GX/SX 480**	**GX/SX 570**	**GX/SX 660**	**GX 900**	**GX 1150**
M20K (MB)	1.12	1.37	1.87	2.12	3.5	4.37	5.25	5.87	6.62
DSP	156	191	830	985	1,368	1,523	1,688	1,518	1,518
Mult. (MAC/c.)	312	382	1,660	1,970	2,736	3,046	3,376	3,036	3,036
MobileNet_v1_0.25	●	●	●	●	●	●	●	●	●
MobileNet_v1_0.5			•	•	●	●	●	●	●
MobileNet_v1_0.75					•	•	●	●	●
MobileNet_v1_1.0							•	•	•
SqueezeNet_v1.0			••	••	●	●	●	●	●
SqueezeNet_v1.1			••	••	●	●	●	●	●
MobileNet_v2_0.35				•	●	●	●	●	●
MobileNet_v2_0.5					•	●	●	●	●
MobileNet_v2_0.75					•	•	●	●	●
MobileNet_v2_1.0						•	•	•	•
MobileNet_v2_1.3								•	•
MobileNet_v2_1.4									•
AlexNet			•	•	•	•	•	•	•
VGG-16						•	•◦	•◦	•◦
GoogLeNet							•	•	•
ResNet-18							•	•	•
ResNet-34								•	•
ResNet-50									◦

Stratix 10

Neural networks compatibility table with DNeuro, in terms of memory requirement.

Stratix 10	GX/SX 400	GX/SX 650	GX/SX 850	GX/SX 1100	GX/SX 1650	GX/SX 2100	GX/SX 2500	GX/SX 2800
M20K (MB)	3.75	6.12	8.5	13.37	14.25	15.87	24.37	28.62
DSP	648	1,152	2,016	2,592	3,145	3,744	5,011	5,760
Mult. (MAC/c.)	1,296	2,304	4,032	5,184	6,290	7,488	10,022	11,520
MobileNet_v1_0.25	●	●	●	●	●	●	●	●
MobileNet_v1_0.5	●	●	●	●	●	●	●	●
MobileNet_v1_0.75	•	●	●	●	●	●	●	●
MobileNet_v1_1.0		•	●	●	●	●	●	●
SqueezeNet_v1.0	• ●	●	●	●	●	●	●	●
SqueezeNet_v1.1	• ●	●	●	●	●	●	●	●
MobileNet_v2_0.35	●	●	●	●	●	●	●	●
MobileNet_v2_0.5	•	●	●	●	●	●	●	●
MobileNet_v2_0.75	•	●	●	●	●	●	●	●
MobileNet_v2_1.0		•	●	●	●	●	●	●
MobileNet_v2_1.3			•	●	●	●	●	●
MobileNet_v2_1.4			•	●	●	●	●	●
AlexNet	•	•	•	•	•	•	•	•
VGG-16		• ∘	• ∘	• ∘	• ∘	• ∘	• ∘	• ∘
GoogLeNet		•	•	• •	●	●	●	●
ResNet-18		•	•	•	• •	• •	●	●
ResNet-34			•	•	•	•	•	• •
ResNet-50			∘	∘	∘	∘	∘	∘

Zynq UltraScale+

Neural networks compatibility table with DNeuro, in terms of memory requirement.

Zynq UltraScale+	ZU2 EG	ZU3 EG	ZU4 EG	ZU5 EG	ZU6 EG	ZU7 EG	ZU9 EG	ZU11 EG	ZU15 EG	ZU17 EG	ZU19 EG
BRAM (MB)	0.66	0.95	0.56	1.02	3.13	1.37	4.01	2.63	3.27	3.5	4.32
UltraRAM (MB)			1.68	2.25		3.37		2.81	3.93	3.58	4.5
Total RAM (MB)	0.66	0.95	2.24	3.27	3.12	4.74	4.01	5.44	7.2	7.08	8.82
DSP	240	360	728	1,248	1,973	1,728	2,520	2,928	3,528	1,590	1,968
Mult. (MAC/c.)	480	720	1,456	2,496	3,946	3,456	5,040	5,856	7,056	3,180	3,936
MobileNet_v1_0.25	•	●	•	●	●	●	●	●	●	●	●
MobileNet_v1_0.5			•	•	●	•	●	●	●	●	●
MobileNet_v1_0.75				•	•	•	•	•	•	•	•
MobileNet_v1_1.0						•		•	•	•	•
SqueezeNet_v1.0			• • •	•	•	•	●	●	●	●	●
SqueezeNet_v1.1			• • •	•	•	•	●	●	●	●	●
MobileNet_v2_0.35			•	•	•	•	●	•	●	●	●
MobileNet_v2_0.5				•	•	•	●	•	•	•	●
MobileNet_v2_0.75				•	•	•	•	•	•	•	•
MobileNet_v2_1.0						•	•	•	•	•	•
MobileNet_v2_1.3									•	•	•
MobileNet_v2_1.4									•	•	•
AlexNet			•	•	•	•	•	•	•	•	•
VGG-16						• ◦	• ◦	• ◦	• ◦	• ◦	• ◦
GoogLeNet						•	•	•	•	•	•
ResNet-18						•	•	•	•	•	•
ResNet-34									•	•	•
ResNet-50									◦	◦	◦

Kintex UltraScale+

Neural networks compatibility table with DNeuro, in terms of memory requirement.

Kintex UltraScale+	KU3P	KU5P	KU9P	KU11P	KU13P	KU15P
BRAM (MB)	1.58	2.11	4.01	2.63	3.27	4.32
UltraRAM (MB)	1.68	2.25		2.81	3.93	4.5
Total RAM (MB)	3.26	4.36	4.01	5.44	7.2	8.82
DSP	1,368	1,825	2,520	2,928	3,528	1,968
Mult. (MAC/c.)	2,736	3,650	5,040	5,856	7,056	3,936
MobileNet_v1_0.25	●	●	●	●	●	●
MobileNet_v1_0.5	•	•	●	●	●	●
MobileNet_v1_0.75	•	•	•	•	•	•
MobileNet_v1_1.0				•	•	•
SqueezeNet_v1.0	••	•	●	●	●	●
SqueezeNet_v1.1	••	•	●	●	●	●
MobileNet_v2_0.35	•	•	●	•	●	●
MobileNet_v2_0.5	•	•	●	•	•	●
MobileNet_v2_0.75	•	•	•	•	•	•
MobileNet_v2_1.0		•	•	•	•	•
MobileNet_v2_1.3					•	•
MobileNet_v2_1.4					•	•
AlexNet	•	•	•	•	•	•
VGG-16		•	•	•◦	•◦	•◦
GoogLeNet				•	•	•
ResNet-18				•	•	•
ResNet-34					•	•
ResNet-50					◦	◦

15.3 Aerial Imagery Segmentation DEMO

15.3.1 Specifications

Specifications of the Aerial Imagery Segmentation DEMO:

Feature	DEMO	Max.	Description
Input resolution	VGA	720p	
	(640x480)	(1280x720)	
Output resolution	80x60	160x90	Native resolution before upscaling
Precision	INT8	INT8	
Batch	1	2	
NN Complexity	∼1GMAC	∼2.5GMAC	
NN Parameters	∼100k		
Processing speed	∼150 FPS	∼120 FPS	
Objects detected	8		Transport assets:
			aircraft, *large vehicle*
			small vehicle, *ship*
			Ground assets:
			harbor, *sport field*
			swimming pool, *storage tank*
FPGA model	Arria 10 SX 270		
FPGA DSP blocks	830		2 MAC/DSP block with batch 2
FPGA memory	2.17MB		
Mem. usage	1MB	?	
FPGA frequency	200 MHz		
GMAC/s (th.)	166GMAC/s	332GMAC/s	
GMAC/s (real)	∼150GMAC/s	∼300GMAC/s	
MAC/DSP/cycle	0.9	1.8	DSP usage efficiency

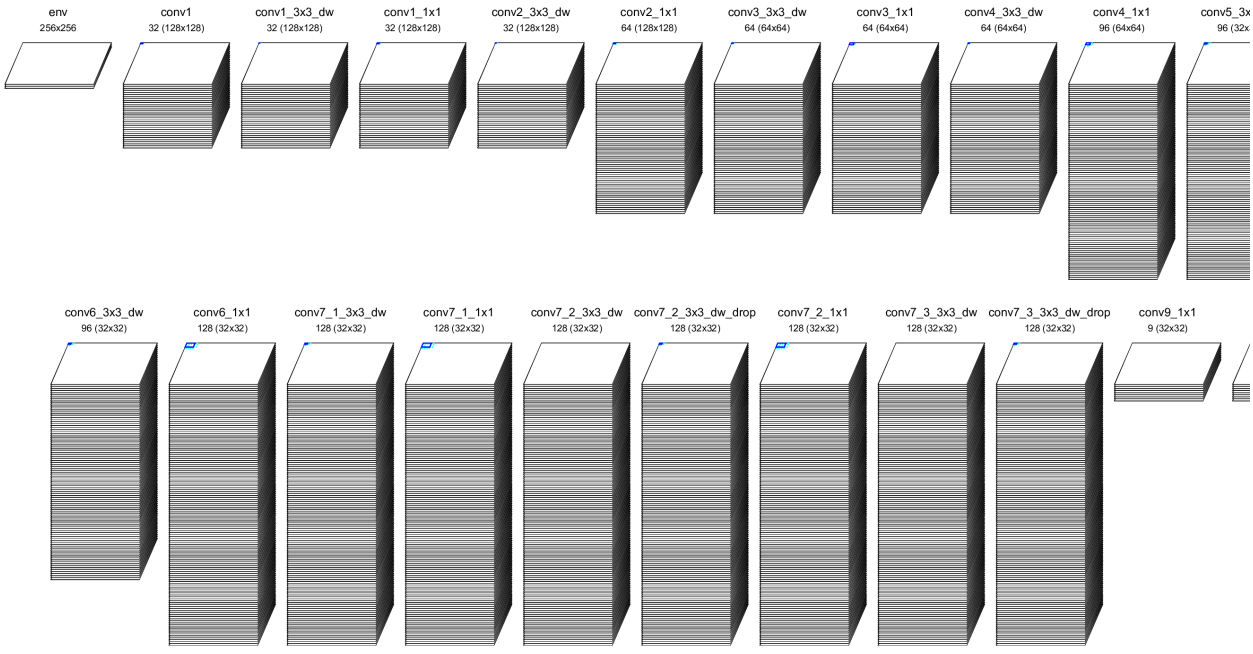


Fig. 1: Neural network used for the application.

15.3.2 Application preview

The application preview is a web-based interface allowing to freely navigate on a map and see the segmentation result in real time. Its main characteristics are:

- Web interface combining the open source *OpenLayers* map visualization API and data from either *IGN-F/Géoportail* or *Microsoft Bing Maps*;
- The neural network is run on a server and the segmentation result is updated and displayed automatically at the right of the aerial view, in real time;
- The same interface is run on the tablet computer with the aerial view map in full screen, to be send via to the DNeuro via the HDMI interface.

To generate the application preview, starting from the learned project in N2D2, create a TensorRT export with the following commands:

```
n2d2 MobileNet_DEMO.ini -export CPP_TensorRT -nbbits -32 -db-export 0
cd export_CPP_TensorRT_float32
make WRAPPER_PYTHON=2.7
cp bin/n2d2_tensorRT_inference.so .
python generate_model.py
```

Copy the files `n2d2_tensorRT_inference.so` and `n2d2_tensorRT_model.dat` in the web server location.

Start the Python web server:

```
./server.py
```

Open the application preview in a navigator:

```
http://127.0.0.1:8888/
```

Fig. 2: Application preview in the navigator.

15.3.3 DNeuro generation

Generate the DNeuro project:

```
n2d2 MobileNet_DEMO.ini -export DNeuro_V2 -fuse -w weights_normalized -db-export 10 -
↪export-parameters MobileNet_DEMO_DNeuro.ini -calib -1 -calib-reload
cd export_DNeuro_V2_int8
```

If you do not have a CUDA-capable NVidia GPU installed, you can use instead of .

If the calibration was already done once, it is possible to reload the calibration data with the `-calib-reload` option.

Description of the arguments:

Argument	Description
`MobileNet_DEMO.ini`	INI model
`-export DNeuro_V2`	Select the DNeuro export type
`-fuse`	Fuse BatchNorm with Conv automatically
`-w weights_normalized`	Use normalized weights for the export (the `weights_normalized` folder is created after the test). This argument is absolutely necessary to avoid weights saturation when converting to 8 bit integers
`-db-export 10`	Specifies the number of stimuli to export for the testbench
`-export-parameters MobileNet_DEMO_DNeuro.ini`	DNeuro parameter file for the export (see section [sec:DNeuroParams])
`-calib -1`	Use automatic calibration for the export. Use the full test dataset for the calibration (-1)
`-calib-reload`	Reload previous calibration data, if it already exists

Example of the output:

```
...

Generating DNeuro_V2 export to "export_DNeuro_V2_int8":
-> Generating network
Using automatic configuration for the network.
-> Generating emulator network
-> Generating cell conv1
-> Generating cell conv1_3x3_dw
-> Generating cell conv1_1x1
-> Generating cell conv2_3x3_dw
-> Generating cell conv2_1x1
-> Generating cell conv3_3x3_dw
-> Generating cell conv3_1x1
-> Generating cell conv4_3x3_dw
-> Generating cell conv4_1x1
-> Generating cell conv5_3x3_dw
-> Generating cell conv5_1x1
-> Generating cell conv6_3x3_dw
-> Generating cell conv6_1x1
-> Generating cell conv7_1_3x3_dw
-> Generating cell conv7_1_1x1
-> Generating cell conv7_2_3x3_dw
-> Generating cell conv7_2_1x1
-> Generating cell conv7_3_3x3_dw
```

(continues on next page)

(continued from previous page)

```
-> Generating cell conv9_1x1
-> Generating cell resize

Estimated usage per layer:
--conv1--

...

--conv9_1x1--
RTL type: CONV_Tn_Oy_CHy_K1_Sy_Pn
Number of MACs: 5529600
Number of affected DSPs: 6
Number of MACs/DSPs: 921600
Memory for weights (bytes): 1152
Memory used for calculations (bytes): 1536

--resize--
RTL type: RESIZE_NEAREST_NEIGHBOUR
Memory for weights (bytes): 0
Memory used for calculations (bytes): 0


Total number of MACs: 855187968
Total number of used DSPs: 794
Total memory required for weights: 74.72 KiB
Total memory required for calculations: 937.50 KiB
Total memory required: 1012.22 KiB

Available DSPs on FPGA: 830
Available memory on FPGA: 1953.12 KiB

Estimated FPS at 200 Mhz: 162.76 FPS
Slowest cell: conv7_2_1x1


Done!
```

Run the network on the emulator:

```
cd EMULATOR
make
```

15.4 Face Detection DEMO

This demo uses the open-source *AppFaceDetection* application that comes with N2D2.

Fig. 3: Face detection DEMO preview on IMDB-WIKI images.

The generate the DNeuro, one must change the *IMDBWIKI.ini* file as follows:

- Uncomment the `[database]` section, in order to be able to perform a calibration on the dataset (the IMDB-WIKI dataset must be present);
- Remove the `[post.Transformation-*]` sections, which are currently not exportable;
- Remove the `[fc3.gender]` and `[fc3.gender.Target]`, as only single-branch networks are currently supported;
- Add a resize block after `[fc3.face]` and use it as target instead of `[fc3.face.Target]` in order to obtain an output of the same size as the input.

The end of the *IMDBWIKI.ini* file should look like:

```
[fc3.face]
...

[resize]
Input=fc3.face
Type=Resize
NbOutputs=[fc3.face]NbOutputs
Mode=NearestNeighbor
OutputWidth=[sp]SizeX
OutputHeight=[sp]SizeY
ConfigSection=resize.config
[resize.config]
AlignCorners=1

[resize.Target]
LabelsMapping=IMDBWIKI_target_face.dat
NoDisplayLabel=0
```

(continues on next page)

(continued from previous page)

```
[common.config]
...
```

The DNeuro project can now be generated (it is possible to re-use the export parameter file from the Aerial Imagery Segmentation DEMO):

```
n2d2 IMDBWIKI.ini -export DNeuro_V2 -fuse -w weights_normalized -db-export 10 -export-
↪parameters MobileNet_DEMO_DNeuro.ini -calib -1 -calib-reload
```

Example of the output for the *IMDBWIKI.ini* network with 640x480 input resolution and a 1,000 DSP maximum constraint:

```
Estimated usage per layer:
--conv1.1--

...

--fc3.face--
RTL type: CONV_Tn_Oy_CHy_K1_Sy_Pn
Number of MACs: 614400
Number of affected DSPs: 1
Number of MACs/DSPs: 614400
Memory for weights (bytes): 128
Memory used for calculations (bytes): 256

--to_rgb--
RTL type: VALUE_TO_RGB
Memory for weights (bytes): 0
Memory used for calculations (bytes): 0

--resize--
RTL type: RESIZE_NEAREST_NEIGHBOUR
Memory for weights (bytes): 0
Memory used for calculations (bytes): 0


Total number of MACs: 10102864576
Total number of used DSPs: 937
Total memory required for weights: 404.30 KiB
Total memory required for calculations: 1024.66 KiB
Total memory required: 1428.95 KiB

Available DSPs on FPGA: 1000
Available memory on FPGA: 1953.12 KiB

Estimated FPS at 200 Mhz: 18.17 FPS
Slowest cell: conv2.2
```

CHAPTER

SIXTEEN

EXPORT: ONNX

Export type: `ONNX`
ONNX export.

```
n2d2 MobileNet_ONNX.ini -seed 1 -w /dev/null -export ONNX
```

16.1 Principle

The ONNX export allows you to generate an ONNX model from a N2D2 model. The generated ONNX model is optimized for inference and can be quantized beforehand with either post-training quantization or Quantization Aware Training (QAT).

16.1.1 Graph optimizations

- Weights are equalized between layers when possible;
- `BatchNorm` is automatically fused with the preceding `Conv` or `Fc` when possible;
- `Padding` layers are fused with `Conv` when possible;
- `Dropout` layers are removed.

16.1.2 Export parameters

Extra parameters can be passed during export using the `-export-parameters params.ini` command line argument. The parameters must be saved in an INI-like file.

List of available parameters:

Argument [default value]	Description
`ImplicitCasting` [0]	If true (1), casting in the graph is implicit and `Cast` ONNX operators are not inserted
`FakeQuantization` [0]	If true (1), the graph is fake quantized, meaning floating-point ONNX operators are used for the computation

The parameters `ImplicitCasting` and `FakeQuantization` are useful only for quantized networks. In this case, a full integer ONNX graph is generated when possible, notably using the ONNX *ConvInteger* and *MatMulInteger* when `-nbbits` is 8 bits. An example of generated graph is shown below, with a `Single-shift` activation rescaling mode (`-act-rescaling-mode`, see *Post-training quantization*):

By default, strict adherence to the ONNX standard is enforced, by adding explicit `Cast` operators when required. The automatic insertion of `Cast` operators can be disabled by setting the `ImplicitCasting` export parameter to true. This results in the simplified graph below:

The `FakeQuantization` parameter allows to export a quantized network using fake quantization, meaning the parameters of the network are quantized (integer) but their representation remains in floating-point and the computation is done with floating-point operators. However, the output values of the network should be almost identical to when the computation is done in integer. The differences are due to numerical errors as all integers cannot be represented exactly with floating-point.

Note: The `FakeQuantization`, when set, implies `ImplicitCasting`, as no casting operator is required in a fully floating-point graph.

16.2 Example

```
n2d2 MobileNet_ONNX.ini -seed 1 -w weights_validation -export ONNX -nbbits 8 -calib -1 -
↪db-export 100 -test
```

This command generates a 8-bits integer quantized ONNX model in the sub-directory `export_ONNX_int8`.

CHAPTER

SEVENTEEN

EXPORT: OTHER / LEGACY

```
n2d2 "mnist24_16c4s2_24c5s2_150_10.ini" -export CPP_OpenCL
```

Export types:

- `C` C export using OpenMP;
- `C_HLS` C export tailored for HLS with Vivado HLS;
- `CPP_OpenCL` C++ export using OpenCL;
- `CPP_Cuda` C++ export using Cuda;
- `CPP_cuDNN` C++ export using cuDNN;
- `SC_Spike` SystemC spike export.

Other program options related to the exports:

Option [default value]	Description
`-nbbits` [8]	Number of bits for the weights and signals. Must be 8, 16, 32 or 64 for integer export, or -32, -64 for floating point export. The number of bits can be arbitrary for the `C_HLS` export (for example, 6 bits). It must be -32 for the `CPP_TensorRT` export, the precision is directly set at runtime
`-calib` [0]	Number of stimuli used for the calibration. 0 = no calibration (default), -1 = use the full test dataset for calibration
`-calib-passes` [2]	Number of KL passes for determining the layer output values distribution truncation threshold (0 = use the max. value, no truncation)
`-no-unsigned`	If present, disable the use of unsigned data type in integer exports
`-db-export` [-1]	Max. number of stimuli to export (0 = no dataset export, -1 = unlimited)

17.1 C export

Test the exported network:

```
cd export_C_int8
make
./bin/n2d2_test
```

The result should look like:

```
...
1652.00/1762    (avg = 93.757094%)
1653.00/1763    (avg = 93.760635%)
1654.00/1764    (avg = 93.764172%)
Tested 1764 stimuli
Success rate = 93.764172%
Process time per stimulus = 187.548186 us (12 threads)

Confusion matrix:
-------------------------------------------------
| T \ E |       0 |       1 |       2 |       3 |
-------------------------------------------------
|     0 |     329 |       1 |       5 |       2 |
|       |  97.63% |   0.30% |   1.48% |   0.59% |
|     1 |       0 |     692 |       2 |       6 |
|       |   0.00% |  98.86% |   0.29% |   0.86% |
|     2 |      11 |      27 |     609 |      55 |
|       |   1.57% |   3.85% |  86.75% |   7.83% |
|     3 |       0 |       0 |       1 |      24 |
|       |   0.00% |   0.00% |   4.00% |  96.00% |
-------------------------------------------------
T: Target    E: Estimated
```

17.2 CPP_OpenCL export

The OpenCL export can run the generated program in GPU or CPU architectures. Compilation features:

Preprocessor command [default value]	Description
`PROFILING` [0]	Compile the binary with a synchronization between each layers and return the mean execution time of each layer. This preprocessor option can decrease performances.
`GENERATE_KBIN` [0]	Generate the binary output of the OpenCL kernel .cl file use. The binary is store in the /bin folder.
`LOAD_KBIN` [0]	Indicate to the program to load an OpenCL kernel as a binary from the /bin folder instead of a .cl file.
`CUDA` [0]	Use the CUDA OpenCL SDK locate at $/usr/local/cuda$
`MALI` [0]	Use the MALI OpenCL SDK locate at $/usr/Mali_OpenCL_SDK_vXXX$
`INTEL` [0]	Use the INTEL OpenCL SDK locate at $/opt/intel/opencl$
`AMD` [1]	Use the AMD OpenCL SDK locate at $/opt/AMDAPPSDK-XXX$

Program options related to the OpenCL export:

Option [default value]	Description
`-cpu`	If present, force to use a CPU architecture to run the program
`-gpu`	If present, force to use a GPU architecture to run the program
`-batch` [1]	Size of the batch to use
`-stimulus` [NULL]	Path to a specific input stimulus to test. For example: -stimulus $/stimulus/env0000.pgm$ command will test the file env0000.pgm of the stimulus folder.

Test the exported network:

```
cd export_CPP_OpenCL_float32
make
./bin/n2d2_opencl_test -gpu
```

17.3 CPP_cuDNN export

The cuDNN export can run the generated program in NVIDIA GPU architecture. It use CUDA and cuDNN library. Compilation features:

Preprocessor command [default value]	Description
`PROFILING` [0]	Compile the binary with a synchronization between each layers and return the mean execution time of each layer. This preprocessor option can decrease performances.
`ARCH32` [0]	Compile the binary with the 32-bits architecture compatibility.

Program options related to the cuDNN export:

Option [default value]	Description
`-batch` [1]	Size of the batch to use
`-dev` [0]	CUDA Device ID selection
`-stimulus` [NULL]	Path to a specific input stimulus to test. For example: -stimulus $/stimulus/env0000.pgm$ command will test the file env0000.pgm of the stimulus folder.

Test the exported network:

```
cd export_CPP_cuDNN_float32
make
./bin/n2d2_cudnn_test
```

17.4 C_HLS export

Test the exported network:

```
cd export_C_HLS_int8
make
./bin/n2d2_test
```

Run the High-Level Synthesis (HLS) with Xilinx Vivado HLS:

```
vivado_hls -f run_hls.tcl
```

17.5 Layer compatibility table

Layer compatibility table in function of the export type:

Layer compatibility table	Export type			
	C	C_HLS	CPP_OpenCL	CPP_TensorRT
Conv	✓	✓	✓	✓
Pool	✓	✓	✓	✓
Fc	✓	✓	✓	✓
Softmax	✓	✗	✓	✓
FMP	✓	✗	✓	✗
Deconv	✗	✗	✗	✓
ElemWise	✗	✗	✗	✓
Resize	✓	✗	✗	✓
Padding	✗	✗	✗	✓
LRN	✗	✗	✗	✓
Anchor	✗	✗	✗	✓
ObjectDet	✗	✗	✗	✓
ROIPooling	✗	✗	✗	✓
RP	✗	✗	✗	✓

BatchNorm is not mentionned because batch normalization parameters are automatically fused with convolutions parameters with the command "-fuse".

CHAPTER

EIGHTEEN

INTRODUCTION

The INI file interface is the primary way of using N2D2. It is a simple, lightweight and user-friendly format for specifying a complete DNN-based application, including dataset instanciation, data pre-processing, neural network layers instanciation and post-processing, with all its hyperparameters.

18.1 Syntax

INI files are simple text files with a basic structure composed of sections, properties and values.

18.1.1 Properties

The basic element contained in an INI file is the property. Every property has a name and a value, delimited by an equals sign (=). The name appears to the left of the equals sign.

```
name=value
```

18.1.2 Sections

Properties may be grouped into arbitrarily named sections. The section name appears on a line by itself, in square brackets ([and]). All properties after the section declaration are associated with that section. There is no explicit "end of section" delimiter; sections end at the next section declaration, or the end of the file. Sections may not be nested.

```
[section]
a=a
b=b
```

18.1.3 Case sensitivity

Section and property names are case sensitive.

18.1.4 Comments

Semicolons (;) or number sign (#) at the beginning or in the middle of the line indicate a comment. Comments are ignored.

```
; comment text
a=a # comment text
a="a ; not a comment" ; comment text
```

18.1.5 Quoted values

Values can be quoted, using double quotes. This allows for explicit declaration of whitespace, and/or for quoting of special characters (equals, semicolon, etc.).

18.1.6 Whitespace

Leading and trailing whitespace on a line are ignored.

18.1.7 Escape characters

A backslash (\) followed immediately by EOL (end-of-line) causes the line break to be ignored.

18.2 Template inclusion syntax

Is is possible to recursively include templated INI files. For example, the main INI file can include a templated file like the following:

```
[inception@inception_model.ini.tpl]
INPUT=layer_x
SIZE=32
ARRAY=2 ; Must be the number of elements in the array
ARRAY[0].P1=Conv
ARRAY[0].P2=32
ARRAY[1].P1=Pool
ARRAY[1].P2=64
```

If the `inception_model.ini.tpl` template file content is:

```
[{{SECTION_NAME}}_layer1]
Input={{INPUT}}
Type=Conv
NbOutputs={{SIZE}}

[{{SECTION_NAME}}_layer2]
Input={{SECTION_NAME}}_layer1
Type=Fc
NbOutputs={{SIZE}}

{% block ARRAY %}
```

(continues on next page)

(continued from previous page)

```
[{{SECTION_NAME}}_array{{#}}]
Prop1=Config{{.P1}}
Prop2={{.P2}}
{% endblock %}
```

The resulting equivalent content for the main INI file will be:

```
[inception_layer1]
Input=layer_x
Type=Conv
NbOutputs=32

[inception_layer2]
Input=inception_layer1
Type=Fc
NbOutputs=32

[inception_array0]
Prop1=ConfigConv
Prop2=32

[inception_array1]
Prop1=ConfigPool
Prop2=64
```

The `SECTION_NAME` template parameter is automatically generated from the name of the including section (before `@`).

18.2.1 Variable substitution

`{{VAR}}` is replaced by the value of the `VAR` template parameter.

18.2.2 Control statements

Control statements are between `{%` and `%}` delimiters.

block

`{% block ARRAY %}` ... `{% endblock %}`

The `#` template parameter is automatically generated from the `{% block ... %}` template control statement and corresponds to the current item position, starting from 0.

for

`{% for VAR in range([START, ]END]) %}` ... `{% endfor %}`

If `START` is not specified, the loop begins at 0 (first value of `VAR`). The last value of `VAR` is `END`-1.

if

`{% if VAR OP [VALUE] %}` ... `[{% else %}]` ... `{% endif %}`

`OP` may be ==, `!=`, `exists` or `not_exists`.

include

`{% include FILENAME %}`

18.3 Global parameters

Option [default value]	Description
`DefaultModel` [`Transcode`]	Default layers model. Can be `Frame`, `Frame_CUDA`, `Transcode` or `Spike`
`DefaultDataType` [`Float32`]	Default layers data type. Can be `Float16`, `Float32` or `Float64`
`InsertBatchNormAfterConv` [0]	If true (1), batch normalization is automatically inserted after each convolution when not already present

CHAPTER

NINETEEN

DATABASES

19.1 Introduction

A `Database` handles the raw data, annotations and how the datasets (learn, validation or test) should be build. N2D2 integrates pre-defined modules for several well-known database used in the deep learning community, such as MNIST, GTSRB, CIFAR10 and so on. That way, no extra step is necessary to be able to directly build a network and learn it on these database.

All the database modules inherit from a base `Database`, which contains some generic configuration options:

Option [default value]	Description
`DefaultLabel` []	Default label for composite image (for areas outside the ROIs). If empty, no default label is created and default label ID is -1
`ROIsMargin` [0]	Margin around the ROIs, in pixels, with no label (label ID = -1)
`RandomPartitioning` [1]	If true (1), the partitioning in the learn, validation and test sets is random, otherwise partitioning is in the order
`DataFileLabel` [1]	If true (1), load pixel-wise image labels, if they exist
`CompositeLabel` [`Auto`]	See the following `CompositeLabel` section
`TargetDataPath` []	Data path to target data, to be used in conjunction with the `DataAsTarget` option in `Target` modules
`MultiChannelMatch` []	See the following *multi-channel handling* section
`MultiChannelReplace`	See the following *multi-channel handling* section

19.1.1 `CompositeLabel` parameter

A label is said to be composite when it is not a single *labelID* for the stimulus (the stimulus label is a matrix of size > 1). For the same stimulus, different type of labels can be specified, i.e. the *labelID*, pixel-wise data and/or ROIs. The way these different label types are handled is configured with the `CompositeLabel` parameter:

- `None`: only the *labelID* is used, pixel-wise data are ignored and ROIs are loaded but ignored as well by `loadStimulusLabelsData()`.
- `Auto`: the label is only composite when pixel-wise data are present or the stimulus *labelID* is -1 (in which case the `DefaultLabel` is used for the whole label matrix). If the label is composite ROIs, if present, are applied. Otherwise, a single ROI is allowed and is automatically extracted when fetching the stimulus.

- `Default`: the label is always composite. The *labelID* is ignored. If there is no pixel-wise data, the `DefaultLabel` is used. ROIs, if present, are applied.
- `Disjoint`: the label is always composite. If there is no pixel-wise data:
  - the *labelID* is used if there is no ROI;
  - the `DefaultLabel` is used if there is any ROI.

  ROIs, if present, are applied.
- `Combine`: the label is always composite. If there is no pixel-wise data, the *labelID* is used. ROIs, if present, are applied.

19.1.2 Multi-channel handling

Multi-channel images are automatically handled and the default image format in N2D2 is **BGR**.

Any `Database` can also handle multi-channel data, where each channel is stored in a different file. In order to be able to interpret a series of files as an additional data channel to a first series of files, the file names must follow a simple yet arbitrary naming convention. A first parameter, `MultiChannelMatch`, is used to match the files constituting a single channel. Then, a second parameter, `MultiChannelReplace` is used to indicate how the file names of the other channels are obtained. See the example below, with the `DIR_Database`:

```
[database]
Type=DIR_Database
...
; Multi-channel handling:
; MultiChannelMatch is a regular expression for matching a single channel (for example␣
↪the first one).
; Here we match anything followed by "_0", followed by "." and anything except
; ".", so we match "_0" before the file extension.
MultiChannelMatch=(.*)_0(\.[^.]+)
; Replace what we matched to obtain the file name of the different channels.
; For the first channel, replace "_0" by "_0", so the name doesn't change.
; For the second channel, replace "_0" by "_1" in the file name.
; To disable the second channel, replace $1_1$2 by ""
MultiChannelReplace=$1_0$2 $1_1$2
```

Note that when `MultiChannelMatch` is not empty, only files matching this parameter regexp pattern (and the associated channels obtained with `MultiChannelReplace`, when they exist) will be loaded. Other files in the dataset not matching the `MultiChannelMatch` filter will be ignored.

Stimuli are loaded even if some channels are missing (in which case, "Notice" messages are issued for the missing channel(s) during database loading). Missing channel values are set to 0.

Annotations are common to all channels. If annotations exist for a specific channel, they are fused with the annotations of the other channels (for geometric annotations). Pixel-wise annotations, obtained when `DataFileLabel` is 1 (true), through the `Database::readLabel()` virtual method, are only read for the match (`MultiChannelMatch`) channel.

19.2 MNIST

MNIST [LBBH98] is already fractionned into a learning set and a testing set, with:

- 60,000 digits in the learning set;
- 10,000 digits in the testing set.

Example:

```
[database]
Type=MNIST_IDX_Database
Validation=0.2  ; Fraction of learning stimuli used for the validation [default: 0.0]
```

Option [default value]	Description
`Validation` [0.0]	Fraction of the learning set used for validation
`DataPath`	Path to the database
[`$N2D2_DATA`/mnist]	

19.3 GTSRB

GTSRB [SSSI12] is already fractionned into a learning set and a testing set, with:

- 39,209 digits in the learning set;
- 12,630 digits in the testing set.

Example:

```
[database]
Type=GTSRB_DIR_Database
Validation=0.2  ; Fraction of learning stimuli used for the validation [default: 0.0]
```

Option [default value]	Description
`Validation` [0.0]	Fraction of the learning set used for validation
`DataPath`	Path to the database
[`$N2D2_DATA`/GTSRB]	

19.4 Directory

Hand made database stored in files directories are directly supported with the `DIR_Database` module. For example, suppose your database is organized as following (in the path specified in the `N2D2_DATA` environment variable):

- `GST/airplanes`: 800 images
- `GST/car_side`: 123 images
- `GST/Faces`: 435 images
- `GST/Motorbikes`: 798 images

You can then instanciate this database as input of your neural network using the following parameters:

```
[database]
Type=DIR_Database
DataPath=${N2D2_DATA}/GST
Learn=0.4 ; 40% of images of the smallest category = 49 (0.4x123) images for each␣
↪category will be used for learning
Validation=0.2 ; 20% of images of the smallest category = 25 (0.2x123) images for each␣
↪category will be used for validation
; the remaining images will be used for testing
```

Each subdirectory will be treated as a different label, so there will be 4 different labels, named after the directory name.

The stimuli are equi-partitioned for the learning set and the validation set, meaning that the same number of stimuli for each category is used. If the learn fraction is 0.4 and the validation fraction is 0.2, as in the example above, the partitioning will be the following:

Label ID	Label name	Learn set	Validation set	Test set
[0.5ex] 0	`airplanes`	49	25	726
1	`car_side`	49	25	49
2	`Faces`	49	25	361
3	`Motorbikes`	49	25	724
	Total:	196	100	1860

Mandatory option

Option [default value]	Description
DataPath	Path to the root stimuli directory
IgnoreMasks	Space-separated list of mask strings to ignore. If any is present in a file path, the file gets ignored. The usual * and + wildcards are allowed.
Learn	If PerLabelPartitioning is true, fraction of images used for the learning; else, number of images used for the learning, regardless of their labels
LoadInMemory [0]	Load the whole database into memory
Depth [1]	Number of sub-directory levels to include. Examples:
	Depth = 0: load stimuli only from the current directory (DataPath)
	Depth = 1: load stimuli from DataPath and stimuli contained in the sub-directories of DataPath
	Depth < 0: load stimuli recursively from DataPath and all its sub-directories
LabelName []	Base stimuli label name
LabelDepth [1]	Number of sub-directory name levels used to form the stimuli labels. Examples:
	LabelDepth = -1: no label for all stimuli (label ID = -1)
	LabelDepth = 0: uses LabelName for all stimuli
	LabelDepth = 1: uses LabelName for stimuli in the current directory (DataPath) and LabelName/*sub-directory name* for stimuli in the sub-directories
PerLabelPartitioning [1]	If true (1), the Learn, Validation and Test parameters represent the fraction of the total stimuli to be partitioned in each set, instead of a number of stimuli
EquivLabelPartitioning [1]	If true (1), the stimuli are equi-partitioned in the learn and validation sets, meaning that the same number of stimuli **for each label** is used (only when PerLabelPartitioning is 1). The remaining stimuli are partitioned in the test set
Validation [0.0]	If PerLabelPartitioning is true, fraction of images used for the validation; else, number of images used for the validation, regardless of their labels
Test [1.0-Learn-Validation]	If PerLabelPartitioning is true, fraction of images used for the test; else, number of images used for the test, regardless of their labels
ValidExtensions []	List of space-separated valid stimulus file extensions (if left empty, any file extension is considered a valid stimulus)
LoadMore []	Name of an other section with the same options to load a different DataPath
ROIFile []	File containing the stimuli ROIs. If a ROI file is specified, LabelDepth should be set to -1
DefaultLabel []	Label name for pixels outside any ROI (default is no label, pixels are ignored)
ROIsMargin [0]	Number of pixels around ROIs that are ignored (and not considered as DefaultLabel pixels)

Note: If EquivLabelPartitioning is 1 (default setting), the number of stimuli per label that will be partitioned in the learn and validation sets will correspond to the number of stimuli from the label with the fewest stimuli.

To load and partition more than one DataPath, one can use the LoadMore option:

```
[database]
Type=DIR_Database
DataPath=${N2D2_DATA}/GST
Learn=0.6
Validation=0.4
```

(continues on next page)

(continued from previous page)

```
LoadMore=database.test

; Load stimuli from the "GST_Test" path in the test dataset
[database.test]
DataPath=${N2D2_DATA}/GST_Test
Learn=0.0
Test=1.0
; The LoadMore option is recursive:
; LoadMore=database.more

; [database.more]
; Load even more data here
```

19.4.1 *Speech Commands Dataset*

Use with Speech Commands Data Set, released by the Google [Warden18].

```
[database]
Type=DIR_Database
DataPath=${N2D2_DATA}/speech_commands_v0.02
ValidExtensions=wav
IgnoreMasks=*/_background_noise_
Learn=0.6
Validation=0.2
```

19.5 CSV data files

`CSV_Database` is a generic driver for handling CSV data files. It can be used to load one or several CSV files where each line is a different stimulus and one column contains the label.

The parameters are the following:

Option [default value]	Description
`DataPath`	Path to the database
`Learn` [0.6]	Fraction of data used for the learning
`Validation` [0.2]	Fraction of data used for the validation
`PerLabelPartitioning` [1]	If true (1), the `Learn`, `Validation` and `Test` parameters represent the fraction of the total stimuli to be partitioned in each set, instead of a number of stimuli
`EquivLabelPartitioning` [1]	If true (1), the stimuli are equi-partitioned in the learn and validation sets, meaning that the same number of stimuli **for each label** is used (only when `PerLabelPartitioning` is 1). The remaining stimuli are partitioned in the test set
`LabelColumn` [-1]	Index of the column containing the label (if < 0, from the end of the row)
`NbHeaderLines` [0]	Number of header lines to skip
`Test` [1.0-`Learn`-`Validation`]	If `PerLabelPartitioning` is true, fraction of images used for the test; else, number of images used for the test, regardless of their labels
`LoadMore` []	Name of an other section with the same options to load a different `DataPath`

Note: If `EquivLabelPartitioning` is 1 (default setting), the number of stimuli per label that will be partitioned in the learn and validation sets will correspond to the number of stimuli from the label with the fewest stimuli.

19.5.1 Usage example

In this example, we load the *Electrical Grid Stability Simulated Data Data Set* (https://archive.ics.uci.edu/ml/datasets/Electrical+Grid+Stability+Simulated+Data+).

The CSV data file (`Data_for_UCI_named.csv`) is the following:

```
"tau1","tau2","tau3","tau4","p1","p2","p3","p4","g1","g2","g3","g4","stab","stabf"
2.95906002455997,3.07988520422811,8.38102539191882,9.78075443222607,3.76308477206316,-0.
↪782603630987543,-1.25739482958732,-1.7230863114883,0.650456460887227,0.859578105752345,
↪0.887444920638513,0.958033987602737,0.0553474891727752,"unstable"
9.3040972346785,4.90252411201167,3.04754072762177,1.36935735529605,5.06781210427845,-1.
↪94005842705193,-1.87274168559721,-1.25501199162931,0.413440056837935,0.862414076352903,
↪0.562139050527675,0.781759910653126,-0.00595746432603695,"stable"
8.97170690932022,8.84842842134833,3.04647874898866,1.21451813833956,3.40515818001095,-1.
↪20745559234302,-1.27721014673295,-0.92049244093498,0.163041039311334,0.766688656526962,
↪0.839444015400588,0.109853244952427,0.00347087904838871,"unstable"
0.716414776295121,7.66959964406565,4.48664083058949,2.34056298396795,3.96379106326633,-1.
↪02747330413905,-1.9389441526466,-0.997373606480681,0.446208906537321,0.976744082924302,
↪0.929380522872661,0.36271777426931,0.028870543444887,"unstable"
3.13411155161342,7.60877161603408,4.94375930178099,9.85757326996638,3.52581081652096,-1.
↪12553095451115,-1.84597485447561,-0.554305007534195,0.797109525792467,0.
↪455449947148291,0.656946658473716,0.820923486481631,0.0498603734837059,"unstable"
...
```

There is one header line and the last column is the label, which is the default.

This file is loaded and the data is splitted between the learning set and the validation set with a 0.7/0.3 ratio in the INI file with the following section:

```
[database]
Type=CSV_Database
Learn=0.7
Validation=0.3
DataPath=Data_for_UCI_named.csv
NbHeaderLines=1
```

19.6 Other built-in databases

19.6.1 Actitracker_Database

Actitracker database, released by the WISDM Lab [LWX+11].

Option [default value]	Description
`Learn` [0.6]	Fraction of data used for the learning
`Validation` [0.2]	Fraction of data used for the validation
`UseUnlabeledForTest` [0]	If true, use the unlabeled dataset for the test
`DataPath`	Path to the database
[`$N2D2_DATA`/WISDM_at_v2.0]	

19.6.2 CIFAR10_Database

CIFAR10 database [Kri09].

