YASS Documentation

Release 0.9

Grossman Center at Columbia University

May 31, 2018

Contents

Note: YASS is in an early stage of development. Although it is stable, it has only been tested with the data in our lab, but we are working to make it more flexible. Feel free to send feedback through Gitter. Expect a lot of API changes in the near future.

CHAPTER 1

Reference

Lee, J. et al. (2017). YASS: Yet another spike sorter. Neural Information Processing Systems. Available in biorxiv: https://www.biorxiv.org/content/early/2017/06/19/151928

CHAPTER 2

Installation

Installing the last stable version:

```
pip install yass-algorithm
```

If you are feeling adventurous, you can install from the master branch:

```
pip install git+git://github.com/paninski-lab/yass@master
```

CHAPTER 3

Example

Quick example of YASS using a sample of the neuropixel data from Nick Steinmetz:

```
# install last stable version
pip install yass-algorithm

# clone repo to get the sample data
git clone https://github.com/paninski-lab/yass

# move to the examples/ folder and run yass in the sample data
cd yass/examples
yass sort config_sample.yaml

# see the spike train
cat data/spike_train.csv
```

You can also use YASS in Python scripts. See the documentation for details.

CHAPTER 4

Documentation

Documentation hosted at https://yass.readthedocs.io

CHAPTER 5

Running tests

To run tests and flake8 checks (from the root folder):

```
pip install -r requirements.txt

make test
```

CHAPTER 6

Building documentation

You need to install graphviz to build the graphs included in the documentation. On macOS:

```
brew install graphviz
```

To build the docs (from the root folder):

```
pip install -r requirements.txt

make docs
```

CHAPTER 7

Contributors

Peter Lee, Eduardo Blancas, Nishchal Dethe, Shenghao Wu, Hooshmand Shokri, Calvin Tong, Catalin Mitelut

CHAPTER 8

Contents

8.1 Getting started

8.1.1 Using YASS pre-built pipelines

YASS configuration file

YASS is configured using a YAML file, below is an example of such configuration:

```
######################################################################
# YASS configuration example (only required values)                  #
# for a complete reference see examples/config_sample_complete.yaml #
######################################################################

data:
  root_folder: data/
  recordings: neuropixel.bin
  geometry: neuropixel_channels.npy

resources:
  max_memory: 200MB

recordings:
  dtype: int16
  sampling_rate: 30000
  n_channels: 10
  spatial_radius: 70
  spike_size_ms: 1
  order: samples

preprocess:
  apply_filter: True
  dtype: float32
```

(continues on next page)

(continued from previous page)

```
detect:
  method: threshold
  temporal_features: 3
```

If you want to use a Neural Network as detector, you need to provide your own Neural Network. YASS provides tools for easily training the model, see this tutorial for details.

If you do now want to use a Neural Network, you can use the *threshold* detector instead.

For details regarding the configuration file see *YASS configuration file*.

Running YASS from the command line

After installing *yass*, you can sort spikes from the command line:

```
yass sort path/to/config.yaml
```

Run the following command for more information:

```
yass sort --help
```

Running YASS in a Python script

```
import logging

import yass
from yass import preprocess
from yass import detect
from yass import cluster
from yass import templates
from yass import deconvolute

# configure logging module to get useful information
logging.basicConfig(level=logging.INFO)

# set yass configuration parameters
yass.set_config('config_sample.yaml')

(standarized_path, standarized_params, channel_index,
 whiten_filter) = preprocess.run()

(score, spike_index_clear,
 spike_index_all) = detect.run(standarized_path,
                               standarized_params,
                               channel_index,
                               whiten_filter)


spike_train_clear, tmp_loc, vbParam = cluster.run(
    score, spike_index_clear)

(templates_, spike_train,
 groups, idx_good_templates) = templates.run(
```

(continues on next page)

(continued from previous page)

```
    spike_train_clear, tmp_loc)

spike_train = deconvolute.run(spike_index_all, templates_)
```

