## Contents

1  User Guide                        3  
2  API Reference                    21  
3  Indices and tables               99  
Python Module Index                101  
Index                               103  

The SNN conversion toolbox contains functions to transform rate-based artificial neural networks into spiking neural networks, and to simulate them. The source code can be found on GitHub. See also the accompanying article.
These sections guide you through the installation, configuration and running of the toolbox. Examples are included.

1.1 Spiking neural network conversion toolbox

1.1.1 Introduction

Artificial neural networks have been studied and used extensively to solve tasks from machine learning and artificial intelligence. Deep Learning has developed increasingly large neural networks, spanning up to thousands of layers and millions of neurons. These networks have proven to be very successful in solving challenging tasks like object detection and recognition, scene segmentation and parsing, video classification, etc. The downside is that running such large networks requires massive amounts of computational resources.

Our research group at the University of Zurich and ETH Zurich develops Spiking Neural Networks (SNNs) that perform the same task but with potentially less computations and energy consumption. The fundamental idea is that in a spiking network, all computation is event-driven, meaning that operations are sparse and occur only when significant changes in the input make them necessary.

Training a deep spiking network (i.e. learning the synaptic weights) is difficult. An alternative approach is to take a pre-trained neural network and convert it into a spiking neural network. We call the original network Analog Neural Network (ANN) because its activations are real-valued, representing spike-rates. In this ANN-to-SNN conversion, we use the weights of the ANN and replace the analog (rate) neurons of the ANN by simple Integrate-and-Fire spiking neurons. This works because over the course of the simulation, the average firing rate of the SNN neurons will approximate the activation of the corresponding neurons in the original ANN. See Citation for details.

This toolbox automates the conversion of pre-trained analog to spiking neural networks (ANN to SNN), and provides tools for testing the SNNs in a spiking neuron simulator.

1.1.2 Internal workflow
Chapter 1. User Guide
**Parsing and converting**

Given a model written in some neural network library, the toolbox parses the provided network files by extracting the relevant information and creating an equivalent Keras model from it. This parsed model serves as common abstraction stage from the input and is internally used by the toolbox to perform the actual conversion to a spiking network.

The conversion toolbox currently supports input networks generated with Keras, Lasagne, or Caffe. See *Extending the toolbox* on how to extend the relevant methods to handle models from other common libraries like Torch.

The following table lists the input files expected by the toolbox.

<table>
<thead>
<tr>
<th>Input library</th>
<th>Keras</th>
<th>Lasagne</th>
<th>Caffe</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Required files</strong></td>
<td>*.h5 (weights and model specs), or: *.h5 (only weights) and *.json (model specs)</td>
<td>*.h5 or *.pkl (weights) and *.py (script containing a method that returns the compiled model and an evaluation function)</td>
<td>*.caffemodel (weights) and *.prototxt (model specs)</td>
</tr>
</tbody>
</table>

The second column in the following table summarized which features of the input model the toolbox knows about and can handle.

**Simulating**

After the input model has been converted, the resulting spiking network can be exported for simulation in a spiking simulator or deployment on dedicated spiking neuron chips. Currently, the following output formats are supported (see *Extending the toolbox* on how to add a simulator to the toolbox):

- **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. If you choose a pyNN simulator backend, we recommend Nest with version 2.14.

- **Brian2.**

- The toolbox integrates MegaSim, an event-driven asynchronous spiking simulator developed at the University of Seville.

- The toolbox provides a built-in simulator based on Keras, called INIsim. This simulator features a 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. This simulator backend is recommended as it supports the most features and its integration is maintained best.

The second column in the table below compares these different simulators with respect to the network features that can be implemented on them.

Additionally, a number of experimental features were implemented to improve and test the spiking network. They are currently only available in INIsim, and include:

- Clamping of membrane potentials for a given time for each layer.
- Clipping membrane potentials to certain bounds.
- Activity-dependent adaptation of spike thresholds of each layer.
- Bias-relaxation.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Fully-connected</td>
<td>All</td>
<td>All</td>
<td>All</td>
<td>All</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Convolutional</td>
<td>All</td>
<td>All</td>
<td>All</td>
<td>All</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max-Pooling</td>
<td>All</td>
<td>All</td>
<td>I¹</td>
<td>All²</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average-Pooling</td>
<td>All</td>
<td>All</td>
<td>All</td>
<td>All</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Batch-Normalization</td>
<td>K, L, C²</td>
<td>All³</td>
<td>All⁴</td>
<td>All⁴</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dropout</td>
<td>All</td>
<td>All</td>
<td>All⁴</td>
<td>All⁴</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flatten</td>
<td>All</td>
<td>All</td>
<td>All</td>
<td>All</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Merge / Concatenate (Inception modules)</td>
<td>K, L</td>
<td>All⁴</td>
<td>All⁴</td>
<td>All⁴</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linear activation</td>
<td>All</td>
<td>All</td>
<td>Repl. by ReLU</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ReLU activation</td>
<td>All</td>
<td>All</td>
<td>All</td>
<td>All</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Softmax activation</td>
<td>All</td>
<td>All</td>
<td>I⁵</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary activation {-1, 1} or {0, 1}</td>
<td>L</td>
<td>All</td>
<td>I</td>
<td>All</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary weights {-1, 1}</td>
<td>L</td>
<td>All</td>
<td>All</td>
<td>All</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-zero biases</td>
<td>All</td>
<td>All</td>
<td>I</td>
<td>All</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

¹ May decrease accuracy in large networks.
² Implemented for Caffe, but not fully tested.
³ Parameters frozen during inference.
⁴ Not used during inference.
⁵ Spiking softmax activation is currently implemented only in INInsim. In all other simulators, softmax is replaced by ReLU.
• Spikes with payloads.
• Various reset mechanisms after spikes.
• Logging and visualization functions to monitor and debug simulation.
• Different input types: In all other simulators, only Poisson input is supported at the moment. INIsim implements constant input currents as well as input from DVS event sequences.
• Batch-wise testing.

1.1.3 GUI (beta)

Note: The GUI has not been maintained since 2017 and is most likely broken.

1.2 Installation

1.2.1 Release version

Run `pip install snntoolbox`. This will install the minimum dependencies needed to get started. Optional dependencies like matplotlib and scipy enable generating output plots.

The toolbox relies on Keras internally, and by default installs Tensorflow as Keras backend. You may want to update this backend to optimally fit your system, e.g. to use the GPU.

Note: The SNN toolbox provides a built-in simulator to run the converted network. This simulator is Keras-based and will use either Tensorflow or Theano as backend. Depending on the backend you choose, different features are available in the toolbox simulator. You can install both backends and switch between them simply by setting the corresponding parameter in the `config file`:

```ini
[simulation]
keras_backend = tensorflow
```

1.2.2 Development version (recommended)

To get the latest version, checkout the repository. In the toolbox root directory `snn_toolbox/`, run `pip install ..`

Note: Using `easy_install` via `python setup.py install` has been reported to fail on some platforms due to dependency issues.

1.2.3 Additional tools

For testing a converted network, the toolbox includes a ready-to-use spiking simulator. In addition, you may install and use one of the simulators described here.
Fig. 1: SNN toolbox GUI. In the main window, the user can specify which tools to use during the experiment. 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: 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.
1.3 Getting started

In a terminal window, type `snntoolbox -h` to get all command-line options.

The basic use is:

```
snntoolbox <config-file> -t
```

The positional argument `<config-file>` specifies the path (including file name) to a plain text file containing the settings to be used. Refer to `Configuration` for details on what this file looks like.

With `-t`, we tell the program to stay in the terminal. Omitting this flag opens the GUI (not actively developed).

Instead of using the terminal, you may also invoke the toolbox within a python script:

```
snntoolbox.bin.run.main(path_to_config_file)
```

See `Examples` for typical usecases.

1.4 Configuration

To configure the toolbox for a specific experiment, create a plain text file and add the parameters you want to set, using INI/conf file syntax. See also our `Examples`. Any settings you do not specify will be filled in with the `default values`.

When starting the toolbox, you may pass the location of this settings file as argument to the program.

The toolbox settings are grouped in the following categories:

1.4.1 [paths]

- **path_wd**: str, optional  Working directory. There, the toolbox will look for ANN models to convert or SNN models to test. If not specified, the toolbox will try to create and use the directory `~/.snntoolbox/data/<filename_ann>/<simulator>/`.

- **dataset_path**: str  Select a directory where the toolbox will find the samples to test. See `dataset_format` for supported input types.

- **log_dir_of_current_run**: str, optional  Path to directory where the output plots and logs are stored. Default: `<path_wd>/log/gui/<runlabel>`.

- **runlabel**: str, optional  Label of current experiment. Default: ‘test’.

- **filename_ann**: str  Name of ANN model to be converted.

- **filename_parsed_model**: string, optional  Name given to parsed SNN model. Default: `<filename_ann>_parsed`.

- **filename_snn**: str, optional  Name given to converted spiking model when exported to test it in a specific simulator. Default: `<filename_ann>_<simulator>`.

Note: Depending on the simulator you use, we recommend installing the toolbox in a virtual environment, because simulators supported by pyNN may require different versions of their dependencies (Brian for instance only works with python2).
filename_clamp_indices: str, optional  Name of file containing a dictionary of clamp indices. Each key specifies a layer index, and the corresponding value defines the number of time steps during which the membrane potential of neurons in this layer are clamped to zero. If this option is not specified, no layers are clamped.

class_idx_path: str, optional  Only needed if the data set is stored as images in folders denoting the class label (i.e. dataset_format = jpg below). Then class_idx_path is the path to a file containing a dictionary that maps the class labels to the corresponding indices of neurons in the output layer.

1.4.2 [input]

model_lib: str  The neural network library used to build the ANN. Currently supported:

- keras
- lasagne
- caffe

dataset_format: str, optional  The following input formats are supported:

A) npz: Compressed numpy format.
B) jpg: Images in directories corresponding to their class.
C) aedat: Sequence of address-events recorded from a Dynamic Vision Sensor.

A) Default. Provide at least two compressed numpy files called x_test.npz and y_test.npz containing the test set and ground truth. In addition, if the network should be normalized, put a file x_norm.npz in the folder. This can be a the training set, or a subset of it. Take care of memory limitations: If numpy can allocate a 4 GB float32 container for the activations to be computed during normalization, x_norm should contain not more than 4*1e9*8bit/(fc*fx*fy*32bit) = 1/n samples, where (fc, fx, fy) is the shape of the largest layer, and n = fc*fx*fy its total cell count.

B) The images are stored in subdirectories of the selected dataset_path, where the names of the subdirectories represent their class label. The toolbox will then use keras.preprocessing.image.ImageDataGenerator to load and process the files batchwise. Setting jpg here works even if the images are actually in .png or .bmp format.

C) Beta stage.

datagen_kwargs: str, optional  Specify keyword arguments for the data generator that will be used to load image files from subdirectories in the dataset_path. Need to be given in form of a python dictionary. See keras.preprocessing.image.ImageDataGenerator for possible values.

dataflow_kwargs: str, optional  Specify keyword arguments for the data flow that will get the samples from the ImageDataGenerator. Need to be given in form of a python dictionary. See keras.preprocessing.image.ImageDataGenerator.flow_from_directory for possible values.

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 200, a fully-on pixel will elicit a Poisson spiketrain of 200 Hz. Turn off for a less noisy simulation. Currently, turning off Poisson input is only possible in INI simulator.

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).

num_poisson_events_per_sample: int, optional  Limits the number of Poisson spikes generated from each frame. Default: -1 (unlimited).
num_dvs_events_per_sample: int, optional  Number of DVS events used in one image classification trial. Can be thought of as being equivalent to one frame. Default: 2000.

eventframe_width: int, optional  To be able to use asynchronous DVS events in our time-stepped simulator, we collect them into frames (binary maps) which are presented to the SNN at subsequent time steps. The option eventframe_width defines how many timesteps the timestamps of events in such a frame should span at most. Default: 10.

label_dict: dict  Dictionary containing the class labels. Only needed with .aedat input.

chip_size: tuple  When using .aedat input, the addresses can be checked for outliers, or may have to be subsampled from the original chip_size to the image dimension required by the network. Set chip_size to the shape of the DVS chip that was used to record the aedat sample, e.g. (240, 180). The image dimension to subsample to will be inferred from the shape of the input layer of the network.

frame_gen_method: str  How to accumulate DVS events into frames.
  - signed_sum: DVS events are added up while their polarity is taken into account. (ON and OFF events cancel each other out.)
  - rectified_sum: Polarity is discarded; all events are considered ON.

is_x_first: bool  Whether the x-address of a DVS events is considered as the first dimension when accumulating events into 2-D frame.

is_x_flipped: bool  Whether to reflect DVS image through vertical axis.

is_y_flipped: bool  Whether to reflect DVS image through horizontal axis.

1.4.3 [tools]

evaluateANN: bool, optional  If enabled, the ANN is tested at two stages:
  1. At the very beginning, using the input model as provided by the user.
  2. After parsing the input model to our internal Keras representation, and applying any optional modifications like replacing Softmax activation by ReLU, replacing MaxPooling by AveragePooling, and normalizing the network parameters.

This ensures all operations on the ANN preserve the accuracy.

parse: bool, optional  If enabled, the input ANN is parsed for further processing in the SNN toolbox. Needs to be done the first time you run your model. The toolbox then saves the parsed model and, if parse=False, will load it the next time.

normalize: bool, optional  If enabled, the parameters of each layer will be normalized by the highest activation value, or by the n-th percentile (see parameter percentile below).

convert: bool, optional  If enabled, load an ANN from path_wd and convert it to spiking.

simulate: bool, optional  If enabled, load SNN from path_wd and test it on the specified simulator (see parameter simulator).

1.4.4 [normalization]

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.

normalization_schedule: bool, optional  Reduce the normalization factor each layer.
**online_normalization**: bool, 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.

### 1.4.5 [conversion]

**softmax_to_relu**: bool, optional  If True, replace softmax by ReLU activation function. This is recommended (default), because the spiking softmax implementation tends to reduce accuracy, especially top-5. It is safe to do this replacement as long as the input to the activation function is not all negative. In that case, the ReLU would not be able to determine the winner.

**maxpool_type**: str, optional  Implementation variants of spiking MaxPooling layers, based on

- **fir_max**: accumulated absolute firing rate (default)
- **avg_max**: moving average of firing rate
- **exp_max**: exponential FIR filter.

**max2avg_pool**: bool, optional  If True, max pooling layers are replaced by average pooling.

**spike_code**: str, optional  Describes the code used to transform analog activation values of the original network into spikes.

- **temporal_mean_rate** (default): Average over number of spikes that occur during simulation duration.
- **temporal_pattern**: Analog activation value is transformed into binary representation of spikes.
- **ttfs**: Instantaneous firing rate is given by the inverse time-to-first-spike.
- **ttfs_dyn_thresh**: Like ttfs, but with a threshold that adapts dynamically to the amount of input a neuron has received.
- **ttfs_corrective**: Allows corrective spikes to be fired to improve the first guess made by ttfs.

**num_bits**: int, optional  Bit-resolution that a binary spike train can maximally encode when using spike_code = temporal_pattern.

### 1.4.6 [simulation]

**simulator**: str, 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.

**num_to_test**: int, optional  How many samples to test.
**sample_idxs_to_test**: `Iterable, optional`  List of sample indices to test.

**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.

**reset_between_nth_sample**: `int, optional`  When testing a video sequence, this option allows turning off the reset between individual samples. Default: 1 (reset after every frame). Set to a negative value to turn off reset completely.

**top_k**: `int, optional`  In addition to the top-1 error, report top_k error during simulation. Default: 5.

**keras_backend**: `str, optional`  The backend to use in INI simulator.

- **theano**: Only works in combination with `spike_code = temporal_mean_rate`.
- **tensorflow**: Does not implement the spiking MaxPool layer when using `spike_code = temporal_mean_rate`.

### 1.4.7 [cell]

**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.

**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.

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

**binarize_weights**: `bool, optional`  If True, the weights are binarized.

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

**payloads**: `bool, optional`  Whether or not to send a float value together with each spike.

**reset**: `str, 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.
- ‘Reset by modulo’: After spike, the membrane potential is reduced by the smallest multiple of the threshold such that the new membrane potential is below threshold.

**leak**: `bool, optional`  Experimental feature. False by default.

### 1.4.8 [parameter_sweep]

Enables running the toolbox with the same settings except for one parameter being varied. In beta stadium.

**param_values**: `list, optional`  Contains the parameter values for which the simulation will be repeated.

**param_name**: `str, optional`  Label indicating the parameter to sweep, e.g. 'v_thresh'.
param_logscale: bool, optional  If True, plot test accuracy vs params in log scale.

1.4.9 [output]


verbose: int, optional  If nonzero (default), print current error rate at every time step during simulation.

overwrite: bool, optional  If False, the save methods will ask for permission to overwrite files before writing parameters, activations, models etc. to disk. Default: True.

plotproperties: dict, optional  Options that modify matplotlib plot properties.

1.4.10 SNN toolbox default settings

```python
[paths]
path_wd =
dataset_path =
log_dir_of_current_run =
runlabel = test
filename_ann =
filename_parsed_model =
filename_snn =
filename_clamp_indices =
filepath_custom_objects =
class_idx_path =

[input]
model_lib = keras
dataset_format = npz
datagen_kwargs = {}
dataflow_kwargs = {}
poisson_input = False
input_rate = 1000
num_poisson_events_per_sample = -1
num_dvs_events_per_sample = 2000
eventframe_width = 10
label_dict = {}
chip_size = None
frame_gen_method =
is_x_first =
is_x_flipped =
is_y_flipped =
maxpool_subsampling = True
do_clip_three_sigma = True

[tools]
evaluate_ann = True
parse = True
normalize = True
```

(continues on next page)
convert = True
simulate = True

[normalization]
percentile = 99.9
normalization_schedule = False
online_normalization = False
diff_to_max_rate = 200
diff_to_min_rate = 100
timestep_fraction = 10

[conversion]\nsoftmax_to_relu = False
maxpool_type = fir_max
max2avg_pool = False
spike_code = temporal_mean_rate
num_bits = 32

[simulation]
simulator = INI
duration = 200
dt = 1
batch_size = 1
num_to_test = 1
sample_idxs_to_test = []
reset_between_nth_sample = 1
top_k = 1
keras_backend = tensorflow
early_stopping = False

[cell]
v_thresh = 1
tau_refrac = 0
v_reset = 0
v_rest = 0
cm = 1
tau_m = 1000
tau_syn_E = 0.01
tau_syn_I = 0.01
delay = 0
binarize_weights = False
quantize_weights = False
scaling_factor = 1000000
payloads = False
reset = Reset by subtraction
leak = False
bias_relaxation = False

[parameter_sweep]
param_values = []
param_name = v_thresh
param_logscale = False

[output]
log_vars = {}
plot_vars = {}
verbose = 1
overwrite = True
use_simple_labels = True
plotproperties = {
    'font.size': 13,
    'axes.titlesize': 'xx-large',
    'axes.labelsize': 'xx-large',
    'xtick.labelsize': 'xx-large',
    'xtick.major.size': 7,
    'xtick.minor.size': 5,
    'ytick.labelsize': 'xx-large',
    'ytick.major.size': 7,
    'ytick.minor.size': 5,
    'legend.fontsize': 'xx-large',
    'figure.figsize': (7, 6),
    'savefig.format': 'png'}

# Use the following section to specify sets of possible values that certain
# config settings may accept. Will be used in `bin.utils.update_setup` to test
# validity of config.

