SNN toolbox Documentation

Release 0.1

Bodo Rueckauer

Aug 29, 2016

CONTENTS

This is a toolbox for converting analog to spiking neural networks (ANN to SNN), and running them in a spiking neuron simulator.

CHAPTER

ONE

CITATION

```
Diehl, P.U. and Neil, D. and Binas, J. and Cook, M. and Liu, S.C. and Pfeiffer, M.
Fast-Classifying, High-Accuracy Spiking Deep Networks Through Weight and Threshold Balancing,
IEEE International Joint Conference on Neural Networks (IJCNN), 2015
```

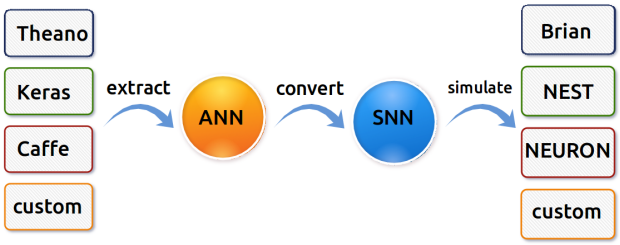

Fig. 1.1: **SNN toolbox workflow.** The input network (e.g. a Keras model) is parsed and all the necessary information stored in an object that is independent of the input. This makes all following steps stable against changes in the input, and allows straight-forward extension of the toolbox to include other input model libraries. The parsed model can then be converted into a spiking network. The resulting SNN can be exported for evaluation in a spiking simulator. At any stage of the pipeline, models and results can be written to disk.

CHAPTER

TWO

FEATURES

- Before conversion, the input model is parsed into a custom class containing only essential model structure and weights in common python containers. This serves to abstract the core conversion process from possible input types. The conversion toolbox currently supports input networks generated with Keras, Lasagne, or Caffe. See *Getting Started* on how to extend the relevant methods to handle models from other common libraries like torch etc.
- During parsing of the input model, several simplifications are performed to prepare the network for subsequent conversion:
- During conversion of the analog neural network to spiking, the toolbox allows normalizing model parameters for achieving higher accuracy in the converted net.
- The resulting spiking network can then be exported to be tested in spiking simulators. The export format depends on the target simulator. See *Getting Started* on how to add a simulator to the toolbox.
- The toolbox currently provides the following output formats:
  - pyNN models. pyNN is a simulator-independent language for building neural network models. It allows running the converted net in a spiking simulator like Brian, Nest, Neuron, or by a custom simulator that allows pyNN models as inputs.
  - Models to be run in Brian2.
  - An output format based on Keras models that can be run for instance on a built-in simulator developed at the University of Zurich.
  - The toolbox integrates MegaSim, an event-driven asynchronous spiking simulator developed at the University of Seville.
- In addition to supporting the simulators listed above, the toolbox includes a ready-to-use simulator developed at INI. This simulator features a very simple integrate-and-fire neuron. By dispensing with redundant parameters and implementing a highly parallel simulation, the run time is reduced by several orders of magnitude, without compromising accuracy.
- Examples for both convolutional networks and fully-connected networks on MNIST and CIFAR10 are provided.
- So far, this toolbox is able to handle classification datasets. For other applications, the `io.load.get_dataset` module needs to be extended.

Getting Started

Installation

We recommend using virtual environments for different simulators, since the simulators supported by pyNN may require different versions of python, numpy, ... (Brian for instance only works with python-2).

