hdnet Documentation

Release v0.1

Christopher Hillar, Felix Effenberger

July 16, 2015

CONTENTS

hdnet stands for Hopfield denoising network. It is a Python package for analysis of neural population spiking data, i.e. parallel spike trains.

In particular, it provides a novel method for finding and extracting salient low-dimensional representations of the dynamics of populations of spiking neurons based on a denoising approach to spatiotemporal patterns (STP) contained in the data.

An always up-to-date online-version of the documentation can be obtained at

http://team-hdnet.github.io/hdnet/

This manual also can be `downloaded in a PDF version`

Contents:

CHAPTER

ONE

INTRODUCTION

hdnet is a Python package for analysis of neural population spiking data, i.e. parallel spike trains.

In particular, it provides a novel method for finding and extracting salient low-dimensional representations of the dynamics of populations of spiking neurons based on a denoising approach to spatiotemporal patterns (STP) contained in the data.

Finding STP is classical problem in data analysis of parallel spike trains, and quite a number of approaches to detect and classify recurring spatiotemporal patterns (STP) of neural population activity were proposed [Gruen2010].

Yet, most published methods so far either focus solely on synchrony detection [Pipa2008], [PicadoMuino2013], [LopesDosSantos2013] or assume a more or less noiseless scenario, seeking to classify exactly recurring STP in neuronal activity (apart from allowing some jitter in spike timing), see e.g. [Gansel2012].

Given the usually high variability of population responses to stimuli, the re-occurrence of such exactly repeating STP becomes more and more unlikely with increasing population size, though. Assuming that despite this variability, network activity is not random per se (under the experimentally well-supported hypothesis [Abeles1993] [Arieli1995] that the population has to code information about stimuli in some form of STP), a much more plausible situation is that some underlying STP appears in several "corrupted" variants, both expressing jitter in spike times and differing in a few missing or excess spikes (characterized by a low Hamming distance to a true, underlying STP).

Our method takes a different aproach. Using Hopfield networks trained with *minimum probability flow* (MPF), the occuring raw spatiotemporal patterns are grouped into clusters of similar patterns in an unsupervised way, assigning to each cluster a *memory* (the fixed point of the Hopfield dynamics in each cluster).

The proposed method is robust to this variability in the signal and able to extract the underlying recurring patterns, even for seldomly occurring STP and large population sizes.

See Section *Mathematical background* for an introduction of the mathematical background behind some of the techniques used in *hdnet*.

CHAPTER

TWO

INSTALLING HDNET

TBD

2.1 Adding OpenBLAS support

The training process of Hopfield networks (from the classes `HopfieldNet` and `HopfieldNetMPF`) can be easily parallelized to exploit all available CPU cores by using a *numpy* installation that makes use of a multi-threaded linear algebra library such as *OpenBLAS*.

For many Linux installations though, the default linear algebra library used by *numpy* is not multithreaded.

For Debian based systems (such as Ubuntu), to compile *numpy* with support for OpenBLAS, a multithreaded linear algebra library, do the following (as described in the following Stackoverflow topic):

- Install some required packages

```
apt-get install build-essential gfortran python-dev git
```

- Create a temporary directory to compile the code

```
mkdir /tmp/openblas && cd /tmp/openblas
```

- Get and compile the OpenBLAS sources. Change the install path from */opt/OpenBLAS* in the example to your liking, also something like *~/.local/openblas* is possible.

```
git clone git://github.com/xianyi/OpenBLAS
cd OpenBLAS && make FC=gfortran
sudo make PREFIX=/opt/OpenBLAS install
sudo ldconfig
```

- Get the *numpy* sources

```
git clone https://github.com/numpy/numpy
cd numpy
```

- Adjust the build properties of numpy (to make it use the OpenBLAS library)

```
cp site.cfg.example site.cfg
```

  Now open your favorite editor (*vim*, is it? Or *emacs*?) and uncomment the following lines:

```
...
[openblas]
libraries = openblas
library_dirs = /opt/OpenBLAS/lib
include_dirs = /opt/OpenBLAS/include
...
```

- Compile numpy with OpenBLAS support (optionally add the *–user* flag to the last python call to install the package only for the current user not using root rights; the optional argument is marked with parentheses [] below)

```
python setup.py config
python setup.py build && python setup.py [--user] install
```

- Now you can test performance with the following script

```
import numpy as np
import sys
import timeit

try:
    import numpy.core._dotblas
    print 'fast BLAS'
except ImportError:
    print 'slow BLAS'

print 'version:', np.__version__
print 'maxint:', sys.maxint
print

x = np.random.random((1000,1000))

setup = 'import numpy as np; x = np.random.random((1000,1000))'
count = 10

t = timeit.Timer('np.dot(x, x.T)', setup=setup)
print 'dot:', t.timeit(count)/count, 'sec'
```

Save it as *dot_performance.py* and run the following, where *X* is the number of CPU cores *numpy* should use for linear algebra operations:

```
OMP_NUM_THREADS=X python dot_performance.py
```

You should see a nice speedup for higher values of *X*.

CHAPTER

THREE

CITING HDNET

If you use *hdnet* in your work please cite it using the following BibTeX entry.

BibTeX:

```
@online{hdnet,
    author      = {Hillar, Christopher  and Effenberger, Felix},
    title       = {hdnet -- a Python package for parallel spike train analysis},
    volume      = {abs/XXX},
    year        = {2015},
    eprinttype  = {arxiv},
    eprint      = {math/0307200v3}
    url         = {https://github.com/team-hdnet/hdnet}
    //ee        = {http://arxiv.org/abs/XXX},
}
```

CHAPTER

FOUR

MATHEMATICAL BACKGROUND

The following sections describe some of the underlying mathematics of methods used in *hdnet*.

4.1 Hopfield Networks

Hopfield network

TODO

Minimum probability flow (MPF)

TODO

4.2 Dichotomized Gaussian

TBD

CHAPTER

FIVE

BASIC EXAMPLE

Let us demonstrate the basic usage of `hdnet` using the following example. The source code for this example can be found in `this example script` in the `examples/` directory.

For demonstration purposes we will start work with a synthetic data set in this tutorial (later we will be working with real spiking data). Spiking activity of 10 hypothetical cells is assumed to be given as independent, identically distributed (i.i.d.) Poisson processes. Upon binning with a given bin width, this yields Bernoulli processes in discrete time. We create such Bernoulli data (as a proxy for real, binned spiking data) and then insert hypothetical correlations by means of a number of co-activations of different cell groups over time (also known as *cell assemblies*).

5.1 Starting off

We first import the necessary modules into our Python session (we recommend using ipython in pylab mode, i.e. running `ipython --pylab` and to run text copied to the clipboard from this tutorial using the magic command `%paste`):

```
import numpy as np
import matplotlib.pyplot as plt
from hdnet.stimulus import Stimulus
from hdnet.spikes import Spikes
from hdnet.spikes_model import SpikeModel, BernoulliHomogeneous, DichotomizedGaussian
```

Next, we create two trials of 200 time bins of spikes from 10 neurons and store them in a *Spikes* container:

```
# Let's first make up some simuilated spikes: 2 trials
spikes = (np.random.random((2, 10, 200)) < .05).astype(int)
spikes[0, [1, 5], ::5] = 1  # insert correlations
spikes[1, [2, 3, 6], ::11] = 1  # insert correlations

spikes = Spikes(spikes=spikes)
```

We can now plot a raster of the trials and covariances:

```
# let's look at the raw spikes and their covariance
plt.figure()
plt.matshow(spikes.rasterize(), cmap='gray')
plt.title('Raw spikes')

plt.figure()
plt.matshow(spikes.covariance().reshape((2 * 10, 10)), cmap='gray')
plt.title('Raw spikes covariance')
plt.show()
```

Next, we would like to model this noisy binary data. First, we try to model each trial with a separate i.i.d. Bernoulli random binary vector having the same neuron means as in each trial:

```
# let's examine the structure in spikes using a spike modeler
spikes_model = BernoulliHomogeneous(spikes=spikes)
BH_sample_spikes = spikes_model.sample_from_model()

plt.figure()
plt.matshow(BH_sample_spikes.rasterize(), cmap='gray')
plt.title('BernoulliHomogeneous sample')
print "%1.4f means" % BH_sample_spikes.spikes.mean()

plt.figure()
plt.matshow(BH_sample_spikes.covariance().reshape((2 * 10, 10)), cmap='gray')
plt.title('BernoulliHomogeneous covariance')

plt.show()
```

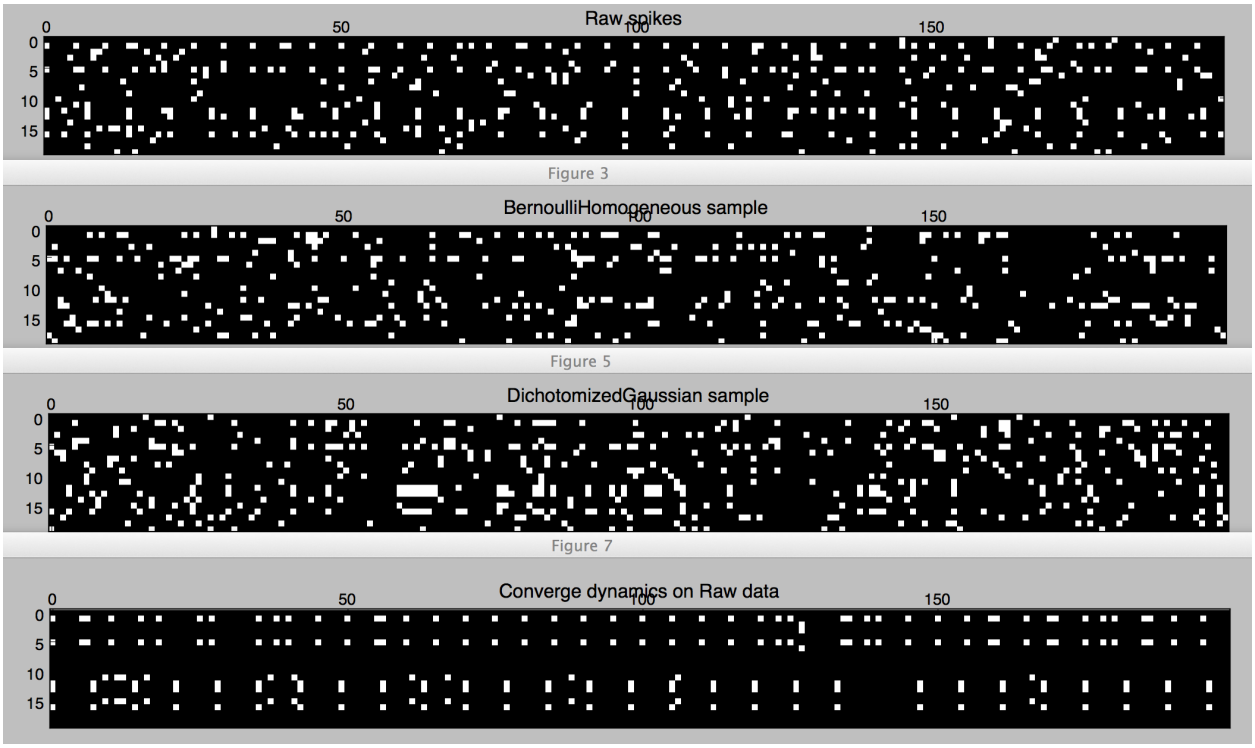

Fig. 5.1: Figure 1. Spikes of two trials with 10 neurons.

As we can see in Figures 1 and 2, the samples from Bernoulli have the correct firing rates in each trial, but not the coordinated aspect (as can be seen in the covariance matrices for each trial, which are basically diagonal matrices). A better model that keeps track of the correlations is the Dichotomized Gaussian [Bethge2008]:

```
# let's model them as DichotomizedGaussian:
# from the paper: Generating spike-trains with specified correlations, Macke et al.
spikes_model = DichotomizedGaussian(spikes=spikes)
DG_sample_spikes = spikes_model.sample_from_model()

plt.figure()
plt.title('DichotomizedGaussian sample')
plt.matshow(DG_sample_spikes.rasterize(), cmap='gray')
```

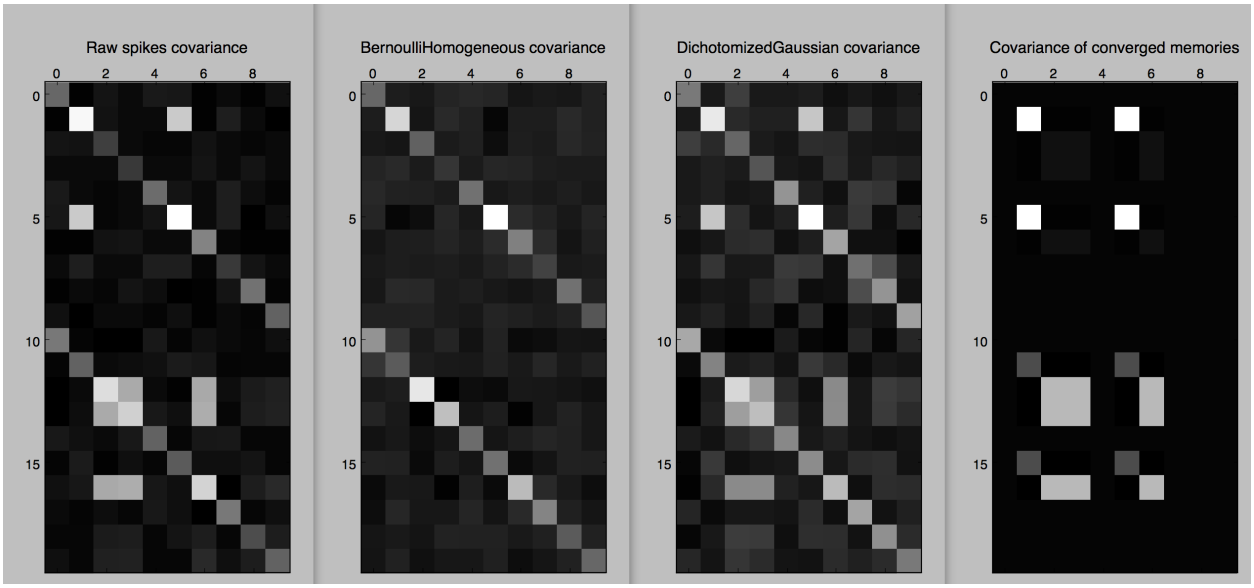

Fig. 5.2: Figure 2. Covariances of two trials with 10 neurons.

```
plt.figure()
plt.matshow(DG_sample_spikes.covariance().reshape((2 * 10, 10)), cmap='gray')
plt.title('DichotomizedGaussian covariance')

plt.show()
```

Finally, we try and model the data with a Hopfield network trained using MPF [HS-DK201] over all the trials:

```
# the basic modeler trains a Hopfield network using MPF on the raw spikes
spikes_model = SpikeModel(spikes=spikes)
spikes_model.fit()  # note: this fits a single network to all trials
spikes_model.chomp()

converged_spikes = Spikes(spikes=spikes_model.hopfield_spikes)

plt.figure()
plt.title('Converge dynamics on Raw data')
plt.matshow(converged_spikes.rasterize(), cmap='gray')

plt.figure()
plt.title('Covariance of converged memories')
plt.matshow(converged_spikes.covariance().reshape((2 * 10, 10)), cmap='gray')

plt.show()
```

5.2 Going further

One thing we would like to do is examine the structure of the memories:

```
# plot memory label (its chronological appearance) as a function of time
plt.figure()
plt.scatter(range(len(spikes_model.memories.sequence)), 1 + np.array(spikes_model.memories.sequence))
```

```
plt.xlabel('time bin')
plt.ylabel('Memory number (chronological order of appearance)')
plt.title('Converged memory label at each time bin')

# versus the raw data
plt.figure()
plt.scatter(range(len(spikes_model.empirical.sequence)), 1 + np.array(spikes_model.empirical.sequence
plt.ylabel('Raw pattern number (chronological order of appearance)')
plt.xlabel('time bin')
plt.title('Raw pattern label at each time bin')

plt.show()
```

Notice in Figures 4 and 4 that the converged dynamics of the trained Hopfield network on the original data does reveal the hidden assemblies for the most part.

