<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 User Guide</td>
<td>3</td>
</tr>
<tr>
<td>2 API Reference</td>
<td>23</td>
</tr>
<tr>
<td>3 Indices and tables</td>
<td>39</td>
</tr>
<tr>
<td>Python Module Index</td>
<td>41</td>
</tr>
</tbody>
</table>
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. Spiking neural network conversion toolbox

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

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 a 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>Required files</td>
<td>*.h5 (weights and model specs), or: *.h5 (only weights) and *.json (model specs)</td>
<td>*.h5 or *.pkl (weights) *.py (script containing a method that returns the compiled model and an evaluation function)</td>
<td>*.caffemodel (weights) *.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.
- 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 very simple integrate-and-fire neuron. By dispensing with redundant parameters and implementing a highly parallel simulation, the run time is reduced by several orders of magnitude, without compromising accuracy.

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

Additionaly, 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.
<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>I(^1)</td>
<td>All(^3)</td>
<td>All(^3)</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, C(^2)</td>
<td>All(^3)</td>
<td>All(^3)</td>
<td>All(^3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dropout</td>
<td>All</td>
<td>All</td>
<td>All(^4)</td>
<td>All(^4)</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>I</td>
<td>I</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linear activation</td>
<td>All</td>
<td>Repl. by ReLU</td>
<td>All</td>
<td>Repl. by ReLU</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>I(^5)</td>
<td>I(^5)</td>
<td>I(^5)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary activation {-1, 1} or {0, 1}</td>
<td>L</td>
<td>I</td>
<td>I</td>
<td>I</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>I</td>
<td>I</td>
<td>I</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 INIsim. In all other simulators, softmax is replaced by ReLU.
• Bias-relaxation.
• 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)

1.2 Installation

1.2.1 Requirements

First, install Theano or Tensorflow.

Note: The SNN toolbox provides a built-in simulator to run the converted network. This simulator is Keras-based and will use either Theano or Tensorflow 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:

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

1.2.2 Release version

Run `pip install snntoolbox`. This will install the other minimum dependencies (Keras, h5py) on the fly.

1.2.3 Development version (recommended)

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

Note: Do not use `python setup.py install` because easy_install caused the installation to fail on some platforms due to dependency issues.

1.2.4 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.

---

1.2. Installation
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).

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>`.

- **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:

1. npz: Compressed numpy format.
2. jpg: Images in directories corresponding to their class.
3. 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 $4*1e9*8\text{bit}/(fc*fx*fy*32\text{bit}) = 1/n$ 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 sub-
directories 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.

3. 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.preprocesing.
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.

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_idxss_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.

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

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

i_offset: float, optional  Offset current in nA.

cm: float, optional  Membrane capacitance in nF.

tau_m: float, optional  Membrane time constant in milliseconds.

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

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

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

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

```yaml
[paths]
path_wd =
dataset_path =
log_dir_of_current_run =
runlabel = test
filename_ann =
filename_parsed_model =
filename_snn =
filename_clamp_indices =
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
normalize = True
convert = True
simulate = True
```

(continues on next page)
[normalization]
percentile = 99.9
normalization_schedule = False
online_normalization = False
diff_to_max_rate = 200
diff_to_min_rate = 100
timestep_fraction = 10

[conversion]
softmax_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 = 100
num_to_test = 1000
sample_idxs_to_test = []
reset_between_nth_sample = 1
top_k = 1
keras_backend = theano
early_stopping = False

cell
v_thresh = 1
tau_refrac = 0
v_reset = 0
v_rest = 0
e_rev_E = 10
e_rev_I = -10
i_offset = 0
cm = 0.09
tau_m = 1000
tau_syn_E = 0.01
tau_syn_I = 0.01
delay = 1
binarize_weights = False
quantize_weights = False
scaling_factor = 10000000
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'}
simulators = %(simulators_pyNN)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 can be spiking:
spiking_layers = {'Dense', 'Conv2D', 'MaxPooling2D', 'AveragePooling2D'}
# Layers that can be implemented by our spiking neuron simulators:
snn_layers = %(spiking_layers)s | {'Flatten', 'Concatenate'}
pyNN_keys = {'v_reset', 'v_rest', 'e_rev_E', 'e_rev_I', 'i_offset', 'cm',
             'tau_m', 'tau_syn_E', 'tau_syn_I', 'delay'}
log_vars = {'activations_n_b_l', 'spiketrains_n_b_l_t', 'input_b_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 tool-
box, put a module named `torch_input_lib` into the `snntoolbox.parsing.model_libs` package. Then create a child class `lasagne_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`. Here we show a number of examples of such configuration files.

