YASS Documentation

Release 0.10dev

Grossman Center at Columbia University

Dec 05, 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[tf]
```

The above command will install yass and all its dependencies (including) tensorflow (CPU), for GPU do `pip install yass-algorithm[tf-gpu]`.

If you have Tensorflow already installed, running `pip install yass-algorithm` will install yass and its dependencies except for Tensorflow. For more information regarding Tensorflow installation see this.

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

We are currently updating our documentation, this section will be updated once we have an example with publicly available data.

CHAPTER 4

Documentation

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

CHAPTER 5

Running tests

Note: this is indented only for YASS developers, our testing data is not publicly available.

Before running the tests, download the testing data:

```
export YASS_TESTING_DATA_URL=[URL-TO-TESTING-DATA]

make download-test-data

make test
```

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 (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/retina/
  # recordings filename (must be a binary file), details about the recordings
  # are specified in the recordings section
  recordings: data.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: geometry.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,
```

(continues on next page)

(continued from previous page)

```
  # '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: 20000
  # number of channels
  n_channels: 49
  # 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: 1.5
  # 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
```

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 not 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 numpy as np

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

np.random.seed(0)

# configure logging module to get useful information
```

(continues on next page)

(continued from previous page)

```
logging.basicConfig(level=logging.INFO)

# set yass configuration parameters
yass.set_config('config_sample.yaml', 'deconv-example')

standarized_path, standarized_params, whiten_filter = preprocess.run()

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


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

(templates_, spike_train,
 groups, idx_good_templates) = templates.run(
    spike_train_clear, tmp_loc)

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

Advanced usage

yass sort is a wrapper for the code in *yass.pipeline.run*, it provides a pipeline implementation with some defaults but you cannot customize it, if you want to use experimental features, the only way to do so is to customize your pipeline:

```
"""
Example for creating a custom YASS pipeline
"""
import logging
import numpy as np

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

from yass.preprocess.experimental import run as experimental_run

from yass.detect import nnet
from yass.detect import nnet_experimental
from yass.detect import threshold

# just for reproducibility..,
np.random.seed(0)

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

# set yass configuration parameters
yass.set_config('config.yaml', 'custom-example')

# run standarization using the stable implementation (by default)
```

(continues on next page)

(continued from previous page)

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

# ...or using the experimental code (see source code for details)
(standarized_path, standarized_params,
 whiten_filter) = preprocess.run(function=experimental_run)

# run detection using threshold detector
(spike_index_clear,
 spike_index_all) = detect.run(standarized_path,
                               standarized_params,
                               whiten_filter,
                               function=threshold.run)

# ...or using the neural network detector (see source code for details)
# on changing the network to use
(spike_index_clear,
 spike_index_all) = detect.run(standarized_path,
                               standarized_params,
                               whiten_filter,
                               function=nnet.run)


# ...or using the experimental neural network detector
# (see source code for details) on changing the network to use and the
# difference between this and the stable implementation
(spike_index_clear,
 spike_index_all) = detect.run(standarized_path,
                               standarized_params,
                               whiten_filter,
                               function=nnet_experimental.run)

# the rest is the same, you can customize the pipeline by passing different
# functions
spike_train_clear, tmp_loc, vbParam = cluster.run(spike_index_clear)

(templates_, spike_train,
 groups, idx_good_templates) = templates.run(
    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/retina/
  # recordings filename (must be a binary file), details about the recordings
  # are specified in the recordings section
```

(continues on next page)

(continued from previous page)

```
  recordings: data.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: geometry.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: 20000
  # number of channels
  n_channels: 49
  # 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: 1.5
  # 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
```

8.3 Using pre-built pipeline

Note: this document is outdated. The default pipeline has changed but not documented yet.

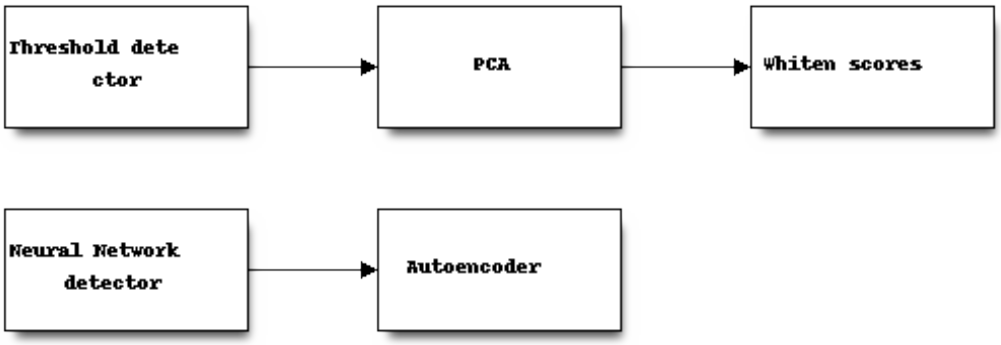
YASS provides with a pre-built pipeline for spike sorting, which consists of five parts: preprocess, detect, cluster, make templates and deconvolute.

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.

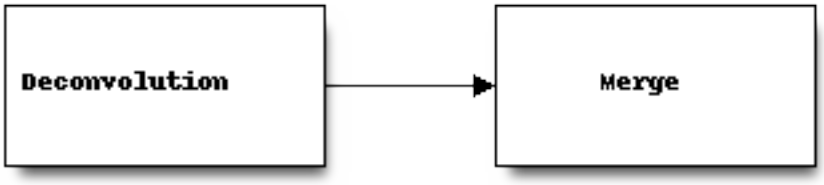
8.3.3 Cluster

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

This module contains functions for preprocessing data (filtering and standarization), the batch/ folder contains code to preprocess data using the YASS batch processor (`yass.batch`), experimental/ has a faster (but unstable implementation). The bottleneck in the stable implementation is due to the BatchProcessor, a re-implementation of the binary reader used there should fix the performance issues.

`yass.preprocess.`**`run`**(*if_file_exists='skip'*, *function=<function run>*, ***function_kwargs*)

Preprocess pipeline: filtering, standarization and whitening filter

This step (optionally) performs filtering on the data, standarizes it and computes a whitening filter. Filtering and standarized data are processed in chunks and written to disk.

Parameters

if_file_exists: str, optional One of 'overwrite', 'abort', 'skip'. Control de behavior for every 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 any of them exist

function: function, optional Which function to use, either yass.preprocess.batch.run or yass.preprocess.experimental.run. See the respective files for differences

****function_kwargs: keyword arguments** Keyword arguments passed to *function*. First argument is CONFIG, then if_file_exists and then this keyword parameters

Returns

standarized_path: str Path to standarized data binary file

standarized_params: str Path to standarized data parameters

whiten_filter: numpy.ndarray Whiten matrix

Notes

Running the preprocessor will generate the followiing files in CONFIG.data.root_folder/output_directory/:

- `preprocess/filtered.bin` - Filtered recordings
- `preprocess/filtered.yaml` - Filtered recordings metadata
- `preprocess/standarized.bin` - Standarized recordings
- `preprocess/standarized.yaml` - Standarized recordings metadata
- `preprocess/whitening.npy` - Whitening filter

Everything is run on CPU.

Examples