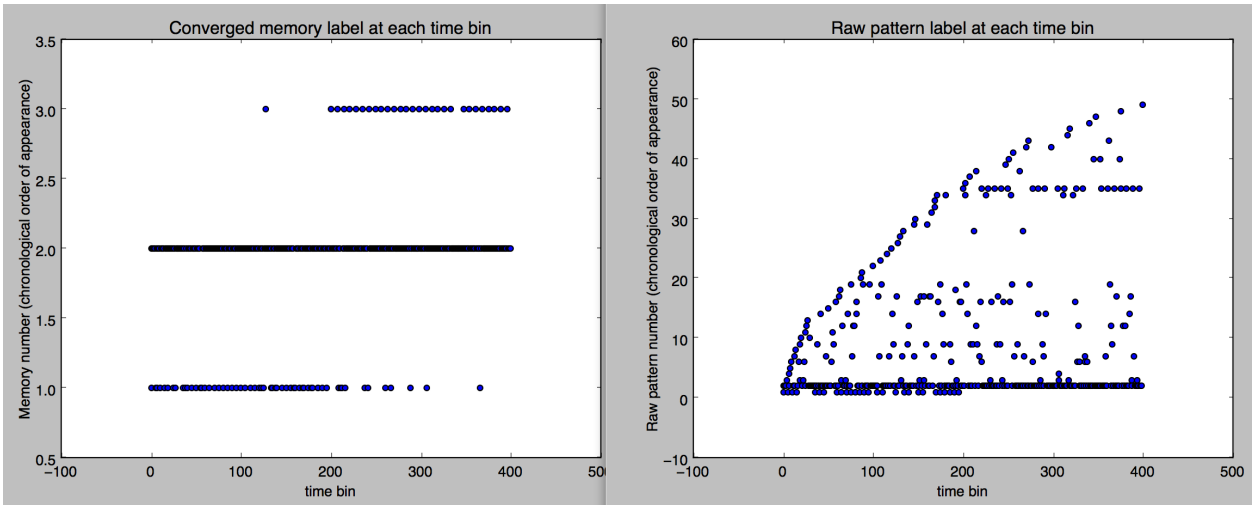

Fig. 5.3: Figure 3. Patterns (converged at left, raw on right) over time bins labeled on the vertical axis by their first appearance in the dataset.

Now that we know there are basically two assemblies, one showing up lots in the first trial and the other in the second, let's look at the memories and their corresponding *Memory Triggered Averages* MTAs that are obtained for each memory by averaging all raw patterns that converge to the given memory under the Hopfield dynamics.

The code below generates Fig. 2, which displaysa matrix whose first 3 columns are the memories in the network and whose next 3 columns are the average of raw data patterns converging to the corresponding memory in the first 3 columns:

```
# memories are ordered by their first appearance
bin_memories = spikes_model.memories.patterns
arr = np.zeros((spikes_model.original_spikes.N, 2 * len(bin_memories)))
for c, memory in enumerate(bin_memories):
    arr[:, c] = spikes_model.memories.fp_to_binary_matrix(c)

for c, memory in enumerate(bin_memories):
    arr[:, c + len(bin_memories)] = spikes_model.memories.mtas[memory] /
            spikes_model.memories.counts[memory]

print "Probabilities of each memory:"
print zip(bin_memories, spikes_model.memories.to_prob_vect())
```

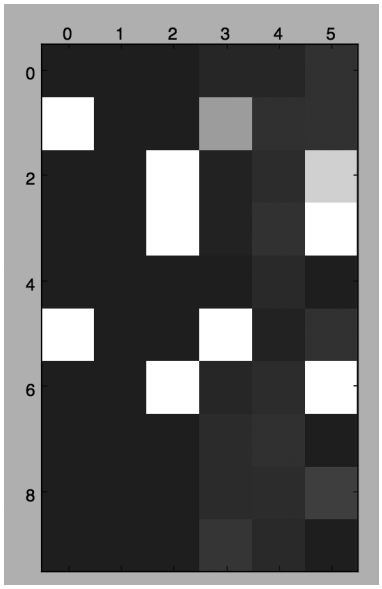

Fig. 5.4: Figure 4. Memories in network (left) and Memory Triggered Averages (at right)

```
# Probabilities of each memory:
# [('0100010000', 0.13), ('0000000000', 0.79249999999999998), /
# ('0011001000', 0.077499999999999999)]
```

Notice that the number of occurrences of the cell assembly with neuron 1 and 5 co-active is about double that of 2, 3, 6 co-active, consistent with our construction.

5.3 Saving and loading

One can save *Spikes*, `Learner`s and :class:.SpikesModel`s:

```
spikes_model.save('my_spikes_model')
loaded_spikes_model = SpikesModel.load('my_spikes_model')
```

Note that a `SpikesModel` already keeps track of the original spikes it was constructed from and all other internal objects (such as the Hopfield network).

5.4 Stimuli

Continuing our example, we now discuss how to incorporate stimuli into our analyses.

First, let's create a fake stimulus consisting of random normal 90 x 100 dimensional numpy arrays unless the fake stimulus is presented, in which case it is either a picture of Hobbes or Calvin (with some small noise added):

In code this looks like this:

```
from hdnet.stimulus import Stimulus

calvin = np.load('data/calvin.npy')  # 90 by 100 numpy array
hobbes = np.load('data/hobbes.npy')
```

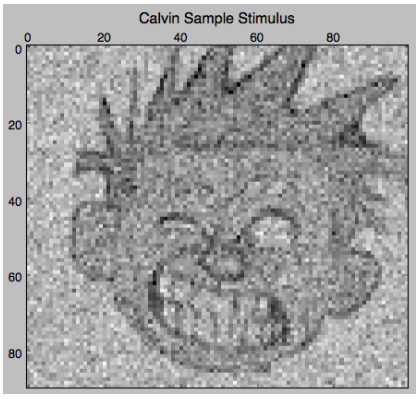

Fig. 5.5: Figure 5. Noisy stimulus: Calvin.

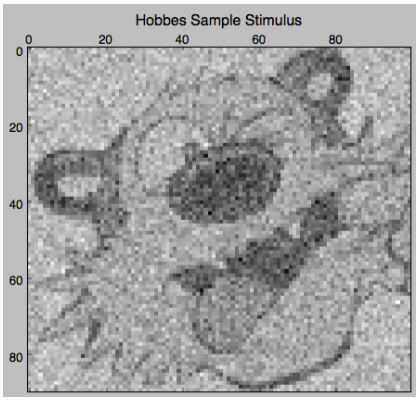

Fig. 5.6: Figure 6. Noisy stimulus: Hobbes.

```
stimulus_arr = 20 \* np.random.randn(2, 200, \*calvin.shape)
stimulus_arr[0, ::5] = calvin + 50 \* np.random.randn(200 / 5, \*calvin.shape)
stimulus_arr[1, ::11] = hobbes + 50 \* np.random.randn(200 / 11 + 1, /
                        \*hobbes.shape)

plt.matshow(stimulus_arr[0, 0], cmap='gray')
plt.title('Calvin Sample Stimulus')
plt.matshow(stimulus_arr[1, 0], cmap='gray')
plt.title('Hobbes Sample Stimulus')
```

Now, let's try and see what were the average stimuli for each fixed-point / memory. We call such features *Memory Triggered Stimulus Averages* (MTSA):

```
stimulus = Stimulus(stimulus_arr=stimulus_arr)
avgs = spikes_model.memories.mem_triggered_stim_avgs(stimulus)

for stm_avg in avgs:
    plt.figure()
    plt.matshow(stm_avg, cmap='gray')
    plt.title('Memory Triggered Stimulus Average')

plt.show()
```

The MTSAs look as following.

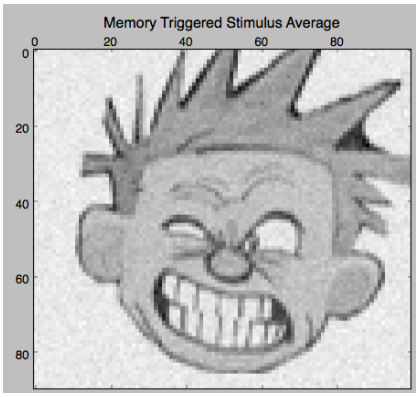

Fig. 5.7: Figure 7. Memory-triggered-stimulus averages of the Calvin spike pattern in the data.

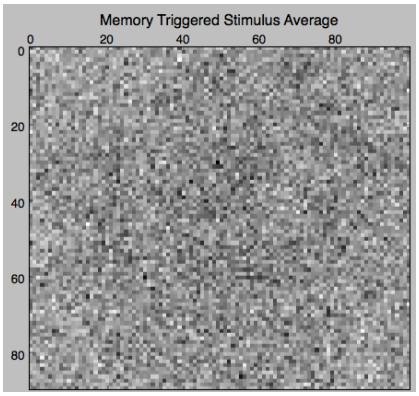

Fig. 5.8: Figure 8. Memory-triggered-stimulus averages of the empty spike pattern in the data.

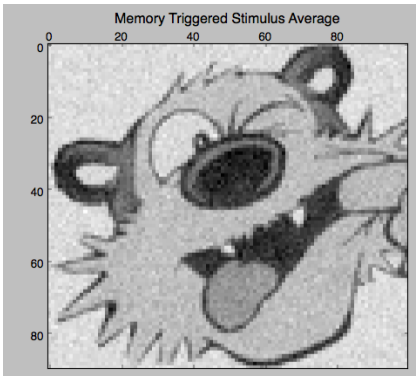

Fig. 5.9: Figure 9. Memory-triggered-stimulus averages of the Hobbes spike pattern in the data.

5.5 Real data

Now, we try these methods out on some real data. First, we download polytrode data recorded by Tim Blanche in the laboratory of Nicholas Swindale, University of British Columbia from the NSF-funded CRCNS Data Sharing website

Let's examine the spontaneous spiking data from anesthetized cat visual cortex area 18 (around 5 minutes of spike-sorted polytrode data from 50 neurons).

First we read the data using the *.spk* format reader integrated in *hdnet*:

```
from hdnet.data import SpkReader
fn = 'data/Blanche/crcns_pvc3_cat_recordings/drifting_bar/spike_data'
spikes = SpkReader.read_spk_folder(fn)
```

Now we fit a Hopfield network on the spike data:

```
spikes_model = SpikeModel(spikes=spikes)
spikes_model.fit()  # note: this fits a single network to all trials
```

After fitting the model we converge the windows of raw data to their Hopfield memories:

```
spikes_model.chomp()
converged_spikes = Spikes(spikes=spikes_model.hopfield_spikes)
```

We can now plot them and their covariance:

```
plt.matshow(converged_spikes.rasterize(), cmap='gray')
plt.title('Converge dynamics on Raw data')
plt.matshow(converged_spikes.covariance().reshape((2 * 10, 10)), cmap='gray')
plt.title('Covariance of converged memories')
```

TBC

CHAPTER

SIX

EXAMPLE: ANALYZING PATTERN SEQUENCES

In this example we will demonstrate further possibilities of examining sequences of patterns obtained from converging Hopfield dynamics on some windowed raw multi-neuron spiking data. This allows us to discover salient underlying dynamical structure in such data.

We again work with a synthetic data set, as in the basic example.

6.1 Creating a synthetic data set

Let us first create our synthetic data:

```
import numpy as np
from hdnet.spikes import Spikes
from hdnet.spikes_model import SpikeModel, BernoulliHomogeneous, DichotomizedGaussian

# Let's first make up some simuilated spikes: 100 trials
spikes = (np.random.random((50, 10, 200)) < .05).astype(int)
spikes[:, [1, 2, 5], 8 - 1::10] = 1  # insert correlations
spikes[:, [1, 4, 6], 9 - 1::20] = 1  # insert correlations
spikes[:, [2, 3, 6], 10 - 1::20] = 1  # insert correlations
spikes = Spikes(spikes=spikes)
```

Fig. 6.1: Figure 2. One trial of synthetic data.

6.2 Fitting a Hopfield network

Again, we fit a Hopfield network to windowed spike trains (window length 1) and collect the memories over the raw data:

```
# the basic modeler trains a Hopfield network using MPF on the raw spikes
spikes_model = SpikeModel(spikes=spikes)
spikes_model.fit()  # note: this fits a single network to all trials
spikes_model.chomp()
converged_spikes = Spikes(spikes=spikes_model.hopfield_spikes)
```

6.3 Examining the pattern sequence

Let us now examine the memory sequence of the converged patterns. First we instantiate a SequenceAnalyzer object on the pattern instance:

```
from hdnet.stats import SequenceAnalyzer
from hdnet.visualization import combine_windows, plot_graph

patterns = spikes_model.hopfield_patterns
sa = SequenceAnalyzer(patterns)
```

We can now compute label probabilities, their Markov transition probabilities and Markov entropies of the labels (defined as the entropy of the Markov transition probabilities for each label):

```
# compute probabilities of labels, markov transition probabilities and
label_probabilities = sa.compute_label_probabilities()
markov_probabilities = sa.compute_label_markov_probabilities()
label_entropy = sa.compute_label_markov_entropies()
n_labels = len(label_probabilities)
```

Let us now plot some of the quantities that we calcualted:

```
# plot label probabilities, markov transition probabilities and node entropy
fig, ax = plt.subplots()
ax.hist(label_probabilities, weights=[1. / n_labels] * n_labels,
    range=(label_probabilities.min(), label_probabilities.max()),
    bins=50, color='k')

ax.set_xlabel('probability')
ax.set_ylabel('fraction')
ax.set_yscale('log', nonposy='clip')
ax.set_xscale('log', nonposx='clip')
plt.tight_layout()
plt.savefig('label_probabilities.png')
plt.close()
```

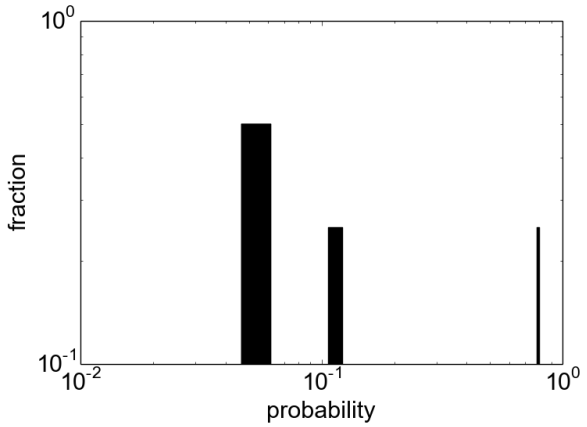

Fig. 6.2: Figure 1. Histogram of label probabilities on a log-log scale.

```
fig = plt.figure(figsize=(8,6))
ax = fig.add_subplot(1,1,1)
cmap = mpl.cm.autumn
```

```
cmap.set_bad('k')
mp_masked = np.ma.masked_where(markov_probabilities < 0.001 , markov_probabilities)
im = ax.matshow(mp_masked, cmap=cmap,
            norm=mpl.colors.LogNorm(vmin=0.001, vmax=1))

ax.set_xlabel('to pattern')
ax.set_ylabel('from pattern')
ax.xaxis.set_ticks([0, 3])
ax.yaxis.set_ticks([0, 3])
plt.colorbar(im)
plt.savefig('label_probabilities_markov.png')
plt.tight_layout()
plt.close()
```