1. LeNet on MNIST
2. BinaryConnect on CIFAR-10
3. BinaryNet on CIFAR-10
4. VGG-16 on ImageNet
5. Inception-V3 on ImageNet

They are stored in the `examples` subdirectory of the repository root, and should also be included in the installation. They contain the network architectures, parameters, the configuration files for running the toolbox, as well as a subset of the data sets. You can simply run them by typing `snntoolbox -t <path_to_config>` in the terminal, where `<path_to_config>` is the path (including filename) to the configuration file corresponding to that particular experiment.

**Note:** In order to successfully run the example models here, make sure the Keras configuration file contains the option "image_data_format": "channels_first".

### 1.6.1 Example A - LeNet on MNIST

Here we test the classic LeNet architecture on MNIST, using one implementation in Keras, one in Lasagne, and one in Caffe.

**Keras**

```bash
[paths]
dataset_path = %(path_wd)s/../../../datasets/mnist
filename_ann = 98.96

[tools]
evaluate_ann = True
```

(continues on next page)
The most important part to specify in the config file is the filename_ann and dataset_path parameter in the paths section. This will tell the toolbox where to look for the input model and data.

It is possible to leave some settings like path_wd open; they will be filled in with the default values. For instance, path_wd is set to the directory where the config file lives.

Note how we can specify relative paths, or even refer to other options (like path_wd) when building other paths. If only a filename (without path) is given, or the path is relative, it is always assumed to be relative to the path_wd option.

In the simulation section we tell the toolbox how long and how many samples to test.

Optionally, we can ask the toolbox to output plots and save some quantities that were monitored during the simulation to disk.

Note: The model 98.96.h5 in the example above was trained using Keras version <= 2.1.6. If you installed the toolbox using a newer Keras version, this model may show a drop in accuracy because of a change in the Flatten layer. Downgrade Keras to maintain accuracy, or set filename_ann = 99.14 to use a model trained with Keras 2.2.4.

Note: If you plan on simulating the converted model on a pyNN simulator (Nest, Brian, ...), please have a look at github issue #25.

### Caffe

You need to have Caffe installed to run this example.

Here we need to change the model_lib option from default keras to caffe. The filename_ann also changes, the rest can stay the same.

```bash
[paths]
dataset_path = %(path_wd)s/../../../datasets/mnist
dataset_path = 96.13

[input]
model_lib = caffe

[simulation]
duration = 30
```
num_to_test = 100
batch_size = 10

Lasagne

You need to have Lasagne installed to run this example.

We can also use Poisson input as shown below:

```
[paths]
dataset_path = %(path_wd)s/../../../datasets/mnist
filename_ann = 99.02

[tools]
evaluate_ann = True

[input]
model_lib = lasagne
poisson_input = True
input_rate = 1000

[simulation]
duration = 30
num_to_test = 100
batch_size = 10
```

1.6.2 Example B - BinaryConnect on CIFAR-10

You need to have Lasagne installed to run this example.

The network has been trained with binary weights and can be tested using either full-precision or binary weights. Set the binarize_weights option accordingly.

```
[paths]
dataset_path = %(path_wd)s/../../datasets/cifar10/binaryconnect
filename_ann = 91.91

[input]
model_lib = lasagne

[tools]
normalize = True

[normalization]
percentile = 99.99

[simulation]
duration = 100
num_to_test = 20
batch_size = 10

[cell]
binarize_weights = False
```
### 1.6.3 Example C - BinaryNet on CIFAR-10

You need to have Lasagne installed to run this example.

This network is like BinaryConnect, but in addition to binary weights also uses binary activations.

Note how we turn off `normalize` in `tools`. Parameter normalization is not required here because activations never exceed threshold anyways.

```yaml
[paths]
dataset_path = %(path_wd)s/../../datasets/cifar10/binarynet
filename_ann = 88.22

[input]
model_lib = lasagne

[tools]
evaluate_ann = True
normalize = False

[simulation]
duration = 30
num_to_test = 20
batch_size = 10

[cell]
binarize_weights = True
reset = Reset to zero
```

### 1.6.4 Example D - VGG-16 on ImageNet

This example shows how to use `jpg` images as `dataset_format`. Options that are needed for this include `class_idx_path`, `datagen_kwargs`, and `dataflow_kwargs`. Please refer to `settings` for details.