```
import logging

import yass
from yass import preprocess

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

# set yass configuration parameters
yass.set_config('config.yaml', 'example-preprocess')

# run preprocessor
(standarized_path,
 standarized_params,
 whiten_filter) = preprocess.run()
```

8.4.2 Detect

The *yass.detect* module implements spike detectors, nnet.py contains a Neural Network detector and threshold.py contains the threshold detector, both of them use the YASS batch processor (`yass.batch`)

`yass.detect.`**`run`**(*standarized_path*, *standarized_params*, *whiten_filter*, *if_file_exists='skip'*, *save_results=False*, *function=<function run>*)

Execute detect step

Parameters

standarized_path: str or pathlib.Path Path to standarized data binary file

standarized_params: dict, str or pathlib.Path Dictionary with standarized data parameters or path to a yaml file

whiten_filter: numpy.ndarray, str or pathlib.Path Whiten matrix or path to a npy file

if_file_exists: str, optional One of 'overwrite', 'abort', 'skip'. Control de behavior for every generated file. If 'overwrite' it replaces the files if any exist, if 'abort' it raises a ValueError exception if any file exists, if 'skip' if skips the operation if any file exists

save_results: bool, optional Whether to save results to disk, defaults to False

Returns

clear_scores: numpy.ndarray (n_spikes, n_features, n_channels) 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

spike_index_clear: numpy.ndarray (n_clear_spikes, 2) 2D array with indexes for clear spikes, first column contains the spike location in the recording and the second the main channel (channel whose amplitude is maximum)

spike_index_all: numpy.ndarray (n_collided_spikes, 2) 2D array with indexes for all spikes, first column contains the spike location in the recording and the second the main channel (channel whose amplitude is maximum)

Notes

Running the preprocessor will generate the followiing files in CONFIG.data.root_folder/output_directory/ (if save_results is True):

- `spike_index_clear.npy` - Same as spike_index_clear returned
- `spike_index_all.npy` - Same as spike_index_collision returned
- `rotation.npy` - Rotation matrix for dimensionality reduction
- `scores_clear.npy` - Scores for clear spikes

Threshold detector runs on CPU, neural network detector runs CPU and GPU, depending on how tensorflow is configured.

Examples

```
"""
Detecting spikes
"""
import logging

import yass
from yass import preprocess
from yass import detect

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

# set yass configuration parameters
yass.set_config('config.yaml', 'example-detect')

# run preprocessor
standarized_path, standarized_params, whiten_filter = preprocess.run()

# run detection
```

(continues on next page)

(continued from previous page)

```
clear, collision = detect.run(standarized_path,
                              standarized_params,
                              whiten_filter,
                              if_file_exists='overwrite')
```

8.4.3 Cluster

The *yass.cluster* module implements spike clustering algorithms, the yass.cluster.legacy implements two old clustering methods *location* and *neigh_channels* they are no longer used and will be removed soon when we add the new clustering algorithm

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

Clustering step

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 numpy as np
import logging

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

np.random.seed(0)

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

# set yass configuration parameters
```

(continues on next page)

(continued from previous page)

```
yass.set_config('config_sample.yaml', 'preprocess-example/')

standarized_path, standarized_params, whiten_filter = preprocess.run()

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

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

8.4.4 Templates

Making templates

yass.templates.**run**(**args*, ***kwargs*)

Compute templates

Parameters

spike_train: numpy.ndarray, str or pathlib.Path Spike train from cluster step or path to npy file

tmp_loc: np.array(n_templates) At which channel the clustering is done.

output_directory: str, optional Output directory (relative to CONFIG.data.root_folder) used to load the recordings to generate templates, defaults to tmp/

recordings_filename: str, optional Recordings filename (relative to CONFIG.data.root_folder/ output_directory) used to generate the templates, defaults to standarized.bin

if_file_exists: str, optional One of 'overwrite', 'abort', 'skip'. Control de behavior for the templates.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 templates to disk (in CONFIG.data.root_folder/relative_to/templates.npy), defaults to False

Returns

templates: npy.ndarray templates

spike_train: np.array(n_data, 3) The 3 columns represent spike time, unit id, weight (from soft assignment)

groups: list(n_units) After template merge, it shows which ones are merged together

idx_good_templates: np.array index of which templates are kept after clean up

Examples

```
import numpy as np
import logging

import yass
from yass import preprocess
```

(continues on next page)

(continued from previous page)

```
from yass import detect
from yass import cluster
from yass import templates

np.random.seed(0)

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

# set yass configuration parameters
yass.set_config('config_sample.yaml', 'templates-example')

standarized_path, standarized_params, whiten_filter = preprocess.run()

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


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

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

class yass.templates.**TemplatesProcessor**(*templates*)

Provides functions for manipulating templates

choose_with_indexes(*indexes*, *inplace=False*)

Keep only selected templates and from those, only the ones above certain value

crop_spatially(*neighbors*, *geometry*, *inplace=False*)

Swap channels so the first channel is the one with the largest amplitude, the second one is the nearest neighbor, and so on. Keep only n neighbors, determined by *neighbors*

8.4.5 Deconvolute

The *yass.deconvolute* module implements spikes deconvolution, yass.deconvolute.legacy contains and old algorithm that will be removed in the future

yass.deconvolute.**run**(*spike_index*, *templates*, *recordings_filename='standarized.bin'*, *function=<function legacy>*)

Deconvolute spikes

Parameters

spike_index: numpy.ndarray (n_data, 2), str or pathlib.Path A 2D array for all potential spikes whose first column indicates the spike time and the second column the principal channels. Or path to npy file

templates: numpy.ndarray (n_channels, waveform_size, n_templates), str

or pathlib.Path A 3D array with the templates. Or path to npy file

output_directory: str, optional Output directory (relative to CONFIG.data.root_folder) used to load the recordings to generate templates, defaults to tmp/

recordings_filename: str, optional Recordings filename (relative to CONFIG.data.root_folder/ output_directory) used to draw the waveforms from, defaults to standarized.bin