Option [default value]	Description
`Validation` [0.0]	Fraction of the learning set used for validation
`DataPath`	Path to the database
[`$N2D2_DATA`/cifar-10-batches-bin]	

19.6.3 CIFAR100_Database

CIFAR100 database [Kri09].

Option [default value]	Description
`Validation` [0.0]	Fraction of the learning set used for validation
`UseCoarse` [0]	If true, use the coarse labeling (10 labels instead of 100)
`DataPath`	Path to the database
[`$N2D2_DATA`/cifar-100-binary]	

19.6.4 CKP_Database

The Extended Cohn-Kanade (CK+) database for expression recognition [LuceyCohnKanade+10].

Option [default value]	Description
`Learn`	Fraction of images used for the learning
`Validation` [0.0]	Fraction of images used for the validation
`DataPath`	Path to the database
[`$N2D2_DATA`/cohn-kanade-images]	

19.6.5 Caltech101_DIR_Database

Caltech 101 database [FFFP04].

Option [default value]	Description
`Learn`	Fraction of images used for the learning
`Validation` [0.0]	Fraction of images used for the validation
`IncClutter` [0]	If true, includes the BACKGROUND_Google directory of the database
`DataPath`	Path to the database
[`$N2D2_DATA`/	
101_ObjectCategories]	

19.6.6 Caltech256_DIR_Database

Caltech 256 database [GHP07].

Option [default value]	Description
`Learn`	Fraction of images used for the learning
`Validation` [0.0]	Fraction of images used for the validation
`IncClutter` [0]	If true, includes the BACKGROUND_Google directory of the database
`DataPath`	Path to the database
[`$N2D2_DATA`/	
256_ObjectCategories]	

19.6.7 CaltechPedestrian_Database

Caltech Pedestrian database [DollarWSP09].

Note that the images and annotations must first be extracted from the seq video data located in the *videos* directory using the `dbExtract.m` Matlab tool provided in the "Matlab evaluation/labeling code" downloadable on the dataset website.

Assuming the following directory structure (in the path specified in the `N2D2_DATA` environment variable):

- `CaltechPedestrians/data-USA/videos/...` (from the *setxx.tar* files)
- `CaltechPedestrians/data-USA/annotations/...` (from the *setxx.tar* files)
- `CaltechPedestrians/tools/piotr_toolbox/toolbox` (from the Piotr's Matlab Toolbox archive)
- `CaltechPedestrians/*.m` including `dbExtract.m` (from the Matlab evaluation/labeling code)

Use the following command in Matlab to generate the images and annotations:

```
cd([getenv('N2D2_DATA') '/CaltechPedestrians'])
addpath(genpath('tools/piotr_toolbox/toolbox')) % add the Piotr's Matlab Toolbox in the␣
↪Matlab path
dbInfo('USA')
dbExtract()
```

Option [default value]	Description
`Validation` [0.0]	Fraction of the learning set used for validation
`SingleLabel` [1]	Use the same label for "person" and "people" bounding box
`IncAmbiguous` [0]	Include ambiguous bounding box labeled "person?" using the same label as "person"
`DataPath`	Path to the database images
[`$N2D2_DATA`/	
CaltechPedestrians/data-USA/images]	
`LabelPath`	Path to the database annotations
[`$N2D2_DATA`/	
CaltechPedestrians/data-USA/annotations]	

19.6.8 Cityscapes_Database

Cityscapes database [COR+16].

Option [default value]	Description
`IncTrainExtra` [0]	If true, includes the left 8-bit images - trainextra set (19,998 images)
`UseCoarse` [0]	If true, only use coarse annotations (which are the only annotations available for the trainextra set)
`SingleInstanceLabels` [1]	If true, convert group labels to single instance labels (for example, `cargroup` becomes `car`)
`DataPath`	Path to the database images
[`$N2D2_DATA`/	
Cityscapes/leftImg8bit] or	
[`$CITYSCAPES_DATASET`] if defined	
`LabelPath` []	Path to the database annotations (deduced from `DataPath` if left empty)

Warning: Don't forget to install the **libjsoncpp-dev** package on your device if you wish to use this database.

```
# To install JSON for C++ library on Ubuntu
sudo apt-get install libjsoncpp-dev
```

19.6.9 Daimler_Database

Daimler Monocular Pedestrian Detection Benchmark (Daimler Pedestrian).

Option [default value]	Description
`Learn` [1.0]	Fraction of images used for the learning
`Validation` [0.0]	Fraction of images used for the validation
`Test` [0.0]	Fraction of images used for the test
`Fully` [0]	When activate it use the test dataset to learn. Use only on fully-cnn mode

19.6.10 DOTA_Database

DOTA database [XBD+17].

Option [default value]	Description
`Learn`	Fraction of images used for the learning
`DataPath`	Path to the database
[`$N2D2_DATA/DOTA`]	
`LabelPath`	Path to the database labels list file
[]	

19.6.11 FDDB_Database

Face Detection Data Set and Benchmark (FDDB) [JLM10].

Option [default value]	Description
`Learn`	Fraction of images used for the learning
`Validation` [0.0]	Fraction of images used for the validation
`DataPath`	Path to the images (decompressed originalPics.tar.gz)
[`$N2D2_DATA/FDDB`]	
`LabelPath`	Path to the annotations (decompressed FDDB-folds.tgz)
[`$N2D2_DATA/FDDB`]	

19.6.12 GTSDB_DIR_Database

GTSDB database [HSS+13].

Option [default value]	Description
`Learn`	Fraction of images used for the learning
`Validation` [0.0]	Fraction of images used for the validation
`DataPath`	Path to the database
[`$N2D2_DATA/FullIJCNN2013`]	

19.6.13 ILSVRC2012_Database

ILSVRC2012 database [RDS+15].

Option [default value]	Description
`Learn`	Fraction of images used for the learning
`DataPath`	Path to the database
[`$N2D2_DATA/ILSVRC2012`]	
`LabelPath`	Path to the database labels list file
[`$N2D2_DATA/ILSVRC2012/synsets.txt`]	

19.6.14 KITTI_Database

The KITTI Database provide ROI which can be use for autonomous driving and environment perception. The database provide 8 labeled different classes. Utilization of the KITTI Database is under licensing conditions and request an email registration. To install it you have to follow this link: http://www.cvlibs.net/datasets/kitti/eval_tracking.php and download the left color images (15 GB) and the trainling labels of tracking data set (9 MB). Extract the downloaded archives in your `$N2D2_DATA/KITTI` folder.

Option [default value]	Description
`Learn` [0.8]	Fraction of images used for the learning
`Validation` [0.2]	Fraction of images used for the validation

19.6.15 KITTI_Road_Database

The KITTI Road Database provide ROI which can be used to road segmentation. The dataset provide 1 labeled class (road) on 289 training images. The 290 test images are not labeled. Utilization of the KITTI Road Database is under licensing conditions and request an email registration. To install it you have to follow this link: http://www.cvlibs.net/datasets/kitti/eval_road.php and download the "base kit" of (0.5 GB) with left color images, calibration and training labels. Extract the downloaded archive in your `$N2D2_DATA/KITTI` folder.

Option [default value]	Description
`Learn` [0.8]	Fraction of images used for the learning
`Validation` [0.2]	Fraction of images used for the validation

19.6.16 KITTI_Object_Database

The KITTI Object Database provide ROI which can be use for autonomous driving and environment perception. The database provide 8 labeled different classes on 7481 training images. The 7518 test images are not labeled. The whole database provide 80256 labeled objects. Utilization of the KITTI Object Database is under licensing conditions and request an email registration. To install it you have to follow this link: http://www.cvlibs.net/datasets/kitti/eval_object.php?obj_benchmark and download the "lef color images" (12 GB) and the training labels of object data set (5 MB). Extract the downloaded archives in your `$N2D2_DATA/KITTI_Object` folder.

Option [default value]	Description
`Learn` [0.8]	Fraction of images used for the learning
`Validation` [0.2]	Fraction of images used for the validation

19.6.17 LITISRouen_Database

LITIS Rouen audio scene dataset [RG14].

Option [default value]	Description
`Learn` [0.4]	Fraction of images used for the learning
`Validation` [0.4]	Fraction of images used for the validation
`DataPath`	Path to the database
[`$N2D2_DATA`/data_rouen]	

19.6.18 Dataset images slicing

It is possible to automatically slice images from a dataset, with a given slice size and stride, using the `.slicing` attribute. This effectively increases the number of stimuli in the set.

```
[database.slicing]
ApplyTo=NoLearn
Width=2048
Height=1024
StrideX=2048
StrideY=1024
RandomShuffle=1  ; 1 is the default value
```

The `RandomShuffle` option, enabled by default, randomly shuffle the dataset after slicing. If disabled, the slices are added in order at the end of the dataset.

CHAPTER

TWENTY

STIMULI DATA ANALYSIS

You can enable stimuli data reporting with the following section (the name of the section must start with `env.StimuliData`):

```
[env.StimuliData-raw]
ApplyTo=LearnOnly
LogSizeRange=1
LogValueRange=1
```

The stimuli data reported for the full MNIST learning set will look like:

```
env.StimuliData-raw data:
Number of stimuli: 60000
Data width range: [28, 28]
Data height range: [28, 28]
Data channels range: [1, 1]
Value range: [0, 255]
Value mean: 33.3184
Value std. dev.: 78.5675
```

20.1 Zero-mean and unity standard deviation normalization

It it possible to normalize the whole database to have zero mean and unity standard deviation on the learning set using a `RangeAffineTransformation` transformation:

```
; Stimuli normalization based on learning set global mean and std.dev.
[env.Transformation-normalize]
Type=RangeAffineTransformation
FirstOperator=Minus
FirstValue=[env.StimuliData-raw]_GlobalValue.mean
SecondOperator=Divides
SecondValue=[env.StimuliData-raw]_GlobalValue.stdDev
```

The variables `_GlobalValue.mean` and `_GlobalValue.stdDev` are automatically generated in the `[env.StimuliData-raw]` block. Thanks to this facility, unknown and arbitrary database can be analysed and normalized in one single step without requiring any external data manipulation.

After normalization, the stimuli data reported is:

```
env.StimuliData-normalized data:
Number of stimuli: 60000
Data width range: [28, 28]
Data height range: [28, 28]
Data channels range: [1, 1]
Value range: [-0.424074, 2.82154]
Value mean: 2.64796e-07
Value std. dev.: 1
```

Where we can check that the global mean is close to 0 and the standard deviation is 1 on the whole dataset. The result of the transformation on the first images of the set can be checked in the generated *frames* folder, as shown in figure [fig:frame0Mean1StdDev].

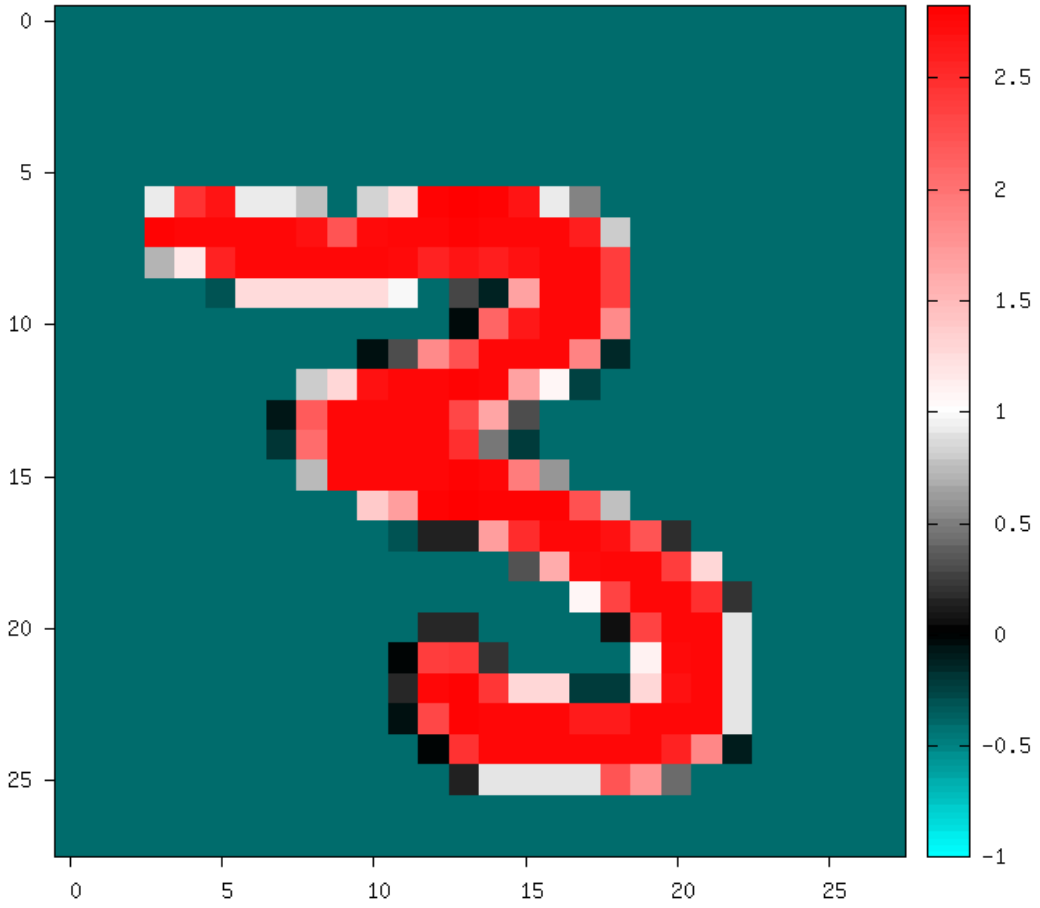


Fig. 1: Image of the set after normalization.

20.2 Substracting the mean image of the set

Using the `StimuliData` object followed with an `AffineTransformation`, it is also possible to use the mean image of the dataset to normalize the data:

```
[env.StimuliData-meanData]
ApplyTo=LearnOnly
MeanData=1 ; Provides the _MeanData parameter used in the transformation

[env.Transformation]
Type=AffineTransformation
FirstOperator=Minus
FirstValue=[env.StimuliData-meanData]_MeanData
```

The resulting global mean image can be visualized in *env.StimuliData-meanData/meanData.bin.png* an is shown in figure [fig:meanData].

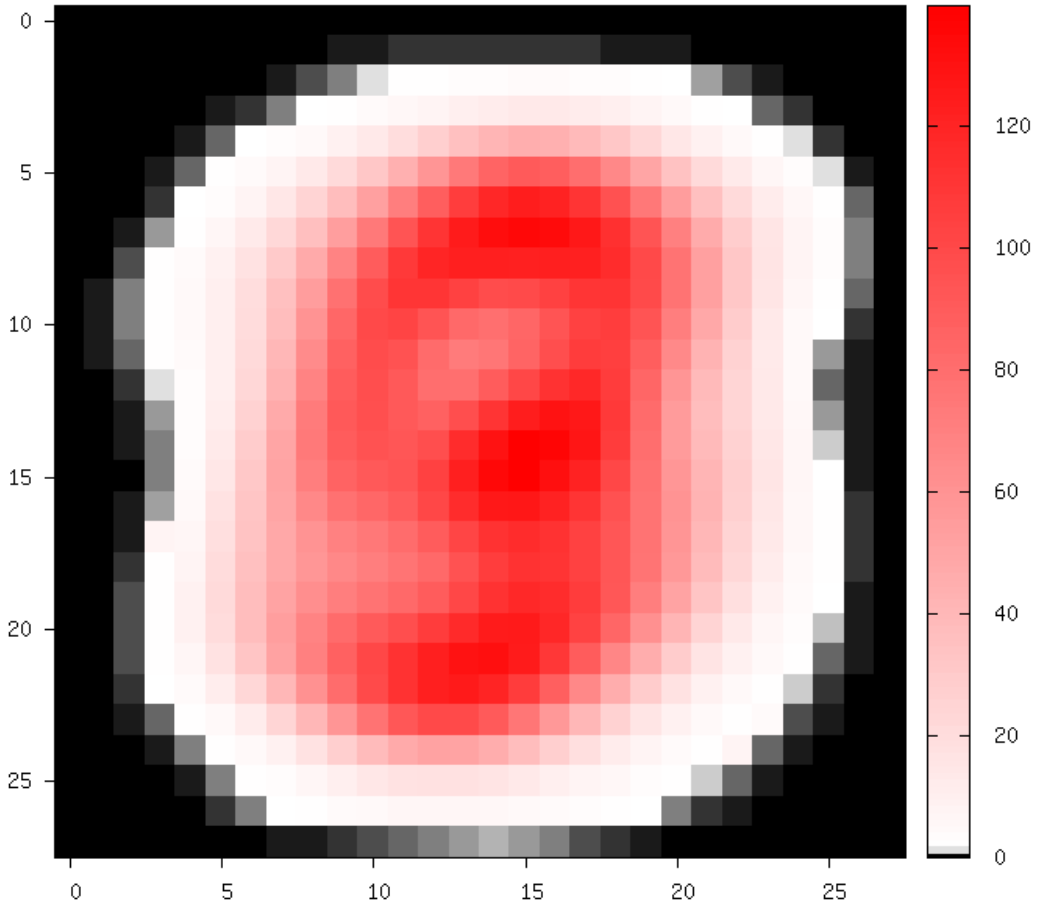


Fig. 2: Global mean image generated by `StimuliData` with the `MeanData` parameter enabled.

After this transformation, the reported stimuli data becomes:

```
env.StimuliData-processed data:
Number of stimuli: 60000
```

(continues on next page)

(continued from previous page)

```
Data width range: [28, 28]
Data height range: [28, 28]
Data channels range: [1, 1]
Value range: [-139.554, 254.979]
Value mean: -3.45583e-08
Value std. dev.: 66.1288
```

The result of the transformation on the first images of the set can be checked in the generated *frames* folder, as shown in figure [fig:frameMinusMean].

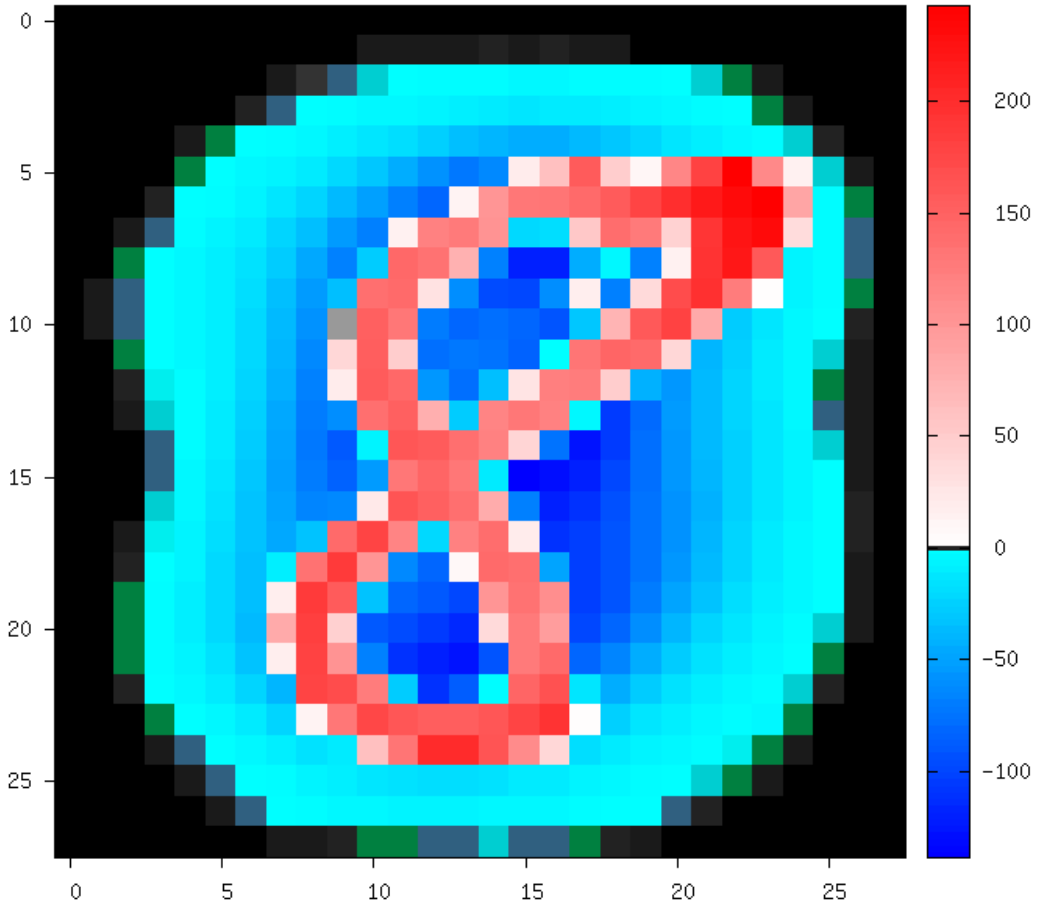


Fig. 3: Image of the set after the `AffineTransformation` substracting the global mean image (keep in mind that the original image value range is [0, 255]).

CHAPTER

TWENTYONE

STIMULI PROVIDER (ENVIRONMENT)

21.1 Introduction

The `database` section must feed a stimuli provider (or environment), which is instantiated with a section named `sp` (or `env`) in the INI file. When the two sections are present in the INI file, they are implicitly connected: the `StimuliProvider` is automatically aware of the `Database` driver that is present. The `StimuliProvider` section specifies the input dimensions of the network (width, height), as well as the batch size.

Example:

```
[sp]
SizeX=24
SizeY=24
BatchSize=12 ; [default: 1]
```

Data augmentation and conditioning `Transformation` blocks and data analysis `StimuliData` blocks can be associated to a stimuli provider as shown below:

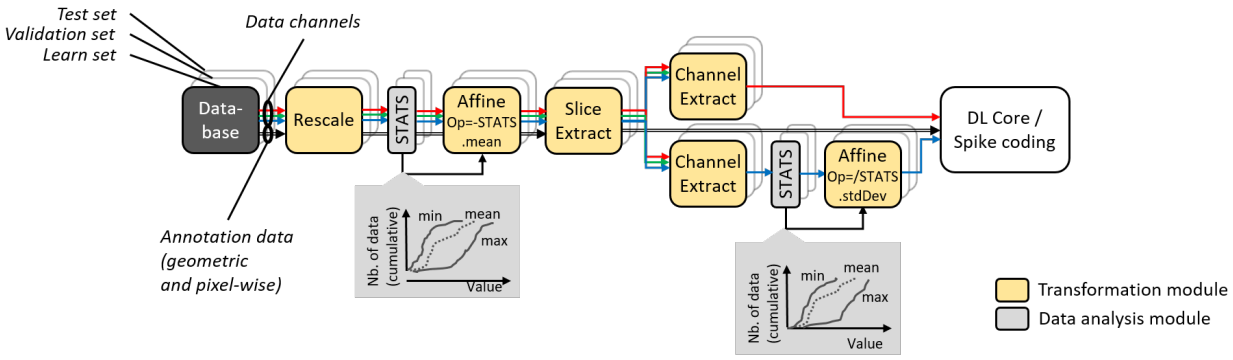


Fig. 1: Data augmentation, conditioning and analysis flow.

The table below summarizes the parameters available for the `sp` section:

Option [default value]	Description
SizeX	Environment width
SizeY	Environment height
NbChannels [1]	Number of channels (applicable only if there is no env.ChannelTransformation[...])
BatchSize [1]	Batch size
CompositeStimuli [0]	If true, use pixel-wise stimuli labels
CachePath []	Stimuli cache path (no cache if left empty)

The env section accepts more parameters dedicated to event-based (spiking) simulation:

Option (env only) [default]	Description
StimulusType [SingleBurst]	Method for converting stimuli into spike trains. Can be any of SingleBurst, Periodic, JitteredPeriodic or Poissonian
DiscardedLateStimuli [1.0]	The pixels in the pre-processed stimuli with a value above this limit never generate spiking events
PeriodMeanMin [50 TimeMs]	Mean minimum period $\overline{T_{min}}$, used for periodic temporal codings, corresponding to pixels in the pre-processed stimuli with a value of 0 (which are supposed to be the most significant pixels)
PeriodMeanMax [12 TimeS]	Mean maximum period $\overline{T_{max}}$, used for periodic temporal codings, corresponding to pixels in the pre-processed stimuli with a value of 1 (which are supposed to be the least significant pixels). This maximum period may be never reached if DiscardedLateStimuli is lower than 1.0
PeriodRelStdDev [0.1]	Relative standard deviation, used for periodic temporal codings, applied to the spiking period of a pixel
PeriodMin [11 TimeMs]	Absolute minimum period, or spiking interval, used for periodic temporal codings, for any pixel

For image segmentation, the parameter CompositeStimuli=1 must always be present, meaning that the labels of the image must have the same dimension than the image (and cannot be a single class value as in classification problem).

21.2 Data range and conversion

A configuration section can be associated to a StimuliProvider, as shown below. The DataSignedMapping=1 parameter specifies that the input value range must be interpreted as signed, even if the values are unsigned, which is usually the case for standard image formats (BMP, JPEG, PNG...). In case of 8-bit images, values from 0 to 255 are therefore mapped to the range -128 to 127 when this parameter is enabled.

```
[sp]
SizeX=[database.slicing]Width
SizeY=[database.slicing]Height
BatchSize=${BATCH_SIZE}
CompositeStimuli=1
ConfigSection=sp.config
```

(continues on next page)

(continued from previous page)

```
[sp.config]
DataSignedMapping=1
```

Note: In N2D2, the integer value input range [0, 255] (or [-128, 127] with the `DataSignedMapping=1` parameter) (for 8-bit images), is implicitly converted to floating point value range [0.0, 1.0] or [-1.0, 1.0] in the `StimuliProvider`, after the transformations, unless one of the transformation changes the representation and/or the range of the data.

Note: The `DataSignedMapping` parameter only has effect when implicit conversion is performed.

The input value range can also be changed explicitly using for example a `RangeAffineTransformation`, like below, in which case no implicit conversion is performed afterwards (and the `DataSignedMapping` parameter has no effect):

```
[sp.Transformation-rangeAffine]
Type=RangeAffineTransformation
FirstOperator=Minus
FirstValue=128.0
SecondOperator=Divides
SecondValue=128.0
```

When running a simulation in N2D2, the graph of the transformations with all their parameters as well as the expected output dimension after each transformation is automatically generated (in the file *transformations.png*). As transformations can be applied only to one of the learn, validation or test datasets, three graphs are generated, as shown in the following figure.

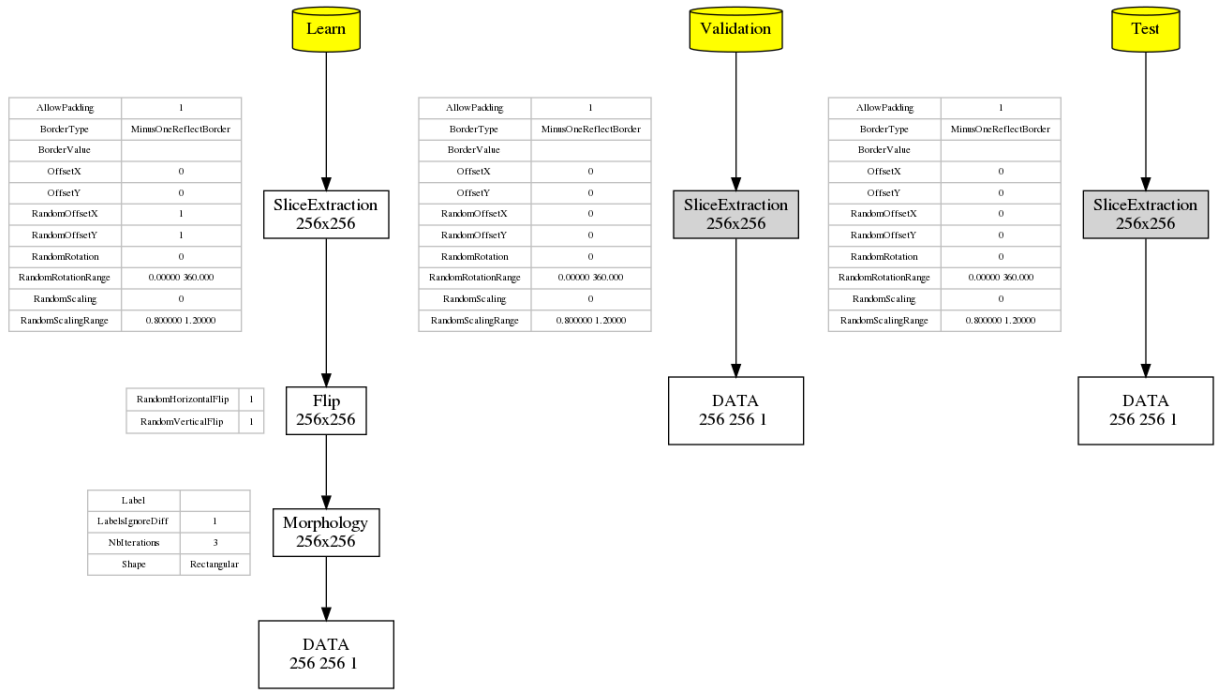


Fig. 2: Graph of the transformations for the learn, validation and test datasets, automatically generated by N2D2.

21.3 Images slicing during training and inference

In N2D2, the input dimensions of a neural network is fixed and cannot be changed dynamically during the training and inference, as images are processed in batch, like any other deep learning framework. Therefore, in order to deal with datasets containing images of variable dimensions, patches or slices of fixed dimensions must be extracted.

In N2D2, two mechanisms are provided to extract slices:

- For training, random slices can be extracted from bigger images for each batch, thus allowing to cover the full images over the training time with the maximum variability. This also act as basic data augmentation. Random slices extraction is achieved using a `SliceExtractionTransformation`, applied only to the training set with the parameter `ApplyTo=LearnOnly`.

```
[sp.OnTheFlyTransformation-1]
Type=SliceExtractionTransformation
Width=${WIDTH}
Height=${HEIGHT}
RandomOffsetX=1
RandomOffsetY=1
AllowPadding=1
ApplyTo=LearnOnly
```

- For inference, one wants to cover the full images once and only once. This cannot be achieved with a N2D2 `Transformation`, but has to be handled by the `Database` driver. In order to do so, any `Database` driver can have an additional "slicing" section in the N2D2 INI file, which will automatically extract regularly strided fixed size slices from the dataset. The example above can be used to extract slides for the validation and testing datasets, with the parameter `ApplyTo=NoLearn`.

```
[database.slicing]
Width=${WIDTH}
Height=${HEIGHT}
StrideX=[database.slicing]Width
StrideY=[database.slicing]Height
Overlapping=1
ApplyTo=NoLearn
```

When an image size is not a multiple of the slices size, the most right and most bottom slices may have a size lower than the intended fixed slice size specified with `Width` and `Height`. There are two ways to deal with these slices:

1) Add the `Overlapping=1` parameter, which allows an overlapping between the right/bottom-most slice and the preceding one. The overlapping area in the right/bottom-most slice is then marked as "ignore" for the labeling, to avoid counting twice the classification result on these pixels.

2) Add a `PadCropTransformation` to pad to the slice target size for `NoLearn` data. In this case the padded area can be either ignored or mirror padding can be used.

21.4 Blending for data augmentation

Complex data augmentation / pre-processing pipelines can be created by combining the different available transformations. It is even possible to use multiple `Database` and `StimuliProvider`, to create for example a "blending" pipeline, which is introduced here and illustrated in the figure below.

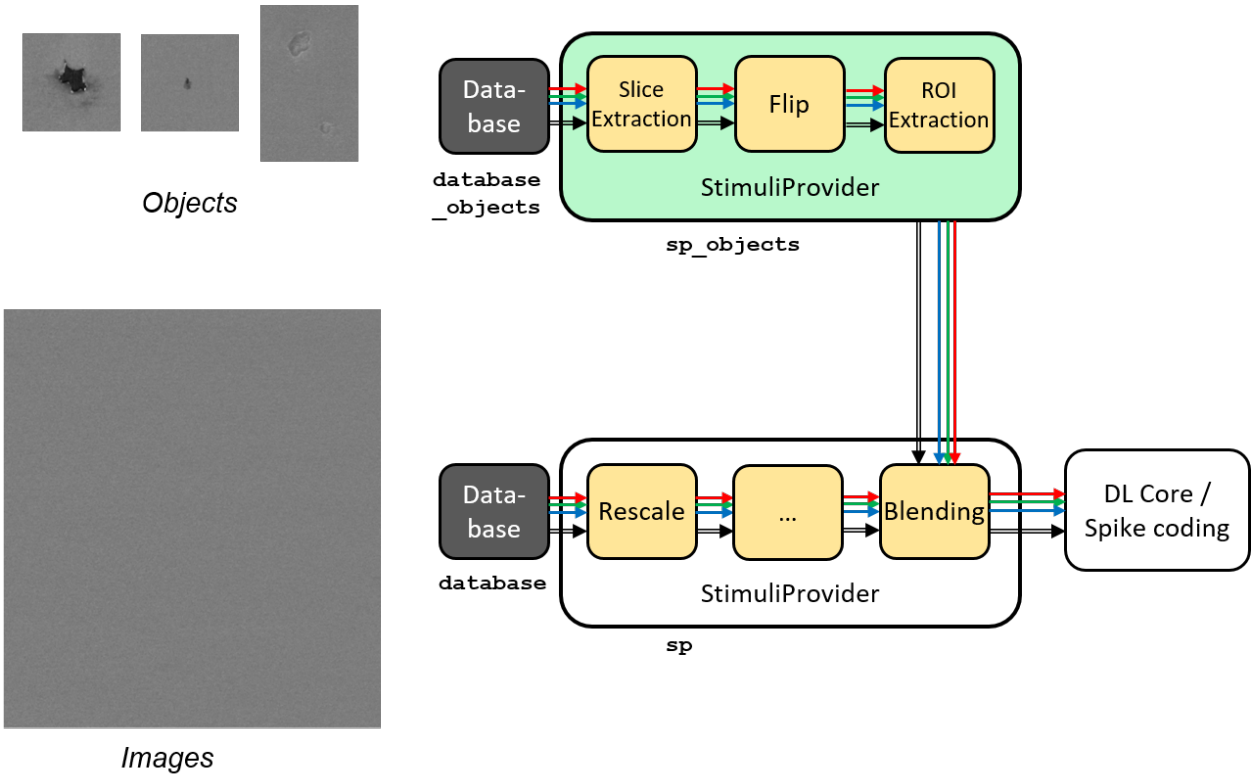


Fig. 3: Blending pipeline working principle.

An example of a blending pipeline in the INI file is given here. The first part is the `BlendingTransformation`, which is inserted in the main image processing pipeline.

```
...

; Here we add a blending transformation, which will perform objects blending
; to images with the specified labels in the dataset, selected by the
; ApplyToLabels parameter.
[sp.OnTheFlyTransformation-blend]
Type=BlendingTransformation
ApplyTo=LearnOnly
Database=database_objects ; database driver to use for the objects to blend
StimuliProvider=sp_objects  ; stimuli provider specifying the transformations
                            ; to apply on the object data before blending
; Specifies the name of the image label(s) on which a blending can be performed.
; Here, any image in a "backgrounds" sub-directory in the dataset will be used
; for the blending
; POSSIBLE FUTURE EXTENSION: possibility to associate some backgrounds to some
; object types only. Adding a background in a "backgrounds" sub-directory in the
```

(continues on next page)

(continued from previous page)

```
; object directory may allow this.
; POSSIBLE FUTURE EXTENSION: specify ROIs for blending some object types.
ApplyToLabels=*backgrounds*
; Indicate whether multiple object types can be mixed on the same background
TypeMixing=0
; Density of the object in the background, from 0.0 to 1.0
DensityRange=0.0 0.2
; Horizontal margin between objects (in pixels)
MarginH=0
; Vertical margin between objects (in pixels)
MarginV=0
; Blending method
; POSSIBLE FUTURE EXTENSION: add other blending methods...
BlendingMethod=SmoothEdge
BlendingSmoothSize=5
; For DEBUG purpose, specifying a non-empty SavePath will save all the generated
; blending with their associated JSON annotation in the SavePath directory.
SavePath=blending

...
```

The second part is the object pre-processing and extraction pipeline, that is attached to the `BlendingTransformation`.

```
; --- BEGIN --- DATA TO BLEND PRE-PROCESSING ---
; Database driver for the objects. Can be a sub-set of the main pipe image
; dataset, or even the full main dataset itself
[database_objects]
Type=DIR_Database
DataPath=${DATA_PATH}
Depth=-1
LabelDepth=1
Learn=1.0
EquivLabelPartitioning=0
; Since we use the same dataset, ignore the background images that contain
; no object to blend.
IgnoreMasks=*backgrounds*
DefaultLabel=background ; Label for pixels outside any ROI (default is no label, pixels␣
↪are ignored)

; Simuli provider for objects => no need to change this part.
[sp_objects]
; Sizes to 0 means any size, require that BatchSize=0
SizeX=0
SizeY=0
BatchSize=0

; Apply random rotation & scaling to objects
; POSSIBLE FUTURE EXTENSION: apply different transformations depending on the
; type of object
[sp_objects.OnTheFlyTransformation-0]
Type=SliceExtractionTransformation
; Sizes to 0 means any size, size will not be changed by the transformation
```

(continues on next page)

(continued from previous page)

```
Width=0
Height=0
RandomRotation=1
RandomScaling=1
RandomScalingRange=0.5 2.0

; ... add here other transformations to apply to objects before extraction and
; blending

; Extend the object labels to have a smooth transition with background
[sp_objects.OnTheFlyTransformation-1]
Type=MorphologyTransformation
Operation=Dilate
Size=3
ApplyToLabels=1
NbIterations=2

; This has to be the last transformation in the pre-processing of the images
; that will be blended.
; After data augmentation, a random object is extracted from the image,
; using ROIs or connected-component labeling.
[sp_objects.OnTheFlyTransformation-2]
Type=ROIExtractionTransformation
; Extract any label ID
Label=-1
; Perform connected-component labeling to the label to obtain objects ROIs.
LabelSegmentation=1
Margin=0
KeepComposite=1
; Possibility to filter the ROIs to keep before random selection of a single
; one:
MinSize=0
FilterMinHeight=0
FilterMinWidth=0
FilterMinAspectRatio=0.0
FilterMaxAspectRatio=0.0
MergeMaxHDist=10
MergeMaxVDist=10
; --- END --- DATA TO BLEND PRE-PROCESSING ---
```

21.5 Built-in transformations

There are 6 possible categories of transformations:

- `env.Transformation[...]` Transformations applied to the input images before channels creation;
- `env.OnTheFlyTransformation[...]` On-the-fly transformations applied to the input images before channels creation;
- `env.ChannelTransformation[...]` Create or add transformation for a specific channel;
- `env.ChannelOnTheFlyTransformation[...]` Create or add on-the-fly transformation for a specific channel;

- `env.ChannelsTransformation[...]` Transformations applied to all the channels of the input images;
- `env.ChannelsOnTheFlyTransformation[...]` On-the-fly transformations applied to all the channels of the input images.

Example:

```
[env.Transformation]
Type=PadCropTransformation
Width=24
Height=24
```

Several transformations can applied successively. In this case, to be able to apply multiple transformations of the same category, a different suffix (`[...]`) must be added to each transformation.

The transformations will be processed in the order of appearance in the INI file regardless of their suffix.

Common set of parameters for any kind of transformation:

Option [default value]	Description
`ApplyTo` [`All`]	Apply the transformation only to the specified stimuli sets. Can be:
	`LearnOnly`: learning set only
	`ValidationOnly`: validation set only
	`TestOnly`: testing set only
	`NoLearn`: validation and testing sets only
	`NoValidation`: learning and testing sets only
	`NoTest`: learning and validation sets only
	`All`: all sets (default)

Example:

```
[env.Transformation-1]
Type=ChannelExtractionTransformation
CSChannel=Gray

[env.Transformation-2]
Type=RescaleTransformation
Width=29
Height=29

[env.Transformation-3]
Type=EqualizeTransformation

[env.OnTheFlyTransformation]
Type=DistortionTransformation
ApplyTo=LearnOnly ; Apply this transformation for the Learning set only
ElasticGaussianSize=21
ElasticSigma=6.0
ElasticScaling=20.0
Scaling=15.0
Rotation=15.0
```

List of available transformations:

21.5.1 AffineTransformation

Apply an element-wise affine transformation to the image with matrixes of the same size.

Option [default value]	Description
`FirstOperator`	First element-wise operator, can be `Plus`, `Minus`, `Multiplies`, `Divides`
`FirstValue`	First matrix file name
`SecondOperator` [`Plus`]	Second element-wise operator, can be `Plus`, `Minus`, `Multiplies`, `Divides`
`SecondValue` []	Second matrix file name

The final operation is the following, with A the image matrix, B_{1st}, B_{2nd} the matrixes to add/substract/multiply/divide and $\odot$ the element-wise operator :

$$f(A) = \left(A \underset{op_{1st}}{\odot} B_{1st}\right) \underset{op_{2nd}}{\odot} B_{2nd}$$

21.5.2 ApodizationTransformation

Apply an apodization window to each data row.

Option [default value]	Description
`Size`	Window total size (must match the number of data columns)
`WindowName` [`Rectangular`]	Window name. Possible values are:
	`Rectangular`: Rectangular
	`Hann`: Hann
	`Hamming`: Hamming
	`Cosine`: Cosine
	`Gaussian`: Gaussian
	`Blackman`: Blackman
	`Kaiser`: Kaiser

Gaussian window

Gaussian window.

Option [default value]	Description
WindowName`.Sigma` [0.4]	Sigma

Blackman window

Blackman window.

Option [default value]	Description
WindowName`.Alpha` [0.16]	Alpha

Kaiser window

Kaiser window.

Option [default value]	Description
WindowName`.Beta` [5.0]	Beta

21.5.3 CentroidCropTransformation

Find the centroid of the image and crop the image so that the center of the image matches the centroid location. The cropping can be done on both axis, or just one axis with the `Axis` parameter. If `Axis` is 1, only the horizontal axis will be cropped so that the centroid x-location is at half the image width.

Option [default value]	Description
`Axis` [-1]	Axis to consider for the centroid (-1 = both, 0 = cols, 1 = rows)

In practice, this transformation can be used in conjunction with the `PadCropTransformation`, in order to obtain cropped images of always of the same dimension (by cropping for example to the smallest image obtained after `CentroidCropTransformation`), all centered on their respective centroid.

21.5.4 BlendingTransformation

N2D2-IP only: available upon request.

This transformation can be used to blend image objects, provided by another `Database` and associated `StimuliProvider`, to the images of the current `StimuliProvider`.