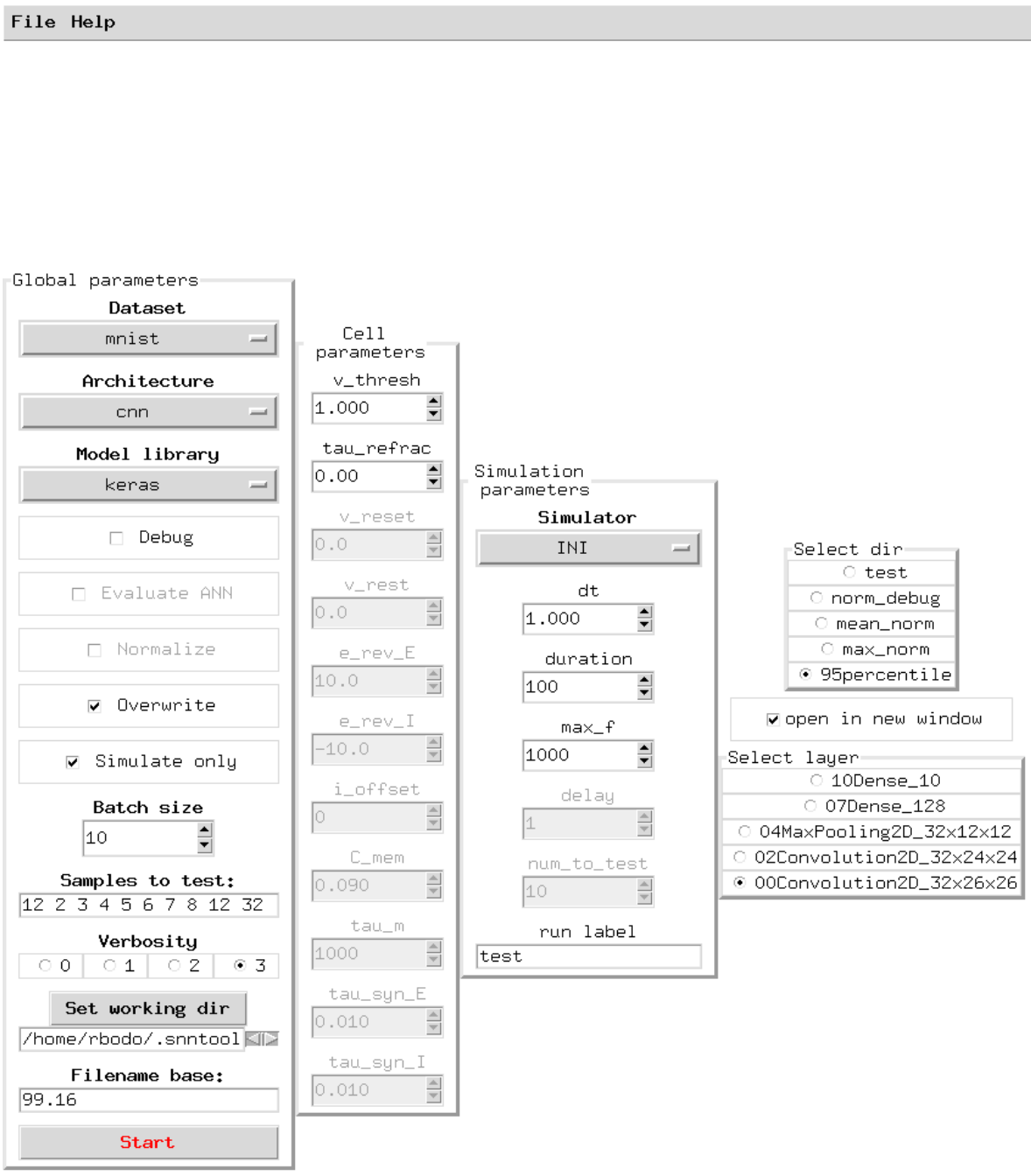

Fig. 2.1: **SNN toolbox GUI.** In the main window, the user can specify which tools to use during the experiment (e.g. whether or not to normalize weights prior to conversion, to evaluate the ANN before converting, to load an already converted net and simulate only, etc.). Also, parameters of the neuron cells used during simulation can be set. The GUI saves and reloads last settings automatically, and allows saving and loading preferences manually. Tooltips explain all functionality.

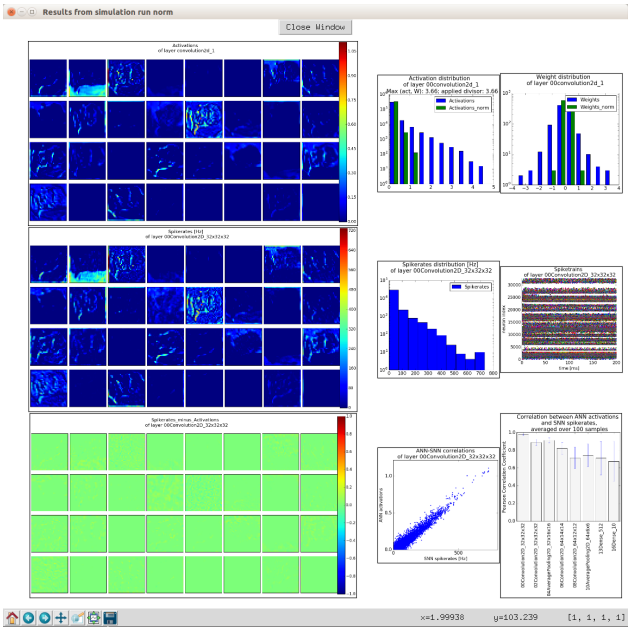

Fig. 2.2: **SNN toolbox GUI plot window.** The toolbox looks for plots in the specified working directory. The user can select one or several layers, for which the results of a test run will be displayed in a way that facilitates examining and comparing results of each layer of the network. The example above compares ANN activations to SNN spikerates for the first convolutional layer on the MNIST dataset.

Requirements

- All dependencies will be installed automatically.
- For testing a converted network, the toolbox includes a ready-to-use spiking simulator developed at INI. In addition, you may install a simulator supported by pyNN, or bring your own custom simulator that accepts a pyNN or Keras model as input.

Prebuild version

- Download the archive `dist/snntoolbox-<version>-py2.py3-none-any.whl`
- Run `pip install snntoolbox-<version>-py2.py3-none-any.whl`.

Development version

- To get the latest version, checkout the repository
- In the toolbox root directory `SNN_toolbox/`, run `python setup.py develop`.

Running the toolbox

In a terminal window, type `snntoolbox` to start the main GUI containing all tools.

Alternitively, read and run example.py in snntoolbox/tests/, which contains a number of typical usecases.

Extending the toolbox

Have a look at the pipeline module to examine the complete pipeline of

1. loading and testing a pretrained ANN,
2. normalizing weights
3. converting it to SNN,
4. running it on a simulator,
5. if given a specified hyperparameter range `params`, repeat simulations with modified parameters.

Input side: Adding a new model library

So far, the toolbox supports input models written in Keras and Lasagne.

The philosophy behind the architecture is to make all steps in the conversion/simulation pipeline independent of the original model format. Therefore, in order to add a new input model library (e.g. Caffe) to the toolbox, put a module named `caffe_input_lib` into the `model_libs` package. Have a look at one of the existing files there to get an idea what functions have to be implemented. The return requirements are specified in their respective docstrings. Basically, all it needs is a function to parse the essential information about layers into a common format used by the toolbox further down the line.

Output side: Adding a custom simulator

Similarly, adding another simulator to run converted networks implies adding a file to the `target_simulators` package. Each file in there allow building a spiking network and exporting it for use in a specific spiking simulator.

To add a simulator called 'custom', put a file named `custom_target_sim.py` into `target_simulators`. Then implement the class `SNN` with its methods (`load`, `save`, `build`, `run`) tailored to 'custom' simulator.

Examples - Fully Connected Network on MNIST

Normally, we would run the toolbox simply by typing `snntoolbox` in the terminal and using the GUI.

If working with a python interpreter, one would specify a set of parameters and then call `tests.util.test_full()`, like this:

```
import snntoolbox

# Define parameters
settings = {'dataset': 'mnist',
            'architecture': 'cnn',
            'filename': '99.16',
            'path': 'example/mnist/cnn/99.16/INI/',
            'evaluateANN': True,  # Test accuracy of input model
            'normalize': True,  # Normalize weights to get better perf.
            'convert': True,  # Convert analog net to spiking
            'simulate': True,  # Simulate converted net
            'verbose': 3,  # Show plots and temporary results
            'v_thresh': 1,  # Threshold potential
            'simulator': 'INI',  # Reset potential
            'duration': 100.0}  # Simulation time

# Run network (including loading the model, weight normalization, conversion
```

```
# and simulation)
snntoolbox.update_setup(settings)
snntoolbox.test_full()
```

However, here are three usecases that allow some more insight into the application of this toolbox:

1. *Conversion only*
2. *Simulation only*
3. *Parameter sweep*

For a description of the possible values for the parameters in `settings`, see *Configuration*.

Usecase A - Conversion only

Steps:

1. Set `convert = True` and `simulate = False`
2. Specify other parameters (working directory, filename, ...)
3. Update settings: `update_setup(settings)`
4. Call `test_full()`. This will
   - load the dataset,
   - load a pretrained ANN from `<path>/<filename>`
   - optionally evaluate it (`evaluate = True`),
   - optionally normalize weights (`normalize = True`),
   - convert to spiking,
   - save SNN to disk.