Further, we use a membrane clamp to reduce transient neuron dynamics in the beginning of the simulation. For this, we set the `filename_clamp_indices` option.

For memory reasons we do not include the model file here, but the example should work when instantiating VGG-16 from the Keras model zoo.

```yaml
[paths]
dataset_path = %(path_wd)s/../../datasets/imagenet/validation
filename_ann = 64.27_85.59
class_idx_path = %(dataset_path)s/../imagenet_class_index_1000.json

[input]
dataset_format = jpg
datagen_kwargs = {'preprocessing_function': 'helper_functions'}
dataflow_kwargs = {'target_size': (224, 224), 'shuffle': True}

[simulation]
duration = 400
batch_size = 1
num_to_test = 2

[normalization]
percentile = 100
```
1.6.5 Example E - Inception-V3 on ImageNet

If we want the Keras.ImageDataGenerator to perform some kind of preprocessing on the data, we need to pass a `preprocessing_function` to the `datagen_kwargs`. This can be done by simply giving the name of a python module containing the preprocessing function. The toolbox will import and use the function from there.

```json
[paths]
dataset_path = %(path_wd)s/../../datasets/imagenet/validation
dataset_path = 76.28_93.03
filename_clamp_indices = clamp_idx_10.json
class_idx_path = %(dataset_path)s/../imagenet_class_index_1000.json

[tools]
evaluate_ann = True

[input]
datagen_kwargs = {'preprocessing_function': 'helper_functions'}
dataflow_kwargs = {'target_size': (299, 299), 'shuffle': True}

[simulation]
duration = 600
batch_size = 1
num_to_test = 2

[normalization]
percentile = 100
```

1.7 Citation

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

Rueckauer, B. and Hu, Y. and Lungu, I.A. and Pfeiffer, M. and Liu, S.-C.
Conversion of continuous-valued deep networks to efficient event-driven networks for image classification,

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()
```

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:

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>test_full</code></td>
<td>Convert an analog network to a spiking network and simulate it.</td>
</tr>
</tbody>
</table>

Continued on next page
update_setup

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

@author: rbodo

**snntoolbox.bin.utils.test_full**(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 \texttt{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

**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

**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.

**snntoolbox.bin.utils.import_target_sim**(config)

**snntoolbox.bin.utils.load_config**(filepath)

Load a config file from filepath.

**snntoolbox.bin.utils.update_setup**(config_filepath)

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.

**snntoolbox.bin.utils.initialize_simulator**(config)

Import a module that contains utility functions of spiking simulator.

**snntoolbox.bin.utils.get_log_keys**(config)

**snntoolbox.bin.utils.get_plot_keys**(config)
2.1.3 snntoolbox.bin.gui

Graphical User interface for the SNN conversion toolbox.

Features

- Allows setting parameters and what tools to use during an experiment.
- Performs basic checks that specified parameters are valid.
- Preferences can be saved and reloaded.
- Tooltips explain the functionality.
- Automatically recognizes result plots and allows displaying them in a separate window.

Note: Due to rapid extensions in the main toolbox, we have not always been able to update the GUI to cover all functionality of the toolbox. We are currently not maintaining the GUI and recommend using the terminal to run experiments.

@author: rbodo

class snntoolbox.bin.gui.SNNToolboxGUI(root, config)
    Bases: object
    define_style()
        Define apperance style.
    initialize_thread()
        Separate thread for conversion process.
    globalparams_widgets()
        Global parameters widgets.
    cellparams_widgets()
        Create a container for individual parameter widgets.
    simparams_widgets()
        Create a container for individual parameter widgets.
    tools_widgets()
        Create tools widgets.
    edit_normalization_settings()
        Settings menu for parameter normalization
    edit_experimental_settings()
        Settings menu for experimental features.
    edit_dataset_settings()
        Settings menu for dataset parameters.
    graph_widgets()
        Create graph widgets.
select_plots_dir_rb()  
Select plots directory.

select_layer_rb()  
Select layer.

draw_canvas()  
Draw canvas figure.

close_window()  
Close window function.

display_graphs()  
Display graphs.

top_level_menu()  
Top level menu settings.

static documentation()  
Open documentation.

static about()  
About message.

quit_toolbox()  
Quit toolbox function.

declare_parameter_vars()  
Preference collection.

restore_default_params()  
Restore default parameters.

set_preferences(p)  
Set preferences.

save_settings()  
Save current settings.

load_settings(s=None)  
Load a preferences settings.

start_processing()  
Start processing.

stop_processing()  
Stop processing.

update()  
Update GUI with items from the queue.

check_sample(p)  
Check samples.

check_file(p)  
Check files.

check_path(p)  
Check path.

check_runlabel(p)  
Check runlabel.

check_dataset_path(p)
Parameters p –

- **set_cwd()**: Set current working directory.
- **set_dataset_path()**: Set path to dataset.
- **toggle_state_pynn(val)**: Toggle state for pyNN.
- **toggle_start_state(val)**: Toggle start state.
- **toggle_stop_state(val)**: Toggle stop state.
- **toggle_num_to_test_state(val)**: Toggle number to test state.
- **toggle_poisson_input_state()**: Toggle poisson input.

---

## 2.2 snntoolbox.parsing

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