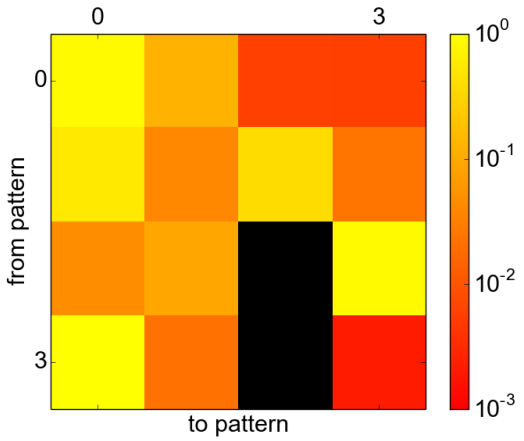

Fig. 6.3: Figure 2. Matrix of Markov transition probabilities between labels.

```
fig, ax = plt.subplots()
plt.hist(label_entropy,
         weights=[1. / n_labels] * n_labels, bins=50, color='k')
plt.xlabel('entropy')
plt.ylabel('fraction')
plt.yscale('log', nonposy='clip')
plt.tight_layout()
plt.savefig('label_entropy.png')
plt.close()
```

6.4 Constructing the Markov graph

The matrix of Markov transition probabilities defines a graph, the so called *Markov graph*. Let us construct and plot it using a force based layout for the nodes:

```
# construct markov graph
markov_graph = sa.compute_markov_graph()
print "Markov graph has %d nodes, %d edges" % (len(markov_graph.nodes()),
                                               len(markov_graph.edges()))

# plot markov graph
```

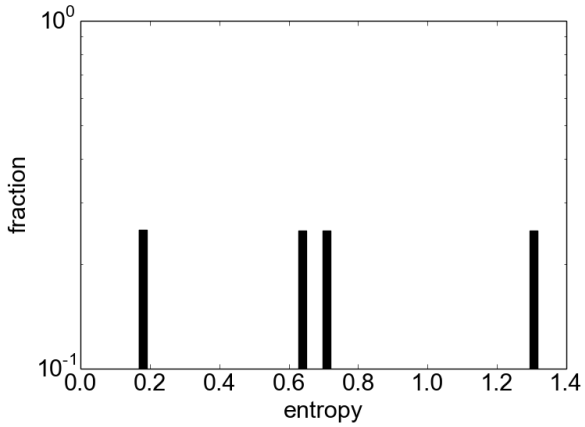

Fig. 6.4: Figure 3. Histogram of label entropies.

```
plot_graph(markov_graph, label_probabilities, cmap1='cool', cmap2='autumn')
plt.savefig('markov_graph.png')
```

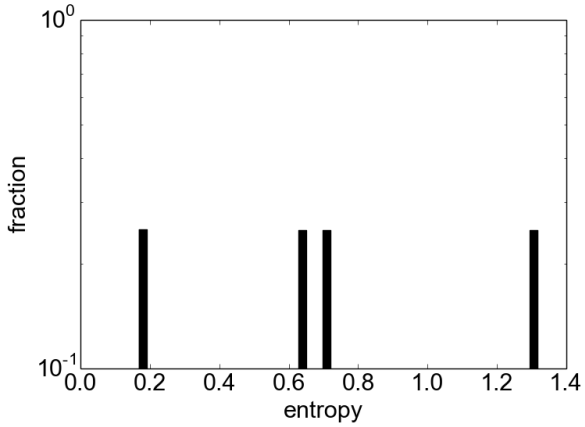

Fig. 6.5: Figure 4. Markov graph drawn with a force-based layout. The base state is 0.

Furthermore, we can plot the memory triggered averages for all the nodes of the graph (where each node corresponds to a Hopfield memory):

```
# plot memory triggered averages for all nodes of markov graph
fig, ax = plt.subplots(1, 4)
for i, node in enumerate(markov_graph.nodes()):
    ax = plt.subplot(1, 4, i + 1)
    ax.matshow(patterns.pattern_to_mta_matrix(node).reshape(10, 1),
               vmin=0, vmax=1, cmap='gray')
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

plt.savefig('mtas.png')
plt.close()
```

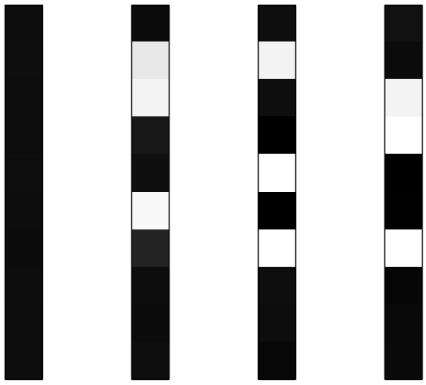

Fig. 6.6: Figure 5. Memory triggered averages of the nodes 0, 1, 2 and 3 (from left to right)

6.5 Indentifying base states

In many cases we will be able to identify one node in the graph that corresponds to the base state of the network; characteristic for a base state is that it has high degree (sum of in- and out-degrees) in the Markov graph:

```
# try to guess base node (resting state memory) as node with highest
# degree (converging and diverging connections)
# -- adjust post hoc if necessary!
markov_degrees = markov_graph.degree()
base_node = max(markov_degrees, key=markov_degrees.get)
print "base node is %d" % base_node
```

As you will see, the base node is 0 in this case.

6.6 Cycles as reliably produced network reponses

Now we calculate simple cycles (i.e. closed simple paths starting and ending at the same node) in the Markov graph starting at the base node. Each cycle can be thought of as a cycle in the state space of the network, corresponding to an excitation cycle of the network and describing how it is brought out of the base state, passing through a series of transient excited states to finally fall back into the base state. This essentially corresponds extracting several 1-dimensional aspects of the network dynamics.

As a measure for how reliably the network generates these cycles in the state space we use the Markov entropies of the nodes in the cycle: lower entropy of a memory means that the following state is more predictable, i.e. the path is more stably visited, whereas higher entropy means that the path is scattered when passing through a memory. We score all cycles by their entropy (where the entropy of a cycles is a weighted sum of the entropies of the nodes it consists of). The lower the entropy, the more stably that cycle occurrs in the data:

```
# calculate cycles of entropies around base node
# adjust weighting and weighting per element if needed
print "calculating cycles around base node.."
cycles, scores = sa.calculate_cycles_entropy_scores(
                                                base_node,
                                                min_len=2,
                                                max_len=20)
print "%d cycles" % (len(cycles))
```

Let is plot some statistics about the extracted cycles:

```
# plot cycle statistics
n_cycles = len(cycles)
cycle_len = np.array(map(len, cycles))
fig, ax = plt.subplots()
ax.hist(cycle_len, weights=[1. / n_cycles] * n_cycles, bins=50, color='k')
ax.set_xlabel('cycle length')
ax.set_ylabel('fraction')
plt.locator_params(nbins=3)
plt.tight_layout()
plt.savefig('cycle_lengths.png')
plt.close()
```

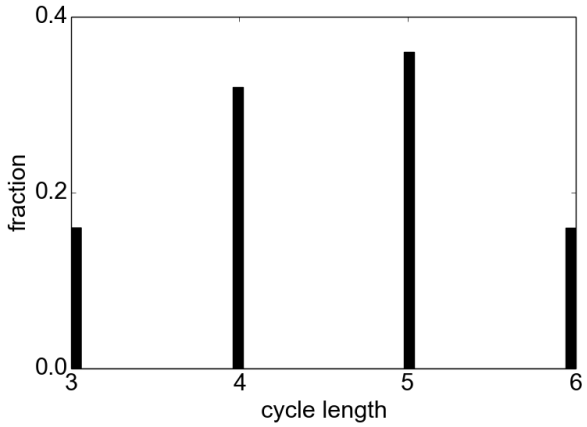

Fig. 6.7: Figure 6. Distribution of cycle lengths.

```
fig, ax = plt.subplots()
plt.hist(scores, weights=[1. / n_cycles] * n_cycles, bins=50, color='k')
plt.xlabel('cycle score')
plt.ylabel('fraction')
plt.locator_params(nbins=3)
plt.tight_layout()
plt.savefig('cycle_scores.png')
plt.close()
```

```
fig, ax = plt.subplots()
plt.scatter(cycle_len, scores, color='k')
plt.xlabel('cycle length')
plt.ylabel('cycle score')
plt.locator_params(nbins=3)
plt.tight_layout()
plt.savefig('cycle_lengths_vs_scores_scatter.png')
plt.close()
```

Let us now combine the memories of the cycles and plot the mean network response for each cycle:

```
for i, cycle in enumerate(cycles):
    mta_sequence = [patterns.pattern_to_mta_matrix(l).reshape(10, 1)
                    for l in cycle]
    combined = combine_windows(np.array(mta_sequence))
    fig, ax = plt.subplots()
```

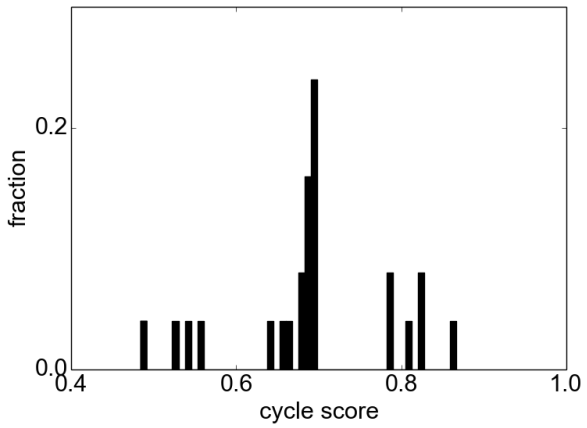

Fig. 6.8: Figure 7. Distribution of cycle scores.

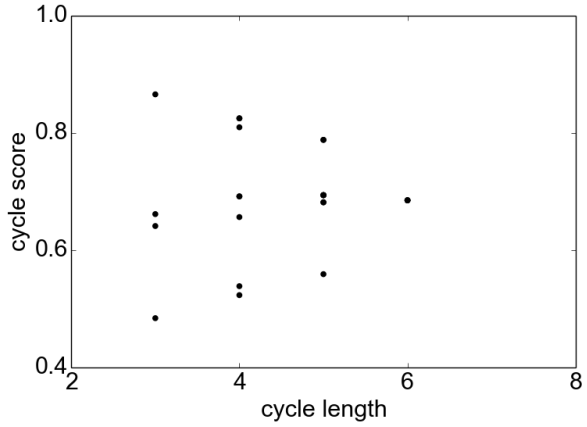

Fig. 6.9: Figure 8. Scatter plot of cycle lengths vs cycle scores.

```
    plt.matshow(combined, cmap='gray', vmin=0, vmax=1)
    plt.axis('off')
    plt.title('cycle %d\nlength %d\nscore %f' % \
              (i, len(cycle), scores[i]), loc='left')
    plt.savefig('likely-%04d.png' % i)
    plt.close()
```

As we can see, these responses exactly correspond to the sequence of cell assembly activations planted in the data. The method was thus able to extract these recurring sequences in noisy data.

CHAPTER

SEVEN

HDNET PACKAGE

hdnet stands for Hopfield denoising network. It is a Python package providing functionality for data analysis of parallel neural spike trains and methods for denoising and finding structure in such data sets.

7.1 hdnet.data

Data import and transformation functionality for hdnet. Contains import / read functionality for a number of different formats such as KlustaKwick and Matlab. Regarding Matlab files note the two different (incompatible) file formats that can be read with *MatlabReaderLegacy* and *MatlabReaderHDF5*, depending on the version of Matlab that you use.

class hdnet.data.**Binner**

Bases: object

Spike time binner class. Provides methods that take spike times as list of times and bin them.

Example: Combined use with KlustaKwick reader:

```
spikes_times = KlustaKwickReader.read_spikes(DIRECTORY, SAMPLING_RATE)
# bin to 1ms bins
binned_spikes_1ms = Binner.bin_spike_times(spikes_times, 0.001)
```

static bin_spike_times(*spike_times*, *bin_size*, *cells=None*, *t_min=None*, *t_max=None*)

Bins given spike_times into bins of size bin_size. Spike times expected in seconds (i.e. 1.0 for a spike at second 1, 0.5 for a spike happening at 500ms).

Takes optional arguments cells, t_min and t_max that can be used to restrict the cell indices (defaults to all cells) and time range (default t_min = minimum of all spike times in spike_times, default t_max = maximum of all spike times in spike_times).

Parameters

- **spike_times** (*2d numpy array*) – 2d array of spike times of cells, cells as rows
- **bin_size** (*float*) – bin size to be used for binning (1ms = 0.001)
- **cells** (*array_like, optional*) – indices of cells to process (default None, i.e. all cells)
- **t_min** (*float, optional*) – time of leftmost bin (default None)
- **t_max** (*float, optional*) – time of rightmost bin (default None)

Returns **spikes** – Spikes class containing binned spikes.

Return type *Spikes*

class hdnet.data.**KlustaKwickReader**
Bases: *hdnet.data.Reader*

Class for reading KlustaKwick data sets.

static read_spikes(*path_or_files*, *rate*, *first_cluster=2*, *filter_silent=True*, *return_status=False*)
Reader for KlustaKwick files.

Parameters

- **path_or_files** (*string*) – path of data set or list of *.res.* files to load
- **rate** (*float*) – sampling rate [in Hz]
- **discard_first_cluster** (*integer, optional*) – discard first n clusters, commonly used for unclassified spikes (default 2)
- **filter_silent** (*boolean, optional*) – filter out clusters that have no spikes (default True)
- **return_status** (*boolean, optional*) – if True returns a status dictionary along with data as second return value (default False)

Returns **spikes_times** – returns numpy array of spike times in all clusters. Float values represent spike times in seconds (i.e. a value of 1.0 represents a spike at time 1s)

Return type numpy array

class hdnet.data.**MatlabReaderHDF5**(*file_name*)
Bases: *hdnet.data.Reader*

Class for reading new Matlab .mat files created by Matlab versions >= 7.3.

Instantiate with `reader = MatlabReaderHDF5(file_name)`. List available Matlab objects with `get_keys()`. Access contents with `reader['NAME_OF_MATLAB_OBJECT']`. (The object will be converted to a numpy array. To access the underlying HD5 object, call *get_object_raw()*)

Note: This class needs the Python module `h5py`

Parameters **file_name** (*str*) – File name of Matlab file

close()
Closes the Matlab file if currently open.

Returns

Return type Nothing

get_hdf5()
Returns underlying HDF5 file object belonging to Matlab file.

Returns **file** – HDF5 file containing Matlab objects

Return type hdf5 file

get_object_numpy(*key*)
Returns object with given name `key` from the Matlab file in numpy format and `None` if no file currently open or an object with given name does not exist.

Parameters **key** (*str*) – Name of object to be loaded

Returns **object** – Matlab object as numpy array, or `None`

Return type numpy array or `None`

get_object_raw(*key*)
Returns object with given name `key` from the Matlab file in raw h5py representation and `None` if no file currently open or an object with given name does not exist.

Parameters **key** (*str*) – Name of object to be loaded

Returns **object** – Matlab object, or `None`

Return type h5py object

keys()
Returns names of Matlab objects in file, `None` if no file open.

Returns **keys** – List of Matlab objects in file

Return type list of str, or `None`

open(*file_name*)
Opens a Matlab file of HDF format (version >= 7.3). Do not forget to close the file with *close()* after reading its contents.

Parameters **file_name** (*str*) – Name of file to read

Returns **file** – Opened Matlab file

Return type `h5py.File` object

class hdnet.data.**MatlabReaderLegacy**(*file_name*)
Bases: *hdnet.data.Reader*

Class for reading legacy Matlab .mat files created by Matlab versions prior to 7.3

Instantiate with `reader = MatlabReaderLegacy(file_name)`. List available Matlab objects with *get_keys()*. Access contents with `reader['NAME_OF_MATLAB_OBJECT']` or `reader.get_object('NAME_OF_MATLAB_OBJECT')`.