Usecase B - Simulation only

Steps:

1. Set `convert = False` and `simulate = True`
2. Specify other parameters (working directory, simulator to use, ...)
3. Update settings: `update_setup(settings)`
4. Call `test_full()`. This will
   - load the dataset,
   - load your already converted SNN,
   - run the net on a spiking simulator,
   - plot spikerates, spiketrains, activations, correlations, etc.

Note: It is assumed that a network has already been converted (e.g. with Usecase A). I.e. there should be a folder in `<path>` containing the converted network, named `snn_<filename>_<simulator>`.

Usecase C - Parameter sweep

Steps:

1. Specify parameters and update settings with `update_setup(settings)`
2. Define a parameter range to sweep, e.g. for *v_thresh*, using for instance the helper function `get_range()`
3. Call `test_full`. This will
   - load an already converted SNN or perform a conversion as specified in settings.
   - run the SNN repeatedly on a spiking simulator while varying the hyperparameter
   - plot accuracy vs. hyperparameter

Usecase C is shown in full in the example below.

```
import snntoolbox

# Parameters
settings = {'dataset': 'mnist',
            'architecture': 'cnn',
            'filename': '99.16',
            'path': 'example/mnist/cnn/99.16/INI/',
            'evaluateANN': True,  # Test accuracy of input model
            'normalize': True,  # Normalize weights to get better perf.
            'convert': True,  # Convert analog net to spiking
            'simulate': True,  # Simulate converted net
            'verbose': 3,  # Show plots and temporary results
            'v_thresh': 1,  # Threshold potential
            'simulator': 'INI',  # Reset potential
            'duration': 100.0}  # Simulation time

# Update defaults with parameters specified above:
snntoolbox.update_setup(settings)

# Run network (including loading the model, weight normalization,
# conversion and simulation).

# If set True, the converted model is simulated for three different values
# of v_thresh. Otherwise use parameters as specified above,
# for a single run.
do_param_sweep = True
if do_param_sweep:
    param_name = 'v_thresh'
    params = snntoolbox.get_range(0.1, 1.5, 3, method='linear')
    snntoolbox.test_full(params=params,
                         param_name=param_name,
                         param_logscale=False)
else:
    snntoolbox.test_full()
```

Contact

- Bodo Rueckauer

Configuration

Manage Parameters of SNNToolbox.

In the GUI, the toolbox settings are grouped in three categories:

1. Global parameters `globalparams`, specifying global settings for loading / saving, and what steps of the workflow to include (evaluation, normalization, conversion, simulation, ...)
2. Neuron cell parameters `cellparams`, determining properties of the spiking neurons (e.g. threshold, refractory period, ...). Not all of them are used in all simulators. For instance, our own simulator `'INI'` only uses a threshold, reset and membrane time constant.
3. Simulation parameters `simparams`, specifying for instance length and time resolution of the simulation run.

Parameters

Global Parameters

dataset_path: string Where to load the dataset from. Used for testing the network. Dataset needs to be in npy format.

model_lib: string The neural network library used to build the ANN, e.g.

- 'keras'
- 'lasagne'
- 'caffe'

path_wd: string, optional Working directory. There, the toolbox will look for ANN models to convert or SNN models to test, load the parameters it needs and store (normalized) parameters. If not specified, the toolbox will use as destination for all files it needs to load and save: `~/.snntoolbox/data/<filename_ann>/<simulator>/`. For instance, if we give `'98.29'` as filename of the ANN model to load, and use default parameters otherwise, the toolbox will perform all io-operations in `~/.snntoolbox/data/mnist/mlp/98.29/INI/`.

log_dir_of_current_run: string, optional Path to directory where the output plots are stored. If not specified, will be `<path_wd>/log/gui/<runlabel>`. `<runlabel>` can be specified in the GUI. Will be set to 'test' if None.

filename_ann: string Base name of all loaded and saved files during this run. The ANN model to be converted is expected to be named '<basename>'.

filename_parsed_model: string, optional Name given to parsed SNN models. If not specified by the user, the toolbox sets it to '<basename>_parsed'.

filename_snn: string, optional Name given to converted spiking nets when exported to test it in a specific simulator. If not specified by the user, the toolbox set it to `snn_<basename>_<simulator>`.

evaluateANN: boolean, optional If enabled, test the input model before and after it is parsed, to ensure we do not lose performance. (Parsing extracts all necessary information from the input model and creates a new network with some simplifications in preparation for conversion to SNN.) If you also enabled 'normalization' (see parameter `normalize` below), then the network will be evaluated again after normalization. This operation should preserve accuracy as well.

normalize: boolean, optional Only relevant when converting a network, not during simulation. If enabled, the parameters of the spiking network will be normalized by the highest activation value, or by the `n`-th percentile (see parameter `percentile` below).

percentile: int, optional Use the activation value in the specified percentile for normalization. Set to `50` for the median, `100` for the max. Typical values are `99, 99.9, 100`.

convert: boolean, optional If enabled, load an ANN from `<path_wd>` and convert it to spiking.

simulate: boolean, optional If enabled, try to load SNN from `<path_wd>` and test it on the specified simulator (see parameter `simulator`).

overwrite: boolean, optional If disabled, the save methods will ask for permission to overwrite files before writing parameters, activations, models etc. to disk.

batch_size: int, optional If the builtin simulator 'INI' is used, the batch size specifies the number of test samples that will be simulated in parallel. Important: When using 'INI' simulator, the batch size can only be run usingthe batch size it has been converted with. To run it with a different batch size, convert the ANN from scratch.

verbose: int, optional 0: No intermediate results or status reports. 1: Print progress of simulation and intermediate results. 2: Record spiketrains of all layers for one sample, and save various plots (spiketrains, spikerates, activations, correlations, ...) 3: Record, plot and return the membrane potential of all layers for the last test sample. Very time consuming. Works only with pyNN simulators.

scaling_factor: int, optional Used by the MegaSim simulator to scale the neuron parameters and weights because MegaSim uses integers.