```python
snntoolbox.parsing.model_libs.keras_input_lib
snntoolbox.parsing.model_libs.lasagne_input_lib
snntoolbox.parsing.model_libs.caffe_input_lib
```

These parsers inherit from `AbstractModelParser` in

```python
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
lasagne_input_lib
caffe_input_lib

2.2.2 snntoolbox.parsing.utils

2.3 snntoolbox.conversion

2.3.1 snntoolbox.conversion.utils

2.4 snntoolbox.simulation

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

- snntoolbox.simulation.
  target_simulators.pyNN_target_sim
- snntoolbox.simulation.
  target_simulators.brian2_target_sim
- snntoolbox.simulation.
  target_simulators.MegaSim_target_sim
- snntoolbox.simulation.
  target_simulators.INI_temporal_mean_rate_target_sim
- snntoolbox.simulation.
  target_simulators.INI_temporal_pattern_target_sim
- snntoolbox.simulation.
  target_simulators.INI_ttfs_target_sim
- snntoolbox.simulation.
  target_simulators.INI_ttfs_dyn_thresh_target_sim
- snntoolbox.simulation.
  target_simulators.INI_ttfs_corrective_target_sim

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

- snntoolbox.simulation.utils

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:

- snntoolbox.simulation.backends.
  inisim.temporal_mean_rate_tensorflow

Continued on next page
Finally, utility functions for plotting are contained in

2.4.1 snntoolbox.simulation.utils

2.4.2 snntoolbox.simulation.plotting

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

**snntoolbox.simulation.plotting.output_graphs**

Wrapper function to display / save a number of plots.

Parameters

- **plot_vars (dict)** – Example items:
  - spiketrains_n_b_l_t: list[tuple[np.array, str]] Each entry in spiketrains_batch contains a tuple (spiketimes, label) for each layer of the network (for the first batch only, and excluding Flatten layers). spiketimes is an array where the last index contains the spike times of the specific neuron, and the first indices run over the number of neurons in the layer: (batch_size, n_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.
  - spikecounts_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**

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_chans, 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 spikerates 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’.

```python
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’.

```python
snntoolbox.simulation.plotting.plot_layer_correlation(rates, activations, title, config, path=None)
```

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, flattened 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.

```python
snntoolbox.simulation.plotting.plot_correlations(spikerates, layer_activations)
```

Plot the correlation between SNN spiketrains and ANN activations.

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

**Parameters**

• **spikerates** (list of tuples (spikerate, label).) – spikerate is a 1D array containing the mean firing rates of the neurons in a specific layer.
  
  label is a string specifying both the layer type and the index, e.g. '3Dense'.

• **layer_activations** (list of tuples (activations, label)) – Each entry represents a layer in the ANN for which an activation can be calculated (e.g. Dense, 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'.

```
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`

**Return type**

`list`

```
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[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 the layer type, e.g. 'Dense'.

  - `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.

```python
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.

```python
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.

```python
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.

```python
snntoolbox.simulation.plotting.plot_spiketrains(layer, dt, path=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.