Note: This class needs the Python module `scipy`

Parameters **file_name** (*str*) – File name of Matlab file

get_keys()
Returns names of Matlab objects in file.

Returns **keys** – List of Matlab objects in file

Return type list of str

get_object(*key*)
Returns a Matlab object with given name from the file and `None` if no such object exists.

Parameters **key** (*str*) – Name of Matlab object to retrieve

Returns **object** – Matlab object, `None` if no object with name `key` exists

Return type numpy array

get_objects()
Returns dictionary of all Matlab objects in file.

Returns **objects** – Dictionary of Matlab objects in file

Return type dictionary (key: object)

read(*file_name*)
Reads a Matlab file.

Parameters **file_name** (*str*) – Name of file to read

Returns **contents** – contents of file

Return type dict (key: object)

class hdnet.data.**Reader**
Bases: object

Abstract Reader class, all readers inherit from this class.

class hdnet.data.**SequenceEncoder**
Bases: object

Sequence encoder class. Provides methods that take spike times as list of times and extract firing sequences (just preserving the sequence and discarding other timing information).

Example: Combined use with KlustaKwick reader:

```
spikes_times = KlustaKwickReader.read_spikes(DIRECTORY, SAMPLING_RATE)
# calculate spikes sequence
spikes_sequence = Binner.get_spike_sequence(spikes_times)
```

static **get_spike_sequence**(*spike_times*, *cells=None*, *t_min=None*, *t_max=None*)
Extracts the firing sequence from the given spike times, i.e. a binary matrix S of dimension N x M where N is the number of neurons and M the total number of spikes in the data set. Each column of S contains exactly one non-zero entry, the index of the cell that spiked. Absolute spike timing information is discarded, spike order is preserved.

Takes optional arguments cells, t_min and t_max that can be used to restrict the cell indices (defaults to all cells) and time range (default t_min = minimum of all spike times in spike_times, default t_max = maximum of all spike times in spike_times).

Parameters

- **spike_times** (*array_like*) – 2d array of spike times of cells
- **cells** (*array_like, optional*) – indices of cells to process (default None, i.e. all cells)
- **t_min** (*float, optional*) – time of leftmost bin (default None)
- **t_max** (*float, optional*) – time of rightmost bin (default None)

Returns **sequence** – Spike sequence matrix S

Return type 2d numpy array of int

class hdnet.data.**SpkReader**
Bases: *hdnet.data.Reader*

Reader for spk file format. See CRCNS Tim Blanche data set.

Note: This class needs the bitstring module.

Warning: During testing we encountered errornous data on some Linux 64 bit installations. Take care.

static **load_from_spikes_times**(*spike_times_lists*, *bin_size=1*)
Loads a spike train from a list of arrays of spike times. The j-th item in the list corresponds to the j-th neuron. It is the 1d array of spike times (microsec) for that neuron.

Parameters

- **spike_times_lists** (*Type*) – Description

- **bin_size** (*int, optional*) – Bin size in milliseconds (default 1)

Returns **spikes** – numpy array containing binned spike times

Return type numpy array

static read_spk_files(*spk_files*, *bin_size=1*)

Loads spike times from a list of spk files. The j-th item in the list corresponds to the j-th neuron. It is the 1d array of spike times (microsec) for that neuron.

Parameters

- **spk_files** (*list of str*) – List of strings containing spk file names
- **bin_size** (*int, optional*) – Bin size in milliseconds (default 1)

Returns **spikes** – numpy array containing binned spike times

Return type numpy array

static read_spk_folder(*spk_folder*, *bin_size=1*)

Loads spike times from all spk files in a given folder. The j-th item in the list corresponds to the j-th neuron. It is the 1d array of spike times (microsec) for that neuron.

Parameters

- **spk_folder** (*str*) – Path containing spk file names
- **bin_size** (*int, optional*) – Bin size in milliseconds (default 1)

Returns **spikes** – numpy array containing binned spike times

Return type numpy array

7.2 hdnet.hopfield

Classes providing Hopfield network functionality, both dynamics and training.

class hdnet.hopfield.**HopfieldNet**(*N=None*, *J=None*, *theta=None*, *name=None*, *update='asynchronous'*, *symmetric=True*)

Bases: *hdnet.util.Restoreable*, object

Hopfield network class with binary hopfield dynamics [Hopfield, PNAS, 1982] Built in learning/training of network is outer product learning rule (OPR).

Parameters

- **N** (*int, optional*) – Number of nodes in network (default None)
- **J** (*numpy array, optional*) – Coupling matrix of size N x N, where N denotes the number of nodes in the network (default None)
- **theta** (*numpy array, optional*) – Thresholds vector of size N, where N denotes the number of nodes in the network (default None)
- **name** (*str, optional*) – Name of network (default None)
- **update** (*str, optional*) – Type of Hopfield dynamics update, "synchronous" or "asynchronous" (default "asynchronous")
- **symmetric** (*bool, optional*) – Symmetric coupling matrix (default True)

Returns **network** – Instance of *HopfieldNet* class

Return type *HopfieldNet*

J
Returns the N x N matrix (with N denoting the number of nodes in the network) of coupling strengths of nodes in the network, shortcut for *coupling_matrix()*.

Returns **J** – Coupling matrix of size N x N, where N denotes the number of nodes in the network

Return type numpy array

J_norm()
Returns vector of row 2-norms of coupling matrix J of Hopfield network.

Returns **norm** – Vector of 2-norms of coupling matrix

Return type 1d numpy array, float

N
Returns the number of nodes in the network, shortcut for *num_nodes()*.

Returns **n**

Return type int

__call__(*X*, *converge=True*, *max_iter=100000*, *clamped_nodes=None*)
Usage: network(X) returns the Hopfield dynamics update to patterns stored in rows of M x N matrix X.

If *converge* is False then 1 update run through the neurons is performed, otherwise Hopfield dynamics are run on X until convergence or *max_iter* iterations of updates are reached.

clamped_nodes is dictionary of those nodes not to update during the dynamics.

Parameters

- **X** (*numpy array*) – (M, N)-dim array of binary input patterns of length N, where N is the number of nodes in the network
- **converge** (*bool, optional*) – Flag whether to converge Hopfield dynamics. If False, just one step of dynamics is performed (default True)
- **max_iter** (*int, optional*) – Maximal number of iterations of dynamics (default 10 ** 5)
- **clamped_nodes** (*Type, optional*) – List of clamped nodes that are left untouched during dynamics update (default None)

Returns **patterns** – Converged patterns (memories) of Hopfield dynamics of input argument X

Return type numpy array

bits_recalled(*X*, *converge=True*)
Returns fraction of correctly recalled bits on input data *X*.

Parameters

- **X** (*2d numpy array, int*) – (M, N)-dim array of binary input patterns of length N, where N is the number of nodes in the network
- **converge** (*bool, optional*) – Flag whether to converge Hopfield dynamics. If False, just one step of dynamics is performed (default True)

Returns **recalled** – Fraction of correctly recalled bits

Return type float

compute_kappa(*X*)
Computes minimum marginal of dynamics update.

Parameters **X** (*numpy array*) – (M, N)-dim array of binary input patterns of length N, where N is the number of nodes in the network

Returns **Value** – Description

Return type Type

coupling_matrix
Returns the N x N matrix (with N denoting the number of nodes in the network) of coupling strengths of nodes in the network.

Returns **J** – Coupling matrix of size N x N, where N denotes the number of nodes in the network (default None)

Return type numpy array

energy(*x*)
Calculates the energy of a pattern x according to the Hopfield network.

The energy of a pattern x computes as:

$$E(x) = -\frac{1}{2} x^T \cdot [J - \text{diag}(J)] \cdot x + \theta \cdot x$$

Parameters **X** (*numpy array*) – (M, N)-dim array of binary input patterns of length N, where N is the number of nodes in the network

Returns **energy** – Energy of input pattern according to Hopfield network.

Return type float

exact_recalled(*X*, *converge=True*)
Returns fraction of raw patterns stored as memories in unmodified form.

Parameters

- **X** (*2d numpy array, int*) – (M, N)-dim array of binary input patterns of length N, where N is the number of nodes in the network
- **converge** (*bool, optional*) – Flag whether to converge Hopfield dynamics. If False, just one step of dynamics is performed (default True)

Returns **fraction** – Fraction of exactly stored memories

Return type float

hopfield_binary_dynamics(*X*, *update='asynchronous'*, *clamped_nodes=None*)
Applying Hopfield dynamics on X. Update can be "asynchronous" (default) or "synchronous". clamped_nodes is dict of those nodes *not* to change in the dynamics

Parameters

- **X** (*numpy array*) – (M, N)-dim array of binary input patterns of length N, where N is the number of nodes in the network
- **update** (*str, optional*) – Type of Hopfield dynamics update (default "asynchronous")
- **clamped_nodes** (*dict, optional*) – Dictionary of nodes to leave untouched during dynamics update (default None)

Returns **Value** – Description

Return type Type

learn_all(*X*, *disp=False*)
Learning M patterns in Hopfield network using outer product learning rule (OPR) [Hopfield, 82]

Interface method, calls *store_patterns_using_outer_products()*.

Parameters

- **X** (*numpy array*) – (M, N)-dim array of binary input patterns of length N to be stored, where N is the number of nodes in the network
- **disp** (*bool, optional*) – Display training log messages (default False)

Returns

Return type Nothing

learn_iterations
Returns number of iterations needed in training phase until convergence of network parameters.

Returns **iterations** – Number of iterations until convergence

Return type int

classmethod load(*file_name='hopfield_network'*, *load_extra=False*)
Loads Hopfield network from file.

Parameters

- **file_name** (*str, optional*) – File name to load network from (default 'hopfield_network')
- **load_extra** (*bool, optional*) – Flag whether to load extra file contents, if any (default False)

Returns **network** – Instance of *HopfieldNet* if loaded, *None* upon error

Return type *HopfieldNet*

neuron_order
Missing documentation

Returns **Value** – Description

Return type Type

num_hopfield_iter(*X*, *max_iter=100000*)
Returns array consisting of the number of Hopfield iterations needed to converge elements in *X* to their memories.

Parameters

- **X** (*numpy array*) – (M, N)-dim array of binary input patterns of length N, where N is the number of nodes in the network
- **max_iter** (*int, optional*) – Maximal number if iterations to perform per element (default 10 ** 5)

Returns **count** – Number of iterations performed for each element in *X*

Return type numpy array

num_nodes
Returns the number of nodes in the network.

Returns **n**

Return type int

reset()
Resets the network variables to base state (coupling strengths J, node thresholds theta and other status variables)

Returns

Return type Nothing

save (*file_name='hopfield_network'*, *extra=None*)
Saves Hopfield network to file.

Parameters

- **file_name** (*str, optional*) – File name to save network to (default 'hopfield_network')
- **extra** (*dict, optional*) – Extra information to save to file (default None)

Returns

Return type Nothing

store_patterns_using_outer_products (*X*)
Store patterns in X using outer product learning rule (OPR). Sets coupling matrix J.

Parameters **X** (*numpy array*) – (M, N)-dim array of binary input patterns of length N to be stored, where N is the number of nodes in the network

Returns

Return type Nothing

symmetric
Missing documentation

Returns **Value** – Description

Return type Type

theta
Returns a numpy vector of length N (with N denoting the number of nodes in the network) of thresholds for all nodes, shortcut for *thresholds()*.

Returns **J** – Coupling matrix of size N x N, where N denotes the number of nodes in the network

Return type numpy array

thresholds
Returns a numpy vector of length N (with N denoting the number of nodes in the network) of thresholds for all nodes.

Returns **J** – Coupling matrix of size N x N, where N denotes the number of nodes in the network

Return type numpy array

update
Returns update flag as string, indicating Hopfield update type. Can be 'synchronous; or 'asynchronous'.

Returns **update** – Update flag

Return type str

class hdnet.hopfield.**HopfieldNetMPF** (*N=None*, *J=None*, *theta=None*, *name=None*, *update='asynchronous'*, *symmetric=True*)
Bases: *hdnet.hopfield.HopfieldNet*

Hopfield network, with training using Minimum Probability Flow (MPF) (Sohl-Dickstein, Battaglino, Deweese, 2009) for training / learning of binary patterns

learn_all (*X*, *disp=False*)
Learn from M memory samples with Minimum Probability Flow (MPF)

Parameters

- **X** (*numpy array*) – (M, N)-dim array of binary input patterns of length N, where N is the number of nodes in the network

- **disp** (*bool, optional*) – Display scipy L-BFGS-B output (default False)

Returns

Return type Nothing

learn_from_sampler(*sampler*, *sample_size*, *batch_size=None*, *use_gpu=False*)
Learn from sampler

Parameters

- **sampler** (*Type*) – Description
- **sample_size** (*Type*) – Description
- **batch_size** (*Type, optional*) – Description (default None)
- **use_gpu** (*bool, optional*) – Description (default False)

Returns **Value** – Description

Return type Type

objective_function(*X*, *J=None*)
Note: accepts J with -2 theta on the diagonal Returns the MPF objective function evaluated over patterns X

Parameters

- **X** (*numpy array*) – (M, N)-dim array of binary input patterns of length N, where N is the number of nodes in the network
- **J** (*numpy array, optional*) – Coupling matrix of size N x N, where N denotes the number of nodes in the network (default None)

Returns **objective_func** – MPF objective function evaluated over patterns X

Return type numpy array

objective_function_batched(*sampler*, *sample_size*, *batch_size*, *randstate*, *J=None*)
This is to be able to fit network with more samples X than can be held in memory at once

Parameters

- **sampler** (*Type*) – Description
- **sample_size** (*Type*) – Description
- **batch_size** (*Type*) – Description
- **randstate** (*Type*) – Description
- **J** (*numpy array, optional*) – Coupling matrix of size N x N, where N denotes the number of nodes in the network (default None)

Returns **Value** – Description

Return type Type

objective_gradient(*X*, *J=None*, *return_K=False*)
Computes MPF objective gradient on input data X given coupling strengths J.

Parameters

- **X** (*numpy array*) – (M, N)-dim array of binary input patterns of length N, where N is the number of nodes in the network

- **J** (*numpy array, optional*) – Coupling matrix of size N x N, where N denotes the number of nodes in the network (default None)
- **return_K** (*bool, optional*) – Flag wether to return K (default False)

Returns **dJ [, K]** – Update to coupling matrix J [and K if return_K is True]

Return type numpy array [, numpy array]

objective_gradient_batched(*sampler*, *sample_size*, *batch_size*, *randstate*, *J=None*, *return_K=False*)

Missing documentation

Parameters

- **sampler** (*Type*) – Description
- **sample_size** (*Type*) – Description
- **batch_size** (*Type*) – Description
- **randstate** (*Type*) – Description
- **J** (*numpy array, optional*) – Coupling matrix of size N x N, where N denotes the number of nodes in the network (default None)
- **return_K** (*bool, optional*) – Description (default False)

Returns **Value** – Description

Return type Type

objective_gradient_minfunc(*J*, *X*)

Missing documentation

Parameters

- **J** (*Type*) – Description
- **X** (*numpy array*) – (M, N)-dim array of binary input patterns of length N, where N is the number of nodes in the network

Returns **Value** – Description

Return type Type

objective_gradient_minfunc_batched(*J*, *sampler*, *sample_size*, *batch_size*, *randstate*)

Missing documentation

Parameters

- **J** (*numpy array*) – Coupling matrix of size N x N, where N denotes the number of nodes in the network
- **sampler** (*Type*) – Description
- **sample_size** (*Type*) – Description
- **batch_size** (*Type*) – Description
- **randstate** (*Type*) – Description

Returns **Value** – Description

Return type Type

optcallback(*p*)

Missing documentation

Returns **Value** – Description

Return type Type

store_patterns_using_mpf(*X*, *disp=False*, ***kwargs*)
Stores patterns in X using Minimum Probability Flow (MPF) learning rule.