Cell Parameters

v_thresh: float, optional Threshold in mV defining the voltage at which a spike is fired.

v_reset: float, optional Reset potential in mV of the neurons after spiking.

v_rest: float, optional Resting membrane potential in mV.

e_rev_E: float, optional Reversal potential for excitatory input in mV.

e_rev_I: float, optional Reversal potential for inhibitory input in mV.

i_offset: float, optional Offset current in nA.

cm: float, optional Membrane capacitance in nF.

tau_m: float, optional Membrane time constant in milliseconds.

tau_refrac: float, optional Duration of refractory period in milliseconds of the neurons after spiking.

tau_syn_E: float, optional Decay time of the excitatory synaptic conductance in milliseconds.

tau_syn_I: float, optional Decay time of the inhibitory synaptic conductance in milliseconds.

softmax_clockrate: int, optional In our implementation of a spiking softmax activation function we use an external Poisson clock to trigger calculating the softmax of a layer. The 'softmax_clockrate' parameter sets the firing rate in Hz of this external clock. Note that this rate is limited by the maximum firing rate supported by the simulator (given by the inverse time resolution 1000 * 1 / dt Hz).

Simulation Parameters

simulator: string, optional Simulator with which to run the converted spiking network.

duration: float, optional Runtime of simulation of one input in milliseconds.

dt: float, optional Time resolution of spikes in milliseconds.

delay: float, optional Delay in milliseconds. Must be equal to or greater than the resolution.

poisson_input: float, optional If enabled, the input samples will be converted to Poisson spiketrains. The probability for a input neuron to fire is proportional to the analog value of the corresponding pixel, and limited by the parameter 'input_rate' below. For instance, with an 'input_rate' of 700, a fully-on pixel will elicit a Poisson spiketrain of 700 Hz. Turn off for a less noisy simulation. Currently, turning off Poisson input is only possible in INI simulator.

reset: string, optional Choose the reset mechanism to apply after spike. Reset to zero: After spike, the membrane potential is set to the resting potential. Reset by subtraction: After spike, the membrane potential is reduced by a value equal to the threshold.

input_rate: float, optional Poisson spike rate in Hz for a fully-on pixel of the input image. Note that the input_rate is limited by the maximum firing rate supported by the simulator (given by the inverse time resolution 1000 * 1 / dt Hz).

normalization_schedule: boolean, optional Reduce the normalization factor each layer.

online_normalization: boolean, optional The converted spiking network performs best if the average firing rates of each layer are not higher but also not much lower than the maximum rate supported by the simulator (inverse time resolution). Normalization eliminates saturation but introduces undersampling (parameters are normalized with respect to the highest value in a batch). To overcome this, the spikerates of each layer are monitored during simulation. If they drop below the maximum firing rate by more than 'diff to max rate', we set the threshold of the layer to its highest rate.

diff_to_max_rate: float, optional If the highest firing rate of neurons in a layer drops below the maximum firing rate by more than 'diff to max rate', we set the threshold of the layer to its highest rate. Set the parameter in Hz.

diff_to_min_rate: float, optional When The firing rates of a layer are below this value, the weights will NOT be modified in the feedback mechanism described in 'online_normalization'. This is useful in the beginning of a simulation, when higher layers need some time to integrate up a sufficiently high membrane potential.

timestep_fraction: int, optional If set to 10 (default), the parameter modification mechanism described in 'online_normalization' will be performed at every 10th timestep.

num_to_test: int, optional How many samples to test.

maxpool_type [string] Implementation variants of spiking MaxPooling layers, based on fir_max: accmulated absolute firing rate avg_max: moving average of firing rate

Default values

```
globalparams = {'dataset_path': '',
                'model_lib': 'keras',
                'path_wd': '',
                'log_dir_of_current_run': '',
                'filename_ann': '',
                'filename_parsed_model': '',
                'filename_snn': 'snn_',
                'batch_size': 100,
                'evaluateANN': True,
                'normalize': True,
                'percentile': 99,
                'convert': True,
                'overwrite': True,
                'simulate': True,
                'verbose': 3}
cellparams = {'v_thresh': 1,
              'v_reset': 0,
              'v_rest': 0,
              'e_rev_E': 10,
```

```
                'e_rev_I': -10,
                'i_offset': 0,
                'cm': 0.09,
                'tau_m': 1000,
                'tau_refrac': 0,
                'tau_syn_E': 0.01,
                'tau_syn_I': 0.01,
                'softmax_clockrate': 300}
simparams = {'simulator': 'INI',
             'duration': 200,
             'dt': 1,
             'delay': 1,
             'poisson_input': False,
             'reset': 'Reset by subtraction',
             'input_rate': 1000,
             'timestep_fraction': 10,
             'diff_to_max_rate': 200,
             'num_to_test': 10,
             'diff_to_min_rate': 100}
```

snntoolbox.config.**initialize_simulator**(*simulator=None*)

Import module containing utility functions of spiking simulator.

snntoolbox.config.**update_setup**(*s=None*)

Update parameters.

Check that parameter choices `s` are valid and update the global parameter settings `snntoolbox.config.settings` with the user-specified values. Default values are filled in where user did not give any.

Tools

pipeline

Wrapper script that combines all tools of SNN Toolbox.

Created on Thu May 19 16:37:29 2016

@author: rbodo

core.pipeline.**is_stop**(*queue*)

core.pipeline.**test_full**(*queue=None, params=[1], param_name='v_thresh', param_logscale=False*)

Convert an snn to a spiking neural network and simulate it.

Complete pipeline of

1. loading and testing a pretrained ANN,
2. normalizing parameters
3. converting it to SNN,
4. running it on a simulator,
5. if given a specified hyperparameter range `params`, repeat simulations with modified parameters.

The testsuit allows specification of

- the dataset (e.g. MNIST or CIFAR10)
- the spiking simulator to use (currently Brian, Brian2, Nest, Neuron, MegaSim or INI's simulator.)

Perform simulations of a spiking network, while optionally sweeping over a specified hyper-parameter range. If the keyword arguments are not given, the method performs a single run over the specified number of test samples, using the updated default parameters.

Parameters

- **queue** (*Queue, optional*) – Results are added to the queue to be displayed in the GUI.
- **params** (*ndarray, optional*) – Contains the parameter values for which the simulation will be repeated.
- **param_name** (*string, optional*) – Label indicating the parameter to sweep, e.g. `'v_thresh'`. Must be identical to the parameter's label in `globalparams`.
- **param_logscale** (*boolean, optional*) – If `True`, plot test accuracy vs `params` in log scale. Defaults to `False`.

Returns **results** – List of the accuracies obtained after simulating with each parameter value in param_range.

Return type list

inisim

INI spiking neuron simulator.

A collection of helper functions, including spiking layer classes derived from Keras layers, which were used to implement our own IF spiking simulator.

Not needed when converting and running the SNN in other simulators (pyNN, MegaSim, ...)