Option [default value]	Description
`Database`	Name of the `Database` section to use for the objects to blend
`StimuliProvider`	Name of the `StimuliProvider` section specifying the transformations to apply on the objects data before blending
`ApplyToLabels` []	Space-separated list that specifies the name of the image label(s) on which a blending can be performed (in the current data pipe). The usual * and + wildcards are allowed.
`TypeMixing` [0]	If true (1), multiple object types can be mixed on the same image
`DensityRange` [0.0 0.0]	Range of density of the objects to blend in the image (values are from 0.0 to 1.0). A different random density in this range is used for each image. If the two values are equal, the density is constant. A constant density of 0 (corresponding the default range [0.0 0.0]) means that only a single object is blended in the image in all cases, regardless of the object size. Indeed, the density parameter is checked only *after* the first object was inserted.
`MarginH` [0]	Minimum horizontal margin between inserted objects (in pixels)
`MarginV` [0]	Minimum vertical margin between inserted objects (in pixels)
`BlendingMethod` [Linear]	Blending method to use (see the `BlendingMethod` section)
`BlendingAlpha` [0.2]	Alpha factor for the blending. Depends on the blending method (see the `BlendingMethod` section)
`BlendingBeta` [0.8]	Beta factor for the blending. Depends on the blending method (see the `BlendingMethod` section)
`BlendingSmoothSize` [5]	Smoothing kernel size, used in some blending methods (see the `BlendingMethod` section)
`SavePath` []	If not empty, all the blended images are stored in `SavePath` during the simulation

`BlendingMethod`

In the following equations, O is the object image, I is the image of the current pipe on which objects must be inserted. And R is the resulting image.

`Linear`: no smoothing.

$$R = \alpha.O + \beta.I$$

`LinearByDistance`: limit the blur in the blended object background.

$$\Delta = \frac{\|O-I\| - min(\|O-I\|)}{max(\|O-I\|) - min(\|O-I\|)}$$
$$R = \alpha.O.(1-\Delta) + \beta.I.\Delta$$

`SmoothEdge`: smoothing at the borders of the objects.

$$\alpha = \begin{cases} 1 & \text{when } LABEL \neq 0 \\ 0 & \text{otherwise} \end{cases}$$
$$\alpha' = gaussian\_blur(\alpha)$$
$$R = \alpha'.O + (1-\alpha').I$$

SmoothEdgeLinearByDistance: combines SmoothEdge and LinearByDistance.

$$\alpha = \begin{cases} \Delta & \text{when } LABEL \neq 0 \\ 0 & \text{otherwise} \end{cases}$$
$$\alpha' = gaussian\_blur(\alpha)$$
$$R = \alpha'.O + (1 - \alpha').I$$

Labels mapping

When processing the first batch of data, you might get a message like the following in the console:

```
BlendingTransformation: labels mapping is required with the following mapping:
  1 -> 9   (cat)
  2 -> 12   (dog)
  3 -> 66   (bird)
```

What happens here is that the labels ID from the database containing the objects to blend (specified by the `Database` parameter) must match the correct labels ID from the current database (specified by the `[database]` section). In the log above, the labels ID on the left are the ones from the objects database and the labels ID on the right are the ones from the current database. In N2D2, upon loading a database, a new label ID is created for each new unique label name encoutered, in the loading order (alphabetical for `DIR_Database`, but may be arbitrary for other database drivers). The objects database may contain only a subset of the labels present in the current database, and/or the labels may be loaded in a different order. In both cases, the ID affected to a label name will be different between the two databases. During blending however, one wants that the blended object labels correspond to the labels of the current database. To solve this, labels mapping is automatically performed in N2D2 so that for corresponding label names, the label ID in the objects database is translated to the label ID of current database. In the log above for example, the objects database contains only 3 labels: "cat", "dog" and "bird", with ID 1, 2 and 3 respectively. These labels ID are automatically replaced by the corresponding ID (for identical label name) in the current database, for the blended objects, which are here 9, 12 and 66 respectively.

Note: Each label from the objects database (objects to blend) must match an existing label in the current database. There is a match if:

- There is an identical label name in the current database;
- There is a single label name in the current database that ends with the objects database label name. For example, the label "/dog" in the objects database will match with the "dog" label in the current database.

If the objects database contains a label name that does not exist/match in the current database, an error is emitted:

```
BlendingTransformation: label "xxx" in blending database not present in current database!
```

21.5.5 ChannelDropTransformation

N2D2-IP only: available upon request.

Randomly drop some channels of the image and replace them with a constant value. This can be useful to simulate missing channel data in multi-channel data.

Option [default value]	Description
`DropProb`	Channel's drop probabilities (space-separated list of probabilities, in the order of the image channels)
`DropValue` [0.0]	Value to use for dropped channels pixels

21.5.6 ChannelExtractionTransformation

Extract an image channel.

Option	Description
`CSChannel`	`Blue`: blue channel in the BGR colorspace, or first channel of any colorspace
	`Green`: green channel in the BGR colorspace, or second channel of any colorspace
	`Red`: red channel in the BGR colorspace, or third channel of any colorspace
	`Hue`: hue channel in the HSV colorspace
	`Saturation`: saturation channel in the HSV colorspace
	`Value`: value channel in the HSV colorspace
	`Gray`: gray conversion
	`Y`: Y channel in the YCbCr colorspace
	`Cb`: Cb channel in the YCbCr colorspace
	`Cr`: Cr channel in the YCbCr colorspace

21.5.7 ChannelShakeTransformation

N2D2-IP only: available upon request.

Randomly shift some channels of the image. This can be useful to simulate misalignment between multiple channel data.

Option [default value]	Description
VerticalRange[*]	Vertical shift range (in pixels) for each channel. For example, to randomly shift the second channel by +/- 5 pixels in the vertical direction, use: `VerticalRange[1]=-5.0 5.0`
HorizontalRange[*]	Horizontal shift range (in pixels) for each channel
Distribution [Uniform]	Random distribution to use for the shift
Rounded [1]	If true (1), use integer value for the shifts (no pixel interpolation needed)
BorderType [MinusOneReflectBorder]	Border type used when padding. Possible values: `ConstantBorder`: pad with `BorderValue` `ReplicateBorder`: last element is replicated throughout, like aaaaaa|abcdefgh|hhhhhhh `ReflectBorder`: border will be mirror reflection of the border elements, like fedcba|abcdefgh|hgfedcb `WrapBorder`: it will look like cdefgh|abcdefgh|abcdefg `MinusOneReflectBorder`: same as `ReflectBorder` but with a slight change, like gfedcb|abcdefgh|gfedcba `MeanBorder`: pad with the mean color of the image
BorderValue [0.0 0.0 0.0]	Background color used when padding with `BorderType` is `ConstantBorder`

Distribution

Possible distribution and meaning of the range. For example with `VerticalRange[1]=-5.0 5.0`.

Uniform
: Uniform between -5 and 5.

Normal
: Normal with mean (-5+5)/2=0 and std. dev. = (5-(-5))/6 = 1.67. The range defines the std. dev. such that range = 6 sigma.

TruncatedNormal
: Same as `Normal`, but truncated between -5 and 5.

21.5.8 ColorSpaceTransformation

Change the current image colorspace.

Option	Description
`ColorSpace`	`BGR`: convert any gray, BGR or BGRA image to BGR
	`RGB`: convert any gray, BGR or BGRA image to RGB
	`HSV`: convert BGR image to HSV
	`HLS`: convert BGR image to HLS
	`YCrCb`: convert BGR image to YCrCb
	`CIELab`: convert BGR image to CIELab
	`CIELuv`: convert BGR image to CIELuv
	`RGB_to_BGR`: convert RGB image to BGR
	`RGB_to_HSV`: convert RGB image to HSV
	`RGB_to_HLS`: convert RGB image to HLS
	`RGB_to_YCrCb`: convert RGB image to YCrCb
	`RGB_to_CIELab`: convert RGB image to CIELab
	`RGB_to_CIELuv`: convert RGB image to CIELuv
	`HSV_to_BGR`: convert HSV image to BGR
	`HSV_to_RGB`: convert HSV image to RGB
	`HLS_to_BGR`: convert HLS image to BGR
	`HLS_to_RGB`: convert HLS image to RGB
	`YCrCb_to_BGR`: convert YCrCb image to BGR
	`YCrCb_to_RGB`: convert YCrCb image to RGB
	`CIELab_to_BGR`: convert CIELab image to BGR
	`CIELab_to_RGB`: convert CIELab image to RGB
	`CIELuv_to_BGR`: convert CIELuv image to BGR
	`CIELuv_to_RGB`: convert CIELuv image to RGB

Note that the default colorspace in N2D2 is BGR, the same as in OpenCV.

21.5.9 DFTTransformation

Apply a DFT to the data. The input data must be single channel, the resulting data is two channels, the first for the real part and the second for the imaginary part.

Option [default value]	Description
`TwoDimensional` [1]	If true, compute a 2D image DFT. Otherwise, compute the 1D DFT of each data row

Note that this transformation can add zero-padding if required by the underlying FFT implementation.

21.5.10 DistortionTransformation

Apply elastic distortion to the image. This transformation is generally used on-the-fly (so that a different distortion is performed for each image), and for the learning only.

Option [default value]	Description
`ElasticGaussianSize` [15]	Size of the gaussian for elastic distortion (in pixels)
`ElasticSigma` [6.0]	Sigma of the gaussian for elastic distortion
`ElasticScaling` [0.0]	Scaling of the gaussian for elastic distortion
`Scaling` [0.0]	Maximum random scaling amplitude (+/-, in percentage)
`Rotation` [0.0]	Maximum random rotation amplitude (+/-, in °)

21.5.11 EqualizeTransformation

Image histogram equalization.

Option [default value]	Description
`Method` [`Standard`]	`Standard`: standard histogram equalization
	`CLAHE`: contrast limited adaptive histogram equalization
`CLAHE_ClipLimit` [40.0]	Threshold for contrast limiting (for `CLAHE` only)
`CLAHE_GridSize` [8]	Size of grid for histogram equalization (for `CLAHE` only). Input image will be divided into equally sized rectangular tiles. This parameter defines the number of tiles in row and column.

21.5.12 ExpandLabelTransformation

Expand single image label (1x1 pixel) to full frame label.

21.5.13 FilterTransformation

Apply a convolution filter to the image.

Option [default value]	Description
`Kernel`	Convolution kernel. Possible values are:
	*: custom kernel
	`Gaussian`: Gaussian kernel
	`LoG`: Laplacian Of Gaussian kernel
	`DoG`: Difference Of Gaussian kernel
	`Gabor`: Gabor kernel

* kernel

Custom kernel.

Option	Description
`Kernel.SizeX` [0]	Width of the kernel (numer of columns)
`Kernel.SizeY` [0]	Height of the kernel (number of rows)
`Kernel.Mat`	List of row-major ordered coefficients of the kernel

If both `Kernel.SizeX` and `Kernel.SizeY` are 0, the kernel is assumed to be square.

Note: When providing a custom kernel, no normalization is applied on its coefficients.

Gaussian kernel

Gaussian kernel.

Option [default value]	Description
`Kernel.SizeX`	Width of the kernel (numer of columns)
`Kernel.SizeY`	Height of the kernel (number of rows)
`Kernel.Positive` [1]	If true, the center of the kernel is positive
`Kernel.Sigma` [$\sqrt{2.0}$]	Sigma of the kernel

LoG kernel

Laplacian Of Gaussian kernel.

Option [default value]	Description
`Kernel.SizeX`	Width of the kernel (numer of columns)
`Kernel.SizeY`	Height of the kernel (number of rows)
`Kernel.Positive` [1]	If true, the center of the kernel is positive
`Kernel.Sigma` [$\sqrt{2.0}$]	Sigma of the kernel

DoG kernel

Difference Of Gaussian kernel kernel.

Option [default value]	Description
`Kernel.SizeX`	Width of the kernel (numer of columns)
`Kernel.SizeY`	Height of the kernel (number of rows)
`Kernel.Positive` [1]	If true, the center of the kernel is positive
`Kernel.Sigma1` [2.0]	Sigma1 of the kernel
`Kernel.Sigma2` [1.0]	Sigma2 of the kernel

Gabor kernel

Gabor kernel.

Option [default value]	Description
`Kernel.SizeX`	Width of the kernel (numer of columns)
`Kernel.SizeY`	Height of the kernel (number of rows)
`Kernel.Theta`	Theta of the kernel
`Kernel.Sigma` [$\sqrt{2.0}$]	Sigma of the kernel
`Kernel.Lambda` [10.0]	Lambda of the kernel
`Kernel.Psi` [$\pi/2.0$]	Psi of the kernel
`Kernel.Gamma` [0.5]	Gamma of the kernel

21.5.14 FlipTransformation

Image flip transformation.

Option [default value]	Description
`HorizontalFlip` [0]	If true, flip the image horizontally
`VerticalFlip` [0]	If true, flip the image vertically
`RandomHorizontalFlip` [0]	If true, randomly flip the image horizontally
`RandomVerticalFlip` [0]	If true, randomly flip the image vertically

21.5.15 GradientFilterTransformation

Compute image gradient.

Option [default value]	Description
`Scale` [1.0]	Scale to apply to the computed gradient
`Delta` [0.0]	Bias to add to the computed gradient
`GradientFilter` [`Sobel`]	Filter type to use for computing the gradient. Possible options are: `Sobel`, `Scharr` and `Laplacian`
`KernelSize` [3]	Size of the filter kernel (has no effect when using the `Scharr` filter, which kernel size is always 3x3)
`ApplyToLabels` [0]	If true, use the computed gradient to filter the image label and ignore pixel areas where the gradient is below the `Threshold`. In this case, only the labels are modified, not the image
`InvThreshold` [0]	If true, ignored label pixels will be the ones with a low gradient (low contrasted areas)
`Threshold` [0.5]	Threshold applied on the image gradient
`Label` []	List of labels to filter (space-separated)
`GradientScale` [1.0]	Rescale the image by this factor before applying the gradient and the threshold, then scale it back to filter the labels

21.5.16 LabelFilterTransformation

Filter labels in the image. The specified labels can be removed, kept (meaning all the other labels removed), or merged (the specified labels are replace by the first one).

Option [default value]	Description
`Labels`	Space-separated list of label names to be filtered
`Filter` [`Remove`]	Type of filter to apply: `Remove`, `Keep` (labels not in the list are removed) or `Merge` (labels in the list are all replaced by the first one)
`DefaultLabel` [-2]	Default label, to be used where labels are removed. With the default value (-2), the default label of the associated database is used. If there is no default label, -1 (ignore) is used

This transformation filters both pixel-wise labels and ROIs.

21.5.17 LabelSliceExtractionTransformation

Extract a slice from an image belonging to a given label.

Option [default value]	Description
`Width`	Width of the slice to extract
`Height`	Height of the slice to extract
`Label` [-1]	Slice should belong to this label ID. If -1, the label ID is random
`RandomRotation` [0]	If true, extract randomly rotated slices
`RandomRotationRange` [0.0 360.0]	Range of the random rotations, in degrees, counterclockwise (if `RandomRotation` is enabled)
`SlicesMargin` [0]	Positive or negative, indicates the margin around objects that can be extracted in the slice
`KeepComposite` [0]	If false, the 2D label image is reduced to a single value corresponding to the extracted object label (useful for patches classification tasks). Note that if `SlicesMargin` is > 0, the 2D label image may contain other labels before reduction. For pixel-wise segmentation tasks, set `KeepComposite` to true.
`AllowPadding` [0]	If true, zero-padding is allowed if the image is smaller than the slice to extract
`BorderType` [`MinusOneReflectBorder`]	Border type used when padding. Possible values:
	`ConstantBorder`: pad with `BorderValue`
	`ReplicateBorder`: last element is replicated throughout, like aaaaaa|abcdefgh|hhhhhhh
	`ReflectBorder`: border will be mirror reflection of the border elements, like fedcba|abcdefgh|hgfedcb
	`WrapBorder`: it will look like cdefgh|abcdefgh|abcdefg
	`MinusOneReflectBorder`: same as `ReflectBorder` but with a slight change, like gfedcb|abcdefgh|gfedcba
	`MeanBorder`: pad with the mean color of the image
`BorderValue` [0.0 0.0 0.0]	Background color used when padding with `BorderType` is `ConstantBorder`
`IgnoreNoValid` [1]	If true (1), if no valid slice is found, a random slice is extracted and marked as ignored (-1)
`ExcludeLabels` []	Space-separated list of label ID to exclude from the random extraction (when `Label` is -1)

This transformation is useful to learn sparse object occurrences in a lot of background. If the dataset is very unbalanced towards background, this transformation will ensure that the learning is done on a more balanced set of every labels, regardless of their actual pixel-wise ratio.

Illustration of the working behavior of `LabelSliceExtractionTransformation` with `SlicesMargin` = 0:

When `SlicesMargin` is 0, only slices that fully include a given label are extracted, as shown in figures above. The behavior with `SlicesMargin` < 0 is illustrated in figures below. Note that setting a negative `SlicesMargin` larger in absolute value than `Width`/2 or `Height`/2 will lead in some (random) cases in incorrect slice labels in respect to the majority pixel label in the slice.

Illustration of the working behavior of `LabelSliceExtractionTransformation` with `SlicesMargin` = -32:

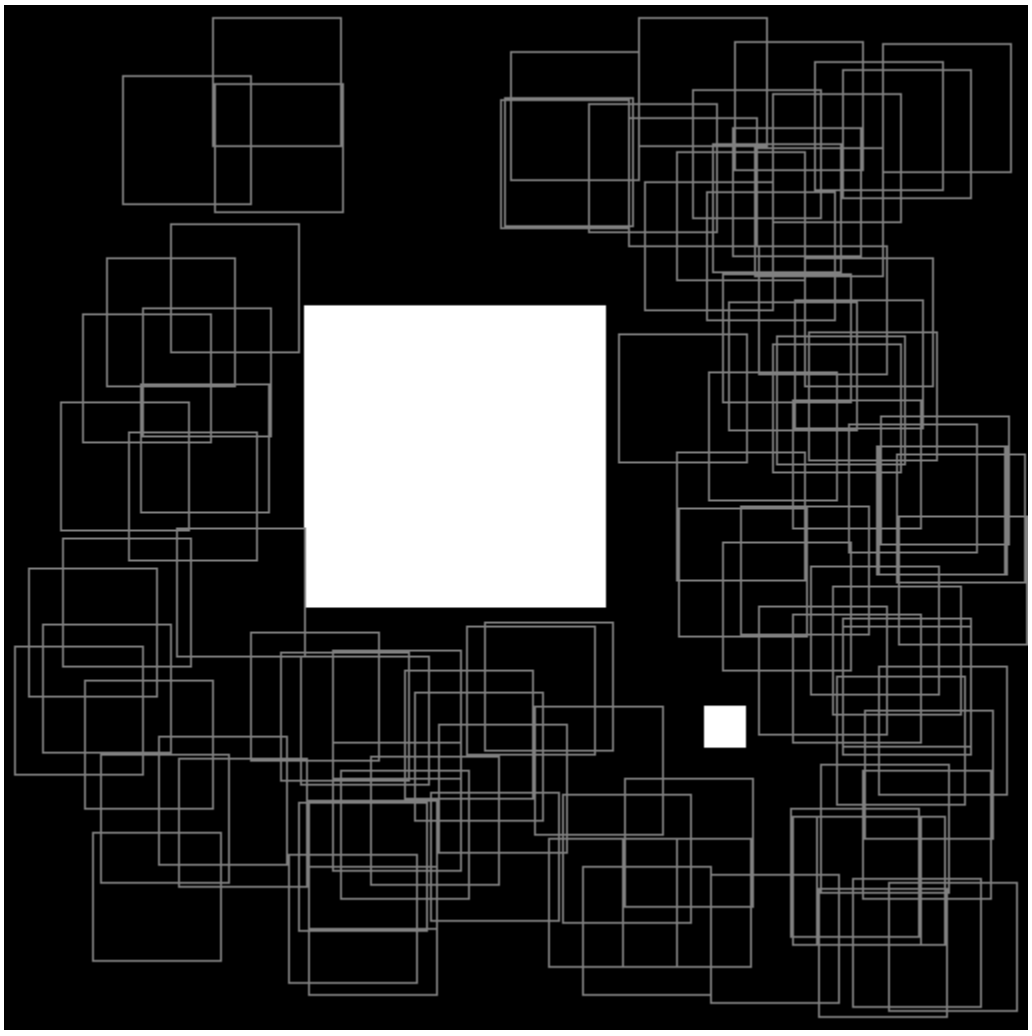


Fig. 4: Randomly extracted slices with label 0.

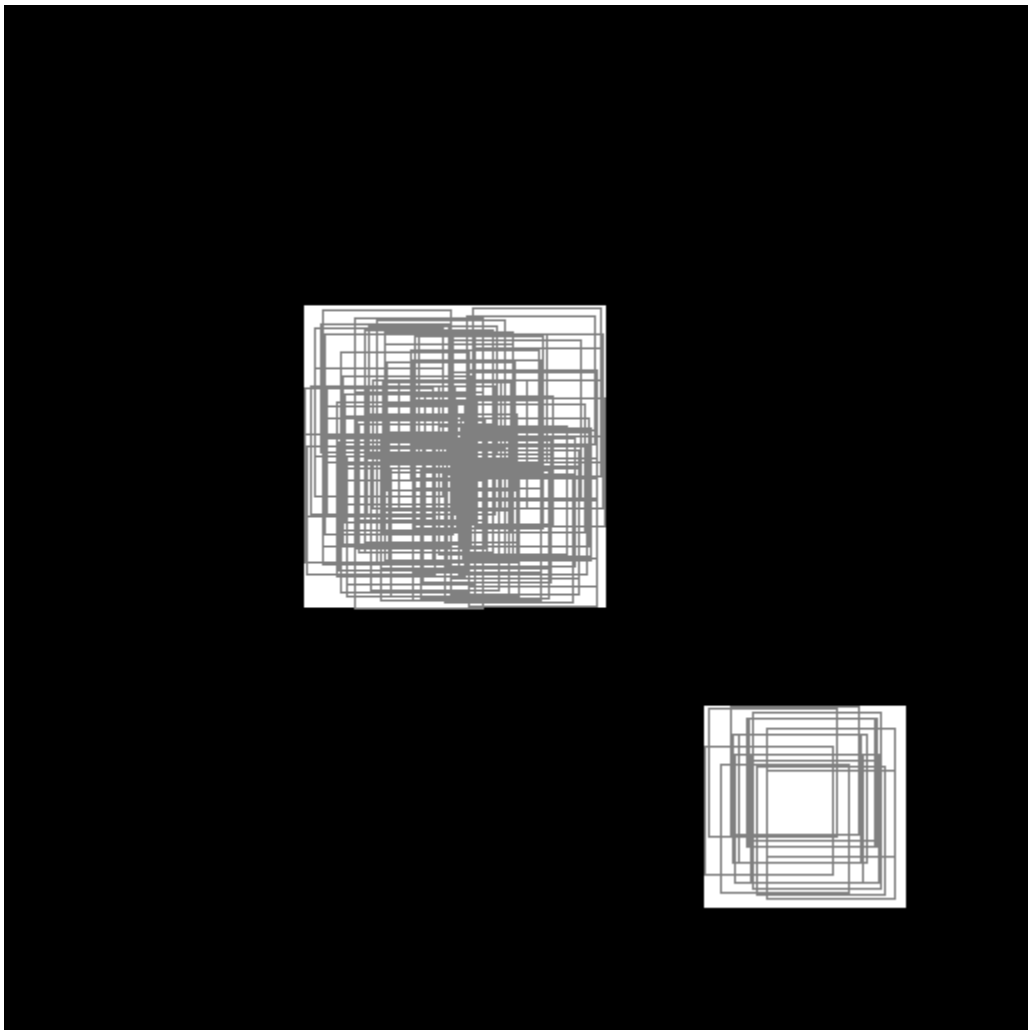


Fig. 5: Randomly extracted slices with label 1.

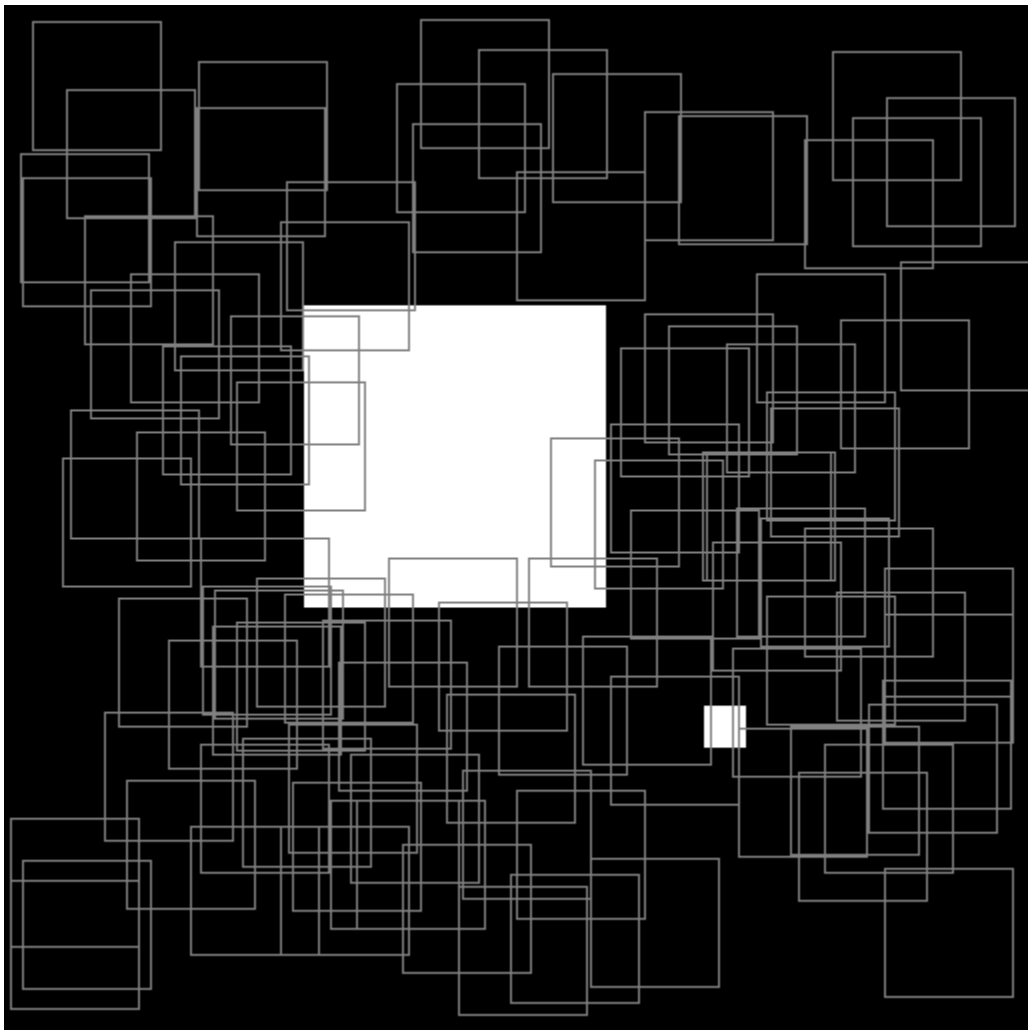


Fig. 6: Randomly extracted slices including label 0.

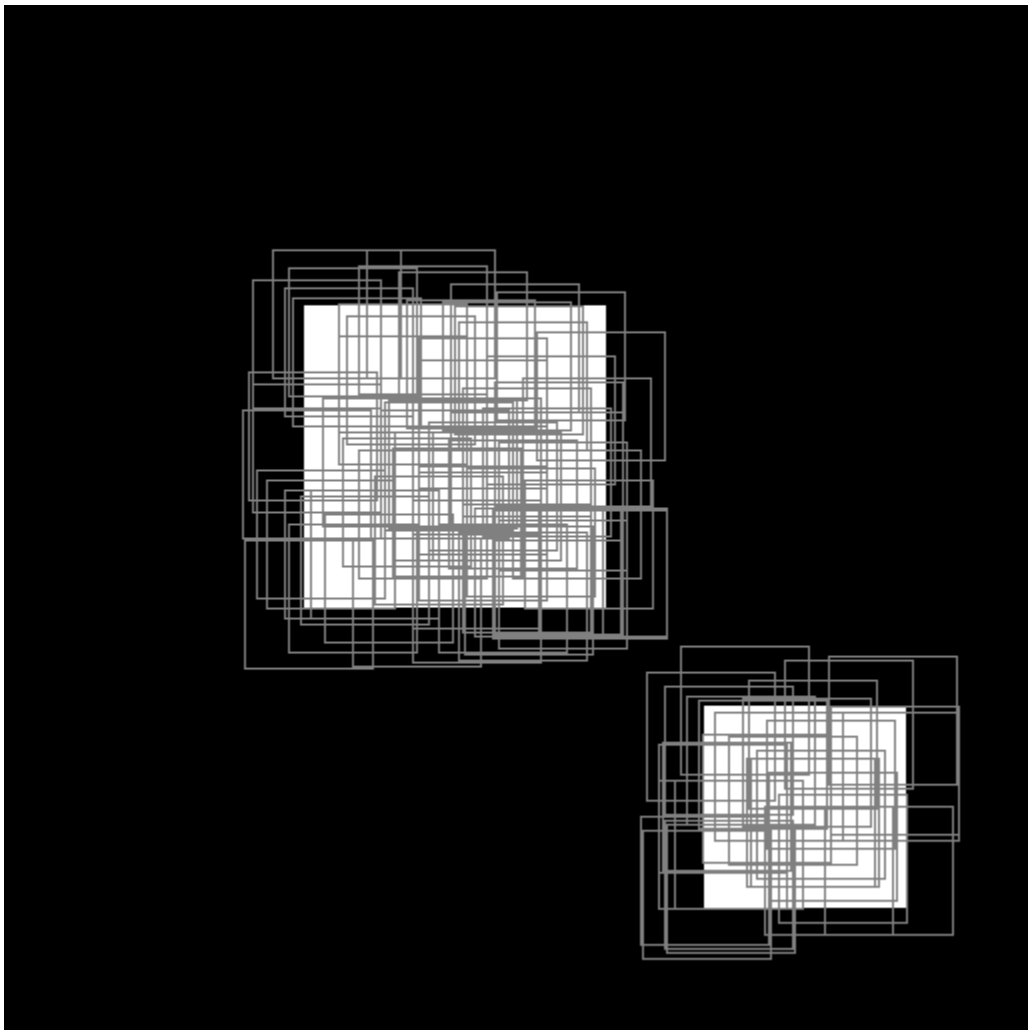


Fig. 7: Randomly extracted slices including label 1.

21.5.18 MagnitudePhaseTransformation

Compute the magnitude and phase of a complex two channels input data, with the first channel x being the real part and the second channel y the imaginary part. The resulting data is two channels, the first one with the magnitude and the second one with the phase.

Option [default value]	Description
`LogScale` [0]	If true, compute the magnitude in log scale

The magnitude is:

$$M_{i,j} = \sqrt{x_{i,j}^2 + x_{i,j}^2}$$

If `LogScale` = 1, compute $M'_{i,j} = log(1 + M_{i,j})$.

The phase is:

$$\theta_{i,j} = atan2(y_{i,j}, x_{i,j})$$

21.5.19 MorphologicalReconstructionTransformation

Apply a morphological reconstruction transformation to the image. This transformation is also useful for post-processing.

Option [default value]	Description
`Operation`	Morphological operation to apply. Can be:
	`ReconstructionByErosion`: reconstruction by erosion operation
	`ReconstructionByDilation`: reconstruction by dilation operation
	`OpeningByReconstruction`: opening by reconstruction operation
	`ClosingByReconstruction`: closing by reconstruction operation
`Size`	Size of the structuring element
`ApplyToLabels` [0]	If true, apply the transformation to the labels instead of the image
`Shape` [`Rectangular`]	Shape of the structuring element used for morphology operations. Can be `Rectangular`, `Elliptic` or `Cross`.
`NbIterations` [1]	Number of times erosion and dilation are applied for opening and closing reconstructions

21.5.20 MorphologyTransformation

Apply a morphology transformation to the image. This transformation is also useful for post-processing.

Option [default value]	Description
`Operation`	Morphological operation to apply. Can be:
	`Erode`: erode operation ($= erode(src)$)
	`Dilate`: dilate operation ($= dilate(src)$)
	`Opening`: opening operation ($open(src) = dilate(erode(src))$)
	`Closing`: closing operation ($close(src) = erode(dilate(src))$)
	`Gradient`: morphological gradient ($= dilate(src) - erode(src)$)
	`TopHat`: top hat ($= src - open(src)$)
	`BlackHat`: black hat ($= close(src) - src$)
`Size`	Size of the structuring element
`ApplyToLabels` [0]	If true, apply the transformation to the labels instead of the image
`Shape` [`Rectangular`]	Shape of the structuring element used for morphology operations. Can be `Rectangular`, `Elliptic` or `Cross`.
`NbIterations` [1]	Number of times erosion and dilation are applied

21.5.21 NormalizeTransformation

Normalize the image.

Option [default value]	Description
`Norm` [`MinMax`]	Norm type, can be:
	`L1`: L1 normalization
	`L2`: L2 normalization
	`Linf`: Linf normalization
	`MinMax`: min-max normalization
`NormValue` [1.0]	Norm value (for `L1`, `L2` and `Linf`)
	Such that $\|\|data\|\|_{L_p} = NormValue$
`NormMin` [0.0]	Min value (for `MinMax` only)
	Such that $min(data) = NormMin$
`NormMax` [1.0]	Max value (for `MinMax` only)
	Such that $max(data) = NormMax$
`PerChannel` [0]	If true, normalize each channel individually

21.5.22 PadCropTransformation

Pad/crop the image to a specified size.

Option [default value]	Description
`Width`	Width of the padded/cropped image
`Height`	Height of the padded/cropped image
`BorderType` [`MinusOneReflectBorder`]	Border type used when padding. Possible values:
	`ConstantBorder`: pad with `BorderValue`
	`ReplicateBorder`: last element is replicated throughout, like aaaaaa|abcdefgh|hhhhhhh
	`ReflectBorder`: border will be mirror reflection of the border elements, like fedcba|abcdefgh|hgfedcb
	`WrapBorder`: it will look like cdefgh|abcdefgh|abcdefg
	`MinusOneReflectBorder`: same as `ReflectBorder` but with a slight change, like gfedcb|abcdefgh|gfedcba
	`MeanBorder`: pad with the mean color of the image
`BorderValue` [0.0 0.0 0.0]	Background color used when padding with `BorderType` is `ConstantBorder`

21.5.23 ROIExtractionTransformation

The transformation is typically used as the last transformation of the object extraction pipeline to be used for blending in a `BlendingTransformation`. A random object of with the label `Label` is extracted from the image.

Option [default value]	Description
`Label` [-1]	Label ID to extract (-1 means any label ID)
`LabelSegmentation` [0]	If true (1), perform connected-component labeling to the label to obtain object ROIs
`Margin` [0]	Margin to keep around the object (in pixels)
`KeepComposite` [1]	If true (1), the extracted object label remains composite. Otherwise, the label is reduced to a single value

When `LabelSegmentation` is 0, this transformation directly extracts one of the annotation ROI whose label matches `Label`. When `LabelSegmentation` is true (1), the annotation ROIs are not used directly. Rather, the flattened pixel-wise annotation is (re-)labeled using connected-component labeling to obtain ROIs to extract. Note that the annotation ROIs are part of the flattened pixel-wise annotation (see also the `Database` `CompositeLabel` parameter).

Additional parameters for ROI filtering, before random selection of a single one:

Parameter	Default value	Description
`MinSize`	0	Minimum number of pixels than can constitute a bounding box. Bounding boxes with fewer than `MinSize` pixels are discarded
`FilterMinHeight`	0	Minimum height of the ROI to keep it
`FilterMinWidth`	0	Minimum width of the ROI to keep it
`FilterMinAspectRatio`	0.0	Minimum aspect ratio (width/height) of the ROI to keep it (default is 0.0 = no minimum)
`FilterMaxAspectRatio`	0.0	Maximum aspect ratio (width/height) of the ROI to keep it (default is 0.0 = no minimum)
`MergeMaxHDist`	1	Maximum horizontal distance for merging (in pixels)
`MergeMaxVDist`	1	Maximum vertical distance for merging (in pixels)

Note that these parameters applies only when `LabelSegmentation` is true (1).

21.5.24 RandomAffineTransformation

Apply a global random affine transformation to the values of the image.

Option [default value]	Description
GainRange [1.0 1.0]	Random gain (α) range (identical for all channels)
GainRange[*] [1.0 1.0]	Random gain (α) range for channel *. Mutually exclusive with GainRange. If any specified, a different random gain will always be sampled for each channel. Default gain is 1.0 (no gain) for missing channels
	The gain control the *contrast* of the image
BiasRange [0.0 0.0]	Random bias (β) range (identical for all channels)
BiasRange[*] [0.0 0.0]	Random bias (β) range for channel *. Mutually exclusive with BiasRange. If any specified, a different random bias will always be sampled for each channel. Default bias is 0.0 (no bias) for missing channels
	The bias control the *brightness* of the image
GammaRange [1.0 1.0]	Random gamma (γ) range (identical for all channels)
GammaRange[*] [1.0 1.0]	Random gamma (γ) range for channel *. Mutually exclusive with GammaRange. If any specified, a different random gamma will always be sampled for each channel. Default gamma is 1.0 (no change) for missing channels
	The gamma control more or less the *exposure* of the image
GainVarProb [1.0]	Probability to have a gain variation for each channel. If only one value is specified, the same probability applies to all the channels. In this case, the same gain variation will be sampled for all the channels only if a single range if specified for all the channels using GainRange. If more than one value is specified, a different random gain will always be sampled for each channel, even if the probabilities and ranges are identical
BiasVarProb [1.0]	Probability to have a bias variation for each channel. If only one value is specified, the same probability applies to all the channels. In this case, the same bias variation will be sampled for all the channels only if a single range if specified for all the channels using BiasRange. If more than one value is specified, a different random bias will always be sampled for each channel, even if the probabilities and ranges are identical
GammaVarProb [1.0]	Probability to have a gamma variation for each channel. If only one value is specified, the same probability applies to all the channels. In this case, the same gamma variation will be sampled for all the channels only if a single range if specified for all the channels using GammaRange. If more than one value is specified, a different random gamma will always be sampled for each channel, even if the probabilities and ranges are identical
DisjointGamma [0]	If true, gamma variation and gain/bias variation are mutually exclusive. The probability to have a random gamma variation is therefore GammaVarProb and the probability to have a gain/bias variation is 1-GammaVarProb.
ChannelsMask []	If not empty, specifies on which channels the transformation is applied. For example, to apply the transformation only to the first and third channel, set ChannelsMask to `1 0 1`

The equation of the transformation is:

$$S = \begin{cases} \text{numeric\_limits<T>::max()} & \text{if is\_integer<T>} \\ 1.0 & \text{otherwise} \end{cases}$$

$$v(i,j) = \text{cv::saturate\_cast<T>}\left(\alpha\left(\frac{v(i,j)}{S}\right)^{\gamma} S + \beta.S\right)$$

21.5.25 RangeAffineTransformation

Apply an affine transformation to the values of the image.

Option [default value]	Description
`FirstOperator`	First operator, can be `Plus`, `Minus`, `Multiplies`, `Divides`
`FirstValue`	First value
`SecondOperator` [`Plus`]	Second operator, can be `Plus`, `Minus`, `Multiplies`, `Divides`
`SecondValue` [0.0]	Second value

The final operation is the following:

$$f(x) = (x \ \overset{o}{op}_{1st} \ val_{1st}) \ \overset{o}{op}_{2nd} \ val_{2nd}$$

21.5.26 RangeClippingTransformation

Clip the value range of the image.

Option [default value]	Description
`RangeMin` [$min(data)$]	Image values below `RangeMin` are clipped to 0
`RangeMax` [$max(data)$]	Image values above `RangeMax` are clipped to 1 (or the maximum integer value of the data type)

21.5.27 RescaleTransformation

Rescale the image to a specified size.

Option [default value]	Description
`Width`	Width of the rescaled image
`Height`	Height of the rescaled image
`KeepAspectRatio` [0]	If true, keeps the aspect ratio of the image
`ResizeToFit` [1]	If true, resize along the longest dimension when `KeepAspectRatio` is true

21.5.28 ReshapeTransformation

Reshape the data to a specified size.

Option [default value]	Description
`NbRows`	New number of rows
`NbCols` [0]	New number of cols (0 = no check)
`NbChannels` [0]	New number of channels (0 = no change)

21.5.29 SliceExtractionTransformation

Extract a slice from an image.

Option [default value]	Description
`Width`	Width of the slice to extract
`Height`	Height of the slice to extract
`OffsetX` [0]	X offset of the slice to extract
`OffsetY` [0]	Y offset of the slice to extract
`RandomOffsetX` [0]	If true, the X offset is chosen randomly
`RandomOffsetY` [0]	If true, the Y offset is chosen randomly
`RandomRotation` [0]	If true, extract randomly rotated slices
`RandomRotationRange` [0.0 360.0]	Range of the random rotations, in degrees, counterclockwise (if `RandomRotation` is enabled)
`RandomScaling` [0]	If true, extract randomly scaled slices
`RandomScalingRange` [0.8 1.2]	Range of the random scaling (if `RandomRotation` is enabled)
`AllowPadding` [0]	If true, zero-padding is allowed if the image is smaller than the slice to extract
`BorderType` [`MinusOneReflectBorder`]	Border type used when padding. Possible values:
	`ConstantBorder`: pad with `BorderValue`
	`ReplicateBorder`: last element is replicated throughout, like aaaaaa|abcdefgh|hhhhhhh
	`ReflectBorder`: border will be mirror reflection of the border elements, like fedcba|abcdefgh|hgfedcb
	`WrapBorder`: it will look like cdefgh|abcdefgh|abcdefg
	`MinusOneReflectBorder`: same as `ReflectBorder` but with a slight change, like gfedcb|abcdefgh|gfedcba
	`MeanBorder`: pad with the mean color of the image
`BorderValue` [0.0 0.0 0.0]	Background color used when padding with `BorderType` is `ConstantBorder`

21.5.30 StripeRemoveTransformation

Remove one or several stripe(s) (a group of rows or columns) from 2D data.

Option [default value]	Description
`Axis`	Axis of the stripe (0 = columns, 1 = rows)
`Offset`	Offset of the beginning of the stripe, in number of rows or columns
`Length`	Length of the stripe, in number of rows or columns (a length of 1 means a single row or column will be removed)
`RandomOffset` [0]	If true (1), the stripe offset will be random along the chosen axis
`NbIterations` [1]	Number of stripes to remove
`StepOffset` [`Offset`]	Offset between successive stripes, when `NbIterations` > 1, not taking into account the length of the stripes

21.5.31 ThresholdTransformation

Apply a thresholding transformation to the image. This transformation is also useful for post-processing.

Option [default value]	Description
`Threshold`	Threshold value
`OtsuMethod` [0]	Use Otsu's method to determine the optimal threshold (if true, the `Threshold` value is ignored)
`Operation` [Binary]	Thresholding operation to apply. Can be:
	`Binary`
	`BinaryInverted`
	`Truncate`
	`ToZero`
	`ToZeroInverted`
`MaxValue` [1.0]	Max. value to use with `Binary` and `BinaryInverted` operations

21.5.32 TrimTransformation

Trim the image.