Parameters

- **X** (*numpy array*) – (M, N)-dim array of binary input patterns of length N, where N is the number of nodes in the network
- **disp** (*bool, optional*) – Display scipy L-BFGS-B output (default False)

Returns **status** – Dictionary containing status information

Return type dict

7.3 hdnet.learner

Class for learning hopfield network on spikes trains

class hdnet.learner.**Learner**(*spikes=None*, *network=None*, *network_file=None*, *window_size=1*, *params=None*)
Bases: *hdnet.util.Restoreable*, object

Takes spikes and learns a network on windowed patterns.

Parameters

- **spikes** (*Spikes*, optional) – Spikes class instance to use (default None)
- **network** (*HopfieldNet*, optional) – HopfieldNetwork class instance to use (default None)
- **network_file** (*str, optional*) – File name of Hopfield network to load (default None)
- **window_size** (*int, optional*) – Size of window in bins (default 1)
- **params** (*dict, optional*) – Dictionary of optional parameters (default None)

Returns **learner** – Instance of class *Learner*

Return type *Learner*

learn_from_binary(*X*, *remove_zeros=False*, *disp=False*)
Trains on M x N matrix X of M N-length binary vects

Parameters

- **X** (*numpy array*) – (M, N)-dim array of binary input patterns of length N, where N is the number of nodes in the network
- **remove_zeros** (*bool, optional*) – Flag whether to remove vectors from X in which all entries are 0 (default True)
- **disp** (*bool, optional*) – Display scipy L-BFGS-B output (default False)

Returns

Return type Nothing

learn_from_spikes(*spikes=None*, *window_size=1*, *trials=None*, *remove_zeros=True*, *disp=False*)
Trains network over spikes contained in instance of *Spikes* class.

Parameters

- **spikes** (*Spikes*, optional) – Instance of Spikes class (default None)
- **window_size** (*int, optional*) – Window size to use (default 1)
- **trials** (*Type, optional*) – Description (default None)
- **remove_zeros** (*bool, optional*) – Flag whether to remove windows in which all entries are 0 (default True)
- **disp** (*bool, optional*) – Display scipy L-BFGS-B output (default False)

Returns

Return type Nothing

learn_from_spikes_rot(*spikes=None*, *window_size=1*, *trials=None*, *remove_zeros=True*, *disp=False*)

Trains network over spikes contained in instance of *Spikes* class, removes windows that are identical modulo a rotation along the first axis.

Parameters

- **spikes** (*Spikes*, optional) – Instance of Spikes class (default None)
- **window_size** (*int, optional*) – Window size to use (default 1)
- **trials** (*Type, optional*) – Description (default None)
- **remove_zeros** (*bool, optional*) – Flag whether to remove windows in which all entries are 0 (default True)
- **disp** (*bool, optional*) – Display scipy L-BFGS-B output (default False)

Returns

Return type Nothing

classmethod load(*folder_name='learner'*, *load_extra=False*)

Loads Learner from file.

Parameters

- **folder_name** (*str, optional*) – Folder name name to load Learner from (default 'learner')
- **load_extra** (*bool, optional*) – Flag whether to load extra file contents, if any (default False)

Returns **learner** – Instance of *Learner* if loaded, *None* upon error

Return type *Learner*

network

Getter for hopfield network of this learner.

Returns **network** – Network of this learner

Return type *HopfieldNet*

params

Getter for parameters of learner.

Returns **parameters** – Parameters of learner

Return type dict

save (*folder_name='learner'*)
Saves Learner to file. Also saves the contained instance of *HopfieldNet*.

Parameters **folder_name** (*str, optional*) – Folder name name to save Learner to (default 'learner')

Returns

Return type Nothing

spikes
Setter for spikes.

Returns **value** – Instance of *Spikes* class

Return type *Spikes*

spikes_file
Getter for spikes file.

Returns **file** – File name of spikes file

Return type str

window_size
Getter for window size.

Returns **value** – Value of window size

Return type int

7.4 hdnet.math

Miscalleaneous mathematical functions for hdnet.

hdnet.maths.**heaviside** (*X*, *dtype=None*)
Heaviside function: given M x N numpy array, return points-wise Heaviside:

$$H(r) = 1 \text{ if } r > 0, \text{ else } 0$$

Parameters

- **X** (*array_like*) – Description
- **dtype** (*Type, optional*) – numpy data type of returned array if None, type is int (default None)

Returns **H** – Array with entries of X heavisided

Return type numpy array

7.5 hdnet.patterns

Record / counts of fixed-points of Hopfield network.

class hdnet.patterns.**Counter** (*counter=None*, *save_sequence=True*)
Bases: *hdnet.util.Restoreable*, object

Catalogues binary vectors and their prevalence.

Parameters

- **counter** (*Counter*, optional) – Counter object to merge with (default None)
- **save_sequence** (*bool, optional*) – Flag whether to save the sequence of pattern labels, labels given by order of appearance (default True)

Returns **counter** – Instance of class *Counter*

Return type *Counter*.

__add__(*other*)
Merges counts of another *Counter* object into this instance. Calls *merge_counts()*.

Parameters **other** (*Counter*) – Other Counter object

Returns **counter** – This instance

Return type *Counter*

__len__()
Returns number of distinct patterns in this Counter.

Returns **length** – Number of distinct patterns

Return type int

add_key(*key*, *value=1*, ***kwargs*)
Adds a new key (pattern) to the collection.

Parameters

- **key** (*str of '0', '1'*) – Key of pattern to add, obtained from *Counter.key_for_pattern()*
- **value** (*int, optional*) – Number of occurrences to add (default 1)
- **raw** (*2d numpy array, int, optional*) – Raw pattern that converged to given memory (default None)

Returns **added** – Flag whether key was previously known

Return type bool

chomp(*X*, *add_new=True*, *rotate=None*)
Counts patterns occurring as row vectors of N x M input matrix *X* and stores them. Calls *chomp_vector()* on row vectors of *X*.

Parameters

- **X** (*M x N numpy array, int*) – Binary source data.
- **add_new** (*bool, optional*) – Flag whether to store new memories (default True)
- **rotate** (*tuple of length 2, int, optional*) – Dimensions of window if patterns are to be collected modulo window rotations (default None)

Returns

Return type Nothing

chomp_spikes(*spikes*, *add_new=True*, *window_size=1*, *trials=None*, *rotate=None*)
Counts and stores patterns occurring over *Spikes* class using a sliding window of size *window_size*.

Parameters

- **spikes** (*Spikes*) – Instance of *Spikes* to operate on
- **window_size** (*int, optional*) – Window size to use (default 1)

- **trials** (*int, optional*) – Number of trials to use for reshape (default None)
- **reshape** (*bool, optional*) – Flag whether to reshape the spike vectors into matrix form before returning (default True)
- **rotate** (*tuple of length 2, int, optional*) – Dimensions of window if patterns are to be collected modulo window rotations (default None)

Returns **counter** – Returns pointer to itself

Return type *Counter*

chomp_vector(*x*, *add_new=True*, *rotate=None*)

Counts occurrences of pattern in vector *x*, assigns it a integer label and stores it.

Parameters

- **x** (*1d numpy array, int*) – Binary source vector.
- **add_new** (*bool, optional*) – Flag whether to store new memories (default True)
- **rotate** (*tuple of length 2, int, optional*) – Dimensions of window if patterns are to be collected modulo window rotations (default None)

Returns **bin_x, new_pattern, numrot** – Key of pattern *x*, Flag whether pattern was seen before, number of rotations performed to obtain pattern identity (if *rotate* was given)

Return type str, bool, int

counts

Returns the counts of each pattern encountered in the raw data.

Returns **counts** – Dictionary of counts of all patterns, indexed by pattern key

Return type dict

counts_by_label

Returns the counts of each pattern encountered in the raw data.

Returns **counts** – Counts of all patterns, indexed by label

Return type 1d numpy array, int

static key_for_pattern(*pattern*)

Computes key (as string) of binary pattern *pattern*. Reverse loopup for method *pattern_for_key()*.

Returns **key** – String representation of binary pattern

Return type str

classmethod load(*file_name='counter'*, *load_extra=False*)

Loads contents from file.

Parameters

- **file_name** (*str, optional*) – File name to load from (default 'counter')
- **load_extra** (*bool, optional*) – Flag whether to load extra file contents, if any (default False)

Returns **counter** – Instance of *Counter* if loaded, *None* upon error

Return type *Counter*

classmethod load_legacy(*file_name='counter'*)

lookup_patterns
Returns the lookup dictionary for the patterns, mapping a string representation of a pattern to a vector representation.

Returns **lookup** – Lookup dictionary

Return type dict

mem_triggered_stim_avgs(*stimulus*)
Returns the average stimulus appearing when a given binary pattern appears.

Parameters **stimulus** (*Stimulus*) – Instance of *Stimulus* class to query

Returns **averages** – Stimulus average calculated

Return type numpy array

merge_counts(*counter*)
Merges counts of another *Counter* object into this instance.

Parameters **other** (*Counter*) – Other Counter object

Returns **counter** – This instance

Return type *Counter*

num_patterns
Returns the number of patterns encountered in the raw data.

Returns **N** – number of distinct patterns in the raw data

Return type int

pattern_correlation_coefficients(*labels=None*, ***kwargs*)
Calculates the matrix of correlation coefficients between memories.

Takes optional argument labels that allows to restrict the selection of patterns to a subset of all memories. Entries in labels have to be in the closed interval [0, self.num_patterns - 1].

Parameters

- **labels** (*array_like, int*) – Labels of patterns to consider
- **kwargs** (*dictionary*) – Additional arguments passed to np.corrcoef

Returns **C** – Matrix of normalized pairwise correlation coefficients

Return type 2d numpy array

static pattern_distance_hamming(*a*, *b*)
Computes a distance measure for two binary patterns based on their normed Hamming distance, defined as

$$d_H(a,b) = \frac{1}{n} |\{j \in \{1,\ldots,n\} \mid a_j \neq b_j\}|,$$

if both a and b have length n.

The similarity measure takes values on the closed interval [0, 1], where a value of 0 is attained for disjoint, i.e. maximally dissimilar patterns a and b and a value of 1 for the case of $a = b$.

Parameters

- **a** (*list or array, int or bool*) – Input pattern
- **b** (*list or array, int or bool*) – Input pattern

Returns **dist** – Normed Hamming distance between a and b.

Return type double

static **pattern_distance_jaccard**(*a*, *b*)

Computes a distance measure for two binary patterns based on their Jaccard-Needham distance, defined as

$$d_J(a,b) = 1 - J(a,b) = \frac{|a \cup b| - |a \cap b|}{|a \cup b|}.$$

The similarity measure takes values on the closed interval [0, 1], where a value of 1 is attained for disjoint, i.e. maximally dissimilar patterns a and b and a value of 0 for the case of $a = b$.

Parameters

- **a** (*list or array, int or bool*) – Input pattern
- **b** (*list or array, int or bool*) – Input pattern

Returns **dist** – Jaccard distance between *a* and *b*.

Return type double

static **pattern_for_key**(*key*)

Computes binary pattern (as numpy matrix) from string representation *key*. Reverse loopup for method *key_for_pattern()*.

Returns **pattern** – binary pattern (as numpy matrix)

Return type numpy array

pattern_to_binary_matrix(*key*)

Returns a binary matrix representation of a pattern with the given key (as string of binary numbers).

Parameters **key** (*str*) – Key of pattern

Returns **pattern** – Representation of pattern as binary vector

Return type 1d numpy array

patterns

Returns the patterns encountered in the raw data as 1d vectors.

Returns **patterns** – Binary array of patterns encountered in the raw data, as 1d vectors

Return type 2d numpy array, int

save(*file_name='counter'*, *extra=None*)

Saves contents to file.

Parameters

- **file_name** (*str, optional*) – File name to save to (default 'counter')
- **extra** (*dict, optional*) – Extra information to save to file (default None)

Returns

Return type Nothing

seen_sequence

Returns the sequence of seen flags for the patterns over the raw data. Each entry is binary and has a value of 1 if the pattern at this position occurred previously already.

Returns **seen** – Sequence of seen flags

Return type 1d numpy array

sequence
Returns the sequence of patterns labels as encountered in the raw data. Pattern labels are allocated as integer numbers starting from 0 over the input data. Whenever a pattern was not encountered before, a new label is allocated.

Returns **sequence** – Sequence of pattern labels over raw data

Return type 1d numpy array, int

skipped_patterns
Returns a binary vector signalling when a pattern was skipped due to rotation symmetry.

Returns **skipped** – Skipped patterns indicator

Return type 1d numpy array

top_binary_matrices(*m*)
Returns the top *m* likely patterns.

Parameters **m** (*int*) – Number of top likely patterns to return

Returns **patterns** – *m* top likely patterns

Return type numpy array

class hdnet.patterns.**PatternsHopfield**(*learner=None*, *patterns_hopfield=None*, *save_sequence=True*, *save_raw=True*)
Bases: *hdnet.patterns.Counter*

Catalogues Hopfield fixed points of binary vectors and their prevalence in raw spiking data, optionally keeping references to the raw data. Subclass of *Counter*, extending its functionality.

Parameters

- **learner** (*Learner*, optional) – Learner instance to use that holds the underlying Hopfield Network (default None)
- **patterns_hopfield** (*PatternsHopfield*, optional) – Hopfield Patterns class to merge with (default None)
- **save_sequence** (*bool, optional*) – Flag whether to save the sequence of pattern labels with labels given by natural numbers in order of appearance (default True)
- **save_raw** (*bool, optional*) – Flag whether to save the raw patterns that converge to each memory under the Hopfield dynamics (default True)

Returns **patterns** – Instance of class *PatternsHopfield*

Return type *PatternsHopfield*.

add_key(*key*, *value=1*, *raw=None*)
Adds a new key (pattern) to the collection.

Parameters

- **key** (*str of '0', '1'*) – Key of memory to add, obtained from *Counter.key_for_pattern()*
- **value** (*int, optional*) – Number of occurrences to add (default 1)
- **raw** (*2d numpy array, int, optional*) – Raw pattern that converged to given memory (default None)

Returns **added** – Flag whether key was previously known

Return type bool

apply_dynamics (*spikes*, *window_size=1*, *trials=None*, *reshape=True*)
Computes Hopfield fixed points over data obtained from *spikes* using a sliding window of size *window_size*.

Parameters

- **spikes** (*Spikes*) – Instance of *Spikes* to operate on
- **window_size** (*int, optional*) – Window size to use (default 1)
- **trials** (*int, optional*) – Number of trials to use for reshape (default None)
- **reshape** (*bool, optional*) – Flag whether to reshape the spike vectors into matrix form before returning (default True)

Returns **spikes** – Instance of spikes class with converged spikes

Return type *Spikes*

approximate_basin_size (*max_corrupt_bits=1*)
Average bits corruption a memory can stand.

Parameters **max_corrupt_bits** (*int, optional*) – Maximal number of corrupted bits to try (default 1)

Returns **basin_sizes** – Approximated basin sizes of each memory

Return type numpy array

chomp (*X*, *add_new=True*, *rotate=None*)
Computes Hopfield fixed points of M rows in N x M input matrix *X* using stored Hopfield network and stores the memories. Calls *chomp_vector()* on row vectors of *X*. The number of columns N has to equal the number of nodes in the underlying Hopfield network.

Parameters

- **X** (*M x N numpy array, int*) – Binary source data to converge.
- **add_new** (*bool, optional*) – Flag whether to store new memories (default True)
- **rotate** (*tuple of length 2, int, optional*) – Dimensions of window if patterns are to be collected modulo window rotations (default None)

Returns

Return type Nothing

chomp_vector (*x*, *y*, *add_new=True*, *rotate=None*)
Associates binary raw data *x* with its Hopfield memory *y*, counting occurrences and storing raw data for the calculation of memory triggered averages (the average of all raw patterns in the data coverging to a given memory).