Created on Tue Dec 8 10:41:10 2015

@author: rbodo

class `core.inisim.`**SpikeAveragePooling2D**(*pool_size=(2, 2)*, *strides=None*, *border_mode='valid'*, *label=None*, ***kwargs*)

Bases: `keras.layers.pooling.AveragePooling2D`

get_name()
Get class name.

get_output(*train=False*)
Get output.

class `core.inisim.`**SpikeConvolution2D**(*nb_filter*, *nb_row*, *nb_col*, *weights=None*, *border_mode='valid'*, *subsample=(1, 1)*, *label=None*, *filter_flip=True*, ***kwargs*)

Bases: `keras.layers.convolutional.Convolution2D`

Spike 2D Convolution.

get_name()
Get class name.

get_output(*train=False*)
Get output.

class core.inisim.**SpikeDense**(*output_dim*, *weights=None*, *label=None*, ***kwargs*)
Bases: keras.layers.core.Dense

Spike Dense layer.

get_name()
Get class name.

get_output(*train=False*)
Get output.

class core.inisim.**SpikeFlatten**(*label=None*, ***kwargs*)
Bases: keras.layers.core.Flatten

Spike flatten layer.

get_name()
Get class name.

get_output(*train=False*)
Get output.

class core.inisim.**SpikeMaxPooling2D**(*pool_size=(2, 2)*, *strides=None*, *border_mode='valid'*, *label=None*, *pool_type='fir_max'*, ***kwargs*)
Bases: keras.layers.pooling.MaxPooling2D

Max Pooling.

get_name()
Get class name.

get_output(*train=False*)
Get output.

core.inisim.**floatX**(*X*)
Return array in floatX settings of Theano.

core.inisim.**get_input**(*self*)
Get input.

core.inisim.**get_new_thresh**(*self*, *time*)
Get new threshhold.

core.inisim.**get_time**(*self*)
Get time.

core.inisim.**get_updates**(*self*)
Get updates.

core.inisim.**init_layer**(*self*, *layer*, *v_thresh*, *tau_refrac*)
Init layer.

core.inisim.**init_neurons**(*self*, *v_thresh=1.0*, *tau_refrac=0.0*, ***kwargs*)
Init neurons.

core.inisim.**linear_activation**(*self*, *time*, *updates*)
Linear activation.

core.inisim.**on_gpu**()
Check if running on GPU board.

core.inisim.**pool_same_size**(*data_in*, *patch_size*, *ignore_border=True*, *st=None*, *padding=(0, 0)*)
Max-pooling in same size.

The indices of maximum values are 1s, else are 0s.

Parameters

- **data_in** (*4-D tensor.*) – input images. Max-pooling will be done over the 2 last dimensions.
- **patch_size** (*tuple*) – with length 2 (patch height, patch width)
- **ignore_border** (*bool*) – When True, (5,5) input with ds=(2,2) will generate a (2,2) output. (3,3) otherwise.
- **st** (*tuple*) – Stride size, which is the number of shifts over rows/cols to get the next pool region. If st is None, it is considered equal to ds (no overlap on pooling regions).
- **padding** (*tuple*) – (pad_h, pad_w) pad zeros to extend beyond four borders of the images, pad_h is the size of the top and bottom margins, and pad_w is the size of the left and right margins.

core.inisim.**reset**(*self*)
Reset.

core.inisim.**sharedX**(*X*, *dtype='float32'*, *name=None*)
Make array as shared array.

core.inisim.**shared_zeros**(*shape*, *dtype='float32'*, *name=None*)
Make shared zeros array.

core.inisim.**skip_spike**(*new_mem*)
Skip spike.

core.inisim.**softmax_activation**(*self*, *time*, *updates*)
Softmax activation.

core.inisim.**trigger_spike**(*new_mem*)
Trigger spike.

core.inisim.**update_neurons**(*self*, *time*, *updates*)
Update neurons according to activation function.

core.inisim.**update_payload**(*self*, *new_mem*, *spikes*, *time*)
Update payloads.

megasim

MegaSim spiking neuron simulator

A collection of helper functions used to get MegaSim's path and executable.

the configuration file will be stored at $HOME/.snntoolbox/preferences/megasim_config.json

Assumes that have write access to the home folder.

Created on Tue Dec 8 10:41:10 2015

@author: evan

core.megasim.**megasim_path**()

util

Helper functions to handle parameters and variables of interest during conversion and simulation of an SNN.

Created on Wed Mar 9 16:18:33 2016

@author: rbodo

core.util.**extract_label**(*label*)
Get the layer number, name and shape from a string.

Parameters **label** (*string*) – Specifies both the layer type, index and shape, e.g. '03Convolution2D_3x32x32'.

Returns

- **layer_num** (*int*) – The index of the layer in the network.
- **name** (*string*) – The type of the layer.
- **shape** (*tuple*) – The shape of the layer

core.util.**get_activ_fn_for_layer**(*model*, *i*)

core.util.**get_activations_batch**(*ann*, *X_batch*)
Compute layer activations of an ANN.

Parameters

- **ann** (*Keras model*) – Needed to compute activations.
- **X_batch** (*float32 array*) – The input samples to use for determining the layer activations. With data of the form (channels, num_rows, num_cols), X has dimension (batch_size, channels*num_rows*num_cols) for a multi-layer perceptron, and (batch_size, channels, num_rows, num_cols) for a convolutional net.

Returns **activations_batch** – Each entry represents a layer in the ANN for which an activation can be calculated (e.g. Dense, Convolution2D). activations containing the activations of a layer. It has the same shape as the original layer, e.g. (batch_size, n_features, n_rows, n_cols) for a convolution layer. label is a string specifying the layer type, e.g. 'Dense'.

Return type list of tuples (activations, label)

core.util.**get_activations_layer**(*get_activ*, *X_train*)
Get activations of a specific layer, iterating batch-wise over the complete data set.

Parameters

- **get_activ** (*Theano function*) – A Theano function computing the activations of a layer.
- **X_train** (*float32 array*) – The samples to compute activations for. With data of the form (channels, num_rows, num_cols), X_train has dimension (batch_size, channels*num_rows*num_cols) for a multi-layer perceptron, and (batch_size, channels, num_rows, num_cols) for a convolutional net.

Returns **activations** – The activations of cells in a specific layer. Has the same shape as the layer.

Return type array

core.util.**get_range**(*start=0.0*, *stop=1.0*, *num=5*, *method='linear'*)
Return a range of parameter values.