```python
snntoolbox.simulation.plotting.plot_potential(times, layer, config, 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.
- `show_legend` (bool) – If True, shows the legend indicating the neuron indices and lines like v_thresh, v_rest, v_reset. Recommended only for layers with few neurons.
- `path` (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)
```

Plot classification error over time.

**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.
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.

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.

snntoolbox.simulation.plotting.plot_input_image(x, label, path=None, data_format=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’.

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.
2.4.3 `snntoolbox.simulation.backends`

inisim

temporal_mean_rate_tensorflow

temporal_mean_rate_theano

temporal_pattern

ttfs

ttfs_dyn_thresh

ttfs_corrective

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_temporal_pattern_target_sim
INI_ttfs_target_sim
INI_ttfs_dyn_thresh_target_sim
INI_ttfs_corrective_target_sim
pyNN_target_sim
brian2_target_sim
MegaSim_target_sim

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

```python
@author: rbodo

snntoolbox.datasets.utils.get_dataset(config)
Get data set, 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 kerasImageDataGenerator.

Parameters config (configparser.ConfigParser) – Settings.

Returns

• normset (dict) – Used to normalized the network parameters.
• testset (dict) – Used to test the networks.
```

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]

`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).

`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
CHAPTER 3

Indices and tables

• genindex
• modindex
• search
S

snntoolbox.bin.gui.gui, 25
snntoolbox.bin.run, 23
snntoolbox.bin.utils, 23
snntoolbox.datasets.utils, 37
snntoolbox.simulation.backends.megasim.megasim, 36
snntoolbox.simulation.plotting, 29
Index

A
about() (snntoolbox.bin.gui.gui.SNNToolboxGUI static method), 26

cellparams_widgets() (snntoolbox.bin.gui.gui.SNNToolboxGUI method), 25

ccheck_dataset_path() (snntoolbox.bin.gui.gui.SNNToolboxGUI method), 26

ccheck_file() (snntoolbox.bin.gui.gui.SNNToolboxGUI method), 26

ccheck_path() (snntoolbox.bin.gui.gui.SNNToolboxGUI method), 26

ccheck_runlabel() (snntoolbox.bin.gui.gui.SNNToolboxGUI method), 26

ccheck_sample() (snntoolbox.bin.gui.gui.SNNToolboxGUI method), 26

close_window() (snntoolbox.bin.gui.gui.SNNToolboxGUI method), 26

cconfig_string_to_set_of_strings() (in module snntoolbox.bin.utils), 24

draw_canvas() (snntoolbox.bin.gui.gui.SNNToolboxGUI method), 26

E
edit_dataset_settings() (snntoolbox.bin.gui.gui.SNNToolboxGUI method), 25
edit_experimental_settings() (snntoolbox.bin.gui.gui.SNNToolboxGUI method), 25
edit_normalization_settings() (snntoolbox.bin.gui.gui.SNNToolboxGUI method), 25

gget_dataset() (in module snntoolbox.datasets.utils), 37
gget_log_keys() (in module snntoolbox.bin.utils), 24
gget_pearson_coefficients() (in module snntoolbox.simulation.plotting), 32
gget_plot_keys() (in module snntoolbox.bin.utils), 24
gglobalparams_widgets() (snntoolbox.bin.gui.gui.SNNToolboxGUI method), 25
ggraph_widgets() (snntoolbox.bin.gui.gui.SNNToolboxGUI method), 25

I
import_target_sim() (in module snntoolbox.bin.utils), 24
initialize_simulator() (in module snntoolbox.bin.utils), 24
initialize_thread() (snntoolbox.bin.gui.gui.SNNToolboxGUI method), 25
isis_stop() (in module snntoolbox.bin.utils), 24

L
load_config() (in module snntoolbox.bin.utils), 24
load_npz() (in module snntoolbox.datasets.utils), 38
load_settings() (snntoolbox.bin.gui.gui.SNNToolboxGUI method), 26
Index
toggle_start_state() (snntoolbox.bin.gui.gui.SNNToolboxGUI method), 27

toggle_state_pynn() (snntoolbox.bin.gui.gui.SNNToolboxGUI method), 27

toggle_stop_state() (snntoolbox.bin.gui.gui.SNNToolboxGUI method), 27

tools_widgets() (snntoolbox.bin.gui.gui.SNNToolboxGUI method), 25

top_level_menu() (snntoolbox.bin.gui.gui.SNNToolboxGUI method), 26

try_get_normset_from_scalefacs() (in module snntoolbox.datasets.utils), 37

U

update() (snntoolbox.bin.gui.gui.SNNToolboxGUI method), 26

update_setup() (in module snntoolbox.bin.utils), 24

Index 45