8.2 YASS configuration file

```
########################################################
# YASS configuration example (all sections and values) #
########################################################

data:
  # project's root folder, data will be loaded and saved here
  # can be an absolute or relative path
  root_folder: data/
  # recordings filename (must be a binary file), details about the recordings
  # are specified in the recordings section
  recordings: neuropixel.bin
  # channel geometry filename , supports txt (one x, y pair per line,
  # separated by spaces) or a npy file with shape (n_channels, 2),
  # where every row contains a x, y pair. see yass.geometry.parse for details
  geometry: neuropixel_channels.npy

resources:
  # maximum memory per batch allowed (only relevant for preprocess
  # and detection step, which perform batch processing)
  max_memory: 200MB
  # maximum memory per batch allowed (only relevant for detection step
  # which uses tensorflow GPU is available)
  max_memory_gpu: 1GB
  # number of processes to use for operations that support parallel execution,
  # 'max' will use all cores, if you as an int, it will use that many cores
  processes: max

recordings:
  # precision of the recording - must be a valid numpy dtype
  dtype: int16
  # recording rate (in Hz)
  sampling_rate: 30000
  # number of channels
  n_channels: 10
  # channels spatial radius to consider them neighbors, see
  # yass.geometry.find_channel_neighbors for details
  spatial_radius: 70
  # temporal length of waveforms in ms
  spike_size_ms: 1
  # recordings order, one of ('channels', 'samples'). In a dataset with k
  # observations per channel and j channels: 'channels' means first k
  # contiguous observations come from channel 0, then channel 1, and so on.
  # 'sample' means first j contiguous data are the first observations from
  # all channels, then the second observations from all channels and so on
  order: samples

preprocess:
  # One of 'overwrite', 'abort', 'skip'. Control de behavior for every
```

(continues on next page)

(continued from previous page)

```
  # generated file. If 'overwrite' it replaces the files if any exist,
  # if 'abort' it raises a ValueError exception if any file exists,
  # if 'skip' it skips the operation (and loads the files) if they exist
  if_file_exists: skip
  # apply butterworth filter in the preprocessing step
  apply_filter: True
  # output dtype for transformed data
  dtype: float32
  # filter configuration
  filter:
    # Order of Butterworth filter
    order: 3
    # Low pass frequency (Hz)
    low_pass_freq: 300
    # High pass factor (proportion of sampling rate)
    high_factor: 0.1

detect:
  # similar to preprocess.if_file_exists
  if_file_exists: skip
  # whether to save results from this step to disk
  save_results: False
  # 'nn' for neural net detction, 'threshold' for amplitude threshold detection
  method: threshold
  # number of features in the temporal dimension to use when applying
  # dimensionality reduction
  temporal_features: 3
  # Configuration parameters when when detect.method = 'nn'
  neural_network_detector:
    # model name, can be any of the models included in yass (detectnet1.ckpt),
    # a relative folder to data.root_fodler (e.g.
    # $ROOT_FOLDER/models/mymodel.ckpt) or an absolute path to a model
    # (e.g. /path/to/my/model.ckpt). In the same folder as your model, there
    # must be a yaml file with the number and size of the filters, the file
    # should be named exactly as your model but with yaml extension
    # see yass/src/assets/models/ for an example
    filename: detect_nn1.ckpt
    # Threshold for spike event detection
    threshold_spike: 0.5
  neural_network_triage:
    # same rules apply as in neural_network_detector.filename but the
    # yaml file should only contain size (not number)
    filename: triage_nn1.ckpt
    # Threshold for clear/collision detection
    threshold_collision: 0.5
  neural_network_autoencoder:
    # same rules apply as in neural_network_detector.filename but no
    # yaml file is needed
    filename: ae_nn1.ckpt
  # Configuration parameters when when detect.method = 'threshold'
  threshold_detector:
    std_factor: 4


# All values are optional
cluster:
  # similar to preprocess.if_file_exists
```

(continues on next page)