Convenience function. For more flexibility, use numpy.linspace, numpy.logspace, numpy.random.random_sample directly.

Parameters

- **start** (*scalar, optional*) – The starting value of the sequence
- **stop** (*scalar, optional*) – End value of the sequence.

- **num** (*int, optional*) – Number of samples to generate. Must be non-negative.
- **method** (*string, optional*) – The sequence will be computed on either a linear, logarithmic or random grid.

Returns **samples** – There are `num` samples in the closed interval [start, stop].

Return type ndarray

`core.util.`**`get_sample_activity_from_batch`**(*activity_batch*, *idx=0*)
Return layer activity for sample `idx` of an `activity_batch`.

`core.util.`**`get_scale_fac`**(*activations*, *idx=0*)
Determine the maximum activation of a layer.

Parameters

- **activations** (*array*) – The activations of cells in a specific layer, flattened to 1-d.
- **idx** (*int, optional*) – The index of the layer. May be used to decrease the scale factor in higher layers, to maintain high spike rates.

Returns **scale_fac** – Maximum (or percentile) of activations in this layer. Parameters of the respective layer are scaled by this value.

Return type float

`core.util.`**`normalize_parameters`**(*model*)
Normalize the parameters of a network.

The parameters of each layer are normalized with respect to the maximum activation, or the `n`-th percentile of activations.

Generates plots of the activity- and weight-distribution before and after normalization. Note that plotting the activity-distribution can be very time- and memory-consuming for larger networks.

`core.util.`**`parse`**(*input_model*)
Create a Keras model suitable for conversion to SNN.

This parsing function takes as input an arbitrary neural network and builds a Keras model from it with the same functionality and performance. The resulting model contains all essential information about the network, independently of the model library in which the original network was built (e.g. Caffe). This makes the SNN toolbox stable against changes in input formats. Another advantage is extensibility: In order to add a new input language to the toolbox (e.g. Lasagne), a developer only needs to add a single module to `model_libs` package, implementing a number of methods (see the respective functions in 'keras_input_lib.py' for more details.)

Parameters **input_model** (*Analog Neural Network*) – A pretrained neural network model.

Returns **parsed_model** – A Keras model functionally equivalent to `input_model`.

Return type Keras model

`core.util.`**`print_description`**(*snn=None*, *log=True*)
Print a summary of the test run, parameters, and network. If `log==True`, the output is written as `settings.txt` file to the folder given by `settings['log_dir_of_current_run']`.

`core.util.`**`spiketrains_to_rates`**(*spiketrains_batch*)
Convert spiketrains to spikerates.

The output will have the same shape as the input except for the last dimension, which is removed by replacing a sequence of spiketimes by a single rate value.

core.util.**wilson_score**(*p*, *n*)
Confidence interval of a binomial distribution.

See https://en.wikipedia.org/wiki/Binomial_proportion_confidence_interval.

Parameters

- **p** (*float*) – The proportion of successes in n experiments.
- **n** (*int*) – The number of Bernoulli-trials (sample size).

Returns

Return type The confidence interval.

source

cifar10

Helper functions to load, process, augment and save Cifar10 dataset.

Created on Mon Jun 6 12:55:10 2016

@author: rbodo

snntoolbox.io_utils.cifar10_load.**get_cifar10**(*path=None*, *filename=None*, *flat=False*)
Get cifar10 classification dataset.

Values are normalized and saved as float32 type. Class vectors are converted to binary class matrices. Output can be flattened for use in fully-connected networks. Can perform preprocessing using a Keras ImageDataGenerator.

Parameters

- **path** (*string, optional*) – If a path is given, the loaded and modified dataset is saved to path directory.
- **filename** (*string, optional*) – If a path is given, the dataset will be written to filename. If filename is not specified, use cifar10 or cifar10_flat.
- **flat** (*Boolean, optional*) – If True, the output is flattened. Defaults to False.

Returns

- Three compressed files path/filename_X_norm.npz,
- path/filename_X_test.npz, and path/filename_Y_test.npz.
- With data of the form (channels, num_rows, num_cols), X_norm and
- X_test have dimension (num_samples, channels*num_rows*num_cols)
- in case flat==True, and (num_samples, channels, num_rows, num_cols)
- otherwise. Y_test has dimension (num_samples, num_classes).

caltech101

Helper functions to load, process, augment and save Caltech101 dataset.

Created on Mon Jun 6 12:55:20 2016

@author: rbodo

snntoolbox.io_utils.caltech101_load.**get_caltech101**(*path*, *filename=None*)

Get caltech101 classification dataset.

Values are normalized and saved as `float32` type. Class vectors are converted to binary class matrices. Output can be flattened for use in fully-connected networks. Can perform preprocessing using a Keras ImageDataGenerator.

Parameters

- **path** (*string, optional*) – If a `path` is given, the loaded and modified dataset is saved to `path` directory.
- **filename** (*string, optional*) – Basename of file to create. Individual files will be appended `_X_norm`, `_X_test`, etc.

Returns

- Three compressed files `path/filename_X_norm.npz`,
- `path/filename_X_test.npz`, and `path/filename_Y_test.npz`.
- With data of the form (channels, num_rows, num_cols), `X_norm` and
- `X_test` have dimension (num_samples, channels, num_rows, num_cols).
- `Y_test` has dimension (num_samples, num_classes).

mnist

Helper functions to load, process, augment and save MNIST dataset.

Created on Mon Jun 6 12:54:49 2016

@author: rbodo

snntoolbox.io_utils.mnist_load.**get_mnist**(*path=None*, *filename=None*, *flat=False*)

Get mnist classification dataset.

Values are normalized and saved as `float32` type. Class vectors are converted to binary class matrices. Output can be flattened for use in fully-connected networks.

Parameters

- **path** (*string, optional*) – If a `path` is given, the loaded and modified dataset is saved to `path` directory.
- **filename** (*string, optional*) – If a `path` is given, the dataset will be written to `filename`. If `filename` is not specified, use `mnist` or `mnist_flat`.
- **flat** (*Boolean, optional*) – If `True`, the output is flattened. Defaults to `False`.

Returns

- *The dataset as a tuple containing the training and test sample arrays*
- *(X_train, Y_train, X_test, Y_test).*
- With data of the form (channels, num_rows, num_cols), `X_train` and
- `X_test` have dimension (num_samples, channels*num_rows*num_cols)
- in case `flat==True`, and
- *(num_samples, channels, num_rows, num_cols) otherwise.*
- `Y_train` and `Y_test` have dimension (num_samples, num_classes).

facedetection

Helper functions to load, process, augment and save facedetection dataset.