[restrictions]
model.libs = {'keras', 'lasagne', 'caffe'}
dataset_formats = {'npz', 'jpg', 'aedat'}
frame_gen_method = {'signed_sum', 'rectified_sum'}
maxpool_types = {'fir_max', 'exp_max', 'avg_max'}
simulators.pyNN = {'nest', 'brian', 'neuron'}
simulators_other = {'INI', 'brian2', 'MegaSim', 'loihi'}
simulators = %s(simulators.pyNN)s | %s(simulators_other)s

# Keras backends:
keras_backends = {'theano', 'tensorflow'}

# Spike coding mechanisms:
spike_codes = {'temporal_mean_rate', 'temporal_pattern', 'ttfs',
               'ttfs_dyn_thresh', 'ttfs_corrective'}

# Layers that have neurons with membrane potentials, from which we can measure spikes:
spiking_layers = {'Dense', 'Conv2D', 'MaxPooling2D', 'AveragePooling2D',
                  'DepthwiseConv2D', 'Conv1D'}

# Layers that can be implemented by our spiking neuron simulators:
snn_layers = %s(spiking_layers)s | {'Reshape', 'Flatten', 'Concatenate', 'ZeroPadding2D',
                                      'DepthwiseConv2D', 'Conv1D'}

cellparams.pyNN = {'v_thres', 'v_reset', 'v_rest', 'cm', 'tau_refrac',
                   'tau_m', 'tau_syn_E', 'tau_syn_I'}

log_vars = {'activations_n_b_l', 'spiketrains_n_b_l_l_t', 'input_b_l_l_t',
            'mem_n_b_l_t', 'synaptic_operations_b_t', 'neuron_operations_b_t',
            'all'}

plot_vars = {'activations', 'spiketrains', 'spikecounts', 'spikerates',
             'input_image', 'error_t', 'confusion_matrix', 'correlation',
             'hist_spikerates_activations', 'normalization_activations',
             'operations', 'v_mem', 'all'}
1.5 Extending the toolbox

1.5.1 Input side: Adding a new model library

The philosophy behind the toolbox 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. Torch) to the toolbox, put a module named torch_input_lib into the snntoolbox.parsing.model_libs package. Then create a child class torch_input_lib.ModelParser inheriting from snntoolbox.parsing.utils.AbstractModelParser, and implement the abstract methods tailored to the new input model library.

1.5.2 Output side: Adding a custom simulator

Similarly, adding another simulator to run converted networks implies adding a file to the snntoolbox.simulation.target_simulators package. Each file in there allows 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 create a child class SNN inheriting from AbstractSNN, and implement the abstract methods tailored to the ‘custom’ simulator.

1.6 Examples

To set up an experiment with the SNN toolbox, we need to create a config file as described in settings. The examples subdirectory of the repository root provides a number of end-to-end tutorials covering how to

- set up a small CNN using Keras and tensorflow,
- train the model on MNIST,
- store model and dataset in a temporary folder on disk,
- create a configuration file for SNN toolbox,
- call the main function of SNN toolbox to convert the trained ANN to an SNN and run it using some simulator backend.

More examples are included in a stand-alone applications repository.

1.7 Citation

If you use this work in your research, please cite our corresponding publication:


1.8 Support

For questions and feedback, please write to
Bodo Rueckauer
rbodo(at)ini(dot)uzh(dot)ch
We are happy to hear from you!
Here you find detailed descriptions of specific functions and classes.

### 2.1 snntoolbox.bin

#### 2.1.1 snntoolbox.bin.run

The purpose of this module is to provide an executable for running the SNN conversion toolbox, either from terminal or using a GUI.

During installation of the toolbox, python creates an entry point to the `main` function of this module. See *Getting started* for how call this executable.

@author: rbodo

```python
snntoolbox.bin.run.main(filepath=None)
```

Entry point for running the toolbox.

**Note:** There is no need to call this function directly, because python sets up an executable during *Installation* that can be called from terminal.

#### 2.1.2 snntoolbox.bin.utils

This module bundles all the tools of the SNN conversion toolbox.

Important functions:

- `run_pipeline` Convert an analog network to a spiking network and simulate it.

Continued on next page
Table 1 – continued from previous page

**update_setup**

Update default settings with user settings and check they are valid.

@author: rbodo

```python
snntoolbox.bin.utils.run_pipeline(config, queue=None)
```

Convert an analog network to a spiking 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. given a specified hyperparameter range `params`, repeat simulations with modified parameters.

**Parameters**

- `config (configparser.ConfigParser)` – ConfigParser containing the user settings.
- `queue (Optional[Queue.Queue])` – Results are added to the queue to be displayed in the GUI.

**Returns**

- `results` – List of the accuracies obtained after simulating with each parameter value in `config.get('parameter_sweep', 'param_values')`.

**Return type** 

`list`

```python
snntoolbox.bin.utils.is_stop(queue)
```

Determine if the user pressed ‘stop’ in the GUI.

**Parameters**

`queue (Queue.Queue)` – Event queue.

**Returns**

- `True` if user pressed ‘stop’ in GUI, `False` otherwise.

**Return type** 

`bool`

```python
snntoolbox.bin.utils.run_parameter_sweep(config, queue)
```

Decorator to perform a parameter sweep using the `run_single` function. Need an additional wrapping layer to be able to pass decorator arguments.

```python
snntoolbox.bin.utils.import_target_sim(config)
```

Load a config file from `filepath`.

```python
snntoolbox.bin.utils.update_setup(configfilepath)
```

Update default settings with user settings and check they are valid.

- Load settings from configuration file at `config_filepath`, and check that parameter choices are valid. Non-specified settings are filled in with defaults.

```python
snntoolbox.bin.utils.initialize_simulator(config)
```

Import a module that contains utility functions of spiking simulator.

```python
snntoolbox.bin.utils.get_log_keys(config)
```

```python
snntoolbox.bin.utils.get_plot_keys(config)
```
2.1.3 snntoolbox.bin.gui

2.2 snntoolbox.parsing

On the input side of the SNN conversion toolbox, models from the following neural network libraries can be parsed:

- `snntoolbox.parsing.model_libs.keras_input_lib` (Keras model parser)
- `snntoolbox.parsing.model_libs.lasagne_input_lib`
- `snntoolbox.parsing.model_libs.caffe_input_lib` (Caffe model parser)

These parsers inherit from `AbstractModelParser` in `snntoolbox.parsing.utils`.

See `Extending the toolbox` on how to extend the toolbox by another input model library.

2.2.1 snntoolbox.parsing.model_libs

`keras_input_lib`

Keras model parser.

@author: rbodo

class snntoolbox.parsing.model_libs.keras_input_lib.ModelParser(input_model, config):

    Bases: snntoolbox.parsing.utils.AbstractModelParser

    get_layer_iterable() – Get an iterable over the layers of the network.

        Returns layers
        Return type list

    get_type(layer) – Get layer class name.

        Returns layer_type – Layer class name.
        Return type str

    get_batchnorm_parameters(layer) – Get the parameters of a batch-normalization layer.

        Returns mean, var_eps_sqrt_inv, gamma, beta, axis
        Return type tuple
get_inbound_layers (layer)
Get inbound layers of layer.

Returns inbound
Return type Sequence

layers_to_skip
Return a list of layer names that should be skipped during conversion to a spiking network.

Returns self._layers_to_skip
Return type List[str]

has_weights (layer)
Return True if layer has weights.

initialize_attributes (layer=None)
Return a dictionary that will be used to collect all attributes of a layer. This dictionary can then be used to instantiate a new parsed layer.

get_input_shape ()
Get the input shape of a network, not including batch size.

Returns input_shape – Input shape.
Return type tuple

get_output_shape (layer)
Get output shape of a layer.

Parameters layer – Layer.

Returns output_shape – Output shape of layer.
Return type Sized

parse_dense (layer, attributes)
Parse a fully-connected layer.

Parameters
• layer – Layer.
• attributes (dict) – The layer attributes as key-value pairs in a dict.

parse_convolution (layer, attributes)
Parse a convolutional layer.

Parameters
• layer – Layer.
• attributes (dict) – The layer attributes as key-value pairs in a dict.

parse_depthwiseconvolution (layer, attributes)
Parse a depthwise convolution layer.

Parameters
• layer – Layer.
• attributes (dict) – The layer attributes as key-value pairs in a dict.

parse_pooling (layer, attributes)
Parse a pooling layer.

Parameters
• **layer** – Layer.

• **attributes** (*dict*) – The layer attributes as key-value pairs in a dict.

### `get_activation(layer)`
Get the activation string of an activation layer.

**Parameters**

- **layer** – Layer

**Returns**

- **activation** – String indicating the activation of the layer.

**Return type** `str`

### `get_outbound_layers(layer)`
Get outbound layers of `layer`.

**Parameters**

- **layer** – Layer.

**Returns**

- **outbound** – Outbound layers of `layer`.

**Return type** `list`

### `parse_concatenate(layer, attributes)`
Parse a concatenation layer.

**Parameters**

- **layer** – Layer.

- **attributes** (*dict*) – The layer attributes as key-value pairs in a dict.

### `snntoolbox.parsing.model_libs.keras_input_lib.load(path, filename, **kwargs)`
Load network from file.

**Parameters**

- **path** (*str*) – Path to directory where to load model from.

- **filename** (*str*) – Name of file to load model from.

**Returns**

A dictionary of objects that constitute the input model. It must contain the following two keys:

- **'model': keras.models.Sequential** Keras model instance of the network.

- **'val_fn': function** Function that allows evaluating the original model.

**Return type** `dict[str, Union[keras.models.Sequential, function]]`

### `snntoolbox.parsing.model_libs.keras_input_lib.evaluate(val_fn, batch_size, num_to_test, x_test=None, y_test=None, dataflow=None)`
Evaluate the original ANN.

Can use either numpy arrays `x_test`, `y_test` containing the test samples, or generate them with a dataflow (Keras.ImageDataGenerator.flow_from_directory object).

**Parameters**

- **val_fn** – Function to evaluate model.

- **batch_size** (*int*) – Batch size

- **num_to_test** (*int*) – Number of samples to test

- **x_test** (Optional[np.ndarray]) –
• **y_test** *(Optional[np.ndarray])*

• **dataflow**(keras.ImageDataGenerator.flow_from_directory)*

**lasagne_input_lib**

**caffe_input_lib**

Caffe model parser.

@author: rbodo

class snntoolbox.parsing.model_libs.caffe_input_lib.ModelParser(input_model, config)

Bases: snntoolbox.parsing.utils.AbstractModelParser

parse_depthwiseconvolution (layer, attributes)

Parse a depthwise convolution layer.

Parameters

• **layer** – Layer.

• **attributes** *(dict)* – The layer attributes as key-value pairs in a dict.

get_layer_iterable()

Get an iterable over the layers of the network.

Returns layers

Return type list

get_type (layer)

Get layer class name.

Returns layer_type – Layer class name.

Return type str

get_batchnorm_parameters (layer)

Get the parameters of a batch-normalization layer.

Returns mean, var_eps_sqrt_inv, gamma, beta, axis

Return type tuple

get_inbound_layers (layer)

Return inbound layers.

Parameters layer *(Union[caffe.layers.Layer, caffe.layers.Concat])–A Caffe layer.*

Returns List of inbound layers.

Return type list[caffe.layers.Layer]

get_input_shape ()

Get the input shape of a network, not including batch size.

Returns input_shape – Input shape.

Return type tuple

get_output_shape (layer)

Get output shape of a layer.
Parameters **layer** – Layer.

Returns **output_shape** – Output shape of **layer**.

Return type **Sized**

**has_weights**(layer)
Return True if layer has weights.

**parse_dense**(layer, attributes)
Parse a fully-connected layer.

Parameters

- **layer** – Layer.
- **attributes**(dict) – The layer attributes as key-value pairs in a dict.

**parse_convolution**(layer, attributes)
Parse a convolutional layer.

Parameters

- **layer** – Layer.
- **attributes**(dict) – The layer attributes as key-value pairs in a dict.

**parse_pooling**(layer, attributes)
Parse a pooling layer.

Parameters

- **layer** – Layer.
- **attributes**(dict) – The layer attributes as key-value pairs in a dict.

**get_activation**(layer)
Get the activation string of an activation **layer**.

Parameters **layer** – Layer

Returns **activation** – String indicating the activation of the **layer**.

Return type **str**

**get_outbound_layers**(layer)
Get outbound layers of **layer**.

Parameters **layer** – Layer.

Returns **outbound** – Outbound layers of **layer**.

Return type **list**

**parse_concatenate**(layer, attributes)
Parse a concatenation layer.

Parameters

- **layer** – Layer.
- **attributes**(dict) – The layer attributes as key-value pairs in a dict.

**snntoolbox.parsing.model_libs.caffe_input_lib.load**(path=None, filename=None)
Load network from file.

Parameters

- **path**(str) – Path to directory where to load model from.
filename (str) – Name of file to load model from.

Returns

model – A dictionary of objects that constitute the input model. It must contain the following two keys:

* 'val_fn': Function that allows evaluating the original model.

Return type dict

snntoolbox.parsing.model_libs.caffe_input_lib.evaluate(val_fn, batch_size, num_to_test, x_test=None, y_test=None, dataflow=None)

Evaluate the original ANN.

Can use either numpy arrays x_test, y_test containing the test samples, or generate them with a dataflow (Keras.ImageDataGenerator.flow_from_directory object).

Parameters

* val_fn – Function to evaluate model.
* batch_size (int) – Batch size
* num_to_test (int) – Number of samples to test
* x_test (Optional[np.ndarray]) –
* y_test (Optional[np.ndarray]) –
* dataflow (keras.ImageDataGenerator.flow_from_directory) –

2.2.2 snntoolbox.parsing.utils

Functions common to input model parsers.

The core of this module is an abstract base class extracts an input model written in some neural network library and prepares it for further processing in the SNN toolbox.

AbstractModelParser

Abstract base class for neural network model parsers.

The idea is to make all further steps in the conversion/simulation pipeline independent of the original model format.

Other functions help navigate through the network in order to explore network connectivity and layer attributes:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_type</td>
<td>Get type of Keras layer.</td>
</tr>
<tr>
<td>has_weights</td>
<td>Return True if layer has weights.</td>
</tr>
<tr>
<td>get_fanin</td>
<td>Return fan-in of a neuron in layer.</td>
</tr>
<tr>
<td>get_fanout</td>
<td>Return fan-out of a neuron in layer.</td>
</tr>
<tr>
<td>get_inbound_layers</td>
<td>Return inbound layers.</td>
</tr>
<tr>
<td>get_inbound_layers_with_params</td>
<td>Iterate until inbound layers are found that have parameters.</td>
</tr>
<tr>
<td>get_inbound_layers_without_params</td>
<td>Return inbound layers.</td>
</tr>
<tr>
<td>get_outbound_layers</td>
<td>Return outbound layers.</td>
</tr>
<tr>
<td>get_outbound_activation</td>
<td>Iterate over 2 outbound layers to find an activation layer.</td>
</tr>
</tbody>
</table>
Abstract base class for neural network model parsers.

**Parameters**

- **input_model** – The input network object.
- **config** (configparser.Configparser) – Contains the toolbox configuration for a particular experiment.

**input_model**
The input network object.

**Type** `dict`

**config**
Contains the toolbox configuration for a particular experiment.

**Type** `configparser.Configparser`

**_layer_list**
A list where each entry is a dictionary containing layer specifications. Obtained by calling `parse`. Used to build new, parsed Keras model.

**Type** `list[dict]`

**_layer_dict**
Maps the layer names of the specific input model library to our standard names (currently Keras).

**Type** `dict`

**parsed_model**
The parsed model.

**Type** `keras.models.Model`

**parse()**
Extract the essential information about a neural network.

This method serves to abstract the conversion process of a network from the language the input model was built in (e.g., Keras or Lasagne).

The methods iterates over all layers of the input model and writes the layer specifications and parameters into `_layer_list`. The keys are chosen in accordance with Keras layer attributes to facilitate instantiation of a new, parsed Keras model (done in a later step by `build_parsed_model`).

This function applies several simplifications and adaptations to prepare the model for conversion to spiking. These modifications include:

- Removing layers only used during training (Dropout, BatchNormalization, ...)
- Absorbing the parameters of BatchNormalization layers into the parameters of the preceeding layer. This does not affect performance because batch-norm-parameters are constant at inference time.
- Removing ReLU activation layers, because their function is inherent to the spike generation mechanism. The information which nonlinearity was used in the original model is preserved in the `activation` key in `_layer_list`. If the output layer employs the softmax function, a spiking version is used when testing the SNN in INIsim or MegaSim simulators.
- Inserting a Flatten layer between Conv and FC layers, if the input model did not explicitly include one.
get_layer_iterable()
Get an iterable over the layers of the network.

Returns  layers
Return type  list

gtype (layer)
Get layer class name.

Returns  layer_type – Layer class name.
Return type  str

gbatchnorm_parameters (layer)
Get the parameters of a batch-normalization layer.

Returns  mean, var_eps_sqrt_inv, gamma, beta, axis
Return type  tuple

g inbound_layers_with_parameters (layer)
Iterate until inbound layers are found that have parameters.

Parameters  layer – Layer
Returns  List of inbound layers.
Return type  list

g inbound_names (layer, name_map)
Get names of inbound layers.

Parameters
• layer – Layer
• name_map (dict) – Maps the name of a layer to the id of the layer object.

Returns  The names of inbound layers.
Return type  list

g inbound_layers (layer)
Get inbound layers of layer.

Returns  inbound
Return type  Sequence

layers_to_skip
Return a list of layer names that should be skipped during conversion to a spiking network.

Returns  self._layers_to_skip
Return type  List[str]

has_weights (layer)
Return True if layer has weights.

initialize_attributes (layer=None)
Return a dictionary that will be used to collect all attributes of a layer. This dictionary can then be used to instantiate a new parsed layer.

g input_shape
Get the input shape of a network, not including batch size.

Returns  input_shape – Input shape.
Return type  tuple

get_batch_input_shape()
Get the input shape of a network, including batch size.

Returns  batch_input_shape – Batch input shape.

Return type  tuple

get_name (layer, idx, layer_type=None)
Create a name for a layer.
The format is <layer_num><layer_type>_<layer_shape>.

>>> # Name of first convolution layer with 32 feature maps and
>>> # dimension 64x64:
"00Conv2D_32x64x64"
>>> # Name of final dense layer with 100 units:
"06Dense_100"

Parameters
• layer – Layer.
• idx (int) – Layer index.
• layer_type (Optional[str]) – Type of layer.

Returns  name – Layer name.
Return type  str

get_output_shape (layer)
Get output shape of a layer.

Parameters  layer – Layer.

Returns  output_shape – Output shape of layer.
Return type  Sized

try_insert_flatten (layer, idx, name_map)

parse_dense (layer, attributes)
Parse a fully-connected layer.

Parameters
• layer – Layer.
• attributes (dict) – The layer attributes as key-value pairs in a dict.

parse_convolution (layer, attributes)
Parse a convolutional layer.

Parameters
• layer – Layer.
• attributes (dict) – The layer attributes as key-value pairs in a dict.

parse_depthwiseconvolution (layer, attributes)
Parse a depthwise convolution layer.

Parameters
parse_pooling (layer, attributes)
Parse a pooling layer.

Parameters

- **layer** – Layer.
- **attributes** (dict) – The layer attributes as key-value pairs in a dict.

absorb_activation (layer, attributes)
Detect what activation is used by the layer.

Sometimes the Dense or Conv layer specifies its activation directly, sometimes it is followed by a dedicated Activation layer (possibly with BatchNormalization in between). Here we try to find such an activation layer, and add this information to the Dense/Conv layer itself. The separate Activation layer can then be removed.

Parameters

- **layer** – Layer.
- **attributes** (dict) – The layer attributes as key-value pairs in a dict.

get_activation (layer)
Get the activation string of an activation layer.

Parameters

- **layer** – Layer.

Returns

- **activation** – String indicating the activation of the layer.

Return type str

get_outbound_layers (layer)
Get outbound layers of layer.

Parameters

- **layer** – Layer.

Returns

- **outbound** – Outbound layers of layer.

Return type list

parse_concatenate (layer, attributes)
Parse a concatenation layer.

Parameters

- **layer** – Layer.
- **attributes** (dict) – The layer attributes as key-value pairs in a dict.

build_parsed_model ()
Create a Keras model suitable for conversion to SNN.

This method uses the specifications in _layer_list to build a Keras model. The resulting model contains all essential information about the original network, independently of the model library in which the original network was built (e.g. Caffe).

Returns

- **parsed_model** – A Keras model, functionally equivalent to input_model.

Return type keras.models.Model
**evaluate** (*batch_size*, *num_to_test*, *x_test=None*, *y_test=None*, *dataflow=None*)

Evaluate parsed Keras model.

Can use either numpy arrays *x_test*, *y_test* containing the test samples, or generate them with a dataflow *(keras.ImageDataGenerator.flow_from_directory object)*.

**Parameters**

- **batch_size** (*int*) – Batch size
- **num_to_test** (*int*) – Number of samples to test
- **x_test** (*Optional[np.ndarray]*) –
- **y_test** (*Optional[np.ndarray]*) –
- **dataflow** (*keras.ImageDataGenerator.flow_from_directory*) –

**snntoolbox.parsing.utils.absorb_bn_parameters** (*weight*, *bias*, *mean*, *var_eps_sqrt_inv*, *gamma*, *beta*, *axis*, *image_data_format*, *is_depthwise=False*)

Absorb the parameters of a batch-normalization layer into the previous layer.