Returns

spike_train: numpy.ndarray (n_clear_spikes, 2) A 2D array with the spike train, first column indicates the spike time and the second column the neuron ID

Examples

```
import logging
import numpy as np

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

np.random.seed(0)

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

# set yass configuration parameters
yass.set_config('config_sample.yaml', 'deconv-example')

standarized_path, standarized_params, whiten_filter = preprocess.run()

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


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

(templates_, spike_train,
 groups, idx_good_templates) = templates.run(
    spike_train_clear, tmp_loc)

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

8.4.6 Geometry

Functions for parsing geometry data

yass.geometry.**find_channel_neighbors**(*geom*, *radius*)
Compute a neighbors matrix by using a radius

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.**make_channel_index**(*neighbors*, *channel_geometry*, *steps=1*)
Compute an array whose whose ith row contains the ordered (by distance) neighbors for the ith channel

yass.geometry.**n_steps_neigh_channels**(*neighbors_matrix*, *steps*)
Compute a neighbors matrix by considering neighbors of neighbors

Parameters

neighbors_matrix: 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.**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

class yass.batch.**BatchProcessor**(*path_to_recordings*, *dtype=None*, *n_channels=None*, *data_order=None*, *max_memory='1GB'*, *buffer_size=0*, *loader='memmap'*, *show_progress_bar=True*)

Batch processing for large numpy matrices

Parameters

path_to_recordings: str Path to recordings file

dtype: str Numpy dtype

n_channels: int Number of channels

data_order: str 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

max_memory: int or str Max memory to use in each batch, interpreted as bytes if int, if string, it can be any of {N}KB, {N}MB or {N}GB

buffer_size: int, optional Buffer size, defaults to 0. Only relevant when performing multi-channel operations

loader: str ('memmap', 'array' or 'python'), optional How to load the data. memmap loads the data using a wrapper around np.memmap (see *MemoryMap* for details), 'array' using numpy.fromfile and 'python' loads it using a wrapper around Python file API. Defaults to 'python'. Beware that the Python loader has limited indexing capabilities, see *BinaryReader* for details

show_progress_bar: bool, optional Show progress bar when running operations, defaults to True

Raises

ValueError If dimensions do not match according to the file size, dtype and number of channels

multi_channel(*from_time=None*, *to_time=None*, *channels='all'*, *return_data=True*)

Generate indexes where each index has observations from more than one channel

Returns

generator: A tuple of size three: the first element is the subset of the data for the ith batch, second element is the slice object with the limits of the data in [observations, channels] format (excluding the buffer), the last element is the absolute index of the data again in [observations, channels] format

Examples

```
# coding: utf-8

# See notebook:
# https://github.com/paninski-lab/yass-examples/blob/master/batch/multi_
↪channel.ipynb

"""
Splitting large files into batches where every batch has n
observations from m channels using BatchProcessor.multi_channel
"""

import os
from yass.batch import BatchProcessor


path_to_neuropixel_data = (os.path.expanduser('~/data/ucl-neuropixel'
                           '/rawDataSample.bin'))


bp = BatchProcessor(path_to_neuropixel_data,
                    dtype='int16', n_channels=385, data_format='wide',
                    max_memory='300MB')


# now, let's to some multi_channel operations, here we will
# traverse all channels and all observations, each batch will
# contain a subset in the temporal dimension, the window size
# is determined by max_memory
data = bp.multi_channel()

for d, _, idx in data:
    print('Shape: {}. Index: {}'.format(d.shape, idx))


# we can specify the temporal limits and subset channels
data = bp.multi_channel(from_time=100000, to_time=200000, channels=[0, 1, 2])

for d, _, idx in data:
    print('Shape: {}. Index: {}'.format(d.shape, idx))


# we can also create a BatchProcessor with a buffer
bp2 = BatchProcessor(path_to_neuropixel_data,
                     dtype='int16', n_channels=385, data_format='wide',
                     max_memory='100KB', buffer_size=10)

data = bp2.multi_channel(from_time=0, to_time=100000, channels=[0, 1, 2])

for d, idx_local, idx in data:
    # d is the batch data (with buffer), d[idx_local] returns the data
    # excluding buffer and idx is the absolute location for the
    # current batch in the recordings
    print('Shape: {}. Local: {}. Absolute: {}\n'
          .format(d.shape, idx_local, idx))
```

multi_channel_apply(*function*, *mode*, *cleanup_function=None*, *output_path=None*, *from_time=None*, *to_time=None*, *channels='all'*, *if_file_exists='overwrite'*, *cast_dtype=None*, *pass_batch_info=False*, *pass_batch_results=False*, *processes=1*, ***kwargs*)

Apply a function where each batch has observations from more than one channel

Parameters

function: callable Function to be applied, first parameter passed will be a 2D numpy array in 'long' shape (number of observations, number of channels). If pass_batch_info is True, another two keyword parameters will be passed to function: 'idx_local' is the slice object with the limits of the data in [observations, channels] format (excluding the buffer), 'idx' is the absolute index of the data again in [observations, channels] format

mode: str 'disk' or 'memory', if 'disk', a binary file is created at the beginning of the operation and each partial result is saved (ussing numpy.ndarray.tofile function), at the end of the operation two files are generated: the binary file and a yaml file with some file parameters (useful if you want to later use RecordingsReader to read the file). If 'memory', partial results are kept in memory and returned as a list

cleanup_function: callable, optional A function to be executed after *function* and before adding the partial result to the list of results (if *memory* mode) or to the biinary file (if in *disk mode*). *cleanup_function* will be called with the following parameters (in that order): result from applying *function* to the batch, slice object with the idx where the data is located (exludes buffer), slice object with the absolute location of the data and buffer size

output_path: str, optional Where to save the output, required if 'disk' mode

force_complete_channel_batch: bool, optional If True, every index generated will correspond to all the observations in a single channel, hence n_batches = n_selected_channels, defaults to True. If True from_time and to_time must be None

from_time: int, optional Starting time, defaults to None

to_time: int, optional Ending time, defaults to None

channels: int, tuple or str, optional A tuple with the channel indexes or 'all' to traverse all channels, defaults to 'all'

if_file_exists: str, optional One of 'overwrite', 'abort', 'skip'. If 'overwrite' it replaces the file if it exists, if 'abort' if raise a ValueError exception if the file exists, if 'skip' if skips the operation if the file exists. Only valid when mode = 'disk'

cast_dtype: str, optional Output dtype, defaults to None which means no cast is done

pass_batch_info: bool, optional Whether to call the function with batch info or just call it with the batch data (see description in the function) parameter

pass_batch_results: bool, optional Whether to pass results from the previous batch to the next one, defaults to False. Only relevant when mode='memory'. If True, function will be called with the keyword parameter 'previous_batch' which contains the computation for the last batch, it is set to None in the first batch

****kwargs** kwargs to pass to function

Returns

output_path, params (when mode is 'disk') Path to output binary file, Binary file params

list (when mode is 'memory' and pass_batch_results is False) List where every element is the result of applying the function to one batch. When pass_batch_results is True, it returns the output of the function for the last batch

Notes

Applying functions will incur in memory overhead, which depends on the function implementation, this is an important thing to consider if the transformation changes the data's dtype (e.g. converts int16 to float64), which means that a chunk of 1MB in int16 will have a size of 4MB in float64. Take that into account when setting max_memory

For performance reasons, outputs data in 'samples' order.

Examples