(continued from previous page)

```
  if_file_exists: skip
  # similar to detect.save_results
  save_results: False
  # Masking threshold
  masking_threshold: [0.9, 0.5]
  # Num. of new clusters in split
  n_split: 5
  # Choose 'location' for location (x and y : 2 features) + main channel
  # features (n_feature dimensional) as the feature space. Calculates the location
  # of the events using a weighted average of the power in the main_channel
  # and neighboring channels.
  # Choose 'neigh_chan' for n_feature x neighboring_channels dimensional feature
  # space. The feature space is defined by feature summarization of the waveforms
  # into n_feature dimensional feature space for only the main_channel and the
  # neighboring channels (This key (clustering.clustering_method) is not optional)
  method: location
  # maximum number of spikes per clustering group
  # if the total number of spikes per clustering group exceeds it,
  # it randomly subsample
  max_n_spikes: 10000
  # minimum number of spikes per cluster
  # if the total number of spikes per cluster is less than this,
  # the cluster is killed
  min_spikes: 0
  # cluster prior information
  prior:
    beta: 1
    a: 1
    lambda0: 0.01
    nu: 5
    V: 2
  # FIXME: docs, seems like this section only applies when cluster.method
  # != location
  triage:
    # number of nearest neighbors to consider
    nearest_neighbors: 20
    # percentage of data to be triaged
    percent: 0.1

  coreset:
    # number of clusters
    clusters: 10
    # distance threshold
    threshold: 0.95

templates:
  # similar to preprocess.if_file_exists
  if_file_exists: skip
  # similar to detect.save_results
  save_results: False
  # how much shift to allow for template alignment
  max_shift: 3
  merge_threshold: [0.8, 0.7]

deconvolution:
  # refractory period violation in time bins
  n_rf: 1.5
```

(continues on next page)

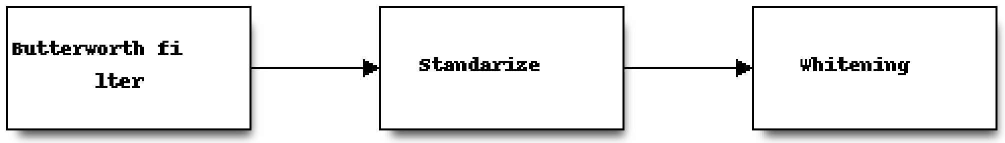
(continued from previous page)

```
# threshold on template scale
threshold_a: 0.3
# threshold on decrease in L2 difference
threshold_dd: 0
# size of windows to look consider around spike time for deconv.
n_explore: 2
# upsampling factor of templates
upsample_factor: 5
```

8.3 Using pre-built pipeline

YASS provides with a pre-built pipeline for spike sorting, which consists of five parts: preprocess, detect, cluster, make templates and deconvolute.

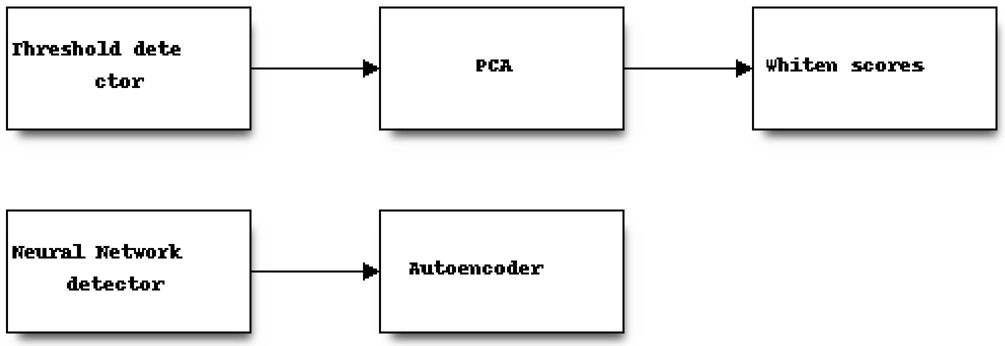
8.3.1 Preprocess



Name	Description
Butterworth filter	Apply filtering to the n_observations x n_channels data matrix (optional)
Standarize	Standarize data matrix
Whitening	Compute whitening filter

See *Preprocess* for details.

8.3.2 Detect



Name	Description
Threshold detector	Detect spikes using a threshold
PCA	Dimensionality reduction using PCA
Whiten scores	Apply whitening to PCA scores
Neural Network detector	Detect spikes using a Neural Network
Autoencoder	Dimensionality reduction using an autoencoder

See *Detect* for details.

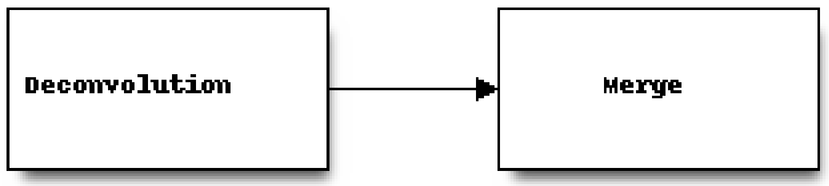
8.3.3 Cluster

See *Cluster* for details.

8.3.4 Templates

See *Templates* for details.