Option [default value]	Description
`NbLevels`	Number of levels for the color discretization of the image
`Method` [`Discretize`]	Possible values are:
	`Reduce`: discretization using K-means
	`Discretize`: simple discretization

21.5.33 WallisFilterTransformation

Apply Wallis filter to the image.

Option [default value]	Description
`Size`	Size of the filter
`Mean` [0.0]	Target mean value
`StdDev` [1.0]	Target standard deviation
`PerChannel` [0]	If true, apply Wallis filter to each channel individually (this parameter is meaningful only if `Size` is 0)

CHAPTER

TWENTYTWO

NETWORK LAYERS

22.1 Layer definition

Common set of parameters for any kind of layer.

Option [default value]	Description
`Input`	Name of the section(s) for the input layer(s). Comma separated
`Type`	Type of the layer. Can be any of the type described below
`Model` [`DefaultModel`]	Layer model to use
`DataType` [`DefaultDataType`]	Layer data type to use. Please note that some layers may not support every data type.
`ConfigSection` []	Name of the configuration section for layer

22.2 Weight fillers

Fillers to initialize weights and biases in the different type of layer.

Usage example:

```
[conv1]
...
WeightsFiller=NormalFiller
WeightsFiller.Mean=0.0
WeightsFiller.StdDev=0.05
...
```

The initial weights distribution for each layer can be checked in the *weights_init* folder, with an example shown in figure [fig:weightsInitDistrib].

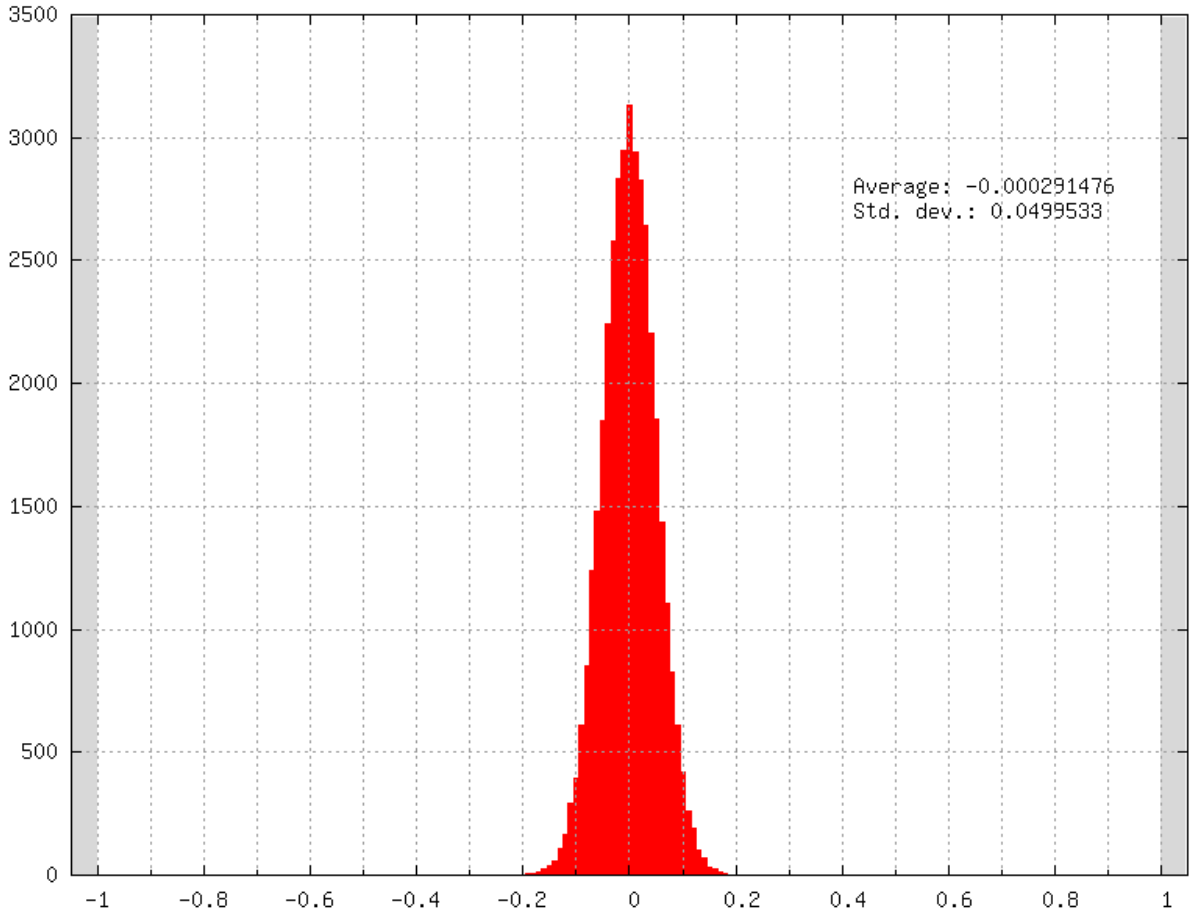


Fig. 1: Initial weights distribution of a layer using a normal distribution (`NormalFiller`) with a 0 mean and a 0.05 standard deviation.

22.2.1 ConstantFiller

Fill with a constant value.

Option	Description
FillerName`.Value`	Value for the filling

22.2.2 HeFiller

Fill with an normal distribution with normalized variance taking into account the rectifier nonlinearity [HZRS15]. This filler is sometimes referred as MSRA filler.

Option [default value]	Description
FillerName`.VarianceNorm` [`FanIn`]	Normalization, can be `FanIn`, `Average` or `FanOut`
FillerName`.Scaling` [1.0]	Scaling factor

Use a normal distribution with standard deviation $\sqrt{\frac{2.0}{n}}$.

- n = $fan\text{-}in$ with `FanIn`, resulting in $Var(W) = \frac{2}{fan\text{-}in}$
- n = $\frac{(fan\text{-}in+fan\text{-}out)}{2}$ with `Average`, resulting in $Var(W) = \frac{4}{fan\text{-}in+fan\text{-}out}$
- n = $fan\text{-}out$ with `FanOut`, resulting in $Var(W) = \frac{2}{fan\text{-}out}$

22.2.3 NormalFiller

Fill with a normal distribution.

Option [default value]	Description
FillerName`.Mean` [0.0]	Mean value of the distribution
FillerName`.StdDev` [1.0]	Standard deviation of the distribution

22.2.4 UniformFiller

Fill with an uniform distribution.

Option [default value]	Description
FillerName`.Min` [0.0]	Min. value
FillerName`.Max` [1.0]	Max. value

22.2.5 XavierFiller

Fill with an uniform distribution with normalized variance [GB10].

Option [default value]	Description
FillerName.`VarianceNorm` [`FanIn`]	Normalization, can be `FanIn`, `Average` or `FanOut`
FillerName.`Distribution` [`Uniform`]	Distribution, can be `Uniform` or `Normal`
FillerName.`Scaling` [1.0]	Scaling factor

Use an uniform distribution with interval $[-scale, scale]$, with $scale = \sqrt{\frac{3.0}{n}}$.

- n = $fan\text{-}in$ with `FanIn`, resulting in $Var(W) = \frac{1}{fan\text{-}in}$
- n = $\frac{(fan\text{-}in+fan\text{-}out)}{2}$ with `Average`, resulting in $Var(W) = \frac{2}{fan\text{-}in+fan\text{-}out}$
- n = $fan\text{-}out$ with `FanOut`, resulting in $Var(W) = \frac{1}{fan\text{-}out}$

22.3 Weight solvers

22.3.1 SGDSolver_Frame

SGD Solver for `Frame` models.

Option [default value]	Description
SolverName.`LearningRate` [0.01]	Learning rate
SolverName.`Momentum` [0.0]	Momentum
SolverName.`Decay` [0.0]	Decay
SolverName.`LearningRatePolicy` [`None`]	Learning rate decay policy. Can be any of `None`, `StepDecay`, `ExponentialDecay`, `InvTDecay`, `PolyDecay`
SolverName.`LearningRateStepSize` [1]	Learning rate step size (in number of stimuli)
SolverName.`LearningRateDecay` [0.1]	Learning rate decay
SolverName.`Clamping` [0]	If true, clamp the weights and bias between -1 and 1
SolverName.`Power` [0.0]	Polynomial learning rule power parameter
SolverName.`MaxIterations` [0.0]	Polynomial learning rule maximum number of iterations

The learning rate decay policies are the following:

- `StepDecay`: every *SolverName*.`LearningRateStepSize` stimuli, the learning rate is reduced by a factor *SolverName*.`LearningRateDecay`;
- `ExponentialDecay`: the learning rate is $\alpha = \alpha_0 \exp(-kt)$, with α_0 the initial learning rate *SolverName*.`LearningRate`, k the rate decay *SolverName* .`LearningRateDecay` and t the step number (one step every *SolverName*.`LearningRateStepSize` stimuli);
- `InvTDecay`: the learning rate is $\alpha = \alpha_0/(1+kt)$, with α_0 the initial learning rate *SolverName*.`LearningRate`, k the rate decay *SolverName*.`LearningRateDecay` and t the step number (one step every *SolverName*.`LearningRateStepSize` stimuli).

- `InvDecay`: the learning rate is $\alpha = \alpha_0 * (1+kt)^{-n}$, with α_0 the initial learning rate *SolverName*`.LearningRate`, k the rate decay *SolverName*`.LearningRateDecay`, t the current iteration and n the power parameter *SolverName*`.Power`
- `PolyDecay`: the learning rate is $\alpha = \alpha_0 * (1-\frac{k}{t})^n$, with α_0 the initial learning rate *SolverName*`.LearningRate`, k the current iteration, t the maximum number of iteration *SolverName*`.MaxIterations` and n the power parameter *SolverName*`.Power`

22.3.2 SGDSolver_Frame_CUDA

SGD Solver for `Frame_CUDA` models.

Option [default value]	Description
SolverName`.LearningRate` [0.01]	Learning rate
SolverName`.Momentum` [0.0]	Momentum
SolverName`.Decay` [0.0]	Decay
SolverName`.LearningRatePolicy` [`None`]	Learning rate decay policy. Can be any of `None`, `StepDecay`, `ExponentialDecay`, `InvTDecay`
SolverName`.LearningRateStepSize` [1]	Learning rate step size (in number of stimuli)
SolverName`.LearningRateDecay` [0.1]	Learning rate decay
SolverName`.Clamping` [0]	If true, clamp the weights and bias between -1 and 1

The learning rate decay policies are identical to the ones in the `SGDSolver_Frame` solver.

22.3.3 AdamSolver_Frame

Adam Solver for `Frame` models [KB14].

Option [default value]	Description
SolverName`.LearningRate` [0.001]	Learning rate (stepsize)
SolverName`.Beta1` [0.9]	Exponential decay rate of these moving average of the first moment
SolverName`.Beta2` [0.999]	Exponential decay rate of these moving average of the second moment
SolverName`.Epsilon` [1.0e-8]	Epsilon

22.3.4 AdamSolver_Frame_CUDA

Adam Solver for `Frame_CUDA` models [KB14].

Option [default value]	Description
SolverName`.LearningRate` [0.001]	Learning rate (stepsize)
SolverName`.Beta1` [0.9]	Exponential decay rate of these moving average of the first moment
SolverName`.Beta2` [0.999]	Exponential decay rate of these moving average of the second moment
SolverName`.Epsilon` [1.0e-8]	Epsilon

22.4 Activation functions

Activation function to be used at the output of layers.

Usage example:

```
[conv1]
...
ActivationFunction=Rectifier
ActivationFunction.LeakSlope=0.01
ActivationFunction.Clipping=20
...
```

22.4.1 Logistic

Logistic activation function.

22.4.2 LogisticWithLoss

Logistic with loss activation function.

22.4.3 Rectifier

Rectifier or ReLU activation function.

Option [default value]	Description
`ActivationFunction.LeakSlope` [0.0]	Leak slope for negative inputs
`ActivationFunction.Clipping` [0.0]	Clipping value for positive outputs

22.4.4 Saturation

Saturation activation function.

22.4.5 Softplus

Softplus activation function.

22.4.6 Tanh

Tanh activation function.

Computes $y = tanh(\alpha x)$.

Option [default value]	Description
`ActivationFunction.Alpha` [1.0]	α parameter

22.4.7 TanhLeCun

Tanh activation function with an α parameter of $1.7159 \times (2.0/3.0)$.

22.5 Anchor

Anchor layer for Faster R-CNN or Single Shot Detector.

Option [default value]	Description
`Input`	This layer takes one or two inputs. The total number of input channels must be `ScoresCls` + 4, with `ScoresCls` being equal to 1 or 2.
`Anchor[*]`	Anchors definition. For each anchor, there must be two space-separated values: the root area and the aspect ratio.
`ScoresCls`	Number of classes per anchor. Must be 1 (if the scores input uses logistic regression) or 2 (if the scores input is a two-class softmax layer)
`FeatureMapWidth` [`StimuliProvider.Width`]	Reference width use to scale anchors coordinate.
`FeatureMapHeight` [`StimuliProvider.Height`]	Reference height use to scale anchors coordinate.

22.5.1 Configuration parameters (*Frame* models)

Option [default value]	Model(s)	Description
`PositiveIoU` [0.7]	*all Frame*	Assign a positive label for anchors whose IoU overlap is higher than `PositiveIoU` with any ground-truth box
`NegativeIoU` [0.3]	*all Frame*	Assign a negative label for non-positive anchors whose IoU overlap is lower than `NegativeIoU` for all ground-truth boxes
`LossLambda` [10.0]	*all Frame*	Balancing parameter λ
`LossPositiveSample` [128]	*all Frame*	Number of random positive samples for the loss computation
`LossNegativeSample` [128]	*all Frame*	Number of random negative samples for the loss computation

Usage example:

```
; RPN network: cls layer
[scores]
Input=...
Type=Conv
KernelWidth=1
KernelHeight=1
; 18 channels for 9 anchors
NbOutputs=18
...

[scores.softmax]
Input=scores
```

(continues on next page)

(continued from previous page)

```
Type=Softmax
NbOutputs=[scores]NbOutputs
WithLoss=1

; RPN network: coordinates layer
[coordinates]
Input=...
Type=Conv
KernelWidth=1
KernelHeight=1
; 36 channels for 4 coordinates x 9 anchors
NbOutputs=36
...

; RPN network: anchors
[anchors]
Input=scores.softmax,coordinates
Type=Anchor
ScoresCls=2 ; using a two-class softmax for the scores
Anchor[0]=32 1.0
Anchor[1]=48 1.0
Anchor[2]=64 1.0
Anchor[3]=80 1.0
Anchor[4]=96 1.0
Anchor[5]=112 1.0
Anchor[6]=128 1.0
Anchor[7]=144 1.0
Anchor[8]=160 1.0
ConfigSection=anchors.config

[anchors.config]
PositiveIoU=0.7
NegativeIoU=0.3
LossLambda=1.0
```

22.5.2 Outputs remapping

Outputs remapping allows to convert *scores* and *coordinates* output feature maps layout from another ordering that the one used in the N2D2 `Anchor` layer, during weights import/export.

For example, lets consider that the imported weights corresponds to the following output feature maps ordering:

```
0 anchor[0].y
1 anchor[0].x
2 anchor[0].h
3 anchor[0].w
4 anchor[1].y
5 anchor[1].x
6 anchor[1].h
7 anchor[1].w
8 anchor[2].y
```

(continues on next page)

(continued from previous page)

```
9 anchor[2].x
10 anchor[2].h
11 anchor[2].w
```

The output feature maps ordering required by the `Anchor` layer is:

```
0 anchor[0].x
1 anchor[1].x
2 anchor[2].x
3 anchor[0].y
4 anchor[1].y
5 anchor[2].y
6 anchor[0].w
7 anchor[1].w
8 anchor[2].w
9 anchor[0].h
10 anchor[1].h
11 anchor[2].h
```

The feature maps ordering can be changed during weights import/export:

```
; RPN network: coordinates layer
[coordinates]
Input=...
Type=Conv
KernelWidth=1
KernelHeight=1
; 36 channels for 4 coordinates x 9 anchors
NbOutputs=36
...
ConfigSection=coordinates.config

[coordinates.config]
WeightsExportFormat=HWCO ; Weights format used by TensorFlow
OutputsRemap=1:4,0:4,3:4,2:4
```

22.6 BatchNorm

Batch Normalization layer [IS15].

Option [default value]	Description
`NbOutputs`	Number of output neurons
`ActivationFunction` []	Activation function. Can be any of `Logistic`, `LogisticWithLoss`, `Rectifier`, `Softplus`, `TanhLeCun`, `Linear`, `Saturation` or `Tanh` (none by default)
`ScalesSharing` []	Share the scales with an other layer
`BiasesSharing` []	Share the biases with an other layer
`MeansSharing` []	Share the means with an other layer
`VariancesSharing` []	Share the variances with an other layer

22.6.1 Configuration parameters (*Frame* models)

Option [default value]	Model(s)	Description
`Solvers.*`	*all Frame*	Any solver parameters
`ScaleSolver.*`	*all Frame*	Scale solver parameters, take precedence over the `Solvers.*` parameters
`BiasSolver.*`	*all Frame*	Bias solver parameters, take precedence over the `Solvers.*` parameters
`Epsilon` [0.0]	*all Frame*	Epsilon value used in the batch normalization formula. If 0.0, automatically choose the minimum possible value.
`MovingAverageMomentum` [0.1]	*all Frame*	MovingAverageMomentum: used for the moving average of batch-wise means and standard deviations during training. The closer to 1.0, the more it will depend on the last batch.

22.7 Conv

Convolutional layer.

Option [default value]	Description
`KernelWidth`	Width of the kernels
`KernelHeight`	Height of the kernels
`KernelDepth` []	Depth of the kernels (implies 3D kernels)
`KernelSize` []	Kernels size (implies 2D square kernels)
`KernelDims` []	List of space-separated dimensions for N-D kernels
`NbOutputs`	Number of output channels
`SubSampleX` [1]	X-axis subsampling factor of the output feature maps
`SubSampleY` [1]	Y-axis subsampling factor of the output feature maps
`SubSampleZ` []	Z-axis subsampling factor of the output feature maps
`SubSample` [1]	Subsampling factor of the output feature maps
`SubSampleDims` []	List of space-separated subsampling dimensions for N-D kernels
`StrideX` [1]	X-axis stride of the kernels

Table 1 – continued from previous page

Option [default value]	Description
`StrideY` [1]	Y-axis stride of the kernels
`StrideZ` []	Z-axis stride of the kernels
`Stride` [1]	Stride of the kernels
`StrideDims` []	List of space-separated stride dimensions for N-D kernels
`PaddingX` [0]	X-axis input padding
`PaddingY` [0]	Y-axis input padding
`PaddingZ` []	Z-axis input padding
`Padding` [0]	Input padding
`PaddingDims` []	List of space-separated padding dimensions for N-D kernels
`DilationX` [1]	X-axis dilation of the kernels
`DilationY` [1]	Y-axis dilation of the kernels
`DilationZ` []	Z-axis dilation of the kernels
`Dilation` [1]	Dilation of the kernels
`DilationDims` []	List of space-separated dilation dimensions for N-D kernels
`ActivationFunction` []	Activation function. Can be any of `Logistic`, `LogisticWithLoss`, `Rectifier`
`WeightsFiller` [`NormalFiller(0.0, 0.05)`]	Weights initial values filler
`BiasFiller` [`NormalFiller(0.0, 0.05)`]	Biases initial values filler
`Mapping.NbGroups` []	Mapping: number of groups (mutually exclusive with all other Mapping.* options
`Mapping.ChannelsPerGroup` []	Mapping: number of channels per group (mutually exclusive with all other Mappi
`Mapping.SizeX` [1]	Mapping canvas pattern default width
`Mapping.SizeY` [1]	Mapping canvas pattern default height
`Mapping.Size` [1]	Mapping canvas pattern default size (mutually exclusive with `Mapping.SizeX` ar
`Mapping.StrideX` [1]	Mapping canvas default X-axis step
`Mapping.StrideY` [1]	Mapping canvas default Y-axis step
`Mapping.Stride` [1]	Mapping canvas default step (mutually exclusive with``Mapping.StrideX`` and Ma
`Mapping.OffsetX` [0]	Mapping canvas default X-axis offset
`Mapping.OffsetY` [0]	Mapping canvas default Y-axis offset
`Mapping.Offset` [0]	Mapping canvas default offset (mutually exclusive with `Mapping.OffsetX` and M
`Mapping.NbIterations` [0]	Mapping canvas pattern default number of iterations (0 means no limit)
`Mapping(in).SizeX` [1]	Mapping canvas pattern default width for input layer `in`
`Mapping(in).SizeY` [1]	Mapping canvas pattern default height for input layer `in`
`Mapping(in).Size` [1]	Mapping canvas pattern default size for input layer `in` (mutually exclusive with Ma
`Mapping(in).StrideX` [1]	Mapping canvas default X-axis step for input layer `in`
`Mapping(in).StrideY` [1]	Mapping canvas default Y-axis step for input layer `in`
`Mapping(in).Stride` [1]	Mapping canvas default step for input layer `in` (mutually exclusive with `Mapping`
`Mapping(in).OffsetX` [0]	Mapping canvas default X-axis offset for input layer `in`
`Mapping(in).OffsetY` [0]	Mapping canvas default Y-axis offset for input layer `in`
`Mapping(in).Offset` [0]	Mapping canvas default offset for input layer `in` (mutually exclusive with `Mappin`
`Mapping(in).NbIterations` [0]	Mapping canvas pattern default number of iterations for input layer `in` (0 means n
`WeightsSharing` []	Share the weights with an other layer
`BiasesSharing` []	Share the biases with an other layer

22.7.1 Configuration parameters (*Frame* models)

Option [default value]	Model(s)	Description
`NoBias` [0]	*all Frame*	If true, don't use bias
`Solvers.*`	*all Frame*	Any solver parameters
`WeightsSolver.*`	*all Frame*	Weights solver parameters, take precedence over the `Solvers.*` parameters
`BiasSolver.*`	*all Frame*	Bias solver parameters, take precedence over the `Solvers.*` parameters
`WeightsExportFormat` [`OCHW`]	*all Frame*	Weights import/export format. Can be `OCHW` or `OCHW`, with `O` the output feature map, `C` the input feature map (channel), `H` the kernel row and `W` the kernel column, in the order of the outermost dimension (in the leftmost position) to the innermost dimension (in the rightmost position)
`WeightsExportFlip` [0]	*all Frame*	If true, import/export flipped kernels

22.7.2 Configuration parameters (*Spike* models)

Experimental option (implementation may be wrong or susceptible to change)

Option [default value]	Model(s)	Description
IncomingDelay [1 TimePs;100 TimeFs]	*all Spike*	Synaptic incoming delay w_{delay}
Threshold [1.0]	Spike, Spike_RRAM	Threshold of the neuron I_{thres}
BipolarThreshold [1]	Spike, Spike_RRAM	If true, the threshold is also applied to the absolute value of negative values (generating negative spikes)
Leak [0.0]	Spike, Spike_RRAM	Neural leak time constant τ_{leak} (if 0, no leak)
Refractory [0.0]	Spike, Spike_RRAM	Neural refractory period T_{refrac}
WeightsRelInit [0.0;0.05]	Spike	Relative initial synaptic weight w_{init}
WeightsMinMean [1;0.1]	Spike_RRAM	Mean minimum synaptic weight w_{min}
WeightsMaxMean [100;10.0]	Spike_RRAM	Mean maximum synaptic weight w_{max}
WeightsMinVarSlope [0.0]	Spike_RRAM	OXRAM specific parameter
WeightsMinVarOrigin [0.0]	Spike_RRAM	OXRAM specific parameter
WeightsMaxVarSlope [0.0]	Spike_RRAM	OXRAM specific parameter
WeightsMaxVarOrigin [0.0]	Spike_RRAM	OXRAM specific parameter
WeightsSetProba [1.0]	Spike_RRAM	Intrinsic SET switching probability P_{SET} (upon receiving a SET programming pulse). Assuming uniform statistical distribution (not well supported by experiments on RRAM)
WeightsResetProba [1.0]	Spike_RRAM	Intrinsic RESET switching probability P_{RESET} (upon receiving a RESET programming pulse). Assuming uniform statistical distribution (not well supported by experiments on RRAM)
SynapticRedundancy [1]	Spike_RRAM	Synaptic redundancy (number of RRAM device per synapse)
BipolarWeights [0]	Spike_RRAM	Bipolar weights
BipolarIntegration [0]	Spike_RRAM	Bipolar integration
LtpProba [0.2]	Spike_RRAM	Extrinsic STDP LTP probability (cumulative with intrinsic SET switching probability P_{SET})
LtdProba [0.1]	Spike_RRAM	Extrinsic STDP LTD probability (cumulative with intrinsic RESET switching probability P_{RESET})
StdpLtp [1000 TimePs]	Spike_RRAM	STDP LTP time window T_{LTP}
InhibitRefractory [0 TimePs]	Spike_RRAM	Neural lateral inhibition period $T_{inhibit}$
EnableStdp [1]	Spike_RRAM	If false, STDP is disabled (no synaptic weight change)
RefractoryIntegration [1]	Spike_RRAM	If true, reset the integration to 0 during the refractory period
DigitalIntegration [0]	Spike_RRAM	If false, the analog value of the devices is integrated, instead of their binary value

22.8 Deconv

Deconvolution layer.

Option [default value]	Description
`KernelWidth`	Width of the kernels
`KernelHeight`	Height of the kernels
`KernelDepth` []	Depth of the kernels (implies 3D kernels)
`KernelSize` []	Kernels size (implies 2D square kernels)
`KernelDims` []	List of space-separated dimensions for N-D kernels
`NbOutputs`	Number of output channels
`SubSampleX` [1]	X-axis subsampling factor of the output feature maps
`SubSampleY` [1]	Y-axis subsampling factor of the output feature maps
`SubSampleZ` []	Z-axis subsampling factor of the output feature maps
`SubSample` [1]	Subsampling factor of the output feature maps
`SubSampleDims` []	List of space-separated subsampling dimensions for N-D kernels
`StrideX` [1]	X-axis stride of the kernels
`StrideY` [1]	Y-axis stride of the kernels
`StrideZ` []	Z-axis stride of the kernels
`Stride` [1]	Stride of the kernels
`StrideDims` []	List of space-separated stride dimensions for N-D kernels
`PaddingX` [0]	X-axis input padding
`PaddingY` [0]	Y-axis input padding
`PaddingZ` []	Z-axis input padding
`Padding` [0]	Input padding
`PaddingDims` []	List of space-separated padding dimensions for N-D kernels
`DilationX` [1]	X-axis dilation of the kernels
`DilationY` [1]	Y-axis dilation of the kernels
`DilationZ` []	Z-axis dilation of the kernels
`Dilation` [1]	Dilation of the kernels
`DilationDims` []	List of space-separated dilation dimensions for N-D kernels
`ActivationFunction` []	Activation function. Can be any of `Logistic`, `LogisticWithLoss`, `Rectifier`
`WeightsFiller` [`NormalFiller(0.0, 0.05)`]	Weights initial values filler
`BiasFiller` [`NormalFiller(0.0, 0.05)`]	Biases initial values filler
`Mapping.NbGroups` []	Mapping: number of groups (mutually exclusive with all other Mapping.* options
`Mapping.ChannelsPerGroup` []	Mapping: number of channels per group (mutually exclusive with all other Mappi
`Mapping.SizeX` [1]	Mapping canvas pattern default width
`Mapping.SizeY` [1]	Mapping canvas pattern default height
`Mapping.Size` [1]	Mapping canvas pattern default size (mutually exclusive with `Mapping.SizeX` ar
`Mapping.StrideX` [1]	Mapping canvas default X-axis step
`Mapping.StrideY` [1]	Mapping canvas default Y-axis step
`Mapping.Stride` [1]	Mapping canvas default step (mutually exclusive with``Mapping.StrideX`` and Ma
`Mapping.OffsetX` [0]	Mapping canvas default X-axis offset
`Mapping.OffsetY` [0]	Mapping canvas default Y-axis offset
`Mapping.Offset` [0]	Mapping canvas default offset (mutually exclusive with `Mapping.OffsetX` and M
`Mapping.NbIterations` [0]	Mapping canvas pattern default number of iterations (0 means no limit)
`Mapping(in).SizeX` [1]	Mapping canvas pattern default width for input layer `in`
`Mapping(in).SizeY` [1]	Mapping canvas pattern default height for input layer `in`
`Mapping(in).Size` [1]	Mapping canvas pattern default size for input layer `in` (mutually exclusive with Ma
`Mapping(in).StrideX` [1]	Mapping canvas default X-axis step for input layer `in`

Table 2 – continued from previous page

Option [default value]	Description
`Mapping(in).StrideY` [1]	Mapping canvas default Y-axis step for input layer `in`
`Mapping(in).Stride` [1]	Mapping canvas default step for input layer `in` (mutually exclusive with `Mapping`
`Mapping(in).OffsetX` [0]	Mapping canvas default X-axis offset for input layer `in`
`Mapping(in).OffsetY` [0]	Mapping canvas default Y-axis offset for input layer `in`
`Mapping(in).Offset` [0]	Mapping canvas default offset for input layer `in` (mutually exclusive with `Mappin`
`Mapping(in).NbIterations` [0]	Mapping canvas pattern default number of iterations for input layer `in` (0 means r
`WeightsSharing` []	Share the weights with an other layer
`BiasesSharing` []	Share the biases with an other layer

22.8.1 Configuration parameters (*Frame* models)

Option [default value]	Model(s)	Description
`NoBias` [0]	*all Frame*	If true, don't use bias
`BackPropagate` [1]	*all Frame*	If true, enable backpropagation
`Solvers.*`	*all Frame*	Any solver parameters
`WeightsSolver.*`	*all Frame*	Weights solver parameters, take precedence over the `Solvers.*` parameters
`BiasSolver.*`	*all Frame*	Bias solver parameters, take precedence over the `Solvers.*` parameters
`WeightsExportFormat` [`OCHW`]	*all Frame*	Weights import/export format. Can be `OCHW` or `OCHW`, with `O` the output feature map, `C` the input feature map (channel), `H` the kernel row and `W` the kernel column, in the order of the outermost dimension (in the leftmost position) to the innermost dimension (in the rightmost position)
`WeightsExportFlip` [0]	*all Frame*	If true, import/export flipped kernels

22.9 Dropout

Dropout layer [SHK+12].

Option [default value]	Description
`NbOutputs`	Number of output neurons

22.9.1 Configuration parameters (*Frame* models)

Option [default value]	Model(s)	Description
`Dropout` [0.5]	*all Frame*	The probability with which the value from input would be dropped

22.10 ElemWise

Element-wise operation layer.

Option [default value]	Description
`NbOutputs`	Number of output neurons
`Operation` [`Sum`]	Type of operation (`Sum`, `AbsSum`, `EuclideanSum`, `Prod`, or `Max`)
`Weights` [1.0]	Weights for the `Sum`, `AbsSum`, and `EuclideanSum` operation, in the same order as the inputs
`Shifts` [0.0]	Shifts for the `Sum` and `EuclideanSum` operation, in the same order as the inputs
`ActivationFunction` []	Activation function. Can be any of `Logistic`, `LogisticWithLoss`, `Rectifier`, `Softplus`, `TanhLeCun`, `Linear`, `Saturation` or `Tanh` (none by default)

Given N input tensors T_i, performs the following operation:

22.10.1 Sum operation

$T_{out} = \sum_1^N (w_i T_i + s_i)$

22.10.2 AbsSum operation

$T_{out} = \sum_1^N (w_i |T_i|)$

22.10.3 EuclideanSum operation

$T_{out} = \sqrt{\sum_1^N (w_i T_i + s_i)^2}$

22.10.4 Prod operation

$T_{out} = \prod_1^N (T_i)$

22.10.5 Max operation

$T_{out} = MAX_1^N(T_i)$

22.10.6 Examples

Sum of two inputs ($T_{out} = T_1 + T_2$):

```
[elemwise_sum]
Input=layer1,layer2
Type=ElemWise
NbOutputs=[layer1]NbOutputs
Operation=Sum
```

Weighted sum of two inputs, by a factor 0.5 for `layer1` and 1.0 for `layer2` ($T_{out} = 0.5 \times T_1 + 1.0 \times T_2$):

```
[elemwise_weighted_sum]
Input=layer1,layer2
Type=ElemWise
NbOutputs=[layer1]NbOutputs
Operation=Sum
Weights=0.5 1.0
```

Single input scaling by a factor 0.5 and shifted by 0.1 ($T_{out} = 0.5 \times T_1$ + 0.1):

```
[elemwise_scale]
Input=layer1
Type=ElemWise
NbOutputs=[layer1]NbOutputs
Operation=Sum
Weights=0.5
Shifts=0.1
```

Absolute value of an input ($T_{out} = |T_1|$):

```
[elemwise_abs]
Input=layer1
Type=ElemWise
NbOutputs=[layer1]NbOutputs
Operation=Abs
```

22.11 FMP

Fractional max pooling layer [Gra14].

Option [default value]	Description
`NbOutputs`	Number of output channels
`ScalingRatio`	Scaling ratio. The output size is $round\left(\frac{\text{input size}}{\text{scaling ratio}}\right)$.
`ActivationFunction` []	Activation function. Can be any of `Logistic`, `LogisticWithLoss`, `Rectifier`, `Softplus`, `TanhLeCun`, `Linear`, `Saturation` or `Tanh` (none by default)

22.11.1 Configuration parameters (*Frame* models)

Option [default value]	Model(s)	Description
`Overlapping` [1]	*all Frame*	If true, use overlapping regions, else use disjoint regions
`PseudoRandom` [1]	*all Frame*	If true, use pseudorandom sequences, else use random sequences

22.12 Fc

Fully connected layer.

Option [default value]	Description
`NbOutputs`	Number of output neurons
`WeightsFiller`	Weights initial values filler
`[NormalFiller(0.0, 0.05)]`	
`BiasFiller`	Biases initial values filler
`[NormalFiller(0.0, 0.05)]`	
`ActivationFunction` []	Activation function. Can be any of `Logistic`, `LogisticWithLoss`, `Rectifier`, `Softplus`, `TanhLeCun`, `Linear`, `Saturation` or `Tanh` (none by default)

22.12.1 Configuration parameters (*Frame* models)

Option [default value]	Model(s)	Description
`NoBias` [0]	*all Frame*	If true, don't use bias
`BackPropagate` [1]	*all Frame*	If true, enable backpropagation
`Solvers.*`	*all Frame*	Any solver parameters
`WeightsSolver.*`	*all Frame*	Weights solver parameters, take precedence over the `Solvers.*` parameters
`BiasSolver.*`	*all Frame*	Bias solver parameters, take precedence over the `Solvers.*` parameters
`DropConnect` [1.0]	`Frame`	If below 1.0, fraction of synapses that are disabled with drop connect

22.12.2 Configuration parameters (*Spike* models)

Option [default value]	Model(s)	Description
`IncomingDelay` [1 `TimePs`;100 `TimeFs`]	*all Spike*	Synaptic incoming delay w_{delay}
`Threshold` [1.0]	`Spike`, `Spike_RRAM`	Threshold of the neuron I_{thres}
`BipolarThreshold` [1]	`Spike`, `Spike_RRAM`	If true, the threshold is also applied to the absolute value of negative values (generating negative spikes)
`Leak` [0.0]	`Spike`, `Spike_RRAM`	Neural leak time constant τ_{leak} (if 0, no leak)
`Refractory` [0.0]	`Spike`, `Spike_RRAM`	Neural refractory period T_{refrac}
`TerminateDelta` [0]	`Spike`, `Spike_RRAM`	Terminate delta
`WeightsRelInit` [0.0;0.05]	`Spike`	Relative initial synaptic weight w_{init}
`WeightsMinMean` [1;0.1]	`Spike_RRAM`	Mean minimum synaptic weight w_{min}
`WeightsMaxMean` [100;10.0]	`Spike_RRAM`	Mean maximum synaptic weight w_{max}
`WeightsMinVarSlope` [0.0]	`Spike_RRAM`	OXRAM specific parameter
`WeightsMinVarOrigin` [0.0]	`Spike_RRAM`	OXRAM specific parameter
`WeightsMaxVarSlope` [0.0]	`Spike_RRAM`	OXRAM specific parameter
`WeightsMaxVarOrigin` [0.0]	`Spike_RRAM`	OXRAM specific parameter
`WeightsSetProba` [1.0]	`Spike_RRAM`	Intrinsic SET switching probability P_{SET} (upon receiving a SET programming pulse). Assuming uniform statistical distribution (not well supported by experiments on RRAM)
`WeightsResetProba` [1.0]	`Spike_RRAM`	Intrinsic RESET switching probability P_{RESET} (upon receiving a RESET programming pulse). Assuming uniform statistical distribution (not well supported by experiments on RRAM)
`SynapticRedundancy` [1]	`Spike_RRAM`	Synaptic redundancy (number of RRAM device per synapse)
`BipolarWeights` [0]	`Spike_RRAM`	Bipolar weights
`BipolarIntegration` [0]	`Spike_RRAM`	Bipolar integration
`LtpProba` [0.2]	`Spike_RRAM`	Extrinsic STDP LTP probability (cumulative with intrinsic SET switching probability P_{SET})
`LtdProba` [0.1]	`Spike_RRAM`	Extrinsic STDP LTD probability (cumulative with intrinsic RESET switching probability P_{RESET})
`StdpLtp` [1000 `TimePs`]	`Spike_RRAM`	STDP LTP time window T_{LTP}
`InhibitRefractory` [0 `TimePs`]	`Spike_RRAM`	Neural lateral inhibition period $T_{inhibit}$
`EnableStdp` [1]	`Spike_RRAM`	If false, STDP is disabled (no synaptic weight change)
`RefractoryIntegration` [1]	`Spike_RRAM`	If true, reset the integration to 0 during the refractory period
`DigitalIntegration` [0]	`Spike_RRAM`	If false, the analog value of the devices is integrated, instead of their binary value

22.13 LRN

Local Response Normalization (LRN) layer.

Option [default value]	Description
`NbOutputs`	Number of output neurons

The response-normalized activity $b^i_{x,y}$ is given by the expression:

$$b^i_{x,y} = \frac{a^i_{x,y}}{\left(k + \alpha \sum\limits_{j=max(0,i-n/2)}^{min(N-1,i+n/2)} \left(a^j_{x,y}\right)^2\right)^\beta}$$

22.13.1 Configuration parameters (*Frame* models)

Option [default value]	Model(s)	Description
`N` [5]	*all Frame*	Normalization window width in elements
`Alpha` [1.0e-4]	*all Frame*	Value of the alpha variance scaling parameter in the normalization formula
`Beta` [0.75]	*all Frame*	Value of the beta power parameter in the normalization formula
`K` [2.0]	*all Frame*	Value of the k parameter in normalization formula

22.14 LSTM

Long Short Term Memory Layer [HS97].

22.14.1 Global layer parameters (*Frame_CUDA* models)

Option [default value]	Description
`SeqLength`	Maximum sequence length that the LSTM can take as an input.
`BatchSize`	Number of sequences used for a single weights actualisation process : size of the batch.
`InputDim`	Dimension of every element composing a sequence.
`HiddenSize`	Dimension of the LSTM inner state and output.
`SingleBackpropFeeding` [1]	If disabled return the full output sequence.
`Bidirectional` [0]	If enabled, build a bidirectional structure.
`AllGatesWeightsFiller`	All Gates weights initial values filler.
`AllGatesBiasFiller`	All Gates bias initial values filler.
`WeightsInputGateFiller`	Input gate previous layer and recurrent weights initial values filler. Take precedence over AllGatesWeightsFiller parameter.
`WeightsForgetGateFiller`	Forget gate previous layer and recurrent weights initial values filler. Take precedence over AllGatesWeightsFiller parameter.
`WeightsCellGateFiller`	Cell gate (or new memory) previous layer and recurrent weights initial values filler. Take precedence over AllGatesWeightsFiller parameter.
`WeightsOutputGateFiller`	Output gate previous layer and recurrent weights initial values filler. Take precedence over AllGatesWeightsFiller parameter.
`BiasInputGateFiller`	Input gate previous layer and recurrent bias initial values filler. Take precedence over AllGatesBiasFiller parameter.
`BiasRecurrentForgetGateFiller`	Forget gate recurrent bias initial values filler. Take precedence over AllGatesBiasFiller parameter. Often set to 1.0 to show better convergence performance.
`BiasPreviousLayerForgetGateFiller`	Forget gate previous layer bias initial values filler. Take precedence over AllGatesBiasFiller parameter.
`BiasCellGateFiller`	Cell gate (or new memory) previous layer and recurrent bias initial values filler. Take precedence over AllGatesBiasFiller parameter.
`BiasOutputGateFiller`	Output gate previous layer and recurrent bias initial values filler. Take precedence over AllGatesBiasFiller parameter.
`HxFiller`	Recurrent previous state initialisation. Often set to 0.0
`CxFiller`	Recurrent previous LSTM inner state initialisation. Often set to 0.0

22.14.2 Configuration parameters (*Frame_CUDA* models)

Option [default value]	Model(s)	Description
`Solvers.*`	*all Frame*	Any solver parameters
`Dropout` [0.0]	*all Frame*	The probability with which the value from input would be dropped.
`InputMode` []	*all Frame*	If enabled, drop the matrix multiplication of the input data.
`Algo` [0]	*all Frame*	Allow to choose different cuDNN implementation. Can be 0 : STANDARD, 1 : STATIC, 2 : DYNAMIC. Case 1 and 2 aren't supported yet.

22.14.3 Current restrictions

- Only Frame_Cuda version is supported yet.
- The implementation only support input sequences with a fixed length associated with a single label.
- CuDNN structures requires the input data to be ordered as [1, InputDim, BatchSize, SeqLength]. Depending on the use case (like sequential-MNIST), the input data would need to be shuffled between the stimuli provisder and the RNN in order to process batches of data. No shuffling layer is yet operational. In that case, set batch to one for first experiments.

22.14.4 Further development requirements

When it comes to RNN, two main factors needs to be considered to build proper interfaces :

1. Whether the input data has a variable or a fixed length over the data base, that is to say whether the input data will have a variable or fixed Sequence length. Of course the main strength of a RNN is to process variable length data.
2. Labelling granularity of the input data, that is to say wheteher every elements of a sequence is labelled or the sequence itself has only one label.

For instance, let's consider sentences as sequences of words in which every word would be part of a vocabulary. Sentences could have a variable length and every element/word would have a label. In that case, every relevant element of the output sequence from the recurrent structure is turned into a prediction throught a fully connected layer with a linear activation fonction and a softmax.

On the opposite, using sequential-MNIST database, the sequence length would be the same regarding every image and there is only one label for an image. In that case, the last element of the output sequence is the most relevant one to be turned into a prediction as it carries the information of the entire input sequence.

To provide flexibility according to these factors, the first implementation choice is to set a maximum sequence length emphSeqLength as an hyperparameter that the User provide. Variable length senquences can be processed by padding the remaining steps of the input sequence.

Then two cases occur as the labeling granularity is scaled at each element of the sequence or scaled at the sequence itself:

1. The sequence itself has only one label :

   The model has a fixed size with one fully connected mapped to the relevant element of the output sequence according to the input sequence.