```
# coding: utf-8

# See notebook:
# https://github.com/paninski-lab/yass-examples/blob/master/batch/multi_
↪channel_apply_disk.ipynb

"""
Applying transformations to large files in batches:

BatchProcessor.multi_channel_apply lets you apply transformations to
batches of data where every batch has observations from every channel.

This example show how to process a large file in batches and save the
results to disk.
"""

import logging
import os

import matplotlib.pyplot as plt

from yass.batch import BatchProcessor
from yass.batch import RecordingsReader


# configure logging to get information about the process
logging.basicConfig(level=logging.INFO)


# raw data file
path_to_neuropixel_data = (os.path.expanduser('~/data/ucl-neuropixel'
                           '/rawDataSample.bin'))

# output file
path_to_modified_data = (os.path.expanduser('~/data/ucl-neuropixel'
                         '/tmp/modified.bin'))


# out example function just adds one to every observation
def sum_one(batch):
    """Add one to every element in the batch
    """
    return batch + 1
```

(continues on next page)

(continued from previous page)

```
# create batch processor for the data
bp = BatchProcessor(path_to_neuropixel_data,
                    dtype='int16', n_channels=385, data_format='wide',
                    max_memory='500MB')

# appply a multi channel transformation, each batch will be a temporal
# subset with observations from all selected n_channels, the size
# of the subset is calculated depending on max_memory. Each batch is
# processed and when done, results are save to disk, the next batch is
# then loaded and so on
bp.multi_channel_apply(sum_one,
                       mode='disk',
                       output_path=path_to_modified_data,
                       channels=[0, 1, 2])


# let's visualize the results
raw = RecordingsReader(path_to_neuropixel_data, dtype='int16',
                       n_channels=385, data_format='wide')

# you do not need to specify the format since multi_channel_apply
# saves a yaml file with such parameters
filtered = RecordingsReader(path_to_modified_data)

fig, (ax1, ax2) = plt.subplots(2, 1)
ax1.plot(raw[:2000, 0])
ax2.plot(filtered[:2000, 0])
plt.show()
```

```
# coding: utf-8

# See notebook:
# https://github.com/paninski-lab/yass-examples/blob/master/batch/multi_
↪channel_apply_memory.ipynb

"""
Applying transformations to large files in batches:

BatchProcessor.multi_channel_apply lets you apply transformations to
batches of data where every batch has observations from every channel.

This example show how to extract information from a large file by
processing it in batches.
"""

import logging
import os

import numpy as np

from yass.batch import BatchProcessor


# configure logging to get information about the process
logging.basicConfig(level=logging.INFO)
```

(continues on next page)

(continued from previous page)

```
# raw data file
path_to_neuropixel_data = (os.path.expanduser('~/data/ucl-neuropixel'
                           '/rawDataSample.bin'))


# on each batch, we find the maximum value in every channel
def max_in_channel(batch):
    """Add one to every element in the batch
    """
    return np.max(batch, axis=0)


# create batch processor for the data
bp = BatchProcessor(path_to_neuropixel_data,
                    dtype='int16', n_channels=385, data_format='wide',
                    max_memory='10MB')

# appply a multi channel transformation, each batch will be a temporal
# subset with observations from all selected n_channels, the size
# of the subset is calculated depending on max_memory. Results
# from every batch are returned in a list
res = bp.multi_channel_apply(max_in_channel,
                             mode='memory',
                             channels=[0, 1, 2])


# we have one element per batch
len(res)


# output for the first batch
res[0]


# stack results from every batch
arr = np.stack(res, axis=0)


# let's find the maximum value along every channel in all the dataset
np.max(arr, axis=0)
```

single_channel(*force_complete_channel_batch=True*, *from_time=None*, *to_time=None*, *channels='all'*)

Generate batches where each index has observations from a single channel

Returns

A generator that yields batches, if force_complete_channel_batch is False, each generated value is a tuple with the batch and the channel for the index for the corresponding channel

Examples

```
# coding: utf-8

# See notebook:
# https://github.com/paninski-lab/yass-examples/blob/master/batch/single_
↪channel.ipynb

"""
Splitting large file into batches where every batch contains n
observations from 1 channel
"""

import os
from yass.batch import BatchProcessor


path_to_neuropixel_data = (os.path.expanduser('~/data/ucl-neuropixel'
                           '/rawDataSample.bin'))


bp = BatchProcessor(path_to_neuropixel_data,
                    dtype='int16', n_channels=385, data_format='wide',
                    max_memory='1MB')

# there are two ways of traversing the data: single_channel and multi_channel
# single_channel means that the data in a single batch comes from only one
# channel, multi_channel means that a batch can contain data from multiple
# channels, let's take a look at single_channel operations

# traverse the whole dataset, one channel at a time
data = bp.single_channel()

# the next for loop will raise an error since we cannot fit
# all observations for a single channel in memory, so we
# either increase max_memory or set
# force_complete_channel_batch to False

# for d in data:
#     print(d.shape)


# When force_complete_channel_batch is False, each batch does not necessarily
# correspond to all observations in the channel, the channel can be splitted
# in several batches (although every batch data is guaranteed to come from
# a single channel), in this case, every channel is splitted in two parts
data = bp.single_channel(force_complete_channel_batch=False,
                         channels=[0, 1, 2])

for d, ch in data:
    print(d.shape, 'Data from channel {}'.format(ch))