**snntoolbox.parsing.utils.modify_parameter_precision** (*weights*, *biases*, *config*, *attributes*)

**snntoolbox.parsing.utils.padding_string** (*pad*, *pool_size*)

Get string defining the border mode.

**Parameters**

- **pad** (*tuple[int]*) – Zero-padding in x- and y-direction.
- **pool_size** (*list[int]*) – Size of kernel.

**Returns** padding – Border mode identifier.

**Return type** *str*

**snntoolbox.parsing.utils.has_weights** (*layer*)

Return True if layer has weights.

**Parameters** layer (*keras.layers.Layer*) – Keras layer

**Returns** True if layer has weights.

**Return type** *bool*

**snntoolbox.parsing.utils.get_inbound_layers_with_params** (*layer*)

Iterate until inbound layers are found that have parameters.

**Parameters** layer (*keras.layers.Layer*) – Layer

**Returns** List of inbound layers.

**Return type** *list*

**snntoolbox.parsing.utils.get_inbound_layers_without_params** (*layer*)

Return inbound layers.

**Parameters** layer (*Keras.layers*) – A Keras layer.

**Returns** List of inbound layers.

**Return type** *list[Keras.layers]*

**snntoolbox.parsing.utils.get_inbound_layers** (*layer*)

Return inbound layers.
**Parameters** *layer* (*Keras.layers*) – A Keras layer.

**Returns** List of inbound layers.

**Return type** list[Keras.layers]

snntoolbox.parsing.utils.get_outbound_layers(*layer*)

Return outbound layers.

**Parameters** *layer* (*Keras.layers*) – A Keras layer.

**Returns** List of outbound layers.

**Return type** list[Keras.layers]

snntoolbox.parsing.utils.get_outbound_activation(*layer*)

Iterate over 2 outbound layers to find an activation layer. If there is no activation layer, take the activation of the current layer.

**Parameters** *layer* (*Union[keras.layers.Conv2D, keras.layers.Dense]*) – Layer

**Returns** *activation* – Name of outbound activation type.

**Return type** str

snntoolbox.parsing.utils.get_fanin(*layer*)

Return fan-in of a neuron in *layer*.

**Parameters** *layer* (*Subclass[keras.layers.Layer]*) – Layer.

**Returns** *fanin* – Fan-in.

**Return type** int

snntoolbox.parsing.utils.get_fanout(*layer, config*)

Return fan-out of a neuron in *layer*.

**Parameters**

• *layer* (*Subclass[keras.layers.Layer]*) – Layer.

• *config* (*configparser.ConfigParser*) – Settings.

**Returns** *fanout* – Fan-out. The fan-out of a neuron projecting onto a convolution layer varies between neurons in a feature map if the stride of the convolution layer is greater than unity. In this case, return an array of the same shape as the layer.

**Return type** Union[int, ndarray]

snntoolbox.parsing.utils.has_stride_unity(*layer*)

Return True if the strides in all dimensions of a *layer* are 1.

snntoolbox.parsing.utils.get_fanout_array(*layer_pre, layer_post, is_depthwise_conv=False*)

Return an array of the same shape as *layer_pre*, where each entry gives the number of outgoing connections of a neuron. In convolution layers where the post-synaptic layer has stride > 1, the fan-out varies between neurons.

snntoolbox.parsing.utils.get_type(*layer*)

Get type of Keras layer.

**Parameters** *layer* (*Keras.layers.Layer*) – Keras layer.

**Returns** Layer type.

**Return type** str
snntoolbox.parsing.utils.get_quantized_activation_function_from_string(activation_str)

Parse a string describing the activation of a layer, and return the corresponding activation function.

**Parameters**

- **activation_str** (*str*) – Describes activation.

**Returns**

- **activation** – Activation function.

**Return type**

functools.partial

**Examples**

```python
>>> f = get_quantized_activation_function_from_string('relu_Q1.15')
>>> f
functools.partial(<function reduce_precision at 0x7f919af92b70>, f='15', m='1')
>>> print(f.__name__)
relu_Q1.15
```

snntoolbox.parsing.utils.get_clamped_relu_from_string(activation_str)

If `activation_str` describes a custom activation function, import this function from `snntoolbox.utils.utils` and return it. If custom activation function is not found or implemented, return the `activation_str` in place of the activation function.

**Parameters**

- **activation_str** (*str*) – Describes activation.

**Returns**

- **activation** – Activation function.

- **activation_str** (*str*) – Describes activation.

snntoolbox.parsing.utils.get_custom_activation(activation_str)

Import all implemented custom activation functions so they can be used when loading a Keras model.

**Parameters**

- **filepath** (*Optional[str]*) – Path to json file containing additional custom objects.

snntoolbox.parsing.utils.precision(y_true, y_pred)

Precision metric.

Computes the precision, a metric for multi-label classification of how many selected items are relevant. Only computes a batch-wise average of precision.

### 2.3 snntoolbox.conversion

### 2.3.1 snntoolbox.conversion.utils

This module performs modifications on the network parameters during conversion from analog to spiking.

<table>
<thead>
<tr>
<th>normalize_parameters</th>
<th>Normalize the parameters of a network.</th>
</tr>
</thead>
</table>

@author: rbodo

snntoolbox.conversion.utils.normalize_parameters(model, config, **kwargs)

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.

\texttt{snntoolbox.conversion.utils.get_scale_fac(activations, percentile)}

Determine the activation value at \texttt{percentile} of the layer distribution.

\textbf{Parameters}

\begin{itemize}
  \item \texttt{activations (np.array)} – The activations of cells in a specific layer, flattened to 1-d.
  \item \texttt{percentile (int)} – Percentile at which to determine activation.
\end{itemize}

\textbf{Returns} \texttt{scale_fac} – Maximum (or percentile) of activations in this layer. Parameters of the respective layer are scaled by this value.

\textbf{Return type} float

\texttt{snntoolbox.conversion.utils.get_percentile(config, layer_idx=None)}

Get percentile at which to draw the maximum activation of a layer.

\textbf{Parameters}

\begin{itemize}
  \item \texttt{config (configparser.ConfigParser)} – Settings.
  \item \texttt{layer_idx (Optional[int])} – Layer index.
\end{itemize}

\textbf{Returns} Percentile.

\textbf{Return type} int

\texttt{snntoolbox.conversion.utils.apply_normalization_schedule(perc, layer_idx)}

Transform percentile according to some rule, depending on layer index.

\textbf{Parameters}

\begin{itemize}
  \item \texttt{perc (float)} – Original percentile.
  \item \texttt{layer_idx (int)} – Layer index, used to decrease the scale factor in higher layers, to maintain high spike rates.
\end{itemize}

\textbf{Returns} Modified percentile.

\textbf{Return type} int

\texttt{snntoolbox.conversion.utils.get_activations_layer(layer_in, layer_out, x, batch_size=None)}

Get activations of a specific layer, iterating batch-wise over the complete data set.

\textbf{Parameters}

\begin{itemize}
  \item \texttt{layer_in (keras.layers.Layer)} – The input to the network.
  \item \texttt{layer_out (keras.layers.Layer)} – The layer for which we want to get the activations.
  \item \texttt{x (np.array)} – The samples to compute activations for. With data of the form (channels, num_rows, num_cols), \texttt{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.
  \item \texttt{batch_size (Optional[int])} – Batch size
\end{itemize}

\textbf{Returns} \texttt{activations} – The activations of cells in a specific layer. Has the same shape as \texttt{layer_out}.
Return type  ndarray

snntoolbox.conversion.utils.get_activations_batch(ann, x_batch)

Compute layer activations of an ANN.

Parameters

- ann (keras.models.Model) – Needed to compute activations.
- x_batch (np.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 tuple (activations, label) represents a layer in the ANN for which an activation can be calculated (e.g. Dense, Conv2D). 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[tuple[np.array, str]]

snntoolbox.conversion.utils.try_reload_activations(layer, model, x_norm, batch_size, activ_dir)

2.4 snntoolbox.simulation

On the output side of the toolbox, the following simulators are currently implemented:

<table>
<thead>
<tr>
<th>Simulator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pyNN_target_sim</td>
<td>Building and simulating spiking neural networks using pyNN.</td>
</tr>
<tr>
<td>MegaSim_target_sim</td>
<td>MegaSim asynchronous spiking simulator.</td>
</tr>
<tr>
<td>INI_temporal_mean_rate_target_sim</td>
<td>INI simulator with temporal mean rate code.</td>
</tr>
<tr>
<td>INI_temporal_pattern_target_sim</td>
<td>INI simulator with temporal pattern code.</td>
</tr>
<tr>
<td>INI_ttfs_target_sim</td>
<td>INI simulator with time-to-first-spike code.</td>
</tr>
<tr>
<td>INI_ttfs_dyn_thresh_target_sim</td>
<td>INI simulator with time-to-first-spike code and a dynamic threshold.</td>
</tr>
<tr>
<td>INI_ttfs_corrective_target_sim</td>
<td>INI simulator with time-to-first-spike code and corrective spikes.</td>
</tr>
</tbody>
</table>

The abstract base class AbstractSNN for the simulation tools above is contained here:

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>snntoolbox.simulation.utils</td>
<td>Common functions for spiking simulators.</td>
</tr>
</tbody>
</table>

See Extending the toolbox on how to extend the toolbox by another simulator.

The backends for our built-in simulator INIsim and the custom simulator MegaSim are included here:

<table>
<thead>
<tr>
<th>Backend</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>temporal_mean_rate_tensorflow</td>
<td>INI temporal mean rate simulator with Tensorflow backend.</td>
</tr>
<tr>
<td>snntoolbox.simulation.backends.inisim.temporal_mean_rate_theano</td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
Table 9 – continued from previous page

<table>
<thead>
<tr>
<th>temporal_pattern</th>
<th>INI temporal pattern simulator backend.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ttfs</td>
<td>INI time-to-first-spike simulator backend.</td>
</tr>
<tr>
<td>ttfs_dyn_thresh</td>
<td>INI time-to-first-spike simulator backend with dynamic threshold.</td>
</tr>
<tr>
<td>ttfs_corrective</td>
<td>INI time-to-first-spike simulator backend with corrective spikes.</td>
</tr>
<tr>
<td>megasim</td>
<td>MegaSim spiking neuron simulator.</td>
</tr>
</tbody>
</table>

Finally, utility functions for plotting are contained in

```
snntoolbox.simulation.plotting  Various functions to visualize connectivity, activity and accuracy of the network.
```

### 2.4.1 snntoolbox.simulation.utils

Common functions for spiking simulators.

Most notably, this module defines the abstract base class `AbstractSNN` used to create spiking neural networks. This class has to be inherited from when another simulator is added to the toolbox (see *Extending the toolbox*).

```python
@author: rbodo

``` Python

```python
class snntoolbox.simulation.utils.AbstractSNN(config, queue=None)
   Bases: object

   Abstract base class for creating spiking neural networks.

   This class provides the basic structure to compile and simulate a spiking neural network. It has to be instantiated as in `target_simulators.pyNN_target_sim` with concrete methods tailored to the target simulator.

   Core methods that usually do not have to be overwritten include:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>build</code></td>
<td>Assemble a spiking neural network to prepare for simulation.</td>
</tr>
<tr>
<td><code>run</code></td>
<td>Simulate a spiking network.</td>
</tr>
<tr>
<td><code>get_recorded_vars</code></td>
<td>Retrieve neuron variables recorded during simulation.</td>
</tr>
</tbody>
</table>

   Relevant methods that in most cases will have to be overwritten include:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add_input_layer</code></td>
<td>Add input layer.</td>
</tr>
<tr>
<td><code>add_layer</code></td>
<td>Do anything that concerns adding any layer independently of its type.</td>
</tr>
<tr>
<td><code>build_dense</code></td>
<td>Build spiking fully-connected layer.</td>
</tr>
<tr>
<td><code>build_convolution</code></td>
<td>Build spiking convolutional layer.</td>
</tr>
<tr>
<td><code>build_pooling</code></td>
<td>Build spiking pooling layer.</td>
</tr>
<tr>
<td><code>build_flatten</code></td>
<td>Build flatten layer.</td>
</tr>
<tr>
<td><code>compile</code></td>
<td>Compile the spiking network.</td>
</tr>
<tr>
<td><code>simulate</code></td>
<td>Simulate a spiking network for a certain duration, and record any variables of interest (spike trains, membrane potentials, …)</td>
</tr>
</tbody>
</table>
```
Notes

In the attribute definitions below, we use suffixes to denote variable shape. In `spiketrains_n_b_l_t` for instance, \(n\) represents a dimension across the network (i.e. indexing the layers); \(b\) is an index for a batch of samples; \(l\) stands for the layer dimensions; \(t\) indicates the time axis.

`config`
Settings.

    Type  configparser.ConfigParser

`queue`
Used to detect stop signals from user to abort simulation.

    Type  Queue.Queue

`parsed_model`
The parsed model.

    Type  keras.models.Model

`is_built`
Whether or not the SNN has been built.

    Type  bool

`batch_size`
The batch size for parallel testing of multiple samples.

    Type  int

`spiketrains_n_b_l_t`
Spike trains of a batch of samples of all neurons in the network over the whole simulation time. Each entry in `spiketrains_batch` contains a tuple `(spiketimes, label)` for each layer of the network. `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. '03Conv2D_32x64x64'.

    Type  list[tuple[np.array, str]]

`activations_n_b_l`
Activations of the ANN.

    Type  list[tuple[np.array, str]]

`mem_n_b_l_t`
Membrane potentials of the SNN.

    Type  list[tuple[np.array, str]]

`input_b_l_t`
Input to the SNN over time.

    Type  ndarray

`top1err_b_t`
Top-1 error of SNN over time. Shape: `(batch_size, _num_timesteps)`.

    Type  ndarray

`top5err_b_t`
Top-5 error of SNN over time. Shape: `(batch_size, _num_timesteps)`.

    Type  ndarray
**synaptic_operations_b_t**
Number of synaptic operations of SNN over time. Shape: \((\text{batch}_\text{size}, \_\text{num}_\text{timesteps})\)

Type ndarray

**neuron_operations_b_t**
Number of updates of state variables of SNN over time, e.g. caused by leak / bias. Shape: \((\text{batch}_\text{size}, \_\text{num}_\text{timesteps})\)

Type ndarray

**operations_ann**
Number of operations of ANN.

Type float

**top1err_ann**
Top-1 error of ANN.

Type float

**top5err_ann**
Top-5 error of ANN.

Type float

**num_neurons**
Number of neurons in the network (one entry per layer).

Type List[int]

**num_neurons_with_bias**
Number of neurons with bias in the network (one entry per layer).

**fanin**
Number of synapses targeting a neuron (one entry per layer).

Type List[int]

**fanout**
Number of outgoing synapses. Usually one entry (integer) per layer. If the post- synaptic layer is a convolution layer with stride > 1, the fanout varies between neurons.

Type List[Union[int, ndarray]]

**rescale_fac**
Scales spike probability when using Poisson input.

Type float

**num_classes**
Number of classes of the data set.

Type int

**top_k**
By default, the toolbox records the top-1 and top-k classification errors.

Type int

**sim**
Module containing utility functions of spiking simulator. Result of calling `snntoolbox.bin.utils.initialize_simulator()`. For instance, if using Brian simulator, this initialization would be equivalent to `import pyNN.brian as sim`.

Type Simulator
flatten_shapes
List containing the (name, shape) tuples for each Flatten layer.

Type  list[[str, list]]

is_parallelizable
Whether or not the simulator is able to test multiple samples in parallel.

add_input_layer (input_shape)
Add input layer.

Parameters input_shape (list / tuple) – Input shape to the network, including the
batch size as first dimension.

add_layer (layer)
Do anything that concerns adding any layer independently of its type.

Parameters layer (keras.layers.Layer | keras.layers.Conv) – Layer

build_dense (layer)
Build spiking fully-connected layer.

Parameters layer (keras.layers.Dense) – Layer

build_convolution (layer)
Build spiking convolutional layer.

Parameters layer (keras.layers.Layer) – Layer

build_pooling (layer)
Build spiking pooling layer.

Parameters layer (keras.layers.pooling._Pooling2D) – Layer

build_flatten (layer)
Build flatten layer.

May not be needed depending on the simulator.

Parameters layer (keras.layers.Layer) – Layer

compile ()
Compile the spiking network.

simulate (**kwargs)
Simulate a spiking network for a certain duration, and record any variables of interest (spike trains, mem-
brane potentials, ...)

Returns output_b_l_t – Array of shape (batch_size, num_classes,
num_timesteps), containing the number of output spikes of the neurons in the
final layer, for each sample and for each time step during the simulation.

Return type  ndarray

reset (sample_idx)
Reset network variables.

Parameters sample_idx (int) – Index of sample that has just been simulated. In certain
applications (video data), we may want to turn off reset between samples.

end_sim ()
Clean up after simulation.

save (path, filename)
Write model architecture and parameters to disk.
Parameters

- **path**(string) – Path to directory where to save model.
- **filename**(string) – Name of file to write model to.

load (path, filename)
Load model architecture and parameters from disk.

Parameters

- **path**(str) – Path to directory where to load model from.
- **filename**(str) – Name of file to load model from.

init_cells()
Set cell parameters of neurons in each layer and initialize membrane potential.

get_spiketrains(**kwargs)
Get spike trains of a layer.

Returns spiketrains_b_l_t – Spike trains.

Return type ndarray

get_spiketrains_input()
Get spike trains of input layer.

Returns spiketrains_b_l_t – Spike trains of input.

Return type ndarray

get_spiketrains_output()
Get spike trains of output layer.

Returns spiketrains_b_l_t – Spike trains of output.

Return type ndarray

get_vmem(**kwargs)
Get membrane potentials of a layer.

Returns mem_b_l_t – Membrane potentials of layer.

Return type ndarray

build (parsed_model, **kwargs)
Assemble a spiking neural network to prepare for simulation.

Parameters parsed_model (keras.models.Model) – Parsed input model.

run (x_test=None, y_test=None, dataflow=None, **kwargs)
Simulate a spiking network.

This method takes care of preparing the dataset for batch-wise processing, and allocates variables for quantities measured during the simulation. The simulate method (overwritten by a concrete target simulator) is responsible for actually simulating the network over a given duration, and reporting the measured quantities. The run method then deals with evaluating this data to print statistics, plot figures, etc.

Parameters

- **x_test**(float32 array) – The input samples to test. With data of the form (channels, num_rows, num_cols), x_test has dimension (num_samples, channels*num_rows*num_cols) for a multi-layer perceptron, and (num_samples, channels, num_rows, num_cols) for a convolutional net.
• **y_test** (*float32 array*) – Ground truth of test data. Has dimension *(num_samples, num_classes)*

• **dataflow** (*keras.DataFlowGenerator*) – Loads images from disk and processes them on the fly.

• **kwargs** (*Optional[dict]*) – Optional keyword arguments, for instance
  – **path** (*Optional[str]*) Where to store the output plots. If no path given, this value is taken from the settings dictionary.

**Returns top1acc_total** – Number of correctly classified samples divided by total number of test samples.

**Return type** float

**adjust_batchsize**()
Reduce batch size to single sample if necessary.

Not every simulator is able to simulate multiple samples in parallel. If this is the case (indicated by *is_parallelizable*), set **batch_size** to 1.

**restore_snn**()
Restore both the spiking and the parsed network from disk.

This method works for spiking Keras models.

**init_log_vars**()
Initialize variables to record during simulation.

**reset_log_vars**()
Reset variables to record during simulation.

**set_connectivity**()
Set connectivity statistics needed to compute the number of operations in the network. This includes e.g. the members *fanin, fanout, num_neurons, num_neurons_with_bias*.

**get_recorded_vars**(*layers*)
Retrieve neuron variables recorded during simulation.

If recorded, spike trains and membrane potentials will be inserted into the respective class members *spiketrains_n_b_l_t* and *mem_n_b_l_t*. In any case, this function must return an array containing the output spikes for each sample and time step.

**Parameters** *layers* – List of SNN layers.

**Returns output_b_l_t** – The output spikes. Shape: *(batch_size, layer_shape, num_timesteps)*

**Return type** ndarray

**reset_container_counters**()

**set_mem_stats**(*mem, v_thresh*)
Write recorded membrane potential out and plot it.

**set_spiketrain_stats_input**()
Count number of operations based on the input spike activity during simulation.

**set_spiketrain_stats**(*spiketrains_b_l_t*)
Count number of operations based on the spike activity during simulation.

**Parameters** *spiketrains_b_l_t* (*ndarray*) – A batch of spikes for a layer over the simulation time. Shape: *(batch_size, layer_shape, num_timesteps)*
**reshape_flattened_spiketrains** *(spiketrains, shape, is_list=True)*

Convert list of spike times into array where nonzero entries (indicating spike times) are properly spread out across array. Then reshape the flat array into original layer shape.