2. Every elements of a sequence is labelled :

The model has a fixed size with one big fully connected (or Tmax fully connected) mapped to the relevant elements of the output sequence according to the input sequence. The remaining elements need to be masked so it doesn't influence longer sequences.

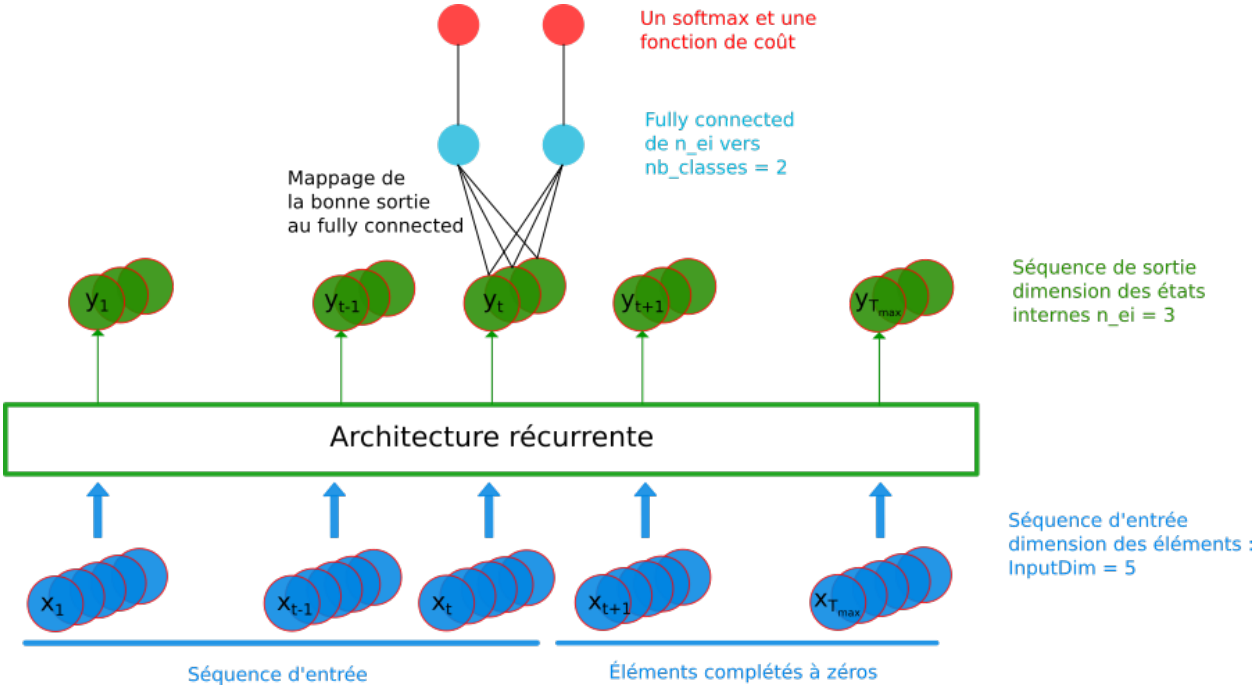


Fig. 2: RNN model : variable sequence length and labeling scaled at the sequence

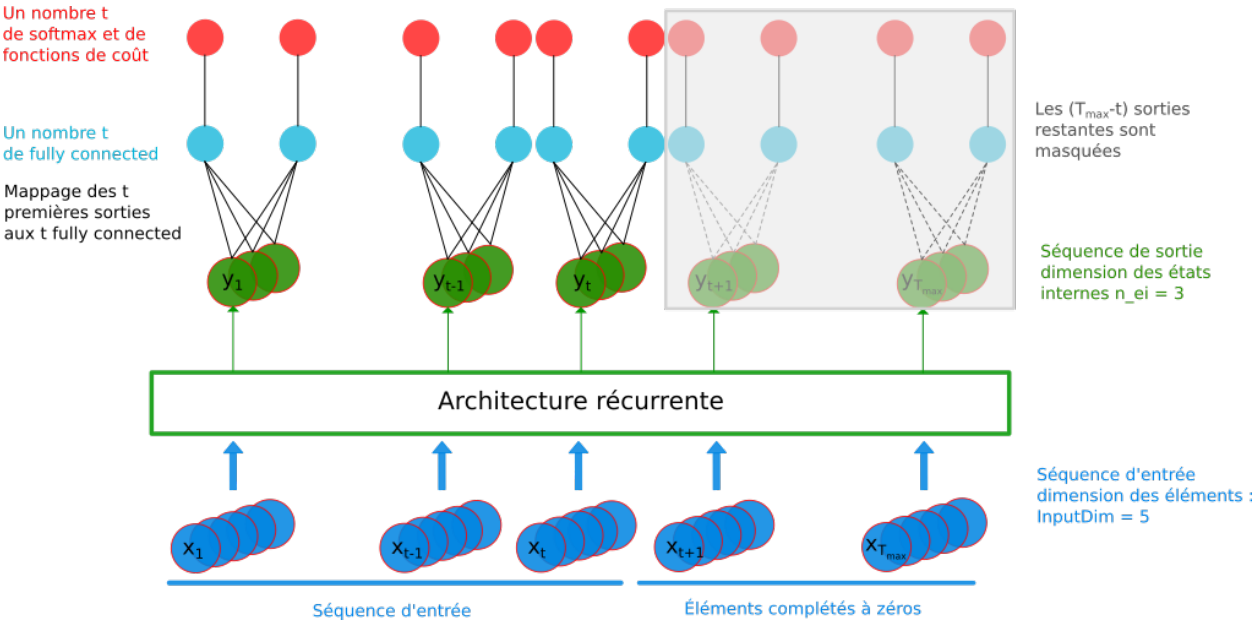


Fig. 3: RNN model : variable sequence length and labeling scaled at each element of the sequence

22.14.5 Development guidance

- Replace the inner local variables of LSTMCell_Frame_Cuda with a generic layer of shuffling (on device) to enable the the process of data batch.
- Develop some kind of label embedding within the layer to better articulate the labeling granularity of the input data.
- Adapt structures to support the STATIC and DYNAMIC algorithm of cuDNN functions.

22.15 Normalize

Normalize layer.

Option [default value]	Description
`NbOutputs`	Number of output feature maps
`Norm`	Norm to be used. Can be, `L1` or `L2`

22.16 Padding

Padding layer.

Option [default value]	Description
`NbOutputs`	Number of output neurons
`TopPadding`	Size of the top padding (positive or negative)
`BottomPadding`	Size of the bottom padding (positive or negative)
`LeftPadding`	Size of the left padding (positive or negative)
`RightPadding`	Size of the right padding (positive or negative)

The padding layer allow to insert asymmetric padding for each layer axes.

22.17 Pool

Pooling layer.

There are two CUDA models for this cell:

`Frame_CUDA`, which uses CuDNN as back-end and only supports one-to-one input to output map connection;

`Frame_EXT_CUDA`, which uses custom CUDA kernels and allows arbitrary connections between input and output maps (and can therefore be used to implement Maxout or both Maxout and Pooling simultaneously).

22.17.1 Maxout example

In the following INI section, one implements a Maxout between each consecutive pair of 8 input maps:

```
[maxout_layer]
Input=...
Type=Pool
Model=Frame_EXT_CUDA
PoolWidth=1
PoolHeight=1
NbOutputs=4
Pooling=Max
Mapping.SizeY=2
Mapping.StrideY=2
```

The layer connectivity is the following:

# input map	1	X			
	2	X			
	3		X		
	4		X		
	5			X	
	6			X	
	7				X
	8				X
		1	2	3	4
		# output map			

Option [default value]	Description
`Pooling` [`Max`]	Type of pooling (`Max` or `Average`)
`PoolWidth`	Width of the pooling area
`PoolHeight`	Height of the pooling area
`PoolDepth` []	Depth of the pooling area (implies 3D pooling area)
`PoolSize` []	Pooling area size (implies 2D square pooling area)
`PoolDims` []	List of space-separated dimensions for N-D pooling area
`NbOutputs`	Number of output channels
`StrideX` [1]	X-axis stride of the pooling area
`StrideY` [1]	Y-axis stride of the pooling area
`StrideZ` []	Z-axis stride of the pooling area
`Stride` [1]	Stride of the pooling area
`StrideDims` []	List of space-separated stride dimensions for N-D pooling area
`PaddingX` [0]	X-axis input padding
`PaddingY` [0]	Y-axis input padding
`PaddingZ` []	Z-axis input padding
`Padding` [0]	Input padding
`PaddingDims` []	List of space-separated padding dimensions for N-D pooling area
`ActivationFunction` []	Activation function. Can be any of `Logistic`, `LogisticWithLoss`, `Rectifier`, `Softplus`, `T`
`Mapping.NbGroups` []	Mapping: number of groups (mutually exclusive with all other Mapping.* options)
`Mapping.ChannelsPerGroup` []	Mapping: number of channels per group (mutually exclusive with all other Mapping.* options)
`Mapping.SizeX` [1]	Mapping canvas pattern default width
`Mapping.SizeY` [1]	Mapping canvas pattern default height

Table 3 – continued from previous page

Option [default value]	Description
`Mapping.Size` [1]	Mapping canvas pattern default size (mutually exclusive with `Mapping.SizeX` and `Mapping.S`
`Mapping.StrideX` [1]	Mapping canvas default X-axis step
`Mapping.StrideY` [1]	Mapping canvas default Y-axis step
`Mapping.Stride` [1]	Mapping canvas default step (mutually exclusive with``Mapping.StrideX`` and `Mapping.Strid`
`Mapping.OffsetX` [0]	Mapping canvas default X-axis offset
`Mapping.OffsetY` [0]	Mapping canvas default Y-axis offset
`Mapping.Offset` [0]	Mapping canvas default offset (mutually exclusive with `Mapping.OffsetX` and `Mapping.Offs`
`Mapping.NbIterations` [0]	Mapping canvas pattern default number of iterations (0 means no limit)
`Mapping(in).SizeX` [1]	Mapping canvas pattern default width for input layer `in`
`Mapping(in).SizeY` [1]	Mapping canvas pattern default height for input layer `in`
`Mapping(in).Size` [1]	Mapping canvas pattern default size for input layer `in` (mutually exclusive with `Mapping(in).S`
`Mapping(in).StrideX` [1]	Mapping canvas default X-axis step for input layer `in`
`Mapping(in).StrideY` [1]	Mapping canvas default Y-axis step for input layer `in`
`Mapping(in).Stride` [1]	Mapping canvas default step for input layer `in` (mutually exclusive with `Mapping(in).Stride`
`Mapping(in).OffsetX` [0]	Mapping canvas default X-axis offset for input layer `in`
`Mapping(in).OffsetY` [0]	Mapping canvas default Y-axis offset for input layer `in`
`Mapping(in).Offset` [0]	Mapping canvas default offset for input layer `in` (mutually exclusive with `Mapping(in).Offse`
`Mapping(in).NbIterations` [0]	Mapping canvas pattern default number of iterations for input layer `in` (0 means no limit)

22.17.2 Configuration parameters (*Spike* models)

Option [default value]	Model(s)	Description
`IncomingDelay` [1 `TimePs`;100 `TimeFs`]	*all Spike*	Synaptic incoming delay w_{delay}
value		

22.18 Rbf

Radial basis function fully connected layer.

Option [default value]	Description
`NbOutputs`	Number of output neurons
`CentersFiller`	Centers initial values filler
`[NormalFiller(0.5, 0.05)]`	
`ScalingFiller`	Scaling initial values filler
`[NormalFiller(10.0, 0.05)]`	

22.18.1 Configuration parameters (*Frame* models)

Option [default value]	Model(s)	Description
`Solvers.*`	*all Frame*	Any solver parameters
`CentersSolver.*`	*all Frame*	Centers solver parameters, take precedence over the `Solvers.*` parameters
`ScalingSolver.*`	*all Frame*	Scaling solver parameters, take precedence over the `Solvers.*` parameters
`RbfApprox` [`None`]	`Frame`	Approximation for the Gaussian function, can be any of: `None`, `Rectangular` or `SemiLinear`

22.19 Resize

Resize layer can be applied to change dimension of features maps or of stimuli provider.

Option [default value]	Description
`NbOutputs`	Number of output feature maps
`OutputHeight`	Output height dimension
`OutputWidth`	Output width dimension
`Mode`	Resize interpolation mode. Can be, `Bilinear` or `BilinearTF` (TensorFlow implementation)

22.19.1 Configuration parameters

Option [default value]	Model(s)	Description
`AlignCorners` [True]	*all Frame*	Corner alignement mode if `BilinearTF` is used as interpolation mode

22.20 Softmax

Softmax layer.

Option [default value]	Description
`NbOutputs`	Number of output neurons
`WithLoss` [0]	Softmax followed with a multinomial logistic layer
`GroupSize` [0]	Softmax is applied on groups of outputs. The group size must be a divisor of `NbOutputs` parameter.

The softmax function performs the following operation, with $a^i_{x,y}$ and $b^i_{x,y}$ the input and the output respectively at position (x, y) on channel i:

$$b^i_{x,y} = \frac{\exp(a^i_{x,y})}{\sum_{j=0}^{N} \exp(a^j_{x,y})}$$

and

$$\mathrm{d}a^i_{x,y} = \sum_{j=0}^{N} \left(\delta_{ij} - a^i_{x,y}\right) a^j_{x,y} \mathrm{d}b^j_{x,y}$$

When the `WithLoss` option is enabled, compute the gradient directly in respect of the cross-entropy loss:

$$L_{x,y} = \sum_{j=0}^{N} t^j_{x,y} \log(b^j_{x,y})$$

In this case, the gradient output becomes:

$$\mathrm{d}a^i_{x,y} = \mathrm{d}b^i_{x,y}$$

with

$$\mathrm{d}b^i_{x,y} = t^i_{x,y} - b^i_{x,y}$$

22.21 Transformation

Transformation layer, which can apply any transformation described in [sec:transformations]. Useful for fully CNN post-processing for example.

Option [default value]	Description
`NbOutputs`	Number of outputs
`Transformation`	Name of the transformation to apply

The `Transformation` options must be placed in the same section.

Usage example for fully CNNs:

```
[post.Transformation-thres]
Input=... ; for example, network's logistic of softmax output layer
NbOutputs=1
Type=Transformation
Transformation=ThresholdTransformation
Operation=ToZero
Threshold=0.75

[post.Transformation-morpho]
Input=post.Transformation-thres
NbOutputs=1
Type=Transformation
Transformation=MorphologyTransformation
Operation=Opening
Size=3
```

22.22 Threshold

Apply a thresholding.

Option [default value]	Description
`NbOutputs`	Number of output feature maps
`Threshold`	Threshold value

22.22.1 Configuration parameters (*Frame* models)

Option [default value]	Model(s)	Description
`Operation` [Binary]	*all Frame*	Thresholding operation to apply. Can be:
		`Binary`
		`BinaryInverted`
		`Truncate`
		`ToZero`
		`ToZeroInverted`
`MaxValue` [1.0]	*all Frame*	Max. value to use with `Binary` and `BinaryInverted` operations

22.23 Unpool

Unpooling layer.

Option [default value]	Description
`Pooling`	Type of pooling (`Max` or `Average`)
`PoolWidth`	Width of the pooling area
`PoolHeight`	Height of the pooling area
`PoolDepth` []	Depth of the pooling area (implies 3D pooling area)
`PoolSize` []	Pooling area size (implies 2D square pooling area)
`PoolDims` []	List of space-separated dimensions for N-D pooling area
`NbOutputs`	Number of output channels
`ArgMax`	Name of the associated pool layer for the argmax (the pool layer input and the unpool layer outp
`StrideX` [1]	X-axis stride of the pooling area
`StrideY` [1]	Y-axis stride of the pooling area
`StrideZ` []	Z-axis stride of the pooling area
`Stride` [1]	Stride of the pooling area
`StrideDims` []	List of space-separated stride dimensions for N-D pooling area
`PaddingX` [0]	X-axis input padding
`PaddingY` [0]	Y-axis input padding
`PaddingZ` []	Z-axis input padding
`Padding` [0]	Input padding
`PaddingDims` []	List of space-separated padding dimensions for N-D pooling area
`ActivationFunction` []	Activation function. Can be any of `Logistic`, `LogisticWithLoss`, `Rectifier`, `Softplus`, `T`
`Mapping.NbGroups` []	Mapping: number of groups (mutually exclusive with all other Mapping.* options)
`Mapping.ChannelsPerGroup` []	Mapping: number of channels per group (mutually exclusive with all other Mapping.* options)
`Mapping.SizeX` [1]	Mapping canvas pattern default width
`Mapping.SizeY` [1]	Mapping canvas pattern default height

Table 4 – continued from previous page

Option [default value]	Description
`Mapping.Size` [1]	Mapping canvas pattern default size (mutually exclusive with `Mapping.SizeX` and `Mapping.S`
`Mapping.StrideX` [1]	Mapping canvas default X-axis step
`Mapping.StrideY` [1]	Mapping canvas default Y-axis step
`Mapping.Stride` [1]	Mapping canvas default step (mutually exclusive with``Mapping.StrideX`` and `Mapping.Strid`
`Mapping.OffsetX` [0]	Mapping canvas default X-axis offset
`Mapping.OffsetY` [0]	Mapping canvas default Y-axis offset
`Mapping.Offset` [0]	Mapping canvas default offset (mutually exclusive with `Mapping.OffsetX` and `Mapping.Offs`
`Mapping.NbIterations` [0]	Mapping canvas pattern default number of iterations (0 means no limit)
`Mapping(in).SizeX` [1]	Mapping canvas pattern default width for input layer `in`
`Mapping(in).SizeY` [1]	Mapping canvas pattern default height for input layer `in`
`Mapping(in).Size` [1]	Mapping canvas pattern default size for input layer `in` (mutually exclusive with `Mapping(in).S`
`Mapping(in).StrideX` [1]	Mapping canvas default X-axis step for input layer `in`
`Mapping(in).StrideY` [1]	Mapping canvas default Y-axis step for input layer `in`
`Mapping(in).Stride` [1]	Mapping canvas default step for input layer `in` (mutually exclusive with `Mapping(in).Stride`
`Mapping(in).OffsetX` [0]	Mapping canvas default X-axis offset for input layer `in`
`Mapping(in).OffsetY` [0]	Mapping canvas default Y-axis offset for input layer `in`
`Mapping(in).Offset` [0]	Mapping canvas default offset for input layer `in` (mutually exclusive with `Mapping(in).Offse`
`Mapping(in).NbIterations` [0]	Mapping canvas pattern default number of iterations for input layer `in` (0 means no limit)

CHAPTER

TWENTYTHREE

TARGETS (OUTPUTS & LOSSES)

A `Target` is an output point of the neural network. A `Target` specifies how the error must be computed and back-propagated at the output of a layer, and computes a classification score. A target also specifies how the database labels must be mapped to the output neurons.

To specify that the back-propagated error must be computed at the output of a given layer (generally the last layer, or output layer), one must add a target section named *LayerName*`.Target`:

```
...
[LayerName.Target]
TargetValue=1.0 ; default: 1.0
DefaultValue=0.0 ; default: -1.0
```

23.1 From labels to targets

Usually, there are as many output neurons as labels and each label is mapped to a different neuron. This is the default behavior in N2D2: each label in the dataset, by order of appearance when loading the data and label files (files are always loaded in the alphabetical order), is mapped to the next available output neuron. If there is more labels than output neurons, an error is thrown at runtime at the first occurrence of a new label exceeding the output neurons capacity.

This default behavior can be changed using a label-mapping file, where the label mapping is explicitly specified, which is useful to map several labels to the same output neuron for example.

To specify a target in the INI file, one must simply add a *LayerName*`.Target` section, where *LayerName* is the name of the layer section to which the target must be associated, as illustrated below:

```
; Output layer
[seg_output]
Input=...
Type=Conv
KernelWidth=1
KernelHeight=1
NbOutputs=1
Stride=1
ActivationFunction=LogisticWithLoss
WeightsFiller=XavierFiller
ConfigSection=common.config

[seg_output.Target]
LabelsMapping=mapping.dat
```

The `LabelsMapping` parameter, when present, is the path to the file containing the mapping of the dataset labels to the outputs of the layer. In the present case, there is a single output neuron (per output pixel) in the *seg_output* layer, which is a special case where two labels can be mapped since the activation used is a logistic function. One label can be mapped to the output value 0 and one label can be mapped to the output value 1. With more than one output neuron (per output pixel) however, it only makes sense to map a single label per output.

The label mapping file format is a two-columns, space separated, text table, with the first column corresponding to the name of the label in the dataset and the second column to the index of the associated output neuron.

Two special label names exist:

- * corresponding to annotations without valid label (label ID is -1 in N2D2), sometimes referred to as "ignore label" in N2D2;
- `default` meaning any valid label name that is not explicitly listed in the label mapping file;

The `background` name is not a reserved label name, it is simply the name that is used as `DefaultLabel` in the `Database` driver.

Here is an example of label mapping file for the single output layer *seg_output*:

```
# background (no defect)
background 0

# padding
* -1

# defect
default 1
```

Wildcards can be used as well in the name of the label:

- * meaning any one or several character(s) starting from this position (non greedy, cannot be used alone as it would refer to the special name for invalid label);
- + meaning any single character at this position.

The actual label mapping of every labels present in the dataset for a given output target is generated automatically when running the simulation. It is strongly advised to check this generated file to make sure that there is no error in the `LabelsMapping` file and that the mapping is done as intended. The file is generated in *seg_output.Target/labels_mapping.log.dat*.

23.1.1 Pixel-wise segmentation

Pixel-wise segmentation consists of directly learning a label for each output pixel of the network, typically in a fully convolutional network. Without upscaling, unpooling or deconvolution, the output size is generally smaller than the input size, by a factor *S*, corresponding to the product of the strides of the successive layers.

In practice, because of the scaling factor *S* of the network, each output pixel classify an input macro-pixel of size *SxS*. It is perfectly possible to get rid of this scaling factor by rescaling the output to the input size before the `Softmax` layer, using bilinear sampling for example. This can be done during training, in order to precisely back-propagate the pixel-wise annotations, as the bilinear sampling algorithm is easily differentiable. However, for inference, the resampling of the output does not add information to the segmentation (no new information is created when upscaling an image with context-agnostic algorithms). This is why the scaling factor *S* may be kept for inference, without resampling, in order to reduce the computing and memory cost of processing the outputs.

Other strategies are possible to back-propagate the pixel-wise annotations, which need to take into account the scaling factor *S*:

- Take the majority annotation within the macro-pixel *SxS*: the label attributed to the output pixel is the label which occurs the most often in the *SxS* macro-pixel;
- Take the majority annotation within the macro-pixel *SxS*, at the exception of a weak annotation. In this case, any label other than the weak label in the macro-pixel takes precedence over the weak one. In N2D2, this is implemented with the `WeakTarget` parameter in `[*.target]` sections:
- `WeakTarget=-1` means any target other than "ignore" takes precedence. This is useful if the background is ignored. If there is only a few pixels in the macro-pixel that are not background, they take precedence so that the macro-pixel is not considered as background.
- `WeakTarget=-2` means there is no weak label.

23.2 Loss functions

The loss function in N2D2 is always implicitly defined. For the `Softmax` layer or the `Logistic` activation, the loss is the *cross entropy loss*, when used with the `WithLoss=1` parameter. Otherwise, the default loss is the *MSE (L2) loss*.

The reason is that the error is defined at the output of a layer with the `Cell_Frame[_CUDA]::setOutputTarget()` or `Cell_Frame[_CUDA]::setOutputTargets()`, which set the value of the input gradient for the cell to *(target - output)*. These functions are called in the `Target` class.

So, if a `Target` is attached to any cell, the corresponding loss function would be the MSE loss, as the simple difference above is the derivative. For the softmax or the logistic, the special parameter `WithLoss`, when enabled, will simply bypass the function derivative and directly set the output gradient of the function to the difference above. This effectively results to a cross entropy loss with regards to the input gradient of these functions, as per the mathematical simplification of the cross entropy loss derivative multiplied by the functions gradient.

Demonstration

The cross entropy loss for a single image is:

$$L = -\sum_{j=1}^{M} y_j \log(p_j)$$

Note:

- M - number of classes (dog, cat, fish)
- log - the natural log
- y - binary indicator (0 or 1) if class label j is the correct classification for this image
- p - predicted probability that the image is of class j

The softmax performs the following operation:

$$p_i = \frac{\exp x_i}{\sum_k \exp x_k}$$

To perform the back-propagation, we need to compute the derivative of the loss L with respect to the inputs x_i:

$$\begin{aligned}\frac{\partial L}{\partial x_i} &= -\sum_k y_k \frac{\partial \log p_k}{\partial x_i} = -\sum_k y_k \frac{1}{p_k}\frac{\partial p_k}{\partial x_i} \\ &= -y_i(1-p_i) - \sum_{k\neq i} y_k \frac{1}{p_k}(-p_k p_i) \\ &= -y_i(1-p_i) + \sum_{k\neq i} y_k (p_i) \\ &= -y_i + y_i p_i + \sum_{k\neq i} y_k (p_i) \\ &= p_i \left(\sum_k y_k\right) - y_i \\ &= p_i - y_i\end{aligned}$$

given that $\sum_k y_k = 1$, as y is a vector with only one non-zero element, which is 1.

23.3 Target types

23.3.1 Target

Base `Target` class.

Base parameters:

Parameter	Default value	Description
`Type`	`TargetScore`	Type of `Target`
`TargetValue`	1.0	Target value for the target output neuron(s) (for classification)
`DefaultValue`	0.0	Default value for the non-target output neuron(s) (for classification)
`TopN`	1	The top-N estimated targets per output neuron to save
`BinaryThreshold`	0.5	Threshold for single output (binary classification).

Labels to targets parameters:

Parameter	Default value	Description
DataAsTarget	0	If true (1), the data, and not the labels, is the target (for auto-encoders)
LabelsMapping		Path to the file containing the labels to target mapping
CreateMissingLabels	0	If true (1), labels present in the labels mapping file but that are non-existent in the database are created (with 0 associated stimuli)
WeakTarget	-2	When attributing a target to an output macropixel, any target other than WeakTarget in the macropixel takes precedence over WeakTarget, regardless of their respective occurrence. • Value can be -1 (meaning any target other than "ignore" takes precedence). • Default value is -2 (meaning that there is no weak target, as a target is >= -1).

Masking parameters:

Parameter	Default value	Description
MaskLabelTarget		Name of the Target to use for MaskedLabel
MaskedLabel	-1	If >= 0, only estimated targets with ID MaskedLabel in the MaskLabelTarget target are considered in the estimated targets
MaskedLabelValue	0	If true (1), the considered estimated targets values are weighted by the estimated targets values with ID MaskedLabel in the MaskLabelTarget

Estimated output images parameters:

Parameter	Default value	Description
NoDisplayLabel	-1	If >= 0, the corresponding label ID is ignored in the estimated output image
LabelsHueOffset	0	Hue offset for the first label ID (starting from 0), for the estimated output image
EstimatedLabelsValueDisplay	1	If true (1), the value in the HSV colorspace is equal to the estimated value. Otherwise, displayed value is 255 regardless of the confidence.
ValueThreshold	0.0	Threshold for estimated value to be considered in the output logs.
ImageLogFormat	jpg	If left empty, use the database image origin format

23.3.2 TargetScore

The default target, which automatically compute the confusion matrix, confusion metrics and score, for classification or segmentation networks.

Confusion matrix:

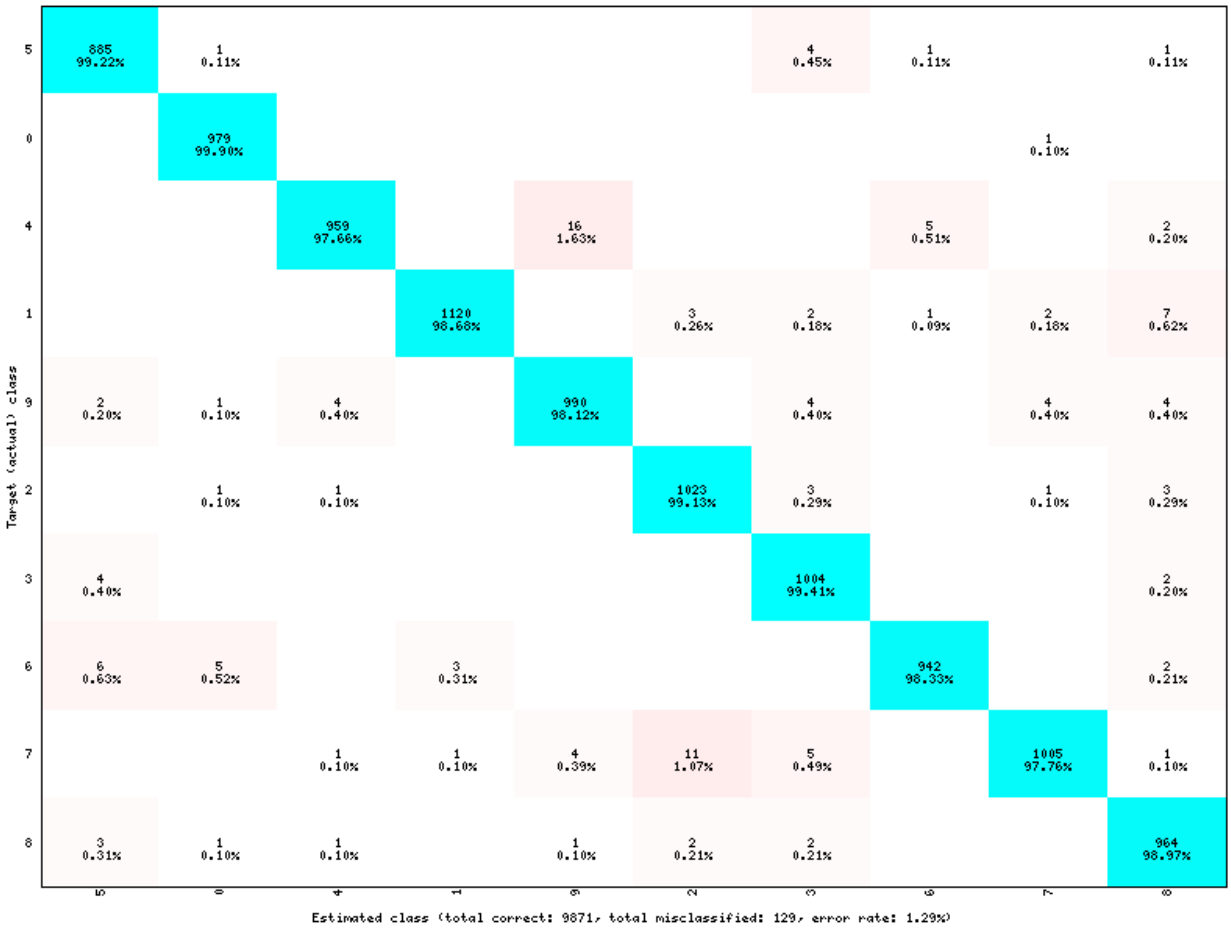


Fig. 1: Example of confusion matrix generated by a `TargetScore`.

Confusion metrics:

Score:

23.3.3 TargetROIs

The `TargetROIs` allow to perform connected-component labeling (CCL) on pixel-wise segmentation networks, to retrieve bounding boxes.

This approach is different from classical object detection networks, like SSD or Yolo, where bounding boxes are directly inferred from anchors.

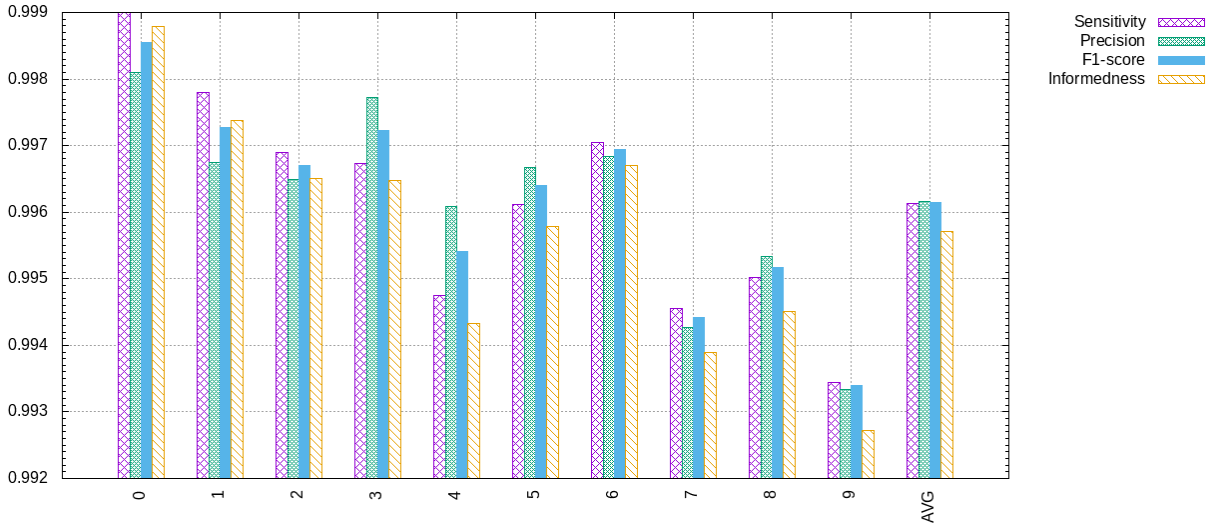


Fig. 2: Example of confusion metrics generated by a `TargetScore`.

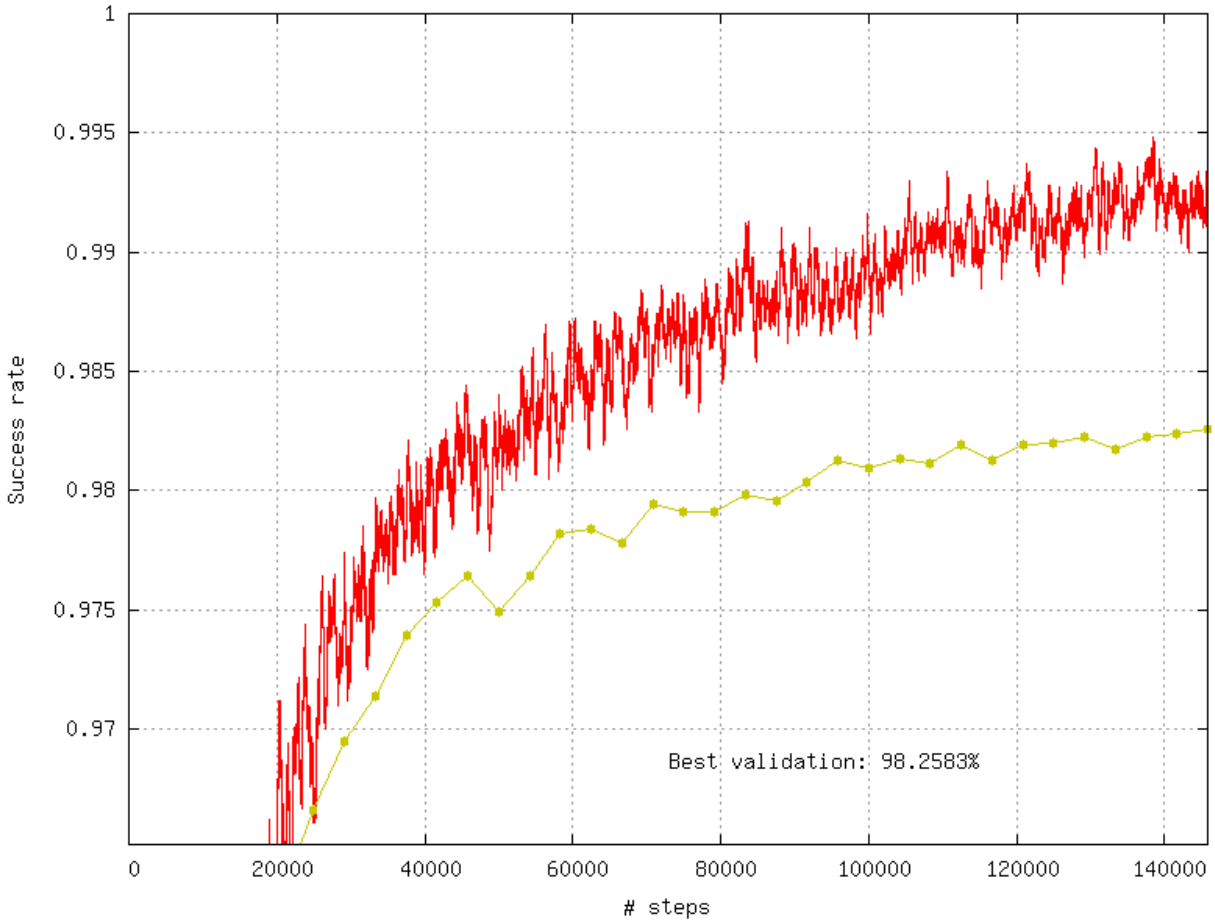


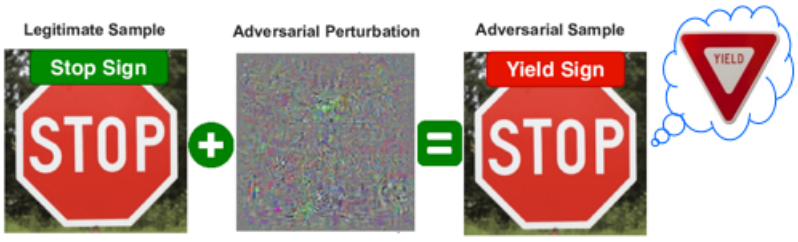
Fig. 3: Example of validation score generated by a `TargetScore`.

Parameter	Default value	Description
`MinSize`	0	Minimum number of macro-pixels above threshold than can constitute a bounding box. Bounding boxes with fewer than `MinSize` macro-pixels above threshold are discarded
`MinOverlap`	0.5	Minimum overlap (IoU) of a bounding box with an annotation to be considered a match
`FilterMinHeight`	0	Minimum height of the ROI to keep it
`FilterMinWidth`	0	Minimum width of the ROI to keep it
`FilterMinAspectRatio`	0.0	Minimum aspect ratio (width/height) of the ROI to keep it (default is 0.0 = no minimum)
`FilterMaxAspectRatio`	0.0	Maximum aspect ratio (width/height) of the ROI to keep it (default is 0.0 = no minimum)
`MergeMaxHDist`	1	Maximum horizontal distance for merging (in macro-pixels)
`MergeMaxVDist`	1	Maximum vertical distance for merging (in macro-pixels)
`ScoreTopN`	1	TopN number of class scores to keep for the ROI

CHAPTER

TWENTYFOUR

ADVERSARIAL MODULE



This module aims to provide users several tools to simulate an adversarial attack on a neural network.

Adversarial attacks can threaten the security of users. They consist in deceiving the network without the user's knowledge by making imperceptible changes to the input data.

This module give you the possibility to run an adversarial attack, study the consequences of differents attacks and implement your own attacks.

Contents

24.1 For the users

24.1.1 Run an adversarial attack

In order to perform an adversarial attack simulation, you will need to add in the Ini file of your network a new section called `[sp.Adversarial]`. In this section, you can specify the type of attack you wish to run as well as some parameters to design your attack.

```
; Environment
[sp]
SizeX=32
```

(continues on next page)

(continued from previous page)

```
SizeY=32
BatchSize=128

[sp.Adversarial]
Attack=PGD
Eps=0.2
```

The parameters you can modify are indicated in the following table:

Option [default value]	Description
`Attack` [`None`]	Name of the attack (None, Vanilla, GN, FGSM, PGD)
`Eps` [`0.1`]	Degradation rate
`NbIterations` [`10`]	Number of iterations (if the attack requires several iterations)
`RandomStart` [`false`]	If `true`, randomize every pixel between pixel-Eps and pixel+Eps
`Targeted` [`false`]	If `true`, activate targeted mode (label+1 found by the deepNet)

After specifying the design of the attack, you can run the regular N2D2 options like `-test` or `-learn`. Therefore, you can test your network against adversarial attacks by running the test option.

```
$ ./n2d2 ResNet-18-BN.ini -test
```

Moreover, running a learning with the adversarial module will execute a robust learning.

24.1.2 1st function to study adversarial attacks

This function can allow you to perform an adversarial attack on a single batch. The function indicates the successful attacks and stores the original and the modified inputs in the `testAdversarial` folder.

To perform the function, please run

```
$ ./n2d2 ResNet-18-BN.ini -dev 0 -testAdv Solo
```

You should see on your terminal screen

```
PGD attack
Untargeted mode
BatchPos 0: Successful attack (label: 7, estimated: 9 with 55.54%)
BatchPos 1: Successful attack (label: 3, estimated: 5 with 43.59%)
BatchPos 2: Successful attack (label: 1, estimated: 7 with 53.33%)
BatchPos 3: Successful attack (label: 8, estimated: 2 with 50.92%)
BatchPos 4: Successful attack (label: 5, estimated: 8 with 51.02%)
BatchPos 5: Successful attack (label: 0, estimated: 6 with 50.26%)
BatchPos 6: Successful attack (label: 8, estimated: 3 with 59.07%)
BatchPos 7: Successful attack (label: 7, estimated: 9 with 52.42%)
BatchPos 8: Successful attack (label: 9, estimated: 7 with 62.47%)
BatchPos 9: Successful attack (label: 5, estimated: 0 with 61.88%)
Time elapsed: 6.85 s
```

24.1.3 2nd function to study adversarial attacks

This function can allow you to perform an adversarial attack on multiple batches (2000 images). The function indicates the ratio of successful attacks. It also provides the ratio of successful attacks for each class.

To perform the function, please run

```
$ ./n2d2 ResNet-18-BN.ini -dev 0 -testAdv Multi
```

You should see on your terminal screen

```
Treating 2000/2000
Analysis of the results...
Successful attacks: 2.20%
including network errors: 1.35%
- successful attacks on class 0: 0.00% (0/218)
- successful attacks on class 1: 0.00% (0/219)
- successful attacks on class 2: 2.78% (6/216)
- successful attacks on class 3: 2.16% (4/185)
- successful attacks on class 4: 1.60% (3/188)
- successful attacks on class 5: 4.89% (9/184)
- successful attacks on class 6: 2.55% (5/196)
- successful attacks on class 7: 3.37% (7/208)
- successful attacks on class 8: 3.66% (7/191)
- successful attacks on class 9: 1.54% (3/195)
Time elapsed: 4.62 s
```

24.2 For the developers

It's now your turn to implement your own attacks !