# finally, we can traverse a single channel in a temporal subset
data = bp.single_channel(from_time=100000, to_time=200000, channels=[0, 1, 2])

for d in data:
```

(continues on next page)

(continued from previous page)

```
    print(d.shape)
```

single_channel_apply(*function*, *mode*, *output_path=None*, *force_complete_channel_batch=True*, *from_time=None*, *to_time=None*, *channels='all'*, *if_file_exists='overwrite'*, *cast_dtype=None*, ***kwargs*)

Apply a transformation where each batch has observations from a single channel

Parameters

function: callable Function to be applied, must accept a 1D numpy array as its first parameter

mode: str 'disk' or 'memory', if 'disk', a binary file is created at the beginning of the operation and each partial result is saved (ussing numpy.ndarray.tofile function), at the end of the operation two files are generated: the binary file and a yaml file with some file parameters (useful if you want to later use RecordingsReader to read the file). If 'memory', partial results are kept in memory and returned as a list

output_path: str, optional Where to save the output, required if 'disk' mode

force_complete_channel_batch: bool, optional If True, every index generated will correspond to all the observations in a single channel, hence n_batches = n_selected_channels, defaults to True. If True from_time and to_time must be None

from_time: int, optional Starting time, defaults to None

to_time: int, optional Ending time, defaults to None

channels: int, tuple or str, optional A tuple with the channel indexes or 'all' to traverse all channels, defaults to 'all'

if_file_exists: str, optional One of 'overwrite', 'abort', 'skip'. If 'overwrite' it replaces the file if it exists, if 'abort' if raise a ValueError exception if the file exists, if 'skip' if skips the operation if the file exists. Only valid when mode = 'disk'

cast_dtype: str, optional Output dtype, defaults to None which means no cast is done

****kwargs** kwargs to pass to function

Notes

When applying functions in 'disk' mode will incur in memory overhead, which depends on the function implementation, this is an important thing to consider if the transformation changes the data's dtype (e.g. converts int16 to float64), which means that a chunk of 1MB in int16 will have a size of 4MB in float64. Take that into account when setting max_memory.

For performance reasons in 'disk' mode, output data is in 'channels' order

Examples

```
# coding: utf-8

# See notebook:
# https://github.com/paninski-lab/yass-examples/blob/master/batch/single_
↪channel_apply.ipynb
```

(continues on next page)

(continued from previous page)

```
"""
Apply functions to large files using
BatchProcessor.single_channel_apply
"""

import logging
import os

import matplotlib.pyplot as plt

from yass.batch import BatchProcessor
from yass.batch import RecordingsReader
from yass.preprocess.filter import butterworth


logging.basicConfig(level=logging.INFO)


path_to_neuropixel_data = (os.path.expanduser('~/data/ucl-neuropixel'
                           '/rawDataSample.bin'))
path_to_filtered_data = (os.path.expanduser('~/data/ucl-neuropixel'
                         '/tmp/filtered.bin'))


# create batch processor for the data
bp = BatchProcessor(path_to_neuropixel_data,
                    dtype='int16', n_channels=385, data_format='wide',
                    max_memory='500MB')


# appply a single channel transformation, each batch will be all observations
# from one channel, results are saved to disk
bp.single_channel_apply(butterworth,
                        mode='disk',
                        output_path=path_to_filtered_data,
                        low_freq=300, high_factor=0.1,
                        order=3, sampling_freq=30000,
                        channels=[0, 1, 2])


# let's visualize the results
raw = RecordingsReader(path_to_neuropixel_data, dtype='int16',
                       n_channels=385, data_format='wide')

# you do not need to specify the format since single_channel_apply
# saves a yaml file with such parameters
filtered = RecordingsReader(path_to_filtered_data)

fig, (ax1, ax2) = plt.subplots(2, 1)
ax1.plot(raw[:2000, 0])
ax2.plot(filtered[:2000, 0])
plt.show()
```

Batch Pipeline

class yass.batch.**BatchPipeline**(*path_to_input*, *dtype*, *n_channels*, *data_order*, *max_memory*, *output_path*, *from_time=None*, *to_time=None*, *channels='all'*)

Chain batch operations and write partial results to disk

Parameters

path_to_input: str Path to input file

dtype: str Numpy dtype

n_channels: int Number of channels

data_order: str 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

max_memory: int or str Max memory to use in each batch, interpreted as bytes if int, if string, it can be any of {N}KB, {N}MB or {N}GB

output_path: str Folder indicating where to store the files from every step

from_time: int, optional Starting time, defaults to None, which means start from time 0

to_time: int, optional Ending time, defaults to None, which means end at the last observation

channels: int, tuple or str, optional A tuple with the channel indexes or 'all' to traverse all channels, defaults to 'all'

Examples

```
import logging
import os

import matplotlib.pyplot as plt

from yass.batch.pipeline import BatchPipeline, PipedTransformation
from yass.batch import RecordingsReader
from yass.preprocess.filter import butterworth, standarize

logging.basicConfig(level=logging.DEBUG)


path_to_neuropixel_data = (os.path.expanduser('~/data/ucl-neuropixel'
                                              '/rawDataSample.bin'))
path_output = os.path.expanduser('~/data/ucl-neuropixel/tmp')


pipeline = BatchPipeline(path_to_neuropixel_data, dtype='int16',
                         n_channels=385, data_format='wide',
                         max_memory='500MB',
                         from_time=None, to_time=None, channels='all',
                         output_path=path_output)

butterworth_op = PipedTransformation(butterworth, 'filtered.bin',
                                     mode='single_channel_one_batch',
```

(continues on next page)

(continued from previous page)

```
                                        keep=True, low_freq=300, high_factor=0.1,
                                        order=3, sampling_freq=30000)

standarize_op = PipedTransformation(standarize, 'standarized.bin',
                                    mode='single_channel_one_batch',
                                    keep=True, sampling_freq=30000)

pipeline.add([butterworth_op, standarize_op])

pipeline.run()


raw = RecordingsReader(path_to_neuropixel_data, dtype='int16',
                       n_channels=385, data_format='wide')
filtered = RecordingsReader(os.path.join(path_output, 'filtered.bin'))
standarized = RecordingsReader(os.path.join(path_output, 'standarized.bin'))