Parameters

- **x** (*1d numpy array, int*) – Binary source data to converge.
- **y** (*1d numpy array, int*) – Binary converged data.
- **add_new** (*bool, optional*) – Flag whether to store new memories (default True)
- **rotate** (*tuple of length 2, int, optional*) – Dimensions of window if patterns are to be collected modulo window rotations (default None)

Returns

Return type Nothing

classmethod load(*file_name='patterns_hopfield'*, *load_extra=False*)
Loads contents from file.

Parameters

- **file_name** (*str, optional*) – File name to load from (default 'patterns_hopfield')
- **load_extra** (*bool, optional*) – Flag whether to load extra file contents, if any (default False)

Returns **patterns** – Instance of *PatternsHopfield* if loaded, *None* upon error

Return type *PatternsHopfield*

classmethod load_legacy(*file_name='patterns_hopfield'*)

merge_counts(*patterns_hopfield*)
Combines counts with another PatternsHopfield class.

Parameters **patterns_hopfield** (*PatternsHopfield*) – Other *PatternsHopfield* class to merge counts with

Returns **patterns** – Returns pointer to itself

Return type *PatternsHopfield*

mtas
Returns the memory triggered averages (MTAs) of all stored memories.

Returns a Python dictionary keys of which are strings of binary digits representing the memory (the original memory can be obtained from the key using *Counter.pattern_for_key()*) and values are 2d numpy arrays representing the memory triggered average.

Returns **mtas** – Dictionary of MTAs of all stored memories

Return type dict

mtas_raw
Returns the set of all raw patterns encountered that converged to each stored memory. For each memory, the average of those patterns corresponds to its memory triggered average (MTA).

Returns a Python dictionary keys of which are strings of binary digits representing the memory (the original memory can be obtained from the key using *Counter.pattern_for_key()*) and values are lists of 2d numpy arrays representing raw patterns that converge to the given memory.

Returns **raw_patterns** – Dictionary of lists of raw patterns converging to a given memory

Return type dict

pattern_to_mta_matrix(*label*)
Returns the average of all raw patterns encountered that converged to a given stored memory pattern with label *label*. This average is called memory triggered average (MTA).

Parameters **label** (*int*) – Label of pattern to look up

Returns **mta** – MTA of memory with label *label*

Return type 1d numpy array

pattern_to_mtv(*m*)
Returns the element-wise variance of each position in a pattern across all raw patterns encountered that converged to a given stored memory pattern with label *label*. This is called the *Memory Triggered Variance* (MTV). It is a meansure for how diverse the underlying patterns are converging to the same memory and can be seen as a proxy for the basin size of that memory (i.e. fixed point) under the given Hopfield dynamics.

Parameters **label** (*int*) – Label of pattern to look up

Returns **mtv** – Element-wise variance

Return type 1d numpy array

pattern_to_raw_patterns(*label*)
Returns the list of all raw patterns encountered that converged to a given stored memory pattern with label *label*.

Parameters **label** (*int*) – Label of pattern to look up

Returns **raw** – Raw patterns converging to memory with label *label*

Return type list of 1d numpy array

pattern_to_trial_raster(*label*, *start=0*, *stop=None*, *trials=None*)
Returns binary matrix signalling when the memory with the given label *label* apprears in the data.

Parameters

- **label** (*int*) – Label of pattern to look up
- **start** (*int, optional*) – First index in each trial (default 0)
- **stop** (*int, optional*) – Last index in each trial, if None whole trial will be used (default None)
- **trials** (*int, optional*) – Number of trials, if None taken from underlying *Learner* class (default None)

Returns **hits** – Binary array with a value of 1 encoding the occurrence of the pattern with the given label *label*

Return type 2d numpy array

save(*file_name='patterns_hopfield'*, *extra=None*)
Saves contents to file.

Parameters

- **file_name** (*str, optional*) – File name to save to (default 'patterns_hopfield')
- **extra** (*dict, optional*) – Extra information to save to file (default None)

Returns

Return type Nothing

top_mta_matrices(*count*)
Returns a list of memory triggered averages (MTAs) of the memories occurring the most in the encountered data.

Parameters **count** (*int*) – Number of mostly occurring memories to consider

Returns **mtas** – Array of MTAs belonging to top occurring memories

Return type 2d numpy array

class hdnet.patterns.**PatternsRaw**(*patterns_raw=None*, *save_sequence=True*)
Bases: *hdnet.patterns.Counter*

Catalogues binary vectors and their prevalence in raw spiking data. Subclass of *Counter*, extending its functionality.

Parameters

- **patterns_raw** (*PatternsRaw*, optional) – Patterns object to merge with (default None)
- **save_sequence** (*bool, optional*) – Flag whether to save the sequence of pattern labels, labels given by order of appearance (default True)

Returns **patterns** – Instance of class *PatternsRaw*

Return type *PatternsRaw*.

classmethod load(*file_name='patterns_raw'*, *load_extra=False*)
Loads contents from file.

Parameters

- **file_name** (*str, optional*) – File name to load from (default 'patterns_raw')
- **load_extra** (*bool, optional*) – Flag whether to load extra file contents, if any (default False)

Returns **patterns** – Instance of *PatternsRaw* if loaded, *None* upon error

Return type *PatternsRaw*

save(*file_name='patterns_raw'*, *extra=None*)
Saves contents to file.

Parameters

- **file_name** (*str, optional*) – File name to save to (default 'patterns_raw')
- **extra** (*dict, optional*) – Extra information to save to file (default None)

Returns

Return type Nothing

7.6 hdnet.sampling

Some simple routines for sampling from certain distributions.

hdnet.sampling.**dg_second_moment**(*u*, *gauss_mean1*, *gauss_mean2*, *support1*, *support2*)
Missing documentation

Parameters

- **u** (*Type*) – Description
- **gauss_mean1** (*Type*) – Description
- **gauss_mean2** (*Type*) – Description
- **support1** (*Type*) – Description
- **support2** (*Type*) – Description

Returns **Value** – Description

Return type Type

hdnet.sampling.**energy**(*J*, *theta*, *x*)

Ising Energy of binary pattern x is: Ex = -.5 x^T[J-diag(J)]x + theta*x

Parameters

- **J** (*Type*) – Description
- **theta** (*Type*) – Description
- **x** (*Type*) – Description

Returns **Value** – Description

Return type Type

hdnet.sampling.**find_dg_any_marginal**(*pmfs*, *bin_cov*, *supports*, *accuracy=1e-10*)

[gammas,Lambda,joints2D] = FindDGAnyMarginal(pmfs,Sigma,supports) Finds the paramters of a Multivariate Discretized Gaussian with specified marginal distributions and covariance matrix

Inputs: pmfs: the probability mass functions of the marginal distribution of the input-random variables. Must be a cell-array with n elements, each of which is a vector which sums to one Sigma: The covariance matrix of the input-random variable. The function does not check for admissability, i.e. results might be wrong if there exists no random variable which has the specified marginals and covariance. supports: The support of each dimension of the input random variable. Must be a cell-array with n elements, each of whcih is a vector with increasing entries giving the possible values of each random variable, e.g. if the first dimension of the rv is 1 with probability .2, 3 with prob .8, then pmfs{1}=[.2,.8], supports{1}=[1,3]; If no support is specified, then each is taken to be [0:numel(pdfs{k}-1];

Outputs: gammas: the discretization thresholds, as described in the paper. When sampling. The k-th dimension of the output random variable is f if e.g. supports{k}(1)=f and gammas{k}(f) <= U(k) <= gammas{k}(f+1) Lambda: the covariance matrix of the latent Gaussian random variable U joints2D: An n by n cell array, where each entry contains the 2 dimensional joint distribution of a pair of dimensions of the DG.

Code from the paper: 'Generating spike-trains with specified correlations', Macke et al., submitted to Neural Computation

Adapted from http://www.kyb.mpg.de/bethgegroup/code/efficientsampling

Parameters

- **pmfs** (*Type*) – Description
- **bin_cov** (*Type*) – Description
- **supports** (*Type*) – Description
- **accuracy** (*int, optional*) – Description (default 1e-10)

Returns **Value** – Description

Return type Type

hdnet.sampling.**find_latent_gaussian**(*bin_means*, *bin_cov*, *accuracy=1e-10*)

Compute parameters for the hidden Gaussian random vector U generating the binary Bernulli vector X with mean m and covariances c according to X = 0 <=> U < -g X = 1 <=> U > -g

Adapted from www.kyb.mpg.de/bethgegroup/code/efficientsampling

Parameters

- **bin_means** (*Type*) – Description
- **bin_cov** (*Type*) – Description
- **accuracy** (*int, optional*) – Description (default 1e-10)

Returns **Value** – Description

Return type Type

hdnet.sampling.**integer_to_binary**(*state*, *N*)
Given state 0, ..., 2**N - 1, returns corresponding binary vector x

Parameters

- **state** (*Type*) – Description
- **N** (*Type*) – Description

Returns **Value** – Description

Return type Type

hdnet.sampling.**ltqnorm**(*p*)
Modified from the author's original perl code (original comments follow below) by dfield@yahoo-inc.com. May 3, 2004.

Lower tail quantile for standard normal distribution function.

This function returns an approximation of the inverse cumulative standard normal distribution function. I.e., given P, it returns an approximation to the X satisfying P = Pr{Z <= X} where Z is a random variable from the standard normal distribution.

The algorithm uses a minimax approximation by rational functions and the result has a relative error whose absolute value is less than 1.15e-9.

Author: Peter John Acklam Time-stamp: 2000-07-19 18:26:14 E-mail: pjacklam@online.no WWW URL: http://home.online.no/~pjacklam

Returns **Value** – Description

Return type Type

hdnet.sampling.**ltqnorm_nd**(*arr*)
Missing documentation

Returns **Value** – Description

Return type Type

hdnet.sampling.**poisson_marginals**(*means*, *accuracy=1e-10*)
Finds the probability mass functions (pmfs) and approximate supports of a set of Poisson random variables with means specified in input "means". The second argument, "acc", specifies the desired degree of accuracy. The "support" is taken to consist of all values for which the pmfs is greater than acc.

Inputs: means: the means of the Poisson RVs acc: desired accuracy

Outputs: pmfs: a cell-array of vectors, where the k-th element is the probability mass function of the k-th Poisson random variable. supports: a cell-array of vectors, where the k-th element is a vector of integers of the states that the k-th Poisson random variable would take with probability larger than "acc". E.g., P(kth RV==supports{k}(1))=pmfs{k}(1);

Code from the paper: 'Generating spike-trains with specified correlations', Macke et al., submitted to Neural Computation

Adapted from http://www.kyb.mpg.de/bethgegroup/code/efficientsampling

Parameters

- **means** (*Type*) – Description
- **accuracy** (*int, optional*) – Description (default 1e-10)

Returns **Value** – Description

Return type Type

hdnet.sampling.**sample_dg_any_marginal**(*gauss_means*, *gauss_cov*, *num_samples*, *supports=None*)

[samples,hists]=SampleDGAnyMarginal(gammas,Lambda,supports,Nsamples) Generate samples for a Multivariate Discretized Gaussian with parameters gammas" and "Lambda" and "supports". The number of samples generated is "Nsamples"

input and output arguments are as described in "DGAnyMarginal"

Usage: Code from the paper: 'Generating spike-trains with specified correlations', Macke et al., submitted to Neural Computation

Adapted from http://www.kyb.mpg.de/bethgegroup/code/efficientsampling

Parameters

- **gauss_means** (*Type*) – Description
- **gauss_cov** (*Type*) – Description
- **num_samples** (*Type*) – Description
- **supports** (*Type, optional*) – Description (default None)

Returns **Value** – Description

Return type Type

hdnet.sampling.**sample_from_bernoulli**(*p*, *M=1*)

Returns N x M numpy array with M Bernoulli(p) N-bit samples

Parameters

- **p** (*Type*) – Description
- **M** (*int, optional*) – Description (default 1)

Returns **Value** – Description

Return type Type

hdnet.sampling.**sample_from_dichotomized_gaussian**(*bin_means*, *bin_cov*, *num_samples*, *gauss_means=None*, *gauss_cov=None*, *accuracy=1e-10*)

Missing documentation

Parameters

- **bin_means** (*Type*) – Description
- **bin_cov** (*Type*) – Description
- **num_samples** (*Type*) – Description
- **gauss_means** (*Type, optional*) – Description (default None)
- **gauss_cov** (*Type, optional*) – Description (default None)
- **accuracy** (*int, optional*) – Description (default 1e-10)

Returns **Value** – Description

Return type Type

hdnet.sampling.**sample_from_ising**(*J*, *theta*, *num_samples=2*)

Given an Ising model *J*, *theta* on N neurons produces *num_samples* samples Returns: a (N x num_samples) binary matrix with each column a binary vector (Ising sample)

Parameters

- **J** (*Type*) – Description
- **theta** (*Type*) – Description
- **num_samples** (*int, optional*) – Description (default 2)

Returns **Value** – Description

Return type Type

hdnet.sampling.**sample_from_prob_vector**(*p*, *num_samples=1*)

Given numpy probability vector p on N states produce num_samples samples returns: a (num_samples) integer vector with state labeled 0, ..., N-1

Parameters

- **p** (*Type*) – Description
- **num_samples** (*int, optional*) – Description (default 1)

Returns **Value** – Description

Return type Type

7.7 hdnet.spikes

Spikes class handling multi-neuron , multi-trial spike trains.

class hdnet.spikes.**Spikes**(*spikes=None*, *bin_size=None*, *preprocess=True*)

Bases: *hdnet.util.Restoreable*, object

Class for handling binary time-series datasets.

Creates a *Spikes* class from a binary 2d or 3d numpy array spikes.

If the array is 2-dimensional, the first dimension is assumed to represent neurons and the second one to represent bins.

If the array is 3-dimensional, the first dimension is assumed to represent trials, the second one to represent neurons and the third one to represent bins.

Parameters

- **spikes** (*numpy array*) – Raw binned spikes
- **bin_size** (*float, optional*) – Bin size in seconds (default None)
- **preprocess** (*bool, optional*) – If True makes data binary (Heaviside), if False leaves data untouched (default True)

Returns **spikes**

Return type instance of *Spikes* class

M

Returns number of bins represented by this class, shortcut for *num_bins()*.

Returns **number of bins**

Return type int

N

Returns number of neurons represented by this class, shortcut for *num_neurons()*.

Returns **number of neurons**

Return type int

T
Returns number of trials represented by this class, shortcut for *num_trials()*.

Returns number of trials

Return type int

bin_size
Returns the bin size in seconds of the data set.

Returns bin_size

Return type float

covariance(*trials=None*, *start=0*, *stop=None*, *save_png_name=None*)
return *new* numpy array of size (T x N x N) which is covariance matrix betwn neurons trials: e.g. [0, 1, 5, 6], None is all save_png_name: if not None then only saves

Parameters

- **trials** (*Type, optional*) – Description (default None)
- **start** (*int, optional*) – Description (default 0)
- **stop** (*Type, optional*) – Description (default None)
- **save_png_name** (*Type, optional*) – Description (default None)

Returns **Value** – Description

Return type Type

classmethod load(*file_name='spikes'*, *load_extra=False*)
Loads contents from file.

Parameters

- **file_name** (*str, optional*) – File name to load from (default 'spikes')
- **load_extra** (*bool, optional*) – Flag whether to load extra file contents, if any (default False)

Returns **spikes** – Instance of *Spikes* if loaded, *None* upon error

Return type *Spikes*

mean_activity()
Computes the mean activities of all cells (measured in the mean number of spikes per bin) in the data set. Multiply with *1./bin_size* (*bin_size* in seconds) to obtain firing rates in Hz. You can obtain a sorted list of neuron indices by activities by calling *mean_activity.argsort()* on the returned numpy array.