To integrate your attack inside N2D2, you will have to change the following files:

- include/Adversarial.hpp

```
enum Attack_T {
    None,
    Vanilla,
    GN,
    FGSM,
    PGD,
    My_attack     <---------------
};

...

namespace {
template <>
const char* const EnumStrings<N2D2::Adversarial::Attack_T>::data[]
    = {"None", "Vanilla", "GN", "FGSM", "PGD", "My_attack"};
}

...
```

(continues on next page)

(continued from previous page)

```
void FFGSM_attack(std::shared_ptr<DeepNet>& deepNet,
          const float eps,
          const float alpha,
          const bool targeted = false);

void PGD_attack(std::shared_ptr<DeepNet>& deepNet,
                const float eps,
                const unsigned int nbIter,
                const float alpha,
                const bool targeted = false,
                const bool random_start = false);

void My_attack();                                <---------------
```

- src/Adversarial.cpp

  in the attackLauncher function, indicate your attack in the switch.

```
case Vanilla:
    Vanilla_attack();
    break;

case FGSM:
    FGSM_attack(deepNet, mEps, mTargeted);
    break;

case My_attack:
    My_attack();                                <---------------
    break;
```

```
void N2D2::My_attack()
{
    /* My implementation */
}
```

- src/Generator/AdversarialGenerator.cpp

  If you need to add new parameters, add them in the class Adversarial and don't forget to save them during the generation of the Adversarial layer. You can implement some setter methods in the AdversarialGenerator file.

```
adv->setEps(iniConfig.getProperty<float>("Eps", adv->getEps()));
adv->setNbIterations(iniConfig.getProperty<unsigned int>("NbIterations",␣
↪adv->getNbIterations()));
adv->setRandomStart(iniConfig.getProperty<bool>("RandomStart", adv->
↪getRandomStart()));
adv->setTargeted(iniConfig.getProperty<bool>("Targeted", adv->
↪getTargeted()));

return adv;
```

CHAPTER

TWENTYFIVE

INTRODUCTION

For notation purposes, we will refer to the python library of N2D2 as n2d2. This library uses the core function of N2D2 and add an extra layer of abstraction to make the experience more user friendly. With the library you can import data, pre-process them, create a deep neural network model, train it and realize inference with it. You can also import a network using the *ini file configuration* or the ONNX library.

Here are the functionalities available with the Python API :

Feature	Available	Python API Only
Import a network from an INI file	✓	
Import a network from an ONNX file	✓	
Build a network with the API	✓	
Load and apply transformation to a dataset	✓	
Train a network	✓	
Flexible definition of the computation graph	✓	✓
Test a network with the N2D2 analysis tools	✓	
Torch interoperability	✓	✓
Keras interoperability	✓	✓
Multi GPU support	✓	
Exporting network	✓	

25.1 Installation of the virtual environment

To run the python API, it's good practice to use `python 3.7` or a newer version in a virtual environment.
To set up your environment, please follow these steps:

```
# Create your python virtual environment
virtualenv -p python3.7 env

# Activate the virtual environment
source env/bin/activate

# Check versions
python --version
pip --version

# To leave the virtual environment
deactivate
```

If everything went well, you should have the version `3.7` of python.

25.2 Installation of the Python API

There are multiple methods to install the python API on your device.
Feel free to use the method of your choice.

25.2.1 With the Python Package Index (Py Pi)

> **Warning:** This method is not supported anymore, we are working on it !

You can have access to the last stable version of the python API by using `pip` and importing the package `n2d2`.

```
pip install n2d2
```

25.2.2 From the N2D2 Github repository

You can have access to the developer version by importing the API from the N2D2 Github repository via `pip`.

```
pip install git+https://github.com/CEA-LIST/N2D2
```

25.2.3 If you have already cloned the Github repository

You can still build the python API with a cloned N2D2 repository. Go at the root of the N2D2 projet and follow the following steps (don't forget to activate your virtual environment before).

```
# Build the N2D2 library
python setup.py bdist_wheel

# Install the n2d2 python packages in your virtual environment
pip install .
```

25.2.4 Installation for developer

If you want to install n2d2 as seomeone who wants to contribute to n2d2, we recommand the following setup :

Inside your n2d2 project, create a build folder and compile N2D2 inside it :

```
mkdir build && cd build
cmake .. && make -j 8
```

Once this is done, you have generated the shared object : `lib/n2d2.*.so`.

You can add the generated *lib* folder and the python source in your `PYTHONPATH` with the command :

```
export PYTHONPATH=$PYTHONPATH:<N2D2_BUILD_PATH>/lib:<N2D2_PATH>/python
```

Note: Add this line in your bashrc to always have a good `PYTHONPATH` setup !

To check if your PYTHONPATH works properly you can try to import `N2D2` (verify that the compilation went well) and then `n2d2` (verify that your `PYTHONPATH` point the n2d2 python API).

25.2.5 Frequent issues

Module not found N2D2

If when you import `n2d2` you get this error :

```
ModuleNotFoundError: No module named 'N2D2'
```

This is likely due to your python version not matching with the one used to compile N2D2.

You can find in your `site-packages` (or in your `build/lib` if you have compiled N2D2 with CMake) a `.so` file named like this : `N2D2.cpython-37m-x86_64-linux-gnu.so`.

This file name indicates the python version used to compile N2D2, in this example 3.7.

You should either make sure to use a virtualenv with the right python version or check the bellow section.

N2D2 doesn't compile with the right version of Python

When compiling N2D2 you can use an argument to specify the python version you want to compile N2D2 for.

```
cmake -DPYTHON_EXECUTABLE=<path_to_python_binary> <path_to_n2d2_cmakefile>
```

Note: On linux you can use `$(which python)` to use your default python binary.

You can then check the version of python on the shared object in `build/lib`.

For example, this shared object `N2D2.cpython-37m-x86_64-linux-gnu.so` have been compiled for python3.7.

Lib not found when compiling

If CMake fails to find lib files when compiling, this may be due to the absence of the dependency `python3-dev`.

When generating a new virtualenv after installing the dependency, you should see `include/python3.7m` inside the generated folder.

If not, you may need to reboot in order to update system variables.

25.3 Test of the Python API

Whatever the method you chose, it should compile the n2d2 libraries and add them to your virtual environnement.

You can test it by trying to import n2d2 in your python interpreter :

```
python
>>> import n2d2
>>> print(n2d2.Tensor([2,3]))
n2d2.Tensor([
0 0 0
0 0 0
], device=cpu, datatype=float)
>>> exit()
```

You can find more examples in the Python API section if you want to test every feature.

It might be possible you could find some issues by using the API.
So please notify us at https://github.com/CEA-LIST/N2D2/issues if you find any problem or any possible improvement.

25.4 Default values

The python API used default values that you can modify at any time in your scripts.

25.4.1 List of modifiable parameters

Here we will list parameters which can be directly modified in your script.

Default parameters	Description
`default_model`	If you have compiled N2D2 with **CUDA**, you can use `Frame_CUDA`, default= `Frame`
`default_datatype`	Datatype of the layer of the neural network. Can be `double` or `float`, default= `float` **Important :** This variable doesn't affect the data type of `n2d2.Tensor` objects.
`verbosity`	Level of verbosity, can be `n2d2.global_variables.Verbosity.graph_only`, `n2d2.global_variables.Verbosity.short` or `n2d2.global_variables.Verbosity.detailed`, default= `n2d2.global_variables.Verbosity.detailed`
`seed`	Seed used to generate random numbers(0 = time based), default = `0`
`cuda_device`	Device to use for GPU computation with CUDA, you can enable multi GPU by giving a tuple of device, default = `0`

25.4.2 Example

```
n2d2.global_variables.default_model = "Frame_CUDA"

n2d2.global_variables.default_datatype = "double"

n2d2.global_variables.verbosity = n2d2.global_variables.Verbosity.graph_only

n2d2.global_variables.seed = 1

n2d2.global_variables.cuda_device = 1
# Multi GPU example :
n2d2.global_variables.cuda_device = 0, 1
```

CHAPTER

TWENTYSIX

DATABASES

26.1 Introduction

The python library integrates pre-defined modules for several well-known database used in the deep learning community, such as MNIST, GTSRB, CIFAR10 and so on. That way, no extra step is necessary to be able to directly build a network and learn it on these database. The library allow you to add pre-process data with built in Transformation.

26.2 Database

The python library provide you with multiple object to manipulate common database.

Loading hand made database can be done using `n2d2.database.DIR`.

Like in the following example :

```
# Creating the database object
db = n2d2.database.DIR()

provider = n2d2.provider.DataProvider(db, data_dims)

# The zeroes represent the depth to seek the data.
db.load(data_path, 0, label_path, 0)

# With this line we put all the data in the learn partition:
db.partition_stimuli(learn=1, validation=0, test=0)
provider.set_partition("Learn")

inputs_tensor = provider.read_random_batch()
```

26.2.1 DIR

Loading a custom database

Hand made database stored in files directories are directly supported with the `DIR_Database` module. For example, suppose your database is organized as following :

- `GST/airplanes`: 800 images
- `GST/car_side`: 123 images
- `GST/Faces`: 435 images

- `GST/Motorbikes`: 798 images

You can then instanciate this database as input of your neural network using the following line:

```
database = n2d2.database.DIR("./GST", learn=0.4, validation=0.2)
```

Each subdirectory will be treated as a different label, so there will be 4 different labels, named after the directory name.

The stimuli are equi-partitioned for the learning set and the validation set, meaning that the same number of stimuli for each category is used. If the learn fraction is 0.4 and the validation fraction is 0.2, as in the example above, the partitioning will be the following:

Label ID	Label name	Learn set	Validation set	Test set
[0.5ex] 0	`airplanes`	49	25	726
1	`car_side`	49	25	49
2	`Faces`	49	25	361
3	`Motorbikes`	49	25	724
	Total:	196	100	1860

Note: If `equiv_label_partitioning` is 1 (default setting), the number of stimuli per label that will be partitioned in the learn and validation sets will correspond to the number of stimuli from the label with the fewest stimuli.

To load and partition more than one `DataPath`, one can use the `n2d2.database.Database.load()` method.

This method will load data in the partition `Unpartitionned`, you can move the stimuli in the `Learn`, `Validation` or `Test` partition using the `n2d2.database.Database.partition_stimuli()` method.

Handling labelization

By default, your labels will be ordered by alphabetical order. If you need your label to be in a specific order, you can specify it using an exterior file we will name it `label.dat` for this example :

```
airplanes 0
car_side 1
Motorbikes 3
Faces 2
```

Then to load the database we will use :

```
database = n2d2.database.DIR("./GST", learn=0.4, validation=0.2, label_path="./label.dat
↪", label_depth=0)
```

Warning: It is important to specify `label_depth=0` if you are specifying `label_path` !

26.2.2 MNIST

26.2.3 ILSVRC2012

26.2.4 CIFAR10

26.2.5 CIFAR100

26.2.6 Cityscapes

26.2.7 GTSRB

26.3 Transformations

26.3.1 Composite

26.3.2 PadCrop

26.3.3 Distortion

26.3.4 Rescale

26.3.5 Reshape

26.3.6 ColorSpace

26.3.7 Flip

26.3.8 RangeAffine

26.3.9 SliceExtraction

26.3.10 RandomResizeCrop

26.3.11 ChannelExtraction

26.4 Sending data to the Neural Network

Once a database is loaded, n2d2 use `n2d2.provider.DataProvider` to provide data to the neural network.

The `n2d2.provider.DataProvider` will automatically apply the `n2d2.transform.Transformation` to the dataset. To add a transformation to the provider, you should use the method `n2d2.transform.Transformation.add_transformation()`.

26.5 Example

In this example, we will show you how to create a `n2d2.database.Database`, `n2d2.provider.Provider` and apply `n2d2.transformation.Transformation` to the data.

We will use the `n2d2.database.MNIST` database driver, rescale the images to a 32x32 pixels size and then print the data used for the learning.

```
# Loading data
database = n2d2.database.MNIST(data_path=path, validation=0.1)

# Initializing DataProvider
provider = n2d2.provider.DataProvider(database, [32, 32, 1], batch_size=batch_size)

# Applying Transformation
provider.add_transformation(n2d2.transform.Rescale(width=32, height=32))

# Setting the partition of data we will use
provider.set_partition("Learn")

# Iterating other the inputs
for inputs in provider:
        print(inputs)
```

CHAPTER

TWENTYSEVEN

CELLS

27.1 Introduction

Cell objects are the atomics elements that compose a deep neural network.

They are the node of the computation graph. `n2d2.cells.NeuralNetworkCell` are not dependant of a DeepNet this allow a dynamic management of the computation.

Cells are organize with the following logic :

- `n2d2.cells.NeuralNetworkCell` : Atomic cell of a neural network;
- `n2d2.cells.Block` : Store a collection of `n2d2.cells.NeuralNetworkCell`, the storage order does **not** determine the graph computation;
- `n2d2.cells.DeepNetCell` : This cell allow you to use an `N2D2.DeepNet`, it can be used for *ONNX* and *INI* import or to run optimize learning;
- `n2d2.cells.Iterable` : Similar to `n2d2.cells.Block` but the order of storage determine the computation graph;
- `n2d2.cells.Sequence` : A vertical structure to create neural network;
- `n2d2.cells.Layer` : An horizontal structure to create neural network.

27.1.1 Block

27.1.2 Sequence

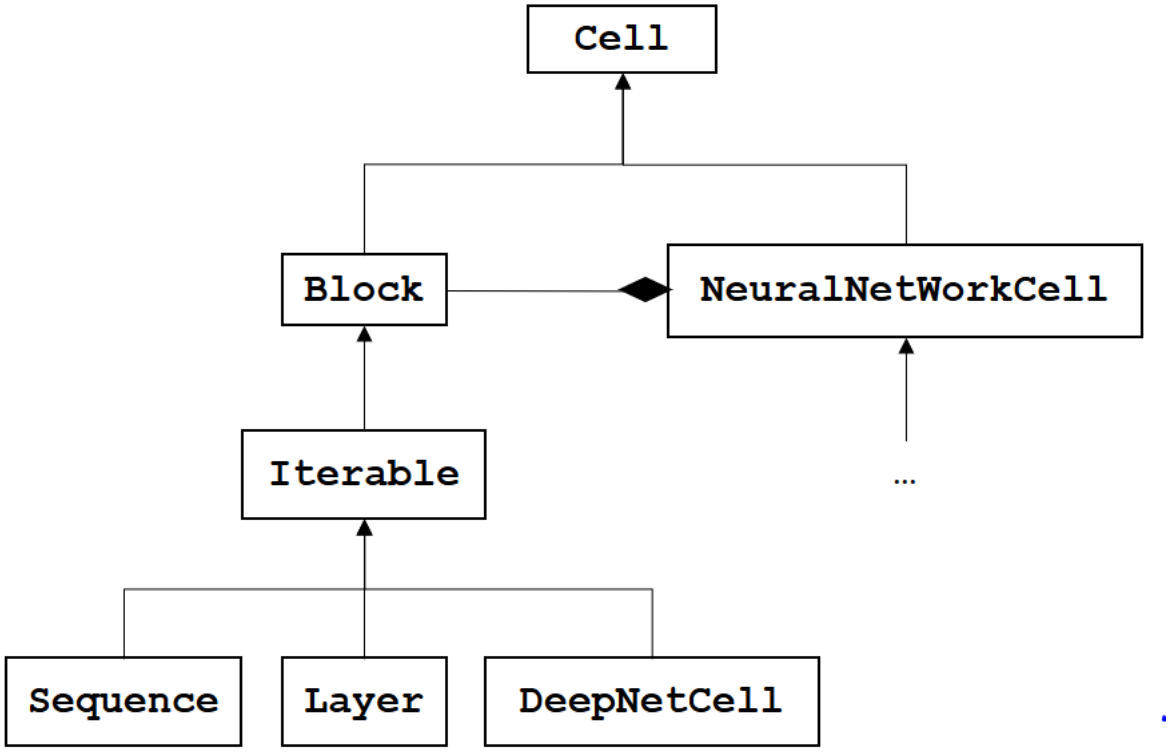
27.1.3 Layer

27.1.4 DeepNetCell

The `n2d2.cells.DeepNetCell` constructor require an `N2D2.DeepNet`. In practice, you will not use the constructor directly.

There are three methods to generate a `n2d2.cells.DeepNetCell` : `n2d2.cells.DeepNetCell.load_from_ONNX()`, `n2d2.cells.DeepNetCell.load_from_INI()`, `n2d2.cells.Sequence.to_deepnet_cell()`

The DeepNetCell can be used to train the neural network in an efficient way thanks to `n2d2.cells.DeepNetCell.fit()`.



Example

You can create a DeepNet cell with `n2d2.cells.DeepNetCell.load_from_ONNX()` :

```
database = n2d2.database.MNIST(data_path=DATA_PATH, validation=0.1)
provider = n2d2.provider.DataProvider(database, [28, 28, 1], batch_size=BATCH_SIZE)
model = n2d2.cells.DeepNetCell.load_from_ONNX(provider, ONNX_PATH)
model.fit(nb_epochs)
model.run_test()
```

Using `n2d2.cells.DeepNetCell.fit()` method will reduce the learning time as it will parallelize the loading of the batch of data and the propagation.

If you want to use the dynamic computation graph provided by the API, you can use the `n2d2.cells.DeepNetCell` as a simple cell.

```
database = n2d2.database.MNIST(data_path=DATA_PATH, validation=0.1)
provider = n2d2.provider.DataProvider(database, [28, 28, 1], batch_size=BATCH_SIZE)
model = n2d2.cells.DeepNetCell.load_from_ONNX(provider, ONNX_PATH)
sequence = n2d2.cells.Sequence([model, n2d2.cells.Softmax(with_loss=True)])
input_tensor = n2d2.Tensor(DIMS)
output_tensor = sequence(input_tensor)
```

27.2 Cells

27.2.1 NeuralNetworkCell

27.2.2 Conv

27.2.3 Deconv

27.2.4 Fc

27.2.5 Dropout

27.2.6 ElemWise

27.2.7 Padding

27.2.8 Softmax

27.2.9 BatchNorm2d

27.2.10 Pool

27.2.11 Activation

27.2.12 Reshape

27.2.13 Resize

27.2.14 Scaling

27.2.15 Transformation

27.2.16 Transpose

27.3 Saving parameters

You can save the parameters (weights, biases ...) of your network with the method *export_free_parameters*. To load those parameters you can use the method *import_free_parameters*.

With n2d2 you can choose wether you want to save the parameters of a part of your network or of all your graph.

Object	Save parameters	Load parameters
`n2d2.cells.NeuralNetworkCell`	`n2d2.cells.NeuralNetworkCell.export_free_parameters()`	`n2d2.cells.NeuralNetworkCell.import_free_parameters()`
`n2d2.cells.Block`	`n2d2.cells.Block.import_free_parameters()`	`n2d2.cells.Block.import_free_parameters()`

27.4 Configuration section

If you want to add the same parameters to multiple cells, you can use a `n2d2.ConfigSection`.

`n2d2.ConfigSection` are used like dictionaries and passes to the constructor of classes like `kwargs`.

27.4.1 Usage example

```
conv_config = n2d2.ConfigSection(no_bias=True)
n2d2.cells.Conv(3, 32, [4, 4], **conv_config)
```

This creates a `n2d2.cells.Conv` with the parameter *no_bias=True*. This functionality allow you to write more concise code, when multiple cells share the same parameters.

Warning: If you want to pass an object as a parameter for multiple n2d2 object. You need to create a wrapping function to create your object. Example :

```
def conv_def():
        return n2d2.ConfigSection(weights_solver=n2d2.solver.SGD())
n2d2.cells.Conv(3, 32, [4, 4], **conv_def())
```

27.5 Mapping

You can change the mapping of the input for some cells (see if they have `mapping` parameter available).

You can create a mapping manually with a `n2d2.Tensor` object :

```
mapping=n2d2.Tensor([15, 24], datatype="bool")
mapping.set_values([
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1],
[1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1],
[0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1],
[0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1],
[0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1],
[0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1],
[0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1],
[0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1]])
```

Or use the Mapping object :

```
mapping=n2d2.mapping.Mapping(nb_channels_per_group=2).create_mapping(15, 24)
```

Which create the following mapping :

```
1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
```

27.6 Solver

You can associate at construction and run time a `n2d2.solver.Solver` object to a cell. This solver object will optimize the parameters of your cell using a specific algorithm.

27.6.1 Usage example

In this short example we will see how to associate a solver to a model and to a cell object at construction and at runtime.

Set solver at construction time

Let's create a couple of `n2d2.cells.Fc` cell and add them to a `n2d2.cells.Sequence`. At construction time we will set the solver of one of them to a `n2d2.solver.SGD` with a `learning_rate=0.1`.

```
import n2d2

cell1 = n2d2.cells.Fc(2,2, solver=n2d2.solver.SGD(learning_rate=0.1))
cell2 = n2d2.cells.Fc(2,2)

model = n2d2.cells.Sequence([cell1, cell2])

print(model)
```

Output :

```
'Sequence_0' Sequence(
        (0): 'Fc_0' Fc(Frame<float>)(nb_inputs=2, nb_outputs=2 | back_propagate=True,␣
↪drop_connect=1.0, no_bias=False, normalize=False, outputs_remap=, weights_export_
↪format=OC, activation=None, weights_solver=SGD(clamping=, decay=0.0, iteration_size=1,␣
↪learning_rate=0.1, learning_rate_decay=0.1, learning_rate_policy=None, learning_rate_
↪step_size=1, max_iterations=0, min_decay=0.0, momentum=0.0, polyak_momentum=True,␣
↪power=0.0, warm_up_duration=0, warm_up_lr_frac=0.25), bias_solver=SGD(clamping=,␣
↪decay=0.0, iteration_size=1, learning_rate=0.1, learning_rate_decay=0.1, learning_rate_
```

(continues on next page)

(continued from previous page)

```
↪policy=None, learning_rate_step_size=1, max_iterations=0, min_decay=0.0, momentum=0.0,␣
↪polyak_momentum=True, power=0.0, warm_up_duration=0, warm_up_lr_frac=0.25), weights_
↪filler=Normal(mean=0.0, std_dev=0.05), bias_filler=Normal(mean=0.0, std_dev=0.05),␣
↪quantizer=None)
        (1): 'Fc_1' Fc(Frame<float>)(nb_inputs=2, nb_outputs=2 | back_propagate=True,␣
↪drop_connect=1.0, no_bias=False, normalize=False, outputs_remap=, weights_export_
↪format=OC, activation=None, weights_solver=SGD(clamping=, decay=0.0, iteration_size=1,␣
↪learning_rate=0.01, learning_rate_decay=0.1, learning_rate_policy=None, learning_rate_
↪step_size=1, max_iterations=0, min_decay=0.0, momentum=0.0, polyak_momentum=True,␣
↪power=0.0, warm_up_duration=0, warm_up_lr_frac=0.25), bias_solver=SGD(clamping=,␣
↪decay=0.0, iteration_size=1, learning_rate=0.01, learning_rate_decay=0.1, learning_
↪rate_policy=None, learning_rate_step_size=1, max_iterations=0, min_decay=0.0,␣
↪momentum=0.0, polyak_momentum=True, power=0.0, warm_up_duration=0, warm_up_lr_frac=0.
↪25), weights_filler=Normal(mean=0.0, std_dev=0.05), bias_filler=Normal(mean=0.0, std_
↪dev=0.05), quantizer=None)
)
```

Set a solver for a specific parameter

We can set a new solver for the bias of the second cell fully connected cell. This solver will be different than the weight parameter one.

Note: Here we access the cell via its instanciate object but we could have used its name : `model["Fc_1"].bias_solver=n2d2.solver.Adam()`.

```
cell2.bias_solver=n2d2.solver.Adam()

print(model)
```

Output :

```
'Sequence_0' Sequence(
        (0): 'Fc_0' Fc(Frame<float>)(nb_inputs=2, nb_outputs=2 | back_propagate=True,␣
↪drop_connect=1.0, no_bias=False, normalize=False, outputs_remap=, weights_export_
↪format=OC, activation=None, weights_solver=SGD(clamping=, decay=0.0, iteration_size=1,␣
↪learning_rate=0.1, learning_rate_decay=0.1, learning_rate_policy=None, learning_rate_
↪step_size=1, max_iterations=0, min_decay=0.0, momentum=0.0, polyak_momentum=True,␣
↪power=0.0, warm_up_duration=0, warm_up_lr_frac=0.25), bias_solver=SGD(clamping=,␣
↪decay=0.0, iteration_size=1, learning_rate=0.1, learning_rate_decay=0.1, learning_rate_
↪policy=None, learning_rate_step_size=1, max_iterations=0, min_decay=0.0, momentum=0.0,␣
↪polyak_momentum=True, power=0.0, warm_up_duration=0, warm_up_lr_frac=0.25), weights_
↪filler=Normal(mean=0.0, std_dev=0.05), bias_filler=Normal(mean=0.0, std_dev=0.05),␣
↪quantizer=None)
        (1): 'Fc_1' Fc(Frame<float>)(nb_inputs=2, nb_outputs=2 | back_propagate=True,␣
↪drop_connect=1.0, no_bias=False, normalize=False, outputs_remap=, weights_export_
↪format=OC, activation=None, weights_solver=SGD(clamping=, decay=0.0, iteration_size=1,␣
↪learning_rate=0.01, learning_rate_decay=0.1, learning_rate_policy=None, learning_rate_
↪step_size=1, max_iterations=0, min_decay=0.0, momentum=0.0, polyak_momentum=True,␣
↪power=0.0, warm_up_duration=0, warm_up_lr_frac=0.25), bias_solver=Adam(beta1=0.9,␣
↪beta2=0.999, clamping=, epsilon=1e-08, learning_rate=0.001), weights_
```

(continues on next page)

(continued from previous page)

```
↪filler=Normal(mean=0.0, std_dev=0.05), bias_filler=Normal(mean=0.0, std_dev=0.05),␣
↪quantizer=None)
)
```

Set a solver for a model

We can set a solver to the whole `n2d2.cells.Sequence` with the method `n2d2.cells.Sequence.set_solver()`.

```
model.set_solver(n2d2.solver.Adam(learning_rate=0.1))

print(model)
```

Output :

```
'Sequence_0' Sequence(
        (0): 'Fc_0' Fc(Frame<float>)(nb_inputs=2, nb_outputs=2 | back_propagate=True,␣
↪drop_connect=1.0, no_bias=False, normalize=False, outputs_remap=, weights_export_
↪format=OC, activation=None, weights_solver=Adam(beta1=0.9, beta2=0.999, clamping=,␣
↪epsilon=1e-08, learning_rate=0.1), bias_solver=Adam(beta1=0.9, beta2=0.999, clamping=,␣
↪epsilon=1e-08, learning_rate=0.1), weights_filler=Normal(mean=0.0, std_dev=0.05), bias_
↪filler=Normal(mean=0.0, std_dev=0.05), quantizer=None)
        (1): 'Fc_1' Fc(Frame<float>)(nb_inputs=2, nb_outputs=2 | back_propagate=True,␣
↪drop_connect=1.0, no_bias=False, normalize=False, outputs_remap=, weights_export_
↪format=OC, activation=None, weights_solver=Adam(beta1=0.9, beta2=0.999, clamping=,␣
↪epsilon=1e-08, learning_rate=0.1), bias_solver=Adam(beta1=0.9, beta2=0.999, clamping=,␣
↪epsilon=1e-08, learning_rate=0.1), weights_filler=Normal(mean=0.0, std_dev=0.05), bias_
↪filler=Normal(mean=0.0, std_dev=0.05), quantizer=None)
)
```

27.6.2 SGD

27.6.3 Adam

27.7 Filler

You can associate to a cell at construction time a `n2d2.filler.Filler` object. This object will fill weights and biases using a specific method.

27.7.1 Usage example

In this short example we will see how to associate a filler to a cell object, how to get the weights and biases and how to set a new filler and refill the weights.

Setting a filler at construction time

We begin by importing `n2d2` and creating a `n2d2.cells.Fc` object. We will associate a `n2d2.filler.Constant` filler.

Note: If you want to set a filler only for weights (or biases) you could have used the parameter `weight_filler` (or `bias_filler`).

```
import n2d2
cell = n2d2.cells.Fc(2,2, filler=n2d2.filler.Constant(value=1.0))
```

If you print the weights, you will see that they are all set to one.

```
print("--- Weights ---")
for channel in cell.get_weights():
for value in channel:
        print(value)
```

Output :

```
--- Weights ---
n2d2.Tensor([
1
], device=cpu, datatype=f)
n2d2.Tensor([
1
], device=cpu, datatype=f)
n2d2.Tensor([
1
], device=cpu, datatype=f)
n2d2.Tensor([
1
], device=cpu, datatype=f)
```

Same with the biases

```
print("--- Biases ---")
for channel in cell.get_biases():
print(channel)
```

Output :

```
--- Biases ---
n2d2.Tensor([
1
], device=cpu, datatype=f)
n2d2.Tensor([
1
], device=cpu, datatype=f)
```

Changing the filler of an instanciated object

You can set a new filler for bias by changing the `bias_filler` attribute (or `weight_filler` for only weights or `filer` for both).

However changing the filler doesn't change the parameter values, you need to use the method `n2d2.cells.Fc.refill_bias()` (see also `n2d2.cells.Fc.refill_weights()`)

Note: You can also use the method `n2d2.cells.Fc.set_filler()`, `n2d2.cells.Fc.set_weights_filler()` and `n2d2.cells.Fc.set_biases_filler()`. Which have a refill option.

```
cell.bias_filler=n2d2.filler.Normal()
cell.refill_bias()
```

You can then observe the new biases :

```
print("--- New Biases ---")
for channel in cell.get_biases():
print(channel)
```

Output :

```
--- New Biases ---
n2d2.Tensor([
1.32238
], device=cpu, datatype=f)
n2d2.Tensor([
-0.0233932
], device=cpu, datatype=f)
```

27.7.2 He

27.7.3 Normal

27.7.4 Constant

27.7.5 Xavier

27.8 Activations

You can associate to some cell an activation function.

27.8.1 Linear

27.8.2 Rectifier

27.8.3 Tanh

27.9 Target

Last cell of the network this object computes the loss.

To understand what the Target does, please refer to this part of the documentation : *Target INI*.

27.9.1 Usage example

How to use a *Target* to train your model :

```
# Propagation & BackPropagation example
output = model(stimuli)
loss = target(output)
loss.back_propagate()
loss.update()
```

Log performance analysis of your training :

```
### After validation ###
# save computational stats of the network
target.log_stats("name")
# save a confusion matrix
target.log_confusion_matrix("name")
# save a graph of the loss and the validation score as a function of the number of steps
target.log_success("name")
```

CHAPTER

TWENTYEIGHT

TENSOR

28.1 Introduction

`n2d2.Tensor` is a wrapper of the `Tensor` object available in N2D2 (see *Tensor*).

The class `n2d2.Tensor` contains a reference to the element which produce it and can be seen as the edge of the computation graph.

28.2 Tensor

28.3 Manipulating tensors

For setting and getting value we will be using the following tensor as an example :

```
tensor = n2d2.Tensor([2, 3])
```

```
0 0 0
0 0 0
```

You can set and get values using :

28.3.1 Coordinates

```
tensor[1,0] = 1 # Using coordinates
value = tensor[1,0]
```

If you print the tensor you will see :

```
0 0 0
1 0 0
```

28.3.2 Index

You can use an index to get or set elements of a tensor. The index correspond to the flatten representation of your tensor.

```
tensor[0] = 2
value = tensor[0]
```

If you print the tensor you will see :

```
2 0 0
0 0 0
```

28.3.3 Slice

Note: Slice are supported only for assignment !

```
tensor[1:3] = 3
```

If you print the tensor you will see :

```
0 3 3
0 0 0
```

28.3.4 Set values method

If you want to set multiple values easily, you can use the method `n2d2.Tensor.set_values()`

```
tensor.set_values([[1,2,3], [4,5,6]])
```

If you print the tensor you will see :

```
1 2 3
4 5 6
```

28.4 Numpy

28.4.1 To Numpy

You can create a `numpy.array` using a `n2d2.Tensor` with the class method : `n2d2.Tensor.to_numpy()`

```
tensor = n2d2.Tensor([2, 3])
np_array = tensor.to_numpy()
```

This will create the following tensor :

```
0 0 0
0 0 0
```

By default the `numpy.array` doesn't create a memory copy meaning that if you want to manipulate a `n2d2.Tensor` you can use the numpy library.

```
np_array[0] = 1
print(tensor)
```

```
1 1 1
0 0 0
```

Note: If you do not want to create a memory copy, you should set the parameter `copy=True`.

```
np_array = tensor.to_numpy(copy=True)
```

28.4.2 From Numpy

You can create a `n2d2.Tensor` using a `numpy.array` with the class method : `n2d2.Tensor.from_numpy()`

```
np_array = numpy.array([[1,2,3], [4,5,6]])
tensor = n2d2.Tensor.from_numpy(np_array)
```

This will create the following tensor :

```
1 2 3
4 5 6
```

Note: You cannot create a `n2d2.Tensor` from a `numpy.array` without a memory copy because Tensor require a contiguous memory space which is not required for an array.

28.5 CUDA Tensor

You can store your tensor with CPU or GPU (using `CUDA`). By default, n2d2 creates a CPU tensor.

If you want to create a `CUDA` Tensor you can do so by setting the parameter `cuda` to True in the constructor

```
tensor = n2d2.Tensor([2,3], cuda=True)
```

You can switch from CPU to GPU at anytime :

```
tensor.cpu()  # Converting to a CPU tensor
tensor.cuda() # Converting to a CUDA tensor
```

When working on a `CUDA` tensor you have to understand that they are stored in two different places.

The host and the device. The device is the GPU. The host correspond to your interface with the tensor that exists in the GPU. You cannot access the device directly, the GPU don't have input/output functions.

This is why you have two methods to synchronized these two versions (`n2d2.Tensor.htod()` and `n2d2.Tensor.dtoh()`).

Synchronizing the device and the host can be an important overhead, it is recommended to compute everything on the device and to synchronize the host at the end.

28.5.1 Synchronization example

Let's consider the following CUDA Tensor :

```
t = n2d2.Tensor([2, 2], cuda=True)
```

```
0 0
0 0
```

We set the following values :

```
t.set_values([[1, 2], [3, 4]])
```

```
1 2
3 4
```

Then we will synchronized the device with the host. this mean that we send the values to the GPU.

```
t.htod()
```

```
1 2
3 4
```

As you can see, nothing change when printing the tensor. We have updated the GPU with the new values. Now let's change the values stored in the tensor :

```
t.set_values([[2, 3], [4, 5]])
```

```
2 3
4 5
```

When printing the tensor we see the new values we just set. Now let's synchronize the host with the device !

```
t.dtoh()
```

```
1 2
3 4
```

As you can see when printing the tensor, we now have the old values of the tensor.

CHAPTER

TWENTYNINE

INTEROPERABILITY

In this section, we will present how you can use n2d2 with other python framework.

29.1 Keras *[experimental feature]*

29.1.1 Presentation

The Keras interoperability allow you to train a model using the N2D2 backend with the TensorFlow/Keras frontend.

The interoperability consist of a wrapper around the N2D2 Network.

In order to integrate N2D2 into the Keras environment, we run TensorFlow in eager mode.

29.1.2 Documentation

Changing the optimizer

> **Warning:** Due to the implementation, n2d2 parameters are not visible to `Keras` and thus cannot be optimized by a `Keras` optimizer.

When compiling the `keras_to_n2d2.CustomSequential`, you can pass an `n2d2.solver.Solver` object to the parameter *optimizer*. This will change the method used to optimize the parameters.

```
model.summary() # Use the default SGD solver.
model.compile(loss="categorical_crossentropy", optimizer=n2d2.solver.Adam(), metrics=[
↪"accuracy"])
model.summary() # Use the newly defined Adam solver.
```

29.1.3 Example

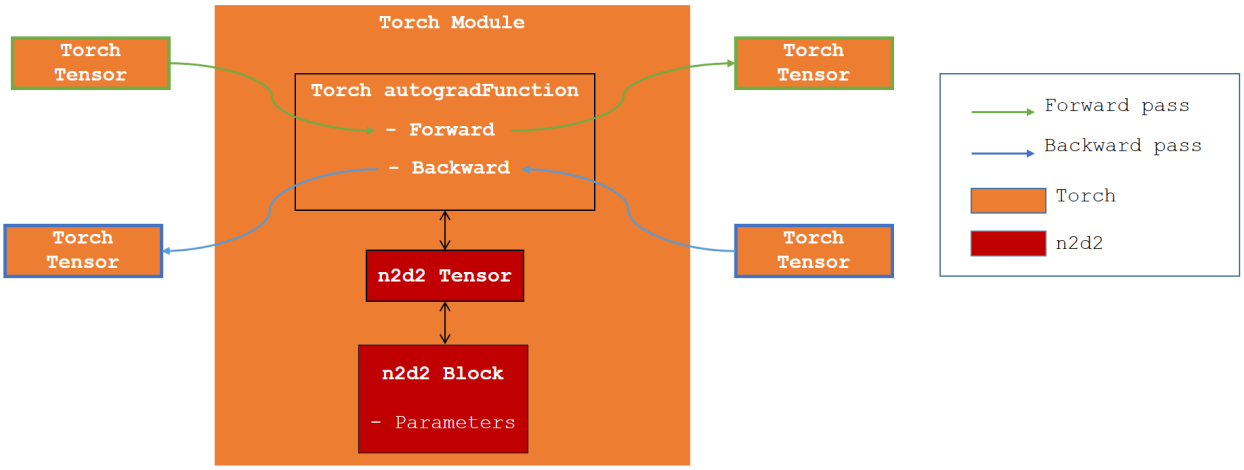
See the *keras example* section.

29.2 PyTorch *[experimental feature]*

29.2.1 Presentation

The PyTorch interoperability allow you to run an n2d2 model by using the Torch functions.

The interoperability consist of a wrapper around the N2D2 Network. We created an autograd function which on `Forward` call the n2d2 `Propagate` method and on `Backward` call the n2d2 `Back Propagate` and `Update` methods.



Warning: Due to the implementation n2d2 parameters are not visible to `Torch` and thus cannot be trained with a torch `Optimizer`.

29.2.2 Tensor conversion

In order to achieve this interoperability, we need to convert Tensor from `Torch` to `n2d2` and vice versa.

`n2d2.Tensor` require a contiguous memory space which is not the case for `Torch`. Thus the conversion `Torch` to `n2d2` require a memory copy. The opposite conversion is done with no memory copy.

If you work with `CUDA` tensor, the conversion `Torch` to `n2d2` is also done with no copy on the GPU (a copy on the host is however required).

29.2.3 Documentation

29.2.4 Example

See the *torch example* section.

CHAPTER

THIRTY

EXPORT

In this section, we will present how to generate an N2D2 export with the Python API. Exports are standalone code which are generated by N2D2.

If you want more specific information about an export please refer to it in the export section.

With the Python API, you can only export an `n2d2.cells.DeepNetCell`.

Once you have trained your model, you can convert your `n2d2.cells.Sequence` into a `n2d2.cells.DeepNetCell` with the method `n2d2.cells.Sequence.to_deepnet_cell()`.

If you have used another way to train your model such as the interoperability with Keras or PyTorch, you can retrieve the `n2d2.cells.DeepNetCell` with the appropriate `get_deepnet_cell` method.

Warning: When using `interoperability`, you do not associate a `n2d2.provider.DataProvider` to the `n2d2.cells.DeepNetCell`. So if you want to calibrate your network, you need ot specify a data provider otherwise N2D2 will fail to generate the export.

30.1 Listing available cells for an export

If you want to get the list available cells for an export you can use the function `n2d2.export.list_exportable_cell()`.

30.2 Export C

30.2.1 Exportable cells

30.2.2 Documentation

30.2.3 Example

```
n2d2.export.export_c(
    DEEPNET_CELL,
    nb_bits=8,
    export_nb_stimuli_max=-1,
    calibration=-1)
```

30.2.4 Frequently asked question

Scaling and ElemWise are the only layers supported in Fixed-point scaling mode

If you try to export an untrained model to C in *int8*, you may come across this error :

```
RuntimeError: Scaling and ElemWise are the only layers supported in Fixed-point scaling␣
↪mode.
```

This is due to the weights parameter being initialized with high value. If you look at the log you may see warning lines like this :

```
Scaling (8.78614) > 1 for layer "model/dense/MatMul:0" is not supported with Single/
↪Double-shift scaling. Falling back to Fixed-point scaling for this layer.
```

Training the model before exporting it will probably solve this issue.

30.3 Export CPP

30.3.1 Exportable cells

30.3.2 Documentation

30.3.3 Example

```
n2d2.export.export_cpp(
    DEEPNET_CELL,
    nb_bits=8,
    export_nb_stimuli_max=-1,
    calibration=-1)
```

30.4 Export CPP TensorRT

30.4.1 Exportable cells

30.4.2 Documentation

30.4.3 Example

```
n2d2.export.export_tensor_rt(DEEPNET_CELL)
```

CHAPTER

THIRTYONE

EXAMPLE

You will find here a list of examples on how to use the Python API.

If you find an example not up to date, please consider leaving an issue here : https://github.com/CEA-LIST/N2D2/issues.

31.1 Data augmentation

In this example, we will see how to use `n2d2.provider.DataProvider` and `n2d2.transform.Transformation` to load data and do some data augmentation.

You can find the full python script here `data_augmentation.py`.

31.1.1 Preliminary

For this tutorial, we will use n2d2 for data augmentation, and numpy and matplotlib for the visualization.

We will create a method plot_tensor to save the generated images from an `n2d2.Tensor`

```
import n2d2
import matplotlib.pyplot as plt

def plot_tensor(tensor, path):
    plt.imshow(tensor[0,0,:], cmap='gray', vmin=0, vmax=255)
    plt.savefig(path)
```

31.1.2 Loading data

We will begin by creating a `n2d2.database.MNIST` driver to load the MNIST dataset. We will then create a provider to get the images, we use a batch size of 1 to get only one image.

```
database = n2d2.database.MNIST(data_path="/local/DATABASE/mnist", validation=0.1)
provider = n2d2.provider.DataProvider(database, [28, 28, 1], batch_size=1)
```

You can get the number of data per partition by using the method `n2d2.database.Database.get_partition_summary()` which will print the paritionement of data.

```
database.get_partition_summary()
```

Output :

```
Number of stimuli : 70000
Learn         : 54000 stimuli (77.14%)
Test          : 10000 stimuli (14.29%)
Validation    : 6000 stimuli (8.57%)
Unpartitioned : 0 stimuli (0.0%)
```

To select which partition you want to read from you need to use the method `n2d2.provider.DataProvider.set_partition()`

To read data from a `n2d2.provider.DataProvider` you can use multiple methods.

You can use the methods `n2d2.provider.DataProvider.read_batch()` or `n2d2.provider.DataProvider.read_random_batch()`.

Note: Since `n2d2.provider.DataProvider` is an *iterable*, so you can use the `next()` function or a for loop !

```python
# for loop example
for data in provider:
    pass
# next example
data = next(provider)
```

For this tutorial we will use `n2d2.provider.DataProvider.read_batch()` !

With this code we will get the first image and plot it :

```python
image = provider.read_batch(idx=0).to_numpy() * 255
plot_tensor(image, "first_stimuli.png")
```

31.1.3 Data augmentation

To do data augmentation with N2D2 we use `n2d2.transform.Transformation`. You can add transformation to provider with the method `n2d2.provider.DataProvider.add_on_the_fly_transformation()` and `n2d2.provider.DataProvider.add_transformation()`.