**Parameters**

- **spiketrains** *(ndarray)* – Spike times.
- **shape** – Layer shape.
- **is_list** *(Optional[bool])* – If True (default), spiketrains is a list of spike times. In this case, we distribute the spike times across a numpy array. If False, spiketrains is already a 2D array of shape (num_neurons, num_timesteps).

**Returns**

spiketrains_b_l_t – A batch of spikes for a layer over the simulation time. Shape: *(batch_size, shape, num_timesteps)*

**Return type** *ndarray*

**get_avg_rate_from_trains()**

Compute spike rate of neurons averaged over batches, the neurons in the network, and the simulation time.

**preprocessing(****kwargs***)

**Parameters**

**kwargs** –

- **snntoolbox.simulation.utils.get_samples_from_list** *(x_test, y_test, dataflow, config)*
  
  If user specified a list of samples to test with config.get('simulation', 'sample_idxs_to_test'), this function extracts them from the test set.

- **snntoolbox.simulation.utils.build_convolution** *(layer, delay, transpose_kernel=False)*
  
  Build convolution layer.

  **Parameters**

  - **layer** *(keras.layers.Conv2D)* – Parsed model layer.
  - **delay** *(float)* – Synaptic delay.
  - **transpose_kernel** *(bool)* – Whether or not to convert kernels from Tensorflow to Theano format (correlation instead of convolution).

  **Returns**

  - **connections** *(List[tuple])* – A list where each entry is a tuple containing the source neuron index, the target neuron index, the connection strength (weight), and the synaptic delay.
  - **i_offset** *(ndarray)* – Flattened array containing the biases of all neurons in the layer.

- **snntoolbox.simulation.utils.build_pooling** *(layer, delay)*
  
  Build average pooling layer.

  **Parameters**

  - **layer** *(keras.layers.Pool2D)* – Parsed model layer.
  - **delay** *(float)* – Synaptic delay.

  **Returns**

  **connections** – A list where each entry is a tuple containing the source neuron index, the target neuron index, the connection strength (weight), and the synaptic delay. The weight is given by \( \frac{1}{k_x k_y} \), where \( k_x, k_y \) are the dimensions of the pooling kernel.

  **Return type** *List[tuple]*
snntoolbox.simulation.utils.spikecounts_to_rates(spikecounts_n_b_l_t)

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.

**Parameters**

- **spikecounts_n_b_l_t** (list[tuple[np.array, str]]) -

**Returns**

- spikerates_n_b_l

**Return type**

- list[tuple[np.array, str]]

snntoolbox.simulation.utils.spikecounts_to_rates(spikecounts_n_b_l_t, duration, spike_code)

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.

**Parameters**

- **spikecounts_n_b_l_t** (list[tuple[np.array, str]]) -
- **duration** (int) – Duration of simulation.
- **spike_code** (str) – String specifying the spike encoding mechanism. For instance, with ‘ttfs’, the spike rates are computed using the time to first spike.

**Returns**

- spikerates_n_b_l

**Return type**

- list[tuple[np.array, str]]

snntoolbox.simulation.utils.get_sample_activity_from_batch(activity_batch, idx=0)

Return layer activity for sample idx of an activity_batch.

snntoolbox.simulation.utils.get_spiking_outbound_layers(layer, config)

Iterate until spiking outbound layers are found.

**Parameters**

- **layer** (keras.layers.Layer) – Layer
- **config** (configparser.ConfigParser) – Settings.

**Returns**

- List of outbound layers.

**Return type**

- list

snntoolbox.simulation.utils.get_layer_synaptic_operations(spikecounts_b_l, fanout)

Return total number of synaptic operations in the layer for a batch of samples.

**Parameters**

- **spikecounts_b_l** (ndarray) – Batch of spiketrains of a layer. Shape: (batch_size, layer_shape)
- **fanout** (Union[int, ndarray]) – Number of outgoing connections per neuron. Can be a single integer, or an array of the same shape as the layer, if the fanout varies from neuron to neuron (as is the case in convolution layers with stride > 1).

**Returns**

- layer_ops – The total number of operations in the layer for a batch of samples.

**Return type**

- int
**snntoolbox.simulation.utils.get_ann_ops**\(\text{(num\_neurons, num\_neurons\_with\_bias, fanin)}\)

Compute number of operations performed by an ANN in one forward pass.

**Parameters**

- **num\_neurons** \((\text{list[int]})\) – Number of neurons per layer, starting with input layer.
- **num\_neurons\_with\_bias** \((\text{list[int]})\) – Number of neurons with bias.
- **fanin** \((\text{list[int]})\) – List of fan-in of neurons in Conv, Dense and Pool layers. Input and Pool layers have fan-in 0 so they are not counted.

**Returns** Number of operations.

**Return type** int

**snntoolbox.simulation.utils.estimate_snn_ops**\(\text{(activations\_n\_b\_l, fanouts\_n, num\_timesteps)}\)

**snntoolbox.simulation.utils.is_spiking**\(\text{(layer, config)}\)

Test if layer is going to be converted to a layer that spikes.

**Parameters**

- **layer** \((\text{Keras.layers.Layer})\) – Layer of parsed model.
- **config** \((\text{configparser.ConfigParser})\) – Settings.

**Returns** True if converted layer will have spiking neurons.

**Return type** bool

**snntoolbox.simulation.utils.get_shape_from_label**\(\text{(label)}\)

Extract the output shape of a flattened pyNN layer from the layer name generated during parsing.

**Parameters** **label** \((\text{str})\) – Layer name containing shape information after a `_' separator.

**Returns** The layer shape.

**Return type** list

**Example**

```python
>>> get_shape_from_label('02Conv2D_16x32x32')
[16, 32, 32]
```

### 2.4.2 snntoolbox.simulation.plotting

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

@author: rbodo

**snntoolbox.simulation.plotting.output_graphs**\(\text{(plot\_vars, config, path=\text{None}, idx=0, data\_format=\text{None})}\)

Wrapper function to display / save a number of plots.

**Parameters**

- **plot\_vars** \((\text{dict})\) – Example items:
  - **spiketrains\_n\_b\_l\_t** \((\text{list[tuple[np.array, str]]})\) Each entry in spiketrains\_batch contains a tuple \((\text{spiketimes, label})\) 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_chnl, n_rows, n_cols, duration) label is a string specifying both the layer type and the index, e.g. '03Dense'.

- activations_n_b_l: list[tuple[np.array, str]] Activations of the ANN.
- spiketrains_n_b_l: list[tuple[np.array, str]] Spikecounts of the SNN. Used to compute spikerates.

• config (configparser.ConfigParser) – Settings.
• path (Optional[str]) – If not None, specifies where to save the resulting image. Else, display plots without saving.
• idx (int) – The index of the sample to display. Defaults to 0.
• data_format (Optional[str]) – One of ‘channels_first’ or ‘channels_last’.

snntoolbox.simulation.plotting.plot_layer_summaries(plot_vars, config, path=None, data_format=None)

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

Parameters

• plot_vars (dict) – Example items:
  - spikerates: list[tuple[np.array, str]] 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[tuple[np.array, str]] Contains the activations of a net. Same structure as spikerates.
  - spiketrains: list[tuple[np.array, str]] 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_chnl, n_rows, n_cols, duration)
    label is a string specifying both the layer type and the index, e.g. '03Dense'.

• config (configparser.ConfigParser) – Settings.
• path (Optional[str]) – If not None, specifies where to save the resulting image. Else, display plots without saving.
• data_format (Optional[str]) – One of ‘channels_first’ or ‘channels_last’.

snntoolbox.simulation.plotting.plot_layer_activity(layer, title, path=None, limits=None, data_format=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[np.array, str]) - (activity, label)*.

*activity* is an array of the same shape as the original layer, containing e.g. the spik- 
erates or activations of neurons in a layer.

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

**title** *(str)* – Figure title.

**path** *(Optional[str])* – If not None, specifies where to save the resulting image. 
Else, display plots without saving.

**limits** *(Optional[tuple]*) – If not None, the colormap of the resulting image is 
limited by this tuple.

**data_format** *(Optional[str])* – One of ‘channels_first’ or ‘channels_last’.

snntoolbox.simulation.plotting.plot_activations(*model, x_test, path, 
*data_format=None)*

Plot activations of a network.

**Parameters**

- **model** *(keras.models.Model)* – Keras model.
- **x_test** *(ndarray)* – The samples.
- **path** *(str)* – Where to save plot.
- **data_format** *(Optional[str])* – One of ‘channels_first’ or ‘channels_last’.

snntoolbox.simulation.plotting.plot_activations_minus_rates(*activations, rates, 
*label, path=None, 
*data_format=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**

- **activations** *(ndarray)* – 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.
- **rates** *(ndarray)* – 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.
- **label** *(str)* – Layer label.
- **path** *(Optional[str])* – If not None, specifies where to save the resulting image. 
Else, display plots without saving.
- **data_format** *(Optional[str])* – One of ‘channels_first’ or ‘channels_last’.

snntoolbox.simulation.plotting.plot_layer_correlation(*rates, activations, ti- 
tle, config, path=None, 
same_xlim=True)*

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

**Parameters**

- **rates** *(np.array)* – The spikerates of a layer, flattened to 1D.
- **activations** *(Union[ndarray, Iterable])* – The activations of a layer, flat- 
tened to 1D.
• **title** (*str*) – Plot title.
• **config** (*configparser.ConfigParser*) – Settings.
• **path** (*Optional*[str]*) – If not None, specifies where to save the resulting image. Else, display plots without saving.
• **same_xlim** (*Optional*[bool]*) – Whether to use the same axis limit on the rates and activations. If True, the maximum is chosen. Default: True.

```python
snntoolbox.simulation.plotting.plot_correlations(a, b, path=None, filename=None)
```

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, Conv2D).

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'.

```python
snntoolbox.simulation.plotting.get_pearson_coefficients(spikerates_batch, activations_batch, max_rate)
```

Compute Pearson coefficients.

**Parameters**

- **spikerates_batch**
- **activations_batch**
- **max_rate** (*float*) – Highest spike rate.

**Returns**

- **co** (*list*)

```python
snntoolbox.simulation.plotting.plot_pearson_coefficients(spikerates_batch, activations_batch, config, path=None)
```

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

**Parameters**

- **spikerates_batch** (list of tuple [np.array, str]) – 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[tuple[np.array, str]])* – Contains the activations of a net. Same structure as spikerates_batch.
- **config** *(configparser.ConfigParser)* – Settings.
- **path** *(Optional[str])* – Where to save the output.

`snntoolbox.simulation.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 inserted into plot title.

`snntoolbox.simulation.plotting.plot_activ_hist(h, title=None, layer_label=None, path=None, scale_fac=None)`

Plot a histogram over all activities of a network.

**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 inserted into plot title.

`snntoolbox.simulation.plotting.plot_max_activ_hist(h, title=None, layer_label=None, path=None, scale_fac=None)`

Plot a histogram over the maximum activations.

**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 inserted into plot title.
snntoolbox.simulation.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.simulation.plotting.plot_param_sweep(results, n, params, param_name, param_logscale)

Plot accuracy versus parameter.

Parameters

- **results (list[float])** – 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[float])** – The parameter values that changed during each experiment.
- **param_name (str)** – The name of the parameter that varied.
- **param_logscale (bool)** – Whether to plot the parameter axis in log-scale.

snntoolbox.simulation.plotting.plot_spiketrains(layer, dt, path=None, data_format=None)

Plot which neuron fired at what time during the simulation.

Parameters

- **layer (tuple[np.array, str])** – (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'.
- **dt (float)** – Time resolution of simulation.
- **path (Optional[str])** – If not None, specifies where to save the resulting image. Else, display plots without saving.
- **data_format (Optional[str])** – One of ‘channels_first’ or ‘channels_last’.

snntoolbox.simulation.plotting.plot_potential(times, layer, config, v_thresh=None, show_legend=False, path=None)

Plot the membrane potential of a layer.

Parameters

- **times (np.array)** – The time values where the potential was sampled.
- **layer (tuple[np.array, str])** – (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'.
- **config (configparser.ConfigParser)** – Settings.
- **v_thresh (float)** – Threshold.
• **show_legend (bool)** – If True, shows the legend indicating the neuron indices and lines like \(v_{\text{thresh}}, v_{\text{rest}}, v_{\text{reset}}\). Recommended only for layers with few neurons.

• **path (Optional [str])** – If not None, specifies where to save the resulting image. Else, display plots without saving.

```python
snntoolbox.simulation.plotting.plot_confusion_matrix(y_test, y_pred, path=None, class_labels=None)
```

**Parameters**

- **y_test (list)** –
- **y_pred (list)** –
- **path (Optional [str])** – Where to save the output.
- **class_labels (Optional [list])** – List of class labels.

```python
snntoolbox.simulation.plotting.plot_error_vs_time(top1err_d_t, top5err_d_t, duration, dt, top1err_ann=None, top5err_ann=None, path=None)
```

Plot classification error over time.

**Parameters**

- **top1err_d_t (np.array)** – Batch of top-1 errors over time. Shape: (num_samples, duration). Data type: boolean (correct / incorrect classification).
- **top5err_d_t (np.array)** – Batch of top-5 errors over time. Shape: (num_samples, duration). Data type: boolean (correct / incorrect classification).
- **duration (int)** – Simulation duration.
- **dt (float)** – Simulation time resolution.
- **top1err_ann (Optional [float])** – The top-1 error of the ANN.
- **top5err_ann (Optional [float])** – The top-5 error of the ANN.
- **path (Optional [str])** – Where to save the output.

```python
snntoolbox.simulation.plotting.plot_ops_vs_time(operations_b_t, duration, dt, path=None)
```

Plot total number of operations over time.

**Parameters**

- **operations_b_t (ndarray)** – Number of operations. Shape: (batch_size, num_timesteps)
- **duration (int)** – Simulation duration.
- **dt (float)** – Simulation time resolution.
- **path (Optional [str])** – Where to save the output.

```python
snntoolbox.simulation.plotting.plot_spikecount_vs_time(spiketrains_n_b_l_t, duration, dt, path=None)
```

Plot total spikenumber over time.

**Parameters**

- **spiketrains_n_b_l_t** –
- **duration (int)** – Simulation duration.
• **dt** (*float*) – Simulation time resolution.
• **path** (*Optional[str]*) – Where to save the output.

```python
snntoolbox.simulation.plotting.plot_input_image(x, label, path=None, data_format=None, filename=None)
```

Show an input image.

**Parameters**

- **x** (*ndarray*) – The sample to plot.
- **label** (*int*) – Class label (index) of sample.
- **path** (*Optional[str]*) – Where to save the image.
- **data_format** (*Optional[str]*) – One of ‘channels_first’ or ‘channels_last’.
- **filename** (*Optional[str]*) – Name of file to save.

```python
snntoolbox.simulation.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.

```python
snntoolbox.simulation.plotting.plot_probe(probe, path, filename)
```

### 2.4.3 `snntoolbox.simulation.backends`

#### inisim

#### temporal_mean_rate_tensorflow

INI temporal mean rate simulator with Tensorflow backend.

This module defines the layer objects used to create a spiking neural network for our built-in INI simulator `INI_temporal_mean_rate_target_sim`.

The coding scheme underlying this conversion is that the analog activation value is represented by the average over number of spikes that occur during the simulation duration.

@author: rbodo

```python
class snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow.SpikeLayer(**kwargs):
    Bases: keras.engine.base_layer.Layer
    Base class for layer with spiking neurons.

    reset(sample_idx)
    Reset layer variables.

    class_name
    Get class name.

    update_neurons()  
    Update neurons according to activation function.

    update_payload(residuals, spikes)
    Update payloads.
    Uses the residual of the membrane potential after spike.
```
linear_activation(mem)
   Linear activation.

binary_sigmoid_activation(mem)
   Binary sigmoid activation.

binary_tanh_activation(mem)
   Binary tanh activation.

softmax_activation(mem)
   Softmax activation.

quantized_activation(mem, m, f)
   Activation with precision reduced to fixed point format Qm.f.

get_new_mem()
   Add input to membrane potential.

set_reset_mem(mem, spikes)
   Reset membrane potential mem array where spikes array is nonzero.

get_new_thresh()
   Get new threshold.

get_time()
   Get simulation time variable.

   Returns time – Current simulation time.

   Return type float

set_time(time)
   Set simulation time variable.

   Parameters time (float) – Current simulation time.

init_membrane_potential(output_shape=None, mode='zero')
   Initialize membrane potential.

   Helpful to avoid transient response in the beginning of the simulation. Not needed when reset between
   frames is turned off, e.g. with a video data set.

   Parameters

   • output_shape (Optional[tuple]) – Output shape
   • mode (str) – Initialization mode.
      - 'uniform': Random numbers from uniform distribution in [-thr, thr].
      - 'bias': Negative bias.
      - 'zero': Zero (default).

   Returns init_mem – A tensor of self.output_shape (same as layer).

   Return type ndarray

reset_spikevars(sample_idx)
   Reset variables present in spiking layers. Can be turned off for instance when a video sequence is tested.

init_neurons(input_shape)
   Init layer neurons.

get_layer_idx()
   Get index of layer.
get_clamp_idx()
Get time step when to stop clamping membrane potential.

Returns Time step when to stop clamping.
Return type int

update_avg_variance(spikes)
Keep a running average of the spike-rates and their variance.
Parameters spikes – Output spikes.

update_b()
Get a new value for the bias, relaxing it over time to the true value.

snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow.add_payloads(prev_layer, input_spikes)
Get payloads from previous layer.

snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow.spike_call(call)

snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow.get_isi_from_impulse(impulse, epsilon)

class snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow.SpikeConcatenate()
Bases: keras.layers.merge.Concatenate
Spike merge layer
static get_time()
static reset(sample_idx)
Reset layer variables.

class_name
Get class name.

class snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow.SpikeFlatten(**kwargs)
Bases: keras.layers.core.Flatten
Spike flatten layer.
call(x, mask=None)
This is where the layer’s logic lives.

# Arguments inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

# Returns A tensor or list/tuple of tensors.
static get_time()
static reset(sample_idx)
Reset layer variables.

class_name
Get class name.
class snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow.SpikeDense(units, activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None, **kwargs)

Bases: keras.layers.core.Dense, snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow.SpikeLayer

Spike Dense layer.

**build**(input_shape)

Creates the layer neurons and connections.

**Parameters**

**input_shape** *(Union[list, tuple, Any]) – Keras tensor (future input to layer) or list/tuple of Keras tensors to reference for weight shape computations.*

**call**(x)
class snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow.SpikeConv2D(filters, kernel_size, strides=(1, 1), padding='valid', data_format=None, dilation_rate=(1, 1), activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None, **kwargs)

Bases: keras.layers.convolutional.Conv2D, snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow.SpikeLayer

Spike 2D Convolution.

build(input_shape)

Creates the layer weights. Must be implemented on all layers that have weights.

Parameters input_shape (Union[list, tuple, Any]) – Keras tensor (future input to layer) or list/tuple of Keras tensors to reference for weight shape computations.

call(x)

class snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow.SpikeAveragePooling2D(pool_size=(2, 2), strides=None, padding='valid', data_format=None, **kwargs)

Bases: keras.layers.pooling.AveragePooling2D, snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow.SpikeLayer

Spike Average Pooling.

build(input_shape)

Creates the layer weights. Must be implemented on all layers that have weights.

Parameters input_shape (Union[list, tuple, Any]) – Keras tensor (future input to layer) or list/tuple of Keras tensors to reference for weight shape computations.
call \( x \)

class snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow.SpikeMaxPooling2D
Bases: keras.layers.pooling.MaxPooling2D, snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow.SpikeLayer

Spike Max Pooling.

build \((\text{input\_shape})\)

Creates the layer neurons and connections.

Parameters

\textbf{input\_shape} (Union[\text{list, tuple, Any}]) – Keras tensor (future input to layer) or list/tuple of Keras tensors to reference for weight shape computations.

call \( x \)

temporal\_mean\_rate\_theano

temporal\_pattern

INI temporal pattern simulator backend.

This module defines the layer objects used to create a spiking neural network for our built-in INI simulator :py:mod:`snntoolbox.simulation.target_simulators.INI_temporal_pattern_target_sim`.

The coding scheme underlying this conversion is that the analog activation value is transformed into a binary representation of spikes.

This simulator works only with Keras backend set to Tensorflow.

@author: rbodo

class snntoolbox.simulation.backends.inisim.temporal_pattern.SpikeLayer(**\text{kwargs})
Bases: keras.engine.base_layer.Layer

Base class for layer with spiking neurons.

\textbf{class\_name}

Get class name.

\textbf{get\_time}()

\textbf{static\_reset}(\text{sample\_idx})

Reset layer variables.

\textbf{init\_neurons}(\text{input\_shape})

Init layer neurons.

\textbf{update\_spikevars}(\text{x})

snntoolbox.simulation.backends.inisim.temporal_pattern.spike\_call(call)

snntoolbox.simulation.backends.inisim.temporal_pattern.to\_binary(x, num\_bits)

Transform an array of floats into binary representation.

Parameters

- \textbf{x} (ndarray) – Input array containing float values. The first dimension has to be of length 1.
- \textbf{num\_bits} (int) – The fixed point precision to be used when converting to binary.
**Returns binary_array** – Output boolean array. The first dimension of x is expanded to length bits. The binary representation of each value in x is distributed across the first dimension of binary_array.

**Return type**  ndarray

snntoolbox.simulation.backends.inisim.temporal_pattern.to_binary_numpy(x, num_bits)

Transform an array of floats into binary representation.

**Parameters**

- **x** (*ndarray*) – Input array containing float values. The first dimension has to be of length 1.
- **num_bits** (*int*) – The fixed point precision to be used when converting to binary.

**Returns binary_array** – Output boolean array. The first dimension of x is expanded to length bits. The binary representation of each value in x is distributed across the first dimension of binary_array.

**Return type**  ndarray