Created on Mon Jun 6 12:55:20 2016

@author: rbodo

snntoolbox.io_utils.facedetection_load.**get_facedetection**(*sourcepath*, *imagepath*, *targetpath=None*, *filename=None*)

Get facedetection dataset.

Values are normalized and saved as `float32` type. Class vectors are converted to binary class matrices. Output can be flattened for use in fully-connected networks.

Parameters

- **sourcepath** (*string*) – Where to find text file containing the filenames and labels of the image samples.
- **imagepath** (*string*) – Path to image folder.
- **targetpath** (*string, optional*) – If a `path` is given, the loaded and modified dataset is saved to `path` directory.
- **filename** (*string, optional*) – If a `path` is given, the dataset will be written to `filename`. If `filename` is not specified, use `facedetection` or `facedetection_flat`.

Returns

- *The dataset as a tuple containing the training and test sample arrays*
- *(X_train, Y_train, X_test, Y_test).*
- With data of the form (channels, num_rows, num_cols), `X_train` and
- `X_test` have dimension (num_samples, channels, num_rows, num_cols).
- `Y_train` and `Y_test` have dimension (num_samples, num_classes).

snntoolbox.io_utils.facedetection_load.**load_paths_from_files**(*sourcepath*, *imagepath*, *filename*)

snntoolbox.io_utils.facedetection_load.**load_samples**(*filepaths*, *nb_samples=None*)

plotting

Various functions to visualize connectivity, activity and accuracy of the network.

Created on Wed Nov 18 13:57:37 2015

@author: rbodo

snntoolbox.io_utils.plotting.**output_graphs**(*spiketrains_batch*, *activations_batch*, *path=None*, *idx=0*)

Wrapper function to display / save a number of plots.

Parameters

- **spiketrains_batch** (*list*) – Each entry in `spiketrains_batch` contains a tuple `(spiketimes, label)` for each layer of the network (for the first batch only, and excluding `Flatten` layers). `spiketimes` is an array where the last index contains the spike times of the specific neuron, and the first indices run over the number of neurons in

the layer: (batch_size, n_chnls, n_rows, n_cols, duration) `label` is a string specifying both the layer type and the index, e.g. `'03Dense'`.

- **activations_batch** (*list*) – Activations of the SNN.
- **path** (*string, optional*) – If not `None`, specifies where to save the resulting image. Else, display plots without saving.
- **idx** (*int, optional*) – The index of the sample to display. Defaults to 0.

snntoolbox.io_utils.plotting.**plot_confusion_matrix**(*Y_test*, *Y_pred*, *path=None*, *class_labels=None*)

snntoolbox.io_utils.plotting.**plot_correlations**(*spikerates*, *layer_activations*)

Plot the correlation between SNN spiketrains and ANN activations.

For each layer, the method draws a scatter plot, showing the correlation between the average firing rate of neurons in the SNN layer and the activation of the corresponding neurons in the ANN layer.

Parameters

- **spikerates** (list of tuples `(spikerate, label)`.) – `spikerate` is a 1D array containing the mean firing rates of the neurons in a specific layer.

  `label` is a string specifying both the layer type and the index, e.g. `'3Dense'`.
- **layer_activations** (list of tuples `(activations, label)`) – Each entry represents a layer in the ANN for which an activation can be calculated (e.g. `Dense`, `Convolution2D`).

  `activations` is an array of the same dimension as the corresponding layer, containing the activations of Dense or Convolution layers.

  `label` is a string specifying the layer type, e.g. `'Dense'`.

snntoolbox.io_utils.plotting.**plot_error_vs_time**(*err*, *path=None*)

snntoolbox.io_utils.plotting.**plot_hist**(*h*, *title=None*, *layer_label=None*, *path=None*, *scale_fac=None*)

Plot a histogram over two datasets.

Parameters

- **h** (*dict*) – Dictionary of datasets to plot in histogram.
- **title** (*string, optional*) – Title of histogram.
- **layer_label** (*string, optional*) – Label of layer from which data was taken.
- **path** (*string, optional*) – If not `None`, specifies where to save the resulting image. Else, display plots without saving.
- **scale_fac** (*float, optional*) – The value with which parameters are normalized (maximum of activations or parameter value of a layer). If given, will be insterted into plot title.

snntoolbox.io_utils.plotting.**plot_hist_combined**(*data*, *path=None*)

Plot a histogram over several datasets.

Parameters

- **data** (*dict*) – Dictionary of datasets to plot in histogram.
- **path** (*string, optional*) – If not `None`, specifies where to save the resulting image. Else, display plots without saving.

snntoolbox.io_utils.plotting.**plot_history**(*h*)

Plot the training and validation loss and accuracy at each epoch.

Parameters **h** (*Keras history object*) – Contains the training and validation loss and accuracy at each epoch during training.

snntoolbox.io_utils.plotting.**plot_layer_activity**(*layer*, *title*, *path=None*, *limits=None*)

Visualize a layer by arranging the neurons in a line or on a 2D grid.

Can be used to show average firing rates of individual neurons in an SNN, or the activation function per layer in an ANN. The activity is encoded by color.

Parameters

- **layer** (*tuple*) – `(activity, label)`.

  `activity` is an array of the same shape as the original layer, containing e.g. the spikerates or activations of neurons in a layer.

  `label` is a string specifying both the layer type and the index, e.g. `'3Dense'`.

- **title** (*string*) – Figure title.
- **path** (*string, optional*) – If not `None`, specifies where to save the resulting image. Else, display plots without saving.
- **limits** (*tuple, optional*) – If not `None`, the colormap of the resulting image is limited by this tuple.

snntoolbox.io_utils.plotting.**plot_layer_correlation**(*rates*, *activations*, *title*, *path=None*)

Plot correlation between spikerates and activations of a specific layer, as 2D-dot-plot.

Parameters

- **rates** (*list*) – The spikerates of a layer, flattened to 1D.
- **activations** (*list*) – The activations of a layer, flattened to 1D.
- **title** (*string*) – Plot title.
- **path** (*string, optional*) – If not `None`, specifies where to save the resulting image. Else, display plots without saving.

snntoolbox.io_utils.plotting.**plot_layer_summaries**(*spikerates*, *activations*, *spiketrains=None*, *path=None*)

Display or save a number of plots for a specific layer.

Parameters

- **spikerates** (*list*) – Each entry in `spikerates` contains a tuple `(rates, label)` for each layer of the network (for the first batch only, and excluding `Flatten` layers).