Warning: Since we already loaded the first image the method `n2d2.provider.DataProvider.add_transformation()` would not apply the transformation to the image.

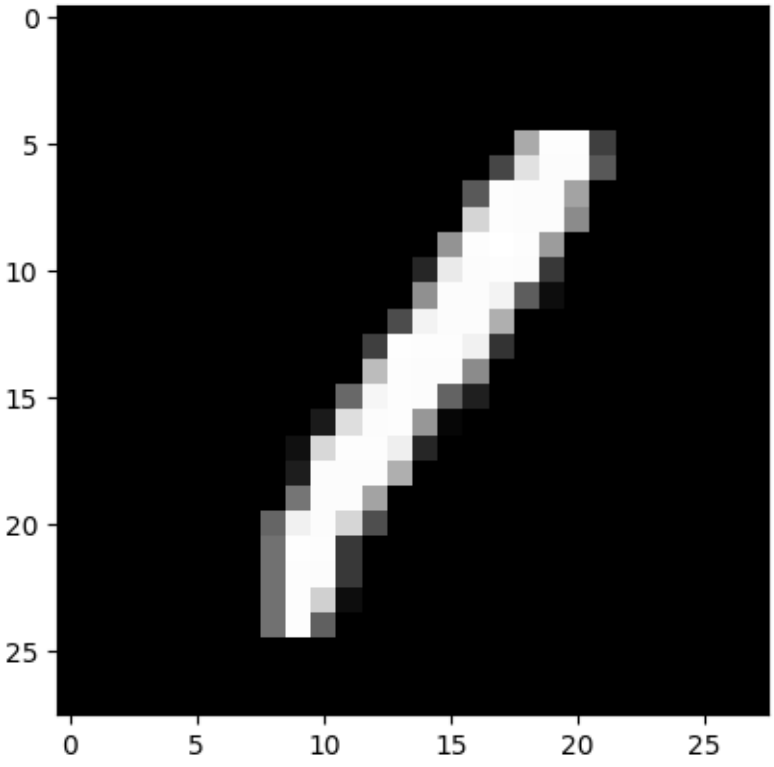
By using the transformation `n2d2.transform.Flip` we will flip vertically our image.

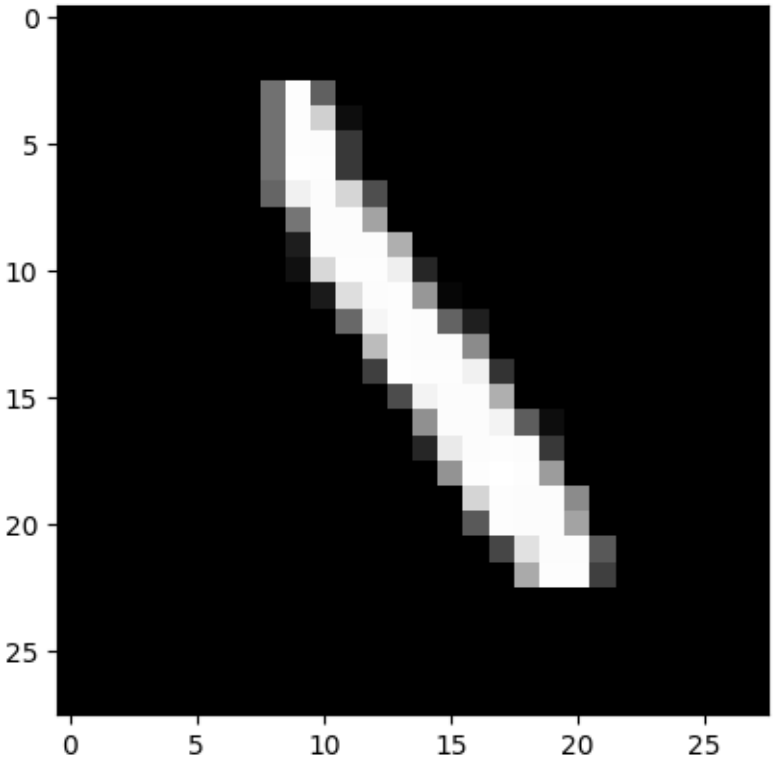
```python
provider.add_on_the_fly_transformation(n2d2.transform.Flip(vertical_flip=True))

image = provider.read_batch(idx=0).to_numpy() * 255
plot_tensor(image, "first_stimuli_fliped.png")
```

We will negate the first transformation with another `n2d2.transform.Flip` which we will add with the method `n2d2.provider.DataProvider.add_transformation()`.

```python
# negating the first transformation with another one
provider.add_transformation(n2d2.transform.Flip(vertical_flip=True))
```

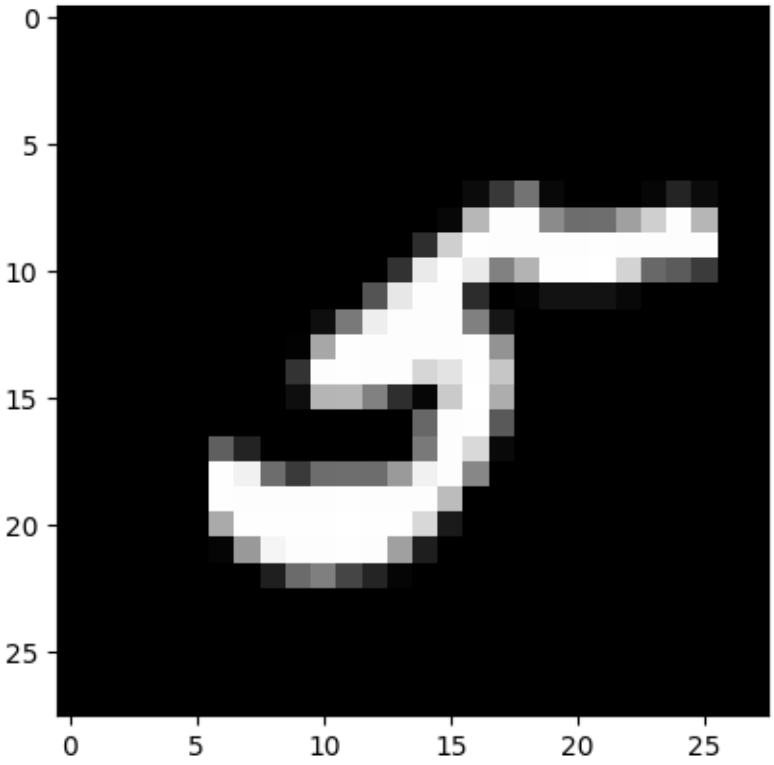
(continues on next page)


0
5
10
15
20
25
0
5
10
15
20
25


0
5
10
15
20
25
0
5
10
15
20
25

(continued from previous page)

```
image = provider.read_batch(idx=1).to_numpy() * 255
plot_tensor(image, "second_stimuli.png")
```



31.1.4 Getting labels

To get the labels

```
print("Second stimuli label : ", provider.get_labels()[0])
```

Output :

```
Second stimuli label : 5
```

31.2 Performance analysis

In this example we will create a simple neural network model, train it and use analysis tools to see the performances. You can see the full script of this example here : `performance_analysis.py`.

31.2.1 Use-case presentation

We propose to recognize traffic signs, for an advanced driver-assistance systems (ADAS). The traffic signs are already segmented and extracted from images, taken from a front-view camera embedded in a car.

To build our classifier, we will use the `German Traffic Sign Benchmark (GTSRB)` (https://benchmark.ini.rub.de/) a multi-class, single-image classification challenge held at the International Joint Conference on Neural Networks (IJCNN) 2011.

This benchmark has the following properties:

- Single-image, multi-class classification problem;
- More than 40 classes;
- More than 50,000 images in total;
- Large, lifelike database.

31.2.2 Creation of the network

Defining the inputs of the Neural Network

First of all, if you have CUDA available, you can enable it with the following line :

```
n2d2.global_variables.default_model = "Frame_CUDA"
```

The default model is `Frame`.

Once this is done, you can load the database by using the appropriate driver `n2d2.database.GTSRB`. We set 20% of the data to be used for validation. To feed data to the network, you need to create a `n2d2.provider.DataProvider` this class will define the input fo neural network.

```
db = n2d2.database.GTSRB(0.2)
db.load(data_path) # Enter the path of your database !
provider = n2d2.provider.DataProvider(db, [29, 29, 1], batch_size=BATCH_SIZE)
```

You can apply pre-processing to the data with `n2d2.transform.Transformation` objects. We have set the size of the input images to be 29x29 pixels with 1 channel when declaring the `n2d2.provider.DataProvider`. But the dataset is composed of image of size comprised from 15x15 to 250x250 pixels with 3 channel (R,G,B). We will add a `n2d2.transform.Rescale` to rescale the size of the image to be 29x29 pixels. The other transformation `n2d2.transform.ChannelExtraction` will set the number of channel to 1 by creating images in nuance of grey.

```
provider.add_transformation(n2d2.transform.ChannelExtraction('Gray'))
provider.add_transformation(n2d2.transform.Rescale(width=29, height=29))
```

Defining the neural network

Now that we have defined the inputs, we can declare the neural network. We will create a network inspired from the well-known LeNet network.

Before we define the network, we will create default configuration for the different type of layer with `n2d2.ConfigSection`. This will allow us to create cells more concisely. `n2d2.ConfigSection` are used like python dictionary.

```
solver_config = ConfigSection(
    learning_rate=0.01,
    momentum=0.9,
    decay=0.0005,
    learning_rate_decay=0.993)

fc_config   = ConfigSection(weights_filler=Xavier(),
                            no_bias=True,
                            weights_solver=SGD(**solver_config))
conv_config = ConfigSection(activation=Rectifier(),
                            weights_filler=Xavier(),
                            weights_solver=SGD(**solver_config),
                            no_bias=True)
```

For the ReLU activation function to be effective, the weights must be initialized carefully, in order to avoid dead units that would be stuck in the $]\infty,0]$ output range before the ReLU function. In N2D2, one can use a custom WeightsFiller for the weights initialization. This is why we will use the `n2d2.filler.Xavier` algorithm to fill the weights of the different cells.

To define the network, we will use `n2d2.cells.Sequence` that take a list of `n2d2.nn.NeuralNetworkCell`.

```
model = n2d2.cells.Sequence([
    Conv(1, 32, [4, 4], **conv_config),
    Pool2d([2, 2], stride_dims=[2, 2], Pooling='Max'),
    Conv(32, 48, [5, 5], mapping=conv2_mapping, **conv_config),
    Pool2d([3, 3], stride_dims=[3, 3], Pooling='Max'),
    Fc(48*3*3, 200, activation=Rectifier(), **fc_config),
    Dropout(),
    Fc(200, 43, activation=Linear(), **fc_config),
    Softmax(with_loss=True)
])
```

Note that in LeNet, the conv2 layer is not fully connected to the Pooling layer. In n2d2, a custom mapping can be defined for each input connection. We can do this with the mapping argument by passing a `n2d2.Tensor`. The connection of n-th output map to the inputs is defined by the n-th column of the matrix below, where the rows correspond to the inputs.

```
conv2_mapping=n2d2.Tensor([32, 48], datatype="bool")
conv2_mapping.set_values([
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
[1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
[0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
[0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
[0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
[0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
[0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
[0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0,␣
```

(continues on next page)

(continued from previous page)

```
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,␣
↪1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1]])
```

After creating our network, we will create a `n2d2.target.Target`. This object will compute gradient and log training information that we will display later.

```
target = n2d2.target.Score(provider)
```

The `n2d2.application.CrossEntropyClassifier` deals with the output of the neural network, it computes the loss and propagates the gradient through the network.

Training the neural network

Once the neural network is defined, you can train it with the following loop :

```
for epoch in range(NB_EPOCH):
    print("\n\nEpoch : ", epoch)
    print("### Learning ###")
    provider.set_partition("Learn")
    model.learn()
    provider.set_reading_randomly(True)
    for stimuli in provider:
        output = model(stimuli)
        loss = target(output)
        loss.back_propagate()
        loss.update()
        print("Batch number : " + str(provider.batch_number()) + ", loss: " + "{0:.3f}".
↪format(loss[0]), end='\r')
    print("\n### Validation ###")
    target.clear_success()
    provider.set_partition('Validation')
    model.test()
    for stimuli in provider:
        x = model(stimuli)
        x = target(x)
        print("Batch number : " + str(provider.batch_number()) + ", val success: "
                + "{0:.2f}".format(100 * target.get_average_success()) + "%", end='\r')
```

Once the learning phase is ended, you can test your network with the following loop :

```
print("\n### Testing ###")
provider.set_partition('Test')
model.test()
for stimuli in provider:
    x = model(stimuli)
    x = target(x)
    print("Batch number : " + str(provider.batch_number()) + ", test success: "
        + "{0:.2f}".format(100 * target.get_average_success()) + "%", end='\r')
print("\n")
```

31.2.3 Performance analysis tools

Once the training is done, you can log various statistics to analyze the performance of your network.

If you have done the testing loop you can use the following line to see the results :

```
# save a confusion matrix
target.log_confusion_matrix("vis_GTSRB")
# save a graph of the loss and the validation score as a function of the number of steps
target.log_success("vis_GTSRB")
```

These methods will create images in a folder with the name of your Target. In this folder, you will find the confusion matrix :

And the training curve :

If you want to visualize the performance analysis of your neural network you can use the following line :

```
# save computational stats on the network
target.log_stats("vis_GTSRB")
```

This will generate the following statistics :

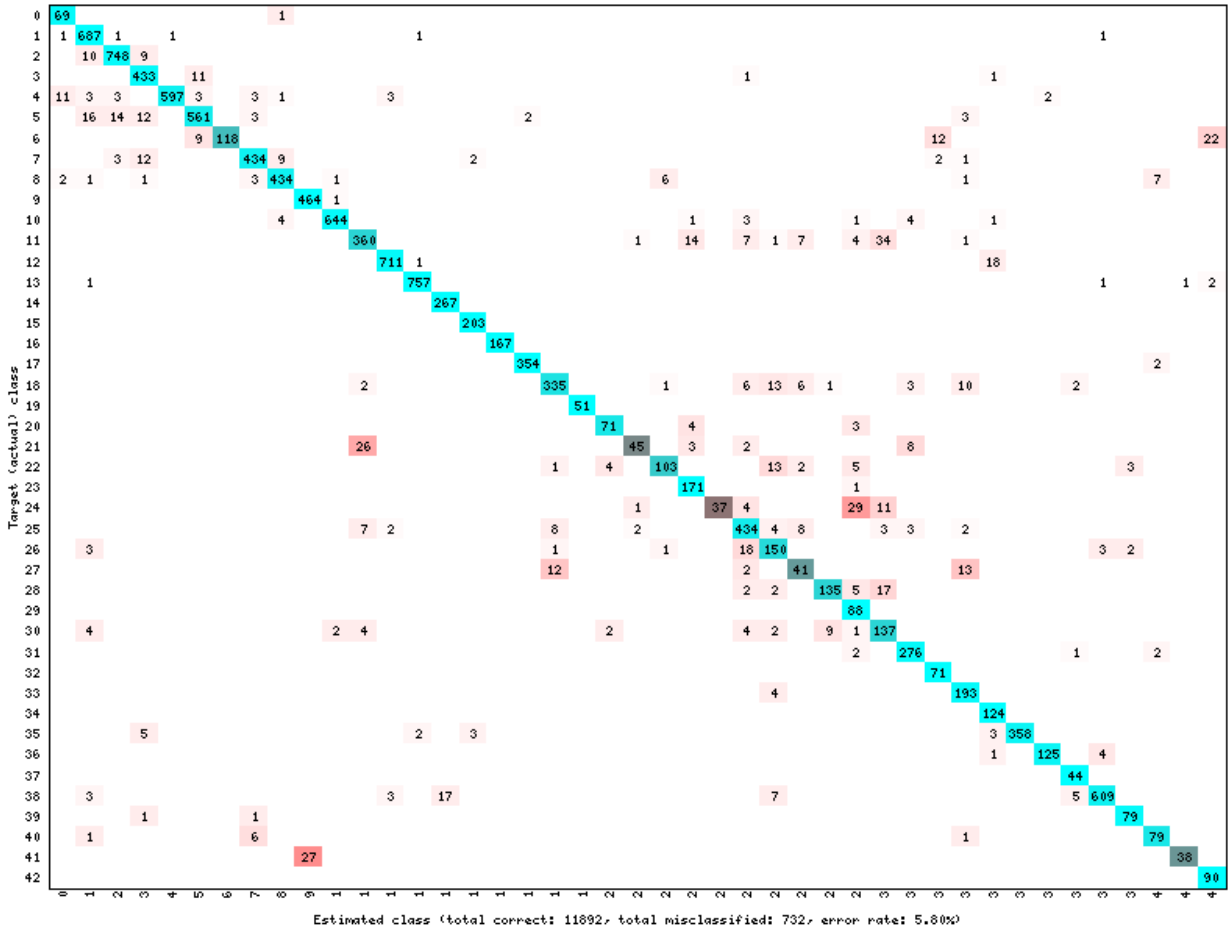
- Number of Multiply-ACcumulate (MAC) operations per layers;
- Number of parameters per layers;
- Memory footprint per layers.

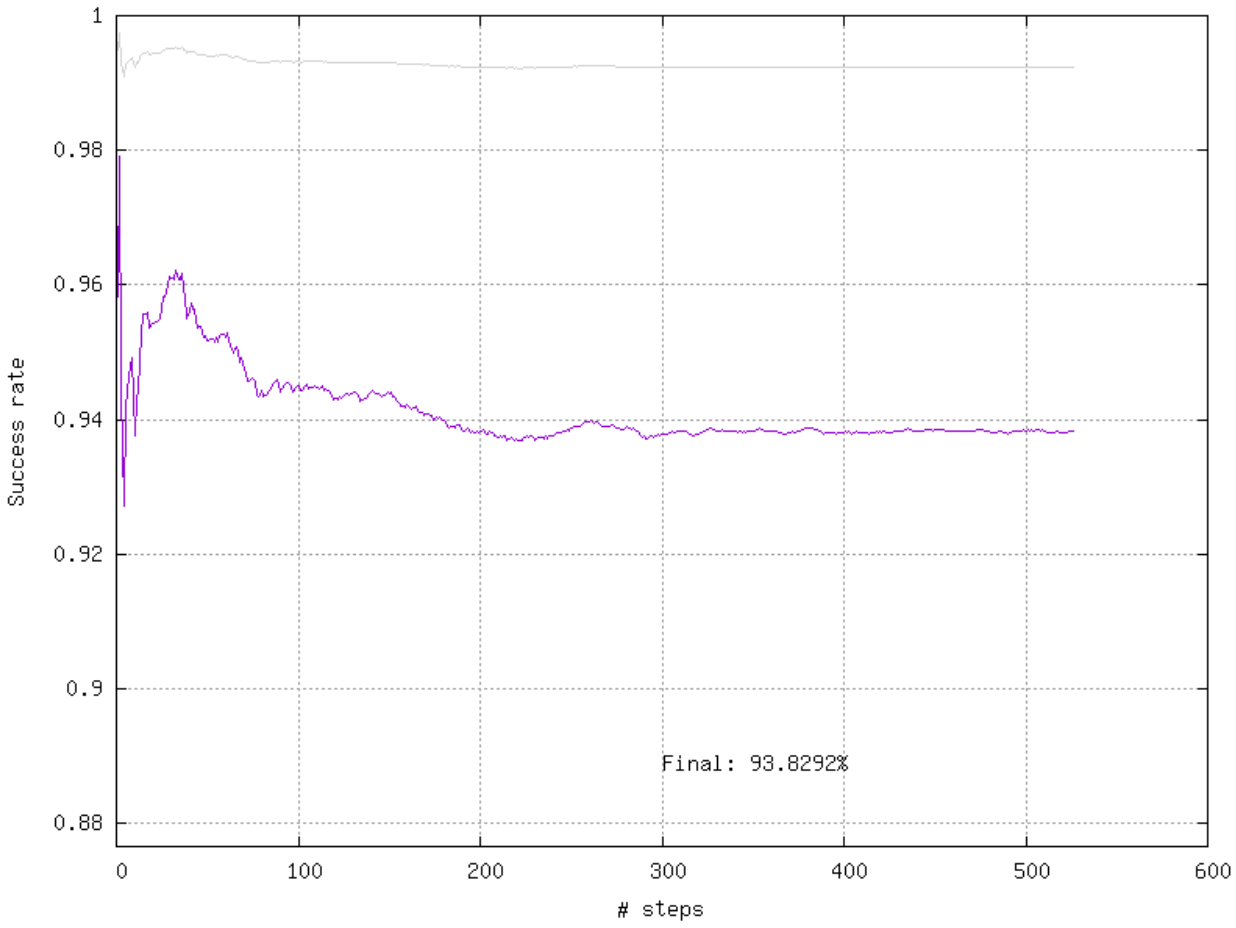
These data are available with a logarithm scale or a relative one.

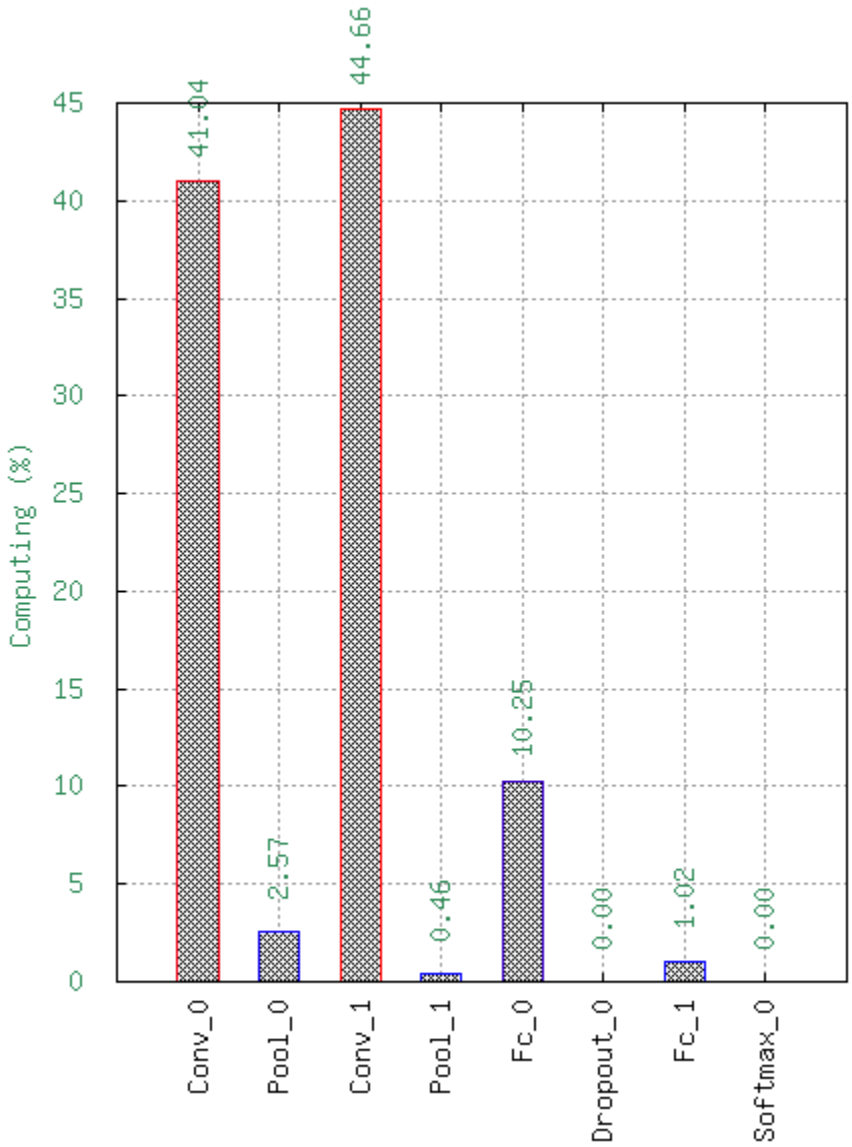
31.3 Load from ONNX

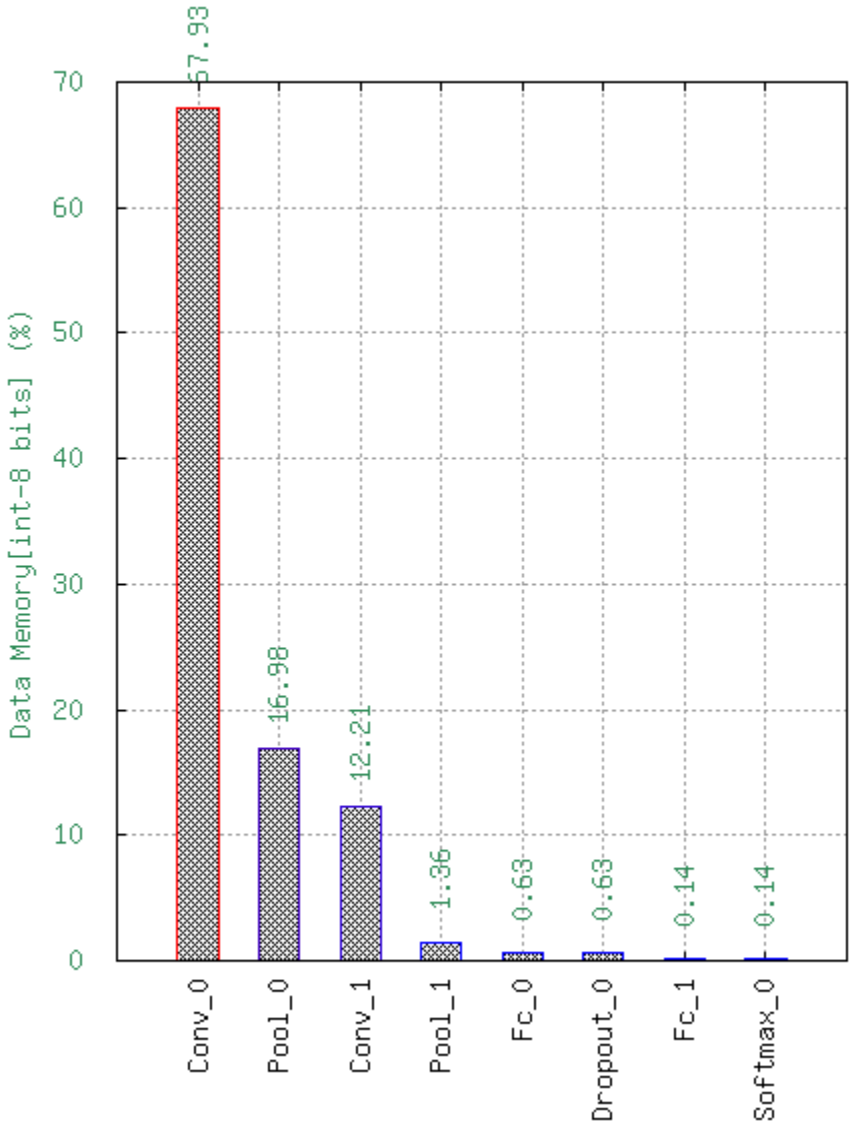
In this example, we will see step by step how to load a model from ONNX.

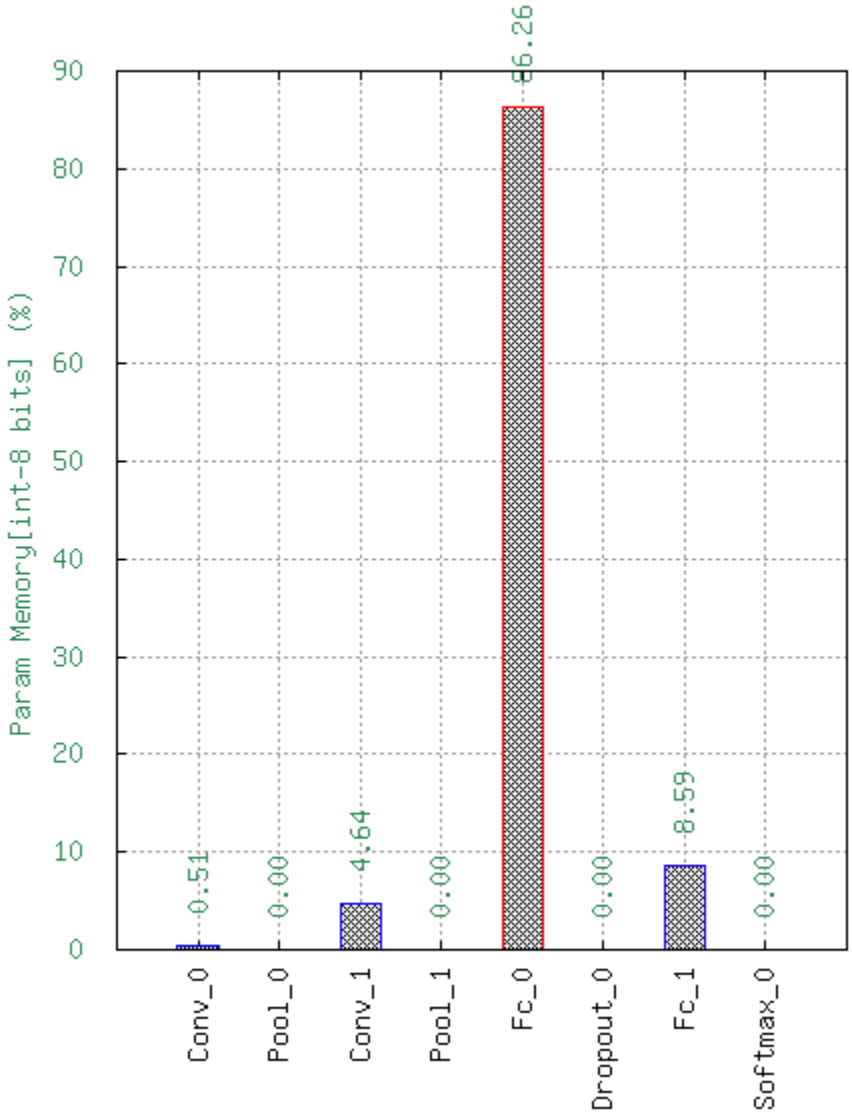
You can find the full python script here `lenet_onnx.py` with the associated onnx file here `LeNet.onnx`.

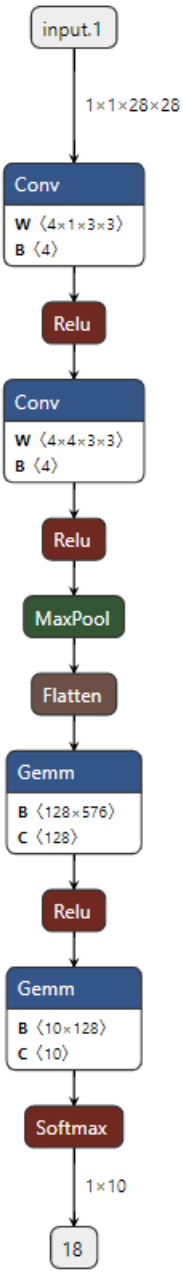

Target (actual) class
Estimated class (total correct: 11892, total misclassified: 732, error rate: 5.80%)


1
0.98
0.96
0.94
0.92
0.9
0.88
Success rate
Final: 93.8292%
0
100
200
300
400
500
600
steps


Computing (%)
45
40
35
30
25
20
15
10
5
0
41.04
2.57
44.66
0.46
10.25
0.00
1.02
0.00
Conv_0
Pool_0
Conv_1
Pool_1
Fc_0
Dropout_0
Fc_1
Softmax_0


Data Memory[int-8 bits] (%)
0
10
20
30
40
50
60
70
67.93
16.98
12.21
1.36
0.63
0.63
0.14
0.14
Conv_0
Pool_0
Conv_1
Pool_1
Fc_0
Dropout_0
Fc_1
Softmax_0


Param Memory[int-8 bits] (%)
90
80
70
60
50
40
30
20
10
0
0.51
0.00
4.64
0.00
86.26
0.00
8.59
0.00
Conv_0
Pool_0
Conv_1
Pool_1
Fc_0
Dropout_0
Fc_1
Softmax_0


input.1
1×1×28×28
Conv
W 〈4×1×3×3〉
B 〈4〉
Relu
Conv
W 〈4×4×3×3〉
B 〈4〉
Relu
MaxPool
Flatten
Gemm
B 〈128×576〉
C 〈128〉
Relu
Gemm
B 〈10×128〉
C 〈10〉
Softmax
1×10
18

31.3.1 Loading an ONNX

Before loading the ONNX file, we need to create an `n2d2.database.MNIST` and `n2d2.provider.DataProvider` objects.

```
# Change default model to run with CUDA
n2d2.global_variables.default_model = "Frame_CUDA"
# Change cuda device (default 0)
n2d2.global_variables.cuda_device = args.device
nb_epochs = args.epochs
batch_size = 54

print("\n### Create database ###")
database = n2d2.database.MNIST(data_path=args.data_path, validation=0.1)

print("\n### Create Provider ###")
provider = n2d2.provider.DataProvider(database, [28, 28, 1], batch_size=batch_size)
provider.add_transformation(n2d2.transform.Rescale(width=28, height=28))
print(provider)
```

Once this is done, you can load your ONNX file with one line :

```
print("\n### Loading Model ###")
model = n2d2.cells.DeepNetCell.load_from_ONNX(provider, args.onnx)
print(model)
```

You should observe a verbose output of the loaded ONNX model :

```
'DeepNetCell_0' DeepNetCell(
        (0): '9' Conv(Frame_CUDA<float>)(nb_inputs=1, nb_outputs=4, kernel_dims=[3, 3],␣
↪sub_sample_dims=[1, 1], stride_dims=[1, 1], padding_dims=[0, 0], dilation_dims=[1, 1]␣
↪| back_propagate=True, no_bias=False, outputs_remap=, weights_export_flip=False,␣
↪weights_export_format=OCHW, activation=Rectifier(clipping=0.0, leak_slope=0.0,␣
↪quantizer=None), weights_solver=SGD(clamping=, decay=0.0, iteration_size=1, learning_
↪rate=0.01, learning_rate_decay=0.1, learning_rate_policy=None, learning_rate_step_
↪size=1, max_iterations=0, min_decay=0.0, momentum=0.0, polyak_momentum=True, power=0.0,
↪ warm_up_duration=0, warm_up_lr_frac=0.25), bias_solver=SGD(clamping=, decay=0.0,␣
↪iteration_size=1, learning_rate=0.01, learning_rate_decay=0.1, learning_rate_
↪policy=None, learning_rate_step_size=1, max_iterations=0, min_decay=0.0, momentum=0.0,␣
↪polyak_momentum=True, power=0.0, warm_up_duration=0, warm_up_lr_frac=0.25), weights_
↪filler=Normal(mean=0.0, std_dev=0.05), bias_filler=Normal(mean=0.0, std_dev=0.05),␣
↪quantizer=None)
        (1): '11' Conv(Frame_CUDA<float>)(nb_inputs=4, nb_outputs=4, kernel_dims=[3, 3],␣
↪sub_sample_dims=[1, 1], stride_dims=[1, 1], padding_dims=[0, 0], dilation_dims=[1, 1]␣
↪| back_propagate=True, no_bias=False, outputs_remap=, weights_export_flip=False,␣
↪weights_export_format=OCHW, activation=Rectifier(clipping=0.0, leak_slope=0.0,␣
↪quantizer=None), weights_solver=SGD(clamping=, decay=0.0, iteration_size=1, learning_
↪rate=0.01, learning_rate_decay=0.1, learning_rate_policy=None, learning_rate_step_
↪size=1, max_iterations=0, min_decay=0.0, momentum=0.0, polyak_momentum=True, power=0.0,
↪ warm_up_duration=0, warm_up_lr_frac=0.25), bias_solver=SGD(clamping=, decay=0.0,␣
↪iteration_size=1, learning_rate=0.01, learning_rate_decay=0.1, learning_rate_
↪policy=None, learning_rate_step_size=1, max_iterations=0, min_decay=0.0, momentum=0.0,␣
↪polyak_momentum=True, power=0.0, warm_up_duration=0, warm_up_lr_frac=0.25), weights_
↪filler=Normal(mean=0.0, std_dev=0.05), bias_filler=Normal(mean=0.0, std_dev=0.05),␣
```

(continues on next page)

(continued from previous page)

```
↪quantizer=None)(['9'])
        (2): '13' Pool(Frame_CUDA<float>)(pool_dims=[2, 2], stride_dims=[2, 2], padding_
↪dims=[0, 0], pooling=Pooling.Max | activation=None)(['11'])
        (3): '15' Fc(Frame_CUDA<float>)(nb_inputs=576, nb_outputs=128 | back_
↪propagate=True, no_bias=False, normalize=False, outputs_remap=, weights_export_
↪format=OC, activation=Rectifier(clipping=0.0, leak_slope=0.0, quantizer=None), weights_
↪solver=SGD(clamping=, decay=0.0, iteration_size=1, learning_rate=0.01, learning_rate_
↪decay=0.1, learning_rate_policy=None, learning_rate_step_size=1, max_iterations=0, min_
↪decay=0.0, momentum=0.0, polyak_momentum=True, power=0.0, warm_up_duration=0, warm_up_
↪lr_frac=0.25), bias_solver=SGD(clamping=, decay=0.0, iteration_size=1, learning_rate=0.
↪01, learning_rate_decay=0.1, learning_rate_policy=None, learning_rate_step_size=1, max_
↪iterations=0, min_decay=0.0, momentum=0.0, polyak_momentum=True, power=0.0, warm_up_
↪duration=0, warm_up_lr_frac=0.25), weights_filler=Normal(mean=0.0, std_dev=0.05), bias_
↪filler=Normal(mean=0.0, std_dev=0.05), quantizer=None)(['13'])
        (4): '17' Fc(Frame_CUDA<float>)(nb_inputs=128, nb_outputs=10 | back_
↪propagate=True, no_bias=False, normalize=False, outputs_remap=, weights_export_
↪format=OC, activation=Linear(clipping=0.0, quantizer=None), weights_
↪solver=SGD(clamping=, decay=0.0, iteration_size=1, learning_rate=0.01, learning_rate_
↪decay=0.1, learning_rate_policy=None, learning_rate_step_size=1, max_iterations=0, min_
↪decay=0.0, momentum=0.0, polyak_momentum=True, power=0.0, warm_up_duration=0, warm_up_
↪lr_frac=0.25), bias_solver=SGD(clamping=, decay=0.0, iteration_size=1, learning_rate=0.
↪01, learning_rate_decay=0.1, learning_rate_policy=None, learning_rate_step_size=1, max_
↪iterations=0, min_decay=0.0, momentum=0.0, polyak_momentum=True, power=0.0, warm_up_
↪duration=0, warm_up_lr_frac=0.25), weights_filler=Normal(mean=0.0, std_dev=0.05), bias_
↪filler=Normal(mean=0.0, std_dev=0.05), quantizer=None)(['15'])
        (5): '18' Softmax(Frame_CUDA<float>)(with_loss=True, group_size=0 |␣
↪activation=None)(['17'])
)
```

The model has been exported successfully !

31.3.2 Training and exporting the model

You can now do what you want with your imported model, like training it :

```
model.fit(learn_epoch=nb_epochs, valid_metric='Accuracy')
model.run_test()
```

And even exporting it to CPP in int 8 !

Warning: Don't forget to remove the softmax layer first because N2D2 does not export this layer for the CPP export.

```
model.remove("18") # removing Softmax layer before export !
n2d2.export_cpp(model, nb_bits=8, calibration=1)
```

31.4 Graph manipulation

In this example we will see :

- How the N2D2 graph is generated;
- How to draw the graph;
- How to concatenate two Sequences;
- How to get the output of a specific cell;
- How to save only a certain part of the graph.

You can see the full script of this example here : `graph_example.py`.

For the following examples we will use the following objects :

```
fc1 = n2d2.cells.Fc(28*28, 50, activation=n2d2.activation.Rectifier())
fc2 = n2d2.cells.Fc(50, 10)
```

31.4.1 Printing n2d2 graph

The python API possess different vebosity level (default=`detailed`).

Short representation: only with compulsory constructor arguments

```
n2d2.global_variables.verbosity = n2d2.global_variables.Verbosity.short
print(fc1)
print(fc2)
```

Output :

```
'Fc_0' Fc(Frame<float>)(nb_inputs=784, nb_outputs=50)
'Fc_1' Fc(Frame<float>)(nb_inputs=50, nb_outputs=10)
```

Verbose representation: show graph and every arguments

```
n2d2.global_variables.verbosity = n2d2.global_variables.Verbosity.detailed
print(fc1)
print(fc2)
```

Output :

```
'Fc_0' Fc(Frame<float>)(nb_inputs=784, nb_outputs=50 | back_propagate=True, drop_
↪connect=1.0, no_bias=False, normalize=False, outputs_remap=, weights_export_format=OC,␣
↪activation=Rectifier(clipping=0.0, leak_slope=0.0, quantizer=None), weights_
↪solver=SGD(clamping=, decay=0.0, iteration_size=1, learning_rate=0.01, learning_rate_
↪decay=0.1, learning_rate_policy=None, learning_rate_step_size=1, max_iterations=0, min_
↪decay=0.0, momentum=0.0, polyak_momentum=True, power=0.0, warm_up_duration=0, warm_up_
↪lr_frac=0.25), bias_solver=SGD(clamping=, decay=0.0, iteration_size=1, learning_rate=0.
↪01, learning_rate_decay=0.1, learning_rate_policy=None, learning_rate_step_size=1, max_
↪iterations=0, min_decay=0.0, momentum=0.0, polyak_momentum=True, power=0.0, warm_up_
↪duration=0, warm_up_lr_frac=0.25), weights_filler=Normal(mean=0.0, std_dev=0.05), bias_
↪filler=Normal(mean=0.0, std_dev=0.05), quantizer=None)
'Fc_1' Fc(Frame<float>)(nb_inputs=50, nb_outputs=10 | back_propagate=True, drop_
```

(continues on next page)

(continued from previous page)

```
↪connect=1.0, no_bias=False, normalize=False, outputs_remap=, weights_export_format=OC,␣
↪activation=None, weights_solver=SGD(clamping=, decay=0.0, iteration_size=1, learning_
↪rate=0.01, learning_rate_decay=0.1, learning_rate_policy=None, learning_rate_step_
↪size=1, max_iterations=0, min_decay=0.0, momentum=0.0, polyak_momentum=True, power=0.0,
↪ warm_up_duration=0, warm_up_lr_frac=0.25), bias_solver=SGD(clamping=, decay=0.0,␣
↪iteration_size=1, learning_rate=0.01, learning_rate_decay=0.1, learning_rate_
↪policy=None, learning_rate_step_size=1, max_iterations=0, min_decay=0.0, momentum=0.0,␣
↪polyak_momentum=True, power=0.0, warm_up_duration=0, warm_up_lr_frac=0.25), weights_
↪filler=Normal(mean=0.0, std_dev=0.05), bias_filler=Normal(mean=0.0, std_dev=0.05),␣
↪quantizer=None)
```

Graph representation: show the object and the cell associated.