8.3.5 Deconvolve



Name	Description
Deconvolution	Deconvolute unclear spikes using the templates
Merge	Merge all spikes to produce the final ouput

See *Deconvolute* for details.

8.4 API Reference

8.4.1 Preprocess

8.4.2 Detect

8.4.3 Cluster

Clustering spikes

`yass.cluster.run(*args, **kwargs)`

Spike clustering

Parameters

scores: numpy.ndarray (n_spikes, n_features, n_channels), str or Path 3D array with the scores for the clear spikes, first simension is the number of spikes, second is the nymber of features and third the number of channels. Or path to a npy file

spike_index: numpy.ndarray (n_clear_spikes, 2), str or Path 2D array with indexes for spikes, first column contains the spike location in the recording and the second the main channel (channel whose amplitude is maximum). Or path to an npy file

output_directory: str, optional Location to store/look for the generate spike train, relative to CONFIG.data.root_folder

if_file_exists: str, optional One of 'overwrite', 'abort', 'skip'. Control de behavior for the spike_train_cluster.npy. file If 'overwrite' it replaces the files if exists, if 'abort' it raises a ValueError exception if exists, if 'skip' it skips the operation if the file exists (and returns the stored file)

save_results: bool, optional Whether to save spike train to disk (in CONFIG.data.root_folder/relative_to/spike_train_cluster.npy), defaults to False

Returns

spike_train: (TODO add documentation)

Examples

```
import logging

import yass
from yass import preprocess
from yass import detect
from yass import cluster

```

(continues on next page)

(continued from previous page)

```
# configure logging module to get useful information
logging.basicConfig(level=logging.INFO)

# set yass configuration parameters
yass.set_config('config_sample.yaml')

(standarized_path, standarized_params, channel_index,
 whiten_filter) = preprocess.run()

(score, spike_index_clear,
 spike_index_all) = detect.run(standarized_path,
                               standarized_params,
                               channel_index,
                               whiten_filter)


spike_train_clear, tmp_loc, vbParam = cluster.run(
    score, spike_index_clear)
```

8.4.4 Templates

8.4.5 Deconvolute

8.4.6 Geometry

Functions for parsing geometry data

`yass.geometry.`**`find_channel_neighbors`**(*geom*, *radius*)

Compute a channel neighbors matrix

Parameters

geom: np.array Array with the cartesian coordinates for the channels

radius: float Maximum radius for the channels to be considered neighbors

Returns

numpy.ndarray (n_channels, n_channels) Symmetric boolean matrix with the i, j as True if the ith and jth channels are considered neighbors

`yass.geometry.`**`make_channel_groups`**(*n_channels*, *neighbors*, *geom*)

[DESCRIPTION]

Parameters

n_channels: int Number of channels

neighbors: numpy.ndarray Neighbors matrix

geom: numpy.ndarray geometry matrix

Returns

list List of channel groups based on [?]

`yass.geometry.`**`n_steps_neigh_channels`**(*neighbors*, *steps*)

Compute a neighbors matrix by considering neighbors of neighbors

Parameters

neighbors: numpy.ndarray Neighbors matrix

steps: int Number of steps to still consider channels as neighbors

Returns

numpy.ndarray (n_channels, n_channels) Symmetric boolean matrix with the i, j as True if the ith and jth channels are considered neighbors

`yass.geometry.`**`order_channels_by_distance`**(*reference*, *channels*, *geom*)
Order channels by distance using certain channel as reference

Parameters

reference: int Reference channel

channels: np.ndarray Channels to order

geom Geometry matrix

Returns

numpy.ndarray 1D array with the channels ordered by distance using the reference channels

numpy.ndarray 1D array with the indexes for the ordered channels

`yass.geometry.`**`ordered_neighbors`**(*geom*, *neighbors*)
Compute a list of arrays whose ith element contains the ordered (by distance) neighbors for the ith channel

Parameters

geom: numpy.ndarray geometry matrix

neighbors: numpy.ndarray Neighbors matrix

`yass.geometry.`**`parse`**(*path*, *n_channels*)
Parse a geometry txt (one x, y pair per line, separated by spaces) or a npy file with shape (n_channels, 2), where every row contains a x, y pair

path: str Path to geometry file

n_channels: int Number of channels

Returns

numpy.ndarray 2-dimensional numpy array where each row contains the x, y coordinates for a channel

Examples

```
```

from yass import geometry

geom = geometry.parse('path/to/geom.npy', n_channels=500) geom = geometry.parse('path/to/geom.txt', n_channels=500)

8.4.7 Batch Processing

Batch Processor

Batch Pipeline

Recordings Reader

Binary Readers

8.5 Developer's Guide

8.5.1 Using miniconda

The easiest way to work with Python is through miniconda, which helps you create virtual environments isolated of each other and local to your UNIX user. This way you can switch between Python and packages versions.