# plot results
fig, (ax1, ax2, ax3) = plt.subplots(3, 1)
ax1.plot(raw[:2000, 0])
ax1.set_title('Raw data')
ax2.plot(filtered[:2000, 0])
ax2.set_title('Filtered data')
ax3.plot(standarized[:2000, 0])
ax3.set_title('Standarized data')
plt.tight_layout()
plt.show()
```

run()

Run all tasks in the pipeline

Returns

list List with path to output files in the order they were run, if keep is False, path is still returned but file will not exist

list List with parameters

class yass.batch.**PipedTransformation**(*function*, *output_name*, *mode*, *keep=False*, *if_file_exists='overwrite'*, *cast_dtype=None*, ***kwargs*)

Wrapper for functions run with BatchPipeline

Parameters

function: function Function to apply

output_name: str Name of the file for the output

mode: str Operation mode, one of 'single_channel_one_batch' (every batch are all observations from a single channel), 'single_channel' (every batch are observations from a single channel, but can be splitted in several batches to avoid exceeding max_memory) or 'multi_channel' (every batch has observations for every channel selected, batches are splitted not to exceed max_memory)

keep: bool, optional Whether to keep the results from this step, otherwise the file is deleted after the next transformation is done

if_file_exists: str, optional One of 'overwrite', 'abort', 'skip'. If 'overwrite' it replaces the file if it exists, if 'abort' if raise a ValueError exception if the file exists, if 'skip' if skips the

operation if the file exists. Only valid when mode = 'disk'

cast_dtype: str, optional Output dtype, defaults to None which means no cast is done

****kwargs** Function kwargs

Recordings Reader

class yass.batch.**RecordingsReader**(*path_to_recordings*, *dtype=None*, *n_channels=None*, *data_order=None*, *loader='memmap'*, *buffer_size=0*, *return_data_index=False*)

Neural recordings reader. If a file with the same name but yaml extension exists in the directory it looks for dtype, channels and data_order, otherwise you need to pass the parameters in the constructor

Parameters

path_to_recordings: str Path to recordings file

dtype: str Numpy dtype

n_channels: int Number of channels

data_order: str 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

loader: str ('memmap', 'array' or 'python'), optional How to load the data. memmap loads the data using a wrapper around np.memmap (see *MemoryMap* for details), 'array' using numpy.fromfile and 'python' loads it using a wrapper around Python file API. Defaults to 'python'. Beware that the Python loader has limited indexing capabilities, see *BinaryReader* for details

buffer_size: int, optional Adds buffer

return_data_index: bool, optional If True, a tuple will be returned when indexing: the first element will be the data and the second the index corresponding to the actual data (excluding bufffer), when buffer is equal to zero, this just returns they original index since there is no buffer

Raises

ValueError If dimensions do not match according to the file size, dtype and number of channels

Notes

This is just an utility class to index binary files in a consistent way, it does not matter the order of the file ('channels' or 'samples'), indexing is performed in [observations, channels] format. This class is mainly used by other internal YASS classes to maintain a consistent indexing order.

Examples

```
# coding: utf-8

# See notebook:
# https://github.com/paninski-lab/yass-examples/blob/master/batch/reader.ipynb
```

(continues on next page)

(continued from previous page)

```
"""
Reading large files with RecordingsReader
"""

import os

import numpy as np
from yass.batch import RecordingsReader


# generate data
output_folder = os.path.join(os.path.expanduser('~'), 'data/yass')
wide_data = np.random.rand(50, 100000)
long_data = wide_data.T

path_to_wide = os.path.join(output_folder, 'wide.bin')
path_to_long = os.path.join(output_folder, 'long.bin')

wide_data.tofile(path_to_wide)
long_data.tofile(path_to_long)


# load the files using the readers, they are agnostic on the data shape
# and will behave exactly the same
reader_wide = RecordingsReader(path_to_wide, dtype='float64',
                               n_channels=50, data_order='channels')

reader_long = RecordingsReader(path_to_long, dtype='float64',
                               n_channels=50, data_order='samples')