Returns **mean_activity** – Mean activities of all cells (measured in mean number of spikes per bin)

Return type 1d numpy array

mean_activity_hz()
Computes the mean activities of all cells in Hz. Only works if *bin_size* has been set during creation or via *bin_size()*. You can obtain a sorted list of neuron indices by activities by calling *mean_activity.argsort()* on the returned numpy array.

Returns **mean_activity** – Mean activities of all cells in Hz

Return type 1d numpy array

num_bins
Returns number of bins represented by this class.

Returns number of bins

Return type int

num_neurons
Returns number of neurons represented by this class.

Returns number of neurons

Return type int

num_trials
Returns number of trials represented by this class.

Returns number of trials

Return type int

rasterize(*trials=None*, *start=0*, *stop=None*, *save_png_name=None*)
Returns *new* (copied) numpy array of size (TN x M) trials: e.g. [1, 5, 6], None is all save_png_name: if not None then only saves

Parameters

- **trials** (*Type, optional*) – Description (default None)
- **start** (*int, optional*) – Description (default 0)
- **stop** (*Type, optional*) – Description (default None)
- **save_png_name** (*Type, optional*) – Description (default None)

Returns **Value** – Description

Return type Type

restrict_to_indices(*indices*, *copy=False*)
Restricts the spike data to the neurons with the given *indices* in the data set. To get the indices of the neurons in the original data set, call the method *restricted_neurons_indices()*. Note: in the default setting this function does not make a copy but drops the unselected neurons from the data set.

Parameters

- **indices** (*1d list or numpy array*) – list of (0-based) indices of neurons to select
- **copy** (*bool, optional*) – if True returns a new Spikes class with selected neurons, if False the changes are made in place, dropping all but the selected neurons (default False)

Returns **spikes** – an instance of *Spikes* class

Return type *Spikes*

restrict_to_most_active_neurons(*top_neurons=None*, *copy=False*)
Restricts the spike data to the number of *top_neurons* most active neurons in the data set. The neurons are sorted by activity in increasing order. To get the indices of the most active neurons in the original data set, call the method *restricted_neurons_indices()*. Note: in the default setting this function does not make a copy but drops the less active neurons from the data set.

Parameters

- **top_neurons** (*int, optional*) – number of most active neurons to choose, if None all are chosen (default None)

- **copy** (*bool, optional*) – if True returns a new Spikes class with selected neurons, if False the changes are made in place, dropping all but the selected neurons (default False)

Returns **spikes** – an instance of *Spikes* class

Return type *Spikes*

restricted_neurons_indices
Returns a list of the current neuron indices in the original data set after restriction. This will return *None* unless the data set has been restricted with `restrict_to()` or *restrict_to_most_active_neurons()*.

Returns **indices**

Return type list of int

save (*file_name='spikes'*, *extra=None*)
Saves contents to file.

Parameters

- **file_name** (*str, optional*) – File name to save to (default 'spikes')
- **extra** (*dict, optional*) – Extra information to save to file (default None)

Returns

Return type Nothing

spikes
Returns underlying numpy array representing spikes, with dimensions organized as follows (trials, neurons, bins)

Returns **spikes**

Return type 3d numpy array

to_windowed (*window_size=1*, *trials=None*, *reshape=False*)
Computes windowed version of spike trains, using a sliding window.

Returns new Spikes object of 3d numpy arr of windowed spike trains: T (num trials) x (window_size * N) x (M - window_size + 1) binary vector out of a spike time series reshape: returns T(M - window_size + 1) x (ws * N) numpy binary vector

Parameters

- **window_size** (*int, optional*) – Description (default 1)
- **trials** (*Type, optional*) – Description (default None)
- **reshape** (*bool, optional*) – Description (default False)

Returns **Value** – Description

Return type Type

trials_average (*trials=None*)
Computes the average activity over all trials in the data set.

Parameters **trials** (*1d list or numpy array*) – list of (0-based) indices of trials to include, if *None* all trials are used (default None)

Returns **trials_average** – mean activity over trials

Return type 2d numpy array

7.8 hdnet.spikes_model

Null-models for spikes' statistics.

class hdnet.spikes_model.**BernoulliHomogeneous**(*spikes=None*, *stimulus=None*, *window_size=1*, *learner=None*)

Bases: *hdnet.spikes_model.SpikeModel*

Bernoulli model of spikes, homogeneous.

sample_from_model(*trials=None*, *reshape=False*)

Returns Spikes object of 3d numpy arr of windowed iid Bernouli spike trains: (with probabilities = spike rates of each neuron in self at trial t) X: T (num trials) x (window_size * N) x (M - window_size + 1) binary vector out of a spike time series reshape: returns T(M - window_size + 1) x (ws * N) numpy binary vector

Parameters

- **trials** (*Type, optional*) – Description (default None)
- **reshape** (*bool, optional*) – Description (default False)

Returns **Value** – Description

Return type Type

class hdnet.spikes_model.**BernoulliInhomogeneous**(*spikes=None*, *stimulus=None*, *window_size=1*, *learner=None*)

Bases: *hdnet.spikes_model.SpikeModel*

Bernoulli model of spikes, inhomogeneous (i.e. varying rate over time).

sample_from_model(*averaging_window_size=20*, *trials=None*, *reshape=False*)

Missing documentation

Parameters

- **averaging_window_size** (*int, optional*) – Description (default 20)
- **trials** (*Type, optional*) – Description (default None)
- **reshape** (*bool, optional*) – Description (default False)

Returns **Value** – Description

Return type Type

class hdnet.spikes_model.**DichotomizedGaussian**(*spikes=None*, *stimulus=None*, *window_size=1*, *learner=None*)

Bases: *hdnet.spikes_model.SpikeModel*

Class modeling the dichotomized Gaussian model of spikes.

sample_from_model(*trials=None*, *reshape=False*)

Missing documentation

Parameters

- **trials** (*Type, optional*) – Description (default None)
- **reshape** (*bool, optional*) – Description (default False)

Returns **Value** – Description

Return type Type

class hdnet.spikes_model.**DichotomizedGaussianPoisson**(*spikes=None*, *stimulus=None*, *window_size=1*, *learner=None*)

Bases: *hdnet.spikes_model.SpikeModel*

Class modeling the dichotomized Gaussian model of spikes, with Poisson marginals.

sample_from_model(*trials=None*, *reshape=False*)

Missing documentation

Parameters

- **trials** (*Type, optional*) – Description (default None)
- **reshape** (*bool, optional*) – Description (default False)

Returns **Value** – Description

Return type Type

class hdnet.spikes_model.**Ising**(*spikes=None*, *stimulus=None*, *window_size=1*, *learner=None*)

Bases: *hdnet.spikes_model.SpikeModel*

Class modeling the Ising / Hopfield model of spikes

sample_from_model(*J=None*, *theta=None*, *trials=None*, *reshape=False*)

Returns new spikes object with iid Ising spike trains: (with Ising model determined by learning with MPF)

Parameters

- **J** (*Type, optional*) – Description (default None)
- **theta** (*Type, optional*) – Description (default None)
- **trials** (*Type, optional*) – Description (default None)
- **reshape** (*bool, optional*) – Description (default False)

Returns **Value** – Description

Return type Type

class hdnet.spikes_model.**Shuffled**(*spikes=None*, *stimulus=None*, *window_size=1*, *learner=None*)

Bases: *hdnet.spikes_model.SpikeModel*

Shuffled spikes

sample_from_model(*trials=None*, *trial_independence=True*, *reshape=False*)

returns new Spikes object: permutes spikes in time trial_independence: diff permutation for each trial

Parameters

- **trials** (*Type, optional*) – Description (default None)
- **trial_independence** (*bool, optional*) – Description (default True)
- **reshape** (*bool, optional*) – Description (default False)

Returns **Value** – Description

Return type Type

class hdnet.spikes_model.**SpikeModel**(*spikes=None*, *stimulus=None*, *window_size=1*, *learner=None*)

Bases: *hdnet.util.Restoreable*, object

Generic model of spikes (and stimulus).

Parameters

- **spikes** (*spikes to model*) – Description (default None)
- **stimulus** (*corresp stimulus if existent*) – Description (default None)
- **window_size** (*length of time window in binary bins*) – Description (default 1)
- **learner** (*Type, optional*) – Description (default None)
- **Parameters** –
- **spikes** –
- **stimulus** –
- **window_size** –

Returns **Value** – Description

Return type Type

chomp()
Missing documentation

Returns **Value** – Description

Return type Type

distinct_patterns_over_windows(*window_sizes=None*, *trials=None*, *save_couplings=False*, *remove_zeros=False*)
Returns tuple: counts, entropies [, couplings] counts, entropies: arrays of size 2 x T x WSizes (0: empirical from model sample, 1: dynamics from learned model on sample)

Parameters

- **window_sizes** (*Type, optional*) – Description (default None)
- **trials** (*Type, optional*) – Description (default None)
- **save_couplings** (*bool, optional*) – Description (default False)
- **remove_zeros** (*bool, optional*) – Description (default False)

Returns **Value** – Description

Return type Type

fit(*trials=None*, *remove_zeros=False*, *reshape=False*)
Missing documentation

Parameters

- **trials** (*Type, optional*) – Description (default None)
- **remove_zeros** (*bool, optional*) – Description (default False)
- **reshape** (*bool, optional*) – Description (default False)

Returns **Value** – Description

Return type Type

hopfield_patterns
Missing documentation

Returns **Value** – Description

Return type Type

hopfield_spikes
Missing documentation

Returns **Value** – Description

Return type Type

learn_time
Missing documentation

Returns **Value** – Description

Return type Type

learner
Missing documentation

Returns **Value** – Description

Return type Type

classmethod load(*folder_name='spikes_model'*, *load_extra=False*)
Missing documentation

Parameters

- **folder_name** (*str, optional*) – Description (default 'spikes_model')
- **load_extra** (*bool, optional*) – Description (default False)

Returns **Value** – Description

Return type Type

original_spikes
Missing documentation

Returns **Value** – Description

Return type Type

raw_patterns
Missing documentation

Returns **Value** – Description

Return type Type

sample_from_model(*trials=None*, *reshape=False*)
Missing documentation

Parameters

- **trials** (*Type, optional*) – Description (default None)
- **reshape** (*bool, optional*) – Description (default False)

Returns **Value** – Description

Return type Type

sample_spikes
Missing documentation

Returns **Value** – Description

Return type Type

save(*folder_name='spikes_model'*)
saves as npz's: network, params, spikes file_name

Parameters **folder_name** (*str, optional*) – Description (default 'spikes_model')

Returns **Value** – Description

Return type Type

stimulus
Missing documentation

Returns **Value** – Description

Return type Type

window_size
Missing documentation

Returns **Value** – Description

Return type Type

7.9 hdnet.stats

Statistics module. Contains functions for analyzing sequences of memories and miscellaneous statistics functions.

class hdnet.stats.**SequenceAnalyzer**(*counter*)
Bases: object

Analyzes various aspects of sequences of memory labels, such as label probabilities, frequent sub-sequences and Markov transition probabilities.

Parameters **counter** (instance of *Counter*) – Base counter the sequence to operate on

Returns **analyzer** – Instance of *SequenceAnalyzer*

Return type instance of *SequenceAnalyzer*

calculate_cycles_entropy_scores(*node*, *min_len=2*, *max_len=20*, *weighting=None*, *weighting_element=None*, *node_entropies=None*, *graph=None*)
Calculate entropy scores of cycles (simple closed paths) in *graph* starting and terminating in given node *node*. An entropy score of a path is a weighted sum of the entropies of each node contained in the path.

Parameters

- **node** (*str*) – Label of base node
- **min_len** (*int, optional*) – Minimal length of cycle to consider (default 2)
- **max_len** (*int, optional*) – Maximal length of cycle to consider (default 20)
- **weighting** (*Type, optional*) – Weighting function, if None *lambda x: 1./len(x)* is taken (default None)
- **weighting_element** (*Type, optional*) – Weighting function per element, if None *lambda x, p: x* is taken (default None)
- **node_entropies** (*1d numpy array, optional*) – Node entropies to use, if None entropies from Markov transition probabilities of internal sequence are used (default None)
- **graph** (networkx.DiGraph, optional) – Graph to operate on, if None Markov graph belonging to internal sequence is used (default None)

Returns **(cycles, scores)** – Scored cycles, where cycles is a 1d array of cycles and scores a 1d array of cycle scores. Index in scores identical to index in cycles, arrays sorted by score (ascending).

Return type (1d numpy array, 1d numpy array)

calculate_paths_entropy_scores(*node1*, *node2*, *min_len=2*, *max_len=20*, *weighting=None*, *weighting_element=None*, *node_entropies=None*, *graph=None*)

Calculate entropy scores of all simple paths in *graph* from *node1* to *node2*. An entropy score of a path is a weighted sum of the entropies of each node contained in the path.

Parameters

- **node** (*str*) – Label of base node
- **min_len** (*int, optional*) – Minimal length of cycle to consider (default 2)
- **max_len** (*int, optional*) – Maximal length of cycle to consider (default 20)
- **weighting** (*Type, optional*) – Weighting function, if None *lambda x: 1./len(x)* is taken (default None)
- **weighting_element** (*Type, optional*) – Weighting function per element, if None *lambda x, p: x* is taken (default None)
- **node_entropies** (*1d numpy array, optional*) – Node entropies to use, if None entropies from Markov transition probabilities of internal sequence are used (default None)
- **graph** (`networkx.DiGraph`, optional) – Graph to operate on, if None Markov graph belonging to internal sequence is used (default None)

Returns **(paths, scores)** – Scored paths, where paths is a 1d array of paths and scores a 1d array of cycle scores. Index in scores identical to index in paths, arrays sorted by score (ascending).

Return type (1d numpy array, 1d numpy array)

compute_label_markov_entropies(*markov_probabilities=None*, *eps=1e-09*)

Computes the entropy of each label using its Markov transition probabilities in the space of all labels.

Parameters

- **markov_probabilities** (*2d numpy array, float, optional*) – Markov transtion matrix to use, if None *label_markov_probabilities()* is used (default None)
- **eps** (*int, optional*) – Threshold value below which a float is assumed to be 0 (default 1e-9)

Returns **markov_entropies** – Vector of entropies calculated over Markov transition probabilities

Return type 1d numpy array, float

compute_label_markov_probabilities(*sequence=None*)

Computes matrix of Markov transition probabilities of all labels occuring in *sequence* over this sequence.

Parameters **sequence** (*1d numpy array, int, optional*) – Sequence of symbols to consider, if None *sequence()* is used (default None)

Returns **markov_prob** – Matrix of markov transition probabilities of entries in *sequence*

Return type 2d numpy array

compute_label_occurrences(*sequence=None*)

Compute occurrences of labels in sequence

Parameters **sequence** (*1d numpy array, int, optional*) – Sequence of symbols to consider, if None *sequence()* is used (default None)

Returns **occurrences** – Number of occurrences for all labels

Return type dict, label => number of occurrences

compute_label_probabilities(*sequence=None*, *parent=None*)
Compute probability vector of patterns as empirical probabilities. If parent counter object present then return probability vector in that space.

Parameters

- **sequence** (*1d numpy array, int, optional*) – Sequence of symbols to consider, if None *sequence()* is used (default None)
- **parent** (*Counter*, optional) – Parent Counter object (default None)

Returns **probabilities** – Vector of label probabilities

Return type numpy array

compute_markov_graph(*markov_probabilities=None*, *node_labels=None*, *thres=0*, *no_cycle=False*)
Computes the directed state transition graph over all labels using the package NetworkX. Each directed edge (x, y) is assigned as weight the Markov transition probability from label x to label y. markov_probabilities: 2d numpy array node_labels: remapped labels (optional, default None) thres: weight threshold for edges; only edges above that weight are included in the graph (default 0) no_cycle: boolean flag specifiying handling of self-cycles; if set to True, self-cycles are discarded. NetworkX DiGraph with lables as nodes and Markov transition probabilities as edges

---

Note: This function needs the *networkx* package.