  `rates` contains the average firing rates of all neurons in a layer. It has the same shape as the original layer, e.g. (n_features, n_rows, n_cols) for a convolution layer.

  `label` is a string specifying both the layer type and the index, e.g. `'03Dense'`.

- **activations** (*list*) – Contains the activations of a net. Same structure as `spikerates`.
- **spiketrains** (*list, optional*) – Each entry in `spiketrains` contains a tuple `(spiketimes, label)` for each layer of the network (for the first batch only, and excluding `Flatten` layers).

spiketimes is an array where the last index contains the spike times of the specific neuron, and the first indices run over the number of neurons in the layer: (n_chnls, n_rows, n_cols, duration)

label is a string specifying both the layer type and the index, e.g. '03Dense'.

- **path** (*string, optional*) – If not None, specifies where to save the resulting image. Else, display plots without saving.

snntoolbox.io_utils.plotting.**plot_param_sweep**(*results*, *n*, *params*, *param_name*, *param_logscale*)

Plot accuracy versus parameter.

Parameters

- **results** (*list*) – The accuracy or loss for a number of experiments, each of which used different parameters.
- **n** (*int*) – The number of test samples used for each experiment.
- **params** (*list*) – The parameter values that changed during each experiment.
- **param_name** (*string*) – The name of the parameter that varied.
- **param_logscale** (*boolean*) – Whether to plot the parameter axis in log-scale.

snntoolbox.io_utils.plotting.**plot_pearson_coefficients**(*spikerates_batch*, *activations_batch*, *path=None*)

Plot the Pearson correlation coefficients for each layer, averaged over one mini batch.

Parameters

- **spikerates_batch** (*list*) – Each entry in spikerates_batch contains a tuple (spikerates, label) for each layer of the network (for the first batch only, and excluding Flatten layers).

  spikerates contains the average firing rates of all neurons in a layer. It has the same shape as the original layer, e.g. (batch_size, n_features, n_rows, n_cols) for a convolution layer.

  label is a string specifying both the layer type and the index, e.g. '03Dense'.

- **activations_batch** (*list*) – Contains the activations of a net. Same structure as spikerates_batch.

snntoolbox.io_utils.plotting.**plot_potential**(*times*, *layer*, *showLegend=False*, *path=None*)

Plot the membrane potential of a layer.

Parameters

- **times** (*1D array*) – The time values where the potential was sampled.
- **layer** (*tuple*) – (vmem, label).

  vmem is a 2D array where the first index runs over the number of neurons in the layer, and the second index contains the membrane potential of the specific neuron.

  label is a string specifying both the layer type and the index, e.g. '3Dense'.

- **showLegend** (*boolean, optional*) – If True, shows the legend indicating the neuron indices and lines like v_thresh, v_rest, v_reset. Recommended only for layers with few neurons.
- **path** (*string, optional*) – If not None, specifies where to save the resulting image. Else, display plots without saving.

snntoolbox.io_utils.plotting.**plot_rates_minus_activations**(*rates*, *activations*, *label*, *path=None*)

Plot spikerates minus activations for a specific layer.

Spikerates and activations are each normalized before subtraction. The neurons in the layer are arranged in a line or on a 2D grid, depending on layer type.

Activity is encoded by color.

Parameters

- **rates** (*array*) – The spikerates of a layer. The shape is that of the original layer, e.g. (32, 28, 28) for 32 feature maps of size 28x28.
- **activations** (*array*) – The activations of a layer. The shape is that of the original layer, e.g. (32, 28, 28) for 32 feature maps of size 28x28.
- **label** (*string*) – Layer label.
- **path** (*string, optional*) – If not `None`, specifies where to save the resulting image. Else, display plots without saving.

snntoolbox.io_utils.plotting.**plot_spiketrains**(*layer*, *path=None*)

Plot which neuron fired at what time during the simulation.

Parameters

- **layer** (*tuple*) – `(spiketimes, label)`.

  `spiketimes` is a 2D array where the first index runs over the number of neurons in the layer, and the second index contains the spike times of the specific neuron.

  `label` is a string specifying both the layer type and the index, e.g. `'3Dense'`.
- **path** (*string, optional*) – If not `None`, specifies where to save the resulting image. Else, display plots without saving.

common

Functions to load various properties of interest in analog and spiking neural networks from disk.

Created on Wed Nov 18 13:38:46 2015

@author: rbodo

snntoolbox.io_utils.common.**confirm_overwrite**(*filepath*)

If settings['overwrite']==False and the file exists, ask user if it should be overwritten.

snntoolbox.io_utils.common.**download_dataset**(*fname*, *origin*, *untar=False*)

Download a dataset, if not already there.

Parameters

- **fname** (*string*) – Full filename of dataset, e.g. `mnist.pkl.gz`.
- **origin** (*string*) – Location of dataset, e.g. url https://s3.amazonaws.com/img-datasets/mnist.pkl.gz
- **untar** (*boolean, optional*) – If `True`, untar file.

Returns fpath – The path to the downloaded dataset. If the user has write access to `home`, the dataset will be stored in `~/.snntoolbox/datasets/`, otherwise in `/tmp/.snntoolbox/datasets/`.

Return type string

Todo

Test under python2.

snntoolbox.io_utils.common.**load_dataset**(*path*, *filename*)
Load dataset from an `.npy` or `.npz` file.

Parameters **path** (*string, optional*) – Location of dataset to load.

Returns

- *The dataset as a numpy array containing samples. Example*
- With original data of the form (channels, num_rows, num_cols), `X_train`
- and `X_test` have dimension (num_samples, channels*num_rows*num_cols) for
- *a fully-connected network, and (num_samples, channels, num_rows, num_cols)*
- *otherwise.*
- `Y_train` and `Y_test` have dimension (num_samples, num_classes).

snntoolbox.io_utils.common.**load_parameters**(*filepath*)
Load all layer parameters from an HDF5 file.

snntoolbox.io_utils.common.**to_categorical**(*y*, *nb_classes*)
Convert class vector to binary class matrix.

If the input `y` has shape (`nb_samples`,) and contains integers from 0 to `nb_classes`, the output array will be of dimension (`nb_samples`, `nb_classes`).

snntoolbox.io_utils.common.**to_json**(*data*, *path*)
Write `data` dictionary to `path`.

A `TypeError` is raised if objects in `data` are not JSON serializable.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

C

S

R

S

T

U

W