```python
class snntoolbox.simulation.backends.inisim.temporal_pattern.SpikeConcatenate(axis, **kwargs)

Bases: keras.layers.merge.Concatenate

Spike merge layer

static get_time()
static reset(sample_idx)
    Reset layer variables.

class_name
    Get class name.

class snntoolbox.simulation.backends.inisim.temporal_pattern.SpikeFlatten(**kwargs)

Bases: keras.layers.core.Flatten

Spike flatten layer.

static get_time()
static reset(sample_idx)
    Reset layer variables.

class_name
    Get class name.
```
class snntoolbox.simulation.backends.inisim.temporal_pattern.SpikeDense(units, activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None, **kwargs)

Bases: keras.layers.core.Dense, snntoolbox.simulation.backends.inisim.temporal_pattern.SpikeLayer

Spike Dense layer.

build(input_shape)

Creates the layer neurons and connections.

Parameters input_shape (Union[list, tuple, Any]) – Keras tensor (future input to layer) or list/tuple of Keras tensors to reference for weight shape computations.

call(x)
class snntoolbox.simulation.backends.inisim.temporal_pattern.SpikeConv2D(filters, kernel_size, strides=(1, 1), padding='valid', data_format=None, dilation_rate=(1, 1), activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None, **kwargs)

Bases: keras.layers.convolutional.Conv2D, snntoolbox.simulation.backends.inisim.temporal_pattern.SpikeLayer

Spike 2D Convolution.

build(input_shape)

Creates the layer weights. Must be implemented on all layers that have weights.

Parameters

**input_shape** (Union[list, tuple, Any]) – Keras tensor (future input to layer) or list/tuple of Keras tensors to reference for weight shape computations.

call(x)

class snntoolbox.simulation.backends.inisim.temporal_pattern.SpikeAveragePooling2D(**kwargs)

Bases: keras.layers.pooling.AveragePooling2D, snntoolbox.simulation.backends.inisim.temporal_pattern.SpikeLayer

Spike Average Pooling layer.

build(input_shape)

Creates the layer weights. Must be implemented on all layers that have weights.

Parameters

**input_shape** (Union[list, tuple, Any]) – Keras tensor (future input to layer) or list/tuple of Keras tensors to reference for weight shape computations.

call(x, mask=None)

This is where the layer’s logic lives.

# Arguments

inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.
# Returns
A tensor or list/tuple of tensors.

class snntoolbox.simulation.backends.inisim.temporal_pattern.SpikeMaxPooling2D(**kwargs)
Bases: keras.layers.pooling.MaxPooling2D, snntoolbox.simulation.backends.inisim.temporal_pattern.SpikeLayer

Spike Max Pooling.

build(input_shape)

Creates the layer neurons and connections.

Parameters input_shape(Union[list, tuple, Any]) – Keras tensor (future input to layer) or list/tuple of Keras tensors to reference for weight shape computations.

call(x, mask=None)

This is where the layer’s logic lives.

# Arguments
inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

# Returns
A tensor or list/tuple of tensors.

ttfs

INI time-to-first-spike simulator backend.

This module defines the layer objects used to create a spiking neural network for our built-in INI simulator INI_ttfs_target_sim.

The coding scheme underlying this conversion is that the instantaneous firing rate is given by the inverse time-to-first-spike.

This simulator works only with Keras backend set to Tensorflow.

@author: rbodo

class snntoolbox.simulation.backends.inisim.ttfs.SpikeLayer(**kwargs)
Bases: keras.engine.base_layer.Layer

Base class for layer with spiking neurons.

reset(sample_idx)

Reset layer variables.

class_name

Get class name.

update_neurons()

Update neurons according to activation function.

linear_activation(mem)

Linear activation.

static softmax_activation(mem)

Softmax activation.

get_new_mem()

Add input to membrane potential.

set_reset_mem(mem, spikes)

Reset membrane potential mem array where spikes array is nonzero.

get_psp(output_spikes)
get_time()
Get simulation time variable.

Returns time – Current simulation time.

Return type float

set_time(time)
Set simulation time variable.

Parameters time (float) – Current simulation time.

init_membrane_potential(output_shape=None, mode='zero')
Initialize membrane potential.

Helpful to avoid transient response in the beginning of the simulation. Not needed when reset between frames is turned off, e.g. with a video data set.

Parameters

• output_shape (Optional[tuple]) – Output shape
• mode (str) – Initialization mode.
  – 'uniform': Random numbers from uniform distribution in [-thr, thr].
  – 'bias': Negative bias.
  – 'zero': Zero (default).

Returns init_mem – A tensor of self.output_shape (same as layer).

Return type ndarray

reset_spikevars(sample_idx)
Reset variables present in spiking layers. Can be turned off for instance when a video sequence is tested.

init_neurons(input_shape)
Init layer neurons.

get_layer_idx()
Get index of layer.

snntoolbox.simulation.backends.inisim.ttfs.snn unfold (call)
class snntoolbox.simulation.backends.inisim.ttfs.SpikeConcatenate(axis, **kwargs)

Bases: keras.layers.merge.Concatenate

Spike merge layer

static get_time()

static reset(sample_idx)
Reset layer variables.

class_name
Get class name.

class snntoolbox.simulation.backends.inisim.ttfs.SpikeFlatten(**kwargs)

Bases: keras.layers.core.Flatten

Spike flatten layer.

static get_time()

reset(sample_idx)
Reset layer variables.
class_name
Get class name.

class snntoolbox.simulation.backends.inisim.ttfs.SpikeDense(units, activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None, **kwargs)

Bases: keras.layers.core.Dense, snntoolbox.simulation.backends.inisim.ttfs.SpikeLayer

Spike Dense layer.

build(input_shape)
Creates the layer neurons and connections.

Parameters

input_shape (Union[list, tuple, Any]) – Keras tensor (future input to layer) or list/tuple of Keras tensors to reference for weight shape computations.

call(x)

class snntoolbox.simulation.backends.inisim.ttfs.SpikeConv2D(filters, kernel_size, strides=(1, 1), padding='valid', data_format=None, dilation_rate=(1, 1), activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None, **kwargs)

Bases: keras.layers.convolutional.Conv2D, snntoolbox.simulation.backends.inisim.ttfs.SpikeLayer

Spike 2D Convolution.

build(input_shape)
Creates the layer weights. Must be implemented on all layers that have weights.
Parameters **input_shape** *(Union[list, tuple, Any]*) – Keras tensor (future input to layer) or list/tuple of Keras tensors to reference for weight shape computations.

```python
call(x)
```

```python
class snntoolbox.simulation.backends.inisim.tfts.SpikeAveragePooling2D(pool_size=(2, 2), strides=None, padding='valid', data_format=None, **kwargs)
```

Bases: keras.layers.pooling.AveragePooling2D, snntoolbox.simulation.backends.inisim.tfts.SpikeLayer

Average Pooling.

```python
build(input_shape)
```

Creates the layer weights. Must be implemented on all layers that have weights.

Parameters **input_shape** *(Union[list, tuple, Any]*) – Keras tensor (future input to layer) or list/tuple of Keras tensors to reference for weight shape computations.

```python
call(x)
```

```python
class snntoolbox.simulation.backends.inisim.tfts.SpikeMaxPooling2D(pool_size=(2, 2), strides=None, padding='valid', data_format=None, **kwargs)
```

Bases: keras.layers.pooling.MaxPooling2D, snntoolbox.simulation.backends.inisim.tfts.SpikeLayer

Spiking Max Pooling.

```python
build(input_shape)
```

Creates the layer neurons and connections.

Parameters **input_shape** *(Union[list, tuple, Any]*) – Keras tensor (future input to layer) or list/tuple of Keras tensors to reference for weight shape computations.

```python
call(x, mask=None)
```

Layer functionality.

### ttfs_dyn_thresh

INI time-to-first-spike simulator backend with dynamic threshold.

This module defines the layer objects used to create a spiking neural network for our built-in INI simulator `INI_ttfs_dyn_thresh_target_sim`

The coding scheme underlying this conversion is that the instantaneous firing rate is given by the inverse time-to-first-spike. In contrast to `INI_ttfs_target_sim`, this one features a threshold that adapts dynamically to the amount of input a neuron has received.

This simulator works only with Keras backend set to Tensorflow.

@author: rbodo

```python
class snntoolbox.simulation.backends.inisim.tfts_dyn_thresh.SpikeLayer(**kwargs)
```

Bases: keras.engine.base_layer.Layer
Base class for layer with spiking neurons.

**reset**(sample_idx)
Reset layer variables.

**class_name**
Get class name.

**update_neurons()**
Update neurons according to activation function.

**linear_activation**(mem)
Linear activation.

**static softmax_activation**(mem)
Softmax activation.

**get_new_mem()**
Add input to membrane potential.

**set_reset_mem**(mem, spikes)
Reset membrane potential mem array where spikes array is nonzero.

**get_psp**(output_spikes)

**get_time()**
Get simulation time variable.

Returns **time** – Current simulation time.

Return type **float**

**set_time**(time)
Set simulation time variable.

Parameters **time**(float) – Current simulation time.

**init_membrane_potential**(output_shape=None, mode='zero')
Initialize membrane potential.

Helpful to avoid transient response in the beginning of the simulation. Not needed when reset between frames is turned off, e.g. with a video data set.

Parameters

- **output_shape**(Optional[tuple]) – Output shape
- **mode**(str) – Initialization mode.
  - 'uniform': Random numbers from uniform distribution in [-thr, thr].
  - 'bias': Negative bias.
  - 'zero': Zero (default).

Returns **init_mem** – A tensor of self.output_shape (same as layer).

Return type **ndarray**

**reset_spikevars**(sample_idx)
Reset variables present in spiking layers. Can be turned off for instance when a video sequence is tested.

**init_neurons**(input_shape)
Init layer neurons.

**get_layer_idx()**
Get index of layer.
snntoolbox.simulation.backends.inisim.ttfs_dyn_thresh.spike_call(call)

class snntoolbox.simulation.backends.inisim.ttfs_dyn_thresh.SpikeConcatenate(axis, **kwargs)
    Bases: keras.layers.merge.Concatenate
    Spike merge layer
    static get_time()
    static reset(sample_idx)
        Reset layer variables.
    class_name
        Get class name.

class snntoolbox.simulation.backends.inisim.ttfs_dyn_thresh.SpikeFlatten(**kwargs)
    Bases: keras.layers.core.Flatten
    Spike flatten layer.
    call(x, mask=None)
        This is where the layer's logic lives.
        # Arguments
        inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.
        # Returns
        A tensor or list/tuple of tensors.
    static get_time()
    reset(sample_idx)
        Reset layer variables.
    class_name
        Get class name.

class snntoolbox.simulation.backends.inisim.ttfs_dyn_thresh.SpikeDense(units, activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None, **kwargs)
    Bases: keras.layers.core.Dense, snntoolbox.simulation.backends.inisim.ttfs_dyn_thresh.SpikeLayer
    Spike Dense layer.
    build(input_shape)
        Creates the layer neurons and connections.
Parameters **input_shape** *(Union[list, tuple, Any])* – Keras tensor (future input to layer) or list/tuple of Keras tensors to reference for weight shape computations.

call(x)

```python
class snntoolbox.simulation.backends.inisim.ttfs_dyn_thresh.SpikeConv2D(filters, kernel_size, strides=(1, 1), padding='valid', data_format=None, dilation_rate=(1, 1), activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None, **kwargs)
```

Bases: keras.layers.convolutional.Conv2D, snntoolbox.simulation.backends.inisim.ttfs_dyn_thresh.SpikeLayer

Spike 2D Convolution.

**build(input_shape)**

Creates the layer weights. Must be implemented on all layers that have weights.

Parameters **input_shape** *(Union[list, tuple, Any])* – Keras tensor (future input to layer) or list/tuple of Keras tensors to reference for weight shape computations.

call(x)