reader_wide.shape, reader_long.shape


# first index is for observations and second index for channels
obs = reader_wide[10000:20000, 20:30]
obs, obs.shape


# same applies even if your data is in 'wide' shape, first index for
# observations, second for channels, the output is converted to 'long'
# by default but you can change it
obs = reader_long[10000:20000, 20:30]
obs, obs.shape
```

channels
: Number of channels

data
: Underlying numpy data

data_order
: Data order

dtype
: Numpy's dtype

observations
Number of observations

shape
Data shape in (observations, channels) format

Binary Readers

class yass.batch.**BinaryReader**(*path_to_file*, *dtype*, *shape*, *order='F'*)
Reading batches from large array binary files on disk, similar to numpy.memmap. It is essentially just a wrapper around Python files API to read through large array binary file using the array[:,:] syntax.

Parameters

order: str Array order 'C' for 'Row-major order' or 'F' for 'Column-major order'

Notes

https://en.wikipedia.org/wiki/Row-_and_column-major_order

class yass.batch.**MemoryMap**(**args*, ***kwargs*)
Wrapper for numpy.memmap that creates a new memmap on each __getitem__ call to save memory

8.4.8 Augment

yass.augment.make.**load_templates**(*data_folder*, *spike_train*, *CONFIG*, *chosen_templates_indexes*)

Parameters

data_folder: str Folder storing the standarized data (if not exist, run preprocess to automatically generate)

spike_train: numpy.ndarray [number of spikes, 2] Ground truth for training. First column is the spike time, second column is the spike id

chosen_templates_indexes: list List of chosen templates' id's

yass.augment.make.**spikes**(*templates*, *min_amplitude*, *max_amplitude*, *n_per_template*, *spatial_sig*, *temporal_sig*, *make_from_templates=True*, *make_spatially_misaligned=True*, *make_temporally_misaligned=True*, *make_collided=True*, *make_noise=True*, *return_metadata=True*, *templates_kwargs={}*, *collided_kwargs={'min_shift': 5, 'n_per_spike': 1}*, *temporally_misaligned_kwargs={'n_per_spike': 1}*, *spatially_misaligned_kwargs={'force_first_channel_shuffle': True, 'n_per_spike': 1}*, *add_noise_kwargs={'reject_cancelling_noise': False}*)
Make spikes, it creates several types of spikes from templates with a range of amplitudes

Parameters

templates: numpy.ndarray, (n_templates, waveform_length, n_channels) Templates used to generate the spikes

min_amplitude: float Minimum amplitude for the spikes

max_amplitude: float Maximum amplitude for the spikes

n_per_template: int How many spikes to generate per template. This along with min_amplitude and max_amplitude are used to generate spikes covering the desired amplitude range

make_from_templates: bool Whether to return spikes generated from the templates (these are the same as the templates but with different amplitudes)

make_spatially_misaligned: bool Whether to return spatially misaligned spikes (by shuffling channels)

make_temporally_misaligned: bool Whether to return temporally misaligned spikes (by shifting along the temporal axis)

make_collided: bool Whether to return collided spikes

make_noise: bool Whether to return pure noise

return_metadata: bool, optional Return metadata in the generated spikes

Returns

x_all: numpy.ndarray, (n_templates * n_per_template, waveform_length,

n_channels) All generated spikes

x_all_noisy: numpy.ndarray, (n_templates * n_per_template, waveform_length,

n_channels) Noisy versions of all generated spikes

the_amplitudes: numpy.ndarray, (n_templates * n_per_template,) Amplitudes for all generated spikes

slices: dictionary Dictionary where the keys are the kind of spikes ('from templates', 'spatially misaligned', 'temporally misaligned', 'collided', noise') and the values are slice objects with the location for each kind of spike

spatial_SIG

temporal_SIG

yass.augment.make.**training_data**(*CONFIG*, *templates_uncropped*, *min_amp*, *max_amp*, *n_isolated_spikes*, *path_to_standarized*, *noise_ratio=10*, *collision_ratio=1*, *misalign_ratio=1*, *misalign_ratio2=1*, *multi_channel=True*, *return_metadata=False*)

Makes training sets for detector, triage and autoencoder

Parameters

CONFIG: yaml file Configuration file

min_amp: float Minimum value allowed for the maximum absolute amplitude of the isolated spike on its main channel

max_amp: float Maximum value allowed for the maximum absolute amplitude of the isolated spike on its main channel

n_isolated_spikes: int Number of isolated spikes to generate. This is different from the total number of x_detect

path_to_standarized: str Folder storing the standarized data (if not exist, run preprocess to automatically generate)

noise_ratio: int Ratio of number of noise to isolated spikes. For example, if n_isolated_spike=1000, noise_ratio=5, then n_noise=5000

collision_ratio: int Ratio of number of collisions to isolated spikes.

misalign_ratio: int Ratio of number of spatially and temporally misaligned spikes to isolated spikes

misalign_ratio2: int Ratio of number of only-spatially misaligned spikes to isolated spikes

multi_channel: bool If True, generate training data for multi-channel neural network. Otherwise generate single-channel data

Returns

x_detect: numpy.ndarray [number of detection training data, temporal length, number of channels] Training data for the detect net.

y_detect: numpy.ndarray [number of detection training data] Label for x_detect

x_triage: numpy.ndarray [number of triage training data, temporal length, number of channels] Training data for the triage net.

y_triage: numpy.ndarray [number of triage training data] Label for x_triage

x_ae: numpy.ndarray [number of ae training data, temporal length] Training data for the autoencoder: noisy spikes

y_ae: numpy.ndarray [number of ae training data, temporal length] Denoised x_ae

Notes

- **Detection training data**
  - **Multi channel**
    - Positive examples: Clean spikes + noise, Collided spikes + noise
    - Negative examples: Temporally misaligned spikes + noise, Noise
- **Triage training data**
  - **Multi channel**
    - Positive examples: Clean spikes + noise
    - Negative examples: Collided spikes + noise

yass.augment.make.**training_data_detect**(*templates*, *minimum_amplitude*, *maximum_amplitude*, *n_clean_per_template*, *n_collided_per_spike*, *n_temporally_misaligned_per_spike*, *n_spatially_misaliged_per_spike*, *n_noise*, *spatial_SIG*, *temporal_SIG*, *from_templates_kwargs={}*, *collided_kwargs={}*, *temporally_misaligned_kwargs={}*, *add_noise_kwargs={'reject_cancelling_noise': False}*)

Make training data for detector network

Notes

Recordings are passed through the detector network which identifies spikes (clean and collided), it rejects noise and misaligned spikes (temporally and spatially)

yass.augment.make.**training_data_triage**(*templates*, *minimum_amplitude*, *maximum_amplitude*, *n_clean_per_template*, *n_collided_per_spike*, *max_shift*, *min_shift*, *spatial_SIG*, *temporal_SIG*, *from_templates_kwargs*, *collided_kwargs*)

Make training data for triage network

8.4.9 Neural network

Neural Network Detector

class yass.neuralnetwork.**NeuralNetDetector**(*path_to_model*, *filters_size*, *waveform_length*, *n_neighbors*, *threshold*, *channel_index*, *n_iter=50000*, *n_batch=512*, *l2_reg_scale=5e-08*, *train_step_size=0.001*, *load_test_set=False*)

Class for training and running convolutional neural network detector for spike detection

Parameters

C: int spatial filter size of the spatial convolutional layer.

R1: int temporal filter sizes for the temporal convolutional layers.

K1,K2: int number of filters for each convolutional layer.

W1, W11, W2: tf.Variable [temporal_filter_size, spatial_filter_size, input_filter_number, ouput_filter_number] weight matrices for the covolutional layers.

b1, b11, b2: tf.Variable bias variable for the convolutional layers.

saver: tf.train.Saver saver object for the neural network detector.

threshold: int threshold for neural net detection

channel_index: np.array (n_channels, n_neigh) Each row indexes its neighboring channels. For example, channel_index[c] is the index of neighboring channels (including itself) If any value is equal to n_channels, it is nothing but a placeholder in a case that a channel has less than n_neigh neighboring channels

Attributes

SPIKE: int Label assigned to the spike class (1)

NOT_SPIKE: int Label assigned to the not spike class (0)

fit(*x_train*, *y_train*, *test_size=0.3*, *save_test_set=False*)

Trains the neural network detector for spike detection

Parameters

x_train: np.array [number of training data, temporal length, number of channels] augmented training data consisting of isolated spikes, noise and misaligned spikes.

y_train: np.array [number of training data] label for x_train. '1' denotes presence of an isolated spike and '0' denotes the presence of a noise data or misaligned spike.