Parameters

- **markov_probabilities** (*2d numpy array, optional*) – Markov transition probabilities of labels, if None Markov transition probabilities of internal sequence are used (default None)
- **node_labels** (*list or 1d numpy array, optional*) – Node labels to use, if None labels of internal sequence are used (default None)
- **thres** (*int, optional*) – Threshold to exclude nodes occurring less than a given number of times in the sequence (default 0)
- **no_cycle** (*bool, optional*) – Flag whether to not include self-cycles at nodes (default False)

Returns **markov_graph** – Directed graph with edge weights corresponding to Markov transition probabilities.

Return type networkx.DiGraph instance

counter
Returns the *Counter* object this class operates.

Returns **counter**

Return type instance of *Counter*

entropy()
Computes entropy over probability distribution of sequence of pattern labels.

Returns **entropy** – Entropy of probability distribution

Return type float

filter_sequence_repeating_labels(*repetitions=2*, *sequence=None*)
Removes all consecutive repetitions of labels from sequence occurring more than *repetitions* times (default: 2).

Parameters

- **sequence** (*1d numpy array, int, optional*) – Sequence of symbols to consider, if None `sequence()` is used (default None)
- **repetitions** (*int, optional*) – Description (default 2)

Returns **filtered_sequence** – Filtered sequence

Return type 1d numpy array

filter_sequence_threshold(*threshold*, *replacement_label=-1*, *sequence=None*)
Filter out all labels from sequence, occurring less than *threshold* times and replace them with *replacement_label*.

Parameters

- **threshold** (*int*) – Minimal number of occurrences to not filter out label
- **replacement_label** (*int, optional*) – Replacement label used for a label that is dropped (default -1)
- **sequence** (*1d numpy array, int, optional*) – Sequence of symbols to consider, if None `sequence()` is used (default None)

Returns **filtered_sequence** – Filtered sequence

Return type 1d numpy array

filter_sequence_top_occurring(*count*, *replacement_label=-1*, *sequence=None*)
Filter out all labels from sequence, occuring less than *threshold* times and replace them with *replacement_label*.

Parameters

- **count** (*int*) – Number of top occurring labels to keep
- **replacement_label** (*int, optional*) – Replacement label used for a label that is dropped (default -1)
- **sequence** (*1d numpy array, int, optional*) – Sequence of symbols to consider, if None `sequence()` is used (default None)

Returns **filtered_sequence** – Filtered sequence

Return type 1d numpy array

find_subsequences(*thresholds*, *sequence=None*)
Enumerates all subsequences of length *len(thresholds)* in *sequence* (if sequence is *None* the possibly filtered sequence from the stored counter object is taken). Subsequences of length i are only considered if they appear at least *thresholds[i - 1]* times in the sequence.

Parameters

- **thresholds** (*list, int*) – List of threshold values
- **sequence** (*list or numpy array, optional*) – Sequence to consider, if *None* defaults to stored sequence (default *None*)

Returns **sequences** – List of dictionaries containing all found sequences as keys and counts as values. Keys are memory labels separated by ‘,’.

Return type list of dicts

find_subsequences_positions (*subsequence*, *sequence=None*)
Enumerates all positions of given *subsequence* in *sequence* (if sequence is *None* the possibly filtered sequence from the stored counter object is taken).

Parameters

- **subsequence** (*list or numpy array*) – Subsequence to search for
- **sequence** (*list or numpy array, optional*) – Sequence to consider, if *None* defaults to stored sequence (default *None*)

Returns **positions** – List of positions in *sequence* where *subsequence* occurs

Return type 1d numpy array

label_markov_entropies
Returns entropies of Markov transition probabilities of labels in sequence, see *compute_label_markov_entropies()*.

Returns **markov_ent** – Vector of entropies computed from Markov transition probabilities

Return type 1d numpy array, float

label_markov_probabilities
Returns Markov transition probabilities of labels in sequence, see *compute_label_markov_probabilities()*.

Returns **markov_prob** – Matrix of Markov transition probabilities

Return type 2d numpy array, float

label_probabilities
Returns probabilities of labels in sequence, see *compute_label_probabilities()*.

Returns **prob** – Vector of label probabilities

Return type 1d numpy array, float

markov_graph
Returns Markov graph belowing to sequence of labels in sequence, see *compute_markov_graph()*.

Returns **markov_graph** – Directed graph with edge weights corresponding to Markov transition probabilities.

Return type networkx.DiGraph instance

reduce_graph_bridge (*graph=None*)
Removes all "bridge" nodes v from *graph*, where a bridge node is defined as one having only one incoming and only one outgoing edge, i.e. u -> v -> w.

Parameters **graph** (`networkx.DiGraph`, optional) – Graph to operate on, if None Markov graph belonging to internal sequence is used (default None)

Returns **n_removed** – Number of removed nodes

Return type int

reduce_graph_brute (*filtered_nodes*, *graph=None*)
Removes self cycles u -> u of all nodes u in *graph*.

Parameters

- **filtered_nodes** (*Type*) – Description

- **graph** (networkx.DiGraph, optional) – Graph to operate on, if None Markov graph belonging to internal sequence is used (default None)

Returns **n_removed** – Number of removed nodes

Return type int

reduce_graph_cycle(*graph=None*)

Removes all "cycle" nodes v from *graph*, where a cycle node is defined as one having only one incoming and only one outgoing edge, to the same node, i.e. u -> v -> u.

Parameters **graph** (networkx.DiGraph, optional) – Graph to operate on, if None Markov graph belonging to internal sequence is used (default None)

Returns **n_removed** – Number of removed nodes

Return type int

reduce_graph_ncycle(*node*, *n*, *graph=None*)

Removes all edges from *graph*, that belong to simple closed paths (i.e. cycles) around the given *node* that do not have at least length *n*.

Parameters

- **graph** (networkx.DiGraph, optional) – Graph to operate on, if None Markov graph belonging to internal sequence is used (default None)
- **node** (*str*) – Label of starting node
- **n** (*Type*) – Minimal path length
- **graph** – Graph to operate on, if None Markov graph belonging to internal sequence is used (default None)

Returns **n_removed** – Number of removed nodes

Return type int

reduce_graph_out_degree(*thres_max*, *thres_min=1*, *graph=None*)

Removes all nodes from *graph* that have an out-degree of more than *thres_max* or an out-degree of less than *thres_min*.

Parameters

- **thres_max** (*int*) – Maximal out degree to retain
- **thres_min** (*int, optional*) – Minimal out degree to retain (default 1)
- **graph** (networkx.DiGraph, optional) – Graph to operate on, if None Markov graph belonging to internal sequence is used (default None)

Returns **n_removed** – Number of removed nodes

Return type int

reduce_graph_self_cycles(*graph=None*)

Removes self cycles u -> u of all nodes u in *graph*.

Parameters **graph** (networkx.DiGraph, optional) – Graph to operate on, if None Markov graph belonging to internal sequence is used (default None)

Returns **n_removed** – Number of removed nodes

Return type int

reduce_graph_stub (*graph=None*)
Removes all "stub" nodes v from *graph*, where a stub node is defined as one having only only one incoming and no outgoing edges, i.e. u -> v.

Parameters **graph** (networkx.DiGraph, optional) – Graph to operate on, if None Markov graph belonging to internal sequence is used (default None)

Returns **n_removed** – Number of removed nodes

Return type int

reduce_graph_triangles (*graph=None*)
Removes all triangles from *graph* (by deleting edges with the lowest weight) of the form u -> v, u -> w, v -> w. Deletes the edge with the lowest weight to delete the triangle (u, v, w).

Parameters **graph** (networkx.DiGraph, optional) – Graph to operate on, if None Markov graph belonging to internal sequence is used (default None)

Returns **n_removed** – Number of removed nodes

Return type int

sequence
Returns the (possibly filtered) sequence this class currently operates on.

Returns **sequence** – Sequence of integer labels

Return type 1d numpy array, int

static subseqs (*sequence*, *length*)
Enumerates all subsequences of given *length* in *sequence*. Lazy, returns generator object.

Parameters

- **sequence** (*list or numpy array*) – Sequence to enumerate subsequences for
- **length** (*int*) – Length of subsequences to enumerate

Returns **generator** – Lazy generator object for subsequences

Return type generator object

7.10 hdnet.stimulus

Stimuli time-series.

class hdnet.stimulus.**Stimulus** (*stimulus_arr=None*, *npz_file=None*, *h5_file=None*, *preprocess=True*)
Bases: *hdnet.util.Restoreable*, object

class handling time-series stimuli

Parameters

- **stimulus_arr** – file_name of data, a numpy M x X array M = number of stimulus (eg. movie) frames X = stimulus array (a movie, etc)
- **preprocess** – override for other operations on raw data

M
Missing documentation

Returns **Value** – Description

Return type Type

X
Missing documentation

Returns **Value** – Description

Return type Type

classmethod load(*file_name='stimulus'*, *load_extra=False*)
Loads contents from file.

Parameters

- **file_name** (*str, optional*) – File name to load from (default 'stimulus')
- **load_extra** (*bool, optional*) – Flag whether to load extra file contents, if any (default False)

Returns **spikes** – Instance of *Stimulus* if loaded, *None* upon error

Return type *Stimulus*

preprocess()
Missing documentation

Returns **Value** – Description

Return type Type

save(*file_name='stimulus'*, *extra=None*)
Saves contents to file.

Parameters

- **file_name** (*str, optional*) – File name to save to (default 'stimulus')
- **extra** (*dict, optional*) – Extra information to save to file (default None)

Returns

Return type Nothing

snapshot(*start=0*, *stop=None*, *save_png_name=None*)
Returns a matrix or saves a PNG of avg of data between start and stop times save_png_name: if not None then only saves picture

Parameters

- **start** (*int, optional*) – Description (default 0)
- **stop** (*Type, optional*) – Description (default None)
- **save_png_name** (*Type, optional*) – Description (default None)

Returns **Value** – Description

Return type Type

stimulus_arr
Missing documentation

Returns **Value** – Description

Return type Type

7.11 hdnet.util

Utility functions for hdnet

class hdnet.util.**Restoreable**
Bases: object

Mixin class for supporting of saving and loading of contents in compressed numpy format (numpy.savez). Supports file versioning and type identification.

7.12 hdnet.visualization

Visualization functions for hdnet.

hdnet.visualization.**combine_windows**(*windows*)
Missing documentation

Returns **Value** – Description

Return type Type

hdnet.visualization.**pattern_rank_plot**(*empirical*, *patterns*, *color_empirical='g'*, *color_pattern='r'*, *mark_empirical=None*, *mark_converged=None*, *label_empirical='raw'*, *label_patterns='Hopfield'*, *plot_mtas=True*)
Missing documentation

Parameters

- **empirical** (*Type*) – Description
- **patterns** (*Type*) – Description
- **color_empirical** (*str, optional*) – Description (default 'g')
- **color_pattern** (*str, optional*) – Description (default 'r')
- **mark_empirical** (*Type, optional*) – Description (default None)
- **mark_converged** (*Type, optional*) – Description (default None)
- **plot_mtas** (*bool, optional*) – Description (default True)

Returns **Value** – Description

Return type Type

hdnet.visualization.**plot_all_matrices**(*matrices*, *file_names*, *cmap='gray'*, *colorbar=True*, *vmin=None*, *vmax=None*)
Missing documentation

Parameters

- **matrices** (*Type*) – Description
- **file_names** (*Type*) – Description
- **cmap** (*str, optional*) – Description (default 'gray')
- **colorbar** (*bool, optional*) – Description (default True)
- **vmin** (*Type, optional*) – Description (default None)
- **vmax** (*Type, optional*) – Description (default None)

Returns **Value** – Description

Return type Type

hdnet.visualization.**plot_graph**(*g*, *nodeval=None*, *cmap_nodes='cool'*, *cmap_edges='autumn'*, *node_vmin=None*, *node_vmax=None*, *edge_vmin=None*, *edge_vmax=None*, *draw_edge_weights=True*, *edge_weight_format='%.3f'*)

Missing documentation

Parameters

- **g** (*Type*) – Description
- **nodeval** (*Type, optional*) – Description (default None)
- **cmap1** (*str, optional*) – Description (default 'Blues_r')
- **cmap2** (*str, optional*) – Description (default 'bone_r')
- **node_vmin** (*Type, optional*) – Description (default None)
- **node_vmax** (*Type, optional*) – Description (default None)
- **edge_vmin** (*Type, optional*) – Description (default None)
- **edge_vmax** (*Type, optional*) – Description (default None)

Returns **Value** – Description

Return type Type

hdnet.visualization.**plot_matrix_whole_canvas**(*matrix*, ***kwargs*)

Missing documentation

Parameters

- **matrix** (*Type*) – Description
- **kwargs** (*Type*) – Description

Returns **Value** – Description

Return type Type

hdnet.visualization.**plot_memories_distribution_matrix**(*patterns*, *trials*, *t_min=None*, *t_max=None*, *p_min=None*, *p_max=None*, *v_min=None*, *v_max=None*, *cmap='Paired'*)

Missing documentation

Parameters

- **patterns** (*Type*) – Description
- **trials** (*Type*) – Description
- **t_min** (*Type, optional*) – Description (default None)
- **t_max** (*Type, optional*) – Description (default None)
- **p_min** (*Type, optional*) – Description (default None)
- **p_max** (*Type, optional*) – Description (default None)
- **v_min** (*Type, optional*) – Description (default None)
- **v_max** (*Type, optional*) – Description (default None)

- **cmap** (*str, optional*) – Description (default 'Paired')

Returns **Value** – Description

Return type Type

hdnet.visualization.**raster_plot_psth**(*spikes*, *trial=0*, *start_idx=0*, *stop_idx=None*, *bin_size=0.002*, *hist_bin_size=0.005*, *label_x='time [s]'*, *label_y_hist_x='[Hz]'*, *label_y_raster='neuron'*, *label_x_hist_y=None*, *fig_size=None*, *hist_x=True*, *hist_y=False*, *color_raster='#070d0d'*, *color_hist='#070d0d'*)

Missing documentation

Parameters

- **spikes** (*Type*) – Description
- **trial** (*int, optional*) – Description (default 0)
- **start_idx** (*int, optional*) – Description (default 0)
- **stop_idx** (*Type, optional*) – Description (default None)
- **bin_size** (*int, optional*) – Description (default 0.002)
- **hist_bin_size** (*int, optional*) – Description (default 0.005)
- **label_x** (*str, optional*) – Description (default 'time [s]')
- **label_y_hist_x** (*str, optional*) – Description (default '[Hz]')
- **label_y_raster** (*str, optional*) – Description (default 'neuron')
- **label_x_hist_y** (*Type, optional*) – Description (default None)
- **fig_size** (*Type, optional*) – Description (default None)
- **hist_x** (*bool, optional*) – Description (default True)
- **hist_y** (*bool, optional*) – Description (default False)
- **color_raster** (*str, optional*) – Description (default '#070d0d')
- **color_hist** (*str, optional*) – Description (default '#070d0d')

Returns **Value** – Description

Return type Type

hdnet.visualization.**save_matrix_whole_canvas**(*matrix*, *fname*, ***kwargs*)

Missing documentation

Parameters

- **matrix** (*Type*) – Description
- **fname** (*Type*) – Description
- **kwargs** (*Type*) – Description

Returns **Value** – Description

Return type Type

CHAPTER

EIGHT

REFERENCES

CHAPTER

NINE

INDICES AND TABLES

- *References*
- genindex
- modindex
- search

h

Symbols

A

B

C

D

E

F

N

O

P

R

S

T

U

W

X