```python
class snntoolbox.simulation.backends.inisim.ttfs_dyn_thresh.SpikeAveragePooling2D(pool_size=(2, 2), strides=None, padding='valid', data_format=None, **kwargs)
```

Bases: keras.layers.pooling.AveragePooling2D, snntoolbox.simulation.backends.inisim.ttfs_dyn_thresh.SpikeLayer

Average Pooling.
build(input_shape)

Creates the layer weights. Must be implemented on all layers that have weights.

Parameters input_shape(Union[list, tuple, Any]) – Keras tensor (future input to layer) or list/tuple of Keras tensors to reference for weight shape computations.

call(x)

class snntoolbox.simulation.backends.inisim.ttfs_dyn_thresh.SpikeMaxPooling2D(pool_size=(2, 2), strides=None, padding='valid', data_format=None, **kwargs)

Bases: keras.layers.pooling.MaxPooling2D, snntoolbox.simulation.backends.inisim.ttfs_dyn_thresh.SpikeLayer

Spiking Max Pooling.

build(input_shape)

Creates the layer neurons and connections.

Parameters input_shape(Union[list, tuple, Any]) – Keras tensor (future input to layer) or list/tuple of Keras tensors to reference for weight shape computations.

call(x, mask=None)

Layer functionality.

ttfs_corrective

INI time-to-first-spike simulator backend with corrective spikes.

This module defines the layer objects used to create a spiking neural network for our built-in INI simulator INI_ttfs_corrective_target_sim.

The coding scheme underlying this conversion is that the instantaneous firing rate is given by the inverse time-to-first-spike. In contrast to INI_ttfs_target_sim, this one allows corrective spikes to be fired to improve the first guess made by ttfs.

This simulator works only with Keras backend set to Tensorflow.

@author: rbodo

class snntoolbox.simulation.backends.inisim.ttfs_corrective.SpikeLayer(**kwargs)

Base class for layer with spiking neurons.

reset(sample_idx)

Reset layer variables.

class_name

Get class name.

update_neurons()

Update neurons according to activation function.

get_spikes(new_mem)

Linear activation.

get_new_mem()

Add input to membrane potential.

2.4. snntoolbox.simulation
get_psp(output_spikes)

get_time()

    Get simulation time variable.

    Returns time – Current simulation time.

    Return type float

set_time(time)

    Set simulation time variable.

    Parameters time (float) – Current simulation time.

init_membrane_potential(output_shape=None, mode='zero')

    Initialize membrane potential.

    Helpful to avoid transient response in the beginning of the simulation. Not needed when reset between frames is turned off, e.g. with a video data set.

    Parameters

        • output_shape (Optional[tuple]) – Output shape

        • mode (str) – Initialization mode.

            – 'uniform': Random numbers from uniform distribution in [-thr, thr].

            – 'bias': Negative bias.

            – 'zero': Zero (default).

    Returns init_mem – A tensor of self.output_shape (same as layer).

    Return type ndarray

reset_spikevars(sample_idx)

Reset variables present in spiking layers. Can be turned off for instance when a video sequence is tested.

init_neurons(input_shape)

Init layer neurons.

get_layer_idx()

Get index of layer.

snntoolbox.simulation.backends.inisim.ttfs_corrective.spike_call(call)

class snntoolbox.simulation.backends.inisim.ttfs_corrective.SpikeConcatenate(axis, **kwargs)

    Bases: keras.layers.merge.Concatenate

    Spike merge layer

    static get_time()

    static reset(sample_idx)

        Reset layer variables.

    class_name

        Get class name.

class snntoolbox.simulation.backends.inisim.ttfs_corrective.SpikeFlatten(**kwargs)

    Bases: keras.layers.core.Flatten

    Spike flatten layer

    static get_time()
reset(sample_idx)
Reset layer variables.

class_name
Get class name.

class snntoolbox.simulation.backends.inisim.ttfs_corrective.SpikeDense (units,
activation=None,
use_bias=True,
kernel_initializer='glorot_uniform',
bias_initializer='zeros',
kernel_regularizer=None,
bias_regularizer=None,
activity_regularizer=None,
kernel_constraint=None,
bias_constraint=None,**kwargs)

Bases: keras.layers.core.Dense, snntoolbox.simulation.backends.inisim.
ttfs_corrective.SpikeLayer

Spike Dense layer.

build(input_shape)
Creates the layer neurons and connections.

Parameters
input_shape (Union[list, tuple, Any]) – Keras tensor (future input
to layer) or list/tuple of Keras tensors to reference for weight shape computations.

call(x)
class snntoolbox.simulation.backends.inisim.ttfs_corrective.SpikeConv2D(filters, kernel_size, strides=(1, 1), padding='valid', data_format=None, dilation_rate=(1, 1), activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None, **kwargs)

Bases: keras.layers.convolutional.Conv2D, snntoolbox.simulation.backends.inisim.ttfs_corrective.SpikeLayer

Spike 2D Convolution.

build(input_shape)

Creates the layer weights. Must be implemented on all layers that have weights.

Parameters

**input_shape** (Union[list, tuple, Any]) – Keras tensor (future input to layer) or list/tuple of Keras tensors to reference for weight shape computations.

call(x)

class snntoolbox.simulation.backends.inisim.ttfs_corrective.SpikeAveragePooling2D(pool_size=(2, 2), strides=None, padding='valid', data_format=None, **kwargs)

Bases: keras.layers.pooling.AveragePooling2D, snntoolbox.simulation.backends.inisim.ttfs_corrective.SpikeLayer

Average Pooling.

build(input_shape)

Creates the layer weights. Must be implemented on all layers that have weights.

Parameters

**input_shape** (Union[list, tuple, Any]) – Keras tensor (future input to layer) or list/tuple of Keras tensors to reference for weight shape computations.
call(x)

class snntoolbox.simulation.backends.inisim.ttfs_corrective.SpikeMaxPooling2D(pool_size=(2, 2), strides=None, padding='valid', data_format=None, **kwargs)

Bases: keras.layers.pooling.MaxPooling2D, snntoolbox.simulation.backends.inisim.ttfs_corrective.SpikeLayer

Spiking Max Pooling.

build(input_shape)

Creates the layer neurons and connections.

Parameters

input_shape (Union[list, tuple, Any]) – Keras tensor (future input to layer) or list/tuple of Keras tensors to reference for weight shape computations.

call(x)

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.

@author: evan

snntoolbox.simulation.backends.megasim.megasim.megasim_path()

2.4.4 snntoolbox.simulation.target_simulators

INI_target_sim

INI_temporal_mean_rate_target_sim

INI simulator with temporal mean rate code.

@author: rbodo

class snntoolbox.simulation.target_simulators.INI_temporal_mean_rate_target_sim.SNN(config, queue=None)

Bases: snntoolbox.simulation.utils.AbstractSNN

The compiled spiking neural network, using layers derived from Keras base classes (see snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow).

Aims at simulating the network on a self-implemented Integrate-and-Fire simulator using a timestepped approach.

snn

Keras model. This is the output format of the compiled spiking model because INI simulator runs networks of layers that are derived from Keras base classes.

Type keras.models.Model
is_parallelizable
Whether or not the simulator is able to test multiple samples in parallel.

add_input_layer(input_shape)
Add input layer.

Parameters
input_shape (list / tuple) – Input shape to the network, including the
batch size as first dimension.

add_layer(layer)
Do anything that concerns adding any layer independently of its type.

Parameters
layer (keras.layers.Layer / keras.layers.Conv) – Layer

build_dense(layer)
Build spiking fully-connected layer.

Parameters
layer (keras.layers.Dense) – Layer

build_convolution(layer)
Build spiking convolutional layer.

Parameters
layer (keras.layers.Layer) – Layer

build_pooling(layer)
Build spiking pooling layer.

Parameters
layer (keras.layers.pooling._Pooling2D) – Layer

compile()
Compile the spiking network.

simulate(**kwargs)
Simulate a spiking network for a certain duration, and record any variables of interest (spike trains, mem-
brane potentials, . . . )

Returns
output_b_l_t – Array of shape (batch_size, num_classes,
num_timesteps), containing the number of output spikes of the neurons in the
final layer, for each sample and for each time step during the simulation.

Return type
ndarray

reset(sample_idx)
Reset network variables.

Parameters
sample_idx (int) – Index of sample that has just been simulated. In certain
applications (video data), we may want to turn off reset between samples.

end_sim()
Clean up after simulation.

save(path, filename)
Write model architecture and parameters to disk.

Parameters

- path (string) – Path to directory where to save model.
- filename (string) – Name of file to write model to.

load(path, filename)
Load model architecture and parameters from disk.

Parameters

- path (str) – Path to directory where to load model from.
filename (str) – Name of file to load model from.

get_poisson_frame_batch (x_b_l)
Get a batch of Poisson input spikes.

Parameters x_b_l (ndarray) – The input frame. Shape: (batch_size, layer_shape).

Returns input_b_l – Array of Poisson input spikes, with same shape as x_b_l.

Return type ndarray

set_time (t)
Set the simulation time variable of all layers in the network.

Parameters t (float) – Current simulation time.

set_spiketrain_stats_input ()
Count number of operations based on the input spike activity during simulation.

get_spiketrains_input ()
Get spike trains of input layer.

Returns spiketrains_b_l_t – Spike trains of input.

Return type ndarray

scale_first_layer_parameters (t, input_b_l, tau=1)

INI_temporal_pattern_target_sim

INI simulator with temporal pattern code.

@author: rbodo

class snntoolbox.simulation.target_simulators.INI_temporal_pattern_target_sim

Bases: snntoolbox.simulation.target_simulators.INI_temporal_mean_rate_target_sim

SNN

The compiled spiking neural network, using layers derived from Keras base classes (see snntoolbox.simulation.backends.inisim.temporal_pattern).

Aims at simulating the network on a self-implemented Integrate-and-Fire simulator using a timestepped approach.

snn

Keras model. This is the output format of the compiled spiking model because INI simulator runs networks of layers that are derived from Keras layer base classes.

Type keras.models.Model

compile ()
Compile the spiking network.

simulate (**kwargs)
Simulate a spiking network for a certain duration, and record any variables of interest (spike trains, membrane potentials, ...).

Returns output_b_l_t – Array of shape (batch_size, num_classes, num_timesteps), containing the number of output spikes of the neurons in the final layer, for each sample and for each time step during the simulation.

Return type ndarray
**load** *(path, filename)*
Load model architecture and parameters from disk.

**Parameters**
- **path** *(str)* – Path to directory where to load model from.
- **filename** *(str)* – Name of file to load model from.

**set_spiketrains** *(spiketrains_b_l_t, i)*
**set_spikerates** *(spikerates_b_l, i)*
**set_neuron_operations** *(i)*
**set_synaptic_operations** *(spiketrains_b_l_t, i)*
**spikerates_to_trains** *(spikerates_b_l)*

**INI_ttfs_target_sim**

INI simulator with time-to-first-spike code.

@author: rbodo

class snntoolbox.simulation.target_simulators.INI_ttfs_target_sim.SNN(config, queue=None)
    
    Bases: snntoolbox.simulation.target_simulators.INI_temporal_mean_rate_target_sim.SNN

    The compiled spiking neural network, using layers derived from Keras base classes (see snntoolbox.simulation.backends.inisim.ttfs).

    Aims at simulating the network on a self-implemented Integrate-and-Fire simulator using a timestepped approach.

    **simulate** (**kwargs**)
    
    Simulate a spiking network for a certain duration, and record any variables of interest (spike trains, membrane potentials,...)

    **Returns** output_b_l_t – Array of shape *(batch_size, num_classes, num_timesteps)*, containing the number of output spikes of the neurons in the final layer, for each sample and for each time step during the simulation.

    **Return type** ndarray

**load** *(path, filename)*
Load model architecture and parameters from disk.

**Parameters**
- **path** *(str)* – Path to directory where to load model from.
- **filename** *(str)* – Name of file to load model from.

**INI_ttfs_dyn_thresh_target_sim**

INI simulator with time-to-first-spike code and a dynamic threshold.

@author: rbodo
**class** snntoolbox.simulation.target_simulators.INI_ttfs_dyn_thresh_target_sim.SNN(config, queue=None)

**Bases:** snntoolbox.simulation.target_simulators.INI_temporal_mean_rate_target_sim.SNN

The compiled spiking neural network, using layers derived from Keras base classes (see snntoolbox.simulation.backends.inisim.ttfs_dyn_thresh).

Aims at simulating the network on a self-implemented Integrate-and-Fire simulator using a timestepped approach.

**simulate(**kwargs**)

Simulate a spiking network for a certain duration, and record any variables of interest (spike trains, membrane potentials, ...)

**Returns output_b_l_t** – Array of shape `(batch_size, num_classes, num_timesteps)`, containing the number of output spikes of the neurons in the final layer, for each sample and for each time step during the simulation.

**Return type** ndarray

**load** *(path, filename)*

Load model architecture and parameters from disk.

**Parameters**

- **path** *(str)* – Path to directory where to load model from.
- **filename** *(str)* – Name of file to load model from.

INI_ttfs_corrective_target_sim

INI simulator with time-to-first-spike code and corrective spikes.

@author: rbodo

pyNN_target_sim

Building and simulating spiking neural networks using pyNN.

@author: rbodo

**class** snntoolbox.simulation.target_simulators.pyNN_target_sim.SNN(config, queue=None)

**Bases:** snntoolbox.simulation.utils.AbstractSNN

Class to hold the compiled spiking neural network.

Represents the compiled spiking neural network, ready for testing in a spiking simulator.

**layers**

Each entry represents a layer, i.e. a population of neurons, in form of pyNN Population objects.

**Type** list[pyNN.Population]

**connections**

pyNN Projection objects representing the connections between individual layers.

**Type** list[pyNN.Projection]

**cellparams**

Neuron cell parameters determining properties of the spiking neurons in pyNN simulators.
Type: dict

**is_parallelizable**
Whether or not the simulator is able to test multiple samples in parallel.

**add_input_layer**(input_shape)
Add input layer.

Parameters:
**input_shape** (list | tuple) – Input shape to the network, including the batch size as first dimension.

**add_layer**(layer)
Do anything that concerns adding any layer independently of its type.

Parameters:
**layer** (keras.layers.Layer | keras.layers.Conv) – Layer

**build_dense**(layer)

Parameters:
**layer** (keras.layers.Dense) –

**build_convolution**(layer)
Build spiking convolutional layer.

Parameters:
**layer** (keras.layers.Layer) – Layer

**build_pooling**(layer)
Build spiking pooling layer.

Parameters:
**layer** (keras.layers.pooling._Pooling2D) – Layer

**compile**()
Compile the spiking network.

**simulate**(**kwargs)
Simulate a spiking network for a certain duration, and record any variables of interest (spike trains, membrane potentials, ...)

Returns:
**output_b_l_t** – Array of shape (batch_size, num_classes, num_timesteps), containing the number of output spikes of the neurons in the final layer, for each sample and for each time step during the simulation.

Return type:
ndarray

**reset**(sample_idx)
Reset network variables.

Parameters:
**sample_idx** (int) – Index of sample that has just been simulated. In certain applications (video data), we may want to turn off reset between samples.

**end_sim**()
Clean up after simulation.

**save**(path, filename)
Write model architecture and parameters to disk.

Parameters:
- **path** (string) – Path to directory where to save model.
- **filename** (string) – Name of file to write model to.

**load**(path, filename)
Load model architecture and parameters from disk.

Parameters:
• **path** (*str*) – Path to directory where to load model from.

• **filename** (*str*) – Name of file to load model from.

**init_cells** ()
Set cell parameters of neurons in each layer and initialize membrane potential.

**set_biases** (*biases*)
Set biases.

**get_vars_to_record** ()
Get variables to record during simulation.

Returns **vars_to_record** – The names of variables to record during simulation.

Return type **list[**str**]**

**get_spiketrains** (**kwargs**)
Get spike trains of a layer.

Returns **spiketrains_b_l_t** – Spike trains.

Return type **ndarray**

**get_spiketrains_input** ()
Get spike trains of input layer.

Returns **spiketrains_b_l_t** – Spike trains of input.

Return type **ndarray**

**get_spiketrains_output** ()
Get spike trains of output layer.

Returns **spiketrains_b_l_t** – Spike trains of output.

Return type **ndarray**

**get_vmem** (**kwargs**)
Get membrane potentials of a layer.

Returns **mem_b_l_t** – Membrane potentials of layer.

Return type **ndarray**

**save_assembly** (*path, filename*)
Write layers of neural network to disk.

The size, structure, labels of all the population of an assembly are stored in a dictionary such that one can load them again using the **load_assembly** function.

The term “assembly” refers to pyNN internal nomenclature, where Assembly is a collection of layers (Populations), which in turn consist of a number of neurons (cells).

Parameters

• **path** (*str*) – Path to directory where to save layers.

• **filename** (*str*) – Name of file to write layers to.

**save_connections** (*path*)
Write parameters of a neural network to disk.

The parameters between two layers are saved in a text file. They can then be used to connect pyNN populations e.g. with **sim.Projection(layer1, layer2, sim.FromListConnector(filename))**, where sim is a simulator supported by pyNN, e.g. Brian, NEURON, or NEST.
Parameters **path** (*str*) – Path to directory where connections are saved.

Returns
- *Text files containing the layer connections. Each file is named after the layer it connects to, e.g. layer2.txt if connecting layer1 to layer2.*

**save_biases** (*path*)
Enter biases of a neural network to disk.

Parameters **path** (*str*) – Path to directory where connections are saved.

**load_assembly** (*path*, *filename*)
Load the populations in an assembly.

Loads the populations in an assembly that was saved with the `save_assembly` function.

The term “assembly” refers to pyNN internal nomenclature, where Assembly is a collection of layers (Populations), which in turn consist of a number of neurons (cells).

Parameters
- **path** (*str*) – Path to directory where to load model from.
- **filename** (*str*) – Name of file to load model from.

Returns **layers** – List of pyNN Population objects.

Return type `list[pyNN.Population]`

**set_spiketrain_stats_input**()
Count number of operations based on the input spike activity during simulation.

**brian2_target_sim**
Building and simulating spiking neural networks using Brian2.

@author: rbodo

Parameters
- **config**
- **queue** (*None*)

Bases: `object`

A callback which draws a progress bar in the terminal.

**layers**
Each entry represents a layer, i.e. a population of neurons, in form of Brian2 NeuronGroup objects.

Type `list[brian2.NeuronGroup]`

**connections**
Brian2 Synapses objects representing the connections between individual layers.

Type `list[brian2.Synapses]`

**threshold**
Defines spiking threshold.
Type  str

v_reset
  Defines reset potential.
  Type  str

eqs
  Differential equation for membrane potential.
  Type  str

spikemonitors
  Brian2 SpikeMonitor s for each layer that records spikes.
  Type  list[brian2.SpikeMonitor]

statemonitors
  Brian2 StateMonitor s for each layer that records membrane potential.
  Type  list[brian2.StateMonitor]

snn
  The spiking network.
  Type  brian2.Network

is_parallelizable
  Whether or not the simulator is able to test multiple samples in parallel.

add_input_layer (input_shape)
  Add input layer.

  Parameters input_shape (list / tuple) – Input shape to the network, including the
  batch size as first dimension.

add_layer (layer)
  Do anything that concerns adding any layer independently of its type.

  Parameters layer (keras.layers.Layer | keras.layers.Conv) – Layer

build_dense (layer, weights=None)
  Build spiking fully-connected layer.

  Parameters layer (keras.layers.Dense) – Layer

build_convolution (layer, weights=None)
  Build spiking convolutional layer.

  Parameters layer (keras.layers.Layer) – Layer

build_pooling (layer, weights=None)
  Build spiking pooling layer.

  Parameters layer (keras.layers.pooling._Pooling2D) – Layer

compile ()
  Compile the spiking network.

simulate (**kwargs)
  Simulate a spiking network for a certain duration, and record any variables of interest (spike trains, mem-
  brane potentials, . . . )

  Returns output_b_l_t – Array of shape (batch_size, num_classes, num_timesteps), containing the number of output spikes of the neurons in the
  final layer, for each sample and for each time step during the simulation.
Return type  ndarray

```
reset(sample_idx)
```
Reset network variables.

Parameters sample_idx (int) – Index of sample that has just been simulated. In certain applications (video data), we may want to turn off reset between samples.

```
end_sim()
```
Clean up after simulation.

```
save(path, filename)
```
Write model architecture and parameters to disk.

Parameters
- **path** (str) – Path to directory where to save model.
- **filename** (str) – Name of file to write model to.

```
load(path, filename)
```
Load model architecture and parameters from disk.

Parameters
- **path** (str) – Path to directory where to load model from.
- **filename** (str) – Name of file to load model from.

```
init_cells()
```
Set cell parameters of neurons in each layer and initialize membrane potential.

```
get_spiketrains(**kwargs)
```
Get spike trains of a layer.

Returns spiketrains_b_l_t – Spike trains.

Return type  ndarray

```
get_spiketrains_input()
```
Get spike trains of input layer.

Returns spiketrains_b_l_t – Spike trains of input.

Return type  ndarray

```
get_spiketrains_output()
```
Get spike trains of output layer.

Returns spiketrains_b_l_t – Spike trains of output.

Return type  ndarray

```
get_vmem(**kwargs)
```
Get membrane potentials of a layer.

Returns mem_b_l_t – Membrane potentials of layer.

Return type  ndarray

```
set_spiketrain_stats_input()
```
Count number of operations based on the input spike activity during simulation.