Note: Before propagation, no inputs are visible.

```
n2d2.global_variables.verbosity = n2d2.global_variables.Verbosity.graph_only
print(fc1)
print(fc2)
```

Output :

```
'Fc_0' Fc(Frame<float>)
'Fc_1' Fc(Frame<float>)
```

Now if we propagate a tensor to our cells, we will generate the computation graph and we will be able to see the linked cells :

```
x = n2d2.tensor.Tensor(dims=[1, 28, 28], value=0.5)

x = fc1(x)
x = fc2(x)
print(fc1)
print(fc2)
```

Output :

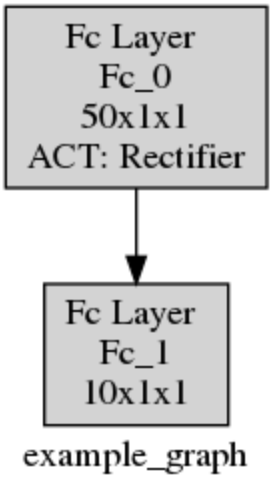
```
'Fc_0' Fc(Frame<float>)(['TensorPlaceholder_0'])
'Fc_1' Fc(Frame<float>)(['Fc_0'])
```

Now we can see the inputs object of each cells !

You can also plot the graph associated to a tensor with the method `n2d2.Tensor.draw_associated_graph()` :

```
x.draw_associated_graph("example_graph")
```

This will generate the following figure :



example_graph

31.4.2 Manipulating Sequences

For this example we will show how you can use n2d2 to encapsulate Sequence.

We will create a LeNet and separate it two parts the extractor and the classifier.

```
from n2d2.cells import Sequence, Conv, Pool2d, Dropout, Fc
from n2d2.activation import Rectifier, Linear

extractor = Sequence([
    Conv(1, 6, kernel_dims=[5, 5]),
    Pool2d(pool_dims=[2, 2], stride_dims=[2, 2], pooling='Max'),
    Conv(6, 16, kernel_dims=[5, 5]),
    Pool2d(pool_dims=[2, 2], stride_dims=[2, 2], pooling='Max'),
    Conv(16, 120, kernel_dims=[5, 5]),
], name="extractor")

classifier = Sequence([
    Fc(120, 84, activation=Rectifier()),
    Dropout(dropout=0.5),
    Fc(84, 10, activation=Linear(), name="last_fully"),
], name="classifier")
```

We can concatenate these two sequences into one :

```
network = Sequence([extractor, classifier])

x = n2d2.Tensor([1,1,32,32], value=0.5)
output = network(x)

print(network)
```

Output

```
'Sequence_0' Sequence(
        (0): 'extractor' Sequence(
                (0): 'Conv_0' Conv(Frame<float>)(['TensorPlaceholder_1'])
```

(continues on next page)

(continued from previous page)

```
                (1): 'Pool2d_0' Pool2d(Frame<float>)(['Conv_0'])
                (2): 'Conv_1' Conv(Frame<float>)(['Pool2d_0'])
                (3): 'Pool2d_1' Pool2d(Frame<float>)(['Conv_1'])
                (4): 'Conv_2' Conv(Frame<float>)(['Pool2d_1'])
        )
        (1): 'classifier' Sequence(
                (0): 'Fc_2' Fc(Frame<float>)(['Conv_2'])
                (1): 'Dropout_0' Dropout(Frame<float>)(['Fc_2'])
                (2): 'last_fully' Fc(Frame<float>)(['Dropout_0'])
        )
)
```

We can also plot the graph :

```
output.draw_associated_graph("full_lenet_graph")
```

We can also easily access the cells inside the encapsulated Sequence

```
first_fully = network["last_fully"]
print("Accessing the first fully connected layer which is encapsulated in a Sequence")
print(first_fully)
```

Output

```
'last_fully' Fc(Frame<float>)(['Dropout_0'])
```

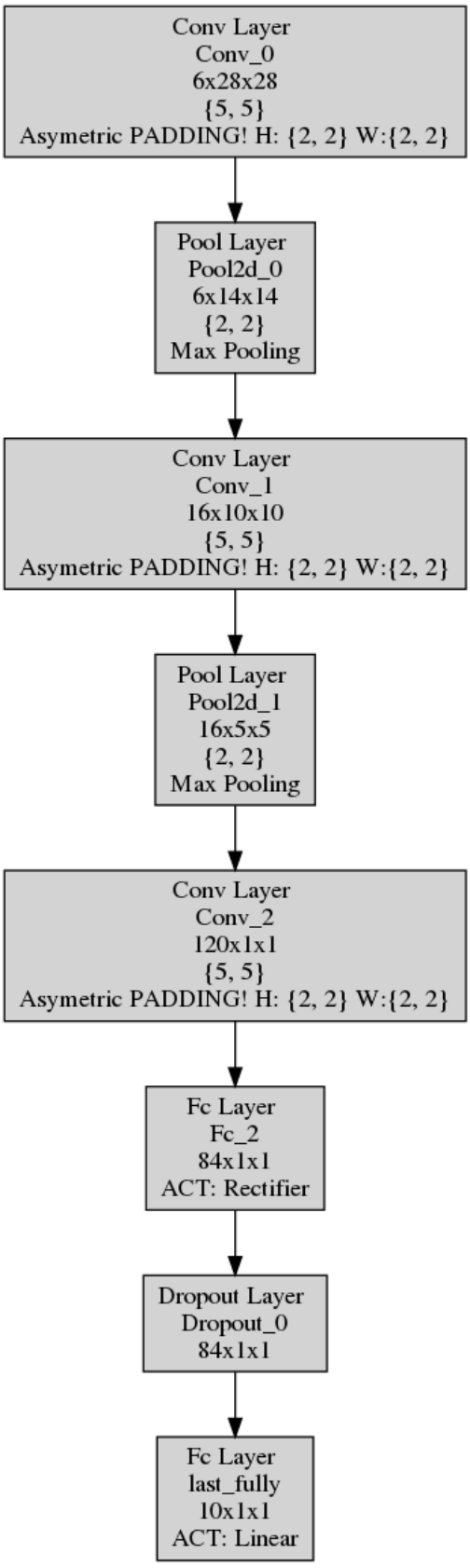
This allow us for example to get the output of any cells after the propagation :

```
print(f"Output of the second fully connected : {first_fully.get_outputs()}")
```

Output

```
Output of the second fully connected : n2d2.Tensor([
[0][0]:
0.0135485
[1]:
0.0359611
[2]:
-0.0285292
[3]:
-0.0732218
[4]:
0.0318365
[5]:
-0.0930403
[6]:
0.0467896
[7]:
-0.108823
[8]:
0.0305202
[9]:
```

(continues on next page)



full_lenet_graph

(continued from previous page)

```
0.0055611
], device=cpu, datatype=f, cell='last_fully')
```

Concatenating `n2d2.cells.Sequence` can be useful if we want for example to only save the parameters of a part of the network.

```
network[0].export_free_parameters("ConvNet_parameters")
```

Output

```
Export to ConvNet_parameters/Conv_0.syntxt
Export to ConvNet_parameters/Conv_0_quant.syntxt
Export to ConvNet_parameters/Pool2d_0.syntxt
Export to ConvNet_parameters/Pool2d_0_quant.syntxt
Export to ConvNet_parameters/Conv_1.syntxt
Export to ConvNet_parameters/Conv_1_quant.syntxt
Export to ConvNet_parameters/Pool2d_1.syntxt
Export to ConvNet_parameters/Pool2d_1_quant.syntxt
Export to ConvNet_parameters/Conv_2.syntxt
Export to ConvNet_parameters/Conv_2_quant.syntxt
```

31.5 Torch interoperability

In this example, we will follow the Torch tutorial : https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html. And run the network with N2D2 instead of Torch.

You can find the full python script here `torch_example.py`.

31.5.1 Example

Firstly, we import the same libraries as in the tutorial plus our `pytorch_to_n2d2` and `n2d2` libraries.

```
import torch
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import n2d2
import pytorch_to_n2d2
```

We then still follow the tutorial and add the code to load the data and we define the Network.

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")


transform = transforms.Compose(
[transforms.ToTensor(),
```

(continues on next page)

(continued from previous page)

```
transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

batch_size = 4

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
        'deer', 'dog', 'frog', 'horse', 'ship', 'truck')



# functions to show an image


def imshow(img, img_path):
        img = img / 2 + 0.5     # unnormalize
        cpu_img = img.cpu()
        npimg = cpu_img.numpy()
        plt.imshow(np.transpose(npimg, (1, 2, 0)))
        plt.savefig(img_path)


class Net(nn.Module):
def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Here we begin to add our code, we intialize the Torch Network and we pass it to the `pytorch_to_n2d2.wrap()` method. This will give us a `torch.nn.Module` which run N2D2 and that we will use instead of the Torch Network.

```
torch_net = Net()
# specify that we want to use CUDA.
n2d2.global_variables.default_model = "Frame_CUDA"
# creating a model which run with N2D2 backend.
net = pytorch_to_n2d2.wrap(torch_net, (batch_size, 3, 32, 32))

criterion = nn.CrossEntropyLoss()
# Reminder : We define an optimizer, but it will not be used to optimized N2D2␣
↪parameters.
# If you want to change the optimizer of N2D2 refer to the N2D2 solver.
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

And that is it ! From this point, we can follow again the tutorial provided by PyTorch and we have a script ready to run. You can compare the N2D2 and the torch version by commenting the code we added and renaming `torch_net` into `net`.

```
for epoch in range(2):  # loop over the dataset multiple times
e_t = time()
running_loss = 0.0
for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data[0].to(device), data[1].to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
        print('[%d, %5d] loss: %.3f' %
                (epoch + 1, i + 1, running_loss / 2000))
        running_loss = 0.0
print(f"Expoch {epoch} : {time()-e_t}")
print('Finished Training')

dataiter = iter(testloader)
images, labels = dataiter.next()
images = images.to(device)
labels = labels.to(device)
# print images
imshow(torchvision.utils.make_grid(images), "torch_inference.png")
print('GroundTruth: ', ' '.join('%5s' % classes[labels[j]] for j in range(4)))
outputs = net(images)

_, predicted = torch.max(outputs, 1)

print('Predicted: ', ' '.join('%5s' % classes[predicted[j]]
```

(continues on next page)

(continued from previous page)

```
                        for j in range(4)))
```

31.6 Keras interoperability

For this example, we will use an example provided in the Keras documentation : https://keras.io/examples/vision/mnist_convnet/

You can find the full python script here `keras_example.py`.

31.6.1 Example

We begin by importing the same library as in the example plus our interoperability library.

```
import numpy as np
from tensorflow import keras
from tensorflow.keras import layers
# Importing the interoperability library
import keras_to_n2d2
```

We then import the data by following the tutorial.

```
# training parameters
batch_size = 128
epochs = 10
# Model / data parameters
num_classes = 10
input_shape = (28, 28, 1)

# the data, split between train and test sets
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# Scale images to the [0, 1] range
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255

# Make sure images have shape (28, 28, 1)
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)
# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

When declaring the model, we will use the `keras_to_n2d2.wrap()` function to generate an `keras_to_n2d2.CustomSequential` which embedded N2D2.

```
tf_model = keras.Sequential([
        keras.Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
```

(continues on next page)

(continued from previous page)

```
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dense(num_classes, activation="softmax"),
])
model = keras_to_n2d2.wrap(tf_model, batch_size=batch_size, for_export=True)
```

Once this is done, we can follow again the tutorial and run the training and the evaluation.

```
model.compile(loss="categorical_crossentropy", metrics=["accuracy"])

model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_split=0.1)
score = model.evaluate(x_test, y_test, verbose=0)
print("Test loss:", score[0])
print("Test accuracy:", score[1])
```

And that is it ! You have successfully trained your model with N2D2 using the keras interface.

You can then retrieve the N2D2 model by using the method `keras_to_n2d2.CustomSequential.get_deepnet_cell()` if you want to perform operation on it.

```
n2d2_model = model.get_deepnet_cell()
```

CHAPTER

THIRTYTWO

CORE N2D2

32.1 Introduction

In this section we will present the C++ core function that are binded to Python with the framework pybind. The binding of the C++ core is straightforward, thus this section can also be seen as a documentation of the C++ core implementation of N2D2.

If you want to use the raw python binding, you need to compile N2D2. This will create a '.so' file in the *lib* folder. If you want to use the raw binding, you will need to have this file at the root of your project or in your *PYTHONPATH*.

You can then access the raw binding by importing N2D2 in your python script with the line *import N2D2*. It is however not recommended to use the raw binding, you should instead use the *n2d2 python library*.

32.2 DeepNet

32.2.1 Introduction

In order to create a neural network in N2D2 using an INI file, you can use the DeepNetGenerator:

```
net = N2D2.Network(seed-1)
deepNet = N2D2.DeepNetGenerator.generate(net, "../models/mnist24_16c4s2_24c5s2_150_10.ini
↪")
```

Before executing the model, the network must first be initialized:

```
deepNet.initialize()
```

In order to test the first batch sample from the dataset, we retrieve the StimuliProvider and read the first batch from the test set:

```
sp = deepNet.getStimuliProvider()
sp.readBatch(N2D2.Database.Test, 0)
```

We can now run the network on this data:

```
deepNet.test(N2D2.Database.Test, [])
```

Finally, in order to retrieve the estimated outputs, one has to retrieve the first and unique target of the model and get the estimated labels and values:

```
target = deepNet.getTargets()[0]
labels = numpy.array(target.getEstimatedLabels()).flatten()
values = numpy.array(target.getEstimatedLabelsValue()).flatten()
results = list(zip(labels, values))

print(results)
```

```
[(1, 0.15989691), (1, 0.1617092), (9, 0.14962792), (9, 0.16899541), (1, 0.16261548), (1,␣
↪0.17289816), (1, 0.13728766), (1, 0.15315214), (1, 0.14424478), (9, 0.17937174), (9, 0.
↪1518211), (1, 0.12860791), (9, 0.17310674), (9, 0.14563303), (1, 0.17823018), (9, 0.
↪14206158), (1, 0.18292117), (9, 0.14831856), (1, 0.22245243), (9, 0.1745578), (1, 0.
↪20414244), (1, 0.26987872), (1, 0.16570412), (9, 0.17435187)]
```

32.2.2 API Reference

32.3 Cells

32.3.1 Cell

AnchorCell

BatchNormCell

Cell

ConvCell

DeconvCell

DropoutCell

ElemWiseCell

FMPCell

FcCell

LRNCell

LSTMCell

NormalizeCell

ObjectDetCell

PaddingCell

PoolCell

ProposalCell

ROIPoolingCell

RPCell

ResizeCell

ScalingCell

SoftmaxCell

TargetBiasCell

ThresholdCell

TransformationCell

UnpoolCell

32.3.2 Frame

AnchorCell_Frame

AnchorCell_Frame_CUDA

BatchNormCell_Frame_float

BatchNormCell_Frame_double

BatchNormCell_Frame_CUDA_float

BatchNormCell_Frame_CUDA_double

Cell_Frame_float

Cell_Frame_double

Cell_Frame_CUDA_float

Cell_Frame_CUDA_double

Cell_Frame_Top

ConvCell_Frame_float

ConvCell_Frame_double

ConvCell_Frame_CUDA_float

ConvCell_Frame_CUDA_double

DeconvCell_Frame_float

DeconvCell_Frame_double

DeconvCell_Frame_CUDA_float

DeconvCell_Frame_CUDA_double

DropoutCell_Frame_float

DropoutCell_Frame_double

DropoutCell_Frame_CUDA_float

DropoutCell_Frame_CUDA_double

ElemWiseCell_Frame

ElemWiseCell_Frame_CUDA

FMPCell_Frame

FMPCell_Frame_CUDA

FcCell_Frame_float

FcCell_Frame_double

FcCell_Frame_CUDA_float

FcCell_Frame_CUDA_double

LRNCell_Frame_float

LRNCell_Frame_double

LRNCell_Frame_CUDA_float

LRNCell_Frame_CUDA_double

LSTMCell_Frame_CUDA_float

LSTMCell_Frame_CUDA_double

NormalizeCell_Frame_float

NormalizeCell_Frame_double

NormalizeCell_Frame_CUDA_float

NormalizeCell_Frame_CUDA_double

ObjectDetCell_Frame

ObjectDetCell_Frame_CUDA

PaddingCell_Frame

PaddingCell_Frame_CUDA

PoolCell_Frame_float

PoolCell_Frame_double

PoolCell_Frame_CUDA_float

PoolCell_Frame_CUDA_double

PoolCell_Frame_EXT_CUDA_float

PoolCell_Frame_EXT_CUDA_double

ProposalCell_Frame

ProposalCell_Frame_CUDA

ROIPoolingCell_Frame

ROIPoolingCell_Frame_CUDA

RPCell_Frame

RPCell_Frame_CUDA

ResizeCell_Frame

ResizeCell_Frame_CUDA

ScalingCell_Frame_float

ScalingCell_Frame_double

ScalingCell_Frame_CUDA_float

ScalingCell_Frame_CUDA_double

SoftmaxCell_Frame_float

SoftmaxCell_Frame_double

SoftmaxCell_Frame_CUDA_float

SoftmaxCell_Frame_CUDA_double

TargetBiasCell_Frame_float

TargetBiasCell_Frame_double

TargetBiasCell_Frame_CUDA_float

TargetBiasCell_Frame_CUDA_double

ThresholdCell_Frame

ThresholdCell_Frame_CUDA

TransformationCell_Frame

TransformationCell_Frame_CUDA

UnpoolCell_Frame

UnpoolCell_Frame_CUDA

32.4 Filler

32.5 Activation

32.5.1 Introduction

Activation functions in N2D2 are passed as arguments to initialize `N2D2.Cell`.

```
tanh = N2D2.TanhActivation_Frame_float()
```

32.5.2 Activation

Activation

LinearActivation

RectifierActivation

TanhActivation

SwishActivation

SaturationActivation

LogisticActivation

SoftplusActivation

32.5.3 Activation_Frame

LinearActivation_Frame

RectifierActivation_Frame

TanhActivation_Frame

SwishActivation_Frame

32.6 Solver

32.7 Target

32.7.1 Introduction

A `N2D2.Target` is associated to a `N2D2.Cell`, it define the output of the network. The computation of the loss and other tools to compute score such as the confusion matrix are also computed with this class.

To train a neural network you need to use `N2D2.Target.provideTargets()` then to `N2D2.cell.propagate()` then `N2D2.Target.process()` and finally `N2D2.Cell.backpropagate()`. (See *the MNIST example*.)

32.8 Databases

32.8.1 Introduction:

N2D2 allow you to import default dataset or to load your own dataset. This can be done suing Database objects.

32.8.2 Download datasets:

To import Data you can use a python Script situated in `./tools/install/install_dataset.py`.

This script will download the data in `/local/$USER/n2d2_data/`. You can change this path with the environment variable `N2D2_data`.

Once the dataset downloaded, you can load it with the appropriate class. Here is an example of the loading of the MNIST dataset :

```
database = N2D2.MNIST_IDX_Database()
database.load(path)
```

In this example, the data are located in the folder **path**.

32.8.3 Database:

Database

MNIST_IDX_Database

Actitracker_Database

AER_Database

Caltech101_DIR_Database

Caltech256_DIR_Database

CaltechPedestrian_Database

CelebA_Database

CIFAR_Database

CKP_Database

DIR_Database

GTSRB_DIR_Database

GTSDB_DIR_Database

ILSVRC2012_Database

IDX_Database

IMDBWIKI_Database

KITTI_Database

KITTI_Object_Database

KITTI_Road_Database

LITISRouen_Database

N_MNIST_Database

DOTA_Database

Fashion_MNIST_IDX_Database

FDDB_Database

Daimler_Database

32.9 StimuliProvider

32.10 Transformation

32.10.1 Introduction

In order to apply transformation to a dataset, we use the transformation object.

Creation of different Transformation object.

```
dist = N2D2.DistortionTransformation()
dist.setParameter("ElasticGaussianSize", "21")
dist.setParameter("ElasticSigma", "6.0")
dist.setParameter("ElasticScaling", "36.0")
dist.setParameter("Scaling", "10.0")
dist.setParameter("Rotation", "10.0")

padcrop = N2D2.PadCropTransformation(24, 24)

ct = N2D2.CompositeTransformation(padcrop)
ct.push_back(dist)
```

To apply Transformation to a dataset, we use an object `N2D2.StimuliProvider` which acts as a data loader.

32.10.2 Transformations

Transformation

DistortionTransformation

PadCropTransformation

CompositeTransformation

AffineTransformation

ChannelExtractionTransformation

ColorSpaceTransformation

CompressionNoiseTransformation

DCTTransformation

DFTTransformation

EqualizeTransformation

ExpandLabelTransformation

WallisFilterTransformation

ThresholdTransformation

SliceExtractionTransformation

ReshapeTransformation

RescaleTransformation

RangeClippingTransformation

RangeAffineTransformation

RandomAffineTransformation

NormalizeTransformation

MorphologyTransformation

MorphologicalReconstructionTransformation

MagnitudePhaseTransformation

LabelSliceExtractionTransformation

LabelExtractionTransformation

GradientFilterTransformation

ApodizationTransformation

FilterTransformation

FlipTransformation

32.11 Containers

32.11.1 Introduction

N2D2 has his own Tensor implementation.

```
N2D2.Tensor_float([1, 2, 3])
```

Tensor can be also be created using numpy.array object.

```
N2D2.CudaTensor_float(numpy.array([[1.0, 2.0], [3.0, 4.0]]))
```

32.11.2 Tensor

32.11.3 CudaTensor

CHAPTER

THIRTYTHREE

EXAMPLE

In this section we will create a simple convolutional neural network for the dataset MNIST using the python binding of N2D2.

33.1 Creation of the network

We first have to create an object Network. This object will be the backbone of the model, linking the different cells.

We have to begin with the initialisation of this object since it creates a seed that generates randomness.

```
net = N2D2.Network()
deepNet = N2D2.DeepNet(net)
```

33.2 Importation of the dataset

To import the MNIST dataset we will use a custom class `N2D2.MNIST_IDX_Database`.

```
database = N2D2.MNIST_IDX_Database()
database.load(path)
```

In the following code, the *path* variable represent the path to the dataset MNIST.

33.3 Applying transformation to the dataset

We can create transformation using the class `N2D2.Transformation`.

```
trans = N2D2.DistortionTransformation()
trans.setParameter("ElasticGaussianSize", "21")
trans.setParameter("ElasticSigma", "6.0")
trans.setParameter("ElasticScaling", "36.0")
trans.setParameter("Scaling", "10.0")
trans.setParameter("Rotation", "10.0")
padcrop = N2D2.PadCropTransformation(24, 24)
```

But to apply them to the data, we need `N2D2.StimuliProvider`.

`N2D2.StimuliProvider` is a class that acts as a data loader for the neural network.

```
stimuli = N2D2.StimuliProvider(database, [24, 24, 1], batchSize, False)
stimuli.addTransformation(N2D2.PadCropTransformation(24, 24), database.StimuliSetMask(0))
stimuli.addOnTheFlyTransformation(trans, database.StimuliSetMask(0))
```

We can apply transformation in two ways. The first one is the standard one, we apply the transformation once to the whole dataset. This is useful for transformation like normalization or `N2D2.PadCropTransformation`. The other way is to add the transformation "on the fly", this mean that each time we load a data, we apply the transformation. This is especially adapted to random transformation like `N2D2.DistortionTransformation` since you add more diversity to the data.

33.4 Defining network topology

To define our network topology, we use `N2D2.Cell` objects.

```
conv1 = N2D2.ConvCell_Frame_float(deepNet, "conv1", [4, 4], 16, [1, 1], [2, 2], [5, 5],␣
↪[1, 1], N2D2.TanhActivation_Frame_float())
conv2 = N2D2.ConvCell_Frame_float(deepNet, "conv2", [5, 5], 24, [1, 1], [2, 2], [5, 5],␣
↪[1, 1], N2D2.TanhActivation_Frame_float())
fc1 = N2D2.FcCell_Frame_float(deepNet, "fc1", 150, N2D2.TanhActivation_Frame_float())
fc2 = N2D2.FcCell_Frame_float(deepNet, "fc2", 10, N2D2.TanhActivation_Frame_float())
```

Once the cells are created, we need to connect them.

```
conv2mapping = [
    True, False, False, False, False, False, False, False, False, False, False, False,␣
↪False, False, False, True, False, False, False, False, False, False, True, True,
    True, True, False, False, False, False, False, False, False, False, False, False,␣
↪False, False, False, True, False, False, False, False, False, False, True, True,
    False, True, True, False, False, False, False, False, False, False, False, False,␣
↪False, False, False, True, True, False, False, False, False, False, True, True,
    False, False, True, True, False, False, False, False, False, False, False, False,␣
↪False, False, False, True, True, False, False, False, False, False, True, True,
    False, False, False, True, True, False, False, False, False, False, False, False,␣
↪False, False, False, False, True, True, False, False, False, False, True, True,
    False, False, False, False, True, True, False, False, False, False, False, False,␣
↪False, False, False, False, True, True, False, False, False, False, True, True,
    False, False, False, False, False, True, True, False, False, False, False, False,␣
↪False, False, False, False, False, True, True, False, False, False, True, True,
    False, False, False, False, False, False, False, True, True, False, False, False,␣
↪False, False, False, False, False, False, True, True, False, False, True, True,
    False, False, False, False, False, False, False, False, True, True, False, False,␣
↪False, False, False, False, False, False, True, True, False, False, True, True,
    False, False, False, False, False, False, False, False, False, False, True, True,␣
↪False, False, False, False, False, False, True, True, False, False, True, True,
    False, False, False, False, False, False, False, False, False, False, False, True,␣
↪True, False, False, False, False, False, False, False, True, True, True, True,
    False, False, False, False, False, False, False, False, False, False, False, False,␣
```

(continues on next page)

(continued from previous page)

```
↪True, True, False, False, False, False, False, False, True, True, True, True,
    False, False, False, False, False, False, False, False, False, False, False, False,␣
↪False, True, True, False, False, False, False, False, False, True, True, True,
    False, False, False, False, False, False, False, False, False, False, False, False,␣
↪False, False, True, False, False, False, False, False, False, True, True, True]

t_conv2mapping = N2D2.Tensor_bool(numpy.array(conv2mapping))

conv1.addInput(stimuli)
conv2.addInput(conv1, t_conv2mapping)
fc1.addInput(conv2)
fc2.addInput(fc1)
```

The first layer receive the `N2D2.StimuliProvider` class as an input. The other layers have their input set with the previous cell. In this example, we also create a different mapping for the `N2D2.ConvCell_Frame_float` *conv2*.

33.5 Learning phase

Once the network is created, we can begin the learning phase. First, we need to create a `N2D2.Target` object. This object defines the output of the network.

```
tar = N2D2.TargetScore('target', fc2, stimuli)

conv1.initialize()
conv2.initialize()
fc1.initialize()
fc2.initialize()
```

Finally, we can initiate the learning loop.

```
for epoch in range(nb_epochs):
    for i in range(epoch_size):
        stimuli.readRandomBatch(set=N2D2.Database.Learn)
        tar.provideTargets(N2D2.Database.Learn)
        conv1.propagate()
        conv2.propagate()
        fc1.propagate()
        fc2.propagate()
        tar.process(N2D2.Database.Learn)
        fc2.backPropagate()
        fc1.backPropagate()
        conv2.backPropagate()
        conv1.backPropagate()
        conv1.update()
        conv2.update()
        fc1.update()
        fc2.update()
```

CHAPTER

THIRTYFOUR

INTRODUCTION

Welcome to the developer documentation. N2D2 is primarily developed in C++11 and CUDA (for the GPU computing kernels). The library used internally for images processing is OpenCV.

N2D2 is structured in module families, each family being defined by a base class, for example `Database`, `Cell`, `Export`... In this introduction, we will review the main N2D2 modules structure, which is summarized in the figure below:

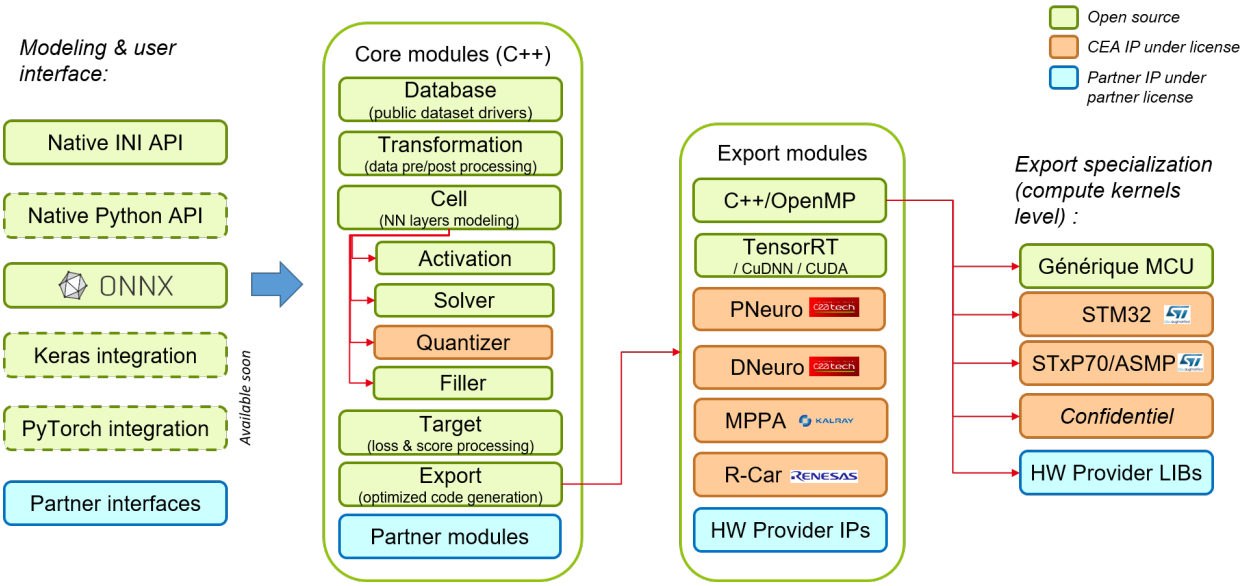


Fig. 1: Main N2D2 modules.

34.1 The `Cell` modules

A `Cell` defines a layer / operator, constituting a base building block of a neural network (or more generally a compute graph), like a convolution, a pooling and so on. The base `Cell` class is abstract and does not make any assumption on the compute model, the data precision and even the coding of the data.

The class hierarchy for `Cell` is shown in the figure below. While it may seem a bit complicated, it was designed this way to really separate different notions:

- *Mathematical model*: the actual mathematical function performed by the `Cell`, regardless of the algorithm used for its implementation. For example, `ConvCell` for a convolution, or `FcCell` for a fully connected layer (or inner product);

- *Coding model*: the model used to code the data. `Frame` is for standard tensor-based deep learning and `Spike` is for neuromorphic spike coding;
- *Programming model*: the programming model that will be used for the implementation, meaning either plain C++ (with OpenMP) for CPU or CUDA for GPU. Other programming model, like OpenCL, may be added in the future. When there is only one programming model, the class hierarchy can be simplified like for `Cell_Spike`. The *Programming model* class can be templated to handle different data precision. For the `Cell_Frame<T>` and `Cell_Frame_CUDA<T>` classes, `half`, `float` and `double` precision are supported, but may not be implemented for every model (it is not mandatory to provide implementation for every possible precision);
- *Implementation*: the actual implementation of the *Mathematical model*, using the inherited *Coding model* and *Programming model*. The implementation may use different algorithm to obtain the same *Mathematical model*, like direct, GEMM or FFT algorithms for the convolution.

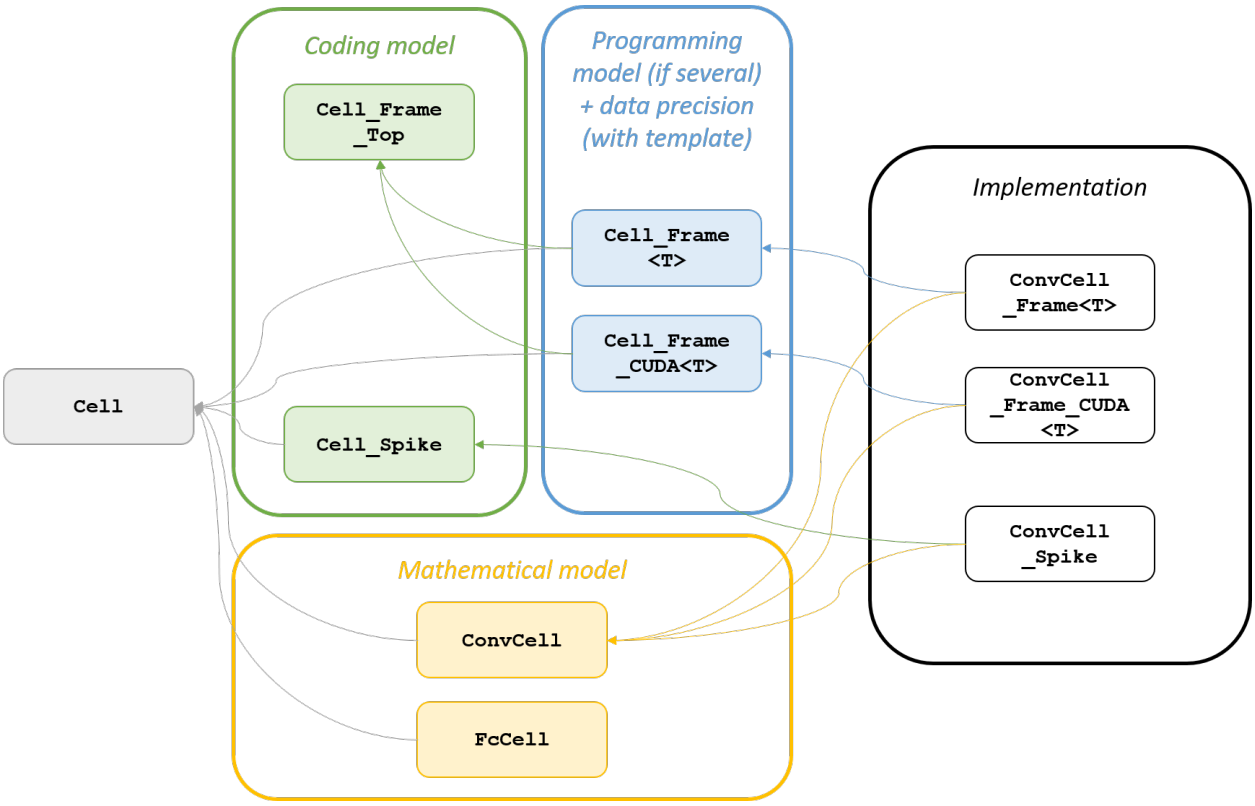


Fig. 2: `Cell` modules class hierarchy.

34.1.1 `Cell` class

The base `Cell` class only handles the topological information: the inputs and outputs dimensions and virtual methods to connect the cells, among other. The main methods are listed here:

Warning: doxygenclass: breathe_default_project value 'N2D2' does not seem to be a valid key for the breathe_projects dictionary

In order to create a computing graph, or neural network, cells can be connected together thanks to the `addInput` method. An `Cell` can be connected to another `Cell` or to a `StimuliProvider`, which constitute an entry point for

the data. It is up to the implementation to allow or not multiple inputs. The `initialize` virtual member initializes the state of the `Cell` in the implementation (initialization may be different depending on the coding or programming model).

Warning: doxygenclass: breathe_default_project value 'N2D2' does not seem to be a valid key for the breathe_projects dictionary

34.1.2 `Cell_Frame_Top` class

The `Cell_Frame_Top` is a purely abstract class, which does not inherit from the `Cell` class. It provides all the actual interface for a given *Coding model*. For the `Frame` coding model, the input/output data is a Nd-tensor. This coding model is also "bi-directional", as it supports differentiation, thus its interface comprise a forward data path and backward data path for the gradient. It is however not mandatory for the implementation to handle the backward path, for non-differentiable cells for example.

The main interfaces provided by `Cell_Frame_Top` are listed below:

Warning: doxygenclass: breathe_default_project value 'N2D2' does not seem to be a valid key for the breathe_projects dictionary

This class also handles the `Activation` and contains the shared pointer to the `Activation` object to use.

Warning: doxygenclass: breathe_default_project value 'N2D2' does not seem to be a valid key for the breathe_projects dictionary

34.1.3 `Cell_Frame<T>` class

The `Cell_Frame<T>` inherits from `Cell` and `Cell_Frame_Top`. It contains the output tensors and input tensor references. `Cell_Frame<T>` should not provide more members than `Cell_Frame_Top` already does.

34.1.4 `ConvCell` class

The `ConvCell` class (or any other model) provides additional interfaces to the base `Cell` class that it inherits, specific to the *Mathematical model* it represents, regardless of the *Coding model*. For a convolution for example, getter and setter members are provided for the weights, but also references to `Filler` and `Solver` for the weights. Some facilities may be specific to a particular coding model. The data structure (tensor) containing the weights is however not defined in this class, but in the implementation (`ConvCell_Frame`), because it can depend on the programming model and data precision.

The main members (for `ConvCell`) are:

Warning: doxygenclass: breathe_default_project value 'N2D2' does not seem to be a valid key for the breathe_projects dictionary

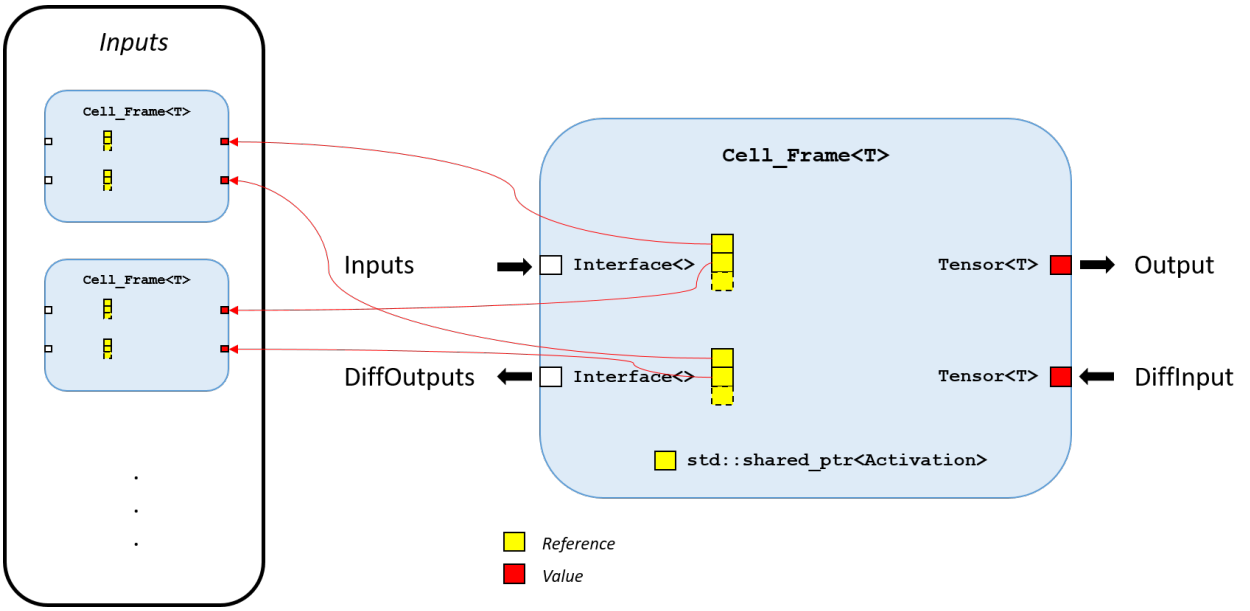


Fig. 3: `Cell_Frame<T>` interface.

34.1.5 `ConvCell_Frame<T>` class

The `ConvCell_Frame<T>` is the final class implementing the convolution *Mathematical model* with the tensor-based `Frame` *Coding model* on standard C++ for CPU *Programming model*. It is only an implementation class, which therefore does not provide any new members.

34.2 The `Tensor<T>` class

The `Tensor<T>` class is the base data structure in N2D2, a Nd-tensor. In N2D2, a tensor has the following properties:

- Its data is guaranteed to be **continuous**. It therefore does not handle arbitrary strides without actual data reorganization. While this property may appear restricting, it is an assumed design choice to simplify the implementation of the many possible coding and programming models;
- It **holds a reference** to the data, meaning no data is actually copied in a tensor assignment and the new tensor will point to the same data. To perform an actual copy, the `clone()` method should be used;
- It is **explicitly typed**, meaning that the data type is part of the tensor type, as a template parameter. When the type does not need to be known or can be arbitrary, a reference to the `BaseTensor` base class should be used. Conversion from one tensor type (or from a `BaseTensor` reference) to another can be performed with the `tensor_cast<T>()` function;
- Down to **zero overhead** type conversion. Memory is only allocated once for a given type conversion during the lifetime of the tensor. Type conversion can be made without any data copy-conversion, with the `tensor_cast_nocopy<T>()` function, when it is known that a previous conversion is still valid, thus incurring zero overhead.

The `Tensor<T>` implementation classes hierarchy is shown in the figure below.

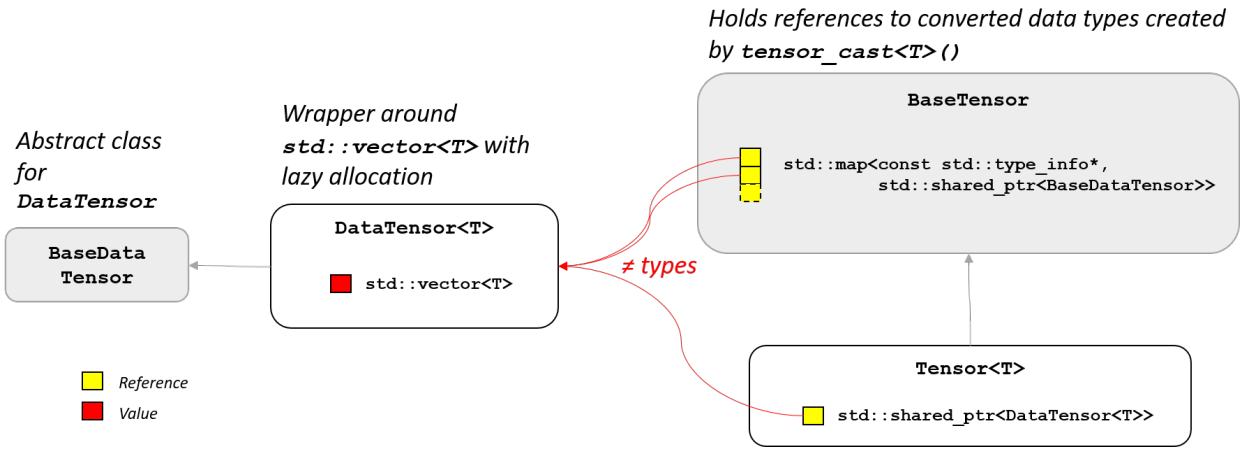


Fig. 4: Tensor<T> implementation classes hierarchy.

CHAPTER

THIRTYFIVE

INDICES AND TABLES

- genindex
- modindex
- search
- References