Installing conda

Download the appropriate installer from here.

Example using 64-bit Linux:

```
# dowload installer
curl https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh -o␣
↪miniconda.sh

# run it
bash miniconda.sh

# follow instructions...
```

Using conda

Create a new environment for your project with this command:

```
conda create --name=project
```

You can specify a Python version:

```
conda create --name=project python=3.5
```

Activate your environment:

```
source activate project
```

Install packages in that environment:

```
pip install numpy
```

Deactivate environment:

```
source deactivate
```

Other resources

- miniconda cheat sheet

8.5.2 Contributing to YASS

Git workflow

Internal contributors have write permissins to the repo, you can create new branches, do your work and submit pull requests:

```
# move to the repo
cd path/to/repo

# when you start working on something new, create a new branch from master
git checkout -b new-feature

# work on new feature and remember to keep in sync with the master branch
# from time to time
git merge master

# remember to push you changes to the remote branch
git push

# when the new feature is done open a pull request to merge new-feature to master

# once the pull request is accepted and merged to master, don't forget to remove
# the branch if you no longer are going to use it
# remove from the remote repository
git push -d origin new-feature
# remove from your local repository
git branch -d new-feature
```

Minimum expected documentation

Every function should contain *at least* a brief description of what it does, as well as input and output description. However, complex functions might require more to be understood.

We use numpydoc style docstrings.

Function example:

```
def fibonacci(n):
    """Compute the nth fibonacci number

    Parameters
    ----------
    n: int
        The index in the fibonacci sequence whose value will be calculated

    Returns
```

(continues on next page)

(continued from previous page)

```
    -------
    int
        The nth fibonacci number
    """
    # fibonacci needs seed values for 0 and 1
    if n == 0:
        return 0
    elif n == 1:
        return 1
    # for n > 1, the nth fibonnacci number is defined as follows
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Object example:

```
class Square(object):
    def __init__(self, l):
        """Represent a square

        Parameters
        ----------
        l: float
            Side length
        """
        self.l = l

    def area(self):
        """Compute the area of the square

        Returns
        -------
        float
            The area of the square
        """
        return self.l**2
```

A note about comments: comments should explain *why* you are doing some operation *not what* operation. The what can be infered from the code itself but the why is harder to infer. You do not need to comment every line, but add them when it may be hard for others to understand what's going on

A note about objects: objects are meant to encapsulate mutable state. Mutable objectsa are hard to debug. When writing scientific software, we usually do not need mutable state, we only want to process input in a stateless manner, so only use objects when absolutely necessary.

Python 3

Write Python 3 code. Python 2 is retiring…

In most cases, it's really easy to write Python 2 and 3 compliant code, here's the official porting guide.

Using logger, not print

Print is *evil*. It does not respect anyone or anything, it just throws stuff into stdout without control. The only case when print makes sense is when developing command line applications. So use logging, it's much better and easy to setup. More about logging here.

Setting up logger in a script:

```
import logging

logger = logging.getLogger(__name__)

def my awesome_function(a):
    logger.info('This is an informative message')

    if something_happens(a):
        logger.debug('This is a debugging message: something happened,'
                     ' it is not an error but we want you to know about it')

    # do stuff...
```

If you want to log inside an object, you need to do something a bit different:

```
import logging

class MyObject(object):

    def __init__():
        self.logger = logging.getLogger(__name__)

    def do_stuff():
        self.logger.debug('Doing stuff...')
```

Code style

```
Beautiful is better than ugly. The Zen of Python
```

To make our code readable and maintanble, we need some standards, Python has a style guide called PEP8. We don't expect you to memorize it, so here's a nice guide with the basics.

If you still skipped the guide, here are the fundamental rules:

1. Variables, functions, methods, packages and modules: `lower_case_with_underscores`
2. Classes and Exceptions: `CapWords`
3. Avoid one-letter variables, except for counters
4. Use 4 spaces, never tabs
5. Line length should be between 80-100 characters

However, there are tools to automatically check if your code complies with the standard. `flake8` is one of such tools, and can check for PEP8 compliance as well as other common errors:

```
pip install flake8
```

To check a file:

```
flake8 my_script.py
```

Most text editors and IDE have plugins to automatically run tools such as `flake8` when you modify a file, here's one for Sublime Text.

If you want to know more about `flake8` and similar tools, this is a nice resource

8.5.3 Installing YASS in development mode

First, we need to install YASS in develop mode.

Clone the repo:

```
git clone https://github.com/paninski-lab/yass
```

Move to the folder containing the `setup.py` file and install the package in development mode:

```
cd yass
pip install --editable .
```

If you install it that way, you can modify the source code and changes will reflect whenever you import the modules (but you need to restart the session).

Make sure you can import the package and that it's loaded from the location where you ran `git clone`. First open a Python intrepreter:

```
python
```

And load the package you installed:

```
import yass
yass
```

You should see something like this:

```
path/to/cloned/repository
```

Developing a package without restarting a session

If you use IPython/Jupyter run these at the start of the session to reload your packages without having to restart your session:

```
%load_ext autoreload
%autoreload 2
```

8.5.4 Pull requests

Once your fix/feature is finished is time to open a pull request, the process will be as follows, let's suppose you are developing a new feature in the `new-feature` branch:

1. Make sure `new-feature` branch passes all the tests
2. Open pull request to merge to the `master` branch
3. A reviewer will go through your code and suggest chances if necessary
4. You will address those suggestions and push the updated code to `new-feature`
5. The pull request is updated automatically
6. A reviewer will go through the pull request, if no more changes are required it will accept your pull request
7. The new feature is now available in the `master` branch

8.5.5 Testing

Running tests

Each time you modify the codebase it's important that you make sure all tests pass and that all files comply with the style guide:

```
# this command will run tests and check the style in all files
pytest --flake8
```

Modifying/adding your own tests

If you are fixing a bug, chances are, you will need to update tests or add more cases. Take a look at the pytest documentation

8.6 Advanced usage

8.6.1 Building your own pipeline

WIP

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

CHAPTER 10

Changelog

10.1 0.9 (2018-05-24)

- Added parallelization to batch processor
- Preprocess step now runs in parallel
- Filtering and standarization running in one step to avoid I/O overhead

10.2 0.8 (2018-04-19)

- It is now possible to save results for every step to resume execution, see *save_results* option
- Fixed a bug that caused excessive logging when logger level was set to DEBUG
- General improvements to the sorting algorithm
- Fixes a bug that was causing and import error in the mfm module (thanks @neil-gallagher for reporting this issue)

10.3 0.7 (2018-04-06)

- New CLI tool for training neural networks *yass train*
- New CLI tool for exporting results to phy *yass export*
- Separated logic in five steps: preprocess, detect, cluster templates and deconvolute
- Improved Neural Network detector speed
- Improved package organization
- Updated examples

- Added integration tests
- Increased testing coverage
- Some examples include links to Jupyter notebooks
- Errors in documentation building are now tested in Travis
- Improved batch processor
- Simplified configuration file
- Preprocessing speedups

10.4 0.6 (2018-02-05)

- New stability metric
- New batch module
- Rewritten preprocessor
- A lot of functions were rewritten and documented
- More partial results are saved to improve debugging
- Removed a lot of legacy code
- Removed batching logic from old functions, they are now using the *batch* module
- Rewritten CLI interface *yass* command is now *yass sort*

10.5 0.5 (2018-01-31)

- Improved logging
- Last release with old codebase

10.6 0.4 (2018-01-19)

- Fixes bug in preprocessing (#38)
- Increased template size
- Updates deconvolution method

10.7 0.3 (2017-11-15)

- Adds new neural network module

10.8 0.2 (2017-11-14)

- Config module refactoring, configuration files are now much simpler
- Fixed bug that was causing spike times to be off due to the buffer
- Various bug fixes
- Updates to input/output structure
- Adds new module for augmented spikes
- Function names changes in score module
- Simplified parameters for score module functions

10.9 0.1.1 (2017-11-01)

- Minor changes to setup.py for uploading to pypi

10.10 0.1 (2017-11-01)

- First release

Python Module Index

Index