```
set_biases(biases)
```
Set biases.
Notes

This has not been tested yet.

MegaSim_target_sim

MegaSim asynchronous spiking simulator.

@author: Evangelos Stromatias

class snntoolbox.simulation.target_simulators.MegaSim_target_sim.Megasim_base
    Bases: abc.ABC

Class that holds the common attributes and methods for the MegaSim modules.

Attributes set to -1 must be set by each subclass, the rest can be used as default values.

Attributes common to all MegaSim modules

n_in_ports
    Number of input ports
    Type int

n_out_ports
    Number of output ports
    Type int

delay_to_process
    Delay to process an input event
    Type int

delay_to_ack
    Delay to acknowledge an event
    Type int

fifo_depth
    Depth of input fifo
    Type int

n_repeat
    Type int

delay_to_repeat
    Type int

#Parameters for the convolutional module and average pooling module

Nx_array
    X dimensions of the feature map
    Type int

Ny_array
    Y dimensions of the feature map
    Type int

Xmin
    start counting from Xmin (=0)
Type int

Ymin
    start counting from Ymin (=0)
    Type int

THplus
    Positive threshold

THplusInfo
    Flag to enable spikes when reaching the positive threshold

THminus
    Negative threshold

THminusInfo
    Flag to enable spikes with negative polarity when reaching the negative threshold

Reset_to_reminder
    After reaching the threshold if set it will set the membrane to the difference

MembReset
    Resting potential (=0)
    Type int

TLplus
    Linear leakage slope from the positive threshold
    Type int

TLminus
    Linear leakage slope from the negative threshold
    Type int

Tmin
    minimum time between 2 spikes
    Type int

T_Refract
    Refractory period
    Type int

# Parameters for the output

crop_xmin
    Xmin crop of the feature map
    Type int

crop xmax
    Type int

crop ymin
    Type int

crop ymax
    Type int
### xshift_pre
X shift before subsampling
Type int

### yshift_pre
Y shift before subsampling
Type int

### x_subsmp
Subsampling (=1 if none)
Type int

### y_subsmp
Type int

### xshift_pos
X shift after subsampling
Type int

### yshift_pos
Type int

### rectify
Flag that if set will force all spikes to have positive polarity
Type int

---

# The fully connected module has population_size instead of Nx_array

### population_size
Number of neurons in the fully connected module
Type int

### Nx_array_pre
Number of neurons in the previous layer
Type int

---

# Needed by the state file

### time_busy_initial
Initial state of the module (=0)
Type int

---

# Scaling factor

### scaling_factor
Scaling factor for the parameters since MegaSim works with integers
Type int

---

### build_state_file:
Input parameters: a string with the full path to the megasim SNN directory
This method is similar to all MegaSim modules. It generates an initial state file per module based on the time_busy_initial.
build_parameter_file:

Input parameters: a string with the full path to the megasim SNN directory

This method generates the module’s parameter file based on its attributes set by the sub-class. This method
depends on the MegaSim module and will raise error if not implemented.

n_in_ports = -1
n_out_ports = 1
delay_to_process = 0
delay_to_ack = 0
fifo_depth = 0
n_repeat = 1
delay_to_repeat = 15
Nx_array = -1
Ny_array = 1
Xmin = 0
Ymin = 0
THplus = 0
THplusInfo = 1
THminus = -2147483646
THminusInfo = 0
Reset_to_reminder = 0
MembReset = 0
TLplus = 0
TLminus = 0
Tmin = 0
T_Refract = 0
crop_xmin = -1
crop_xmax = -1
crop_ymin = -1
crop_ymax = -1
xshift_pre = 0
yshift_pre = 0
x_subsmp = 1
y_subsmp = 1
xshift_pos = 0
yshift_pos = 0
rectify = 0
population_size = -1
Nx_array_pre = -1

time_busy_initial = 0

scaling_factor = 1

build_state_file(dirname)
    dirname = the full path of the

build_parameter_file(dirname)

class snntoolbox.simulation.target_simulators.MegaSim_target_sim.module_input_stimulus(label, pop_size)

Bases: object

A dummy class for the input stimulus.

Parameters

• **label** *(string)* – String to hold the module’s name.

• **pop_size** *(int)* – Integer to store the population size.

**label**
    Type string

**pop_size**
    Type int

**input_stimulus_file**
    String to hold the filename of the input stimulus

    Type string

**module_string**
    String that holds the module name for megasim

    Type string

**evs_files**
    List of strings of the event filenames that will generated when a megasim simulation is over.

    Type list

class snntoolbox.simulation.target_simulators.MegaSim_target_sim.module_flatten(layer_params, input_ports, fm_size)

Bases: snntoolbox.simulation.target_simulators.MegaSim_target_sim.Megasim_base

Megasim_base

A class for the flatten megasim module. The flatten module is used to connect a 3D population to a 1D population. eg A convolutional layer to a fully connected one.

Parameters

• **layer_params** *(Keras layer)* – Layer from parsed input model.

• **input_ports** *(int)* – Number of input ports (eg feature maps from the previous layer)

• **fm_size** *(tuple)* – Tuple of integers that holds the size of the feature maps from the previous layer

**module_string**
    String that holds the module name for megasim

2.4. snntoolbox.simulation
Type  string

output_shapes
Tuple that holds the shape of the output of the module. Used for the plotting.

Type  tuple

evs_files
List of strings of the event filenames that will generated when a megasim simulation is over.

Type  list

build_parameter_file (dirname)

class snntoolbox.simulation.target_simulators.MegaSim_target_sim.Module_average_pooling (layer_params, neuron_params, reset_input_event=False, scaling_factor=1000000)

Bases:  

snntoolbox.simulation.target_simulators.MegaSim_target_sim.Megasim_base

duplicate code with the module_conv class - TODO: merge them

layer_params Attributes: ['label', 'layer_num', 'padding', 'layer_type', 'strides', 'input_shape', 'output_shape', 'get_activ', 'pool_size']

build_parameter_file (dirname)

class snntoolbox.simulation.target_simulators.MegaSim_target_sim.Module_conv (layer_params, neuron_params, flip_kernels=True, reset_input_event=False, scaling_factor=1000000)

Bases:  

snntoolbox.simulation.target_simulators.MegaSim_target_sim.Megasim_base

A class for the convolutional megasim module.

Parameters

- layer_params (Keras layer) – Layer from parsed input model.
- neuron_params (dictionary) – This is the settings dictionary that is set in the config.py module
- flip_kernels (boolean) – If set will flip the kernels upside down.
- scaling_factor (int) – An integer that will be used to scale all parameters.

module_string
String that holds the module name for megasim

Type  string

output_shapes
Tuple that holds the shape of the output of the module. Used for the plotting.

Type  tuple
**evs_files**
List of strings of the event filenames that will generated when a megasim simulation is over.

Type list

**num_of_FMs**
Number of feature maps in this layer

Type int

**w**
list of weights

Type list

**padding**
String with the border mode used for the convolutional layer. So far only the valid mode is implemented

Type string

layer_params Attributes: ['kernel_size', 'activation', 'layer_type', 'layer_num', 'filters', 'output_shape', 'input_shape', 'label', 'parameters', 'padding']

**build_parameter_file**(dirname)

class** snntoolbox.simulation.target_simulators.MegaSim_target_sim.Module_fully_connected**(layer_params, neuron_params, scaling_factor=1000000, reset_input_event=False, enable_softmax=True)

Bases: snntoolbox.simulation.target_simulators.MegaSim_target_sim.Megasim_base

A class for the fully connected megasim module.

Parameters

- **layer_params**(Keras layer) – Layer from parsed input model.
- **neuron_params**(dictionary) – This is the settings dictionary that is set in the config.py module
- **scaling_factor**(int) – An integer that will be used to scale all parameters.
- **enable_softmax**(Boolean) – A flag that if set will use (if the ann uses it) softmax for the output layer. If not set a population of LIF neurons will be used instead.

**module_string**
String that holds the module name for megasim

Type string

**output_shapes**
Tuple that holds the shape of the output of the module. Used for the plotting.

Type tuple

**evs_files**
List of strings of the event filenames that will generated when a megasim simulation is over.

Type list
num_of_FMs
    Number of feature maps in this layer

    Type int

w
    list of weights

    Type list

padding
    String with the border mode used for the convolutional layer. So far only the valid mode is implemented

    Type string

layer_params
Attributes: ['kernel_size', 'activation', 'layer_type', 'layer_num', 'filters', 'output_shape', 'input_shape', 'label', 'parameters', 'padding']

build_parameter_file (dirname)

build_softmax_conrol_events (megadirname, duration, dt, input_rate, softmax_clockrate=300)

class snntoolbox.simulation.target_simulators.MegaSim_target_sim.SNN (config, queue=None)

    Bases: snntoolbox.simulation.utils.AbstractSNN

    Represents the compiled spiking neural network, ready for testing in a spiking simulator.

layers
    Each entry represents a layer.

    Type list

connections
    The connections between layers.

    Type list

megasim_path
    The path to megasim installation directory.

    Type str

megaschematic
    String that holds megasim main schmatic file that is needed to run a simulation

    Type str

megadirname
    String that holds the full path where the generated files for a megasim simulation are stored. These files include the stimulus, parameter, state and schematic files. The event files will be generated in the same folder.

    Type str

input_stimulus_file
    Filename of input stimulus.

    Type str

cellparams
    Neuron cell parameters determining properties of the spiking neurons.

    Type dict
use_biases
Whether or not to use biases.

Type bool

is_parallelizable
Whether or not the simulator is able to test multiple samples in parallel.

add_input_layer (input_shape)
Add input layer.

Parameters input_shape (list | tuple) – Input shape to the network, including the batch size as first dimension.

add_layer (layer)
Do anything that concerns adding any layer independently of its type.

Parameters layer (keras.layers.Layer | keras.layers.Conv) – Layer

build_dense (layer)
Build spiking fully-connected layer.

Parameters layer (keras.layers.Dense) – Layer

build_convolution (layer)
Build spiking convolutional layer.

Parameters layer (keras.layers.Layer) – Layer

build_pooling (layer)
Build spiking pooling layer.

Parameters layer (keras.layers.pooling._Pooling2D) – Layer

build_flatten (layer)
Build flatten layer.

May not be needed depending on the simulator.

Parameters layer (keras.layers.Layer) – Layer

compile ()
Compile the spiking network.

simulate (**kwargs)
Simulate a spiking network for a certain duration, and record any variables of interest (spike trains, membrane potentials, ...)

Returns output_b_l_t – Array of shape (batch_size, num_classes, num_timesteps), containing the number of output spikes of the neurons in the final layer, for each sample and for each time step during the simulation.

Return type ndarray

reset (sample_idx)
Reset network variables.

Parameters sample_idx (int) – Index of sample that has just been simulated. In certain applications (video data), we may want to turn off reset between samples.

disable_output

end_sim ()
Clean up after simulation.

save (path, filename)
Write model architecture and parameters to disk.
Parameters

- `path (string)` – Path to directory where to save model.
- `filename (string)` – Name of file to write model to.

`load (path, filename)`
Load model architecture and parameters from disk.

Parameters

- `path (str)` – Path to directory where to load model from.
- `filename (str)` – Name of file to load model from.

`get_spiketrains (**kwargs)`
Get spike trains of a layer.

Returns `spiketrains_b_l_t` – Spike trains.
Return type ndarray

`get_spiketrains_input ()`  
Get spike trains of input layer.

Returns `spiketrains_b_l_t` – Spike trains of input.
Return type ndarray

`get_spiketrains_output ()`  
Get spike trains of output layer.

Returns `spiketrains_b_l_t` – Spike trains of output.
Return type ndarray

`get_vmem (**kwargs)`
Get membrane potentials of a layer.

Returns `mem_b_l_t` – Membrane potentials of layer.
Return type ndarray

`get_spikes ()`  
Method that fetches all the events from all layers after a simulation is over.

Returns

Return type A list of all the events from all the layers.

`get_output_spikes_batch ()`  
Method that fetches the events from the output layer.

Returns

Return type A numpy array of the output events.

`static spike_count_histogram (events, pop_size=10)`
This method first creates a histogram based on the size of the layer and then returns the argmax of the neuron that fired the most spikes for that particular stimulus.

If there are no spikes it will return -1.

Parameters

- `events (list)` – List of megasim events of a particular layer
- `pop_size (int)` – Size of the fully connected module
Returns

Return type  pop_spike_hist

**static check_megasim_output** *(megalog)*
A method that checks the prints of MegaSim for errors

Parameters  
megalog *(str)* – String returned from executing megasim.

**poisson_spike_generator_megasim** *(mnist_digit)*

Parameters  
mnist_digit *(ndarray)* – A 1d or 2d numpy array of an mnist digit (normalised 0-1).

Returns

• It will store the generated spike trains to a stimulus file in the
  megasim sim folder.

**poisson_spike_generator_batchmode_megasim** *(mnist_digits)*

Parameters  
mnist_digits *(ndarray)* – A 1d or 2d numpy array of an mnist digit (normalised 0-1).

Returns

• It will store the generated spike trains to a stimulus file in the
  megasim sim folder.

**generate_bias_clk** *(timestamp_batches)*
An external periodic (per timestep) event is used to trigger the biases, since megasim simulator is not a
time-stepped simulator.

Parameters  
timestamp_batches *(List[list])* – List that includes the first and last
timestamps of the input source, e.g. [[start0, stop0], [start1, stop1]].

Returns

Return type  Generates a megasim stimulus file in the experiment folder.

**build_schematic_updated** ()
This method generates the main MegaSim schematic file

TODO: this method is quite ugly! need to refactor it 99.20 non normalised first 100 samples = 100% reset
to zero ——-

**clean_megasim_sim_data** ()
A method that removes the previous stimulus file and generated event files before testing a new sample.

**set_spiketrain_stats_input** ()
Count number of operations based on the input spike activity during simulation.

---

### 2.5 snntoolbox.datasets

The modules in this package provide functionality to load and process datasets into the SNN conversion toolbox.

#### 2.5.1 snntoolbox.datasets.utils

The main purpose of this module is to load a dataset from disk and feed it to the toolbox in one of the formats it can handle.
For details see

| get_dataset | Get dataset, either from .npz files or keras.ImageDataGenerator. |

@author: rbodo

```python
snntoolbox.datasets.utils.get_dataset(config)
```

Get dataset, either from .npz files or keras.ImageDataGenerator.

Returns Dictionaries with keys `x_test` and `y_test` if data set was loaded in .npz format, or with `dataflow` key if data will be loaded from .jpg, .png, or .bmp files by a keras.ImageDataGenerator.

Parameters

- `config` (configparser.ConfigParser) – Settings.

Returns

- `normset` (dict) – Used to normalized the network parameters.
- `testset` (dict) – Used to test the networks.

```python
snntoolbox.datasets.utils.try_get_normset_from_scalefacs(config)
```

Instead of loading a normalization data set to calculate scale-factors, try to get the scale-factors stored on disk during a previous run.

Parameters

- `config` (configparser.ConfigParser) – Settings.

Returns

A dictionary with single key `scale_facs`. The corresponding value is itself a dictionary containing the scale factors for each layer. Returns `None` if no scale factors were found.

Return type Union[dict, None]

```python
snntoolbox.datasets.utils.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).

```python
snntoolbox.datasets.utils.load_npz(path, filename)
```

Load dataset from an .npz file.

Parameters

- `filename` (string) – Name of file.
- `path` (string) – Location of dataset to load.

Returns

The dataset as a numpy array containing samples.

Return type tuple[np.array]

## 2.6 snntoolbox.utils

### 2.6.1 snntoolbox.utils.utils

General utility functions on project-level.

@author: rbodo
snntoolbox.utils.utils.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

• \texttt{start} (float) – The starting value of the sequence
• \texttt{stop} (float) – End value of the sequence.
• \texttt{num} (int) – Number of samples to generate. Must be non-negative.
• \texttt{method} (str) – The sequence will be computed on either a linear, logarithmic or random grid.

Returns \texttt{samples} – There are \texttt{num} samples in the closed interval [start, stop].

Return type \texttt{np.array}

snntoolbox.utils.utils.confirm_overwrite\((filepath)\)

If config.get('output', \, 'overwrite')==False and the file exists, ask user if it should be overwritten.

snntoolbox.utils.utils.to_json\((data, \, path)\)

Write data dictionary to path.

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

snntoolbox.utils.utils.import_helpers\((filepath, \, config)\)

Import a module with helper functions from filepath.

Parameters

• \texttt{filepath} (str) – Filename or relative or absolute path of module to import. If only the filename is given, module is assumed to be in current working directory (config.get('paths', \, 'path_wd')). Non-absolute paths are taken relative to working dir.
• \texttt{config} (configparser.ConfigParser) – Settings.

Returns Module with helper functions.

snntoolbox.utils.utils.get_abs_path\((filepath, \, config)\)

Get an absolute path, possibly using current toolbox working dir.

Parameters

• \texttt{filepath} (str) – Filename or relative or absolute path. If only the filename is given, file is assumed to be in current working directory (config.get('paths', \, 'path_wd')). Non-absolute paths are interpreted relative to working dir.
• \texttt{config} (configparser.ConfigParser) – Settings.

Returns \texttt{path} – Absolute path to file.

Return type \texttt{str}

snntoolbox.utils.utils.import_script\((path, \, filename)\)

Import python script independently from python version.

Parameters

• \texttt{path} (string) – Path to directory where to load script from.
• \texttt{filename} (string) – Name of script file.
snntoolbox.utils.utils.binary_tanh(x)
Round a float to -1 or 1.

Parameters  
\( x (\text{float}) \) –

Returns  
Integer in \{-1, 1\}

Return type  
\( \text{int} \)

snntoolbox.utils.utils.binary.sigmoid(x)
Round a float to 0 or 1.

Parameters  
\( x (\text{float}) \) –

Returns  
Integer in \{0, 1\}

Return type  
\( \text{int} \)

snntoolbox.utils.utils.hard.sigmoid(x)

Parameters  
\( x \) –

snntoolbox.utils.utils.binarize_var(w, h=1.0, deterministic=True)
Binarize shared variable.

Parameters

• \( w (\text{keras.backend.Variable}) \) – Weights.

• \( h (\text{float}) \) – Values are round to +/-h.

• \( \text{deterministic} (\text{bool}) \) – Whether to apply deterministic rounding.

Returns  
\( w \) – The binarized weights.

Return type  
\( \text{keras.backend.variable} \)

snntoolbox.utils.utils.binarize(w, h=1.0, deterministic=True)
Binarize weights.

Parameters

• \( w (\text{ndarray}) \) – Weights.

• \( h (\text{float}) \) – Values are round to +/-h.

• \( \text{deterministic} (\text{bool}) \) – Whether to apply deterministic rounding.

Returns  
The binarized weights.

Return type  
\( \text{ndarray} \)

snntoolbox.utils.utils.reduce_precision(x, m, f)
Reduces precision of \( x \) to format \( Q^m.f \).

Parameters

• \( x (\text{ndarray}) \) – Input data.

• \( m (\text{int}) \) – Number of integer bits.

• \( f (\text{int}) \) – Number of fractional bits.

Returns  
\( x_{lp} \) – The input data with reduced precision.

Return type  
\( \text{ndarray} \)

snntoolbox.utils.utils.reduce_precision_var(x, m, f)
Reduces precision of \( x \) to format \( Q^m.f \).
Parameters

- \( x (\text{keras.backend.variable}) \) – Input data.
- \( m (\text{int}) \) – Number of integer bits.
- \( f (\text{int}) \) – Number of fractional bits.

Returns \( x_{lp} \) – The input data with reduced precision.

Return type \( \text{keras.backend.variable} \)

\[ \text{snntoolbox.utils.utils.quantized_relu}(x, m, f) \]
Rectified linear unit activation function with precision of \( x \) reduced to fixed point format \( \text{Qm.f} \).

Parameters

- \( x (\text{keras.backend.variable}) \) – Input data.
- \( m (\text{int}) \) – Number of integer bits.
- \( f (\text{int}) \) – Number of fractional bits.

Returns \( x_{lp} \) – The input data with reduced precision.

Return type \( \text{keras.backend.variable} \)

\[ \text{snntoolbox.utils.utils.ClampedReLU}(\text{threshold}=0.1, \text{max_value}=\text{None}) \]
Rectified linear unit activation function where values in \( x \) below \( \text{threshold} \) are clamped to 0, and values above \( \text{max_value} \) are clipped to \( \text{max_value} \).

- \( \text{threshold} \)
  Type \( \text{Optional[float]} \)

- \( \text{max_value} \)
  Type \( \text{Optional[float]} \)

\[ \text{snntoolbox.utils.utils.wilson_score}(p, n) \]
Confidence interval of a binomial distribution.


Parameters

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

Returns

Return type \( \text{The confidence interval.} \)

\[ \text{snntoolbox.utils.utils.extract_label}(\text{label}) \]
Get the layer number, name and shape from a string.

Parameters \( \text{label} (\text{str}) \) – Specifies both the layer type, index and shape, e.g. \( '03Conv2D_3x32x32' \).

Returns

- layer_num: The index of the layer in the network.
- name: The type of the layer.
- shape: The shape of the layer
Return type: `tuple[int, str, tuple]`

```python
def in_top_k(predictions, targets, k):
    # Arguments
    predictions: A tensor of shape batch_size x classes and type float32.
    targets: A tensor of shape batch_size and type int32 or int64.
    k: An int, number of top elements to consider.

    # Returns
    A tensor of shape batch_size and type int. output_i is 1 if targets_i is within top-k values of predictions_i
```

```python
def top_k_categorical_accuracy(y_true, y_pred, k=5):
    # Parameters
    • y_true
    • y_pred
    • k
```

```python
def echo(text):
    # python 2 version of print(end='', flush=True).
```

```python
def to_list(x):
    # Normalize a list/tensor to a list.
    # If a tensor is passed, returns a list of size 1 containing the tensor.
```

```python
def apply_modifications(model, custom_objects=None):
    # Parameters
    • model (keras.models.Model)
    • custom_objects (dict)

    # Returns
    • The modified model with changes applied. Does not mutate the original
    • model.
```

```python
def import_configparser():
```

Chapter 2. API Reference
CHAPTER 3

Indices and tables

• genindex
• modindex
• search
S

snntoolbox.bin.run, 21
snntoolbox.bin.utils, 21
snntoolbox.conversion.utils, 35
snntoolbox.datasets.utils, 93
snntoolbox.parsing.model_libs.caffe_input_lib, 26
snntoolbox.parsing.model_libs.keras_input_lib, 23
snntoolbox.parsing.utils, 28
snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow, 53
snntoolbox.simulation.backends.inisim.temporal_pattern, 58
snntoolbox.simulation.backends.inisim.ttfs, 62
snntoolbox.simulation.backends.inisim.ttfs_corrective, 69
snntoolbox.simulation.backends.inisim.ttfs_dyn_thresh, 65
snntoolbox.simulation.backends.megasim.megasim, 73
snntoolbox.simulation.plotting, 46
snntoolbox.simulation.target_simulators.brian2_target_sim, 80
snntoolbox.simulation.target_simulators.INI_temporal_mean_rate_target_sim, 73
snntoolbox.simulation.target_simulators.INI_temporal_pattern_target_sim, 75
snntoolbox.simulation.target_simulators.INI_ttfs_corrective_target_sim, 77
snntoolbox.simulation.target_simulators.INI_ttfs_dyn_thresh_target_sim, 76
snntoolbox.simulation.target_simulators.INI_ttfs_target_sim, 76
snntoolbox.simulation.target_simulators.MegaSim_target_sim, 83
snntoolbox.simulation.target_simulators.pyNN_target_sim, 77