test_size: float, optional Proportion of the training set to be used, data is shuffled before splitting, defaults to 0.3

Returns

dict Dictionary with network parameters and metrics

predict(*waveforms*)
Predict classes (higher or equal than threshold)

predict_proba(*waveforms*)
Predict probabilities

predict_recording(*recording*, *output_names=('spike_index',)*, *sess=None*)
Make predictions on recordings

Parameters

output: tuple Which output layers to return, valid options are: spike_index, waveform and probability

Returns

tuple A tuple of numpy.ndarrays, one for every element in output_names

restore(*sess*)
Restore tensor values

Neural Network Triage

class yass.neuralnetwork.**NeuralNetTriage**(*path_to_model*, *filters_size*, *waveform_length*, *n_neighbors*, *threshold*, *n_iter=50000*, *n_batch=512*, *l2_reg_scale=5e-08*, *train_step_size=0.001*, *input_tensor=None*, *load_test_set=False*)
Convolutional Neural Network for spike detection

Parameters

path_to_model: str Where to save the trained model

threshold: float Threshold between 0 and 1, values higher than the threshold are considered spikes

input_tensor

Attributes

C: int spatial filter size of the spatial convolutional layer.

R1: int temporal filter sizes for the temporal convolutional layers.

K1,K2: int number of filters for each convolutional layer.

W1, W11, W2: tf.Variable [temporal_filter_size, spatial_filter_size, input_filter_number, ouput_filter_number] weight matrices for the covolutional layers.

b1, b11, b2: tf.Variable bias variable for the convolutional layers.

saver: tf.train.Saver saver object for the neural network detector.

detector: NeuralNetDetector Instance of detector

threshold: int threshold for neural net triage

CLEAN: int Label assigned to the clean spike class (1)

COLLIDED: int Label assigned to the collided spike class (0)

fit(*x_train*, *y_train*, *test_size=0.3*, *save_test_set=False*)
Trains the triage network

Parameters

x_train: np.array [number of data, temporal length, number of channels] training data for the triage network.

y_train: np.array [number of data] training label for the triage network.

test_size: float, optional Proportion of the training set to be used, data is shuffled before splitting, defaults to 0.3

Returns

dict Dictionary with network parameters and metrics

Notes

Size is determined but the second dimension in x_train

classmethod load(*path_to_model*, *threshold*, *input_tensor=None*, *load_test_set=False*)
Load a model from a file

predict(*waveforms*)
Triage waveforms

restore(*sess*)
Restore tensor values

8.5 Developer's Guide

8.5.1 Important information

If you are planning to dive into YASS codebase, these are some important things to know (as of November, 2018)

- Documentation is good but some parts are outdated, most functions have docstrings but beware that some of them need updates
- There are some examples and tools in https://github.com/paninski-lab/yass-examples. Some of the examples are outdated
- We are testing with date we cannot share, so every time we run Travis, we have to download it. See scripts/ for details
- There is a *version* script in the root folder which automates releases
- We no longer have an example in the README, in the past we provided an example using a sample of neuropixel data but since we are not developing (yet) with that type of data, we removed it since YASS has not been tested with such data yet

Code architecture changes

We have been making some architecture changes, in the past we only kept one implementation of every pipeline step and merged to master when ready. This proved troublesome, in some cases, the steps (e.g. clustering) changed completely but since they are still experimental, we did not have any stable implementation. We are trying out another alternative: keep a stable implementation and a experimental one at the same time. Each step in the pipeline then offers a paramter to choose which implementation to use.

So now each step is just a wrapper for another function which provides convenient features.

In the future, it would be good to separate this "wrapping" logic completely. Right know, we pass the entire CONFIG object to these functions, which makes the code hard to understand (we do not know which parameters the function uses), it would be better for each implementation to provide all their parameters (with nice defaults) in the function signature so that it is clear which parameters they use.

BatchReader and "experimental" implementations

The preprocess and detector experimental implementations were mostly done due to performance issues with the stable implementations. The stable implementation use the BatchReader, which internally uses the RecordingsReader. This reader seems to be slow and is the major bottleneck of the stable implementations. Re-implementing this reader will probably put the stable implementation at comparable speed with the experimental implementations. This is an important step since the stable implementations are much more readable and clean.

There are some features (especially in the detector) that are missing from the stable implementations, but this are not difficult to migrate to the stable code.

YASS Configuration file

Initially, we had the YASS configuration file to hold all parameters for all steps in the pipeline. This caused a lot of trouble since we had to modify the schema.yaml file to reflect changes in experimental code, whenever someone changed any of the steps a bit and wanted to introduce a new parameter we needed to update the schema.yaml and tests to make sure the code still worked.

The new architecture takes out most of the parameters and only keeps the essentials, this comes to a cost: end users will have to write Pythonn code if they want to customize the default pipeline since most parameters are no longer part of the configuration file but parameters in the functions that implement the steps. However, given the modular architecture, this is trivial to do (see examples/pipeline/custom.py)

The advantage of this is that we no longer have to maintain a configuration file, each new implementation is responsible for providing defaults for its parameters.

Testing

Most of the tests are unit or smoke tests (all of them located in tests/unit/, although we should probably split them into two folders). Unit tests check functionality of YASS internal utilities (such as BatchReader). Some tests just make sure the code runs (without checking results).

There are some "reference" tests which check that some deterministic pieces of code still return the same results (such as preprocessing), this was done since the code was being optimized and we wanted to make sure we were not breaking the implementation.

There also integration tests which run the files in examples/.

Towards YASS 1.0

While we have a stable implementation for most of the steps in the pipeline, the latest and greatest is still in development and lacks of a stable, clean implementation. Before making a major release, the followning has to be addressed.

- Re-implement RecordingsReader to put the stable implementations on par with the experimental implementations (preprocess and detect)
- Refactor experimental implementations to use the RecordingsReader/BatchProcessor
- Clean up and test the experimental implementations
- Update documentation

8.5.2 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

8.5.3 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
    -------
    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

8.5.4 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
```

8.5.5 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

8.5.6 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

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

CHAPTER 10

Changelog

10.1 0.10dev

- Colored logs
- Improved testing coverage
- Neural network code was simplified considerably
- Added features to Neural networks objects
- Added neural networks to documentation
- Option to create all tensorflow sessions with a specific configuration
- Shortcut in command line to limit GPU memory in all tensorflow sessions (through gpu_options.per_process_gpu_memory_fraction)
- Refactored test suite
- Integrats new triage network (using Keras)

10.2 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

10.3 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)

10.4 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

10.5 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*

10.6 0.5 (2018-01-31)

- Improved logging
- Last release with old codebase

10.7 0.4 (2018-01-19)

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

10.8 0.3 (2017-11-15)

- Adds new neural network module

10.9 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

10.10 0.1.1 (2017-11-01)

- Minor changes to setup.py for uploading to pypi

10.11 0.1 (2017-11-01)

- First release

Python Module Index

y

Index

Y