snntoolbox.simulation.utils, 38
snntoolbox.utils.utils, 94
Index
generate_bias_clk() (snntoolbox.simulation.target_simulators.MegaSim_target_sim, SNN method), 93
get_abs_path() (in module snntoolbox.utils.utils), 95
get_activation() (snntoolbox.parsing.model_libs.caffe_input_lib.ModelParser method), 27
get_activation() (snntoolbox.parsing.model_libs.keras_input_lib.ModelParser method), 25
get_activation() (snntoolbox.parsing.utils.AbstractModelParser method), 32
get_activations_batch() (in module snntoolbox.conversion.utils), 37
get_activations_layer() (in module snntoolbox.conversion.utils), 36
get_ann_ops() (in module snntoolbox.simulation.utils), 45
get_avg_rate_from_trains() (snntoolbox.simulation.utils.AbstractSNN method), 44
get_batch_input_shape() (snntoolbox.parsing.utils.AbstractModelParser method), 31
get_batchnorm_parameters() (snntoolbox.parsing.model_libs.caffe_input_lib.ModelParser method), 26
get_batchnorm_parameters() (snntoolbox.parsing.model_libs.keras_input_lib.ModelParser method), 23
get_batchnorm_parameters() (snntoolbox.parsing.utils.AbstractModelParser method), 30
get_clamp_idx() (snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow.SpikeLayer method), 54
get_clamped_relu_from_string() (in module snntoolbox.parsing.utils), 35
get_custom_activation() (in module snntoolbox.parsing.utils), 35
get_custom_activations_dict() (in module snntoolbox.parsing.utils), 35
get_dataset() (in module snntoolbox.datasets.utils), 94
get_fanin() (in module snntoolbox.parsing.utils), 34
get_fanout() (in module snntoolbox.parsing.utils), 34
get_fanout_array() (in module snntoolbox.parsing.utils), 34
get_inbound_layers() (snntoolbox.parsing.model_libs.caffe_input_lib.ModelParser method), 26
get_inbound_layers() (snntoolbox.parsing.model_libs.keras_input_lib.ModelParser method), 23
get_inbound_layers() (snntoolbox.parsing.utils.AbstractModelParser method), 30
get_inbound_layers_with_parameters() (snntoolbox.parsing.utils.AbstractModelParser method), 30
get_inbound_layers_with_params() (in module snntoolbox.parsing.utils), 33
get_inbound_layers_without_params() (in module snntoolbox.parsing.utils), 33
get_inbound_names() (snntoolbox.parsing.utils.AbstractModelParser method), 30
get_input_shape() (snntoolbox.parsing.model_libs.caffe_input_lib.ModelParser method), 26
get_input_shape() (snntoolbox.parsing.model_libs.keras_input_lib.ModelParser method), 24
get_input_shape() (snntoolbox.parsing.utils.AbstractModelParser method), 30
get_isi_from_impulse() (in module snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow), 55
get_layer_idx() (snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow.SpikeLayer method), 54
get_layer_idx() (snntoolbox.simulation.backends.inisim.ttfs.SpikeLayer method), 63
get_layer_idx() (snntoolbox.simulation.backends.inisim.ttfs_corrective.SpikeLayer method), 70
get_layer_idx() (snntoolbox.simulation.backends.inisim.ttfs_dyn_thresh.SpikeLayer method), 66
get_layer_iterable() (snntoolbox.parsing.model_libs.caffe_input_lib.ModelParser method), 26
get_layer_iterable() (snntoolbox.parsing.model_libs.keras_input_lib.ModelParser method), 23
get_layer_iterable() (snntoolbox.parsing.utils.AbstractModelParser method), 29
get_layer_synaptic_operations() (in module snntoolbox.simulation.utils), 45
get_log_keys() (in module snntoolbox.bin.utils), 22
get_name() (snntoolbox.parsing.utils.AbstractModelParser method), 31
get_new_mem() (snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow.SpikeLayer method), 54
get_new_mem() (snntoolbox.simulation.backends.inisim.ttfs.SpikeLayer method), 62
get_new_mem() (snntoolbox.simulation.backends.inisim.ttfs_corrective.SpikeLayer method), 69
get_new_mem() (snntoolbox.simulation.backends.inisim.ttfs_dyn_thresh.SpikeLayer method), 66
get_new_thresh() (snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow.SpikeLayer method), 54
get_outbound_activation() (in module snntoolbox.parsing.utils), 34
get_outbound_layers() (in module snntoolbox.parsing.utils), 34
get_outbound_layers() (snntoolbox.parsing.model_libs.caffe_input_lib.ModelParser method), 27
get_outbound_layers() (snntoolbox.parsing.model_libs.keras_input_lib.ModelParser method), 25
get_outbound_layers() (snntoolbox.parsing.utils.AbstractModelParser method), 32
get_output_shape() (snntoolbox.parsing.model_libs.caffe_input_lib.ModelParser method), 26
get_output_shape() (snntoolbox.parsing.model_libs.keras_input_lib.ModelParser method), 24
get_output_shape() (snntoolbox.parsing.utils.AbstractModelParser method), 31
get_output_spikes_batch() (snntoolbox.simulation.target_simulators.MegaSim_target_sim.SNN method), 92
get_pearson_coefficients() (in module snntoolbox.simulation.plotting), 49
get_percentile() (in module snntoolbox.conversion.utils), 36
get_plot_keys() (in module snntoolbox.bin.utils), 22
get_poisson_frame_batch() (snntoolbox.simulation.target_simulators.INI_temporal_mean_rate_target_sim.SNN method), 75
get_psp() (snntoolbox.simulation.backends.inisim.ttfs_corrective.SpikeLayer method), 69
get_psp() (snntoolbox.simulation.backends.inisim.ttfs_dyn_thresh.SpikeLayer method), 66
get_psp() (snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow.SpikeLayer method), 54
get_samples_from_list() (in module snntoolbox.simulation.utils), 45
get_samples_from_list() (snntoolbox.simulation.target_simulators.MegaSim_target_sim.SNN method), 44
get_spikes() (snntoolbox.simulation.backends.inisim.ttfs_corrective.SpikeLayer method), 69
get_spikes() (snntoolbox.simulation.target_simulators.MegaSim_target_sim.SNN method), 92
get_spiketrains() (snntoolbox.simulation.target_simulators.brian2_target_sim.SNN method), 82
get_spiketrains() (snntoolbox.simulation.target_simulators.INI_temporal_mean_rate_target_sim.SNN method), 75
get_spiketrains() (snntoolbox.simulation.target_simulators.pyNN_target_sim.SNN method), 79
get_spiketrains() (snntoolbox.simulation.target_simulators.brian2_target_sim.SNN method), 82
get_spiketrains() (snntoolbox.simulation.target_simulators.pyNN_target_sim.SNN method), 79
get_spiketrains_input() (snntoolbox.simulation.target_simulators.INI_temporal_mean_rate_target_sim.SNN method), 75
get_spiketrains_input() (snntoolbox.simulation.target_simulators.pyNN_target_sim.SNN method), 79
method), 82
get_spiketrains_output() (snntoolbox.simulation.target_simulators.MegaSim_target_sim.SNN method), 66
get_spiketrains_output() (snntoolbox.simulation.target_simulators.pyNN_target_sim.SNN method), 79
get_spiketrains_output() (snntoolbox.simulation.utils.AbstractSNN method), 42
get_spiking_outbound_layers() (in module snntoolbox.simulation.utils), 45
get_time() (snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow.SpikeConcatenate static method), 55
get_time() (snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow.SpikeFlatten static method), 59
get_time() (snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow.SpikeLayer method), 58
get_time() (snntoolbox.simulation.backends.inisim.temporal_pattern.SpikeConcatenate static method), 59
get_time() (snntoolbox.simulation.backends.inisim.temporal_pattern.SpikeFlatten method), 59
get_time() (snntoolbox.simulation.backends.inisim.temporal_pattern.SpikeLayer method), 62
get_time() (snntoolbox.simulation.backends.inisim.tfs.SpikeConcatenate static method), 63
get_time() (snntoolbox.simulation.backends.inisim.tfs.SpikeFlatten static method), 63
get_time() (snntoolbox.simulation.backends.inisim.tfs.SpikeLayer method), 62
get_time() (snntoolbox.simulation.backends.inisim.tfs_corrective.SpikeConcatenate static method), 70
get_time() (snntoolbox.simulation.backends.inisim.tfs_corrective.SpikeFlatten static method), 70
get_time() (snntoolbox.simulation.backends.inisim.tfs_corrective.SpikeLayer method), 70
get_time() (snntoolbox.simulation.backends.inisim.tfs_dyn_thresh.SpikeConcatenate static method), 67
get_time() (snntoolbox.simulation.backends.inisim.tfs_dyn_thresh.SpikeFlatten static method), 67
get_time() (snntoolbox.simulation.backends.inisim.tfs_dyn_thresh.SpikeLayer method), 66
get_type() (in module snntoolbox.parsing.utils), 34
get_type() (snntoolbox.parsing.model_libs.caffe_input_lib.ModelParser method), 26
get_type() (snntoolbox.parsing.model_libs.keras_input_lib.ModelParser method), 23
get_type() (snntoolbox.parsing.utils.AbstractModelParser method), 30
get_vars_to_record() (snntoolbox.simulation.target_simulators.pyNN_target_sim.SNN method), 79
get_vmem() (snntoolbox.simulation.backends.brian2_target_sim.SNN method), 82
get_vmem() (snntoolbox.simulation.backends.MegaSim_target_sim.SNN method), 92
get_vmem() (snntoolbox.simulation.backends.pyNN_target_sim.SNN method), 79
get_vmem() (snntoolbox.simulation.utils.AbstractSNN method), 42
hard_sigmoid() (in module snntoolbox.utils.utils), 96
has_stride_unity() (in module snntoolbox.parsing.utils), 34
has_weights() (in module snntoolbox.parsing.utils), 33
has_weights() (snntoolbox.parsing.model_libs.caffe_input_lib.ModelParser method), 27
has_weights() (snntoolbox.parsing.model_libs.keras_input_lib.ModelParser method), 30
import_configparser() (in module snntoolbox.utils.utils), 98
import_helpers() (in module snntoolbox.utils.utils), 95
import_script() (in module snntoolbox.utils.utils), 95
import_target_sim() (in module snntoolbox.bin.utils), 22
load() (in module snntoolbox.parsing.model_libs.keras_input_lib), 25
load() (snntoolbox.simulation.target_simulators.brian2_target_sim.SNNrage_pooling (class in snntoolbox.simulation.target_simulators.MegaSim_target_sim), 82
load() (snntoolbox.simulation.target_simulators.INI_temporal_mean_rate_target_sim.SNN method), 74
load() (snntoolbox.simulation.target_simulators.INI_temporal_pattern_target_sim.SNN method), 76
load() (snntoolbox.simulation.target_simulators.INI_ttfs_dyn_thresh_target_sim.SNN method), 77
load() (snntoolbox.simulation.target_simulators.INI_ttfs_target_sim.SNN method), 76
load() (snntoolbox.simulation.target_simulators.MegaSim_target_sim.SNN method), 92
load() (snntoolbox.simulation.target_simulators.pyNN_target_sim.SNN method), 78
load() (snntoolbox.simulation.utils.AbstractSNN method), 42
load_assembly() (snntoolbox.simulation.target_simulators.pyNN_target_sim.SNN method), 80
load_config() (in module snntoolbox.bin.utils), 22
load_npz() (in module snntoolbox.datasets.utils), 94

main() (in module snntoolbox.bin.run), 21
max_value (snntoolbox.utils.utils.ClampedReLU attribute), 97
megadirname (snntoolbox.simulation.target_simulators.MegaSim_target_sim.SNN method), 90
megaschematic (snntoolbox.simulation.target_simulators.MegaSim_target_sim.SNN attribute), 90
Megasim_base (class in snntoolbox.simulation.target_simulators.MegaSim_target_sim), 83
megasim_path (snntoolbox.simulation.target_simulators.MegaSim_target_sim.SNN attribute), 90
megasim_path() (in module snntoolbox.simulation.backends.megasim.megasim), 73
mem_n_b_l_t (snntoolbox.simulation.utils.AbstractSNN attribute), 39
MembReset (snntoolbox.simulation.target_simulators.MegaSim_target_sim.SNN attribute), 84, 86
ModelParser (class in snntoolbox.parsing.model_libs.caffe_input_lib), 26
ModelParser (class in snntoolbox.parsing.model_libs.keras_input_lib), 23
modify_parameter_precision() (in module snntoolbox.parsing.utils), 33
modify_parameter_precision() (in module snntoolbox.parsing.utils), 33
Module_average_pooling (class in snntoolbox.simulation.target_simulators.MegaSim_target_sim), 88
Module_conv (class in snntoolbox.simulation.target_simulators.MegaSim_target_sim), 88
module_flatten (class in snntoolbox.simulation.target_simulators.MegaSim_target_sim), 87
Module_fully_connected (class in snntoolbox.simulation.target_simulators.MegaSim_target_sim), 89
module_input_stimulus (class in snntoolbox.simulation.target_simulators.MegaSim_target_sim), 87
module_string (snntoolbox.simulation.target_simulators.MegaSim_target_sim.Module_conv attribute), 88
module_string (snntoolbox.simulation.target_simulators.MegaSim_target_sim.Module_fully_connected attribute), 87
module_string (snntoolbox.simulation.target_simulators.MegaSim_target_sim.Module_input_stimulus attribute), 87
MyProgressBar (class in snntoolbox.simulation.target_simulators.pyNN_target_sim), 80

N
n_in_ports (snntoolbox.simulation.target_simulators.MegaSim_target_sim.SNN attribute), 83, 86
n_out_ports (snntoolbox.simulation.target_simulators.MegaSim_target_sim.SNN attribute), 83, 86
n_repeat (snntoolbox.simulation.target_simulators.MegaSim_target_sim.SNN attribute), 83, 86
neuron_operations_b_t (snntoolbox.simulation.utils.AbstractSNN attribute), 40
normalize_parameters() (in module snntoolbox.simulation.utils.AbstractSNN), 40
num_classes (snntoolbox.simulation.utils.AbstractSNN attribute), 40
num_neurons (snntoolbox.simulation.utils.AbstractSNN attribute), 40
reset_spikevars() (snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow.SpikeLayer method), 54
reset_spikevars() (snntoolbox.simulation.backends.inisim.tfs.SpikeLayer method), 63
reset_spikevars() (snntoolbox.simulation.backends.inisim.ttfs.SpikeLayer method), 70
reset_spikevars() (snntoolbox.simulation.backends.inisim.ttfs_corrective.SpikeLayer method), 70
reset_spikevars() (snntoolbox.simulation.backends.inisim.ttfs_dyn_thresh.SpikeLayer method), 66
Reset_to_reminder (snntoolbox.simulation.target_simulators.MegaSim_target_sim.Megasim_base attribute), 84, 86
reshape_flattened_spiketrains() (snntoolbox.simulation.utils.AbstractSNN method), 43
restore_snn() (snntoolbox.simulation.utils.AbstractSNN method), 43
run() (snntoolbox.simulation.utils.AbstractSNN method), 42
run_parameter_sweep() (in module snntoolbox.bin.utils), 22
run_pipeline() (in module snntoolbox.bin.utils), 22
save() (snntoolbox.simulation.target_simulators.brian2_target_sim.SNN method), 82
save() (snntoolbox.simulation.target_simulators.INI_temporal_mean_rate_target_sim.SNN method), 74
save() (snntoolbox.simulation.target_simulators.MegaSim_target_sim.SNN method), 91
save() (snntoolbox.simulation.target_simulators.pyNN_target_sim.SNN method), 78
save() (snntoolbox.simulation.utils.AbstractSNN method), 41
save_assembly() (snntoolbox.simulation.target_simulators.pyNN_target_sim.SNN method), 79
save_biases() (snntoolbox.simulation.target_simulators.pyNN_target_sim.SNN method), 80
save_connections() (snntoolbox.simulation.target_simulators.pyNN_target_sim.SNN method), 79
scale_first_layer_parameters() (snntoolbox.simulation.target_simulators.INI_temporal_mean_rate_target_sim.SNN method), 75
scaling_factor (snntoolbox.simulation.target_simulators.MegaSim_target_sim.Megasim_base attribute), 85, 87
set_biases() (in module snntoolbox.simulation.target_simulators.brian2_target_sim), 82
set_connectivity() (snntoolbox.simulation.utils.AbstractSNN method), 43
set_reset_mem() (snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow.SpikeLayer method), 54
set_reset_mem() (snntoolbox.simulation.backends.inisim ttfs.SpikeLayer method), 62
set_reset_mem() (snntoolbox.simulation.backends.inisim.ttfs_dyn_thresh.SpikeLayer method), 66
set_spikerates() (snntoolbox.simulation.target_simulators.INI_temporal_pattern_target_sim.SNN method), 76
set_spiketrain_stats() (snntoolbox.simulation.utils.AbstractSNN method), 43
set_spiketrain_stats_input() (snntoolbox.simulation.target_simulators.brian2_target_sim.SNN method), 82
set_spiketrain_stats_input() (snntoolbox.simulation.target_simulators.INI_temporal_mean_rate_target_sim.SNN method), 75
set_spiketrain_stats_input() (snntoolbox.simulation.target_simulators.MegaSim_target_sim.SNN method), 93
set_spiketrain_stats_input() (snntoolbox.simulation.target_simulators.pyNN_target_sim.SNN method), 80
set_synaptic_operations() (snntoolbox.simulation.target_simulators.INI_temporal_pattern_target_sim.SNN method), 76
set_time() (snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow.SpikeLayer method), 54
softmax_activation() (snntoolbox.simulation.backends.inisim.tfss.SpikeLayer static method), 62
softmax_activation() (snntoolbox.simulation.backends.inisim.tfss_dyn_thresh.SpikeLayer box.simulation.backends.inisim.tfss), 64
spike_call() (in module snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow), 55
spike_call() (in module snntoolbox.simulation.backends.inisim.temporal_pattern), 58
spike_call() (in module snntoolbox.simulation.backends.inisim.ttfs), 63
spike_call() (in module snntoolbox.simulation.backends.inisim.ttfs_corrective), 70
spike_call() (in module snntoolbox.simulation.backends.inisim.ttfs_dyn_thresh), 66
spike_count_histogram() (snntoolbox.simulation.target_simulators.MegaSim_target_sim.SNN static method), 92
SpikeAveragePooling2D (class in snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow), 57
SpikeAveragePooling2D (class in snntoolbox.simulation.backends.inisim.temporal_pattern), 59
SpikeAveragePooling2D (class in snntoolbox.simulation.backends.inisim.ttfs), 61
SpikeAveragePooling2D (class in snntoolbox.simulation.backends.inisim.ttfs_corrective), 63
SpikeAveragePooling2D (class in snntoolbox.simulation.backends.inisim.ttfs_dyn_thresh), 65
SpikeFlatten (class in snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow), 55
SpikeFlatten (class in snntoolbox.simulation.backends.inisim.temporal_pattern), 59
SpikeFlatten (class in snntoolbox.simulation.backends.inisim.tkfs_dyn_thresh), 63
SpikeFlatten (class in snntoolbox.simulation.backends.inisim.ttfs_corrective), 67
SpikeConcatenate (class in snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow), 55
SpikeConcatenate (class in snntoolbox.simulation.backends.inisim.temporal_pattern), 59
SpikeConcatenate (class in snntoolbox.simulation.backends.inisim.tkfs_dyn_thresh), 63
SpikeConcatenate (class in snntoolbox.simulation.backends.inisim.ttfs_corrective), 70
SpikeConcatenate (class in snntoolbox.simulation.backends.inisim.ttfs_dyn_thresh), 67
SpikeConv2D (class in snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow), 56
SpikeConv2D (class in snntoolbox.simulation.backends.inisim.temporal_pattern), 60
SpikeConv2D (class in snntoolbox.simulation.backends.inisim.tkfs_dyn_thresh), 64
SpikeConv2D (class in snntoolbox.simulation.backends.inisim.ttfs_corrective), 68
SpikeConv2D (class in snntoolbox.simulation.backends.inisim.ttfs_dyn_thresh), 66
spikecounts_to_rates() (in module snntoolbox.simulation.utils), 44
SpikeDense (class in snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow), 55
SpikeDense (class in snntoolbox.simulation.backends.inisim.temporal_pattern), 59
SpikeDense (class in snntoolbox.simulation.backends.inisim.tkfs_dyn_thresh), 63
SpikeDense (class in snntoolbox.simulation.backends.inisim.ttfs_corrective), 71
SpikeDense (class in snntoolbox.simulation.backends.inisim.ttfs_dyn_thresh), 67
SpikeLayer (class in snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow), 53
SpikeLayer (class in snntoolbox.simulation.backends.inisim.temporal_pattern), 58
SpikeLayer (class in snntoolbox.simulation.backends.inisim.tkfs_dyn_thresh), 62
SpikeLayer (class in snntoolbox.simulation.backends.inisim.ttfs_corrective), 69
SpikeLayer (class in snntoolbox.simulation.backends.inisim.ttfs_dyn_thresh), 66
Index 117
update_neurons() (snntoolbox.simulation.backends.inisim.tfs_corrective.SpikeLayer method), 69
update_neurons() (snntoolbox.simulation.backends.inisim.tfs_dyn_thresh.SpikeLayer method), 66
update_payload() (snntoolbox.simulation.backends.inisim.temporal_mean_rate_tensorflow.SpikeLayer method), 53
update_setup() (in module snntoolbox.bin.utils), 22
update_spikevars() (snntoolbox.simulation.backends.inisim.temporal_pattern.SpikeLayer method), 58
use_biases (snntoolbox.simulation.target_simulators.MegaSim_target_sim.SNN attribute), 90

V
v_reset (snntoolbox.simulation.target_simulators.brian2_target_sim.SNN attribute), 81

W
w (snntoolbox.simulation.target_simulators.MegaSim_target_sim.Module_conv attribute), 89
w (snntoolbox.simulation.target_simulators.MegaSim_target_sim.Module_fully_connected attribute), 90
wilson_score() (in module snntoolbox.utils.utils), 97

X
x_subsmp (snntoolbox.simulation.target_simulators.MegaSim_target_sim.Megasim_base attribute), 85, 86
Xmin (snntoolbox.simulation.target_simulators.MegaSim_target_sim.Megasim_base attribute), 83, 86
xshift_pos (snntoolbox.simulation.target_simulators.MegaSim_target_sim.Megasim_base attribute), 85, 86
xshift_pre (snntoolbox.simulation.target_simulators.MegaSim_target_sim.Megasim_base attribute), 84, 86

Y
y_subsmp (snntoolbox.simulation.target_simulators.MegaSim_target_sim.Megasim_base attribute), 85, 86
Ymin (snntoolbox.simulation.target_simulators.MegaSim_target_sim.Megasim_base attribute), 84, 86
yshift_pos (snntoolbox.simulation.target_simulators.MegaSim_target_sim.Megasim_base attribute), 85, 86
yshift_pre (snntoolbox.simulation.target_simulators.MegaSim_target_sim.Megasim_base attribute), 85, 86